coderush

a productivity extension

for Microsoft Visual Studio

by DevExpress

by Alex Skorkin (c) 2015

About Author

Alex Skorkin is working for DevExpress. His overall experience with Visual Studio productivity tools starts since 2004th. You can reach him at alex@skorkin.com and www.skorkin.com.

ADDICTED TO CodeRush™

by DevExpress

Microsoft CERTIFIED Professional Developer

Windows Developer
Enterprise Application Developer

Microsoft CERTIFIED Technology Specialist

.Net Framework 2.0, Windows Applications
.Net Framework 2.0, Web Applications
.Net Framework 2.0, Distributed Applications
.NET Framework 3.5, Windows Forms Applications
.NET Framework 3.5, Windows Presentation Foundation Applications

Contents at a Glance

Table of Contents

Introduction

DevExpress Visual Studio productivity tools

DevExpress engineers feature-complete visual components and **IDE Tools** productivity add-ins for Visual Studio. NET. IDE Tools are intended to boost your overall coding performance and make coding enjoyable and fun inside Visual Studio. These tools are capable of improving the development environment from the inside out – bringing you new ways to look at code, new ways to generate code, new ways to navigate through code, new ways to refactor your code and new ways to create your own extensions to your development environment.

IDE Tools contains of several products:

Free versions: DXCore, CodeRush Xpress. Pro versions: CodeRush Pro

DevExpress CodeRush Pro

CodeRush is a powerful Visual Studio® .NET add-in that enhances the developer experience by accelerating developer and team productivity via an integrated suite of technologies. **CodeRush** integrates with Visual Studio to automate common code creation tasks, and to simplify code investigation. It clearly shows a code structure, and creates typical code blocks with a minimum of effort. You can create new code blocks with only a few keystrokes, or with a few clicks. Features are organized into three categories: code creation, navigation, and visualization. Features can be chained together to dramatically increase developer speed and efficiency. CodeRush is built on the DXCore engine.

Key advantages and features of **CodeRush**:

- Easy code investigation
- Automation of common code creation tasks
- Easy search and navigation to a required code location
- Fast and easy selection
- On-the-fly code issues search
- Easy and effective mastering of CodeRush features

Write better code faster using CodeRush templates, consume-first declaration, and advanced selection & clipboard features. Improve code with powerful refactorings, background code issue analysis & fixes. Run tests faster. Rename faster. Find all references faster. No tool is more efficient than CodeRush.

DevExpress CodeRush Xpress

CodeRush Xpress is available to all Visual Studio 2009 and 2010 C# and VB developers for free and offers a comprehensive suite of tools that enable you and your team to simplify and shape complex code – making it easier to read and less costly to maintain. It includes code simplification, creation, navigation, visualization and refactoring fea tures. CodeRush Xpress is built based on the DXCore engine.

Key advantages and features of CodeRush Xpress:

- One-key access to refactorings, micro refactorings, and TDD-style code building features.
- Powerful navigation tools help get you to where you need to be efficiently.
- A sampling of editor, selection, and clipboard tools.

You may register your copy of **CodeRush Xpress** to automatically receive product updates with new functionality. The registration procedure takes only a couple of minutes and doesn't require personal details other than your name and email address.

IDE Tools chief architect – Mark Miller

Mark Miller is Chief Architect of the IDE Tools division at DevExpress. Mark is a high-speed developer with strong expertise in architecture, plug-in systems, component development, extreme programming, effective user interfaces, good class design and great UI. Mark is well known for both his sense of humor as well as his software development jam sessions. A recent review described one of Mark's sessions as "spectacular-insightful, fast paced, humorous, and impressive. Mark is one of the few people with the audacity to perform coding improvisations in front of a large audience."

Mark is a C# MVP. He has been developing software since the early 1980s. He spent the 12 years preceeding 2004 coding developer tools, which are his – figuring out how to make developers more efficient. He was awarded the "Spirit of Delphi" by Borland. He created CodeRush for Delphi in 1997, and it has won numerous reader's choice awards.

IDE Tools documentation and info resources

You may look at the following information resources to learn more about IDE Tools products:

- Online help for **IDE Productivity Tools** at http://help.devexpress.com.

- User Guide available via the "**DevExpress** | **User Guide**" menu after the IDE Tools are installed.

- Online Training Videos at http://tv.devexpress.com

- Mark Miller's (the chief architect of the **IDE Tools**) blog at http://community.devexpress.com/blogs/markmiller

- Rory Becker's blog at http://community.devexpress.com/blogs/rorybecker

IDE Tools dedicated community

DXCore Community PlugIns at https://code.google.com/p/dxcorecommunityplugins/

This site is all about Free DXCore plug-ins for IDE Tools and related products. These plug-ins represent additional pieces of functionality for Visual Studio .NET which are designed to improve your experience.

Plugins on this site require:

- A non-express version of either VS2005, VS2008 or VS2010

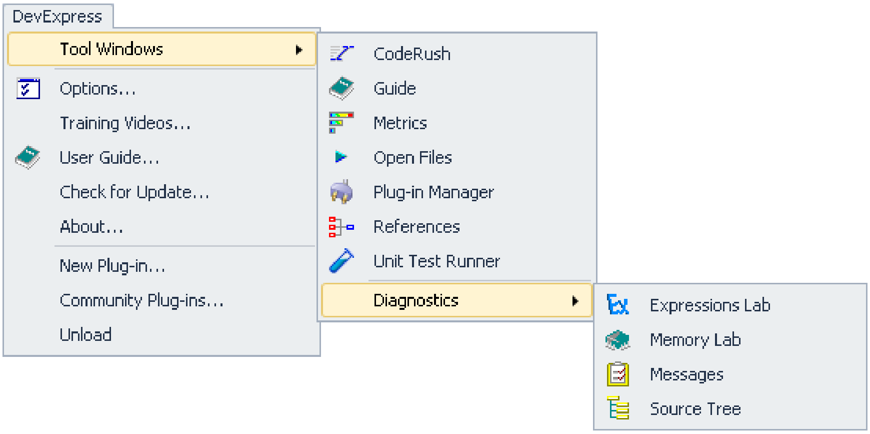
- A DXCore based product (DXCore, CodeRush, CodeRush Xpress or Refactor! Pro)

The required version of these products varies. It is generally recommended to use the latest version available.

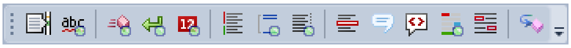
IDE environment changes after IDE Tools are installed

The first two things you'll notice in the IDE (once everything is loaded) include:

- The DevExpress menu. Trough the menu you can access add-in related tool windows, configure options, check for updates, unload tools, etc.

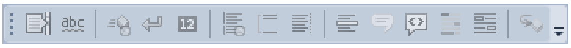


- The DXCore Visualizetoolbar**.** The toolbar, like all other standard Visual Studio toolbars, integrates into the IDE toolbars area when IDE tools are loaded. Here's what it looks like:



The green ball in the right down corner indicates whether a particular feature is enabled or not.

Visualize toolbar is enabled only when a Code Editor is active, otherwise it is disabled:

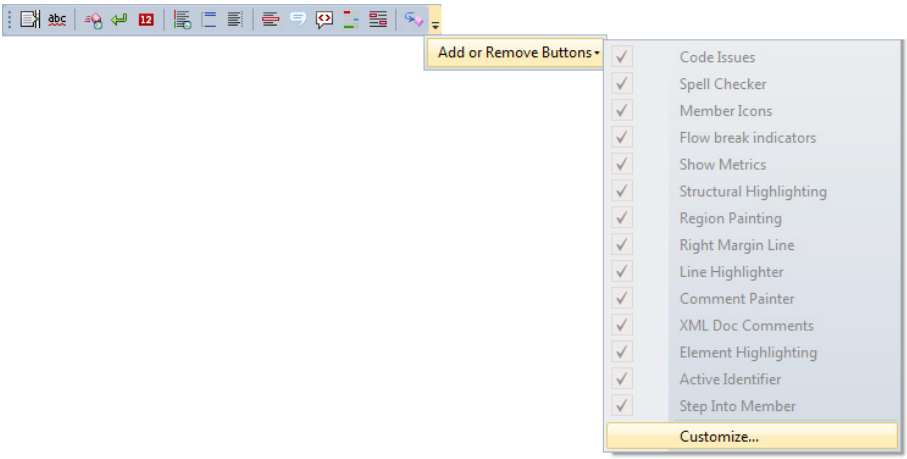


This toolbar contains a set of toggle buttons allowing you to easily turn on/off the following features CodeRush provides:

- Code Issues
- Spell Checker
- Member Icons
- Flow Breaks Evaluation
- Show Metrics
- Structural Highlighting
- Region Painting
- Right Margin line
- Line Highlighter
- XML Doc Comments
- Step Into Member

All of these features are visual, and related to the code editor except "**Step Into Member**" – which is used in Debug Mode.

Although the **Visualize toolbar** behaves as a common toolbar, you aren’t able to customize it using the IDE:

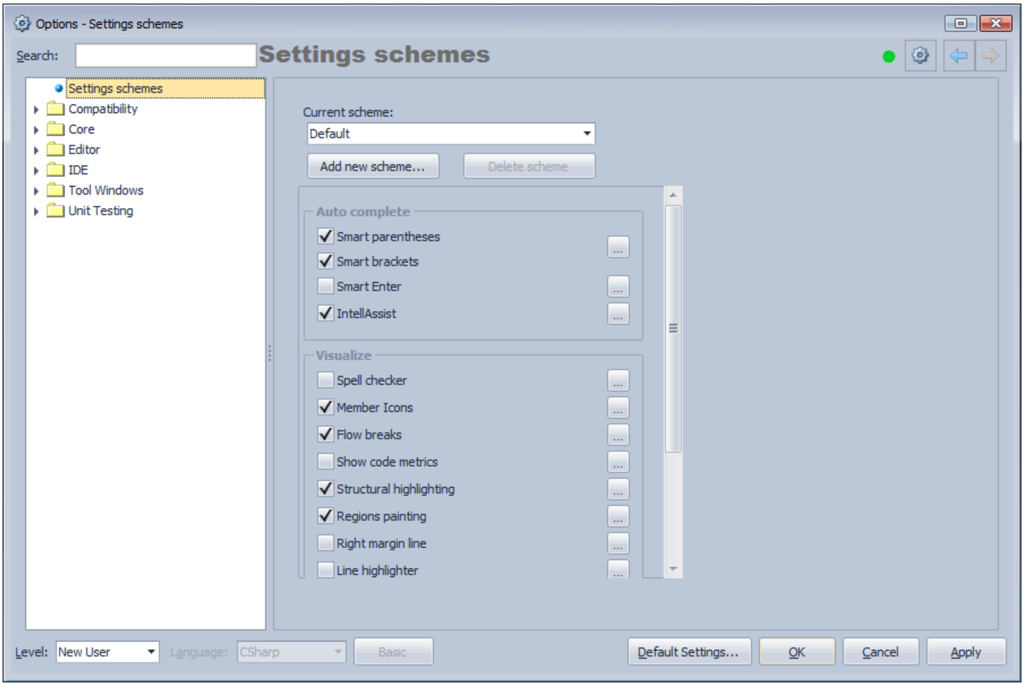


From the other side, this is not really necessary (e.g. remove buttons from the toolbar), but this can be achieved programmatically, however. For example, you can place a new button on the toolbar using the DXCore Action component while creating a new DXCore plug-in.

Settings schemes: FrictionFree and Default modes

CodeRush supports setting schemes. This allows you to have different settings schemes for laptop and desktop computers, for various keyboard layouts, for numerous environments and development tasks: code review, code navigation, code refactoring, etc. You can disable or enabled any features in a particular scheme.

The schemes are controlled on the corresponding option page of the Options Dialog:



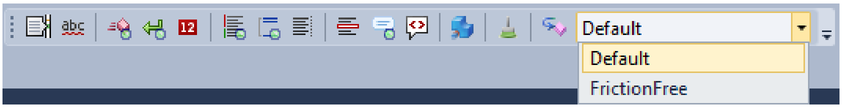
Current scheme: Default

Auto complete
☑ Smart parentheses
☑ Smart brackets
☐ Smart Enter
☑ IntelliAssist

Visualize
☐ Spell checker
☑ Member Icons
☑ Flow breaks
☐ Show code metrics
☑ Structural highlighting
☑ Regions painting
☐ Right margin line
☐ Line highlighter

Level: New User
Language: CSharp

The “Current scheme” shows you the active scheme in the same way as you see it on the DXCore Visualize toolbar:



The “Add new scheme…” and “Delete scheme” buttons are self explanatory.

Then, the list of most “noticeable” CodeRush features follows. If you uncheck the feature, it will be disabled in the current scheme. If you click the small button, the corresponding option page will be opened where you can configure the feature more carefully. The list of features has been added to quickly toggle the features between the two built-in settings schemes:

- Default. This scheme was used in CodeRush for years, and is suggested for users that are familiar with CodeRush and experienced with the most of its features.

- FrictionFree. This scheme is designed for new users who are not very familiar with CodeRush. New users may find some of the features annoying (e.g., unexpected templates expansions) and would like to leave the standard Visual Studio environment unchanged (e.g., no CodeRush visual features). This scheme is set by default on a fresh install.

In the FrictionFree settings scheme, many features are disabled. For instance, the template expansion key is changed to Tab instead of Space Bar. The following features are also inactive in this scheme:

- Visual features: Metrics, Comment Painter, Flow Break Evaluation.

- Smart auto-completion features: Smart Parentheses and Brackets, Smart Enter, Intellassist, Duplicate Line.

- Clipboard features like Intelligent Paste.

- Keyboard shortcuts like F2, Num +, Num -, etc.

- Dozens of code issues.

The built-in schemes are stored separately and can not be removed.

The “What Happened?” (Feature UI) window

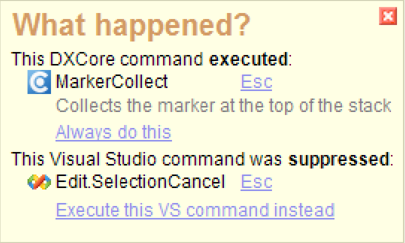
The “**What happened?**” window (also known as the “**Feature UI**” window) appears in the bottom right corner each time a particular IDE tools feature is activated, to inform you what’s going on inside Visual Studio IDE. In the majority of cases, the window is used to simplify learning of the CodeRush specific features. This window informs you of the performed feature, and may suggest the following choices:

Action	Description
OK	Always executes just performed feature without showing the “**What happened?**” window.
Disable	Disables a feature forever.
Options	Opens an appropriate options page for an activated feature.
Template Options	Opens the “Template” options page in the Options Dialog that allows you to customize the expanded template.
Always do this	Same as OK action.

Execute this VS command instead	Same as Disable action. Doesn't suppress the Visual Studio's feature anymore.
Shortcut combination	Opens the Shortcuts options page that allows you to tweak the key combination for this particular feature.

If you perform one of the actions in the "**What Happened?**" window, your choice will be remembered, and the window will no longer appear in the future. Otherwise, the window will always appear until you make a decision on an action for this particular feature. You can tweak your choices for all features performed so far on the "**Core | Features**" options page in the Options Dialog.

Here are several examples of this window:



What happened?

This DXCore command **executed**:

MarkerCollect Esc

Collects the marker at the top of the stack

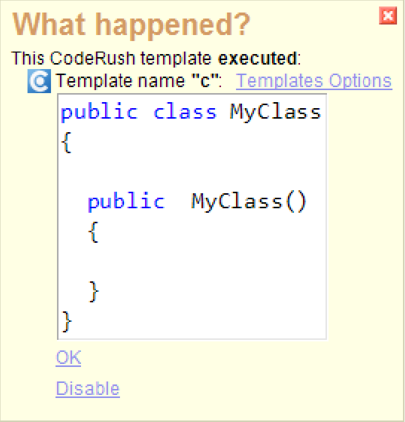
Always do this

This Visual Studio command was **suppressed**:

Edit.SelectionCancel Esc

Execute this VS command instead

A template is expanded:



What happened?

This CodeRush template **executed**:

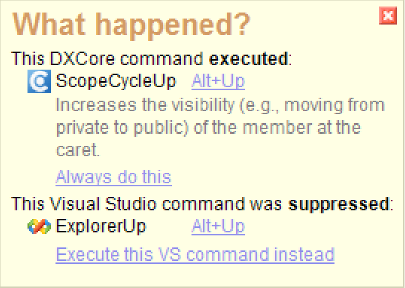
Template name "c": Templates Options

```
public class MyClass
{

  public  MyClass()
  {

  }
}
```

OK

Disable

A conflict shortcut is available:



What happened?

This DXCore command **executed**:

ScopeCycleUp Alt+Up

Increases the visibility (e.g., moving from private to public) of the member at the caret.

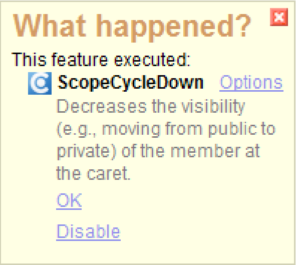
Always do this

This Visual Studio command was **suppressed**:

ExplorerUp Alt+Up

Execute this VS command instead

A particular non-conflicting feature is performed:



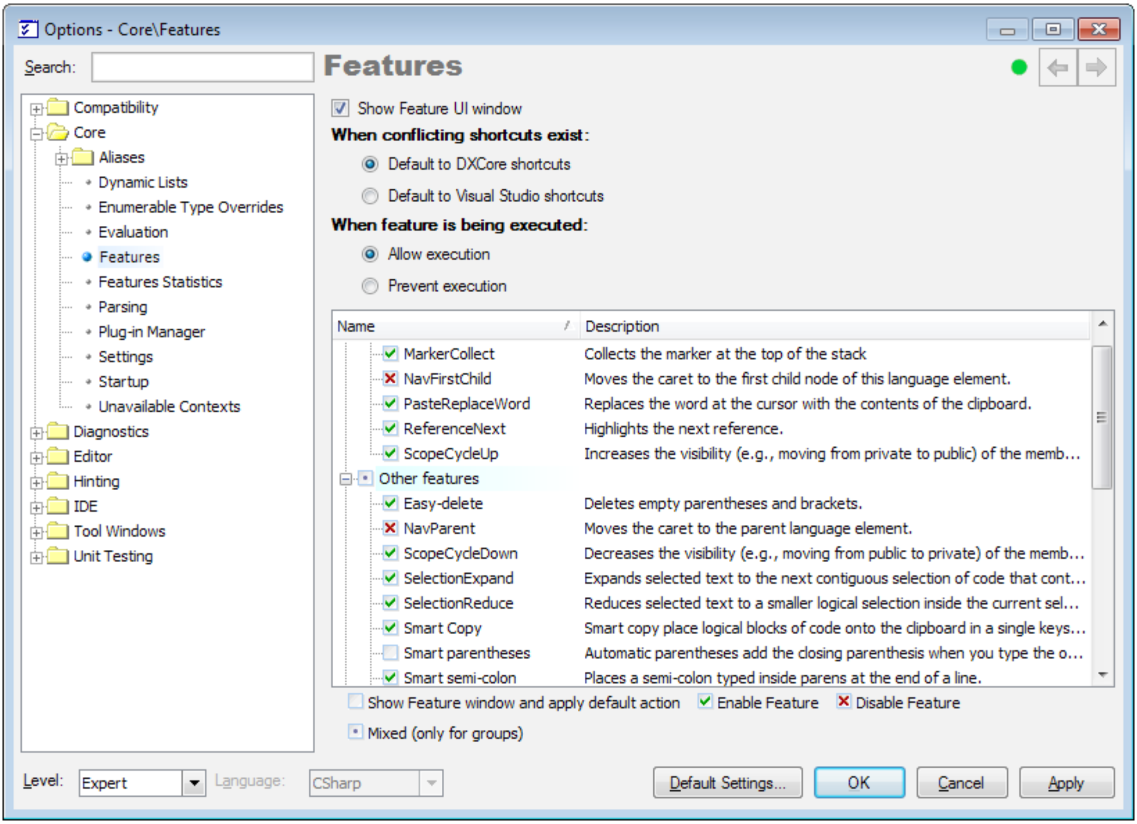
What happened?

This feature executed:

ScopeCycleDown Options

Decreases the visibility (e.g., moving from public to private) of the member at the caret.

You may find this window inconvenient, so you might want to get rid of it (disable it) and ask **IDE Tools** to make decisions automatically for you. This is possible on the same Core | Features options page. To disable the "**What happened?**" window – uncheck the "Show Feature UI window" check box. For automatic decisions on conflicting shortcuts and features execution, change the "When conflicting shortcuts exist" and the "When feature is being executed" options accordingly:



Options - Core\Features

Features

☑ Show Feature UI window

When conflicting shortcuts exist:

● Default to DXCore shortcuts

○ Default to Visual Studio shortcuts

When feature is being executed:

● Allow execution

○ Prevent execution

Name	Description
☑ MarkerCollect	Collects the marker at the top of the stack
☒ NavFirstChild	Moves the caret to the first child node of this language element.
☑ PasteReplaceWord	Replaces the word at the cursor with the contents of the clipboard.
☑ ReferenceNext	Highlights the next reference.
☑ ScopeCycleUp	Increases the visibility (e.g., moving from private to public) of the memb...
Other features	
☑ Easy-delete	Deletes empty parentheses and brackets.
☒ NavParent	Moves the caret to the parent language element.
☑ ScopeCycleDown	Decreases the visibility (e.g., moving from public to private) of the memb...
☑ SelectionExpand	Expands selected text to the next contiguous selection of code that cont...
☑ SelectionReduce	Reduces selected text to a smaller logical selection inside the current sel...
☑ Smart Copy	Smart copy place logical blocks of code onto the clipboard in a single keys...
□ Smart parentheses	Automatic parentheses add the closing parenthesis when you type the o...
☑ Smart semi-colon	Places a semi-colon typed inside parens at the end of a line.

□ Show Feature window and apply default action ☑ Enable Feature ☒ Disable Feature

▪ Mixed (only for groups)

Level: Expert Language: CSharp

Getting up to speed with IDE tools

How to learn the product tips:

- Dock the CodeRush Training window (**DevExpress** | **Tool Windows** | **CodeRush**) next to the editor. The **CodeRush Training** tool window shows you features that can be used while the cursor is at the current position, and is designed to help you learn much of CodeRush and Refactor! without diving into the **User Guide**.

- Read the User Guide. From the DevExpress menu select “**User Guide**”. Use the tree list to drill into topics of interest.

- Access training videos through the **DevExpress** menu or via this link.

- Watch online videos at DevExpress TV Channel.

- Subscribe to this blog.

Useful features to get up to speed:

- Code navigation features

  - Press **Alt**+**Home** to drop a stack-based marker. Press **Esc** to collect it. Markers are navigation waypoints that help you find your way back through important points in your code.

  - Navigate through all references to the identifier or type at the caret, by pressing the **Tab** key (**Shift**+**Tab** navigates back, and/or **Esc** brings you to the place where you first pressed **Tab**).

- Code selectionfeatures

  - Widen the selection by logical blocks by pressing **Num+** on the numeric keypad, or **Ctrl**+**W**. Once a selection has been widened, you can reduce it using **Num-** on the numeric keypad, or **Ctrl**+**Shift**+**W**.

  - Wrap multi-line selections in *try*/*catch* blocks by pressing the letter “**c**“. Look to the training window for more wrapping shortcuts, when the selection extends beyond two or more lines.

- Code generation features

  - Create code blocks with the template (e.g. type the letter “**b**“, and then press the spacebar to get brace block pairs in *C#*, *C++* or *JavaScript* or, similarly, you can wrap a multi-line selection inside braces by pressing the letter “**b**“).

  - Use the Refactor! key for TDD-style development. Create your client calls first, and then press the **Refactor! key** to declare methods, classes, structures, interfaces, fields, locals, etc.

- Code helpers and refactorings

  - Review the list of refactorings in Refactor! Pro. The full list of refactorings you have can be also found on the “**Editor | Refactoring | Catalog**“ in the Options Dialog.

  - Change the visibility of the member or type containing the caret by pressing the **Alt**+**Up** and **Alt**+**Down** keys.

  - Review the list of available shortcuts for CodeRush and Refactor! features.

The CodeRush/Refactor! key (shortcut)

The common shortcut for invoking refactorings is the **CTRL**+` (backtick) key combination. This shortcut invokes the Refactor! (CodeRush) popup menu which lists all the refactorings available for the current context. Note that there are also other ways to perform a code refactoring.

On some non-English keyboards it may not work. To change the shortcut, please do the following:

1. Use the “Ctrl+Shift+Alt+O” shortcut to invoke the Options Dialog.

2. In the tree view on the left, navigate to this folder: IDE

3. Select the Shortcuts options page.

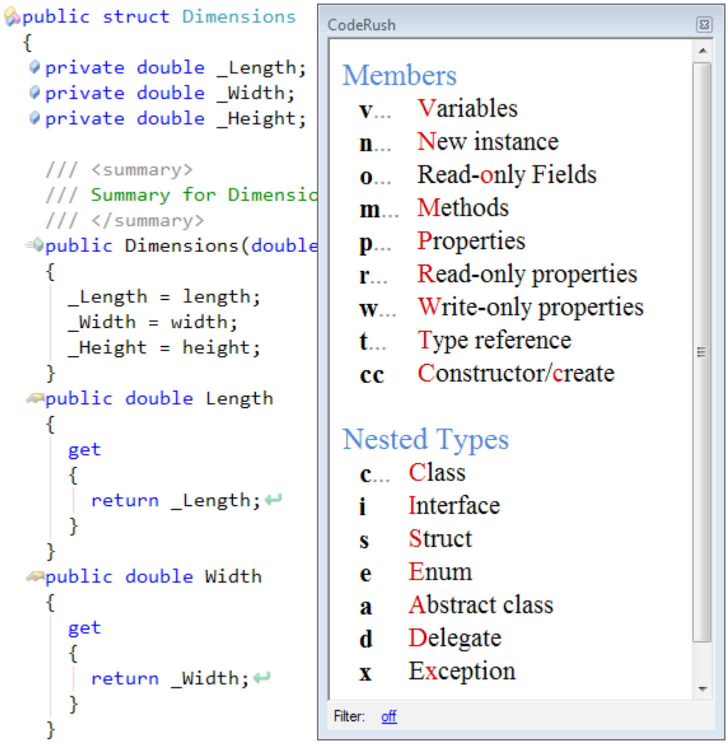
4. Click the Find toolbar button and search for a shortcut, linked to the “**SmartTagExecute**” command.

5. Modify it as you require.

CodeRush Training window

The **CodeRush Training** tool window shows you features that can be used while the cursor is at the current position. The tool window shows you features that can be used while the cursor is at the current position, e.g. it can list available templates, refactorings, selection and navigation features, and others. If a feature can be accessed via a shortcut, this shortcut is shown beside the feature. The window can be accessed via the **DevExpress** | **Tool Windows** | **CodeRush** menu item in your IDE.

For instance, if the cursor is anywhere within a class, the tool window will list all templates available for the current context:



```
public struct Dimensions
{
  private double _Length;
  private double _Width;
  private double _Height;

  /// <summary>
  /// Summary for Dimensio
  /// </summary>
  public Dimensions(double
  {
    _Length = length;
    _Width = width;
    _Height = height;
  }
  public double Length
  {
    get
    {
      return _Length;
    }
  }
  public double Width
  {
    get
    {
      return _Width;
    }
  }
```

CodeRush

Members

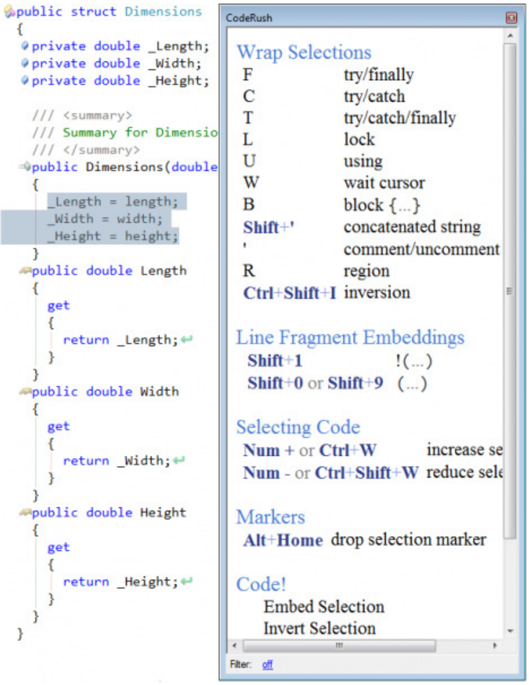
v...	Variables
n...	New instance
o...	Read-only Fields
m...	Methods
p...	Properties
r...	Read-only properties
w...	Write-only properties
t...	Type reference
cc	Constructor/create

Nested Types

c...	Class
i	Interface
s	Struct
e	Enum
a	Abstract class
d	Delegate
x	Exception

Filter: off

Otherwise, if you have certain code selected, you’ll get features to work with selections:



```
public struct Dimensions
  {
   private double _Length;
   private double _Width;
   private double _Height;

    /// <summary>
    /// Summary for Dimensio
    /// </summary>
   public Dimensions(double
    {
      _Length = length;
      _Width = width;
      _Height = height;
    }
   public double Length
    {
      get
      {
        return _Length;
      }
    }
   public double Width
    {
      get
      {
        return _Width;
      }
    }
   public double Height
    {
      get
      {
        return _Height;
      }
    }
  }
```

CodeRush

Wrap Selections

F	try/finally
C	try/catch
T	try/catch/finally
L	lock
U	using
W	wait cursor
B	block {...}
Shift+'	concatenated string
'	comment/uncomment
R	region
Ctrl+Shift+I	inversion

Line Fragment Embeddings

Shift+1	!(...)
Shift+0 or Shift+9	(...)

Selecting Code

Num + or Ctrl+W	increase se
Num - or Ctrl+Shift+W	reduce sele

Markers

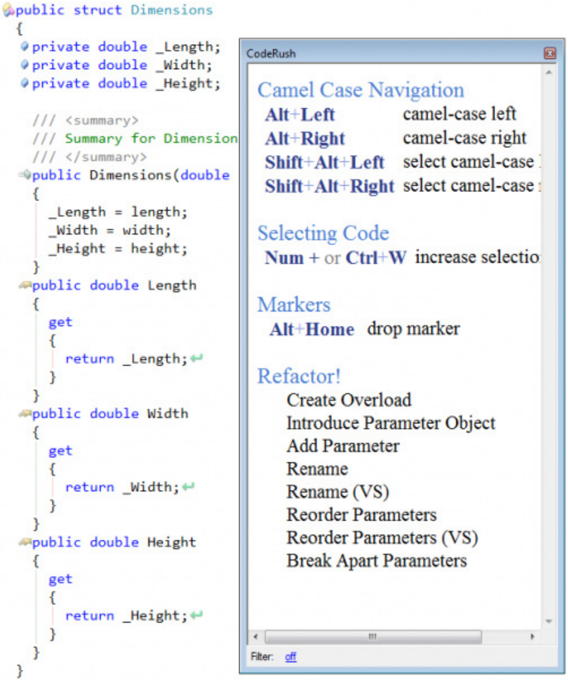
Alt+Home	drop selection marker

Code!

Embed Selection
Invert Selection

Filter: off

If a caret is on a method name, you’ll see the list of refactorings available:



```
public struct Dimensions
  {
   private double _Length;
   private double _Width;
   private double _Height;

    /// <summary>
    /// Summary for Dimension
    /// </summary>
   public Dimensions(double
    {
      _Length = length;
      _Width = width;
      _Height = height;
    }
   public double Length
    {
      get
      {
        return _Length;
      }
    }
   public double Width
    {
      get
      {
        return _Width;
      }
    }
   public double Height
    {
      get
      {
        return _Height;
      }
    }
  }
```

CodeRush

Camel Case Navigation

Alt+Left	camel-case left
Alt+Right	camel-case right
Shift+Alt+Left	select camel-case
Shift+Alt+Right	select camel-case

Selecting Code

Num + or Ctrl+W	increase selectio

Markers

Alt+Home	drop marker

Refactor!

Create Overload
Introduce Parameter Object
Add Parameter
Rename
Rename (VS)
Reorder Parameters
Reorder Parameters (VS)
Break Apart Parameters

Filter: off

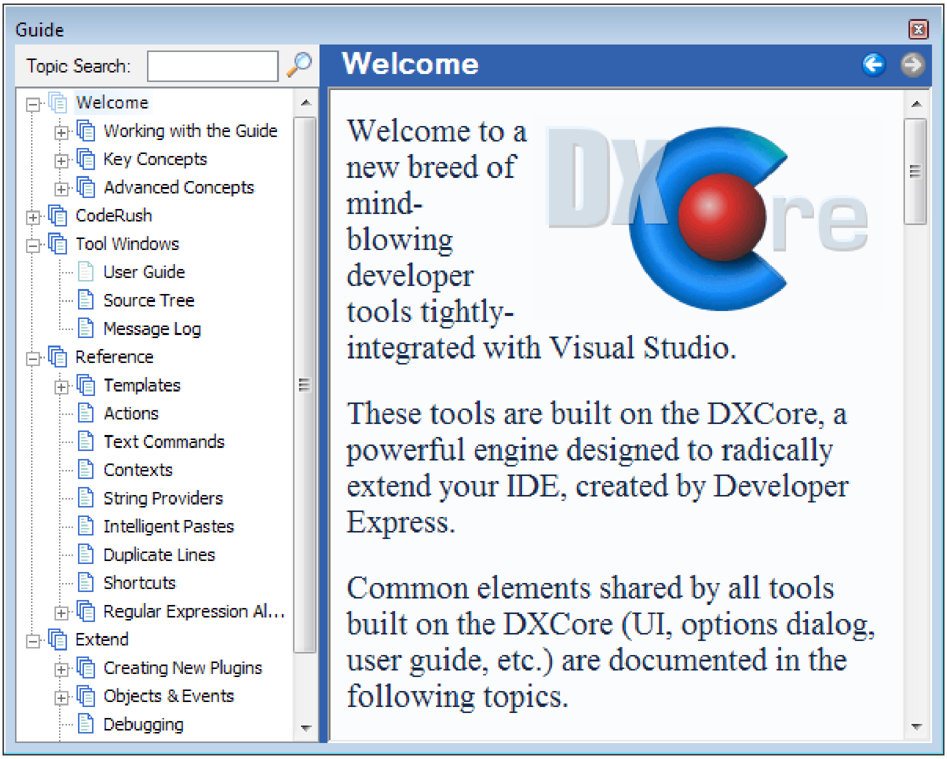
Note: the window is context-sensitive, so it may decrease performance a bit while working in Visual Studio.

IDE Tools User Guide

The **User Guide** documents the DXCore and dependent products (such as CodeRush and/or Refactor! if they're installed) and also includes documentation covering Visual Studio extensibility. This tutorial can be viewed by selecting "**User Guide**…" from the **DevExpress** menu. You can also view it by selecting "**Guide**" from the **DevExpress** | **Tool Windows** menu.

Window overview

The guide consists of an organized hierarchy of topics (the topic tree), a splitter, toolbar, and this content window. You can hide and display this tree list by double-clicking on the dark blue title bar/splitter directly above this text. When the tree list is visible, just above it, you'll find the **Topic Search** edit and the topic history browse buttons. In the **Search** text box, you can enter a few characters of a topic title to display that topic immediately.



Welcome to a new breed of mind-blowing developer tools tightly-integrated with Visual Studio.

These tools are built on the DXCore, a powerful engine designed to radically extend your IDE, created by Developer Express.

Common elements shared by all tools built on the DXCore (UI, options dialog, user guide, etc.) are documented in the following topics.

Orientation/Docking

You can position the topic tree above or adjacent to this content window. You can also have the topic tree view positioned automatically, based on the aspect ratio of the window.

If you would like this user guide to appear as a tabbed document among your source files, right-click on the caption bar of this user guide tool window and uncheck the "*Dockable*" entry. This will display the **user guide** in the document area.

Dynamic Content

Many sections of the user guide are dynamically generated based on current settings, state, and the plug-ins that you have loaded. Dynamic content ensures this guide will stay up to date, even as your **IDE tools** install changes with new modules, and as you explore and configure new features of the product.

Note: this user guide has the ability to document .NET classes using reflection and XML documentation comments. This technology is primarily used to document classes for extending **DXCore**, but it is also possible to drill down into .NET classes (providing those classes are available to our reflection code).

To learn more about **User Guide** window, see the “Working with the Guide” topic in the **User Guide** itself. To learn more about populating it with static and dynamic content – see the appropriate topic.

Prerequisites

Supported Visual Studio versions

Product Name	Visual Studio 2005	Visual Studio 2008	Visual Studio 2010	Visual Studio 2012
DXCore	✗	✓	✓	✓
CodeRush Pro	✗	✓	✓	✓
CodeRush Xpress	✗	✓	✓	✓
Refactor! Pro	✗	✓	✓	✓

*Note: **IDE Tools** can't be used in Visual Studio Express Editions because 3rd party extensibility is not supported, and not allowed in Express editions of Visual Studio.*

*Note2: **DevExpress** stopped supplying the **IDE tools** version compatible with Visual Studio 2002 and Visual Studio 2003. However, these versions are available for customers who purchased one of the latest product versions. So, once you or your company purchases the license to CodeRushor Refactor! Pro, you'll be able to download the version which supports VS 2002 and VS 2003. Please refer to the following page to learn how to access it: How do I obtain an older verion?*

Supported programming languages

Product Name	#C	VB	++C	XML	HTML	XAML	ASP. NET	Java Script
DXCore	✓	✓	✓	✓	✓	✓	✓	✓
CodeRush Pro	✓	✓	✓	✓	✓	✓	✓	✓
CodeRush Xpress	✓	✓	✗	✗	✗	✗	✗	✗
Refactor! Pro	✓	✓	✓	✓	✓	✓	✓	✓

System requirements

This is another frequent question from new CodeRush users, which is not exactly specified anywhere yet bes cause **CodeRush** does not really rely on hardware or software requirements – it should run OK everywhere your Visual Studio is installed.

Supported Operating Systems

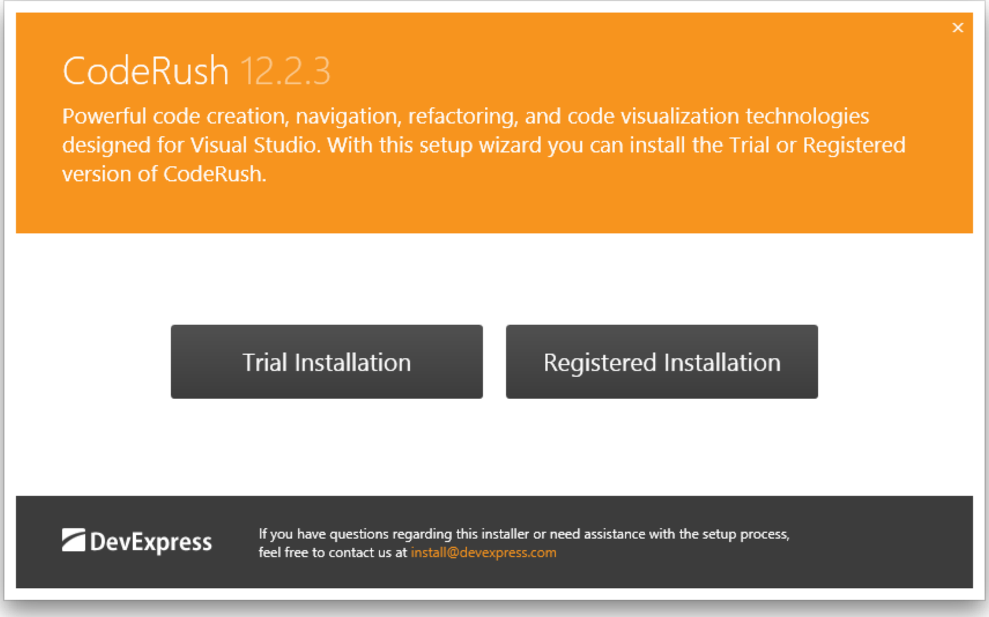
CodeRush supports any operating system no matter whether it is 32 or 64 bits with Visual Studio installed.

Hardware requirements

CodeRush does not require a lot of memory installed on your computer and does not need a high-speed processor.

How to install the IDE Tools

Once you download an installer from the DevExpress website, run it, and follow the instructions on the screen:

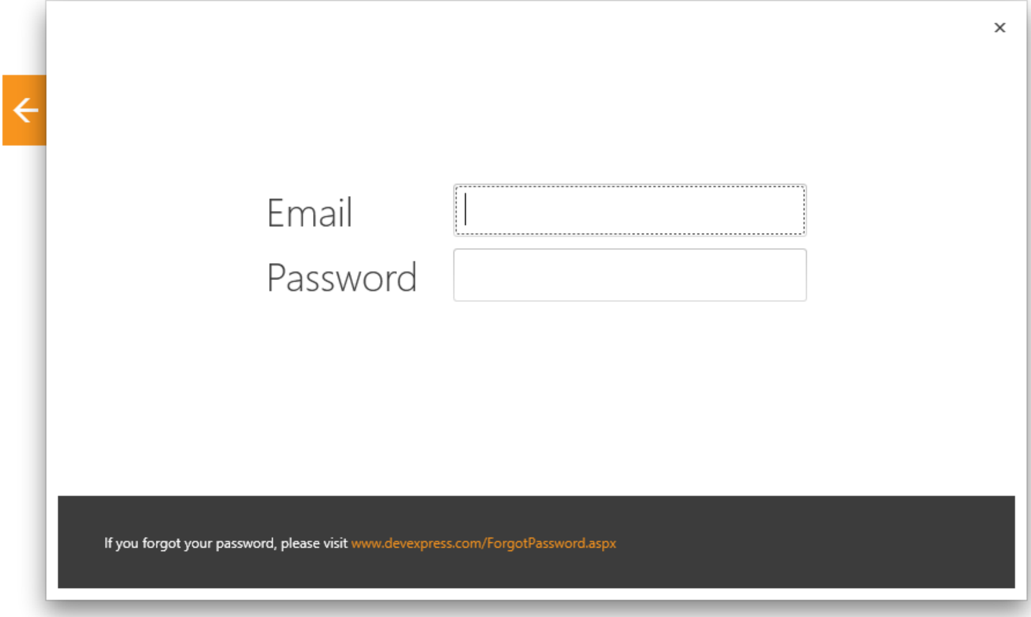


CodeRush 12.2.3

Powerful code creation, navigation, refactoring, and code visualization technologies designed for Visual Studio. With this setup wizard you can install the Trial or Registered version of CodeRush.

If you have questions regarding this installer or need assistance with the setup process, feel free to contact us at install@devexpress.com

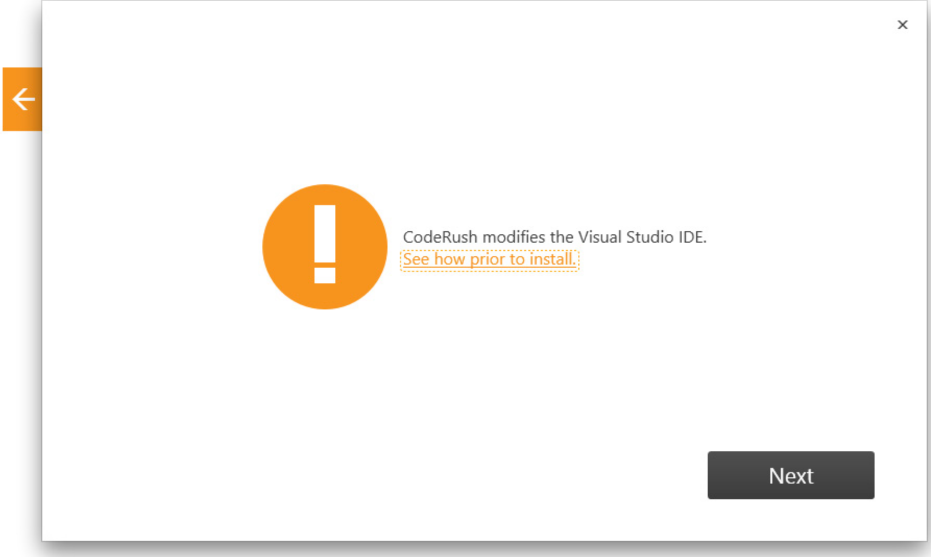
If you’re a registered user, choose the Registered Installation option, and enter your details on the next screen:



If you forgot your password, please visit www.devexpress.com/ForgotPassword.aspx

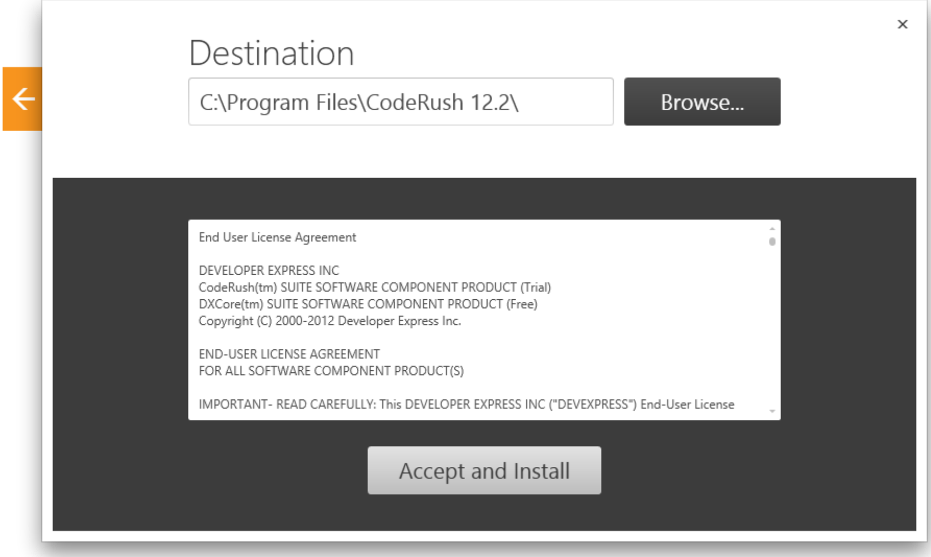
If you don’t have a license, you may install the product in the trial mode that lasts for 30 days. The functionality of the standard products, when installed in trial mode, is not reduced during this period.

On the next screen, you will be notified that CodeRush modifies the Visual Studio IDE:



CodeRush modifies the Visual Studio IDE.
See how prior to install.

I recommend you click the “See how prior to install” link to learn more. Then, select the destination installation folder and accept the terms of the End User License Agreement, if you’re agree with them:



Destination
C:\Program Files\CodeRush 12.2\

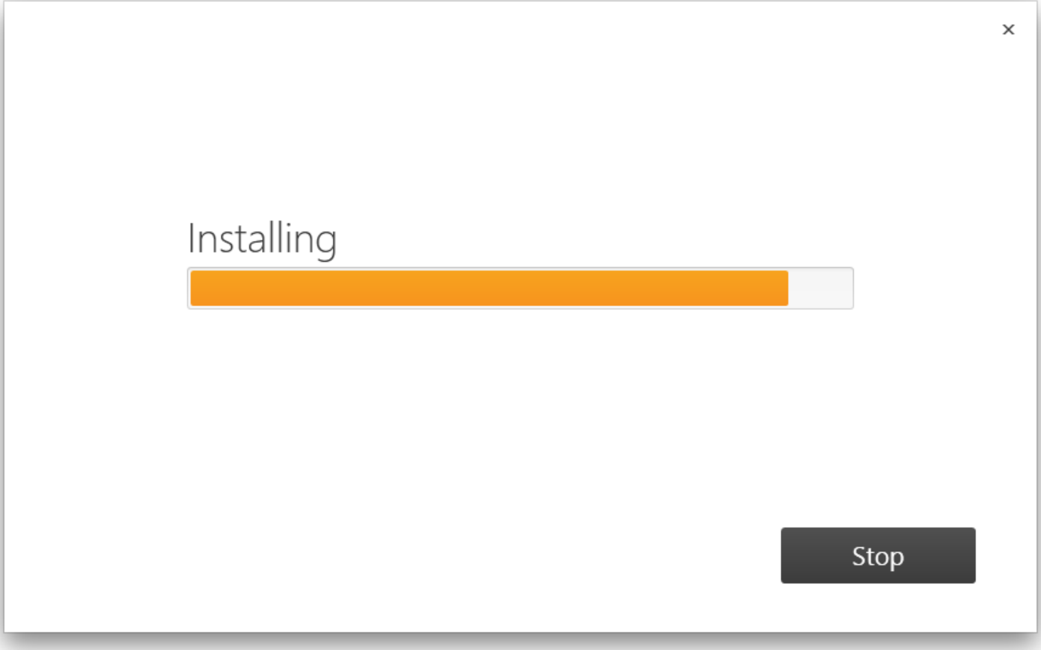
End User License Agreement

DEVELOPER EXPRESS INC
CodeRush(tm) SUITE SOFTWARE COMPONENT PRODUCT (Trial)
DXCore(tm) SUITE SOFTWARE COMPONENT PRODUCT (Free)
Copyright (C) 2000-2012 Developer Express Inc.

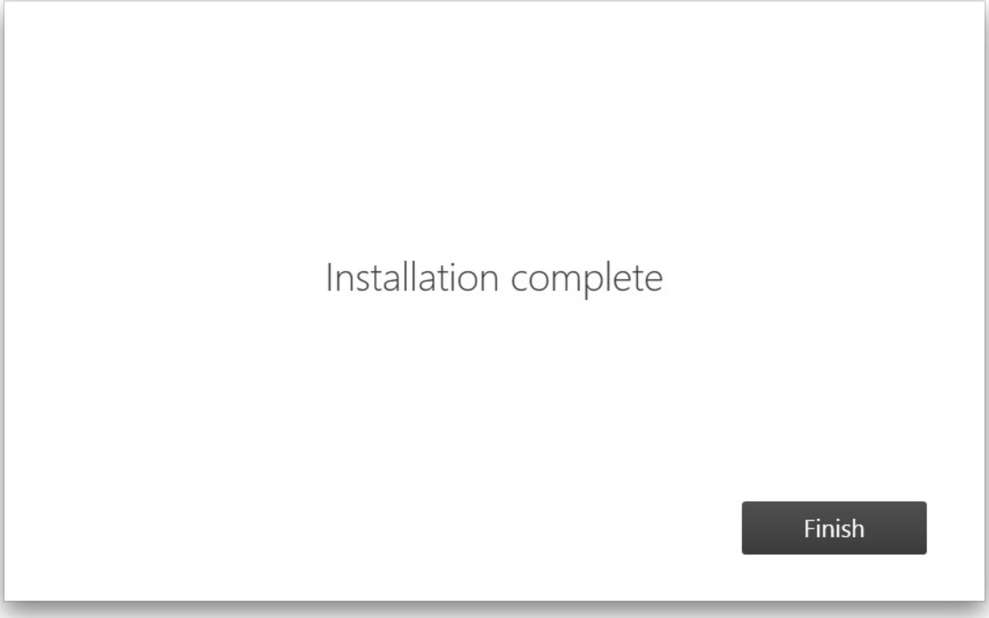
END-USER LICENSE AGREEMENT
FOR ALL SOFTWARE COMPONENT PRODUCT(S)

IMPORTANT- READ CAREFULLY: This DEVELOPER EXPRESS INC ("DEVEXPRESS") End-User License

Installation progress is shown on the next screen. It may take up to several minutes to install, depending on the Visual Studio versions you have installed:



After the installation process is complete, click on the Finish button, and you’re done:



If you have questions regarding this installer or need assistance with the installation process, feel free to contact **DevExpress** by writing to: install@devexpress.com or support@devexpress.com.

How to install multiple IDE Tools versions side-by-side

Different versions of **IDE Tools** can be installed together side-by-side on a single machine. Note, however, you can have only one version running at a given time.

There's a **DXCoreVersion** tool which allows you to switch between installed **IDE Tools** versions any time. You can find the "DXCoreVersion.exe" tool in your bin folder which looks similar to this path:

"%Program Files%\DevExpress %YourVersion%\IDETools\System\DXCore\BIN"

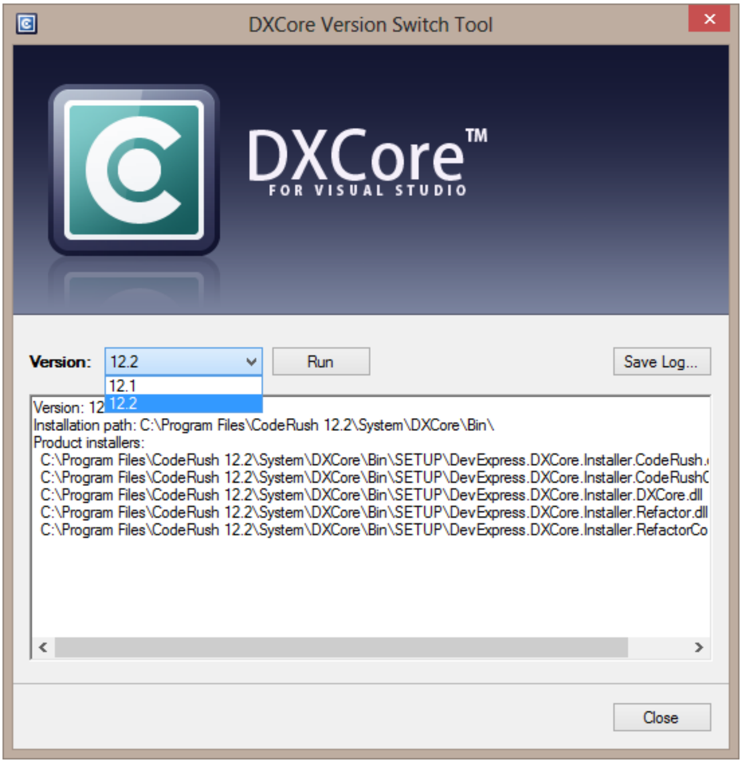
For versions greater than v12.1, the path may look as follows:

"%Program Files%\CodeRush %YourVersion%\System\DXCore\BIN"

where %YourVersion% is a version of **CodeRush (IDE Tools)** you have installed.

Follow these steps to switch your tools version:

1. Close all the running instances of Visual Studio.
2. Go into the "CodeRush\System\DXCore\BIN" folder in the most recent install location. Actually, you could go to the folder for any of the installs, but it is recommended to choose to go to the most recent one just in case there's been a fix or an addition to the version switch tool.
3. Run the DXCoreVersion.exe application:



```
Version: 12
Installation path: C:\Program Files\CodeRush 12.2\System\DXCore\Bin\
Product installers:
  C:\Program Files\CodeRush 12.2\System\DXCore\Bin\SETUP\DevExpress.DXCore.Installer.CodeRush.
  C:\Program Files\CodeRush 12.2\System\DXCore\Bin\SETUP\DevExpress.DXCore.Installer.CodeRush
  C:\Program Files\CodeRush 12.2\System\DXCore\Bin\SETUP\DevExpress.DXCore.Installer.DXCore.dll
  C:\Program Files\CodeRush 12.2\System\DXCore\Bin\SETUP\DevExpress.DXCore.Installer.Refactor.dll
  C:\Program Files\CodeRush 12.2\System\DXCore\Bin\SETUP\DevExpress.DXCore.Installer.RefactorCo
```

4. The version selected in the dropdown box will be the latest version installed, not necessarily the active version. Select the **IDE Tools** version you'd like to be active and running, then click the "Run" button.

5. In the log window you'll see that the version of **IDE Tools** currently set up will be unregistered and the version you selected will be registered (technically, this process doesn't add or remove the installation from your system).

After the process is complete the message box will be shown with a message about your selected version. You may save the log pressing the "Save Log…" button in case your switch process wasn't successful or you'd like to send the log to the DevExpress Support Services. You can use this tool to switch from any installed version you have to any other version at any time. Also, it can be used for fast re-registering of a specific product version if you encounter product installation issues.

How to install/uninstall IDE tools in quiet/silent mode

IDE tools installation can be launched in silent (without GUI and no questions asked) mode. Here's a sample comamand line:

```
IDETools-10.1.0.exe /Q /EMAIL:myaddress@company.com /CUSTOMERID:A1234 /PASSWORD:MYPASS /DEBUG
```

Available switches:

/**Q** – quite mode without GUI, no questions asked.
/**INSTALLPATH**:<path> – allows you to specify an installation directory instead of %ProgramFiles%\Developer Express .NET v10.1 used by default.
/**EMAIL**:email – email is your email address of your Client Center account.
/**CUSTOMERID**:id – id is your Security ID in the Client Center.
/**PASSWORD**:pwd - "pwd" is your Client Center password (case-sensitive).
/**DEBUG** – forces the log file creation.

Available arguments:

-U – uninstall/remove mode

Sample:

```
IDETools-10.1.0.exe /Q -U
```

How to temporally disable IDE tools (load manually)

It is possible to prevent the IDE Tools from being loaded automatically when starting Visual Studio.

To do this, please follow these steps:

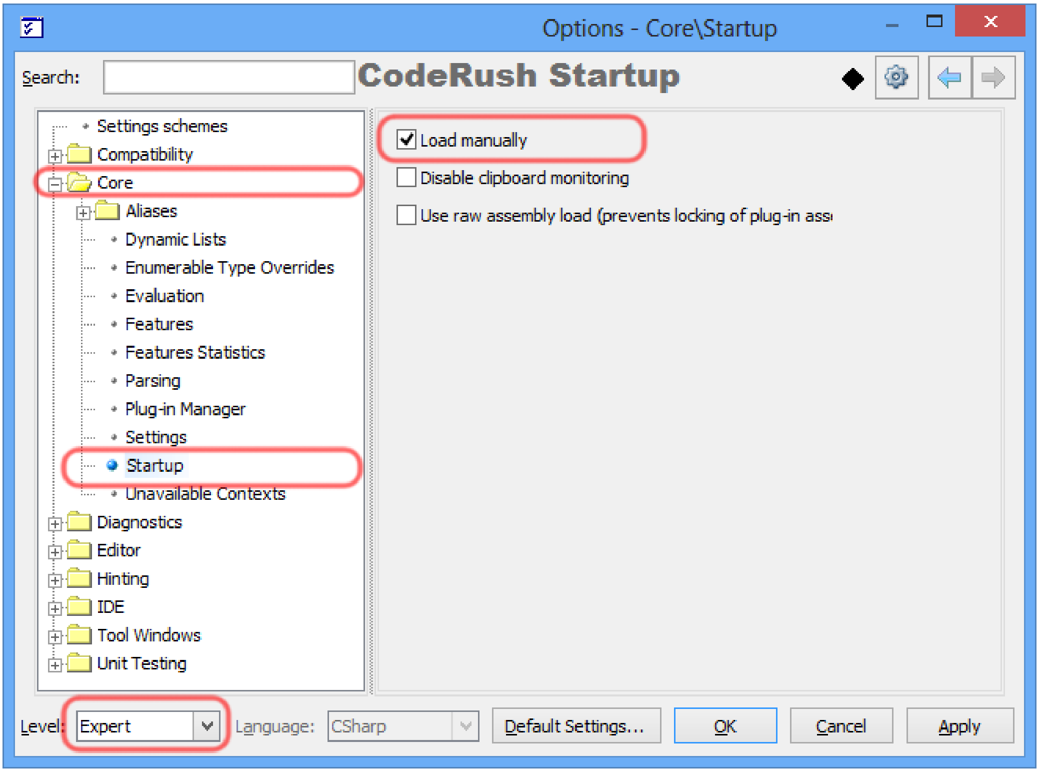
1) In the Visual Studio IDE, select the Options… menu item in the **DevExpress** menu or press the Ctrl+Shift+Alt+O key combination to invoke the Options Dialog.

2) In the tree view on the left, navigate to the "Core" folder.

3) Select the Startup options page.

Note that this page level is Expert, and will only be visible if the Level combo on the lower-left of the Options dialog is set to Expert.

4) Activate the "Load Manually" option and click OK:



Options - Core\Startup

CodeRush Startup

☑ Load manually

□ Disable clipboard monitoring

□ Use raw assembly load (prevents locking of plug-in ass

Level: Expert

Language: CSharp

Another way to change this option is to tweak your Windows Registry key manually (via the regedit.exe tool):

HKEY_CURRENT_USER\Software\Developer Express\CodeRush for VS\%YourProductVersion%\LoadManually

where %YourProductVerions% is a version of the DevExpress IDE Tools you have installed, and the “LoadManually” is the key of type string which value needs to be modified. Set the value to “True” (without quotes) if you would like to prevent the automatic loading of the IDE Tools at Visual Studio startup. Note, if the registry key doesn’t exist you need to create it manually using the Registry editor.

DevExpress support and knowledge base

Technical support is available in a variety of different forms. If you need technical support, please use one of the options listed below. Also, it is recommended that you review different help resources, provided by DevExpress, to help you use their products, before contacting to Support Team.

- Support Center at http://www.devexpress.com/sc

This is the primary channel where you can get answers to your questions, report bugs and make suggestions. Whenever you want to create a new issue (ask a question, post a bug or a request) they advise that you look through the currently posted issues. A similar issue may already have been reported, so you’ll find a quick answer.

- support@devexpress.com

Sending an e-mail to this address is an alternative to using the Support Center. You can e-mail to submit a bug report, request new features or ask any questions related to any **DevExpress** product.

Chapter 1. Duplicate Code Detection and Consolidation

Duplicate Detection and Consolidation overview

Duplicate code, or sometimes referred to as a clone, is a program source code fragment that is very similar to another code fragment. A code clone may occur more than twice, either within a single program or across different programs owned or maintained by the same group of developers. **Code duplication** is considered an expensive practice that should be avoided because it complicates the maintenance and evolution of the software.

There are many reasons why **duplicate code** can appear in source code:

- **Clipboard Inheritance**. Copying and pasting code (and then modifying it as needed) is faster and easier than writing similar code from scratch.

- **Coding Guidelines**. Sometimes the application or the team's coding guidelines may call for a frequently-needed code fragment to appear throughout an application, such as error handling, logging, or wiring up user interface displays. The fragment will intentionally appear throughout the code to maintain the style.

- **Performance Enhancements**. Some code clones may exist for perceived or real performance gains. Systems with incredibly tight time constraints are often hand-optimized by replicating frequent computations.

- **Big Tasks, Little Time**. It is generally difficult for a single developer to understand all the code in a large software system. Pressure to ship features in a short period of time can incentivize the use of example-oriented programming, where a developer copies and adapts existing code already developed.

- **Design Patterns**. Also, the repeated use of design patterns can lead to similar functionality spread across a software system. For example, applying the composite design pattern more than once can lead to multiple distinct classes, each implementing the Leaf and Composite parts. If these implementations share similar properties to support navigation or search (such as access to the Parent or Children), this can lead to similar code located in distinct classes, all designed to implement navigation/search/validation functionality in a similar way, each a specialist for working with one of the different Leaf/Composite pairs.

- **The Cost of Crossing Project Boundaries**. Because it's more expensive to consolidate duplicate functionality across project boundaries, when functionality from one high-level project is needed in another, the cost of consolidating in that moment (which could include an architectural change that needs approve) acts as an incentive to duplicate the code and ensure its survival.

- **Generated Code**. Code generators can create significant quantities of code in a short period of time, and sometimes that code includes repeated patterns of functionality.

- **Unintentionally**. Over time, the spec may call for two functionally similar blocks of code (for example, find best and worst performers from a list), or it may be that two developers were involved in implementing the similar logic, or perhaps it was one developer working at two distinct times in the code. Regardless, it is possible to independently and unintentionally create functionally duplicate code.

Why do I need DDC?

Research in software maintenance has shown that many programs contain a significant amount of duplicate code, estimated to be somewhere between 5% and 20% [1]. Code duplication can significantly increase software maintenance costs because it:

- Increases the time it takes for developers to get up to speed due to the additional code bulk.

- Makes the code harder to ready by obscuring the purpose of each duplicate.

- Forms an effective barrier to enhancing the software since similar changes must be made to all the clones or the clones must consolidated into a single block of code before introducing change.

- Negatively impacts your company's reputation due to update anomalies (e.g., a bug is fixed in an update only to be rediscovered later by customers). Inconsistent updates (e.g., fixing only one of several cloned bugs) can easily turn into unexpected program behavior, where the software only works some of the time.

- Increase the amount of test cases needed, which can lead to more code duplication (in the test cases).

So, we might be interested in finding and consolidating all duplicate code for the following reasons:

- **Decreasing software maintenance costs**. If one is sure that the code segment where a bug is found occurs only once in the system, one can be confident that the bug has been eradicated from the syse tem.

- **Repairing design-flaws**. Code duplication may indicate design problems like missing use of specific programming techniques, such as inheritance.

- **Reducing code size**. Refactoring duplicated code into a single code block reduces the overall size of the code base and result in a faster compile.

Detection

Different algorithms have been proposed to detect duplicate code. One of the primary challenges in detection algorithms is that it is not known beforehand which code fragments can be found multiple times. For example, in a long method, only a subset of that method may be duplicated. Comparing all possible subsets of all methods against each other means you're talking about a huge number of comparisons that grows significantly as the solution size scales.

DDC uses an algorithm that is conceptually independent of the programming language of the source code being analyzed, working at the level of abstract syntax trees (ASTs). This means it can find functionally duplicate code inside the source code written in different languages (e.g. CSharp and Visual Basic). It scans through the code of the entire solution and reports all duplicates found.

Duplicate detection ignores comments, whitespace, curly braces, variable and parameter names, etc., as you would expect from a modern tool. However CodeRush's duplicate detection algorithm goes beyond this to also find code that is functionally similar. The level of similarity defines the size of duplicate code blocks found, whether it is an entire method or a code block consisting of only a few lines.

Consolidation

A primary purpose of **duplicate code detection** is removing duplication from the system, through refactoring, for improving the code quality of the software system. The process of eliminating duplicate code is called code consolidation, and it consists of a single block of code that replaces all code duplications. With consolidation, you can decrease complexity, reduce potential sources of errors emerging from duplicated code, increase readability, and increase system flexibility.

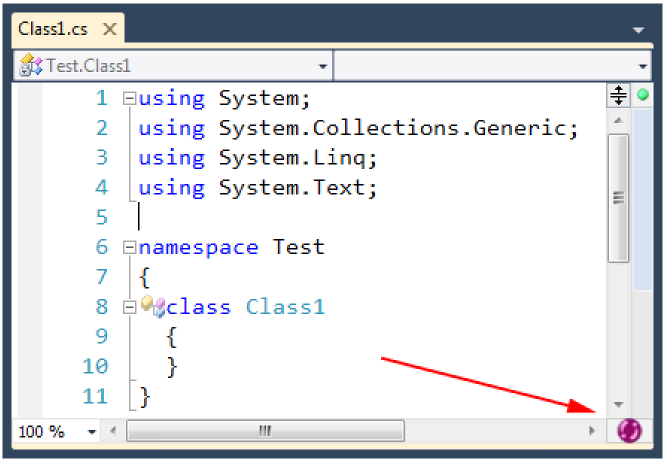
DDC allows you to consolidate most of the duplicate code found automatically, in a single click.

[1] A Survey on Software Clone Detection Research. Chanchal Kumar Roy and James R. Cordy. September, 2007.

See also Mark Miller's blog – Duplicate Detection and Consolidation in CodeRush for Visual Studio.

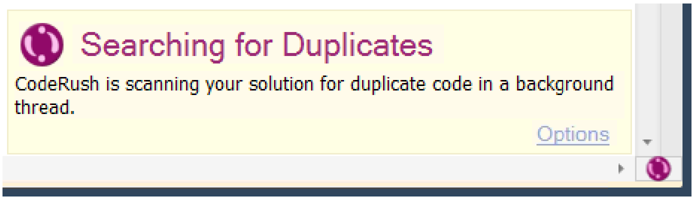
Duplicate Real-time detection in Visual Studio

CodeRush Duplicate detection runs automatically inside Visual Studio IDE in a background thread. To enable it, go to the Duplicate Code options page in the Options Dialog and toggle its availability. Note that you should also have the Code Issues Analysis feature enabled. Once you enable detection, it starts scanning your solution for code duplication. You will see an animated icon in the lower right corner visually indicating the progress of duplicate code analysis:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Test
{
  class Class1
  {
  }
}
```

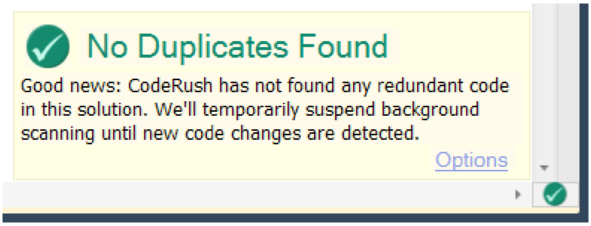
Hover over this icon to see a message hint with additional information:



Searching for Duplicates

CodeRush is scanning your solution for duplicate code in a background thread.

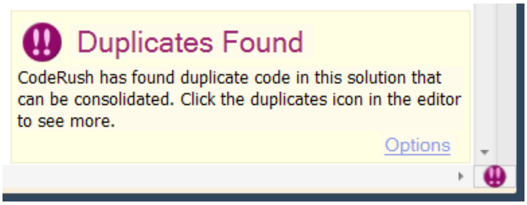
If the analyzed solution does not have duplicate code, the icon state changes:



No Duplicates Found

Good news: CodeRush has not found any redundant code in this solution. We'll temporarily suspend background scanning until new code changes are detected.

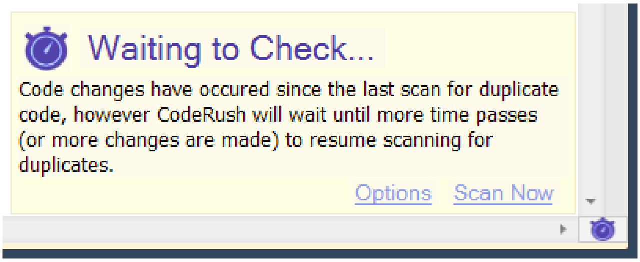
If there is duplicate code in the solution, the following state of the icon is shown:



Duplicates Found

CodeRush has found duplicate code in this solution that can be consolidated. Click the duplicates icon in the editor to see more.

When you make changes to your source code, the analysis is suspended and it will be automatically resumed later, according to your analysis settings. If you want to manually resume the analysis, click the *Scan Now* link in the *Waiting to Check* hint:

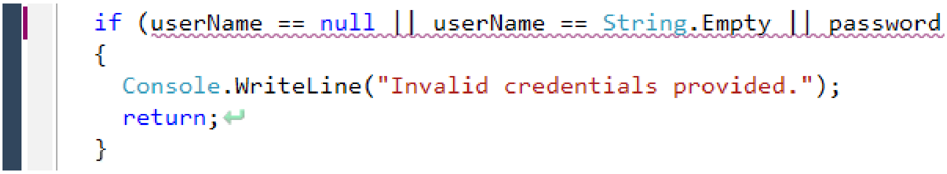


Waiting to Check...

Code changes have occured since the last scan for duplicate code, however CodeRush will wait until more time passes (or more changes are made) to resume scanning for duplicates.

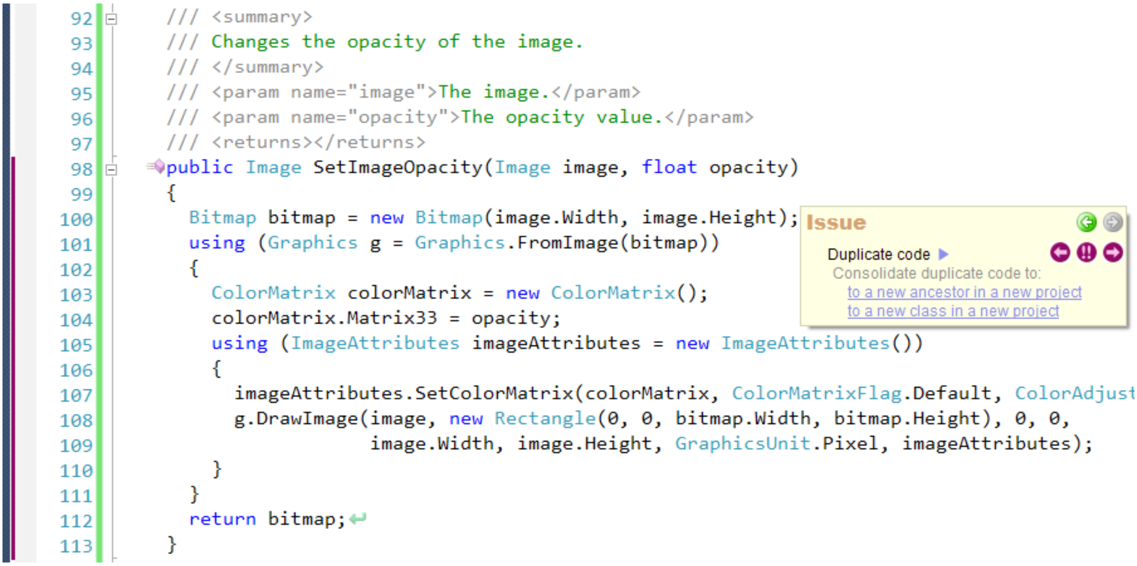
Clicking the *Options* link will bring up the Duplicate Code options page.

Once the duplicate code is found, a code issue of a special type highlights the code duplication:



```
if (userName == null || userName == String.Empty || password
{
  Console.WriteLine("Invalid credentials provided.");
  return;
}
```

Hovering over the code issue will show you a Code Fix hint with the **Duplicate Code** item:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{
  Bitmap bitmap = new Bitmap(image.Width, image.Height);
  using (Graphics g = Graphics.FromImage(bitmap))
  {
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttributes = new ImageAttributes())
    {
      imageAttributes.SetColorMatrix(colorMatrix, ColorMatrixFlag.Default, ColorAdjust
      g.DrawImage(image, new Rectangle(0, 0, bitmap.Width, bitmap.Height), 0, 0,
                  image.Width, image.Height, GraphicsUnit.Pixel, imageAttributes);
    }
  }
  return bitmap;
}
```

Issue
Duplicate code
Consolidate duplicate code to:
to a new ancestor in a new project
to a new class in a new project

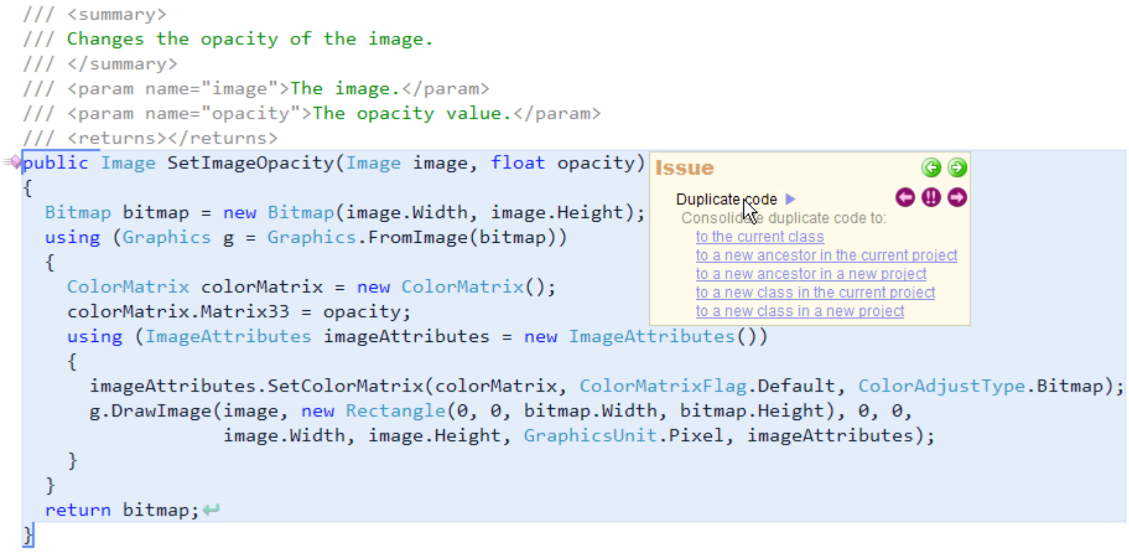
There are several interesting things in this hint related to the **Duplicate Code** code issue:

- Duplicate code label
- Navigation buttons
- The “Why?” link

- The “Report issue” link
- Consolidation fixes

Duplicate code label

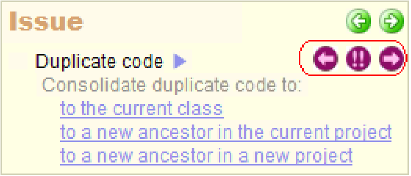
Hover over this label to see the duplicated code highlighted in the Visual Studio code editor:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{
  Bitmap bitmap = new Bitmap(image.Width, image.Height);
  using (Graphics g = Graphics.FromImage(bitmap))
  {
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttributes = new ImageAttributes())
    {
      imageAttributes.SetColorMatrix(colorMatrix, ColorMatrixFlag.Default, ColorAdjustType.Bitmap);
      g.DrawImage(image, new Rectangle(0, 0, bitmap.Width, bitmap.Height), 0, 0,
                  image.Width, image.Height, GraphicsUnit.Pixel, imageAttributes);
    }
  }
  return bitmap;
}
```

Issue
Duplicate code
Consolidate duplicate code to:
to the current class
to a new ancestor in the current project
to a new ancestor in a new project
to a new class in the current project
to a new class in a new project

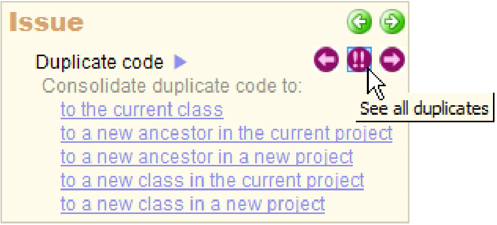
Navigation buttons



Issue
Duplicate code
Consolidate duplicate code to:
to the current class
to a new ancestor in the current project
to a new ancestor in a new project

The “Next duplicate block” and “Previous duplicate block” buttons allow you to navigate between duplicate code blocks of a single cluster, which represent two or more duplicate code blocks of a single piece of code that is duplicated. Clicking one of these two buttons will navigate to another duplicate code block, open a file if necessary and show the **Code Fix hint** for this block with a mouse cursor positioned on the “Next duplicate block” button for further navigation.

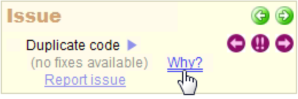
The center “See all duplicates” button opens the **Duplicate Code tool window**, where you can observe all duplicate code found in the entire solution:

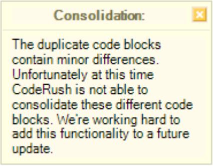


Issue
Duplicate code
Consolidate duplicate code to:
to the current class
to a new ancestor in the current project
to a new ancestor in a new project
to a new class in the current project
to a new class in a new project
See all duplicates

The “Why?” link

This link is shown when CodeRush is unable to offer an automated consolidation option. Consolidation of duplicate code is very challenging, and not all duplicates can be easily unified. Hover over this link to see why consolidation is not available, for example:

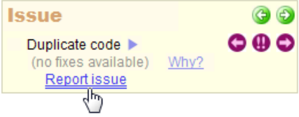




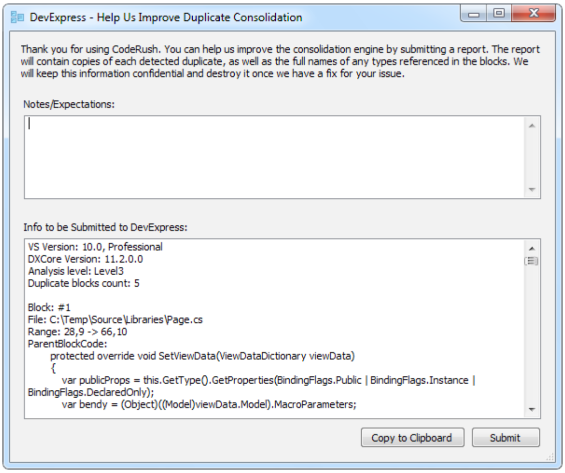
The duplicate code blocks contain minor differences. Unfortunately at this time CodeRush is not able to consolidate these different code blocks. We're working hard to add this functionality to a future update.

If you think CodeRush should be smart enough to consolidate this block of code and its clones automatically, click the “Report issue” link and provide your feedback.

The “Report issue” link



Clicking this link will bring up the dialog for sending feedback to help DevExpress improve Code Consolidation when it is not available:



Thank you for using CodeRush. You can help us improve the consolidation engine by submitting a report. The report will contain copies of each detected duplicate, as well as the full names of any types referenced in the blocks. We will keep this information confidential and destroy it once we have a fix for your issue.

Notes/Expectations:

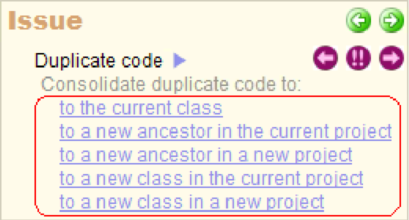
Info to be Submitted to DevExpress:

```
VS Version: 10.0, Professional
DXCore Version: 11.2.0.0
Analysis level: Level3
Duplicate blocks count: 5

Block: #1
File: C:\Temp\Source\Libraries\Page.cs
Range: 28,9 -> 66,10
ParentBlockCode:
      protected override void SetViewData(ViewDataDictionary viewData)
      {
         var publicProps = this.GetType().GetProperties(BindingFlags.Public | BindingFlags.Instance |
BindingFlags.DeclaredOnly);
         var bendy = (Object)((Model)viewData.Model).MacroParameters;
```

The report will contain the duplicate code and a other details that may help better understand the challenge at hand.

Consolidation

If the duplicated code can be consolidated, you will see one or more target destinations where the code can be consolidated:



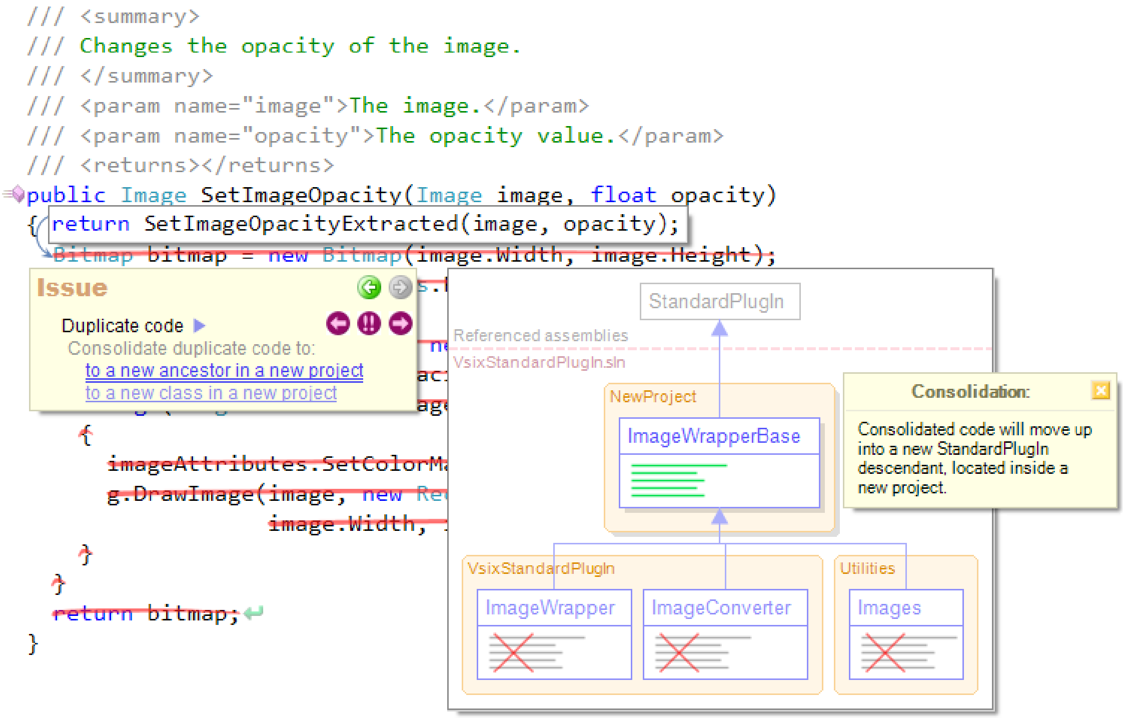
Issue

Duplicate code

Consolidate duplicate code to:

to the current class
to a new ancestor in the current project
to a new ancestor in a new project
to a new class in the current project
to a new class in a new project

Hover over each target consolidation link to see the preview hint of code editor changes and a visual diagram to review the duplicate code changes being made. In the consolidation hint, you can see how many duplicate code blocks exist and their structure, and how many classes and projects are involved in the duplication. The hint also shows the resulting target destination of the consolidated code. For example, consolidating to a new ancestor class in a new project:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{ return SetImageOpacityExtracted(image, opacity);
  ~~Bitmap bitmap = new Bitmap(image.Width, image.Height);~~
      ~~{~~
        ~~imageAttributes.SetColorM~~
        ~~g.DrawImage(image, new Re~~
              ~~image.Width,~~
      ~~}~~
  ~~}~~
  ~~return bitmap;~~
}
```

Issue

Duplicate code

Consolidate duplicate code to:

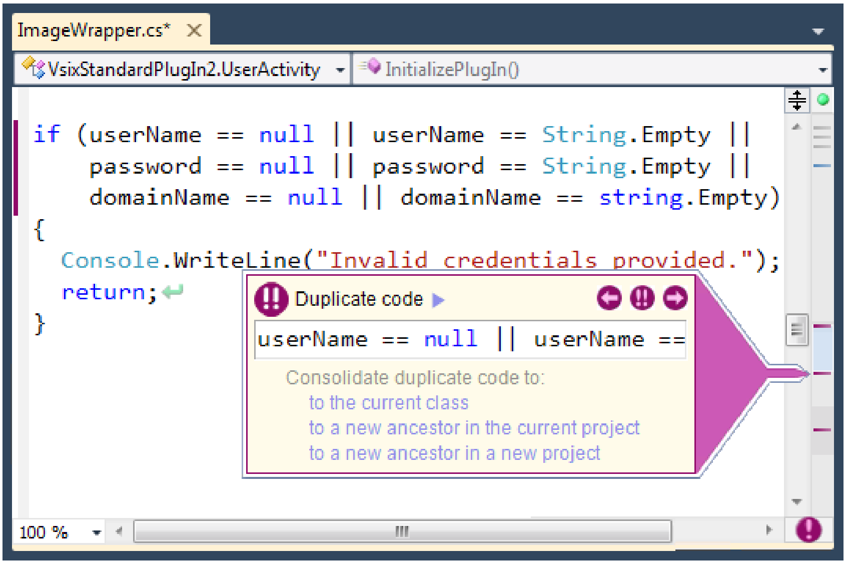
to a new ancestor in a new project
to a new class in a new project

Consolidation:

Consolidated code will move up into a new StandardPlugIn descendant, located inside a new project.

Click the link to apply the code consolidation at the selected location.

Hovering over the document bar will show you a similar hint with **Duplicate Code** issues:



```
if (userName == null || userName == String.Empty ||
    password == null || password == String.Empty ||
    domainName == null || domainName == string.Empty)
{
  Console.WriteLine("Invalid credentials provided.");
  return;
}
```

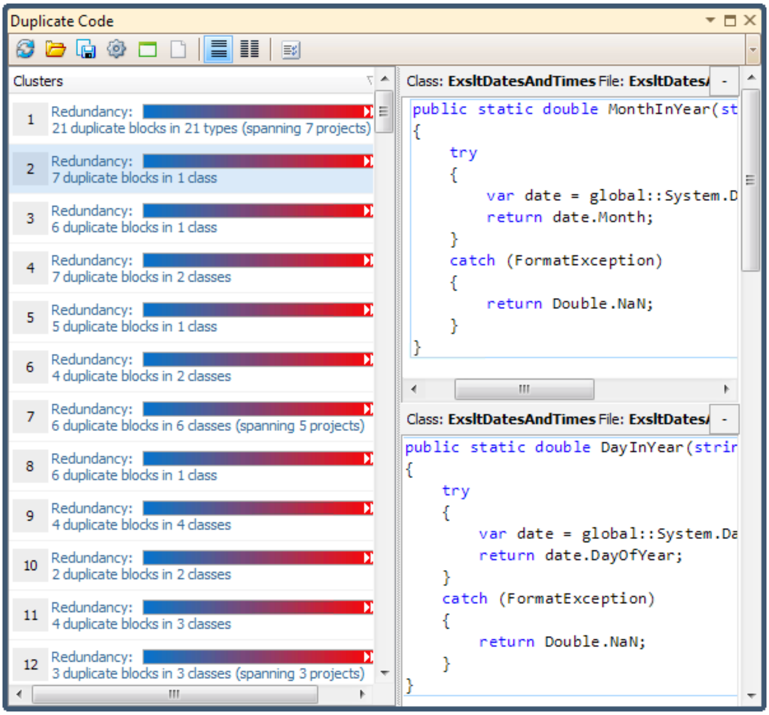
```
userName == null || userName ==
```

Consolidate duplicate code to:
- to the current class
- to a new ancestor in the current project
- to a new ancestor in a new project

Duplicate Detection Visual Studio tool window

The **Duplicate Code tool window** allows you to review the duplicate code found in your entire solution. The tool window is available via the **DevExpress** | **Tool Windows** | **Duplicate code** menu item.

If the duplicate code analysis has been started, you will see the analysis results in the Clusters list on the left, otherwise, you can click the “Run duplicate code analysis” button to perform the duplicate code search. This is what it looks like:



1	Redundancy: 21 duplicate blocks in 21 types (spanning 7 projects)
2	Redundancy: 7 duplicate blocks in 1 class
3	Redundancy: 6 duplicate blocks in 1 class
4	Redundancy: 7 duplicate blocks in 2 classes
5	Redundancy: 5 duplicate blocks in 1 class
6	Redundancy: 4 duplicate blocks in 2 classes
7	Redundancy: 6 duplicate blocks in 6 classes (spanning 5 projects)
8	Redundancy: 6 duplicate blocks in 1 class
9	Redundancy: 4 duplicate blocks in 4 classes
10	Redundancy: 2 duplicate blocks in 2 classes
11	Redundancy: 4 duplicate blocks in 3 classes
12	Redundancy: 3 duplicate blocks in 3 classes (spanning 3 projects)

Class: **ExsltDatesAndTimes** File: **ExsltDates/**

```
public static double MonthInYear(st
{
    try
    {
        var date = global::System.D
        return date.Month;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

Class: **ExsltDatesAndTimes** File: **ExsltDates/**

```
public static double DayInYear(strir
{
    try
    {
        var date = global::System.Da
        return date.DayOfYear;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

The tool bar contains the following buttons:

Icon	Name	Description
🔄	Run duplicate code analysis	.Starts the duplicate code analysis for the currently opened solution
📂	Open duplicate code analysis results	.Imports previously saved duplicate code analysis data
💾	Save duplicate code analysis results	.Exports the duplicate code analysis data for this solution for further investigation
⚙	Options	.Opens the Duplicate Code options page
🗔	Group by Projects	.Shows projects in the Cluster list and groups duplicate code clusters by them
📄	Group by Files	.Shows files in the Cluster list and groups duplicate code clusters by them
≣	Horizontal layout	.Changes the code view layout to horizontal
≣≣	Vertical layout	.Changes the code view layout to vertical
📋	Toggle log	.Toggles output window visibility where the log messages are sent

Clusters and code views

On the left, you can see the list of duplicate code clusters. A cluster consists of two or more functionally similar code blocks.

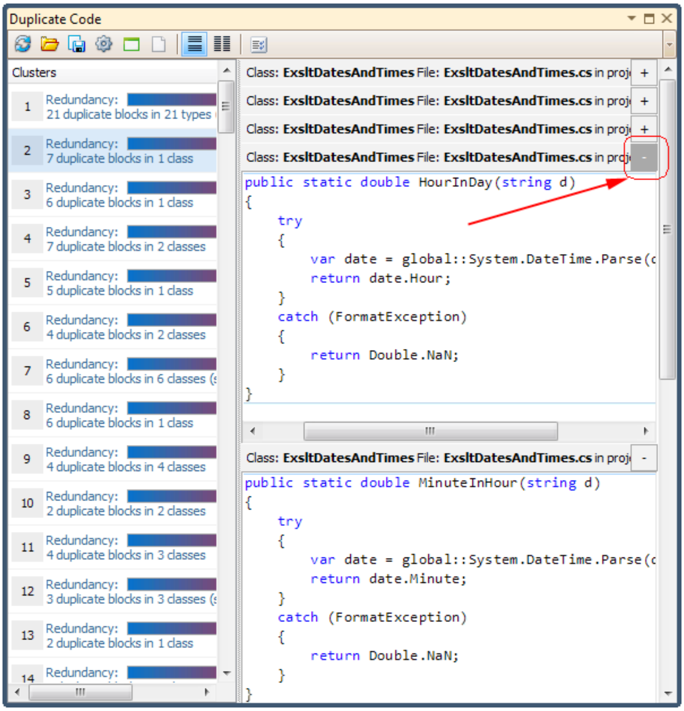
Clusters are sorted and indexed by the redundancy value in descending order. A redundancy is a measure of the redundant code based on its duplication. The greater the redundancy value, the more code that can be removed when the duplicated code is consolidated. Note that a cluster with a small duplicated code block may have a higher redundancy value than a cluster of a bigger duplicate code block because the smaller block has more duplications then the larger one.

Near the redundancy value, you can see how many duplicate code blocks are contained in each cluster and the number of classes these blocks reside in.

Selecting a cluster will display a number of duplicate code views on the right. Each code view has a header showing class name, file name and project where each duplicate code block is located. The source code for each duplicated code block appears below the header.

If you double click a cluster, the source code of the first duplicate code block will be opened in the Visual Studio code editor. You will see the Code Fix hint shown for the opened code block, and the mouse cursor will be positioned on the “Next duplicate block” button which allows you to navigate between duplicate code blocks.

Code views can be arranged horizontally or vertically – using the orientation buttons on the toolbar to change the layout. In the horizontal layout, a code view can be collapsed by clicking the [-] button:



1 Redundancy: 21 duplicate blocks in 21 types
2 Redundancy: 7 duplicate blocks in 1 class
3 Redundancy: 6 duplicate blocks in 1 class
4 Redundancy: 7 duplicate blocks in 2 classes
5 Redundancy: 5 duplicate blocks in 1 class
6 Redundancy: 4 duplicate blocks in 2 classes
7 Redundancy: 6 duplicate blocks in 6 classes (
8 Redundancy: 6 duplicate blocks in 1 class
9 Redundancy: 4 duplicate blocks in 4 classes
10 Redundancy: 2 duplicate blocks in 2 classes
11 Redundancy: 4 duplicate blocks in 3 classes
12 Redundancy: 3 duplicate blocks in 3 classes (
13 Redundancy: 2 duplicate blocks in 1 class
14 Redundancy:

Class: **ExsltDatesAndTimes** File: **ExsltDatesAndTimes.cs** in proj +

Class: **ExsltDatesAndTimes** File: **ExsltDatesAndTimes.cs** in proj +

Class: **ExsltDatesAndTimes** File: **ExsltDatesAndTimes.cs** in proj +

Class: **ExsltDatesAndTimes** File: **ExsltDatesAndTimes.cs** in proj -

```
public static double HourInDay(string d)
{
    try
    {
        var date = global::System.DateTime.Parse(c
        return date.Hour;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

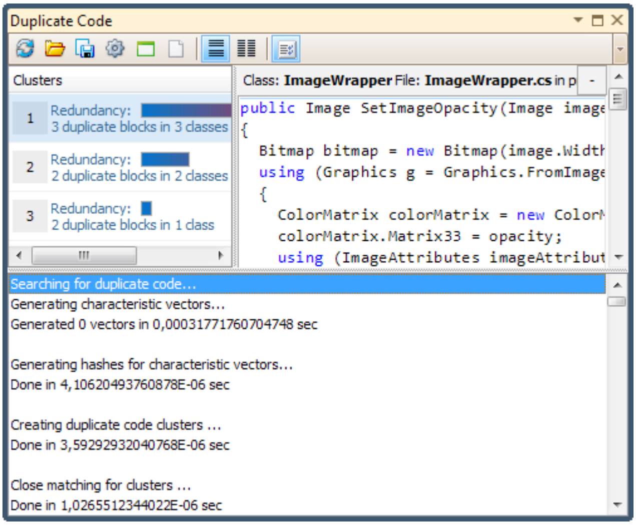
Class: **ExsltDatesAndTimes** File: **ExsltDatesAndTimes.cs** in proj -

```
public static double MinuteInHour(string d)
{
    try
    {
        var date = global::System.DateTime.Parse(c
        return date.Minute;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

and then expanded by clicking the [+] button. Inside a code view, a light blue rectangle highlights the duplicated code.

Output

This part of the window contains messages logged about the current analysis progress and operations performed and the time each operation takes:



1 Redundancy: 3 duplicate blocks in 3 classes
2 Redundancy: 2 duplicate blocks in 2 classes
3 Redundancy: 2 duplicate blocks in 1 class

Class: **ImageWrapper** File: **ImageWrapper.cs** in p -

```
public Image SetImageOpacity(Image image
{
  Bitmap bitmap = new Bitmap(image.Width
  using (Graphics g = Graphics.FromImage
  {
    ColorMatrix colorMatrix = new ColorM
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttribut
```

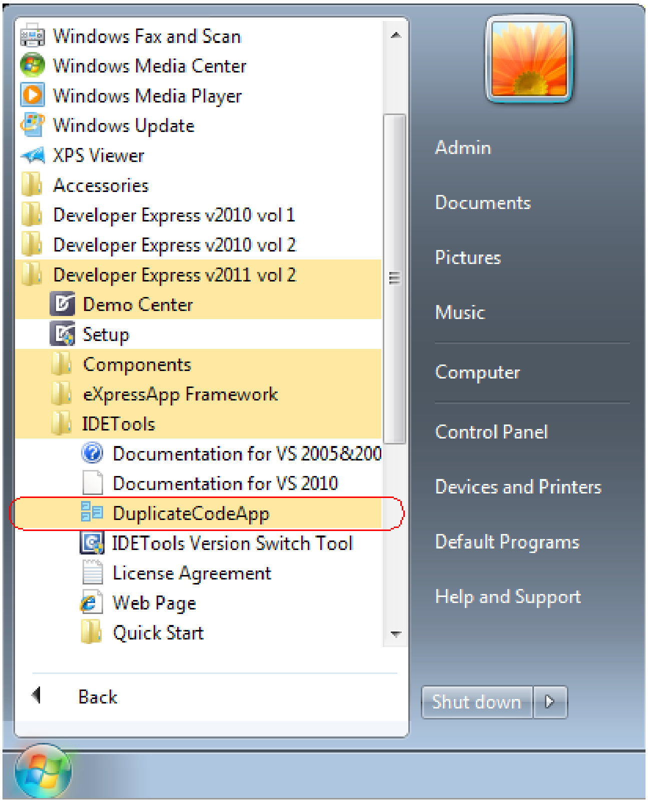
```
Searching for duplicate code...
Generating characteristic vectors...
Generated 0 vectors in 0,00031771760704748 sec

Generating hashes for characteristic vectors...
Done in 4,10620493760878E-06 sec

Creating duplicate code clusters ...
Done in 3,59292932040768E-06 sec

Close matching for clusters ...
Done in 1,0265512344022E-06 sec
```

Duplicate Detection using the Stand-alone Application

CodeRush ships two standalone programs that do not require the Visual Studio IDE to be running for duplicate code detection analysis:

- **Duplicate Code Analysis windows app**
- **Duplicate Code Analysis console app**

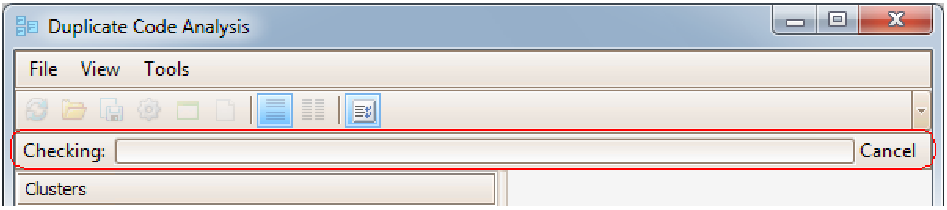
Both applications can be used outside of the IDE to find **duplicate code** inside a specified solution. These applications are located inside the **Plug-ins** folder of your IDE Tools installation. You can find the windows application in the Start menu as well:



DDC windows app

This windows app is similar to the DDC Visual Studio tool window. The window has the same user interface and allows you to review duplicated code discovered in your solution. To start the stand-alone duplicate detection application, from the Windows Start menu select Developer Express v2011 vol 2 | IDETools | DuplicateCodeApp. Once the application is running, you can scan a solution by selecting **File** | **Analyze Solution** or by clicking the corresponding toolbar item.

Once the solution is opened, you will see a progress bar indicating the analysis progress:



You can toggle the Output view in the window, to see more detail on the operations performed and how much time they take.

The application contains several elements:

- Main Menu
- Tool bar
- Clusters list
- Code views
- Output

Main Menu

- **File**

  ○ Analyze Solution – shows the Open File dialog so you can select a solution to analyze. Once a file is selected, analysis begins immediately.

  ○ Import – opens previously saved duplicate code analysis data for review.

  ○ Export – saves duplicate code analysis data for further investigation.

  ○ Exit – closes the application.

- **View**

  ○ Clear – clears the duplicate code analysis results in the clusters tree list and all source code views.

- **Tools**

  ○ Options – opens the Options dialog, which allows you to specify the code analysis settings, such as an analysis level.

Tool bar

The tool bar contains all important tool buttons for the duplicate code detection analysis:

Icon	Name	Description
🔄	Analyze Solution	.Shows the Open File dialog allowing you to choose a solution file for analysis
📂	Import	.Loads previously saved duplicate code analysis data

Export	Saves the duplicate code analysis data for this solution.
Options	Shows the Options dialog with available duplicate code analysis settings.
Show Projects	Displays projects in the cluster tree list and groups clusters within each project. Clusters that span across more than one project will appear multiple times in this view.
Show Files	Displays files in the cluster tree list and groups clusters within each file. Clusters that span across more than one file will appear multiple times in this view.
Horizontal layout	Changes the code view layout, arranging duplicates horizontally.
Vertical layout	Changes the code view layout, arranging duplicates vertically.
Toggle log	Toggles the visibility of the output window where analysis log messages are sent.

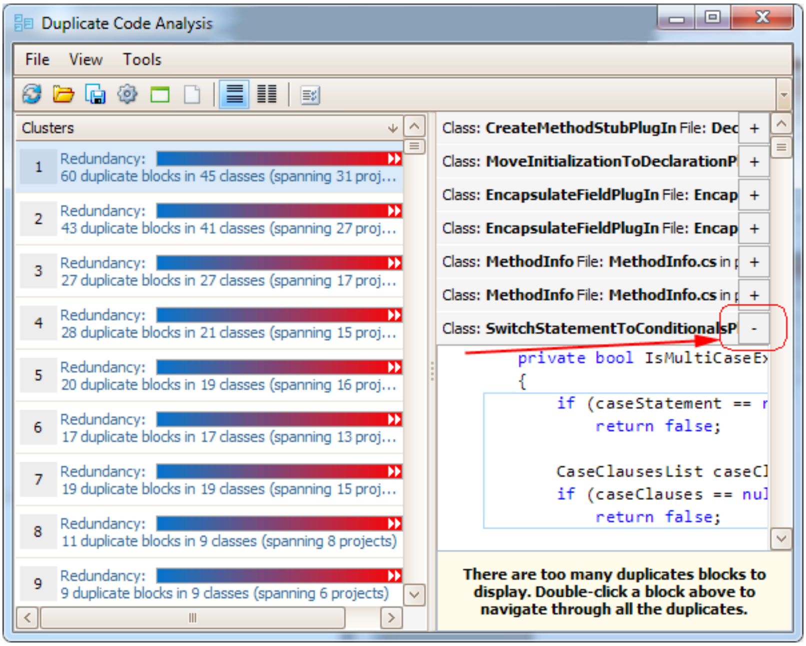
Clusters and code views

On the left, you can see the list of duplicate code clusters. This is the cluster tree list. A cluster represents two or more blocks of source code that are duplicated within a solution.

A cluster in the list has an ordering number the redundancy value. A redundancy is a measure of the amount of code that would be removed if all duplicates were consolidated. All clusters in the list are sorted by the redundancy value in descending order. Near the redundancy value, you can see how many duplicate code blocks are in a cluster and the count of classes these blocks reside in.

Selecting a cluster will reveal the corresponding duplicate code on the right. Each code view has a caption that shows a class name, a file name and a project where a duplicate code block is located, and a view of the source code for each duplicated code block.

Code views can be arranged horizontally or vertically – toggle the appropriate button on the toolbar to change the layout. In the horizontal layout, code views can be collapsed by clicking the [-] button:



1 Redundancy: 60 duplicate blocks in 45 classes (spanning 31 proj…
2 Redundancy: 43 duplicate blocks in 41 classes (spanning 27 proj…
3 Redundancy: 27 duplicate blocks in 27 classes (spanning 17 proj…
4 Redundancy: 28 duplicate blocks in 21 classes (spanning 15 proj…
5 Redundancy: 20 duplicate blocks in 19 classes (spanning 16 proj…
6 Redundancy: 17 duplicate blocks in 17 classes (spanning 13 proj…
7 Redundancy: 19 duplicate blocks in 19 classes (spanning 15 proj…
8 Redundancy: 11 duplicate blocks in 9 classes (spanning 8 projects)
9 Redundancy: 9 duplicate blocks in 9 classes (spanning 6 projects)

Class: **CreateMethodStubPlugIn** File: **Dec** +
Class: **MoveInitializationToDeclarationP** +
Class: **EncapsulateFieldPlugIn** File: **Encap** +
Class: **EncapsulateFieldPlugIn** File: **Encap** +
Class: **MethodInfo** File: **MethodInfo.cs** in p +
Class: **MethodInfo** File: **MethodInfo.cs** in p +
Class: **SwitchStatementToConditionalsP** -

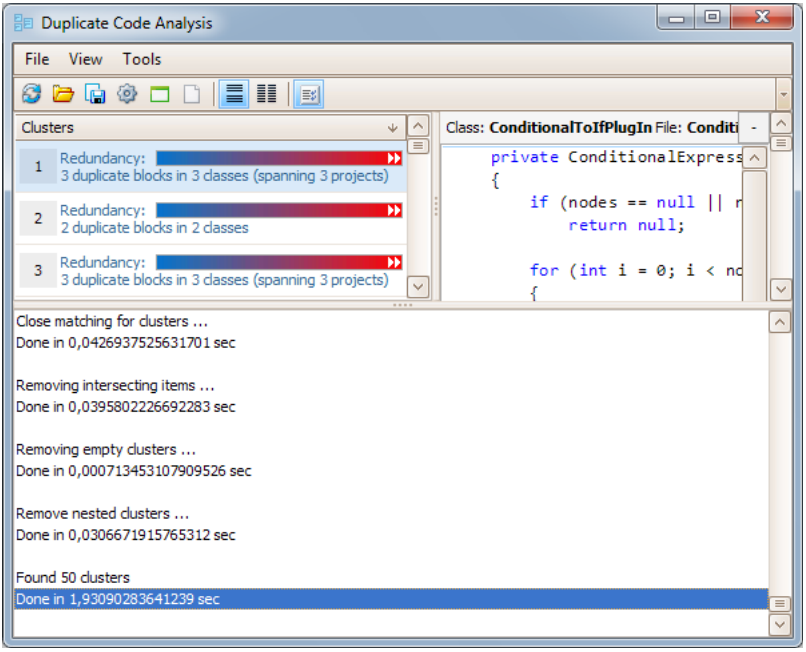
```
private bool IsMultiCaseE›
{
    if (caseStatement == r
        return false;

    CaseClausesList caseCl
    if (caseClauses == nul
        return false;
```

There are too many duplicates blocks to display. Double-click a block above to navigate through all the duplicates.

and expanded by clicking the [+] button. Inside a code view, a rectangle shows the code block that is duplicated.

Output

This part of the window contains messages logged about the current analysis progress and operations performed and the time each operation takes:



1 Redundancy:
3 duplicate blocks in 3 classes (spanning 3 projects)

2 Redundancy:
2 duplicate blocks in 2 classes

3 Redundancy:
3 duplicate blocks in 3 classes (spanning 3 projects)

```
    private ConditionalExpress
    {
        if (nodes == null || r
            return null;

        for (int i = 0; i < nd
        {
```

```
Close matching for clusters ...
Done in 0,0426937525631701 sec

Removing intersecting items ...
Done in 0,0395802226692283 sec

Removing empty clusters ...
Done in 0,000713453107909526 sec

Remove nested clusters ...
Done in 0,0306671915765312 sec

Found 50 clusters
Done in 1,93090283641239 sec
```

There are two versions of this application for different .NET Framework versions: 2.0 and 4.0.

Console app

The console application is used within a command-line window and provides no visual UI. You can specify different behavior by passing arguments in a command-line window. Here is the syntax:

```
DuplicateCodeConsole.exe solutionPath outputFile
```

where "*solutionPath*" is a path to the solution for duplicate code detection analysis and "*outputFile*" is a path to the output file that will contain the resulting analysis data. This file can be opened in either the stand-alone Windows application or in the Duplicate Code tool window.

After valid paths are provided, the application parses the solution and searches for the duplicate code, reporting progress:

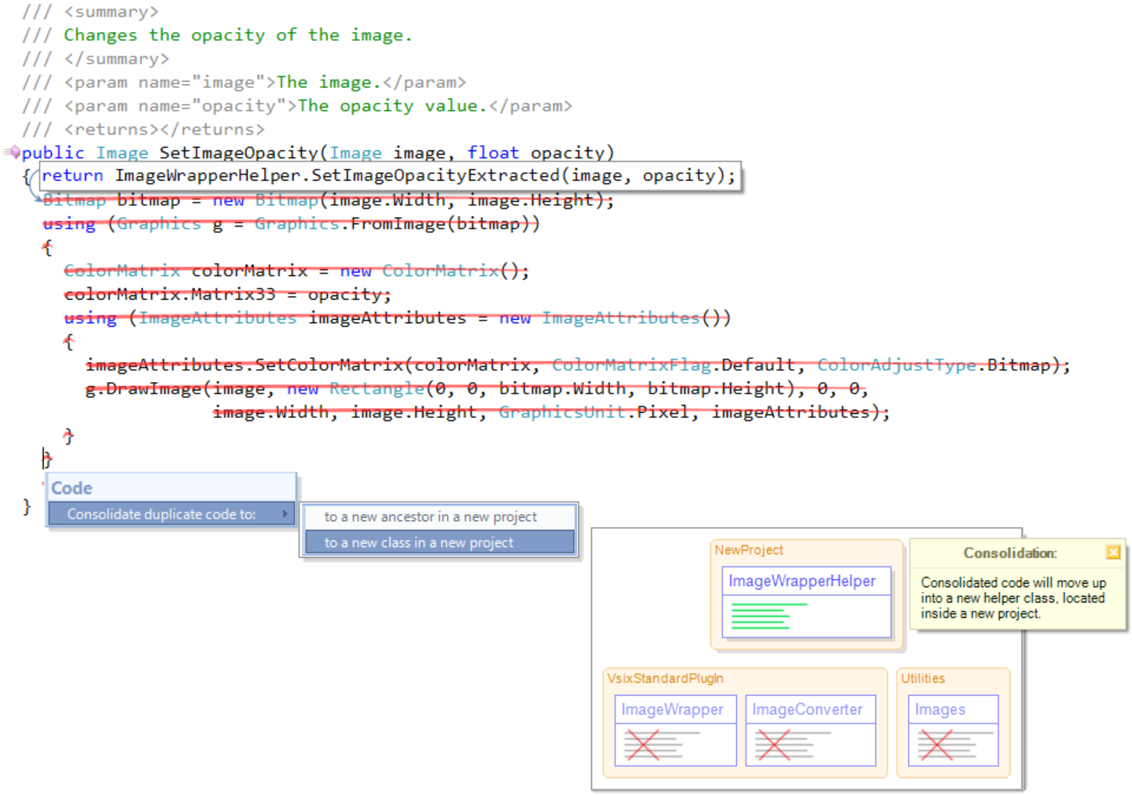
```
Parsing solution... Done in 55,2813345781978 sec. Searching for dupli-
cate code... Generating characteristic vectors... Generated 9050 vec-
tors in 0,0395699571568842 sec Removing empty clusters ... Done in
0,000371098271236394 sec Found 20 clusters Done in 0,785516491288943 sec
```

After the analysis is done, the resulting analysis data is saved to the file you specified in the command line in an XML format. You can open this analysis data in the Duplicate Code Windows application or in Duplicate Code tool window inside Visual Studio.

Duplicate Consolidation inside Visual Studio

The process of removing duplicate code is called **code consolidation**. The result of code consolidation is a single code block that replaces a specific duplicated code in several locations (different files or projects). The resulting consolidated code block is functionally absolutely the same as all duplicated code blocks it replaces.

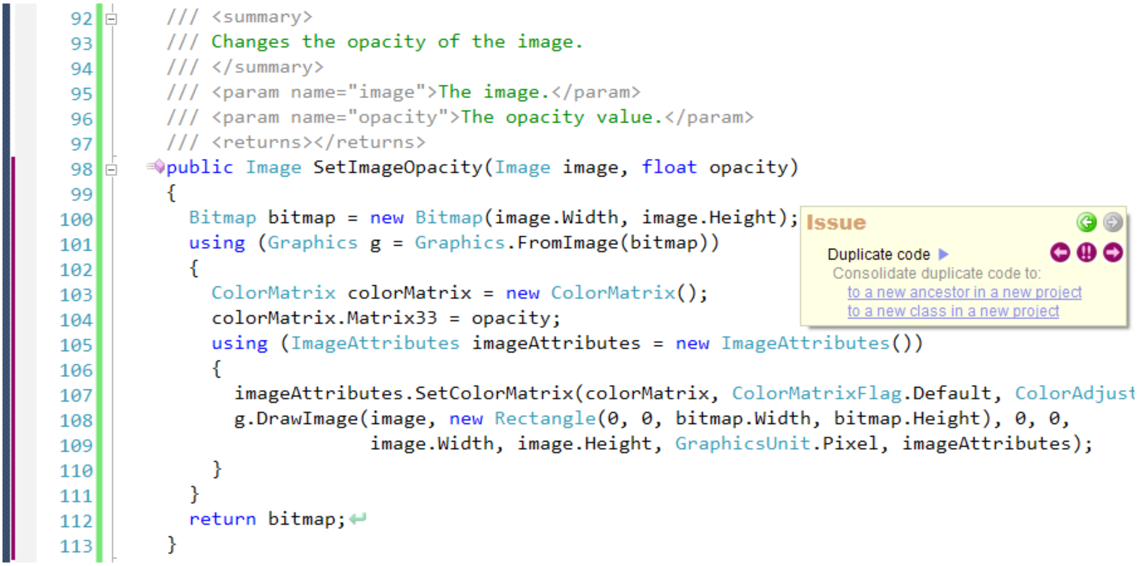
To apply the code consolidation, you can use the **Consolidate Code** provider. It is available when the editor caret is positioned at a Duplicate Code issue via the **CodeRush** popup menu:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{ return ImageWrapperHelper.SetImageOpacityExtracted(image, opacity);
  ~~Bitmap bitmap = new Bitmap(image.Width, image.Height);~~
  ~~using (Graphics g = Graphics.FromImage(bitmap))~~
  ~~{~~
    ~~ColorMatrix colorMatrix = new ColorMatrix();~~
    ~~colorMatrix.Matrix33 = opacity;~~
    ~~using (ImageAttributes imageAttributes = new ImageAttributes())~~
    ~~{~~
      ~~imageAttributes.SetColorMatrix(colorMatrix, ColorMatrixFlag.Default, ColorAdjustType.Bitmap);~~
      ~~g.DrawImage(image, new Rectangle(0, 0, bitmap.Width, bitmap.Height), 0, 0,~~
                  ~~image.Width, image.Height, GraphicsUnit.Pixel, imageAttributes);~~
    ~~}~~
  ~~}~~
}
```

Consolidated code will move up into a new helper class, located inside a new project.

or through the Code Issues visual hints, such as **Code Fix** hint:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{
  Bitmap bitmap = new Bitmap(image.Width, image.Height);
  using (Graphics g = Graphics.FromImage(bitmap))
  {
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttributes = new ImageAttributes())
    {
      imageAttributes.SetColorMatrix(colorMatrix, ColorMatrixFlag.Default, ColorAdjust
      g.DrawImage(image, new Rectangle(0, 0, bitmap.Width, bitmap.Height), 0, 0,
                  image.Width, image.Height, GraphicsUnit.Pixel, imageAttributes);
    }
  }
  return bitmap;
}
```

Issue

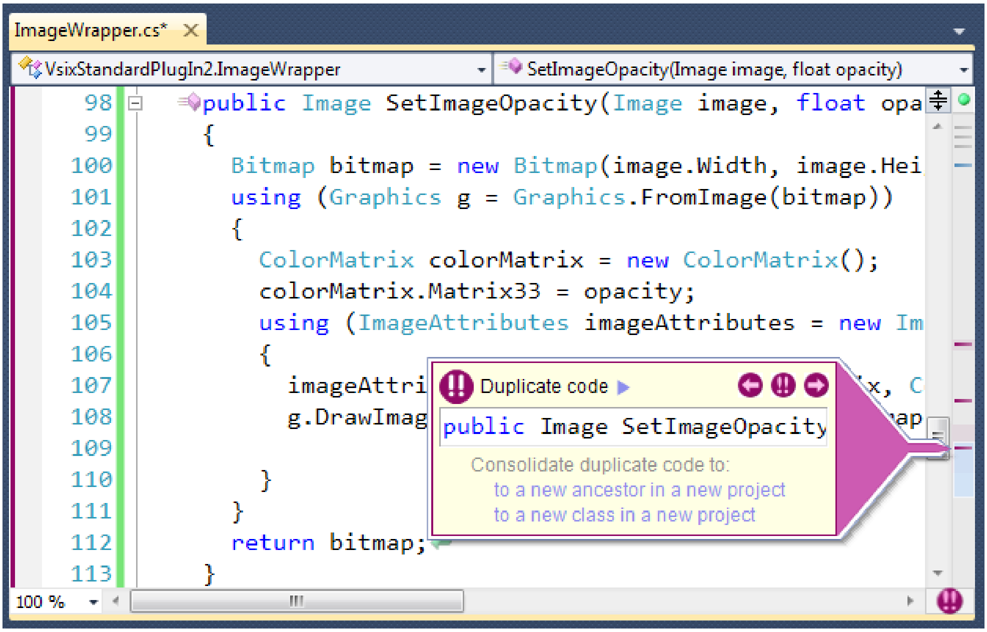
Duplicate code

Consolidate duplicate code to:

to a new ancestor in a new project

to a new class in a new project

and the document bar hint:



ImageWrapper.cs*
VsixStandardPlugIn2.ImageWrapper
SetImageOpacity(Image image, float opacity)
100 %

```
public Image SetImageOpacity(Image image, float opa
{
  Bitmap bitmap = new Bitmap(image.Width, image.Hei
  using (Graphics g = Graphics.FromImage(bitmap))
  {
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttributes = new Im
    {
      imageAttri
      g.DrawImag

    }
  }
  return bitmap;
}
```

Duplicate code

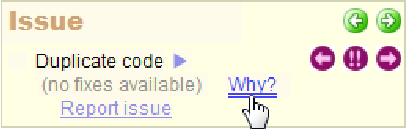
public Image SetImageOpacity

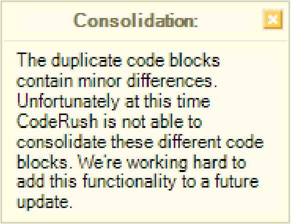
Consolidate duplicate code to:

to a new ancestor in a new project

to a new class in a new project

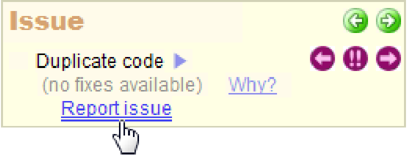
If the **Consolidate Code** provider is not available, you can see why it is not available in the **Code Fix** hint by hovering over the “Why?” link:



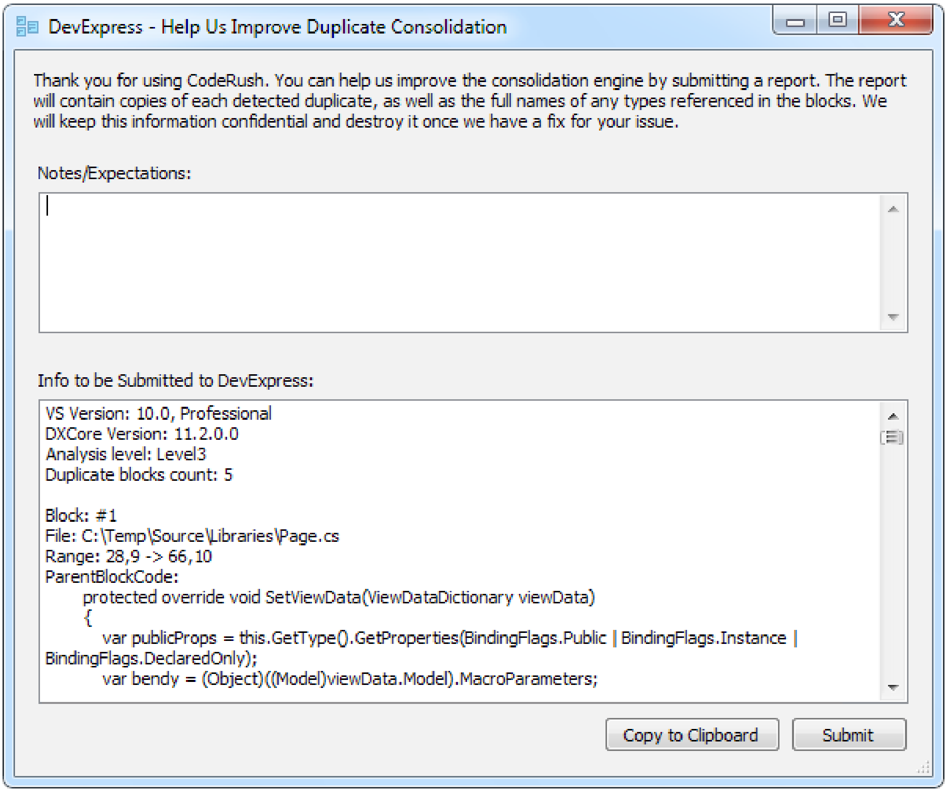


The duplicate code blocks contain minor differences. Unfortunately at this time CodeRush is not able to consolidate these different code blocks. We're working hard to add this functionality to a future update.

If you think that duplicated code can be consolidated, click the “Report Issue” link to provide feedback about your expectations:



Once clicked, a special dialog appear for submitting feedback:



Thank you for using CodeRush. You can help us improve the consolidation engine by submitting a report. The report will contain copies of each detected duplicate, as well as the full names of any types referenced in the blocks. We will keep this information confidential and destroy it once we have a fix for your issue.

Notes/Expectations:

Info to be Submitted to DevExpress:

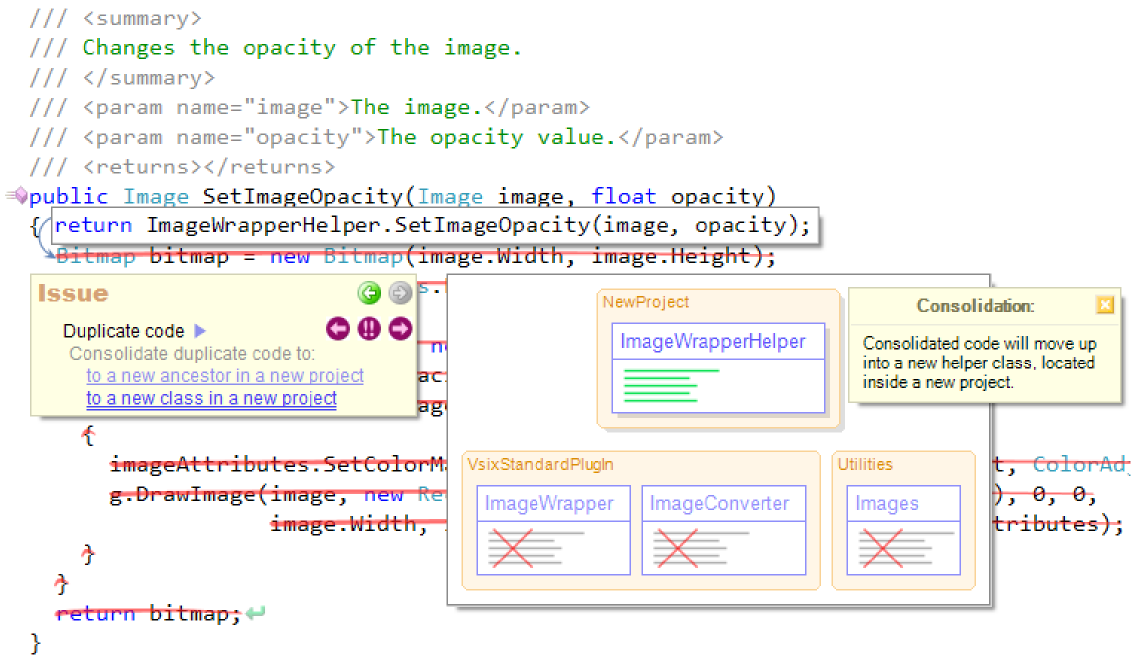
```
VS Version: 10.0, Professional
DXCore Version: 11.2.0.0
Analysis level: Level3
Duplicate blocks count: 5

Block: #1
File: C:\Temp\Source\Libraries\Page.cs
Range: 28,9 -> 66,10
ParentBlockCode:
      protected override void SetViewData(ViewDataDictionary viewData)
      {
         var publicProps = this.GetType().GetProperties(BindingFlags.Public | BindingFlags.Instance |
BindingFlags.DeclaredOnly);
         var bendy = (Object)((Model)viewData.Model).MacroParameters;
```

Consolidation locations

Code consolidation allows you to consolidate code at different locations. The code consolidation will suggest only an appropriate code location, such as:

- **the current class** – if duplicate code blocks are residing in a single type
- **the base class** – if duplicate code blocks are residing in a single class hierarchy
- **a new ancestor class** – if duplicate code blocks are residing in types that can be inherited from a new ancestor
- **the helper class** – if duplicate code blocks are residing in different classes that do not have a common ancestor and a new ancestor cannot be created
- **the existing class**, where previous consolidation has been performed

You may also have different project location options: for the 'helper class' and 'new ancestor class' locations, it is possible to choose whether to use the current project (if duplicate code blocks are residing in a single project) or create a new one (if duplicate code blocks are residing in different projects), so source projects will reference a newly created project:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{ return ImageWrapperHelper.SetImageOpacity(image, opacity);
  ~~Bitmap bitmap = new Bitmap(image.Width, image.Height);~~
      {
        ~~imageAttributes.SetColorM~~                     ~~t, ColorAd~~
        ~~g.DrawImage(image, new Re~~                     ~~), 0, 0,~~
                ~~image.Width,~~                          ~~tributes);~~
      }
   }
  ~~return bitmap;~~
}
```

Issue

Duplicate code

Consolidate duplicate code to:

to a new ancestor in a new project

to a new class in a new project

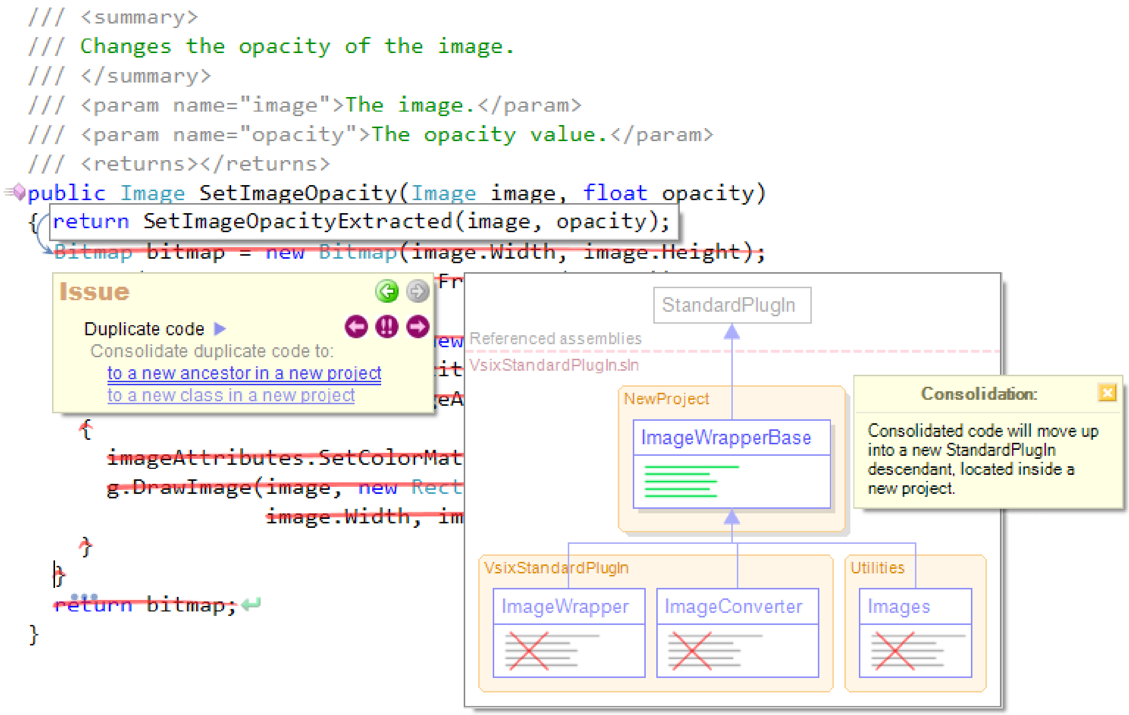
Consolidation:

Consolidated code will move up into a new helper class, located inside a new project.

If duplicate code blocks are residing in different projects and those projects reference other common projects, you can choose one of the referenced projects to create a helper class in it and move the consolidated code there. Once you have consolidated any duplicated code in a particular class (inside the current, new, or referenced project), you can choose that class as a target location when consolidating the next duplicated code.

Preview hint

As you have already seen, hovering over the target location for consolidation in the **Code Fix** hint or **CodeRush Pop-up** menu, will show you a preview hint with the changes being made to the current code and the visual diagram of the duplicated and consolidated code hierarchy:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{ return SetImageOpacityExtracted(image, opacity);
  ~~Bitmap bitmap = new Bitmap(image.Width, image.Height);~~
                                  Fr
                                  ew
                                  it
                                  eA
    {
      ~~imageAttributes.SetColorMat~~
      ~~g.DrawImage(image, new Rect~~
                 ~~image.Width, im~~
    }
  }
  ~~return bitmap;~~
}
```

Issue

Duplicate code

Consolidate duplicate code to:

to a new ancestor in a new project

to a new class in a new project

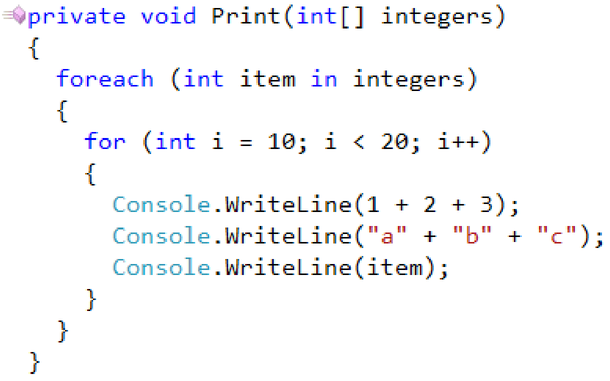
Consolidation:

Consolidated code will move up into a new StandardPlugIn descendant, located inside a new project.

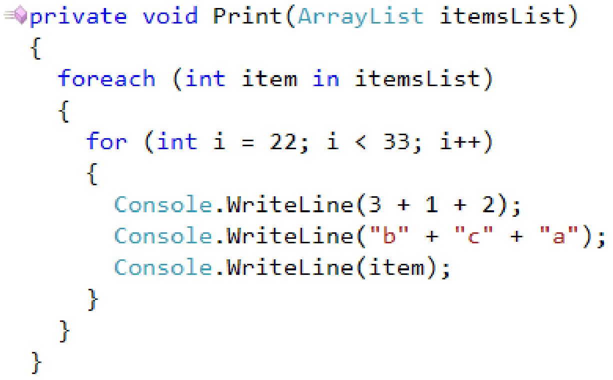
Depending on the duplicate code block count, their location (classes and projects) and target location for the consolidated code, the hint will reflect and correspond to these parameters.

Consolidation in action

For example, a duplicated code may vary by different values used. Consider the following simple example of the duplicated code – it is not real, but it may show how consolidation works here:

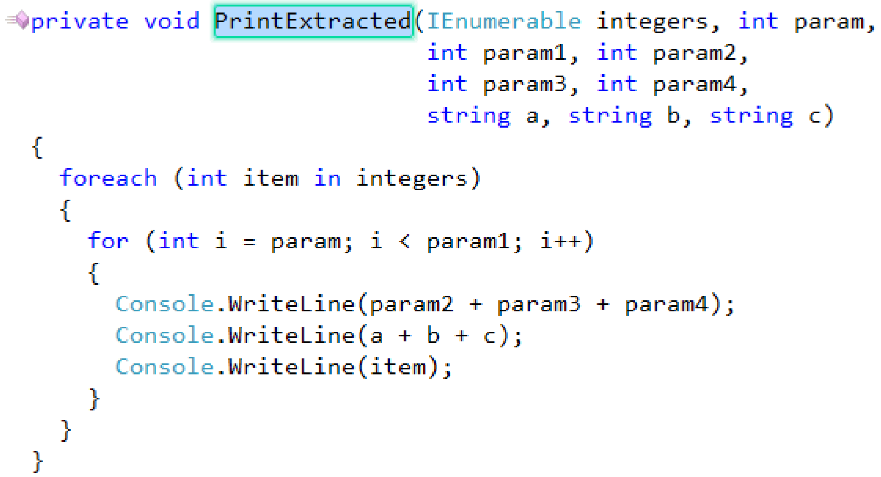


```
private void Print(int[] integers)
{
  foreach (int item in integers)
  {
    for (int i = 10; i < 20; i++)
    {
      Console.WriteLine(1 + 2 + 3);
      Console.WriteLine("a" + "b" + "c");
      Console.WriteLine(item);
    }
  }
}
```



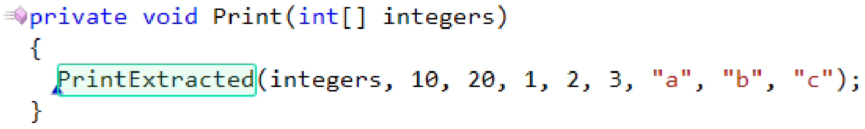
```
private void Print(ArrayList itemsList)
{
  foreach (int item in itemsList)
  {
    for (int i = 22; i < 33; i++)
    {
      Console.WriteLine(3 + 1 + 2);
      Console.WriteLine("b" + "c" + "a");
      Console.WriteLine(item);
    }
  }
}
```

These two methods do the same thing: inside the ‘for’ and ‘foreach’ loops it writes several lines with certain values to the Console. However, note the differences in the values and their different positions inside the code, e.g. passing “1 + 2 + 3″ and “2 + 3 + 1″ as arguments. Consolidating such code will require replacing all differences in values to parameters and pass the correct values as arguments to the resulting method. Also, because the parameters of two duplicate code blocks have different types they should be consolidated to a common type, if possible. In this specific case, the common type for both parameters will be the *System.IEnumerable* type. Here’s the resulting consolidated code:

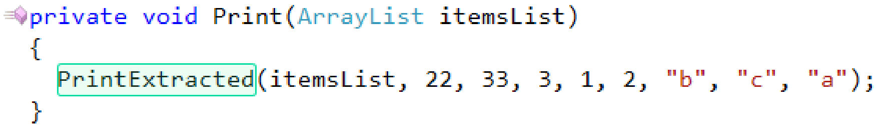


```
private void PrintExtracted(IEnumerable integers, int param,
                            int param1, int param2,
                            int param3, int param4,
                            string a, string b, string c)
{
  foreach (int item in integers)
  {
    for (int i = param; i < param1; i++)
    {
      Console.WriteLine(param2 + param3 + param4);
      Console.WriteLine(a + b + c);
      Console.WriteLine(item);
    }
  }
}
```

and the calling sides:



```
private void Print(int[] integers)
{
  PrintExtracted(integers, 10, 20, 1, 2, 3, "a", "b", "c");
}
```

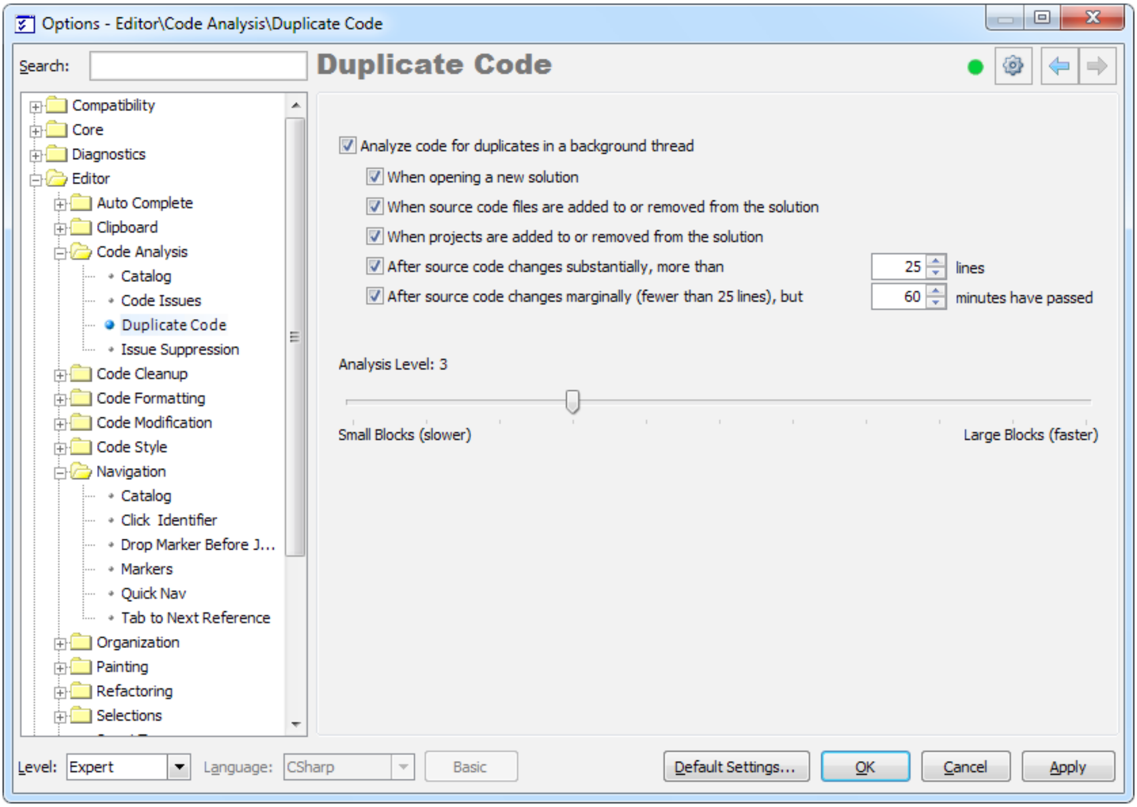


```
private void Print(ArrayList itemsList)
{
  PrintExtracted(itemsList, 22, 33, 3, 1, 2, "b", "c", "a");
}
```

This is a just simple example with local differences, but there can be also other things involved in code duplicates, such as different member references (dependencies), different types of parameters and fields, different project boundaries, etc. So, the consolidation task is rather really complex, with lots of edge cases. The **CodeRush Consolidation Engine** can deal with all of these differences and create a single code that functionally is the same as all replaced duplicated code blocks. If the code consolidation is not yet capable of removing a specific duplicated code, you can always send feedback to DevExpress by clicking the "Report issue" link, so the Consolidation Engine will be improved in the future.

Duplicate Detection analysis options

Duplicate Code Detection options are available at the **Editor** | **Code Analysis** | **Duplicate Code** options page in the Options Dialog. Duplicate code options specify when detection analysis is activated and the analysis level. Here's what this page looks like:



Options - Editor\Code Analysis\Duplicate Code

Duplicate Code

☑ Analyze code for duplicates in a background thread

☑ When opening a new solution

☑ When source code files are added to or removed from the solution

☑ When projects are added to or removed from the solution

☑ After source code changes substantially, more than 25 lines

☑ After source code changes marginally (fewer than 25 lines), but 60 minutes have passed

Analysis Level: 3

Small Blocks (slower) — Large Blocks (faster)

Level: Expert

Language: CSharp

Available options are:

[X] Analyze code for duplicates in a background thread

Specifies whether or not duplicate code detection analysis is enabled and available.

[X] When opening a new solution

Specifies whether or not duplicate code detection analysis should be activated on opening a solution.

[X] When source code files are added to or removed from the solution

Specifies whether or not duplicate code detection analysis should be activated on adding source code files or removing them to/from the current solution.

[X] When projects are added to or removed from the solution

Specifies whether or not duplicate code detection analysis should be activated on adding projects or removing them to/from the current solution.

[X] After source code changes substantially, more than [ ] lines

Specifies whether or not duplicate code detection analysis should be activated when the specified number of source code lines have been changed.

[X] After source code changes marginally (fewer than 25 lines), but [ ] minutes have passed

Specifies whether or not duplicate code detection analysis should be activated when a small number of source code lines (less then 25) have been changed after the specified amount of time.

Analysis Level

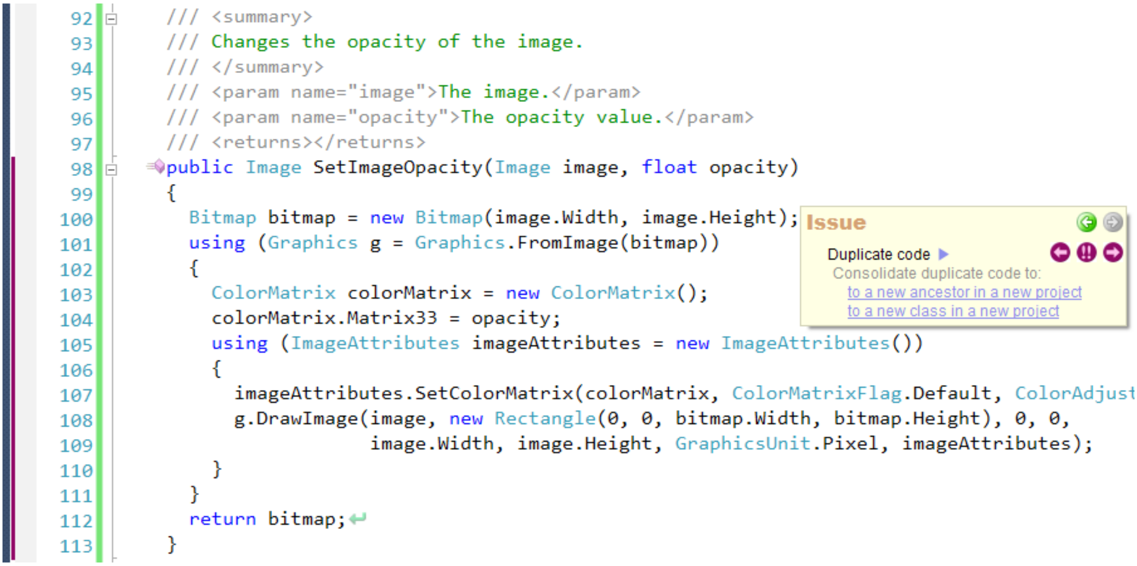
Specifies the analysis level of the similarity of duplicate code blocks that must be found: whether small or large blocks. Note that low level might require significant CPU power and memory for duplicate code detection analysis.

Duplicate Detection in CodeRush versus Code Clone Analysis in Visual Studio 2012

CodeRush Duplicate Code Detection (**DDC**) analysis runs in the background while you work in the Visual Studio IDE. Visual Studio 2011 Developer Preview has a similar feature called **Code Clone Analysis** (**CCA**). Both features find code duplications, so it is possible to undertake a brief overview of each feature and compare them. Please note that this article was written using the Visual Studio 11 Developer Preview, and as such, certain functionality may be subject to change before the final release. The version of the IDE Tools/CodeRush used for comparison is ١١،٢, which is certainly subject to change.

UI & Integration

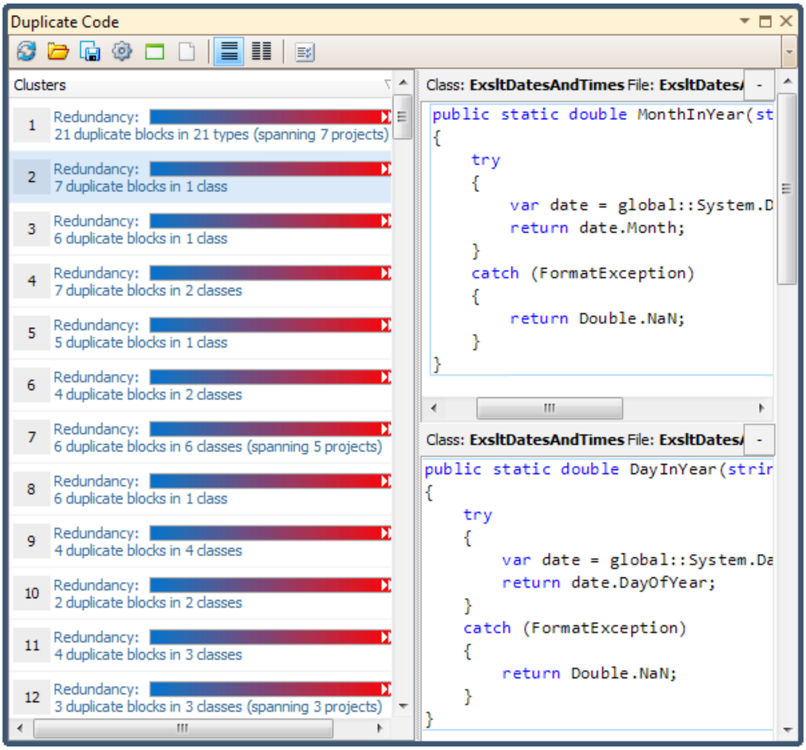
CodeRush DDC highlights code duplicates right inside the code editor:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{
  Bitmap bitmap = new Bitmap(image.Width, image.Height);
  using (Graphics g = Graphics.FromImage(bitmap))
  {
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.Matrix33 = opacity;
    using (ImageAttributes imageAttributes = new ImageAttributes())
    {
      imageAttributes.SetColorMatrix(colorMatrix, ColorMatrixFlag.Default, ColorAdjust
      g.DrawImage(image, new Rectangle(0, 0, bitmap.Width, bitmap.Height), 0, 0,
                  image.Width, image.Height, GraphicsUnit.Pixel, imageAttributes);
    }
  }
  return bitmap;
}
```

Issue
Duplicate code
Consolidate duplicate code to:
to a new ancestor in a new project
to a new class in a new project

It also has a special Duplicate Code tool window that allows you to review all code duplicates:



1 Redundancy: 21 duplicate blocks in 21 types (spanning 7 projects)
2 Redundancy: 7 duplicate blocks in 1 class
3 Redundancy: 6 duplicate blocks in 1 class
4 Redundancy: 7 duplicate blocks in 2 classes
5 Redundancy: 5 duplicate blocks in 1 class
6 Redundancy: 4 duplicate blocks in 2 classes
7 Redundancy: 6 duplicate blocks in 6 classes (spanning 5 projects)
8 Redundancy: 6 duplicate blocks in 1 class
9 Redundancy: 4 duplicate blocks in 4 classes
10 Redundancy: 2 duplicate blocks in 2 classes
11 Redundancy: 4 duplicate blocks in 3 classes
12 Redundancy: 3 duplicate blocks in 3 classes (spanning 3 projects)

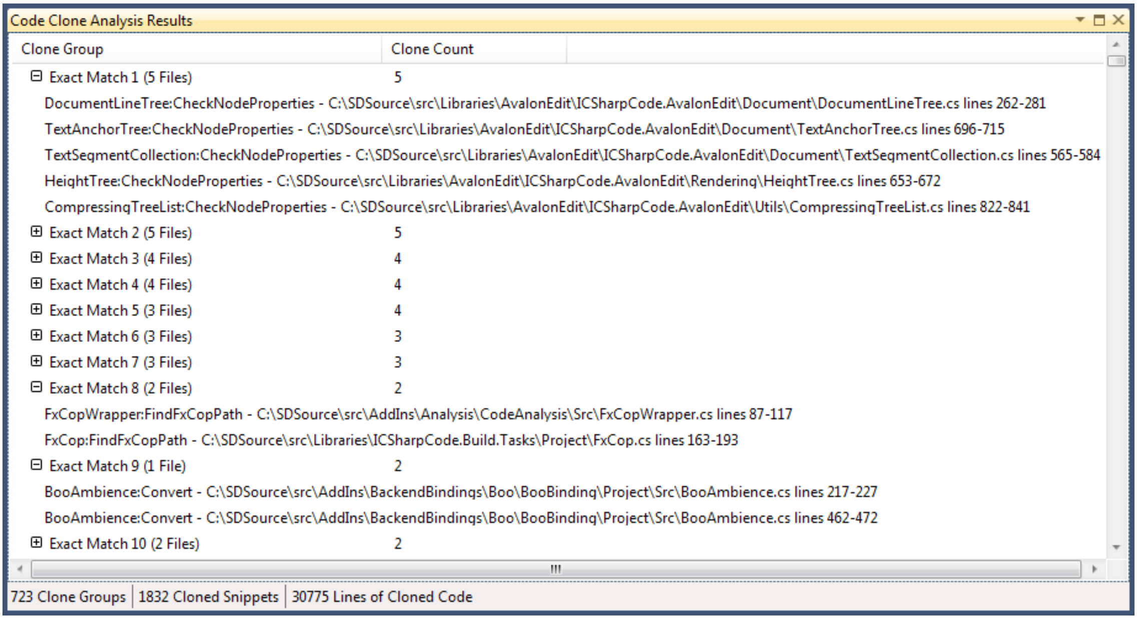
Class: **ExsltDatesAndTimes** File: **ExsltDates**

```
public static double MonthInYear(st
{
    try
    {
        var date = global::System.D
        return date.Month;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

Class: **ExsltDatesAndTimes** File: **ExsltDates**

```
public static double DayInYear(strir
{
    try
    {
        var date = global::System.Da
        return date.DayOfYear;
    }
    catch (FormatException)
    {
        return Double.NaN;
    }
}
```

Visual Studio Code Clone Analysis does not highlight duplicates inside the code editor, but it has a tool window to review all code duplicates:



Clone Group	Clone Count
⊟ Exact Match 1 (5 Files)	5
DocumentLineTree:CheckNodeProperties - C:\SDSource\src\Libraries\AvalonEdit\ICSharpCode.AvalonEdit\Document\DocumentLineTree.cs lines 262-281	
TextAnchorTree:CheckNodeProperties - C:\SDSource\src\Libraries\AvalonEdit\ICSharpCode.AvalonEdit\Document\TextAnchorTree.cs lines 696-715	
TextSegmentCollection:CheckNodeProperties - C:\SDSource\src\Libraries\AvalonEdit\ICSharpCode.AvalonEdit\Document\TextSegmentCollection.cs lines 565-584	
HeightTree:CheckNodeProperties - C:\SDSource\src\Libraries\AvalonEdit\ICSharpCode.AvalonEdit\Rendering\HeightTree.cs lines 653-672	
CompressingTreeList:CheckNodeProperties - C:\SDSource\src\Libraries\AvalonEdit\ICSharpCode.AvalonEdit\Utils\CompressingTreeList.cs lines 822-841	
⊞ Exact Match 2 (5 Files)	5
⊞ Exact Match 3 (4 Files)	4
⊞ Exact Match 4 (4 Files)	4
⊞ Exact Match 5 (3 Files)	4
⊞ Exact Match 6 (3 Files)	3
⊞ Exact Match 7 (3 Files)	3
⊟ Exact Match 8 (2 Files)	2
FxCopWrapper:FindFxCopPath - C:\SDSource\src\AddIns\Analysis\CodeAnalysis\Src\FxCopWrapper.cs lines 87-117	
FxCop:FindFxCopPath - C:\SDSource\src\Libraries\ICSharpCode.Build.Tasks\Project\FxCop.cs lines 163-193	
⊟ Exact Match 9 (1 File)	2
BooAmbience:Convert - C:\SDSource\src\AddIns\BackendBindings\Boo\BooBinding\Project\Src\BooAmbience.cs lines 217-227	
BooAmbience:Convert - C:\SDSource\src\AddIns\BackendBindings\Boo\BooBinding\Project\Src\BooAmbience.cs lines 462-472	
⊞ Exact Match 10 (2 Files)	2

723 Clone Groups | 1832 Cloned Snippets | 30775 Lines of Cloned Code

Analysis speed

Measurements were taken on one of the open source projects that contains 86 projects, and its overall size is 70M bytes.

First analysis run:

	Time
VS Code Clone Analysis	min 15 sec 19
(CR DDC inside VS (default analysis level = 4	min 45 sec 2
(CR DDC inside VS (analysis level = 1	min 32 sec 4

Second analysis run (after VS is reloaded):

	Time
VS Code Clone Analysis	min 0 sec 19
(CR DDC inside VS (default analysis level = 4	min 15 sec 0
(CR DDC inside VS (analysis level = 1	min 37 sec 3

As you can see, there's some difference in the speed of analysis, especially in the second and subsequent analysis runs.

Capabilities

Both features do the same thing – they analyze your code to find code duplicates. Let's see what capabilities both provide:

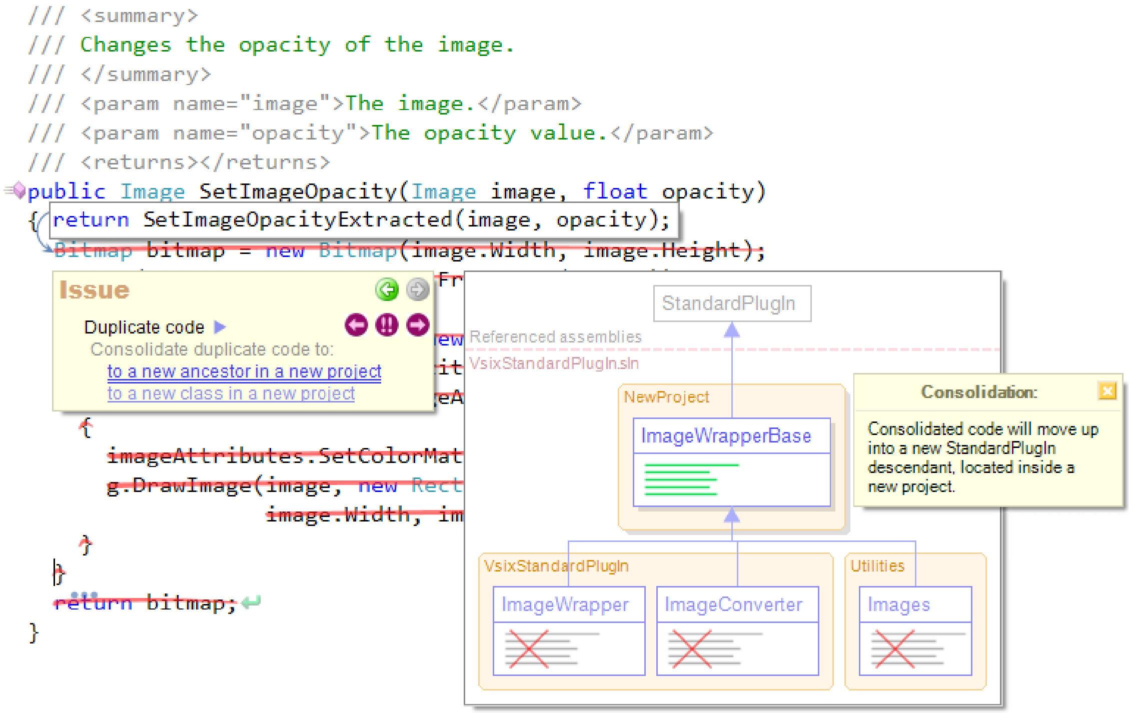
	Visual Studio CCA	CodeRush DDC
Configurable Analysis level	No: always shows matches categorized into Exact/Strong/Medium/Week categories	Yes, the analysis level can be set from 0 to 10th level
Entire solution analysis	*Yes	Yes
Find Matching Clones	Yes	**No
Analysis exclusions	***For specific files	For projects, folders, and files via two mouse clicks
Consolidation	No	Yes

** code clones less than 10 statements long won't be discovered when analyzing the entire solution*
*** planned for the future release*
**** files can be excluded by creating a file in the project with a ".codeclonesettings" extension and then adding some XML data to the created file to exclude the appropriate files in the project.*

Consolidation

Because the primary purpose of duplicate code detection is removing code duplications from the system, it is very important and useful to have a feature that does this automatically for you. **CodeRush DDC** has a Code Consolidation engine that removes code duplications by replacing them with a functionally identical single block of code. Other existing duplicate code analysis tools does not have such a capability at the moment.

The consolidation preview hint allows you to review the changes being made:



```
/// <summary>
/// Changes the opacity of the image.
/// </summary>
/// <param name="image">The image.</param>
/// <param name="opacity">The opacity value.</param>
/// <returns></returns>
public Image SetImageOpacity(Image image, float opacity)
{ return SetImageOpacityExtracted(image, opacity);
  Bitmap bitmap = new Bitmap(image.Width, image.Height);
                                        Fr
                                        ew
                                        it
                                        eA
    {
      imageAttributes.SetColorMat
      g.DrawImage(image, new Rect
                  image.Width, im
    }
  }
  return bitmap;
}
```

Issue

Duplicate code

Consolidate duplicate code to:

to a new ancestor in a new project

to a new class in a new project

Consolidation:

Consolidated code will move up into a new StandardPlugIn descendant, located inside a new project.

Chapter 2. Static code analysis and code issues technology

Code Issues technology overview

Code Issues technology (also known as **Code Analysis**) is essentially background static analyzer of the source code. The code is analyzed during code writing and/or reading without compiling or executing the program.

One of the most prominent uses of a static analyzer is for code defect detection. While you're working in the source code, it will point out the code errors even before it's compiled, and other specifics that can help improve the overall quality of the source code. This can help you improve your coding practices, and learn new language technics. Static-analysis techniques can also detect buffer overflows, security vulnerabilities, memory leaks, timing anomalies (such as race conditions, deadlocks), dead or unused source code segments, and other common programming mistakes.

There are different types of code issues this technology provides, such as errors, warnings, hints, dead code and code smells. All of these issue types are displayed in different highlighting in the Visual Studio code editor, to visually distinguish between them. Issues are also collected in the special Code Issues tool window, which is intended to analyze code issues for the entire solution with filtering, sorting and navigation capability.

Many code issues can be easily fixed with the corresponding code fixes available in the Code Fix hint, shown right inside the code editor. Any issues can also be suppressed for different kinds of scopes like current class, current project, solution or be completely disabled via a single mouse click.

The **Code Issues** technology is extensible, and it is easy to write your own code issue provider using the DXCore plug-in. For example, you can implement style-checking issues intended to verify the compliance of a given source code to your team coding style guidelines.

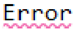
Code Issues types overview

There are different types of code issues that the Code Analysis technology provides. Each issue is highlighted with the appropriate color, and has its own icon, depending on its type. These types are:

- Errors
- Warnings
- Hints
- Dead Code
- Duplicate Code
- Code Smells

Errors

Errors are highlighted with a red wavy underline:



Error

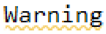
Code errors indicate that the code is invalid. Errors will definitely lead to the corresponding compile-time errors. Such errors indicate code defects (for example, syntax errors or rules violations of the programming language) that must be fixed before the code can be compiled. Seeing errors in the code editor allows you to fix them before starting the compile process, which may take significant time.

Examples of error code issues:

- Cannot create an instance of interface
- Static constructors must be parameterless
- Try statement without catch or finally

Warnings

Warnings are highlighted with an orange wavy underline:


Warning

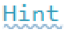
Warnings indicate syntactically valid but very suspicious code. A warning indicates you've done something bad, but not something that will prevent the code from being compiled. You should fix whatever causes warnings, since they often lead to other problems that will not be so easy to find.

Examples of warning code issues:

- Catch block is empty
- Undisposed local

Hints

Hints are highlighted with a blue wavy underline:


Hint

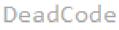
Hints bring your attention to specific code blocks, and may suggest useful recommendations for their improvement and refactoring. These recommendations can be completely ignored; it is up to you to apply or skip them.

Examples of code hints:

- Can initialize conditionally
- Field can be read-only

Dead Code

Dead Code issues highlight the source code in gray:


DeadCode

Dead code means unnecessary, inoperative code that can be, or better yet, should be removed. You can significantly improve the source code quality by removing dead code. While the functionality won't be changed, it will improve the internal quality – the readability of the source code. This will help in maintenance by decreasing the maintained code size, making it easier to understand the program, and preventing errors from being introduced.

There are several types of dead code, such as unused declarations (methods, properties, variables, parameters, return values, event handlers, constants, enumerations, classes, interfaces), unreachable code or statement, unused files, unused visual controls, etc.

Examples of Dead Code issues:

- Empty namespace declaration

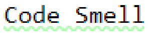
- Redundant destructor

Duplicate Code

This type of a code issue highlights duplicated code. Duplicate code occurs two or more times, either within a single program or across different programs. Code duplication is considered an expensive practice that should be avoided because it complicates the maintenance and evolution of the software. The duplicate code is highlighted in purple.

Code Smells

Code Smells are highlighted in green:



Code Smell

A code smell is a hint that something might be wrong with your source code, but not necessarily. Smells are indicators of where your code might be hard to read, maintain or evolve, rather than things that are specifically wrong. A smell usually indicates that the code has low quality, for example: large complex methods, long parameters lists, duplicated code, huge classes, inconsistent names, etc.

Sometimes a code smell issue may also detect logic errors. A logic error occurs when your program simply doesn't do what you want it to.

Examples of code smell issues:

- Case statement has incorrect range of integers expression

- Complex Member

You can tweak colors of code issues inside the Visual Studio Fonts and Colors configuration dialog.

Code Issues visual presentation

CodeRush code issues technology highlights found issues right inside the Visual Studio code editor with different colors. There are two permanent bars with issue indicators to the left of the code editor, and to the right of the vertical scroll bar. The third optional bar is intended to view solution wide analysis statistics for files and projects. This bar appears at the bottom of the code editor, if enabled.

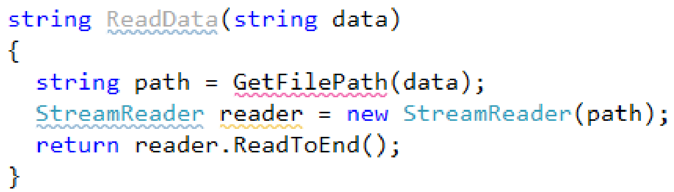
Let's take a closer look at code editor highlighting and these three code issues bars.

As already noted, different types of code issues are highlighted with different color (which you can tweak). There are two ways of issue highlighting:

- a wavy underline

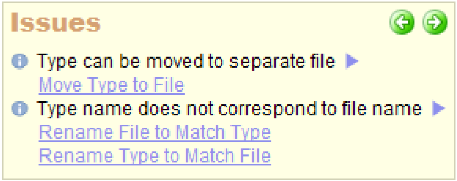
- a gray font for a source code.

A gray font is used for the Dead Code issue type. Other issue types use different colored underlines, such as red, blue, yellow, and green:



```
string ReadData(string data)
{
   string path = GetFilePath(data);
   StreamReader reader = new StreamReader(path);
   return reader.ReadToEnd();
}
```

Hovering over these visual marks inside the code editor will show you the **Code Fix** hint:



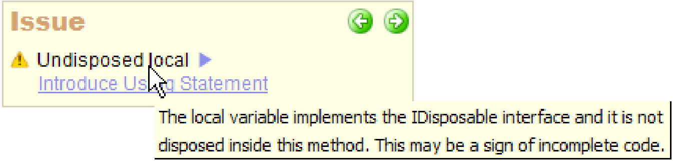
Issues

Type can be moved to separate file ▶
Move Type to File

Type name does not correspond to file name ▶
Rename File to Match Type
Rename Type to Match File

This hint is an important part of visual presentation of code issues. It contains information about code issues and different ways of their fixing. This hint allows you:

- to see the list of issues found for a particular block of code
- to see a brief description for an issue in the list when hovering over it:

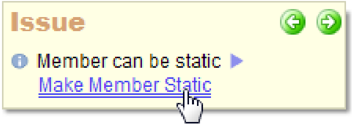


Issue

Undisposed local ▶
Introduce Using Statement

The local variable implements the IDisposable interface and it is not disposed inside this method. This may be a sign of incomplete code.

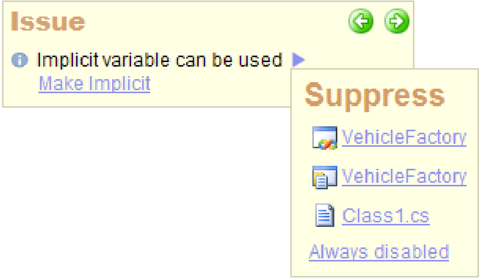
- to correct an issue by clicking the available fix provider link:



Issue

Member can be static ▶
Make Member Static

- to suppress or disable an issue in a different kind of scopes by clicking the blue triangle:



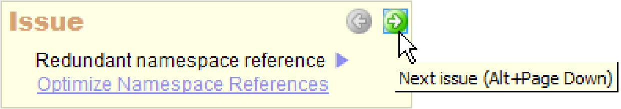
Issue

Implicit variable can be used ▶
Make Implicit

Suppress

VehicleFactory
VehicleFactory
Class1.cs
Always disabled

- to navigate between neighboring code issues using two green navigation buttons:



Issue

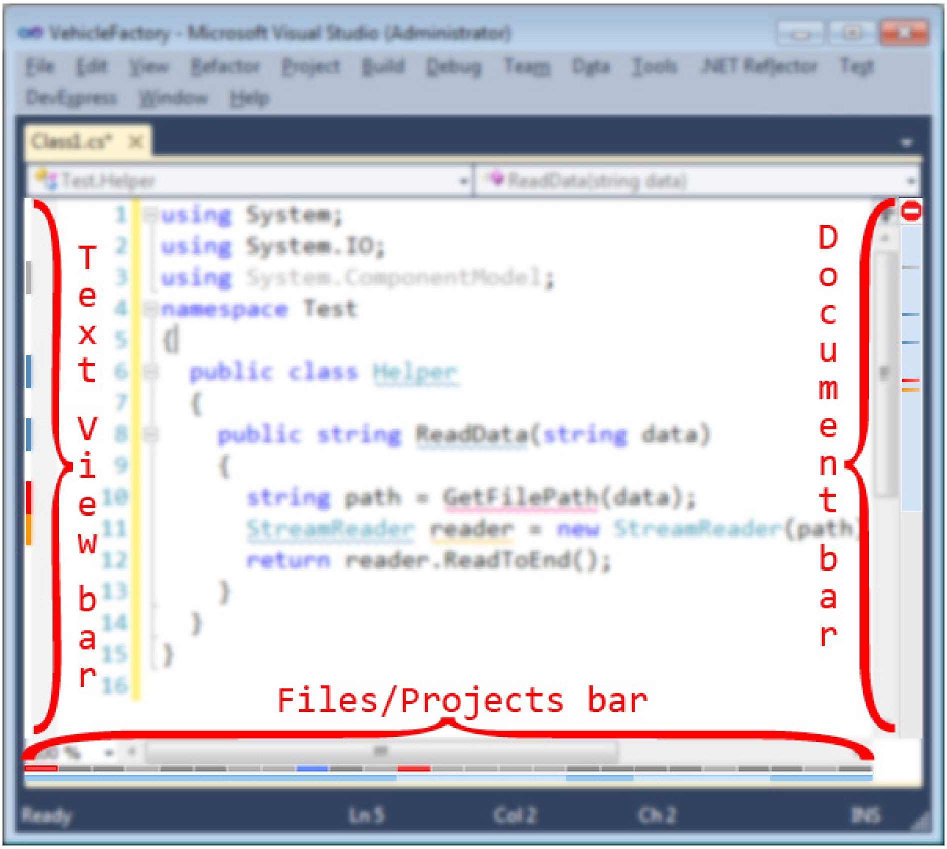
Redundant namespace reference

Optimize Namespace References

If the **Code Fix** hint overlaps the source code, you can move it to another place by dragging using the mouse. The hint can also be controlled using the keyboard. The following shortcuts are available:

Shortcut	Description
Arrow keys	.Selects the next or previous fix link
Tab	Navigates between the fix links and suppression triangles.
Enter	.Applies the selected fix to correct an issue
Esc	.Hides the hint

Now, the three code issues bars.



VehicleFactory - Microsoft Visual Studio (Administrator)
File Edit View Refactor Project Build Debug Team Data Tools .NET Reflector Test
DevExpress Window Help
Class1.cs*
Test.Helper
ReadData(string data)
Text View bar
Document bar
Files/Projects bar
100 %
Ready
Ln 5
Col 2
Ch 2
INS

```
using System;
using System.IO;
using System.ComponentModel;
namespace Test
{
    public class Helper
    {
        public string ReadData(string data)
        {
            string path = GetFilePath(data);
            StreamReader reader = new StreamReader(path);
            return reader.ReadToEnd();
        }
    }
}
```

The bar on the left marks the code issues inside of the active text view. So, the issues you see on the code editor screen are "mirrored" to the left bar on the corresponding lines where issue is highlighted. Hovering over a mark will show you the **Code Fix** hint.

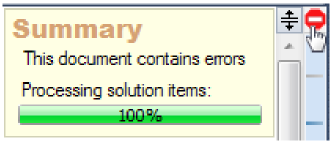
The bar on the right consists of the two parts:

- document summary status on the top represented by a small icon indicating the issues state of the current source file (text document)
- marks of issues for the entire source file (text document)

The document summary indicator shows whether the current source file contains issues. The following states are available:

Icon	Description
●	The current document is being checked.
🟢	The current document does not contain any issues.
ℹ	The current document contains hints.
⚠	The current document contains warnings.
⛔	The current document contains errors.

If you hover over an icon – the hint with the status description appears:



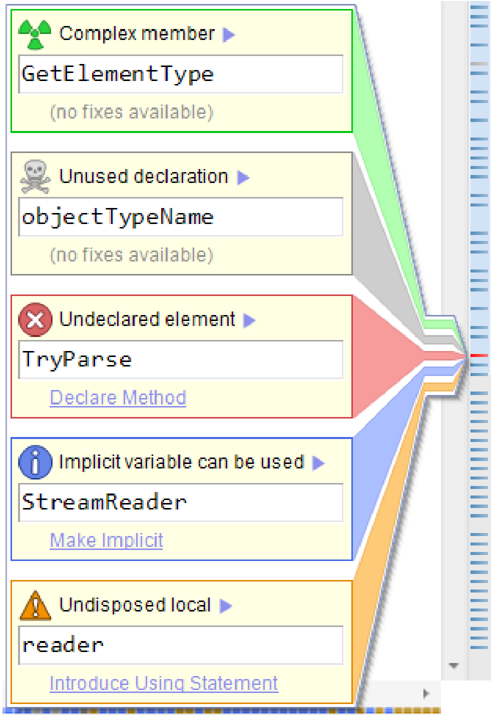
Summary

This document contains errors

Processing solution items:

100%

The other part of the bar on the right shows code issues found inside of the entire source file. Hovering over this bar will show you the **Code Fix** hint or a list of **Code Fix** hints:



Complex member ▶
GetElementType
(no fixes available)

Unused declaration ▶
objectTypeName
(no fixes available)

Undeclared element ▶
TryParse
Declare Method

Implicit variable can be used ▶
StreamReader
Make Implicit

Undisposed local ▶
reader
Introduce Using Statement

These hints shown from this bar has two more options:

- temporarily make visible the block of code with an issue. Once you hover over an issue, the code editor will be scrolled to the code with an issue. And if the hint is closed or mouse cursor is moved out of the hint, the code editor will scroll back to the initial starting position (before the hint appeared).

- navigate to the issue in question by clicking on it. If you click on the hint, the code editor will be permanently scrolled to the code block with an issue, so you can continue to work with it.

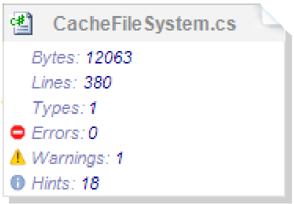
The last Project/File bar at the bottom of the code editor contains of two parts (lines):

- Solution files (at the top)

- Solution projects (at the bottom)

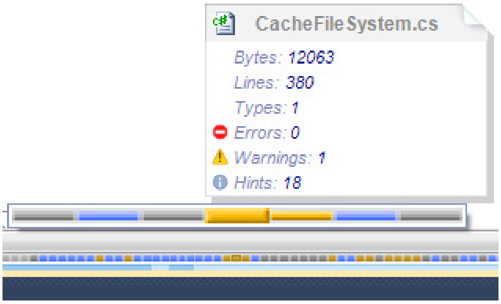
The project line lists all projects of the current solution. Project indicators are highlighted in one of the two blue colors. Hovering over the project line will show you the name of the project.

The files line lists all files of the solution. The indicators representing files are highlighted with a color according to its state. If the file contains errors, the indicator on the bar is highlighted in red, and so on. Hovering over the files line will show you the **File Info** hint with brief information about the file: the size of the file, the number of lines, the number of types and the count of warnings, errors and hints found in it:



CacheFileSystem.cs
Bytes: 12063
Lines: 380
Types: 1
Errors: 0
Warnings: 1
Hints: 18

If the solution contains hundreds or thousands of files, the optional **Magnifier** window is available for the files bar. You can tweak the number of file indicators shown in this window and the minimal width of the file indicator when the magnifier window must appear. This is what it looks like when both the **File Info** hint and **Magnifier** window are shown:



CacheFileSystem.cs
Bytes: 12063
Lines: 380
Types: 1
Errors: 0
Warnings: 1
Hints: 18

Code Issues fixes and suppression

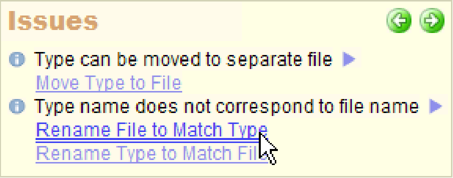
Code Issues can be easily fixed with the corresponding code fixes. The code fixes are operations that allow you to automatically fix the issue by changing the source code, so the issue is no longer valid for the block of code in question. The code fixes are refactoring or code providers assigned to the code issues as fixes.

There are several ways to fix an issue:

1. Code Fix hint
2. Zoom Window
3. Manually apply a fix

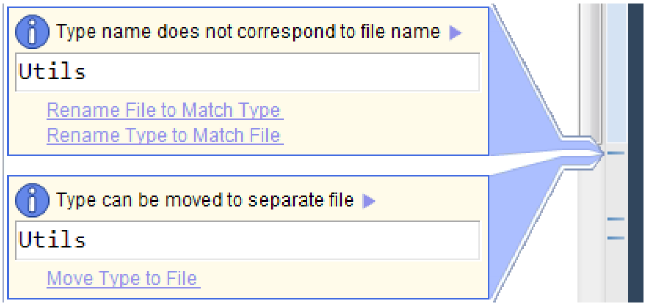
Code Fix hint

The hint shows not only the list of code issues, but when you hover the mouse over the issue highlighted in the code editor, available code fixes. You can also see this hint when hovering over the left view bar. Clicking the link with the name of the code fix will automatically fix the issue. Hovering over a link will also show you a preview hint of changes being made after the fix is applied.


Issues
Type can be moved to separate file
Move Type to File
Type name does not correspond to file name
Rename File to Match Type
Rename Type to Match File

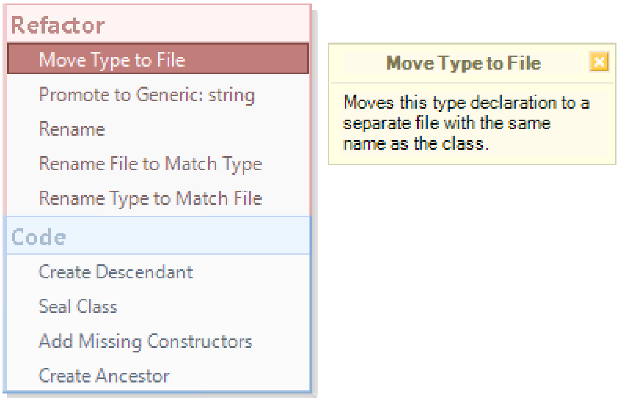
Zoom Window

This window appears when you hover the mouse over the marks on the right document bar. The window behaves as a container of several **Code Fix** hints. Clicking the link works similarly to the *Code Fix* hint.


Type name does not correspond to file name
Utils
Rename File to Match Type
Rename Type to Match File
Type can be moved to separate file
Utils
Move Type to File

Manual Fixes

The fix for an issue can be applied the same way as all other refactoring and/or code operations. For example, from the **Refactor!/Code! popup menu**:



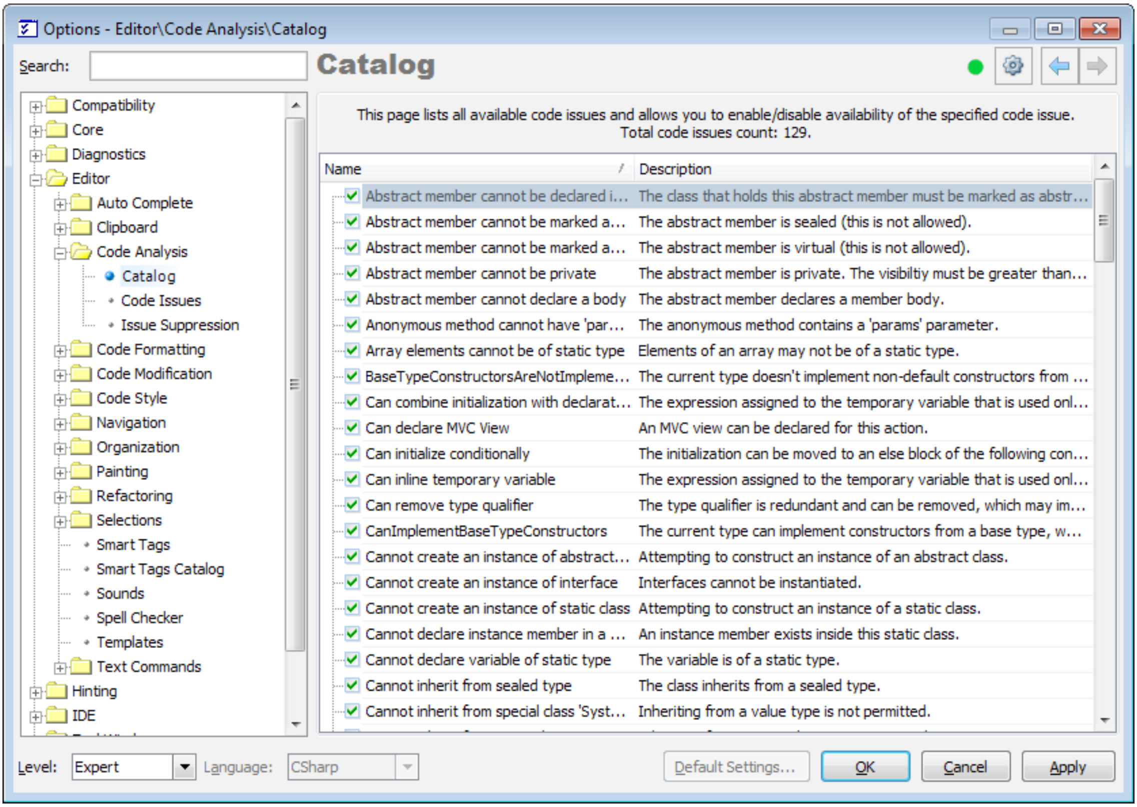
Moves this type declaration to a separate file with the same name as the class.

Issues Suppression

If you don't like some of the issues or don't want them to appear in a specific source code, you can easily suppress them for different scopes, such as:

- Solution
- Project
- File

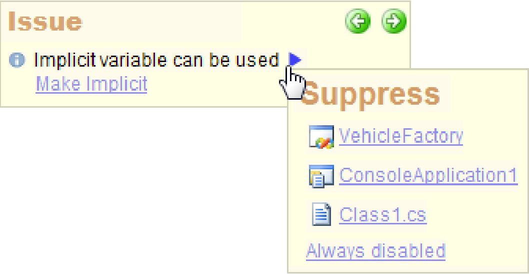
You can also completely disable the issue. If you have disabled the issue, you can reenable it on the code issues *Catalog* options page:



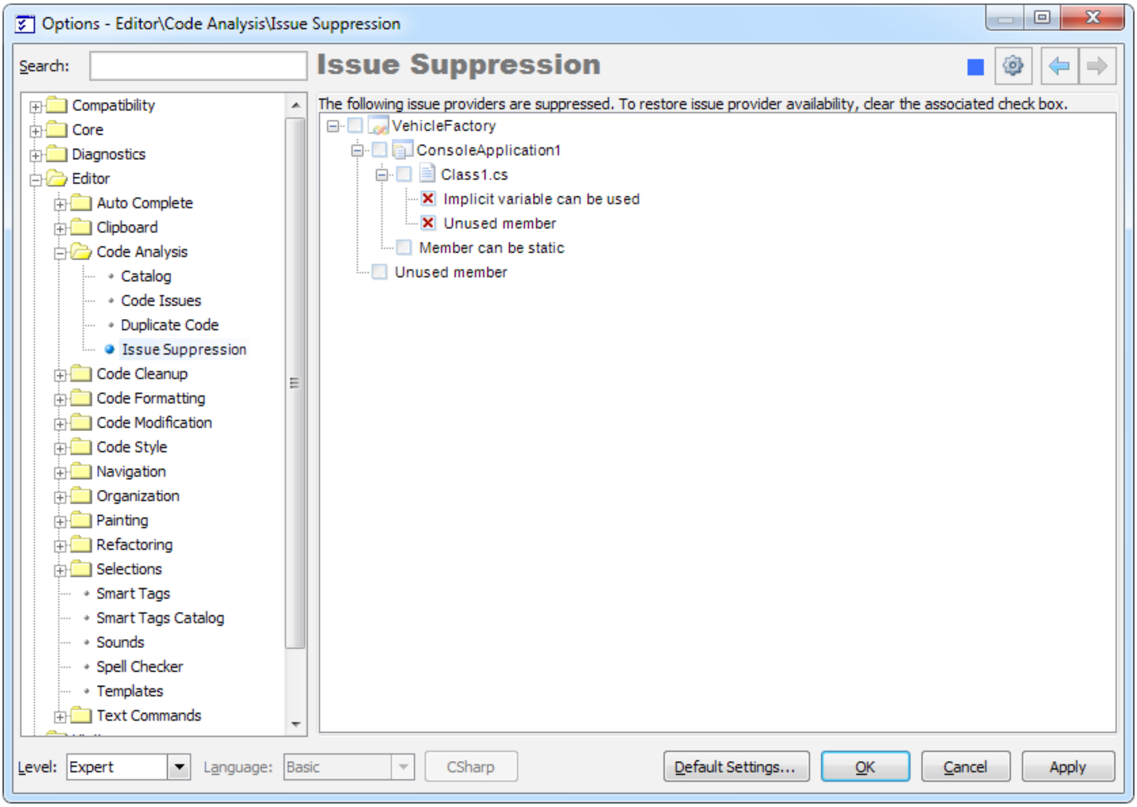
This page lists all available code issues and allows you to enable/disable availability of the specified code issue.
Total code issues count: 129.

Name	Description
☑ Abstract member cannot be declared i...	The class that holds this abstract member must be marked as abstr...
☑ Abstract member cannot be marked a...	The abstract member is sealed (this is not allowed).
☑ Abstract member cannot be marked a...	The abstract member is virtual (this is not allowed).
☑ Abstract member cannot be private	The abstract member is private. The visibiltiy must be greater than...
☑ Abstract member cannot declare a body	The abstract member declares a member body.
☑ Anonymous method cannot have 'par...	The anonymous method contains a 'params' parameter.
☑ Array elements cannot be of static type	Elements of an array may not be of a static type.
☑ BaseTypeConstructorsAreNotImpleme...	The current type doesn't implement non-default constructors from ...
☑ Can combine initialization with declarat...	The expression assigned to the temporary variable that is used onl...
☑ Can declare MVC View	An MVC view can be declared for this action.
☑ Can initialize conditionally	The initialization can be moved to an else block of the following con...
☑ Can inline temporary variable	The expression assigned to the temporary variable that is used onl...
☑ Can remove type qualifier	The type qualifier is redundant and can be removed, which may im...
☑ CanImplementBaseTypeConstructors	The current type can implement constructors from a base type, w...
☑ Cannot create an instance of abstract...	Attempting to construct an instance of an abstract class.
☑ Cannot create an instance of interface	Interfaces cannot be instantiated.
☑ Cannot create an instance of static class	Attempting to construct an instance of a static class.
☑ Cannot declare instance member in a ...	An instance member exists inside this static class.
☑ Cannot declare variable of static type	The variable is of a static type.
☑ Cannot inherit from sealed type	The class inherits from a sealed type.
☑ Cannot inherit from special class 'Syst...	Inheriting from a value type is not permitted.

To suppress an issue, click the blue triangle on the code fix hint, after which the **Suppression** menu appears with several options available:



Once you have suppressed the issue, you are able to restore it on the *Editor* | *Code Analysis* | *Issue Suppression* options page:



The following issue providers are suppressed. To restore issue provider availability, clear the associated check box.

- □ VehicleFactory
  - □ ConsoleApplication1
    - □ Class1.cs
      - ☒ Implicit variable can be used
      - ☒ Unused member
    - □ Member can be static
  - □ Unused member

Here, you can see the list of previously suppressed code issues and remove them from this list. After they are removed, they appear again in the source code if appropriate. On this options page, you can also change the suppression scope for the specific code issue.

The suppression settings are individual for every solution and are stored near the solution cache.

Code Issues navigation techniques

Analyzing the quality of your source code may take some time, especially if there are hundreds or thousands of code issues. While checking the code issues, you have to navigate thought all of them inside the entire solution. Here are available navigation techniques to switch between code issues:

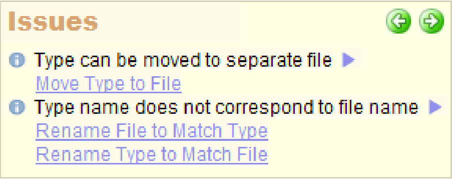
1. Code Fix hint

2. Keyboard shortcuts

3. Code Issues tool window

Code Fix hint

The hint appears inside the Visual Studio code editor or at the left view bar when you hover the mouse at the code issue highlighting. This hint has two green navigation buttons that allow you to switch between code issues inside the active source file. You can navigate to the next or previous code issue:



Issues

Type can be moved to separate file ▶

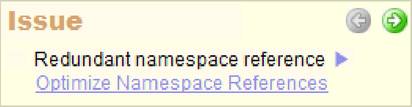
Move Type to File

Type name does not correspond to file name ▶

Rename File to Match Type

Rename Type to Match File

If there is no next or previous issue – the corresponding button becomes disabled:

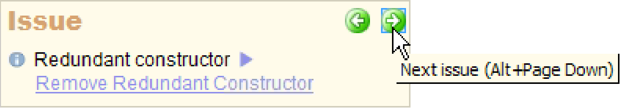


Issue

Redundant namespace reference ▶

Optimize Namespace References

If you hover the mouse over one of these buttons, you can see the keyboard shortcuts currently assigned for the navigation actions:



Issue

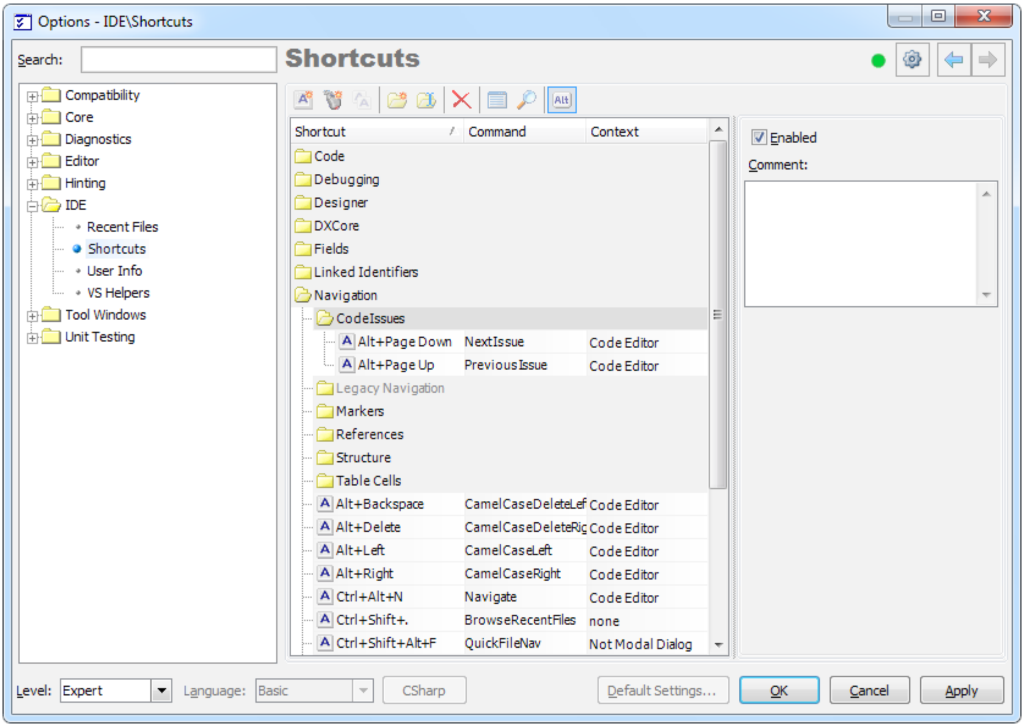
Redundant constructor ▶

Remove Redundant Constructor

Next issue (Alt+Page Down)

Shortcuts

If you hovered the mouse over the navigation buttons, you've already learned the keyboard shortcuts to switch between issues of the current source file. The default shortcuts are: *Alt+Page Down* to go to the next issue and *Al-t+Page Up* to go the previous issue. You may change these shortcuts on the Shortcutsoptions page of the Options Dialog:



Shortcut	Command	Context
Code		
Debugging		
Designer		
DXCore		
Fields		
Linked Identifiers		
Navigation		
CodeIssues		
Alt+Page Down	NextIssue	Code Editor
Alt+Page Up	PreviousIssue	Code Editor
Legacy Navigation		
Markers		
References		
Structure		
Table Cells		
Alt+Backspace	CamelCaseDeleteLef	Code Editor
Alt+Delete	CamelCaseDeleteRig	Code Editor
Alt+Left	CamelCaseLeft	Code Editor
Alt+Right	CamelCaseRight	Code Editor
Ctrl+Alt+N	Navigate	Code Editor
Ctrl+Shift+.	BrowseRecentFiles	none
Ctrl+Shift+Alt+F	QuickFileNav	Not Modal Dialog

☑ Enabled

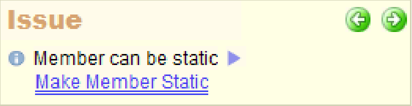
Comment:

Level: Expert

Language: Basic

CSharp

Once the shortcut is pressed, you will be navigated to the nearest code issue from the caret position and the **Code Fix hint** will show up. Here you can apply the fix using other keyboard shortcuts (e.g. *Tab* and *Enter*):

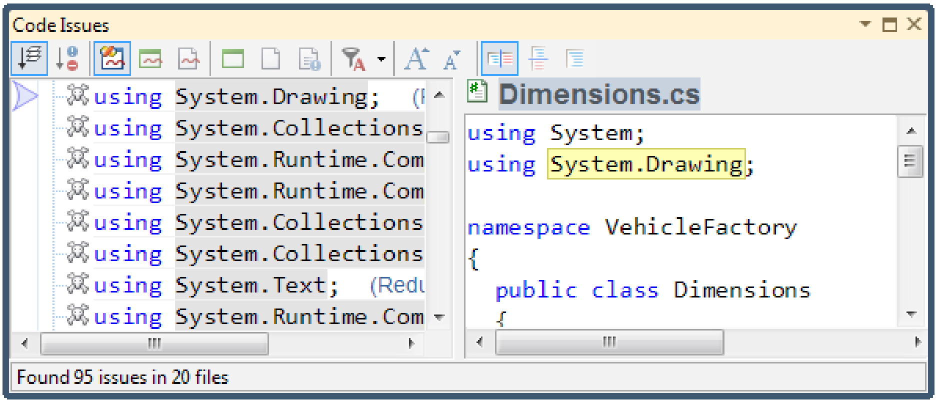


Member can be static

Make Member Static

Tool Window

The Code Issues tool window contains the list of code issues found in the entire solution:



```
using System.Drawing;    (F
using System.Collections
using System.Runtime.Com
using System.Runtime.Com
using System.Collections
using System.Collections
using System.Text;    (Redu
using System.Runtime.Com
```

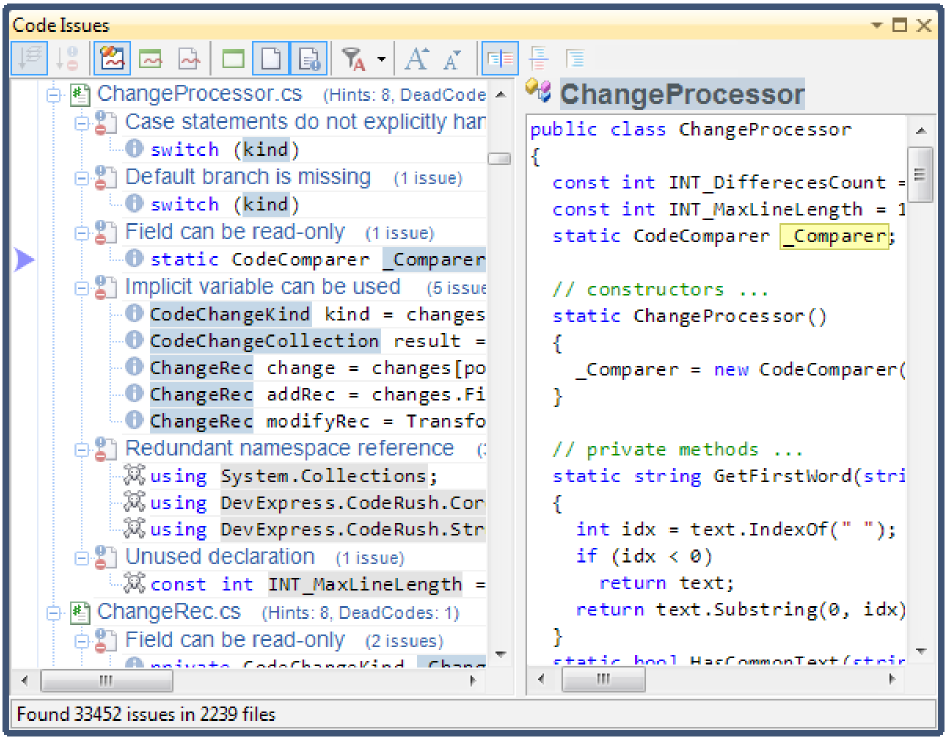
```
using System;
using System.Drawing;

namespace VehicleFactory
{
  public class Dimensions
  {
```

Found 95 issues in 20 files

The list requires using the mouse to switch between code issues. Once you double-click an issue, the appropriate source file will be activated inside the IDE and the caret will be positioned right over the chosen code issue, after that you can review the source code and make necessary changes if required. Bear in mind that using the navigation from the tool window will also leave you with a system gray marker at the previous caret position, so you can easily move back to the original location if necessary.

Code Issues tool window

The **CodeRush Code Issues tool window** shows a summary of code issues found inside the source code within an entire solution. It is intended to help you overview, analyze, navigate and fix issues such as errors, warnings, hints and dead code:



```
ChangeProcessor.cs   (Hints: 8, DeadCode
  Case statements do not explicitly har
    switch (kind)
  Default branch is missing   (1 issue)
    switch (kind)
  Field can be read-only   (1 issue)
    static CodeComparer _Comparer
  Implicit variable can be used   (5 issue
    CodeChangeKind kind = changes
    CodeChangeCollection result =
    ChangeRec change = changes[po
    ChangeRec addRec = changes.Fi
    ChangeRec modifyRec = Transfo
  Redundant namespace reference   (:
    using System.Collections;
    using DevExpress.CodeRush.Cor
    using DevExpress.CodeRush.Str
  Unused declaration   (1 issue)
    const int INT_MaxLineLength =
ChangeRec.cs   (Hints: 8, DeadCodes: 1)
  Field can be read-only   (2 issues)
```

ChangeProcessor

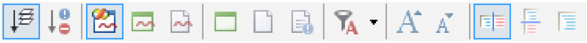
```
public class ChangeProcessor
{
  const int INT_DifferecesCount =
  const int INT_MaxLineLength = 1
  static CodeComparer _Comparer;

  // constructors ...
  static ChangeProcessor()
  {
    _Comparer = new CodeComparer(
  }

  // private methods ...
  static string GetFirstWord(stri
  {
    int idx = text.IndexOf(" ");
    if (idx < 0)
      return text;
    return text.Substring(0, idx)
  }
```

Found 33452 issues in 2239 files

The window contains two customizable views: the tree list view and the code preview, a toolbar and a status bar, by default. In the tree list view, you can see the list of issues found that are being populated live in real-time, without re-compiling the code. If you double click a code issue in the tree list, the corresponding source code file will be opened, and the editor caret will be positioned right on the code with an issue, so you can apply a code fix for it.

The overall number of code issues found and files being checked is shown on the status bar at the bottom of the tool window.

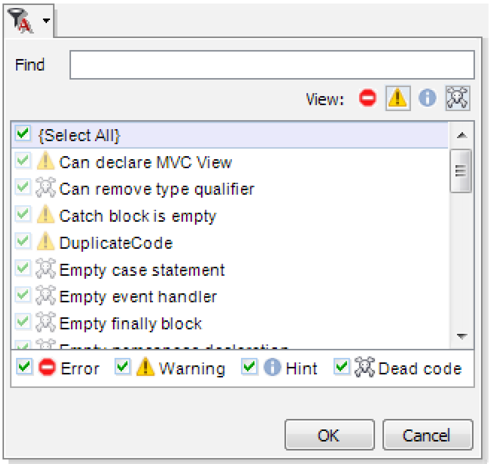
In the code preview view, there is a syntax-highlighted code preview of the selected issue in the tree list. The tree list is fully customizable, and you can arrange items and their view, using the tool bar buttons on the top of the window:



The toolbar has the following buttons in order of their appearance:

Icon	Button Name	Description
	Sort by location	Sorts the tree list items by an issues' location. The location is a source code range (coordinates), highlighted as an issue inside a specific file. The issues on the top of a file (for example, Redundant Namespace Reference code issue) will more likely appear at the top of the list. The button is available only when the list is shown as a plain list without parenting nodes and issue types.
	Sort by severity	Sorts the tree list items by an issues' severity, starting from hits and going to errors. The button is available only when the list is shown as a plain list without parenting nodes and issue types.
	Show code issues from active solution	Shows code issues found inside entire solution.
	Show code issues from active project	Shows code issues found inside currently active project.
	Show code issues from active file	.Shows code issues found inside currently active file
	Show parenting project nodes	Organizes code issues by projects in the tree list.
	Show parenting file nodes	Organizes code issues by files in the tree list.
	Show code issue types	Organizes code issues by their types in the tree list.
A	Filter	Allows you to filter the tree list using the Filter popup dialog. See below for more info on code issue filtering.
A	Increase font	Increases the size of the font used for the tree list view.
A	Decrease font	Decreases the size of the font used for the tree list view.
	(Landscape layout (code preview on right	Sets the landscape layout: tree list on the left and code preview on the right.
	(Portrait layout (code preview below	Sets the portrait layout: tree list on the top and code preview on the bottom.
	(References only (no code preview	Removes the code preview, so it is hidden.

All buttons have the appropriate captions, so it is easy to distinguish between them. The *Filter* button has an additional popup dialog that allows you to specify code issue filtering options for the tree list. Once you click it, the following popup dialog appears:



Find

View:

☑ {Select All}
☑ Can declare MVC View
☑ Can remove type qualifier
☑ Catch block is empty
☑ DuplicateCode
☑ Empty case statement
☑ Empty event handler
☑ Empty finally block
☑ Empty namespace declaration

☑ Error ☑ Warning ☑ Hint ☑ Dead code

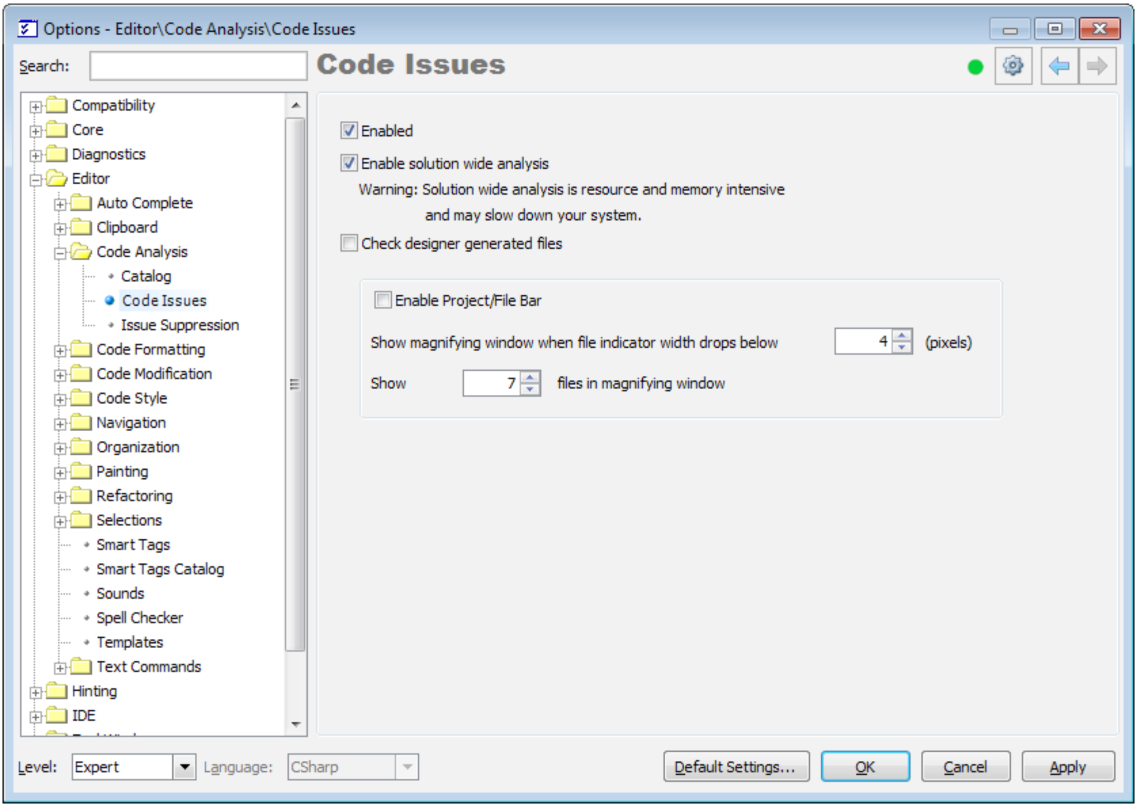
In the center of the dialog, there's a list of all code issues available, sorted alphabetically. Each item of the list has a check box, which specifies whether a particular code issue should be shown in the main tree list view of the tool window (if checked) or hidden (if unchecked). The special "*{Select All}*" item allows you to select code issues from the filtered issues list.

Check boxes on the bottom of the *Filter* popup dialog allow you to check or uncheck a group of code issues by type (errors, warnings, hints or dead code).

In the *Find* text box on the top of the dialog, you can search for a specific issue by typing its name. The *View* buttons provide the capability to specify which types of issues are shown in the filtered list of code issues. The icon represents the code issue type.

Code Issues configuration and options

CodeRush Code Issues configuration is available inside the IDE Tools Options Dialog and also in the Options Dialog of the Visual Studio. The main set of options is located on the **Editor** | **Code Analysis** | **Code Issues** options page:



☑ Enabled

☑ Enable solution wide analysis

Warning: Solution wide analysis is resource and memory intensive and may slow down your system.

☐ Check designer generated files

☐ Enable Project/File Bar

Show magnifying window when file indicator width drops below 4 (pixels)

Show 7 files in magnifying window

Level: Expert

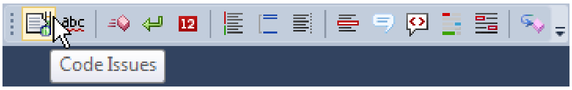
Language: CSharp

Available options are:

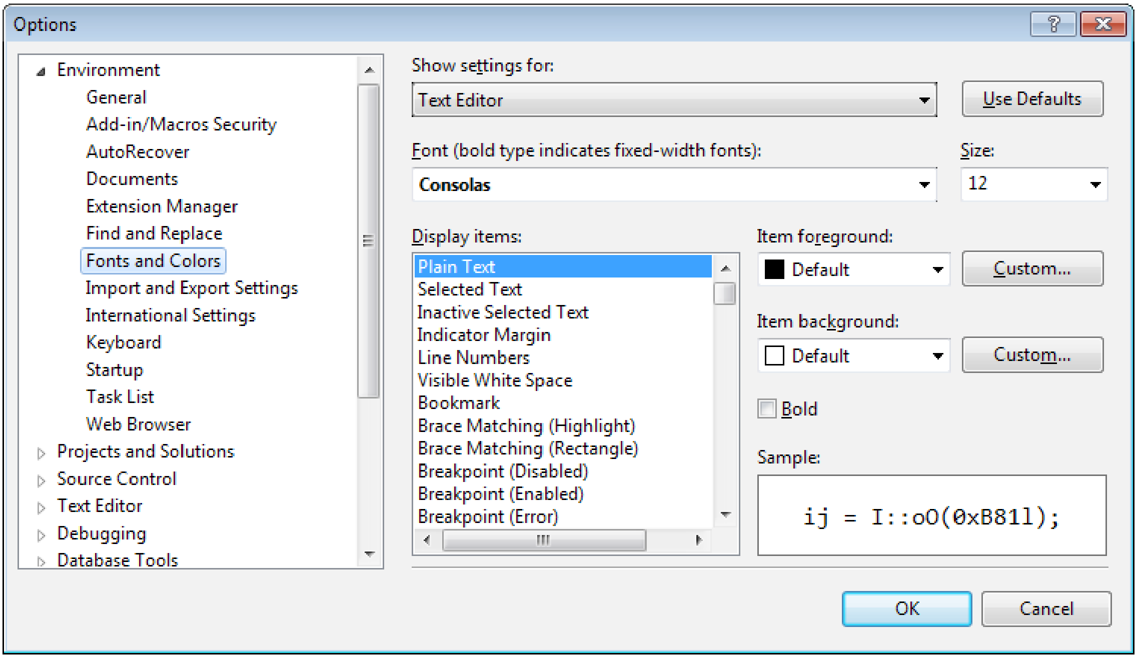
Option Name	Description
Enabled	Toggles the availability of the **Code Issues** analysis.
Show a Code Fix hint when hovering over issues in the code editor	Specifies whether a Code Fix hint should be shown when the mouse cursor is hovering over issue's highlighting.

Enable solution wide analysis	Specifies whether **Code Issues** analysis should analyze all source files in a background together with the opened file.
Check designer generated files	Specifies whether or not to check compiler or designer generated files.
Enabled Project/File Bar	Specifies the visibility of the **Project/File Bar**.
Show magnifying window when file indicator width drops below [X] files	Specifies the minimal number of pixels when the **Magnifier window** appears.
Show [X] files in magnifying window	Specifies the number of files shown in the **Magnifier window**.

You can also toggle the **Code Issues Analysis** on or off on the DXCore Visualize Toolbar:



The other options, such as colors of the different code issue types underlining, are available inside the Visual Studio configuration dialog. From the main IDE window, go to **Tools** | **Options**…, then **Environment** | **Fonts and Colors**:


Options

- Environment
  - General
  - Add-in/Macros Security
  - AutoRecover
  - Documents
  - Extension Manager
  - Find and Replace
  - Fonts and Colors
  - Import and Export Settings
  - International Settings
  - Keyboard
  - Startup
  - Task List
  - Web Browser
- Projects and Solutions
- Source Control
- Text Editor
- Debugging
- Database Tools

Show settings for: Text Editor

Font (bold type indicates fixed-width fonts): **Consolas**

Size: 12

Display items:

- Plain Text
- Selected Text
- Inactive Selected Text
- Indicator Margin
- Line Numbers
- Visible White Space
- Bookmark
- Brace Matching (Highlight)
- Brace Matching (Rectangle)
- Breakpoint (Disabled)
- Breakpoint (Enabled)
- Breakpoint (Error)

Item foreground: ■ Default

Item background: □ Default

□ Bold

Sample:

```
ij = I::oO(0xB81l);
```

In the "**Display items**:" list you can find the following **Code Issues** related items (see code issue types):

Item Caption	Description
Dead Code	Color of the dead code issues (gray font)
On-the-fly Error	Color of the red wavy underlines (e.g. undeclared elements)
On-the-fly Hint	Color for the blue wavy underlines (suggestions/hints)
On-the-fly Warning	Color for the yellow wavy underlines (warnings)
Code Smell {light}	Color for the green wavy underlines (code smells)

Code Smell {moderate}	Reserved, not used.
Code Smell {severe}	Reserved, not used.

For the all items above, you can change the "*Item foreground*" color option. For the "*Dead Code*" item you can also change the "*Item background*" color property.

Code Issues overview

Code Issues hints for declaration and initialization

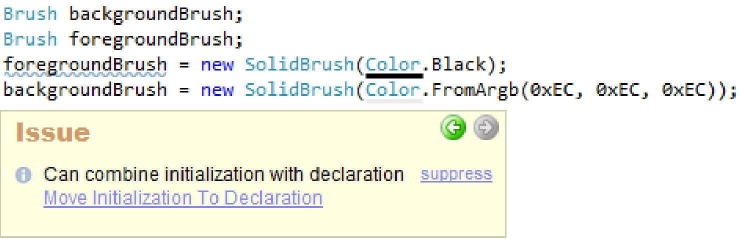
Here are a few simple code issues of a hint type (suggestion) for declarations and initializations.

Can combine initialization with declaration

Cause:

The declaration and its initialization can be combined into a single statement instead of two separate statements. Combining an initialization and declaration may improve code readability.

Sample:



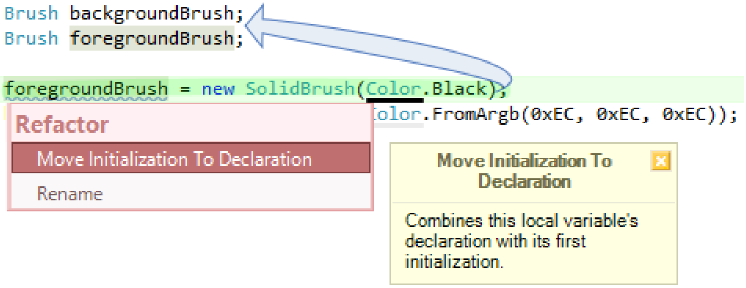
```
Brush backgroundBrush;
Brush foregroundBrush;
foregroundBrush = new SolidBrush(Color.Black);
backgroundBrush = new SolidBrush(Color.FromArgb(0xEC, 0xEC, 0xEC));
```

Issue

Can combine initialization with declaration

How to fix:

- Apply the **Move Initialization to Declaration** refactoring:



```
Brush backgroundBrush;
Brush foregroundBrush;

foregroundBrush = new SolidBrush(Color.Black);
                                 olor.FromArgb(0xEC, 0xEC, 0xEC));
```

Move Initialization To Declaration

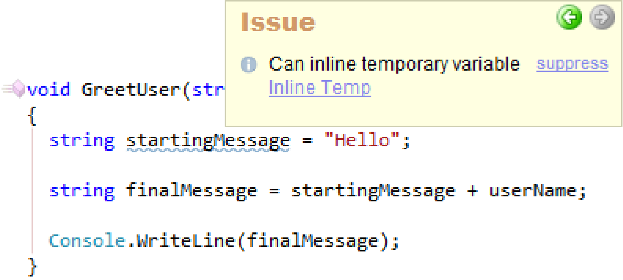
Combines this local variable's declaration with its first initialization.

Can inline temporary variable

Cause:

The expression assigned to the temporary variable that is used only once can be inlined. Removing a temporary variable may improve code readability.

Sample:

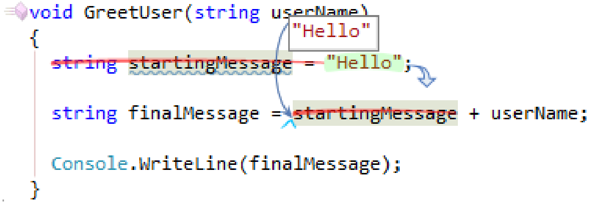


Can inline temporary variable

```
void GreetUser(str
{
    string startingMessage = "Hello";

    string finalMessage = startingMessage + userName;

    Console.WriteLine(finalMessage);
}
```

How to fix:

- Apply the **Inline Temp** refactoring:

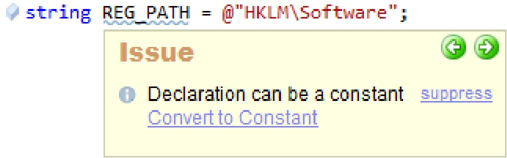


```
void GreetUser(string userName)
{
    ~~string startingMessage~~ = "Hello";

    string finalMessage = ~~startingMessage~~ + userName;

    Console.WriteLine(finalMessage);
}
```

Declaration can be a constant

Cause: If the initialized declaration does not change its value, it can be a constant. A constant differs from a variable because once declared, the value assigned to the constant cannot be modified. Having constant declarations creates a self-documenting and strong code by telling the developer that he/she is not allowed to change the value of the constant variable, which can results in fewer errors. Additionally, constants improve performance over regular variables in many cases, particularly when using integral types such as integer.

Sample:

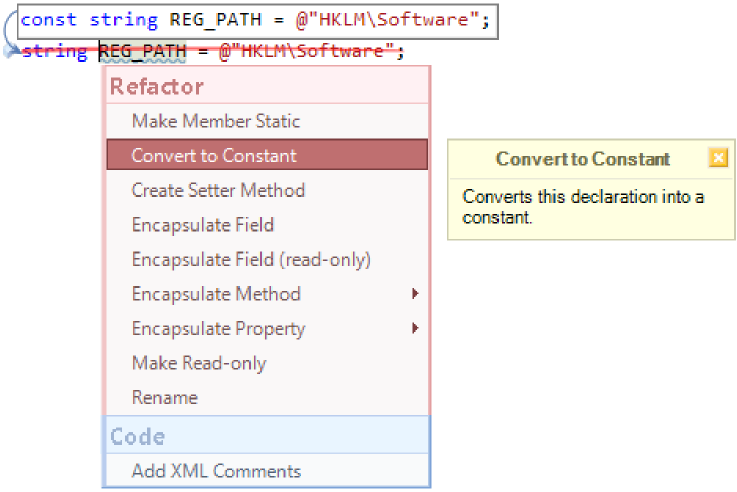


```
string REG_PATH = @"HKLM\Software";
```

Declaration can be a constant

How to fix:

- Apply the **Convert to Constan**t refactoring:



```
const string REG_PATH = @"HKLM\Software";
~~string REG_PATH = @"HKLM\Software";~~
```

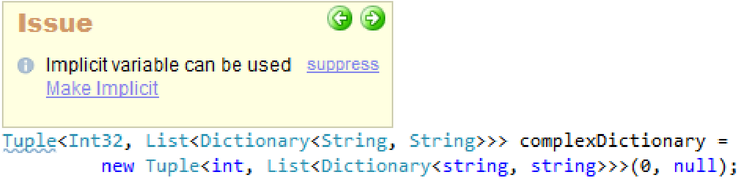
Converts this declaration into a constant.

Implicit variable can be used

Cause:

This code issue notifies a developer that an implicitly typed local variable can be used. Such variables provide a new simplified declaration syntax that instructs the compiler to infer the type of a new variable according to its initial usage. Converting a variable to an implicitly typed local variable may improve code readability.

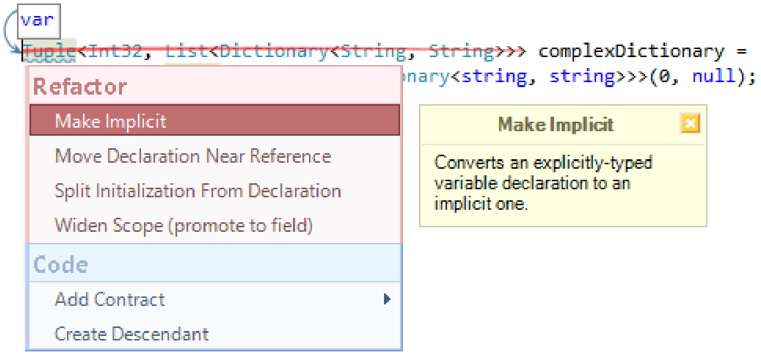
Sample:



```
Tuple<Int32, List<Dictionary<String, String>>> complexDictionary =
        new Tuple<int, List<Dictionary<string, string>>>(0, null);
```

How to fix:

- Apply the Make Implicit refactoring:



```
var
~~Tuple<Int32, List<Dictionary<String, String>>>~~ complexDictionary =
                              nary<string, string>>>(0, null);
```

Refactor
Make Implicit
Move Declaration Near Reference
Split Initialization From Declaration
Widen Scope (promote to field)
Code
Add Contract
Create Descendant
Make Implicit

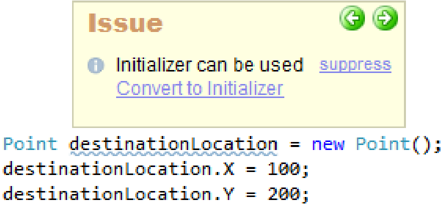
Converts an explicitly-typed variable declaration to an implicit one.

Initializer can be used

Cause:

The instantiation of a variable and subsequent property assignments can be converted into a object initializer expression. Object initializers allow you to set the initial values of fields or properties of an object as part of the single instantiation statement, which is more compact and may improve code readability

Sample:

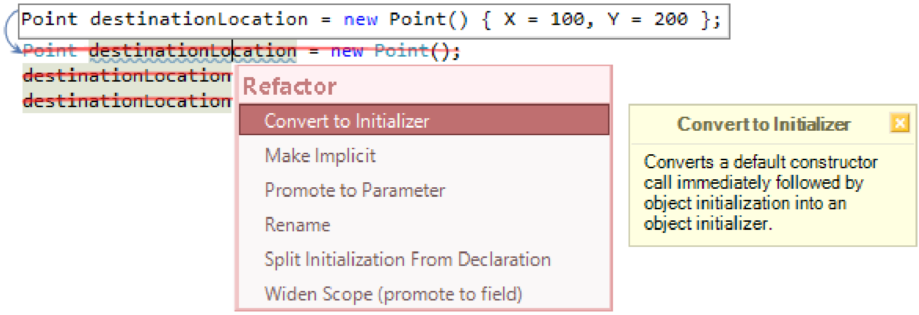


Initializer can be used

```
Point destinationLocation = new Point();
destinationLocation.X = 100;
destinationLocation.Y = 200;
```

How to fix:

- Apply the **Convert to Initializer** refactoring:



```
Point destinationLocation = new Point() { X = 100, Y = 200 };
~~Point destinationLocation = new Point();~~
~~destinationLocation~~
~~destinationLocation~~
```

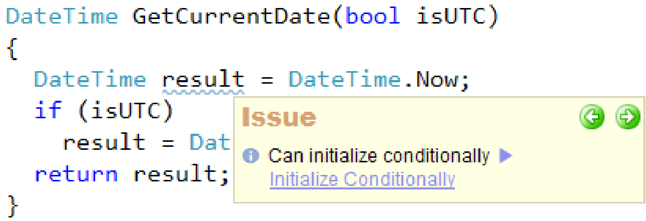
Converts a default constructor call immediately followed by object initialization into an object initializer.

Can initialize conditionally

Cause:

This code issue may improve the efficiency of the code. It is shown if the initialization can be moved to an ‘else’ block of the following conditional statement. If the condition of the conditional statement is met, than the first initialization is unnecessary. This issue is shown as a *suggestion* (*hint*).

Sample:



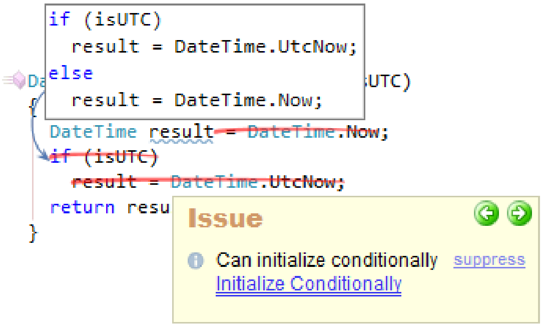
```
DateTime GetCurrentDate(bool isUTC)
{
  DateTime result = DateTime.Now;
  if (isUTC)
    result = Dat
  return result;
}
```

Can initialize conditionally

The method returns the current date and time on the computer. The ‘*isUTC*’ parameter specifies whether this method should return date and time, expressed as the local time (if *false*) or as the Coordinated Universal Time (if *true*). If we pass '*true*' to this method the resulting var is initialized twice, which is inefficient.

How to fix:

Apply the "**Initialize Conditionally**" refactoring:



```
if (isUTC)
  result = DateTime.UtcNow;
else
  result = DateTime.Now;
```

```
Da                                  UTC)
{
  DateTime result ~~= DateTime~~.Now;
  ~~if (isUTC)~~
    ~~result = DateTime.UtcNow;~~
  return resu
}
```

Issue

Can initialize conditionally suppress

Initialize Conditionally

Notice that the resulting var is no longer reinitialized (if we pass '*true*' to this method). In more complicated cases with time-consuming initialization this may improve the overall code performance.

Code Issues hints for expressions

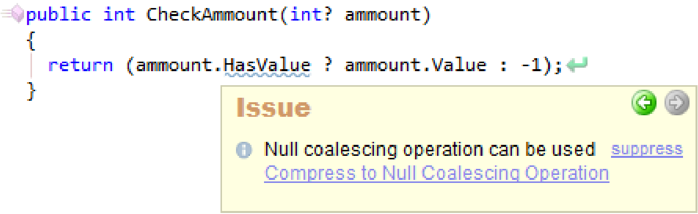
Here are the code issues for different expressions. If the fix is applied, following a hint it may optimize the code and/or improve its readability.

Null coalescing operation can be used

Cause:

The null coalescing operation verifies the value of a variable for 'nullness'. If the value is not equal to 'null', a variable's value is returned. If it is equal to 'null', a substitute value, provided as a second operand of the operation, is returned. The null coalescing operation simplifies the syntax of the ternary operation, for instance. Converting an expression to a null coalescing operation may be increase code readability.

Sample:



```
public int CheckAmmount(int? ammount)
{
  return (ammount.HasValue ? ammount.Value : -1);
}
```

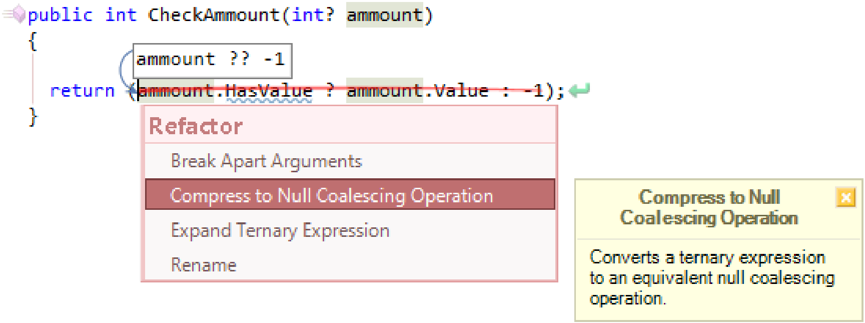
Issue

Null coalescing operation can be used suppress

Compress to Null Coalescing Operation

How to fix:

- Apply the **Compress to Null Coalescing Operation** refactoring:



```
public int CheckAmmount(int? ammount)
{
    return (ammount.HasValue ? ammount.Value : -1);
}
```

ammount ?? -1
Refactor
Break Apart Arguments
Compress to Null Coalescing Operation
Expand Ternary Expression
Rename
Compress to Null Coalescing Operation

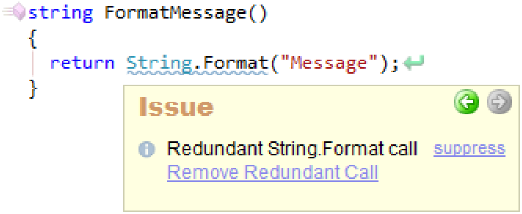
Converts a ternary expression to an equivalent null coalescing operation.

Redundant String.Format call

Cause:

The String.Format call allows composing a string data, that combines other strings and referenced variable values. If the String.Format call does not take referenced variable values, a simple string concatenation operation or even a single string value can be used instead of String.Format. This may optimize the code and improve its clarify.

Sample:

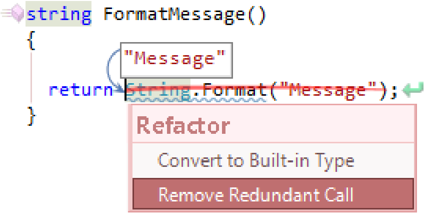


```
string FormatMessage()
{
    return String.Format("Message");
}
```

Redundant String.Format call

How to fix:

- Apply the **Remove Redundant Call** refactoring:



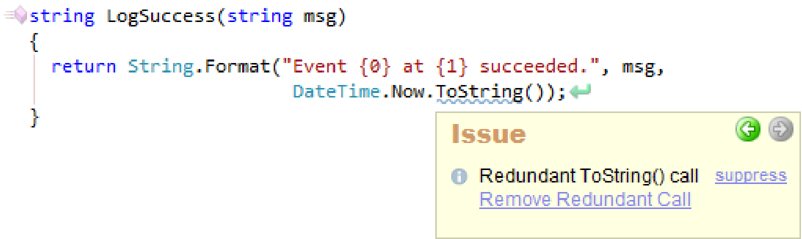
```
string FormatMessage()
{
    return String.Format("Message");
}
```

Redundant ToString() call

Cause:

There are cases when the ToString() call is redundant. In these cases, the compiler automatically calls ToString(). If a call ToString() is not required, a hint is shown suggesting you remove it.

Sample:

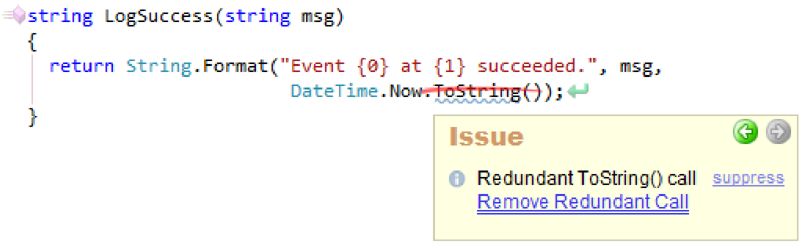


```
string LogSuccess(string msg)
{
  return String.Format("Event {0} at {1} succeeded.", msg,
                       DateTime.Now.ToString());
}
```

Redundant ToString() call

How to fix:

- Apply the **Remove Redundant Call** refactoring:



```
string LogSuccess(string msg)
{
  return String.Format("Event {0} at {1} succeeded.", msg,
                       DateTime.Now~~.ToString()~~);
}
```

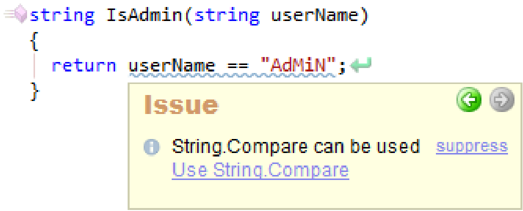
Redundant ToString() call

String.Compare can be used

Cause:

Strings can be compared using the equality operator '=='. However, this operator performs an exact, case-sensitive comparison of two strings.. For better support of string comparison, the String.Compare call can be used. This code issue of a hint type shows you that the equality operator can be changed into the String.Compare call.

Sample:

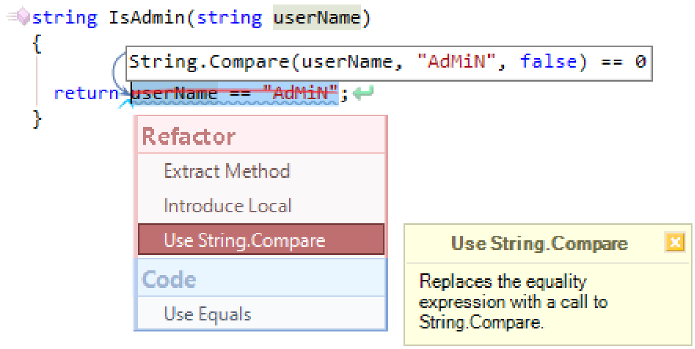


```
string IsAdmin(string userName)
{
  return userName == "AdMiN";
}
```

String.Compare can be used

How to fix:

- Apply the Use String.Compare refactoring:



```
string IsAdmin(string userName)
{
        String.Compare(userName, "AdMiN", false) == 0
    return userName == "AdMiN";
}
```

Use String.Compare

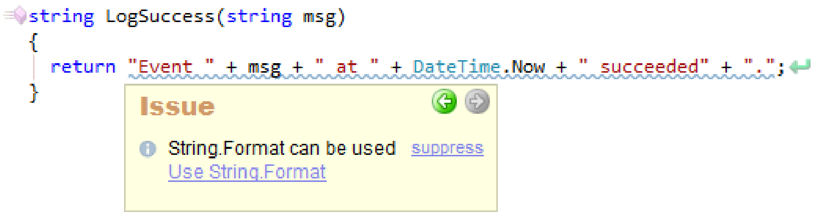
Replaces the equality expression with a call to String.Compare.

String.Format can be used

Cause:

String.Format can process typical strings and variable references. Sometimes, instead of concatenating these pieces with a string concatenation operator, it might be better to use the special String.Format call for this purpose. This may improve code readability and clarity.

Sample:

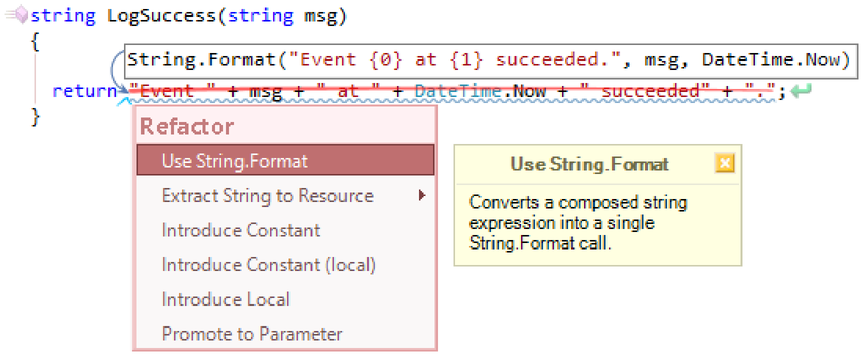


```
string LogSuccess(string msg)
{
    return "Event " + msg + " at " + DateTime.Now + " succeeded" + ".";
}
```

String.Format can be used

How to fix:

- Apply the Use String.Format refactoring:



```
string LogSuccess(string msg)
{
        String.Format("Event {0} at {1} succeeded.", msg, DateTime.Now)
    return "Event " + msg + " at " + DateTime.Now + " succeeded" + ".";
}
```

Use String.Format

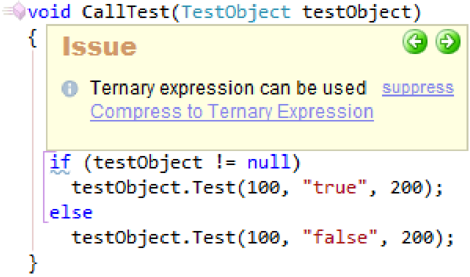
Converts a composed string expression into a single String.Format call.

Ternary expression can be used

Cause:

The conditional operator allows you to define a boolean condition and two expressions as a result of a boolean operation. Using the simplified ternary expression can improve code readability, because it allows you to compress if-else statements to a single expression.

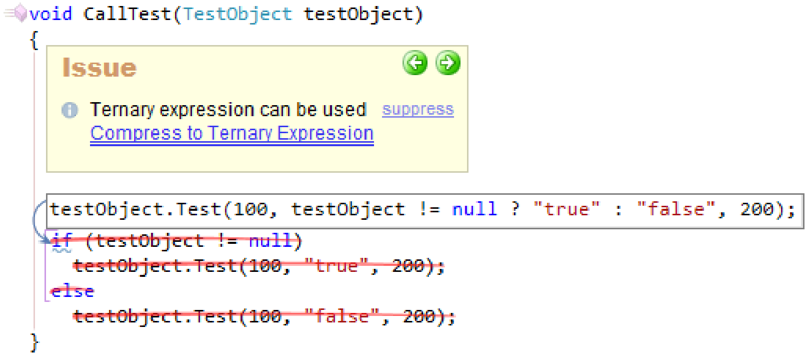
Sample:


Issue
Ternary expression can be used
suppress
Compress to Ternary Expression

```
void CallTest(TestObject testObject)
{
  if (testObject != null)
    testObject.Test(100, "true", 200);
  else
    testObject.Test(100, "false", 200);
}
```

How to fix:

- Apply the **Compress to Ternary Expression** refactoring:


Issue
Ternary expression can be used
suppress
Compress to Ternary Expression

```
void CallTest(TestObject testObject)
{
  testObject.Test(100, testObject != null ? "true" : "false", 200);
  ~~if (testObject != null)~~
    ~~testObject.Test(100, "true", 200);~~
  ~~else~~
    ~~testObject.Test(100, "false", 200);~~
}
```

Environment.NewLine can be used

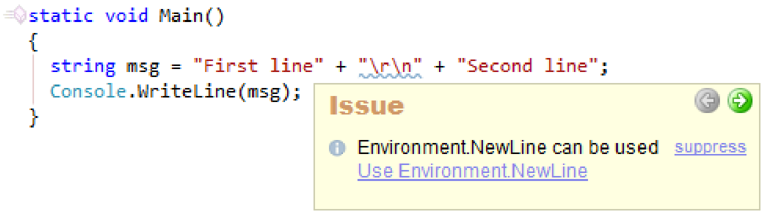
Cause:

When working with strings that contain multiple lines of text, you have to add newline characters to your strings, so each line is separated with a line break. Line breaks are often added by inserting a carriage return line feed escape characters. This can cause cross-platform compatibility issues, because your code might end up being compiled under Mono in Unix, for example.

The "*Environment.NewLine can be used*" CodeRush code issue shows a suggestion to convert escape characters ("*\r\n*") inside your code into the constant value defined in .NET Framework. Changing these strings to the *Environment.NewLine* constant will, firstly, improve the code clarity, so you don't have to use escape characters, and, secondly, fix potential platforms portability issues – so, you don't have to be concerned about such problems.

The *Environment.NewLine* is a static string property from the *System* namespace that is tied to the current executing environment or platform. The property returns a line feed string that is appropriate to the current operating system: "*\r\n*" for non-Unix platforms, or a string containing just a line feed ("*\n*") for Unix platforms.

Sample:

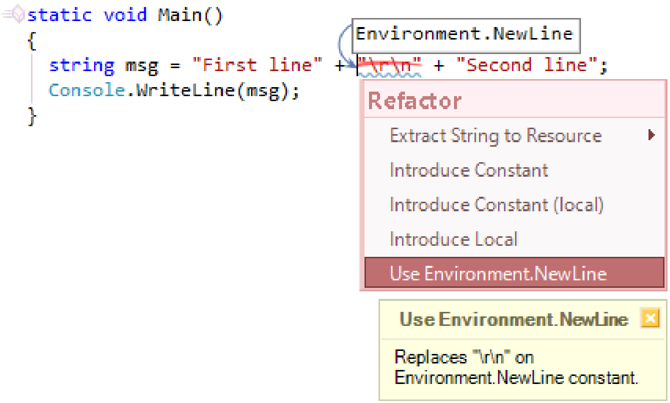


```
static void Main()
{
    string msg = "First line" + "\r\n" + "Second line";
    Console.WriteLine(msg);
}
```

Environment.NewLine can be used

How to fix:

- Apply the Use Environment.NewLine refactoring:



```
static void Main()
{
    string msg = "First line" + "\r\n" + "Second line";
    Console.WriteLine(msg);
}
```

Replaces "\r\n" on Environment.NewLine constant.

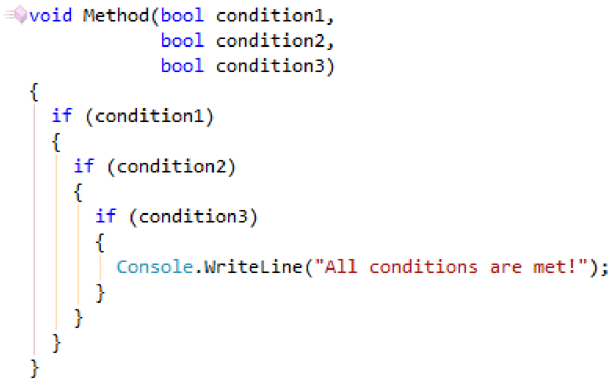
Nested code can be flattened

Cause:

This code issue shows a hint (suggestion) when the nested code structure can be simplified and unintended. This is allowed by converting conditional statements into a guard clauses.

The practical reason for this is that it simplifies your reading of the code and reduces the apparent complexity. There is no specific limit for the amount of nested code blocks, however, if you get your code nested too deep, most likely, it must be refactored. In some cases, it’s better to validate all of your input conditions at the beginning of a method and just bail out if anything is not correct. This follows the “fail fast” principle, and you really start to notice the benefit when you have lots of conditions. You can have a series of single-level if-statement checks that check successively more and more specific things, until you’re confident that your inputs are correct. The rest of the method will then be much easier to write and follow, and will tend to have fewer nested conditionals.

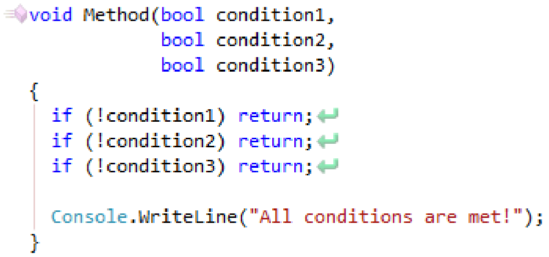
Sample:



```
void Method(bool condition1,
            bool condition2,
            bool condition3)
{
  if (condition1)
  {
    if (condition2)
    {
      if (condition3)
      {
        Console.WriteLine("All conditions are met!");
      }
    }
  }
}
```

How to fix:

Apply the Flatten Conditional refactoring:



```
void Method(bool condition1,
            bool condition2,
            bool condition3)
{
  if (!condition1) return;
  if (!condition2) return;
  if (!condition3) return;

  Console.WriteLine("All conditions are met!");
}
```

Code Issues hints for types and members

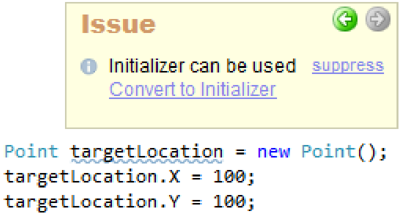
Here are the suggestions (hints) code issues which might improve the readability, clarity and performance of your source code.

Initializer can be used

Cause:

Object initializers allow multiple property values to be set within a single object instance creation operation. They are used in conjunction with existing object constructors. Object initializers provide a simplified syntax for specifying initial values for one or more properties of an object. Using an object initializer may improve code readability.

Sample:



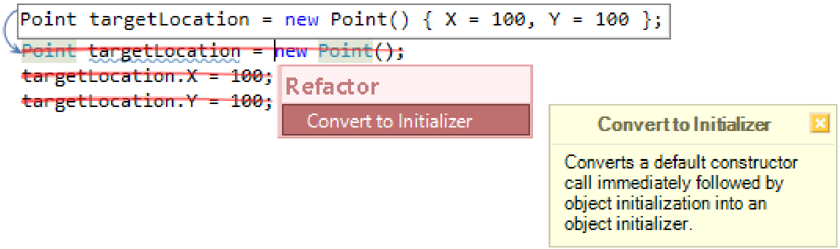
Issue

Initializer can be used

```
Point targetLocation = new Point();
targetLocation.X = 100;
targetLocation.Y = 100;
```

How to fix:

- Apply the **Convert to Initializer** refactoring:



```
Point targetLocation = new Point() { X = 100, Y = 100 };
```

~~Point targetLocation = new Point();~~
~~targetLocation.X = 100;~~
~~targetLocation.Y = 100;~~

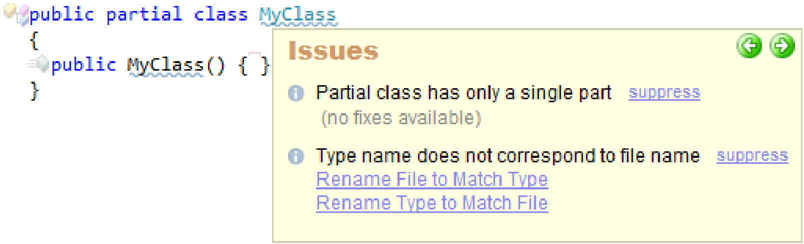
Converts a default constructor call immediately followed by object initialization into an object initializer.

Partial class has only single part

Cause:

A class, structure or interface can be marked as partial to allow its definition to be split between several files. If a type is marked as partial and has only a single part, there is no need to mark it partial. Removing the partial keyword may increase code clarity.

Sample:



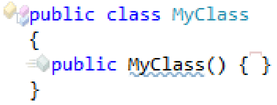
```
public partial class MyClass
{
    public MyClass() { }
}
```

Partial class has only a single part
(no fixes available)

Type name does not correspond to file name

How to fix:

- Remove the ‘partial’ keyword:



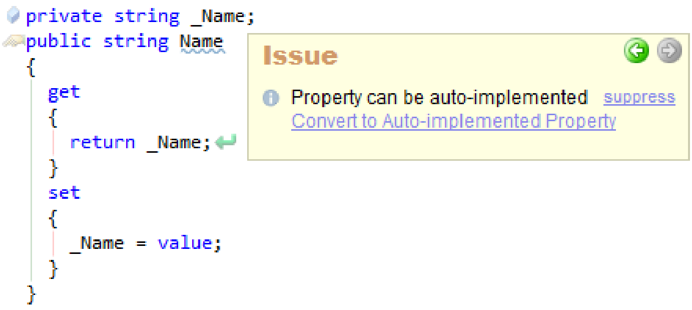
```
public class MyClass
{
    public MyClass() { }
}
```

Property can be auto-implemented

Cause:

If a property uses a simple backing store field and have the corresponding accessors to return the value of the field, and set the value to the field, this property can be simplified to an automatically implemented property. This usual syntax for declaring a read/write property includes a large number of lines of code for a relatively simple task. Converting such a property into an auto-implemented property may increase code readability a lot.

Sample:



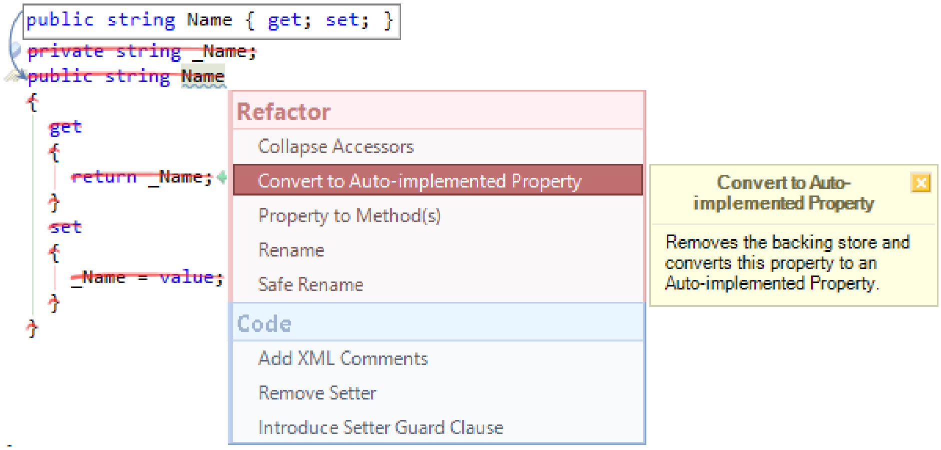
```
private string _Name;
public string Name
{
    get
    {
        return _Name;
    }
    set
    {
        _Name = value;
    }
}
```

Issue

Property can be auto-implemented suppress
Convert to Auto-implemented Property

How to fix:

- Apply the **Convert to Auto-implemented Property** refactoring:



```
public string Name { get; set; }
~~private string _Name;~~
~~public string Name~~
~~{~~
    ~~get~~
    ~~{~~
        ~~return _Name;~~
    ~~}~~
    ~~set~~
    ~~{~~
        ~~_Name = value;~~
    ~~}~~
~~}~~
```

Convert to Auto-implemented Property

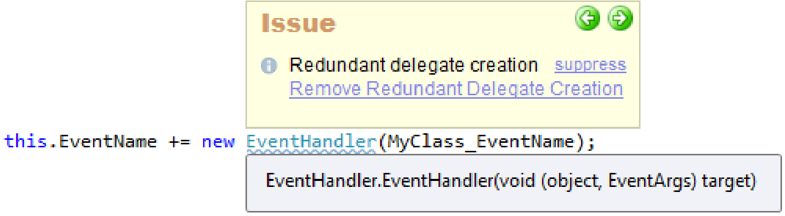
Removes the backing store and converts this property to an Auto-implemented Property.

Redundant delegate creation

Cause:

In some cases, specifying the delegate type explicitly is redundant. For example, it is redundant when you subscribe to an event, because the compiler will infer the type of the delegate automatically from the event handler creation code.

Sample:



Issue

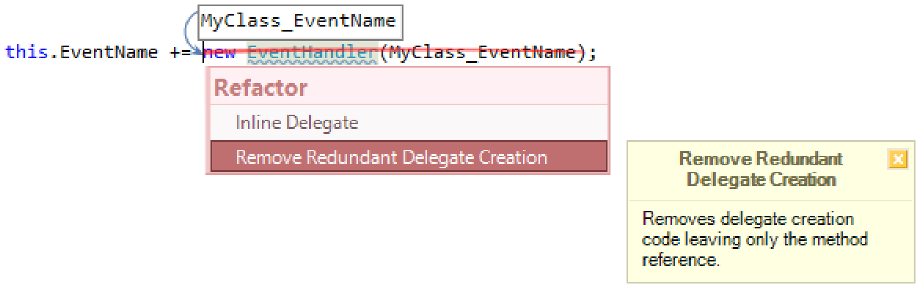
Redundant delegate creation suppress
Remove Redundant Delegate Creation

```
this.EventName += new EventHandler(MyClass_EventName);
```

EventHandler.EventHandler(void (object, EventArgs) target)

How to fix:

- Apply the **Remove Redundant Delegate Creation** refactoring:



```
this.EventName += new EventHandler(MyClass_EventName);
```

Remove Redundant Delegate Creation

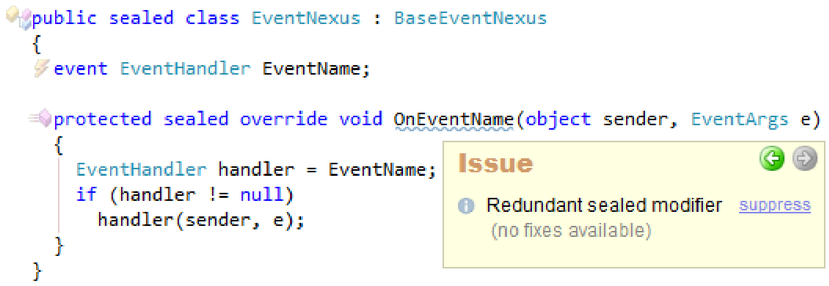
Removes delegate creation code leaving only the method reference.

Redundant sealed modifier

Cause:

If a member marked as ‘sealed’ is located inside the class which is also marked sealed, then there’s no need to mark a member as sealed, because the use of the sealed keyword on a class prevents the class from inheriting functionality for all its members.

Sample:



```
public sealed class EventNexus : BaseEventNexus
  {
   event EventHandler EventName;

   protected sealed override void OnEventName(object sender, EventArgs e)
     {
       EventHandler handler = EventName;
       if (handler != null)
         handler(sender, e);
     }
  }
```

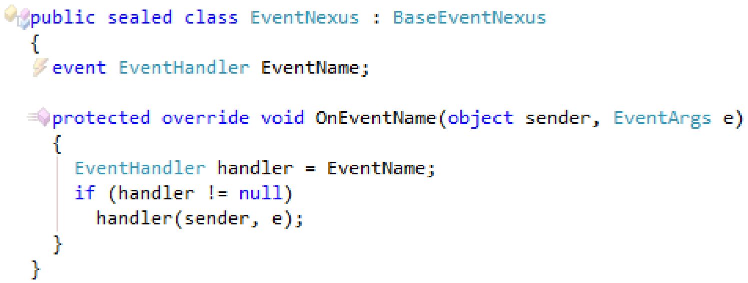
Issue

Redundant sealed modifier

(no fixes available)

How to fix:

- Remove the ‘sealed’ keyword:



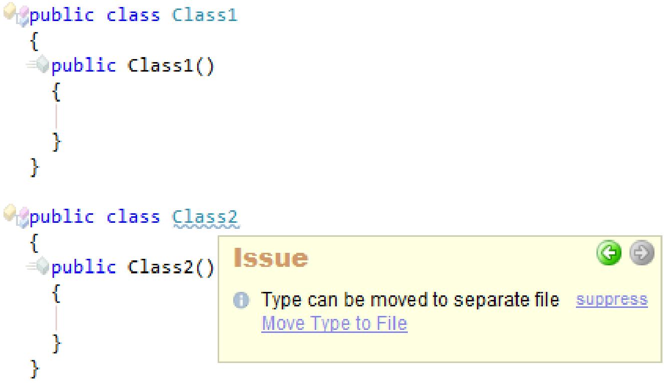
```
public sealed class EventNexus : BaseEventNexus
  {
   event EventHandler EventName;

   protected override void OnEventName(object sender, EventArgs e)
     {
       EventHandler handler = EventName;
       if (handler != null)
         handler(sender, e);
     }
  }
```

Type can be moved to separate file

Cause:

It is recommended to have a single class per source file. If a file contains several neighborhood classes, and a particular class can be moved to its own file (e.g., a type name differs from the file name), a hint is shown indicating that the class can be moved.

Sample:

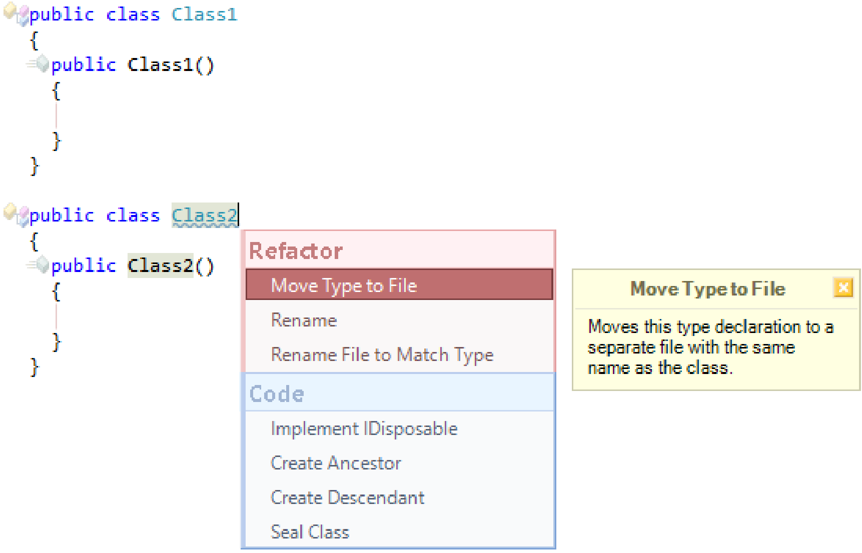


```
public class Class1
{
    public Class1()
    {

    }
}

public class Class2
{
    public Class2()
    {

    }
}
```

Type can be moved to separate file

How to fix:

- Apply the **Move Type to File** refactoring to each class:



```
public class Class1
{
    public Class1()
    {

    }
}

public class Class2
{
    public Class2()
    {

    }
}
```

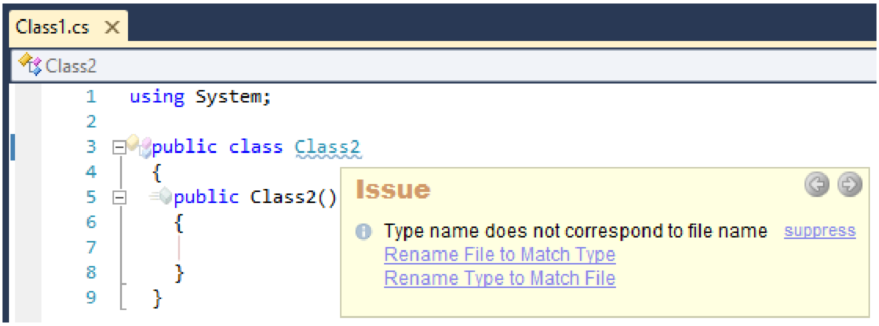
Moves this type declaration to a separate file with the same name as the class.

Type name does not correspond to file name

Cause:

It is also recommended to have a class name that corresponds to the file name, where the class resides. If a class name does not correspond to the file name, the hint is shown suggesting you to rename a file or a class to a single name.

Sample:



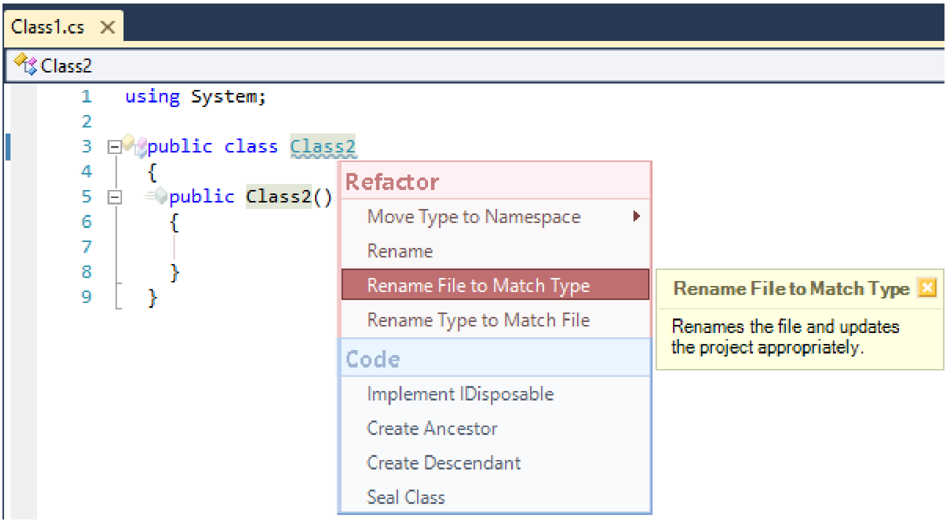
```
using System;

public class Class2
{
    public Class2()
    {

    }
}
```

Issue

Type name does not correspond to file name suppress
Rename File to Match Type
Rename Type to Match File

How to fix:

- Apply the **Rename File to Match Type** refactoring:

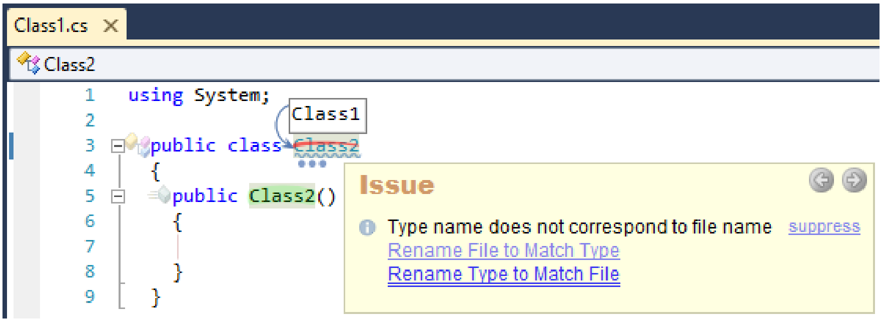


```
using System;

public class Class2
{
    public Class2()
    {

    }
}
```

Rename File to Match Type

Renames the file and updates the project appropriately.

Apply the **Rename Type to Match File** refactoring:



```
using System;

public class Class2
{
    public Class2()
    {

    }
}
```

Issue

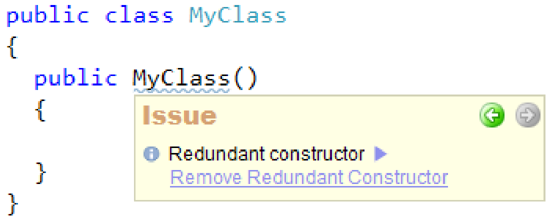
Type name does not correspond to file name suppress
Rename File to Match Type
Rename Type to Match File

Redundant Constructor

Cause:

This code issue shows a hint (suggestion) that a redundant constructor can be safely removed. Redundant constructor is a public, parameterless constructors without any code inside of its body. This constructor doesn't call base type constructors and it is alone in the current class. The constructor is redundant because such constructors are automatically generated by the compiler. Removing the redundant constructors may improve code readability.

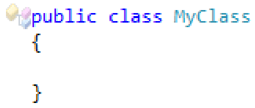
Sample:



```
public class MyClass
{
  public MyClass()
  {

  }
}
```

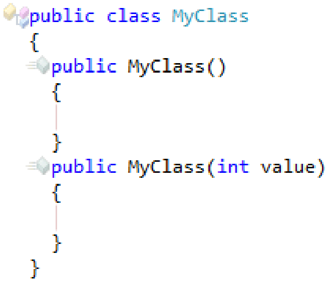
How to fix:

- Remove the redundant constructor by applying the corresponding Remove Redundant Constructor refactoring:



```
public class MyClass
{

}
```

- Add another constructor to the class, so the original constructor is no longer redundant:



```
public class MyClass
{
  public MyClass()
  {

  }
  public MyClass(int value)
  {

  }
}
```

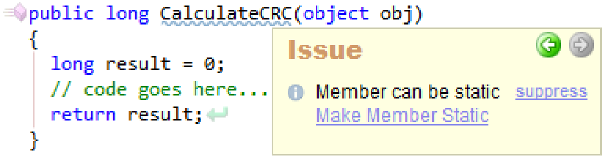
Also, see the full list of code issues specific to constructors.

Member can be static

Cause:

Static members are part of a type and non-static members are part of an instance of that type. If you want to have a shared state or a function between different instances of the same type, a static member will be helpful. This code issue informs you about an instance member that can be converted into a static member.

Sample:

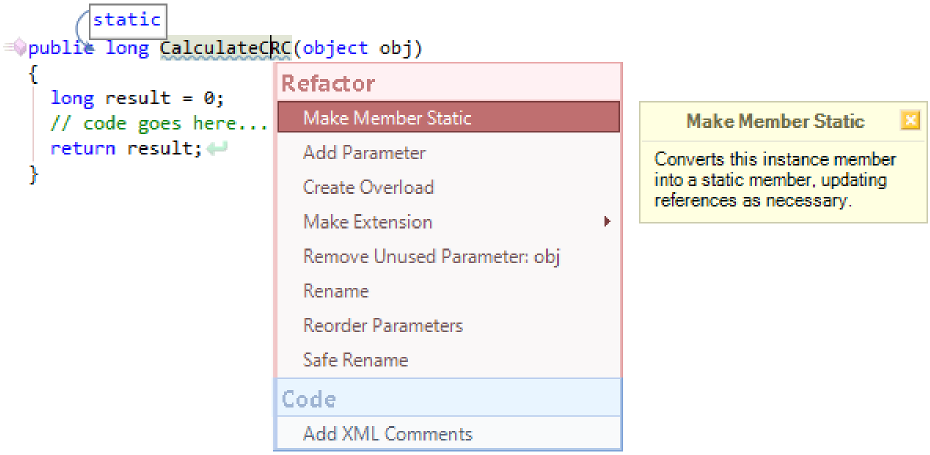


```
public long CalculateCRC(object obj)
{
    long result = 0;
    // code goes here...
    return result;
}
```

Member can be static

How to Fix:

- Apply the Make Member Static refactoring:



```
         static
public long CalculateCRC(object obj)
{
    long result = 0;
    // code goes here...
    return result;
}
```

Converts this instance member into a static member, updating references as necessary.

Can implement base type constructors

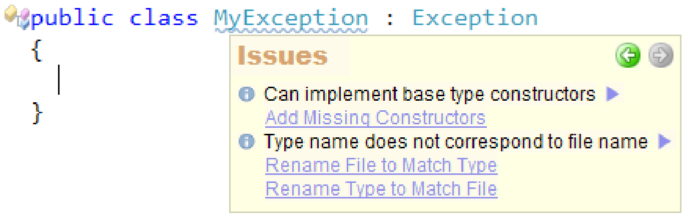
Cause:

The **Can Implement base type constructor** code issue of type hint suggests that you implement the constructors that are missing for the current class, but exist in the ancestor (base) class.

For instance, once you implement your custom exception class derived from the *System.Exception*, you have to provide the full set of its constructors. Failure to provide all of the constructors of the Exception class can make it difficult to correctly handle exceptions. For example, the constructor with the signature "*Exception(string, Exception)"* is used to create exceptions that are caused by other exceptions. Without this constructor, you cannot create and throw an instance of your exception descendant that contains an inner exception, which is what managed code should do in such situations.

The first three constructors of the *Exception* are public by convention. The fourth constructor is protected and implements the custom serialization support, because the *Exception* class implements the *ISerializable* interface.

Sample:



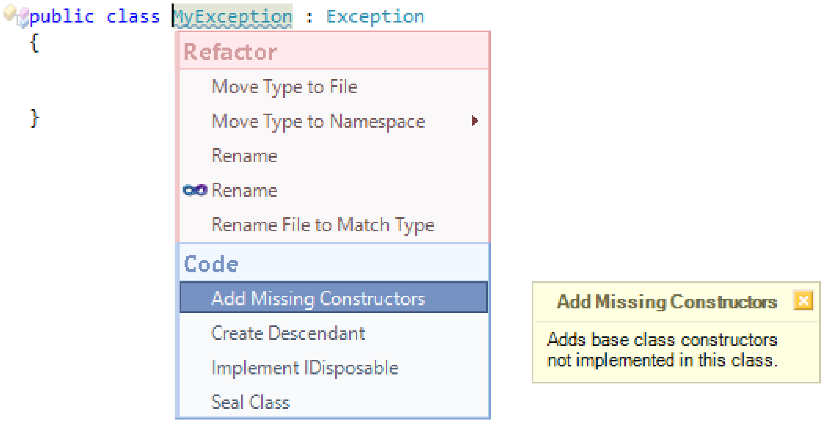
```
public class MyException : Exception
{

}
```

Can implement base type constructors

Type name does not correspond to file name

How to fix:

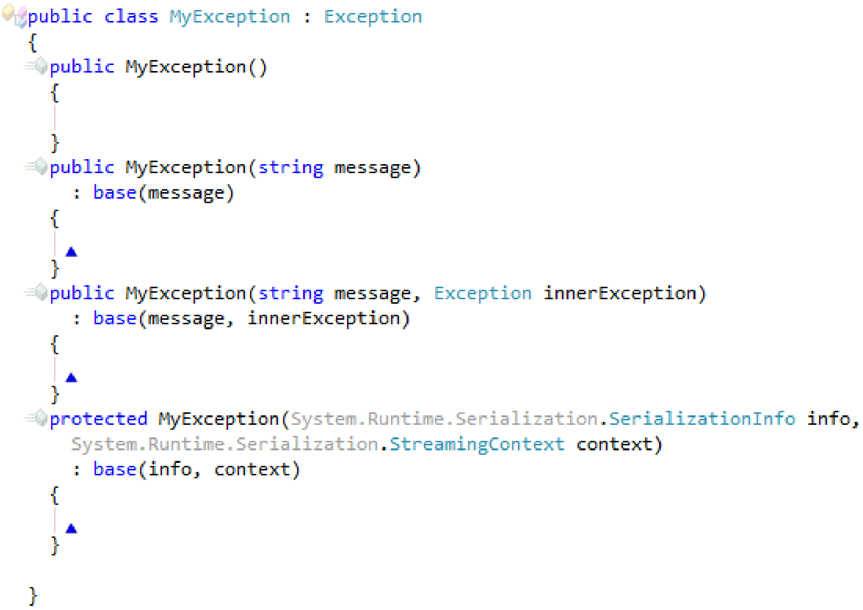
The fix for this code issue is the Add Missing Constructors code provider:



```
public class MyException : Exception
{

}
```

Adds base class constructors not implemented in this class.

After it is performed – it will leave you with all necessary constructors generated automatically:



```
public class MyException : Exception
{
  public MyException()
  {

  }
  public MyException(string message)
    : base(message)
  {

  }
  public MyException(string message, Exception innerException)
    : base(message, innerException)
  {

  }
  protected MyException(System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base(info, context)
  {

  }

}
```

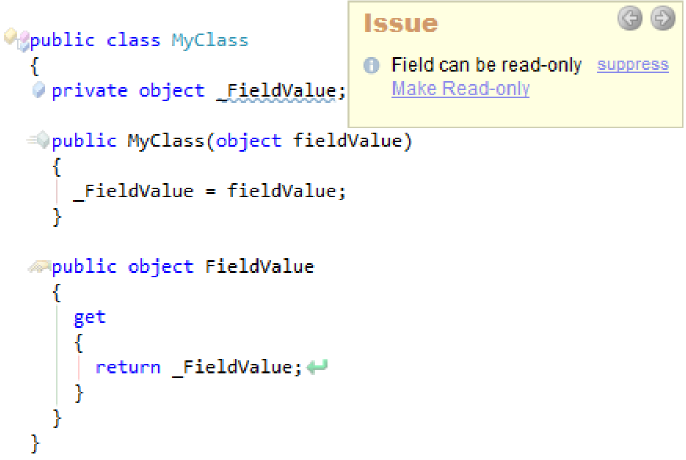
A similar and alternative code issue to this one, is the “Base type constructors are not implemented” code issue, which is not a hint, but an error. Also, see the full list of code issues specific to constructors.

Field can be read-only

Cause:

This code issue shows a hint (suggestion) on the fields which can be marked as read-only. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor (instance or static) in the same class. Read-only fields cannot be changed by other code or the code from team members, thus making it less likely to be messed up by someone who doesn’t understand the purpose of the code.

Sample:



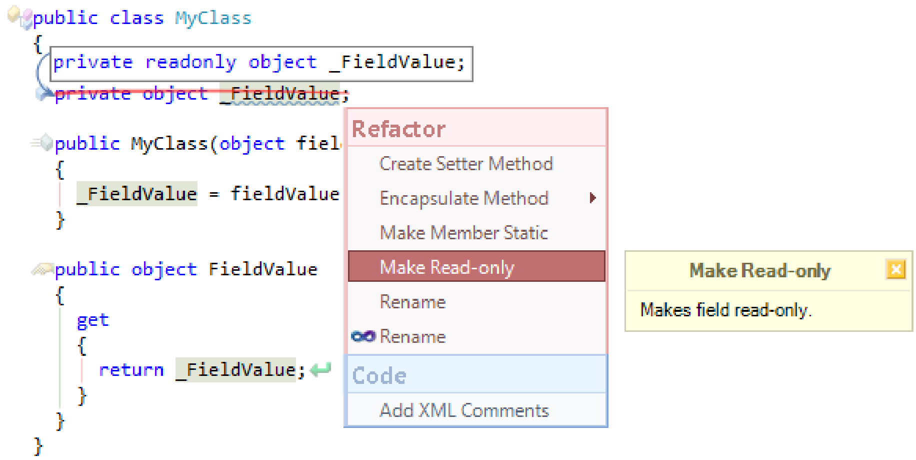
```
public class MyClass
{
  private object _FieldValue;

  public MyClass(object fieldValue)
  {
    _FieldValue = fieldValue;
  }

  public object FieldValue
  {
    get
    {
      return _FieldValue;
    }
  }
}
```

Issue

Field can be read-only suppress
Make Read-only

How to fix:

The **Make Read-only** refactoring is a code fix for this issue:



```
public class MyClass
{
  private readonly object _FieldValue;
  ~~private object _FieldValue;~~

  public MyClass(object fiel
  {
    _FieldValue = fieldValue
  }

  public object FieldValue
  {
    get
    {
      return _FieldValue;
    }
  }
}
```

Make Read-only

Makes field read-only.

Code Issues for anonymous methods and lambda expressions

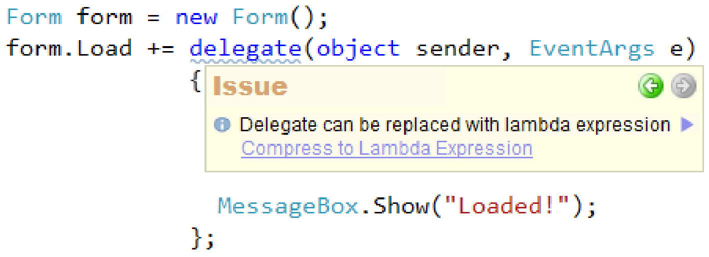
We already reviewed the refactorings for anonymous methods and lambda expressions and know how to deal with these language structures. Now its time to review what Code Issues for anonymous methods and lambda expressions CodeRush Pro provides.

Delegate can be replaced with lambda expression

Cause:

Lambda expressions are simply an improvement to the syntax of anonymous methods. This code issue suggests you that the anonymous method can replaced with a lambda expression.

Sample:



```
Form form = new Form();
form.Load += delegate(object sender, EventArgs e)
             {
               MessageBox.Show("Loaded!");
             };
```

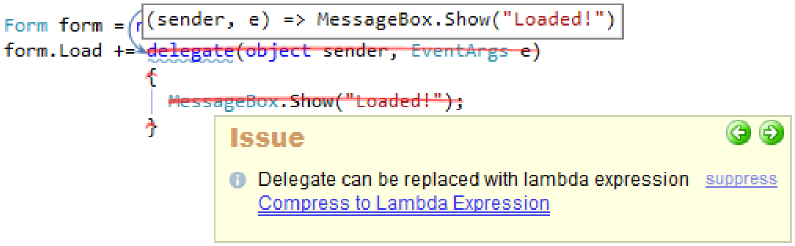
Issue

Delegate can be replaced with lambda expression

Compress to Lambda Expression

How to fix:

- If you prefer lambda expressions, you can convert an anonymous method to a lambda expression by executing the Compress to Lambda Expression code fix for this issue, which may improve code readability:



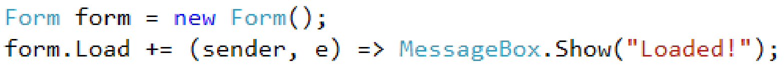
```
Form form = (r(sender, e) => MessageBox.Show("Loaded!")
form.Load += ~~delegate(object sender, EventArgs e)~~
            ~~{~~
              ~~MessageBox.Show("Loaded!");~~
            ~~}~~
```

Issue

Delegate can be replaced with lambda expression suppress

Compress to Lambda Expression

Resulting code:



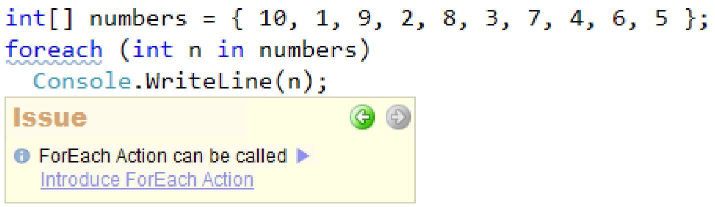
```
Form form = new Form();
form.Load += (sender, e) => MessageBox.Show("Loaded!");
```

ForEach Action can be called

Cause:

This code issue suggests that the code block of the List iteration can be converted to an anonymous method which can be passed as the Action delegate to the List.ForEach method. This may improve code readability.

Sample:



```
int[] numbers = { 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 };
foreach (int n in numbers)
  Console.WriteLine(n);
```

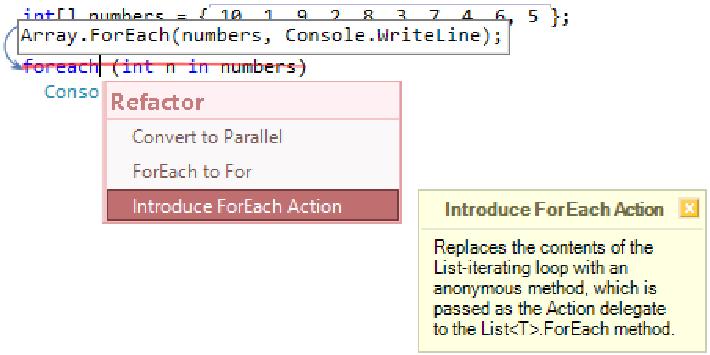
Issue

ForEach Action can be called

Introduce ForEach Action

How to fix:

- Apple the Introduce ForEach Action code fix:

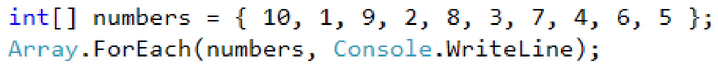


```
int[] numbers = { 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 };
Array.ForEach(numbers, Console.WriteLine);
foreach (int n in numbers)
    Conso
```

Introduce ForEach Action

Replaces the contents of the List-iterating loop with an anonymous method, which is passed as the Action delegate to the List<T>.ForEach method.

Resulting code:



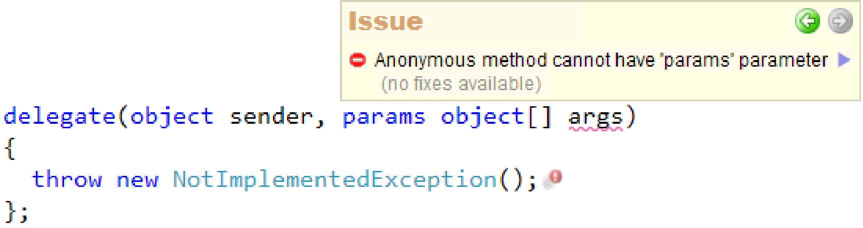
```
int[] numbers = { 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 };
Array.ForEach(numbers, Console.WriteLine);
```

Anonymous method cannot have ‘params’ parameter

Cause:

When the anonymous method contains a ‘params’ parameter, you will see an error in the code editor, because this is a violation of the programming language specification.

Sample:



Issue

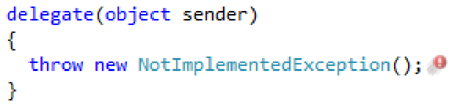
Anonymous method cannot have 'params' parameter

(no fixes available)

```
delegate(object sender, params object[] args)
{
  throw new NotImplementedException();
};
```

How to fix:

- Remove the incorrect ‘params’ parameter:



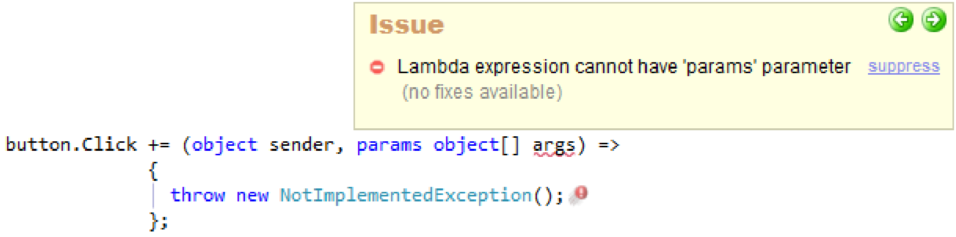
```
delegate(object sender)
{
  throw new NotImplementedException();
}
```

Lambda expression cannot have ‘params’ parameter

Cause:

This is the same language specification violation as the previous code issue. You will see an error, because lambda expressions must have an unchanging number of parameters and cannot have a ‘params’ parameter.

Sample:



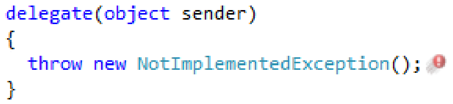
Issue

Lambda expression cannot have 'params' parameter suppress
(no fixes available)

```
button.Click += (object sender, params object[] args) =>
                {
                  throw new NotImplementedException();
                };
```

How to fix:

- Remove the incorrect ‘params’ parameter:



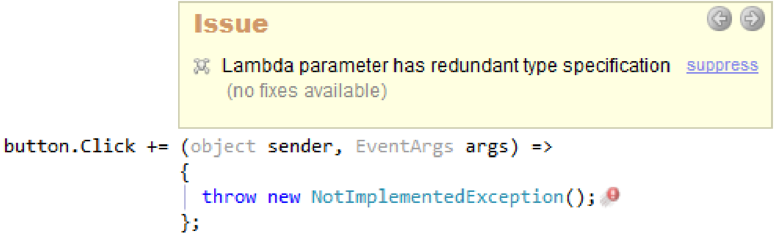
```
delegate(object sender)
{
  throw new NotImplementedException();
}
```

Lambda parameter has redundant type specification

Cause:

The type specifier to the lambda expression parameter is redundant and can be removed. Removing a redundant type specification may improve code readability.

Sample:



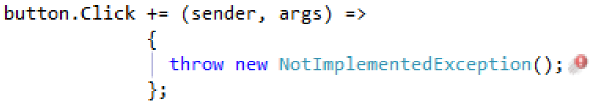
Issue

Lambda parameter has redundant type specification suppress
(no fixes available)

```
button.Click += (object sender, EventArgs args) =>
                {
                  throw new NotImplementedException();
                };
```

How to fix:

- Apply the Remove Redundant Type Specification refactoring:



```
button.Click += (sender, args) =>
                {
                  throw new NotImplementedException();
                };
```

Code Issues of the dead code type for qualifiers

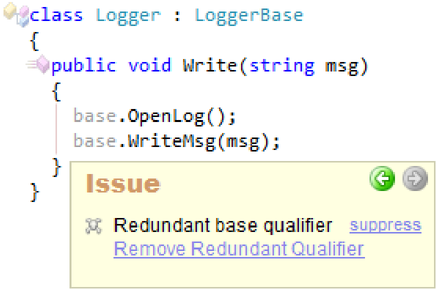
Here are some of the code issues of the dead code type that indicate redundant ‘this’ (‘Me’ in VB), ‘base’ (‘MyBase’ in VB) and type qualifiers.

Redundant base qualifier

Cause:

The only time the base qualifier should be used is when you have an overridden member with the same name in the derived class, but you actually want to call the member in the base class. A keyword like ‘base’ does not make the code readable and should be avoided in all cases unless they are indeed necessary.

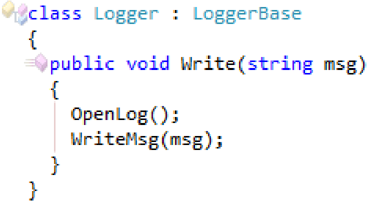
Sample:



```
class Logger : LoggerBase
{
  public void Write(string msg)
  {
    base.OpenLog();
    base.WriteMsg(msg);
  }
}
```

How to fix:

- Apply the **Remove Redundant Qualifier** code fix:



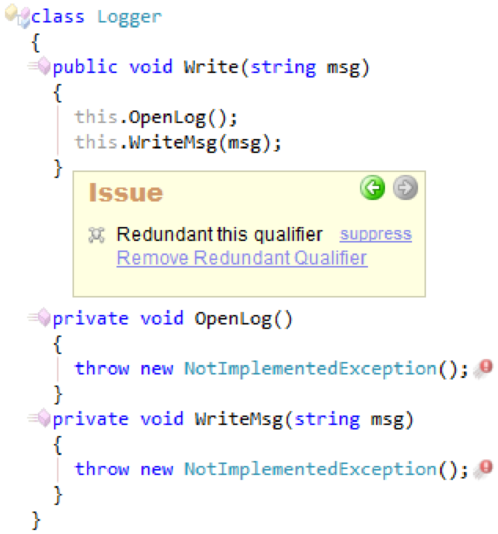
```
class Logger : LoggerBase
{
  public void Write(string msg)
  {
    OpenLog();
    WriteMsg(msg);
  }
}
```

Redundant this qualifier

Cause:

The ‘this’ keyword refers to the current instance of the class. The keyword is most of the time technically redundant, because a member is unique in most scenarios. However, there are situations where you must use the ‘this’ keyword, for example to prevent an ambiguity between a member and a function parameter that has the same name as a member. Another important thing to consider is that the ‘this’ keyword is removed by the compiler. So, it is actually down to a personal preference or the coding standard in your team.

Sample:



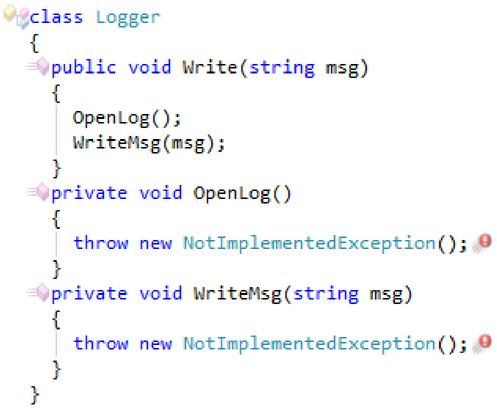
```
class Logger
{
    public void Write(string msg)
    {
        this.OpenLog();
        this.WriteMsg(msg);
    }
    private void OpenLog()
    {
        throw new NotImplementedException();
    }
    private void WriteMsg(string msg)
    {
        throw new NotImplementedException();
    }
}
```

Issue

Redundant this qualifier

How to fix:

- Apply the **Remove Redundant Qualifier** code fix:



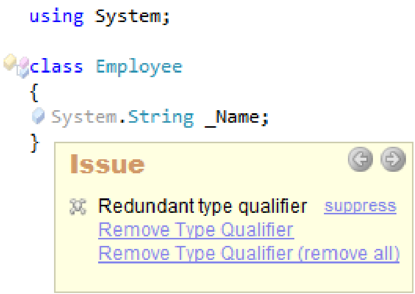
```
class Logger
{
    public void Write(string msg)
    {
        OpenLog();
        WriteMsg(msg);
    }
    private void OpenLog()
    {
        throw new NotImplementedException();
    }
    private void WriteMsg(string msg)
    {
        throw new NotImplementedException();
    }
}
```

Redundant type qualifier

Cause:

Namespaces qualifiers prevent conflicts between the names of classes by isolating the contents of each namespace. A full namespace name qualifier is not required for a class when a corresponding using (or Imports in VB) statement is declared. You can safely remove the namespace qualifier to improve code readability.

Sample:



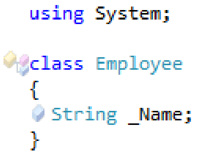
```
using System;

class Employee
{
System.String _Name;
}
```

Issue

Redundant type qualifier suppress
Remove Type Qualifier
Remove Type Qualifier (remove all)

How to fix:

- Apply the **Remove Type Qualifier** or **Remove Type Qualifier (remove all)** code fixes:



```
using System;

class Employee
{
String _Name;
}
```

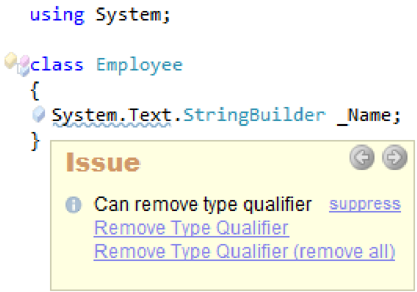
Can remove type qualifier

Cause:

This code issue is similar to the '**Redundant type qualifier**' code issue. The difference is that this one is shown even when no using statement is declared to the namespace qualifier. Removing a namespace qualifier may increase code readability.

Note that the code issue is not shown if the new using statement with the namespace qualifier will lead to the types ambiguity error.

Sample:



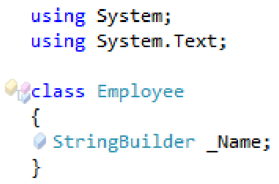
```
using System;

class Employee
{
System.Text.StringBuilder _Name;
}
```

Issue

Can remove type qualifier suppress
Remove Type Qualifier
Remove Type Qualifier (remove all)

How to fix:

- Apply the **Remove Type Qualifier** or **Remove Type Qualifier (remove all)** code fixes:



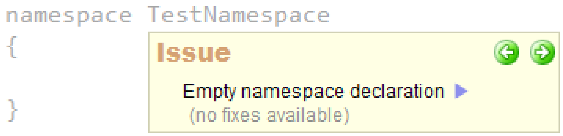
```
using System;
using System.Text;

class Employee
{
  StringBuilder _Name;
}
```

Code Issues – Empty namespace declaration

Cause:

This code issue shows a *dead code*, which is redundant. Empty namespace declarations don't have any value, and can be safely removed.

Sample:



```
namespace TestNamespace
{

}
```

Issue

Empty namespace declaration
(no fixes available)

How to fix:

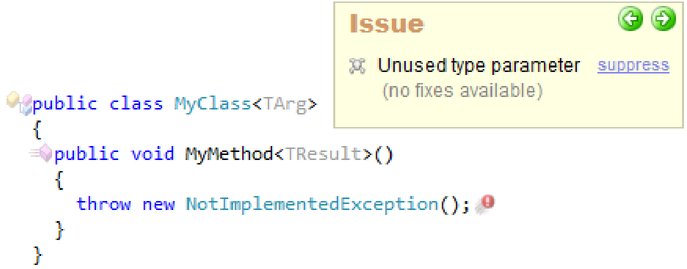
- Remove the empty namespace declaration.

Code Issues – Unused type parameter

Cause:

This code issue of a dead code type shows type parameters to a generic type, or method definitions that are not refere enced within its scope, and can be removed. In a generic type or method definition, a type parameter is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type. Removing the unused type parameter may improve readability.

Sample:



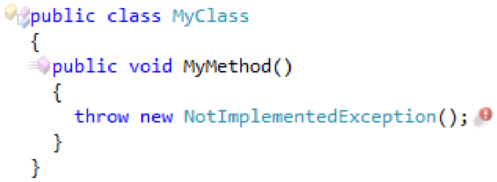
Issue

Unused type parameter suppress
(no fixes available)

```
public class MyClass<TArg>
{
  public void MyMethod<TResult>()
  {
    throw new NotImplementedException();
  }
}
```

How to fix:

- Remove the unused type parameters:



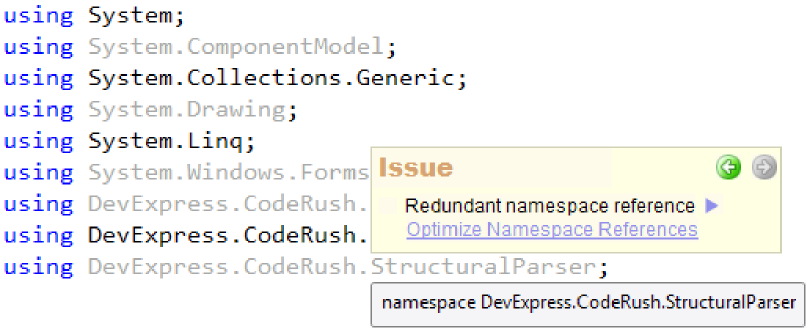
```
public class MyClass
{
  public void MyMethod()
  {
    throw new NotImplementedException();
  }
}
```

Redundant namespace reference

Cause:

The **Redundant namespace reference** code issue highlights unused namespace references that can be safely removed in gray (dead code issue), which may improve readability.

Sample:

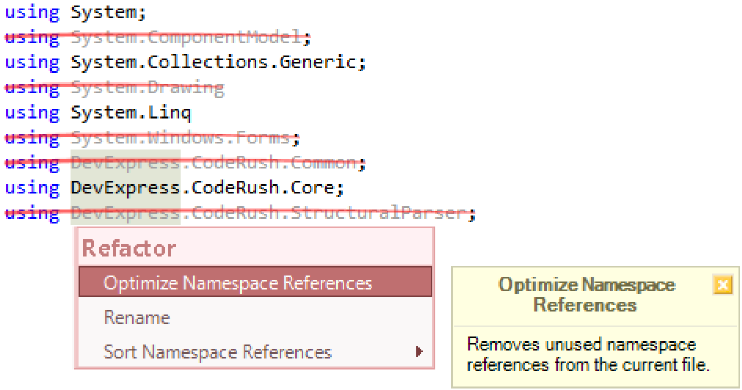


```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms
using DevExpress.CodeRush.
using DevExpress.CodeRush.
using DevExpress.CodeRush.StructuralParser;
```

namespace DevExpress.CodeRush.StructuralParser

How to fix:

- Apply the Optimize Namespace References refactoring:



```
using System;
~~using System.ComponentModel;~~
using System.Collections.Generic;
~~using System.Drawing~~
using System.Linq
~~using System.Windows.Forms;~~
~~using DevExpress.CodeRush.Common;~~
using DevExpress.CodeRush.Core;
~~using DevExpress.CodeRush.StructuralParser;~~
```

Optimize Namespace References

Removes unused namespace references from the current file.

Options of the refactoring are also used for the **Redundant namespace reference** code issue. For example, you can specify a list of references that won't have an influence on the code issue. In other words, if you specify a reference that should be kept (and that is actually unused) and there are no more unused references in the file – a code issue won't be shown.

Code Issues of the dead code type for members and blocks

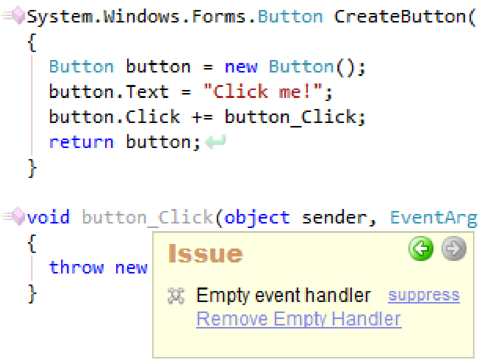
Here are several of the code issues, which highlight redundant and unnecessary code blocks that may be safely removed for improving code readability.

Empty event handler

Cause:

An event is the mechanism for a class that notifies its clients when something happens. Events provide add additional methods (event handlers) to be triggered upon an external event. As an empty event handler does nothing, there is no need for it. An empty handler can be safely removed, which will improve code readability.

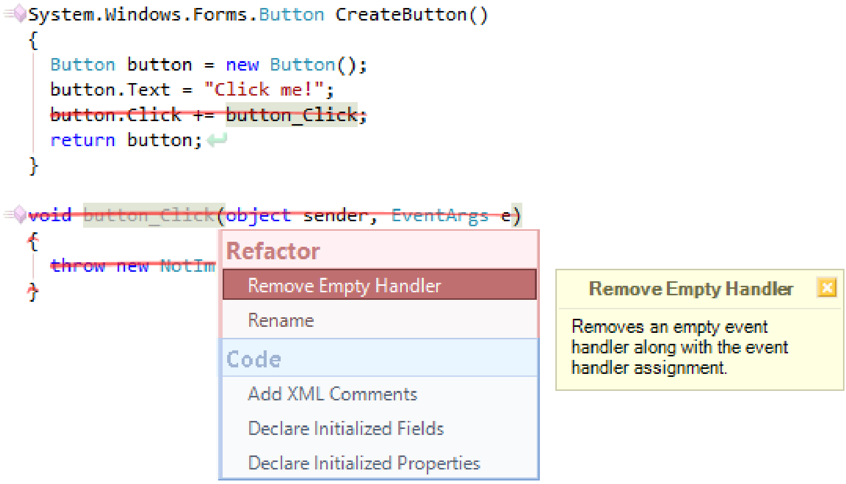
Sample:



```
System.Windows.Forms.Button CreateButton()
{
  Button button = new Button();
  button.Text = "Click me!";
  button.Click += button_Click;
  return button;
}

void button_Click(object sender, EventArgs e)
{
  throw new
}
```

How to fix:

- Apply the **Remove Empty Handler** code provider:



```
System.Windows.Forms.Button CreateButton()
{
  Button button = new Button();
  button.Text = "Click me!";
  ~~button.Click += button_Click;~~
  return button;
}

~~void button_Click(object sender, EventArgs e)~~
~~{~~
  ~~throw new NotIm~~
~~}~~
```

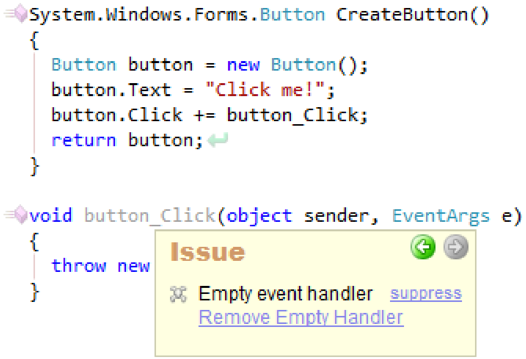
Removes an empty event handler along with the event handler assignment.

Empty finally block

Cause:

A code inside a finally block is executed when a control leaves a try statement and is guaranteed to be executed no matter what happens, including exceptions and return statements. An empty finally block does not make sense, because nothing will happen when it receives a control. Thus, an empty finally block is redundant and can be safely removed, which will improve code readability.

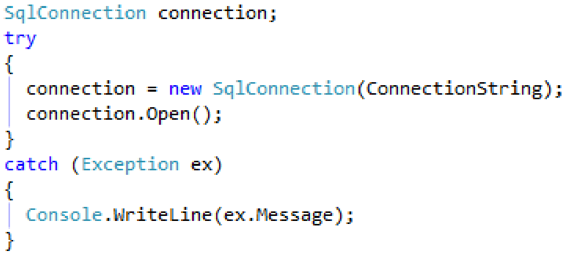
Sample:



```
System.Windows.Forms.Button CreateButton()
{
    Button button = new Button();
    button.Text = "Click me!";
    button.Click += button_Click;
    return button;
}

void button_Click(object sender, EventArgs e)
{
    throw new
}
```

How to fix:

- Remove an empty finally block (including a try block if there is no catch block):



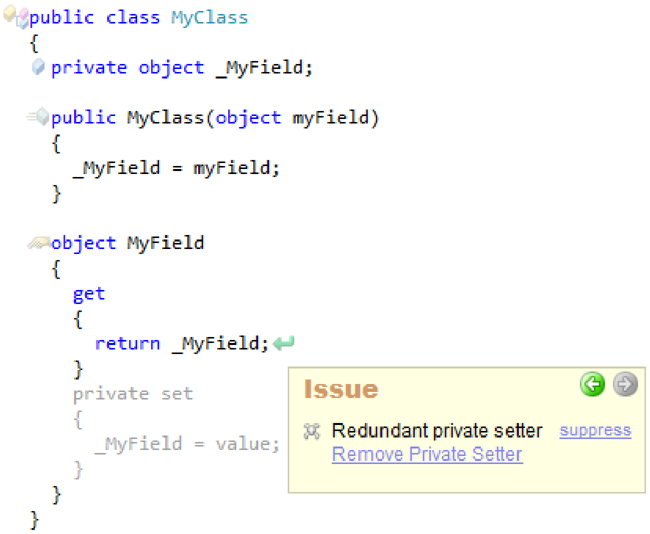
```
SqlConnection connection;
try
{
    connection = new SqlConnection(ConnectionString);
    connection.Open();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Redundant private setter

Cause:

You cannot modify the value of a backing field inside the private setter outside of the current class. Also, if there are no assignments to the property in the current class, it doesn't make sense to declare a private setter at all. In this case, you can safely remove the private setter to improve code readability.

Sample:



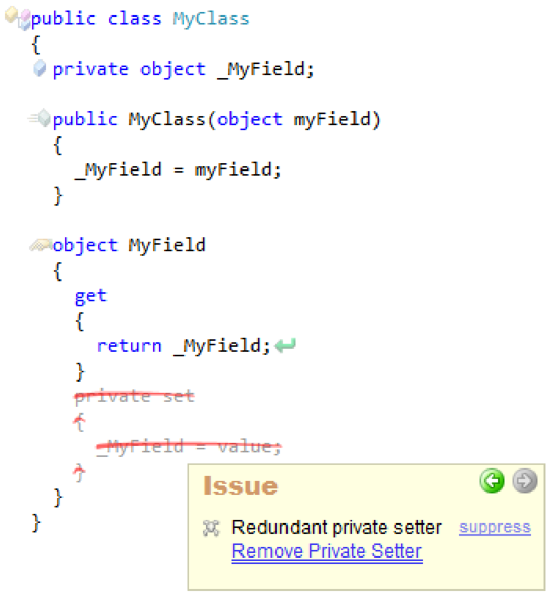
```
public class MyClass
{
  private object _MyField;

  public MyClass(object myField)
  {
    _MyField = myField;
  }

  object MyField
  {
    get
    {
      return _MyField;
    }
    private set
    {
      _MyField = value;
    }
  }
}
```

Issue

Redundant private setter

How to fix:

- Apply the **Remove Private Setter** refactoring:



```
public class MyClass
{
  private object _MyField;

  public MyClass(object myField)
  {
    _MyField = myField;
  }

  object MyField
  {
    get
    {
      return _MyField;
    }
    ~~private set~~
    ~~{~~
      ~~_MyField = value;~~
    ~~}~~
  }
}
```

Issue

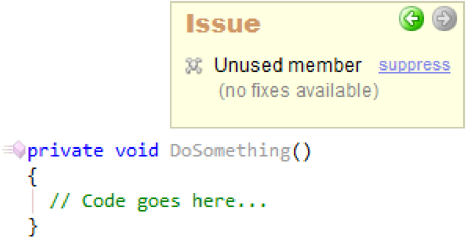
Redundant private setter

Unused member

Cause:

A member which is not referenced anywhere (unused) in the code is useless and can be safely removed to clean up the code and improve readability.

Sample:



Unused member
(no fixes available)

```
private void DoSomething()
{
    // Code goes here...
}
```

How to fix:

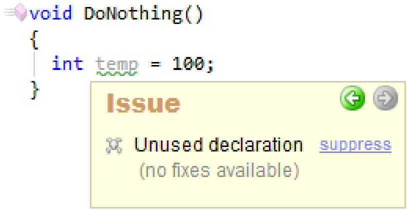
- Remove the unused member.

Unused declaration

Cause:

Similar to the **Unused Member** code issue, this code issue highlights variables that are not used in the code. Removing such variables will improve code readability.

Sample:



```
void DoNothing()
{
    int temp = 100;
}
```

Unused declaration
(no fixes available)

How to fix:

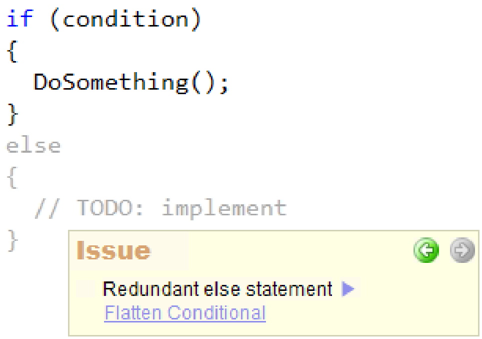
- Remove the variable its assignments, if any.

Redundant else statement

Cause:

The else statement is redundant when it doesn’t contain any code, and can be safely removed to improve readability. The type of code issue is a dead code.

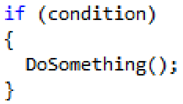
Sample:



```
if (condition)
{
  DoSomething();
}
else
{
  // TODO: implement
}
```

How to fix:

- Apply the Flatten Conditional refactoring:



```
if (condition)
{
  DoSomething();
}
```

Redundant field initialization

Cause:

The **redundant field initialization** code issue of type ‘dead code‘ highlights unecessary initialization of fields that can be safely removed (because the common language runtime initializes all fields to their default values, regardless of their type). Value types are initialized to 0 and reference types are initialized to null. Explicitly initializing a field to its default value is redundant, degrades performance and adds to maintenance costs.

When you expressly initialize your fields, the compiler creates code to set the value of the fields and includes that code as part of your object initialization. When you instantiate an object using the new keyword, the following sequence happens:

1. Your fields are allocated

2. The fields are initialized by the CLR

3. The code for the object constructor is performed

The problem is that if you explicitly initialized your fields, the constructor code will initialize your variables **twice**. This is not a big deal in most cases, but if those objects are instantiated many times, the performance impact may be noticeable.

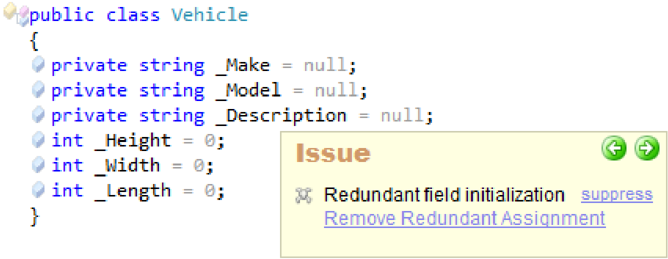
Performing these kinds of tests when fields are initialized explicitly, shows the following result:

- Creating an object and not initializing fields ~1013 ms (100%)

- Creating an object and explictly initializing fields ~1124 ms (110%)

As you can see, there may be a noticeable performance hit.

Sample:



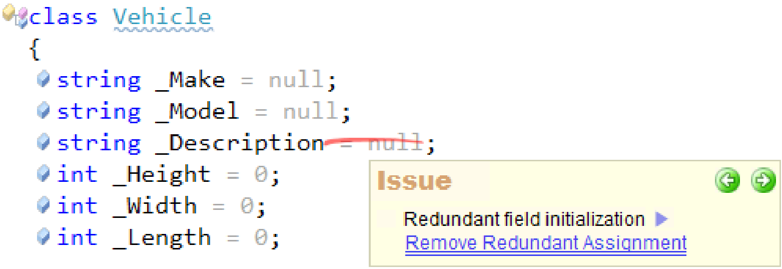
```
public class Vehicle
{
    private string _Make = null;
    private string _Model = null;
    private string _Description = null;
    int _Height = 0;
    int _Width = 0;
    int _Length = 0;
}
```

Issue

Redundant field initialization suppress
Remove Redundant Assignment

How to fix:

- Apply the **Remove Redundant Assignment** code fix:



```
class Vehicle
{
    string _Make = null;
    string _Model = null;
    string _Description = null;
    int _Height = 0;
    int _Width = 0;
    int _Length = 0;
```

Issue

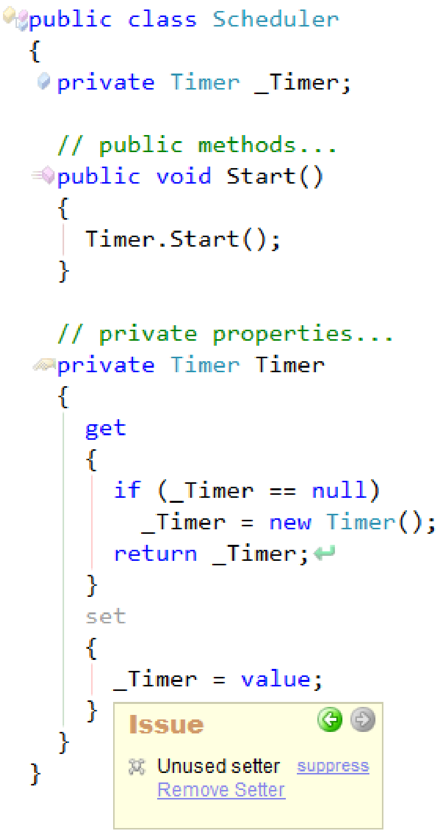
Redundant field initialization
Remove Redundant Assignment

Unused setter

Cause:

This code issue shows a dead code when there are unreferenced private property setters. Unused property setters of private properties can be safely removed, which will improve code readability.

Sample:



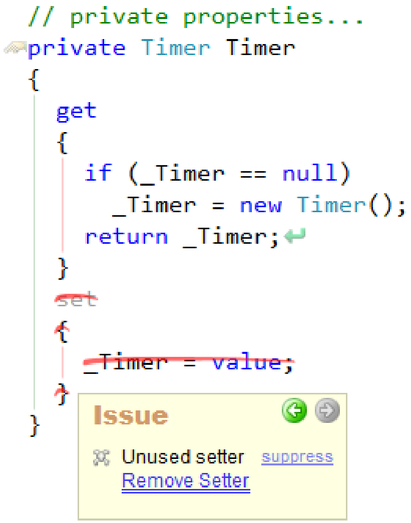
```
public class Scheduler
{
  private Timer _Timer;

  // public methods...
  public void Start()
  {
    Timer.Start();
  }

  // private properties...
  private Timer Timer
  {
    get
    {
      if (_Timer == null)
        _Timer = new Timer();
      return _Timer;
    }
    set
    {
      _Timer = value;
    }
  }
}
```

Issue

Unused setter

How to fix:

- Apply the **Remove Setter** code fix:



```
// private properties...
private Timer Timer
{
  get
  {
    if (_Timer == null)
      _Timer = new Timer();
    return _Timer;
  }
  ~~set~~
  {
    ~~_Timer = value;~~
  }
}
```

Issue

Unused setter

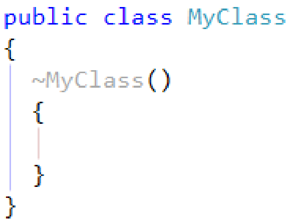
Redundant Destructor

Cause:

Destructors are invoked automatically, and used to destroy instances of classes. The developer has no control over when the destructor is called, because this is determined by the garbage collector. Redundant destructor is a an empty destructor without any code inside, that can be safely removed.

Empty destructors should not be used at all. If a class contains a destructor, an entry is created in the “Finalize queue”, controlled by the garbage collector. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty, this just causes a needless loss of performance.

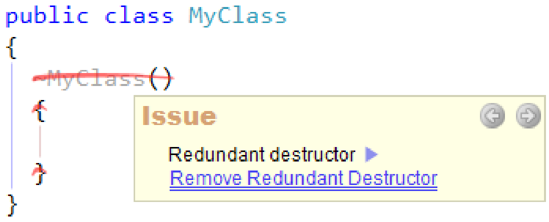
Sample:



```
public class MyClass
{
  ~MyClass()
  {

  }
}
```

How to fix:

- Apply the **Remove Redundant Destructor** refactoring:



```
public class MyClass
{
  ~~~MyClass()~~
  {

  }
}
```

Code Issues for Switch (Select) and Case statements

A switch statement executes logic, dependent on the value of a given expression (parameter). The types of values a switch statement operates on can be boolean, enum, integral types, and strings. Each switch statement can contain any number of case statements, but no two case constants within the same switch statement can have the same value. Each case statement defines a value to compare against the original expression. You may also include a default label following all other case statements. If none of the other choices match, then the default choice is taken and its statements are executed. If there is no default label, control is transferred outside the switch.

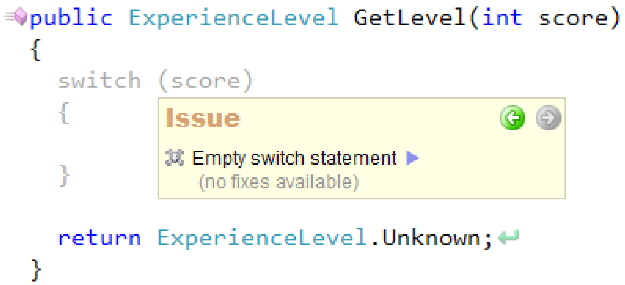
CodeRush Pro provides several code issues specific to a switch statement (C#) or Select statement (VB) as well as its nested case statements. These code issues are of type warning, dead code and code smell.

Empty case statement

Cause:

This code issue is shown when the Switch or Select statement is empty and can be safely removed. If no code is executed inside the case statement, it is redundant.

Sample:



```
public ExperienceLevel GetLevel(int score)
{
  switch (score)
  {

  }

  return ExperienceLevel.Unknown;
}
```

Issue

Empty switch statement
(no fixes available)

How to fix:

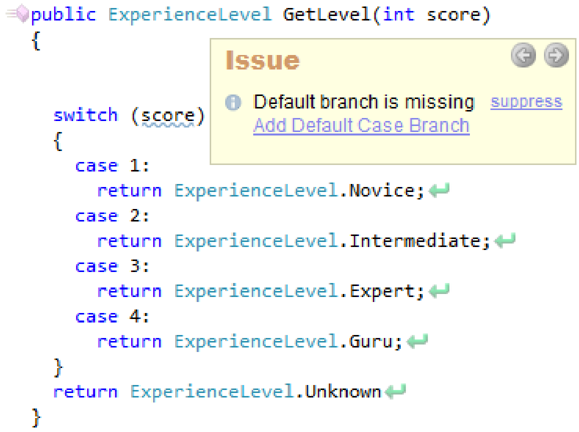
- Remove the empty case statement.

Default branch is missing

Cause:

This is a code issue of the warning type, no matter that the default label is optional. It is shown when the switch or select statement is missing a default label which may be unintentional.

Sample:



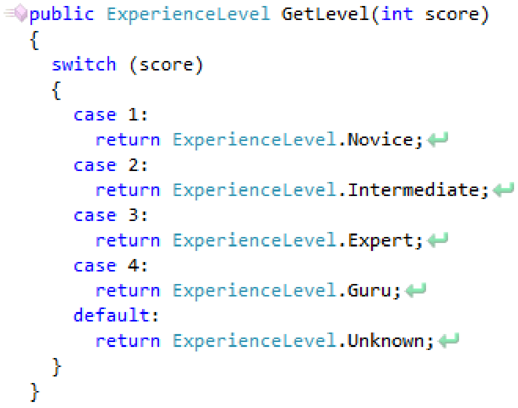
```
public ExperienceLevel GetLevel(int score)
{


  switch (score)
  {
    case 1:
      return ExperienceLevel.Novice;
    case 2:
      return ExperienceLevel.Intermediate;
    case 3:
      return ExperienceLevel.Expert;
    case 4:
      return ExperienceLevel.Guru;
  }
  return ExperienceLevel.Unknown
}
```

Issue

Default branch is missing suppress
Add Default Case Branch

How to fix:

- Apply the Add Default Case Branch code fix:



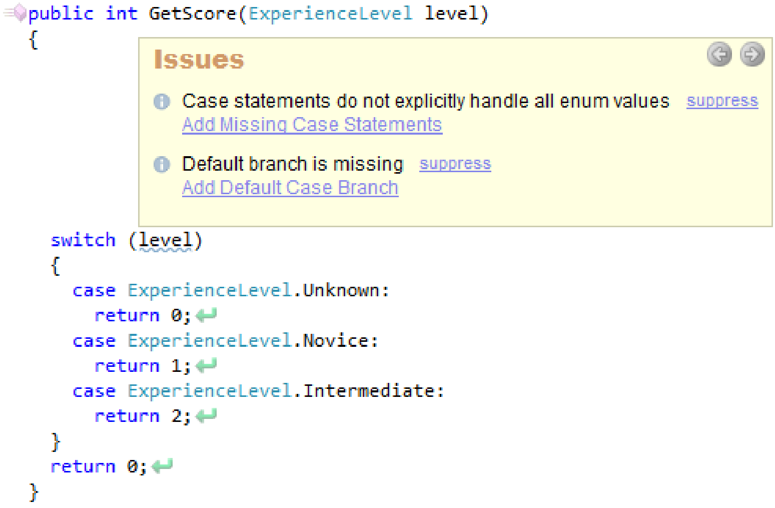
```
public ExperienceLevel GetLevel(int score)
{
  switch (score)
  {
    case 1:
      return ExperienceLevel.Novice;
    case 2:
      return ExperienceLevel.Intermediate;
    case 3:
      return ExperienceLevel.Expert;
    case 4:
      return ExperienceLevel.Guru;
    default:
      return ExperienceLevel.Unknown;
  }
}
```

Case statements do not explicitly handle all enum values

Cause:

An expression of a case statement can be of the enumeration type. Enumeration types have a definite number of enum elements. If the switch or select statement handles only a subset of the possible enumeration values it’s checking for, the code issue will be shown because this may be a sign of incomplete code.

Sample:



Issues

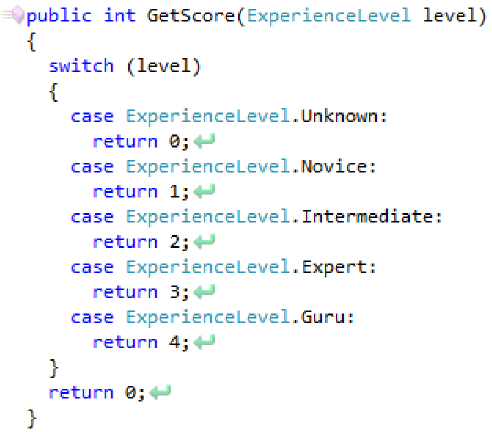
Case statements do not explicitly handle all enum values suppress
Add Missing Case Statements

Default branch is missing suppress
Add Default Case Branch

```
public int GetScore(ExperienceLevel level)
{
  switch (level)
  {
    case ExperienceLevel.Unknown:
      return 0;
    case ExperienceLevel.Novice:
      return 1;
    case ExperienceLevel.Intermediate:
      return 2;
  }
  return 0;
}
```

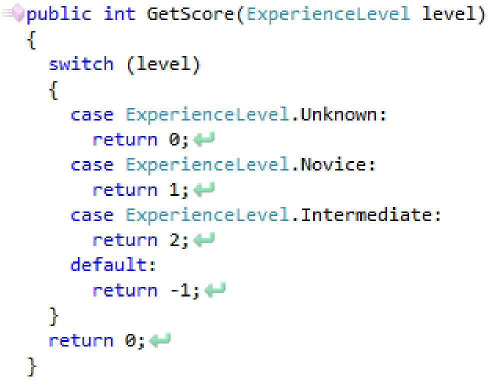
How to fix:

- Apply the Add Missing Case Statements code fix:



```
public int GetScore(ExperienceLevel level)
{
  switch (level)
  {
    case ExperienceLevel.Unknown:
      return 0;
    case ExperienceLevel.Novice:
      return 1;
    case ExperienceLevel.Intermediate:
      return 2;
    case ExperienceLevel.Expert:
      return 3;
    case ExperienceLevel.Guru:
      return 4;
  }
  return 0;
}
```

- Add a default branch:



```
public int GetScore(ExperienceLevel level)
{
  switch (level)
  {
    case ExperienceLevel.Unknown:
      return 0;
    case ExperienceLevel.Novice:
      return 1;
    case ExperienceLevel.Intermediate:
      return 2;
    default:
      return -1;
  }
  return 0;
}
```

Case statement has incorrect range of integers expression

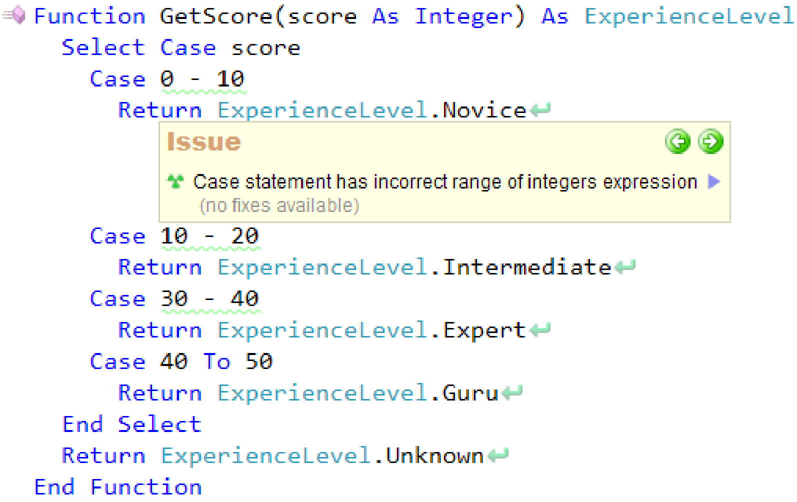
Cause:

This code issue is useful for the Select statement in Visual Basic, because case statements in VB can handle a specific range of values, not only a single constant value. The range of the values is defined according to the language specification using the “To” operator as follows:

“Case 20 To 30″.

However, if the range expression is specified incorrectly, the code issue of the code smell type is shown.

Sample:



```
Function GetScore(score As Integer) As ExperienceLevel
  Select Case score
    Case 0 - 10
      Return ExperienceLevel.Novice
    Case 10 - 20
      Return ExperienceLevel.Intermediate
    Case 30 - 40
      Return ExperienceLevel.Expert
    Case 40 To 50
      Return ExperienceLevel.Guru
  End Select
  Return ExperienceLevel.Unknown
End Function
```

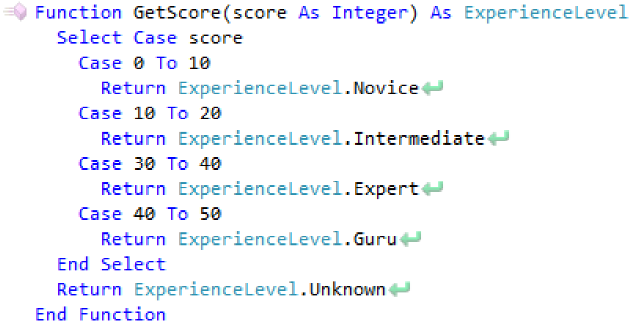
Issue

Case statement has incorrect range of integers expression
(no fixes available)

The case statement in the sample actually specifies a single value "-10″, because the expression range is treated as a mathematical minus operation, in other words: 20 – 30 = -10.

How to fix:

- Replace the incorrect integer expressions with the "To" operator:



```
Function GetScore(score As Integer) As ExperienceLevel
  Select Case score
    Case 0 To 10
      Return ExperienceLevel.Novice
    Case 10 To 20
      Return ExperienceLevel.Intermediate
    Case 30 To 40
      Return ExperienceLevel.Expert
    Case 40 To 50
      Return ExperienceLevel.Guru
  End Select
  Return ExperienceLevel.Unknown
End Function
```

Warning code issues for the IDisposable pattern

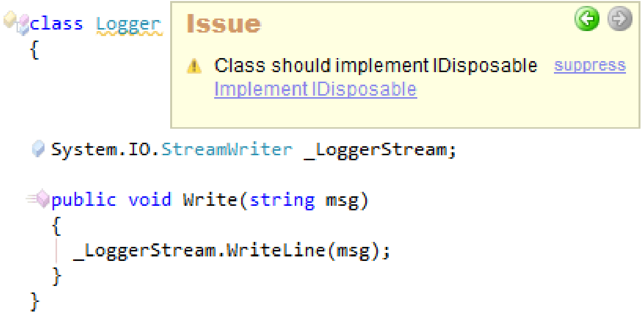
Here are two code issues of the warning type that might be helpful in detecting objects of a class that have not undiseposed. Such objects may lead to temporary unmanaged resource leaks.

Class should implement IDisposable

Cause:

This issue appears when a class contains fields that implement IDisposable, but the class itself does NOT implement IDisposable.

Sample:



Class should implement IDisposable

```
class Logger
{

    System.IO.StreamWriter _LoggerStream;

    public void Write(string msg)
    {
        _LoggerStream.WriteLine(msg);
    }
}
```

How to fix:

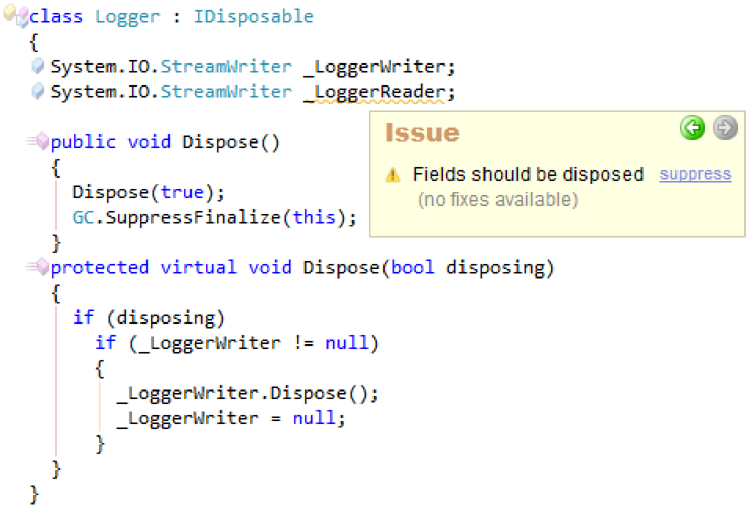
- Apply the Implement IDisposable code fix.

Fields should be disposed

Cause:

This issue appears when fields implementing IDisposable are inside a class that also implements IDisposable, but those fields are NOT disposed.

Sample:

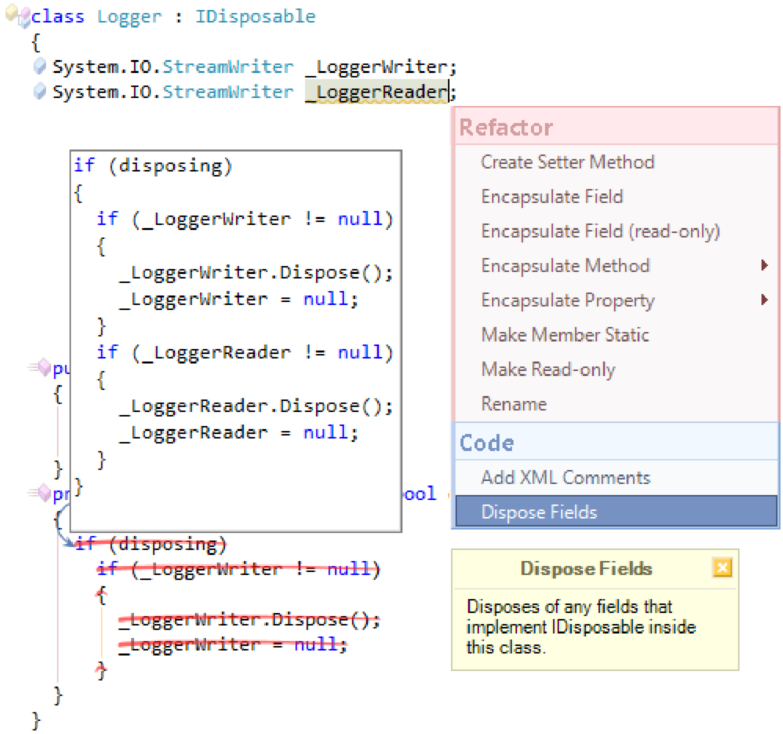


Fields should be disposed
(no fixes available)

```
class Logger : IDisposable
{
    System.IO.StreamWriter _LoggerWriter;
    System.IO.StreamWriter _LoggerReader;

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
            if (_LoggerWriter != null)
            {
                _LoggerWriter.Dispose();
                _LoggerWriter = null;
            }
    }
}
```

How to fix:

- Apply the Dispose Fields code fix:



```
class Logger : IDisposable
{
  System.IO.StreamWriter _LoggerWriter;
  System.IO.StreamWriter _LoggerReader;
```

```
if (disposing)
{
  if (_LoggerWriter != null)
  {
    _LoggerWriter.Dispose();
    _LoggerWriter = null;
  }
  if (_LoggerReader != null)
  {
    _LoggerReader.Dispose();
    _LoggerReader = null;
  }
}
```

```
  pu
  {
  }
  pr                                    ool
  {
  ~~if (disposing)~~
    ~~if (_LoggerWriter != null)~~
    {
      ~~_LoggerWriter.Dispose();~~
      ~~_LoggerWriter = null;~~
    }
  }
}
```

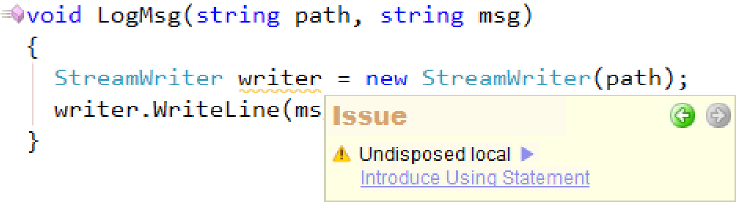
Dispose Fields

Disposes of any fields that implement IDisposable inside this class.

Undisposed local

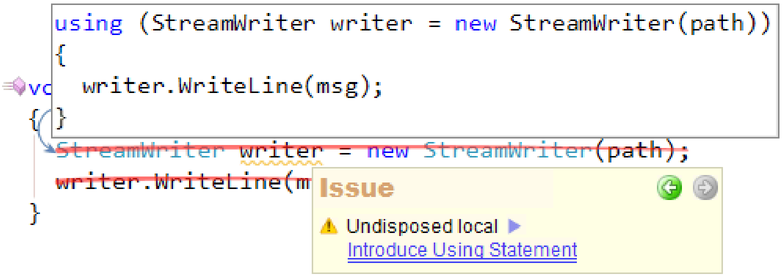
The **Undisposed local** code issue of the warning type highlights local variables that implement the *System.IDisposable* interface and are not explicitly disposed. The *IDisposable* interface was designed to provide a standard way to release unmanaged resources by calling its *Dispose* method. If the object is*IDisposable*, it is a good idea to dispose of it when you no longer need it, especially if the object uses unmanaged resources. These are resources that the .NET garbage collector does not manage on our behalf and is unable to clean-up automatically. They include items such as streams, files, database connections, handles and other operating system objects. If the memory and system resources that they use are not properly released, a program may suffer from memory leaks or problems due to locked resources.

The *Stream* (from *System.IO*) and *Font* (from *System.Drawing*) classes are examples of managed types that access unmanaged resources (in this case file handles and device contexts). If you hold a *FileStream* object, you should dispose this object when releasing resources, because this may not be done immediately by the garbage collector. Calling the *Dispose* method ensures that the unmanaged file handle controlled by the FileStream object is released as early as possible. The writer *FileStream* object is not explicitly disposed, so you can see the **Undisposed local** code issue:



```
void LogMsg(string path, string msg)
{
  StreamWriter writer = new StreamWriter(path);
  writer.WriteLine(ms
}
```

Apply the Introduce Using Statement code provider, to fix it:



```
using (StreamWriter writer = new StreamWriter(path))
{
  writer.WriteLine(msg);
}
```

vo

{

~~StreamWriter writer = new StreamWriter(path);~~

~~writer.WriteLine(m~~

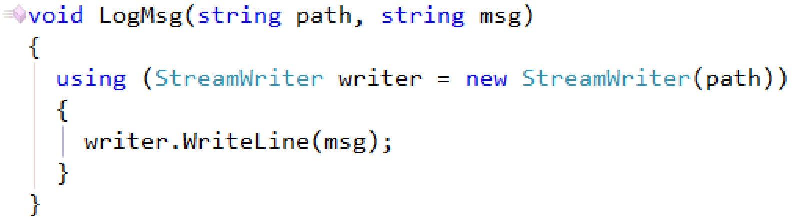
}

Issue

Undisposed local

Introduce Using Statement

The code fix will create a using statement, which ensures that *Dispose* is called even if an exception occurs while you are calling methods on the object:



```
void LogMsg(string path, string msg)
{
  using (StreamWriter writer = new StreamWriter(path))
  {
    writer.WriteLine(msg);
  }
}
```

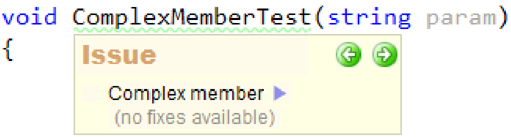
Complex Member

The **Complex Member** code issue of the smell type highlights complex members. A complex member is a member that may have too much code inside. The issue is based on a calculation of the Maintenance Complexity code metric. **Maintenance complexity** is a measure the structural complexity of a node (and its children), and represents how easy or challenging a method will be to understand and maintain. Scores closer to zero are simple. Small methods usually score below ١٥٠, while large/complex methods will exceed ٥٠٠.

Here's the table of the ideal scores for this code metric:

MC Score	Interpretation
<= 100	Simple method. Easy to maintain.
101-200	Medium method. Relatively easy to maintain.
201-300	Large method. A little more challenging to maintain.
300-600	Complex method. Strong candidate for refactoring.
601-1000	Very complex method, challenging to maintain. Should be broken down.
1001+	Ultra-complex method. Extremely challenging to maintain. High priority for simplification.

When a member's maintenance complexity exceeds a threshold of 800 points, you will see the member highlighted:



```
void ComplexMemberTest(string param)
{
```

Issue

Complex member

(no fixes available)

How to fix:

It is recommended to break down a complex member into smaller methods, using the Extract Method refactoring, for example.

Code Issues specific to declaring types

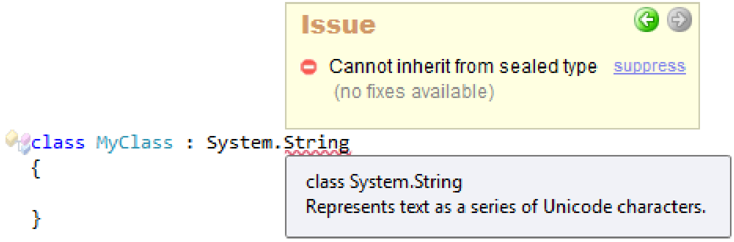
The CSharp and Visual Basic programming language specifications have several restrictions on how type elements such as a class, structure, and delegate are declared in the code. Let's review code issues that demonstrate an incorrect type declaration, and see the error before the compiler informs us about it when we build the code.

Cannot inherit from sealed type

Cause:

The whole point of sealed classes is that you are not supposed to inherit from them. That is why you are not allowed to create descendants of a sealed class.

Sample:



Issue

Cannot inherit from sealed type suppress
(no fixes available)

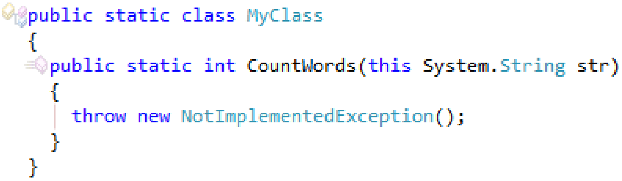
```
class MyClass : System.String
{

}
```

class System.String
Represents text as a series of Unicode characters.

How to fix:

- Remove the 'sealed' ('NotInheritable' in VB) keyword from the ancestor class if appropriate.

- If you inherit the new class to extend it with additional members, or you do not have access to the source declaration of the sealed class, you can use the extensions methods instead:



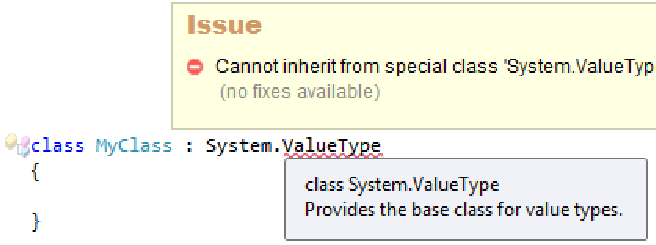
```
public static class MyClass
{
    public static int CountWords(this System.String str)
    {
        throw new NotImplementedException();
    }
}
```

Cannot inherit from special class System.ValueType

Cause:

The System.ValueType provides the base class for value types (structures). Because structures are implicitly sealed, they cannot be inherited. Although ValueType is the implicit base class for value types, you cannot create a class that inherits from ValueType directly.

Sample:



Issue

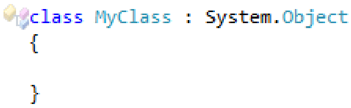
Cannot inherit from special class 'System.ValueType' suppress
(no fixes available)

```
class MyClass : System.ValueType
{

}
```

class System.ValueType
Provides the base class for value types.

How to fix:

- Inherit a descendant of a different non-sealed type rather than the ValueType type:



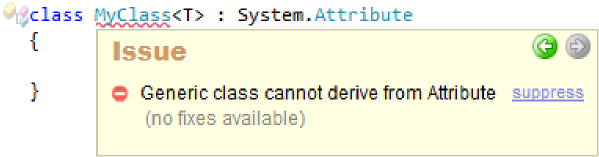
```
class MyClass : System.Object
{

}
```

Generic class cannot derive from Attribute

Cause:

An attribute decorates a class when compiled, but a generic class does not receive its final type information until runtime. Since the attribute can affect compilation, it has to be completed at compile time, so a generic class can not inherit from the Attribute class.

Sample:



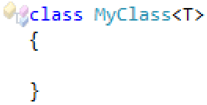
```
class MyClass<T> : System.Attribute
{

}
```

Issue

Generic class cannot derive from Attribute suppress
(no fixes available)

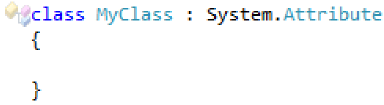
How to fix:

- Remove the attribute from the current class declaration:



```
class MyClass<T>
{

}
```

- Do not mark the current class as generic:



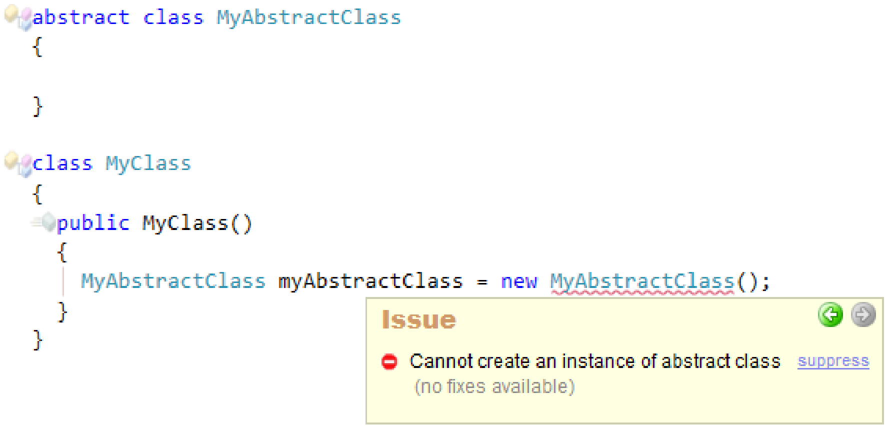
```
class MyClass : System.Attribute
{

}
```

Cannot create an instance of abstract class

Cause:

The abstract classes may contain abstract members that are incomplete and must be implemented in a derived class. The issue occurs because we have not overridden the abstract member to provide their implementation, so we can not instantiate an instance of an abstract class.

Sample:



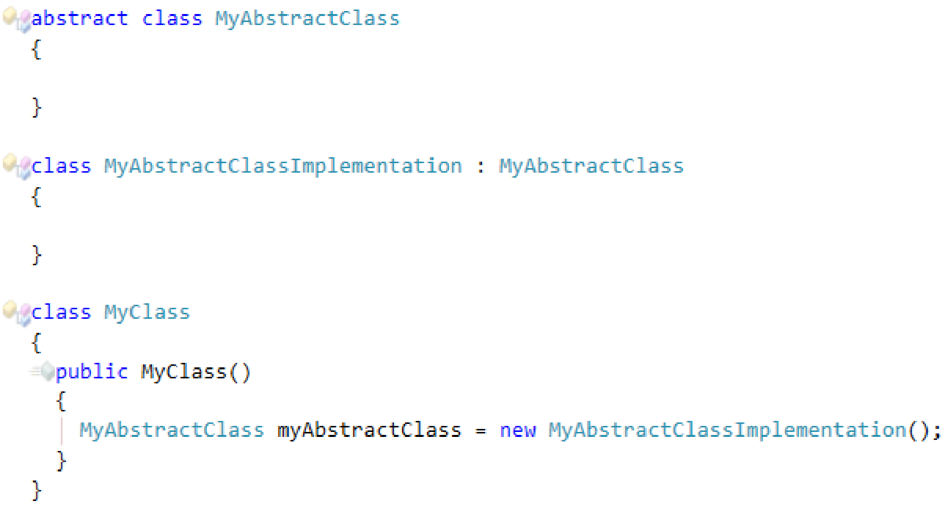
```
abstract class MyAbstractClass
{

}

class MyClass
{
  public MyClass()
  {
    MyAbstractClass myAbstractClass = new MyAbstractClass();
  }
}
```

Issue

Cannot create an instance of abstract class suppress
(no fixes available)

How to fix:

- Create a descendant implementation of the abstract class and instantiate it instead of the abstract class:



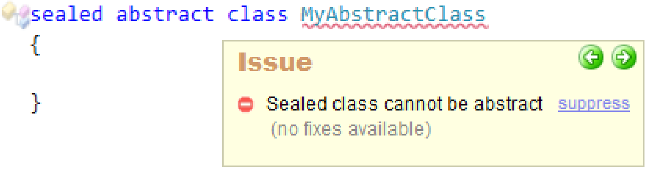
```
abstract class MyAbstractClass
{

}

class MyAbstractClassImplementation : MyAbstractClass
{

}

class MyClass
{
  public MyClass()
  {
    MyAbstractClass myAbstractClass = new MyAbstractClassImplementation();
  }
}
```

Sealed class cannot be abstract

Cause:

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. But, as we just learned, we can not derive from sealed classes, so a class can not be marked as sealed and abstract at once.

Sample:



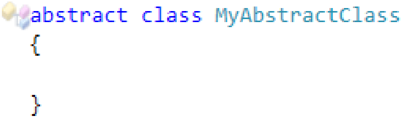
```
sealed abstract class MyAbstractClass
{

}
```

Issue

Sealed class cannot be abstract suppress
(no fixes available)

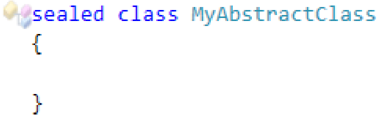
How to fix:

- Remove the ‘sealed’ modifier if a class can be used as an ancestor for derived classes:



```
abstract class MyAbstractClass
{

}
```

- Remove the ‘abstract’ modifier if a class does not contain abstract members:



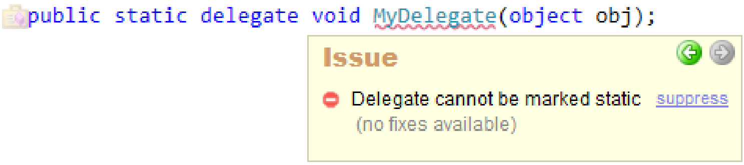
```
sealed class MyAbstractClass
{

}
```

Delegate cannot be marked static

Cause:

A delegate declaration is a type declaration that basically declares a method signature. That is why it doesn’t make sense to make it neither static nor an instance.

Sample:



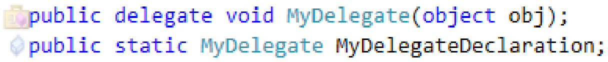
```
public static delegate void MyDelegate(object obj);
```

Issue

Delegate cannot be marked static suppress
(no fixes available)

How to fix:

- Remove the static keyword from a delegate declaration and add a static field instead, if required:



```
public delegate void MyDelegate(object obj);
public static MyDelegate MyDelegateDeclaration;
```

See also the code issues for declaring variables and constants.

Code Issues specific for static classes

A static class can be used as a unit of organization for sets of utility functions that operate on input parameters and do not have to get or set any internal data. These functions can be accessed without creating an instance of the class. In this case, a static class can make your implementation simpler and faster because you do not have to create an object in order to call its methods.

A good example of a static class is the *System.Math* class, which provides constants and static methods for common mathematical functions.

You can use a static class as a convenient container for methods not associated with particular objects. A member of such classes can be used to separate data and behavior that is independent of any object identity: the data and functions do not change regardless of what happens to the object.

Having said that, there are some requirements necessary in order to declare static classes. CodeRush suggests several code issues for correct code writing when static classes declaration requirements are violated. These code issues allow you to see an error due to declaring a static class before compiling the code.

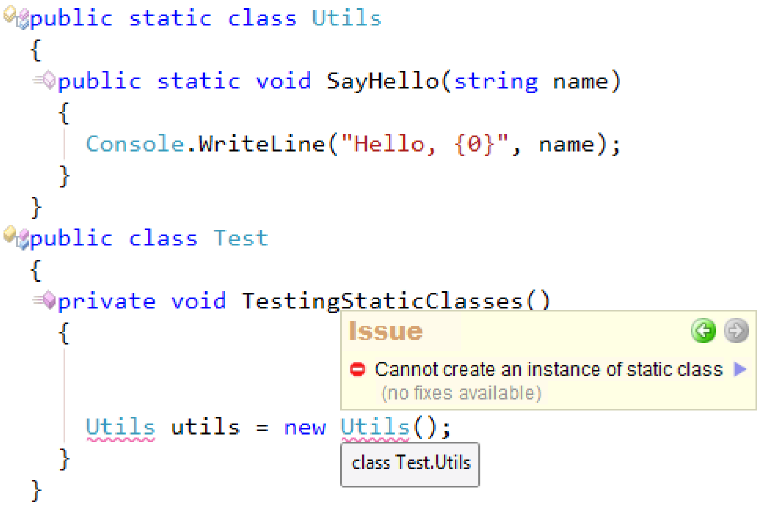
Here they are:

Cannot create an instance of static class

Cause:

It is not possible to create instances of a static class using the ‘new’ keyword. Static classes are loaded automatically by the .NET Framework Common Language Runtime when the program or namespace containing the class is loaded. This code issue has an error type, because your code won’t compile.

Sample:



```
public static class Utils
{
  public static void SayHello(string name)
  {
    Console.WriteLine("Hello, {0}", name);
  }
}
public class Test
{
  private void TestingStaticClasses()
  {


    Utils utils = new Utils();
  }
}
```

Issue

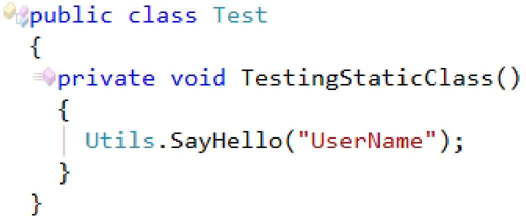
Cannot create an instance of static class

(no fixes available)

class Test.Utils

How to fix:

- Use the static class name instead, and access its member directly without initiating it:



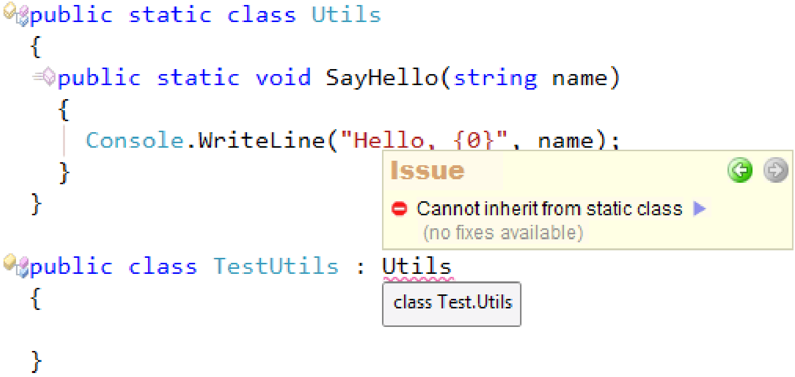
```
public class Test
{
  private void TestingStaticClass()
  {
    Utils.SayHello("UserName");
  }
}
```

Cannot inherit from static class

Cause:

Static classes are designed in a way that prevents inheriting from any class except *System.Object*. A static class implicitly inherits from type *Object*, you can't cannot explicitly specify a base class or a list of implemented interfaces. Static classes are sealed by nature and therefore cannot be inherited.

Sample:



```
public static class Utils
{
  public static void SayHello(string name)
  {
    Console.WriteLine("Hello, {0}", name);
  }
}

public class TestUtils : Utils
{

}
```

Issue

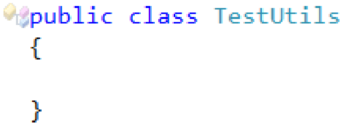
Cannot inherit from static class

(no fixes available)

class Test.Utils

How to fix:

- Do not derive a class from the static class:



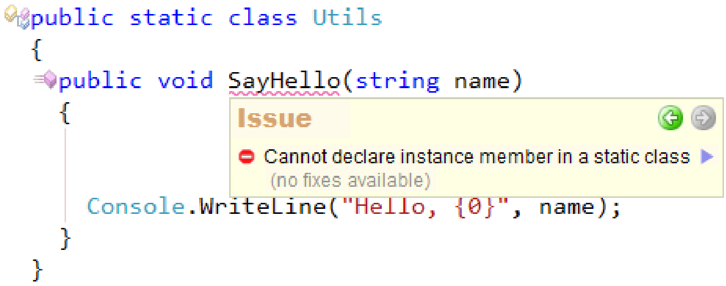
```
public class TestUtils
{

}
```

Cannot declare instance member in a static class

Cause:

When a class declared as a static, it is indicating that it contains only static members, so it cannot declare instance members.

Sample:



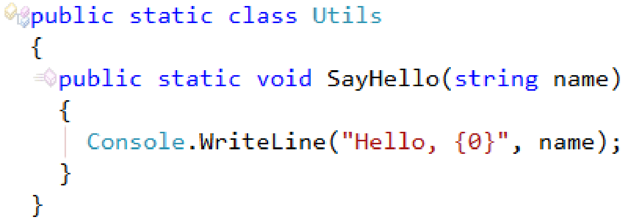
```
public static class Utils
  {
  public void SayHello(string name)
    {

      Console.WriteLine("Hello, {0}", name);
    }
  }
```

Issue

Cannot declare instance member in a static class
(no fixes available)

How to fix:

- Mark the member as static:



```
public static class Utils
  {
  public static void SayHello(string name)
    {
      Console.WriteLine("Hello, {0}", name);
    }
  }
```

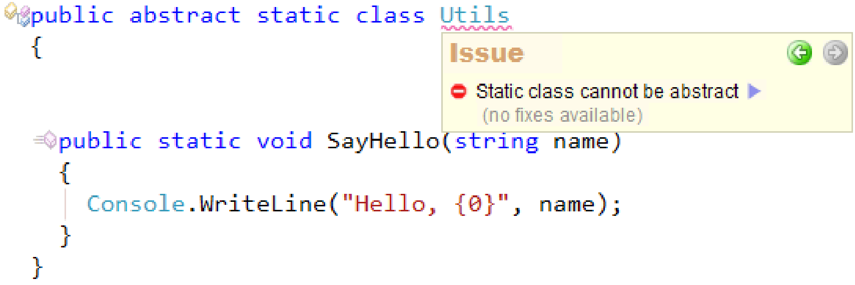
Static class cannot be abstract and Static class cannot be sealed

Cause:

A static class may not include a sealed or abstract modifier. Note however, that since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.

Sample:

An *abstract* static class:

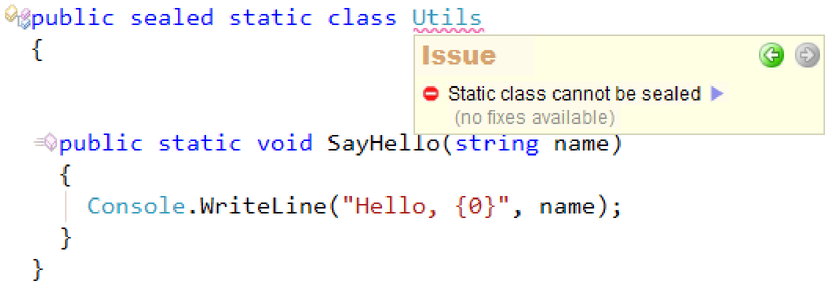


```
public abstract static class Utils
  {


  public static void SayHello(string name)
    {
      Console.WriteLine("Hello, {0}", name);
    }
  }
```

Issue

Static class cannot be abstract
(no fixes available)

A *sealed* static class:



```
public sealed static class Utils
  {


   public static void SayHello(string name)
    {
     Console.WriteLine("Hello, {0}", name);
    }
  }
```

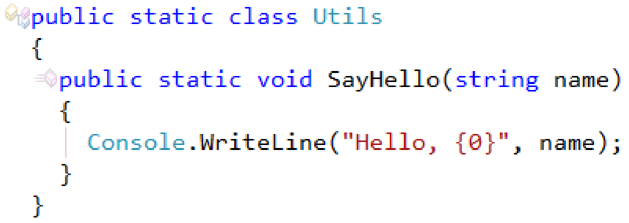
Issue

Static class cannot be sealed

(no fixes available)

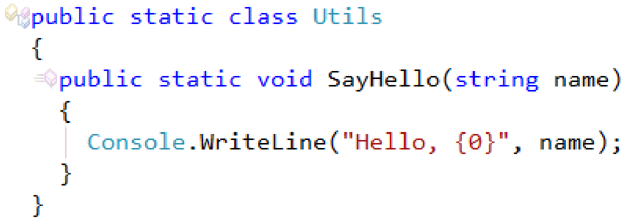
How to fix:

- Do not mark a static class as 'abstract':



```
public static class Utils
  {
   public static void SayHello(string name)
    {
     Console.WriteLine("Hello, {0}", name);
    }
  }
```

- Do not mark a static class as 'sealed':



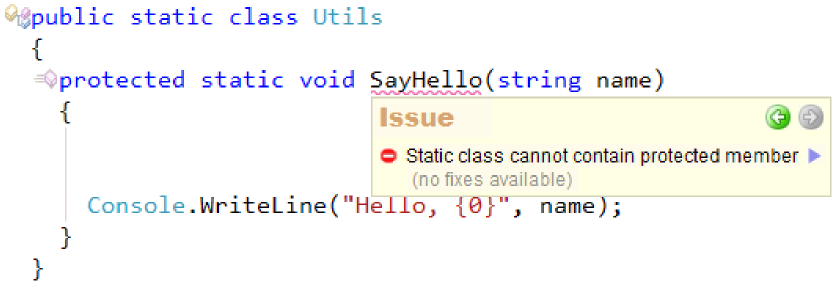
```
public static class Utils
  {
   public static void SayHello(string name)
    {
     Console.WriteLine("Hello, {0}", name);
    }
  }
```

Static class cannot contain protected member

Cause:

A static class can only contain static members, which can be public or private. They cannot be marked '*protected*' or '*protected internal*'. Since Static classes are sealed classes, and sealed classes can not be inherited, hence it can not contain protected members, because protected members can not be inherited.

Sample:



```
public static class Utils
  {
   protected static void SayHello(string name)
    {


     Console.WriteLine("Hello, {0}", name);
    }
  }
```

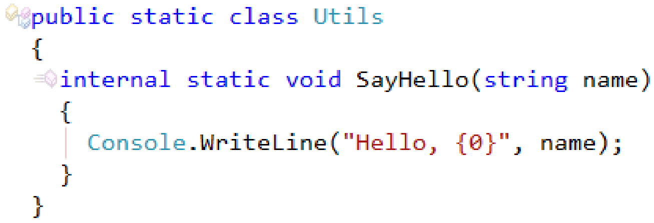
Issue

Static class cannot contain protected member

(no fixes available)

How to fix:

- Change the visibility of the static member, e.g.,:



```
public static class Utils
  {
   internal static void SayHello(string name)
    {
     Console.WriteLine("Hello, {0}", name);
    }
  }
```

Code Issues for interfaces

An interface is a code structure that has no concrete members and similar to an abstract class. An interface can contain public members such as methods, events, properties and indexers, but these members do not provide any functionality. Instead, they define items that must be made concrete within all classes that implement this interface. This means that an interface can be used to define what a class must do, but not how it will achieve it.

Interfaces are declared using several declaration rules: interfaces cannot have fields or constants, interface members cannot have a definition, an interface instance cannot be created, etc. Most of these rules are covered by the CodeRush Code Issues technology that allows you to correctly declare an interface and its members while writing code.

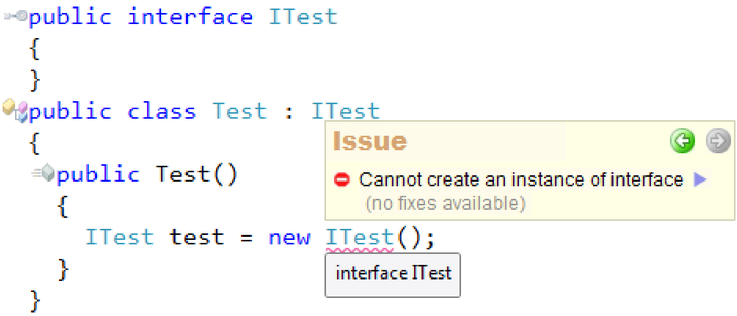
Here they are:

Cannot create an instance of interface

Cause:

We cannot initialize an interfaces instance, because this is not allowed by the .NET Framework guidelines. If you try to create an interface interface, you will see an error.

Sample:



```
public interface ITest
  {
  }
public class Test : ITest
  {
   public Test()
    {
     ITest test = new ITest();
    }
  }
```

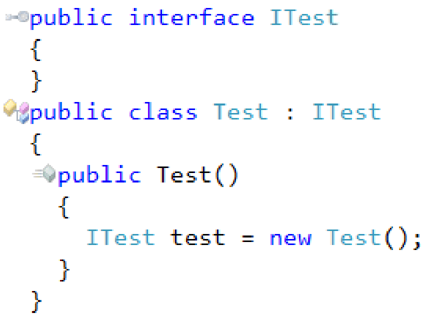
Issue

Cannot create an instance of interface
(no fixes available)

interface ITest

How to fix:

- Instead, we can create an object instance that implements the interface:



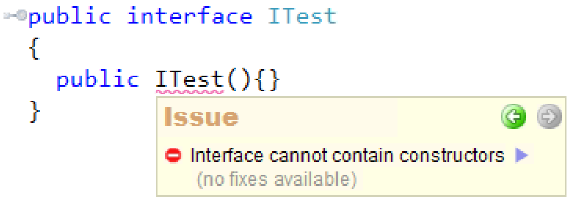
```
public interface ITest
{
}
public class Test : ITest
{
  public Test()
  {
    ITest test = new Test();
  }
}
```

Interface cannot contain constructors

Cause:

An interface cannot have constructors, because you cannot create an interface. Should there be a constructor in an interface, an error message will be shown.

Sample:



```
public interface ITest
{
  public ITest(){}
}
```

Issue

Interface cannot contain constructors

(no fixes available)

How to fix:

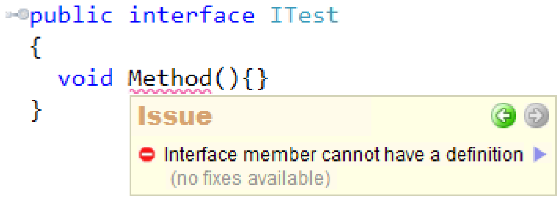
- Remove the incorrect constructor.

Interface member cannot have a definition

Cause:

As I mentioned before, interfaces do not provide any functionality. That is why interface members cannot have a definition. If you provide a definition of a member, an error is shown.

Sample:



```
public interface ITest
{
  void Method(){}
}
```

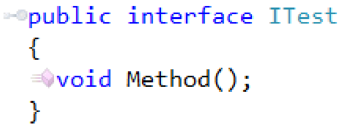
Issue

Interface member cannot have a definition

(no fixes available)

How to fix:

- Remove the body of the member:



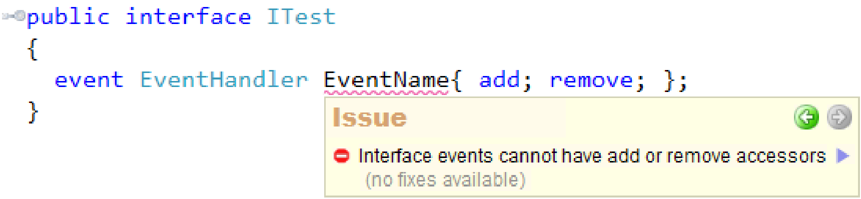
```
public interface ITest
{
    void Method();
}
```

Interface events cannot have add or remove accessors

Cause:

An event in an interface cannot have add or remove accessors. Interface events must be declared without add or remove accessors. This is a rule for interface events.

Sample:



```
public interface ITest
{
    event EventHandler EventName{ add; remove; };
}
```

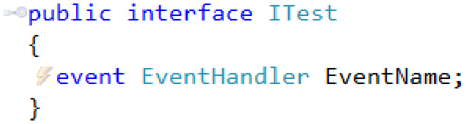
Issue

Interface events cannot have add or remove accessors

(no fixes available)

How to fix:

- Remove the event accessors definition:



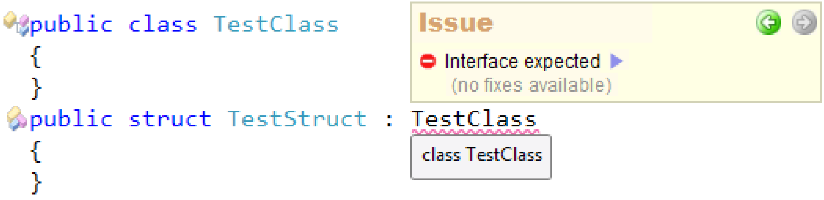
```
public interface ITest
{
    event EventHandler EventName;
}
```

Interface expected

Cause:

This code issue shows an error when the structure is derived from a non-interface type, because structures can only inherit from other classes by their nature and guidelines.

Sample:



```
public class TestClass
{
}
public struct TestStruct : TestClass
{
}
```

Issue

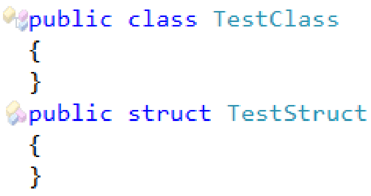
Interface expected

(no fixes available)

class TestClass

How to fix:

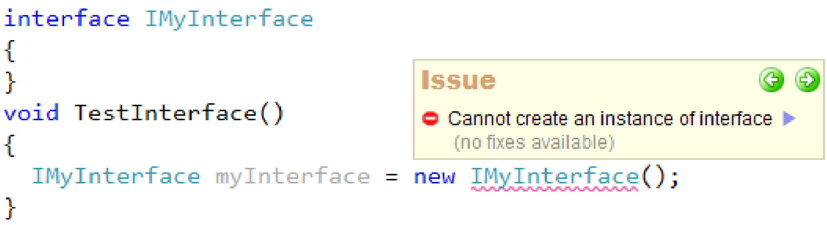
- Do not derive structure from a class:



```
public class TestClass
  {
  }
public struct TestStruct
  {
  }
```

Cannot create an instance of interface

This code issue shows an *error* if there is an attempt to initialize an instance of a interface. Interfaces cannot be instantiated, they contain only the signatures of methods, properties or events. The implementation of these members is done in the class that implements the interface.



```
interface IMyInterface
{
}
void TestInterface()
{
  IMyInterface myInterface = new IMyInterface();
}
```

Issue

Cannot create an instance of interface

(no fixes available)

Code Issues specific for declaring variables and constants

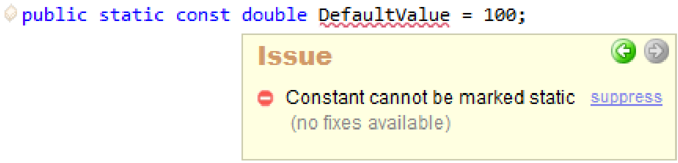
In addition to code issues specific for declaring types, CSharp and Visual Basic language specifications have a few restrictions on declaring constant or simple variable declarations. Having this issue in the code editor right in front of a developer allows him to fix the issue immediately without compiling the project.

Constant cannot be marked static

Cause:

A const member is considered static by the compiler. Although constants are considered static members, a constant declaration neither requires nor allows a static modifier. In other words, if a variable is marked as const, it is also static and there's no need to mark it static.

Sample:



```
public static const double DefaultValue = 100;
```

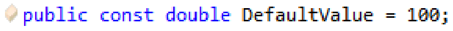
Issue

Constant cannot be marked static suppress

(no fixes available)

How to fix:

- Remove the static modifier from the const declaration:



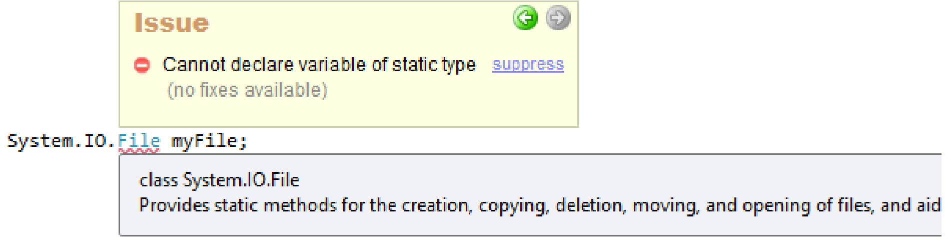
```
public const double DefaultValue = 100;
```

Cannot declare variable of static type

Cause:

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated, because static classes are sealed. You cannot use the new keyword to create a variable of a static class. Creating a variable of a static class is basically the same as creating a class that contains only static members and a private constructor. The compiler will guarantee that instances of a static class cannot be created.

Sample:



Cannot declare variable of static type
(no fixes available)

```
System.IO.File myFile;
```

class System.IO.File
Provides static methods for the creation, copying, deletion, moving, and opening of files, and aid

How to fix:

Because there is no instance variable, you access the members of a static class by using the class name itself. Remove the variable and access the type directly:

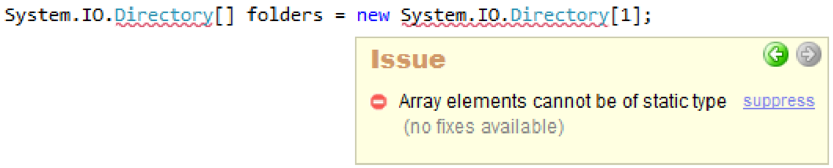
```
System.IO.File.Create(@"c:\FileName");
```

Array elements cannot be of static type

Cause:

The cause of this issue is the same as the cause of the previous issue about creating a variable of a static type. An array of the static type does not make sense, since array elements are instances, but it is not possible to create instances of static types.

Sample:



```
System.IO.Directory[] folders = new System.IO.Directory[1];
```

Array elements cannot be of static type
(no fixes available)

How to fix:

- Do not create an array of the static type and access the type directly:

```
IEnumerable<Directory> folders = System.IO.Directory.EnumerateDirectories(@"c:\");
```

- Make the type non-static if appropriate.

Code issues specific to declaring members

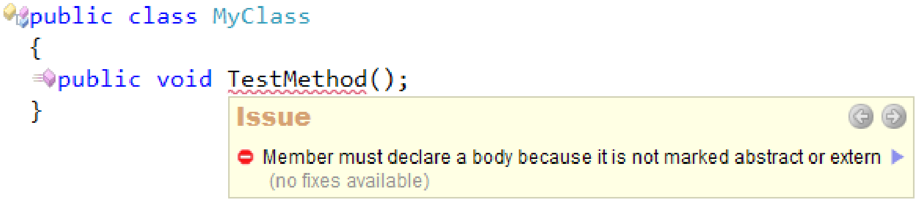
One of the goals of the Code Issues technology is to help you find mistakes when coding before compiling. This ine creases the coding speed and allows you to save time in the future when dialing with those mistakes. Let’s take a look at code issues CodeRush provides specific to declaring members:

Member must declare a body because it is not marked abstract or extern

Cause:

The code issue has the error type, because it is not allowed to declare non-abstract and non-extern members without a body. Non-abstract and non-extern members should have an implementation. This code issue is shown for properties and methods.

Sample #1:

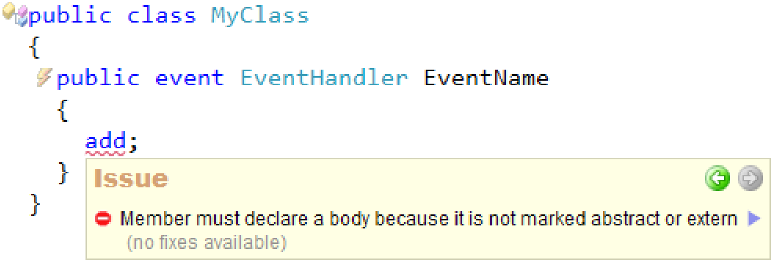


```
public class MyClass
{
  public void TestMethod();
}
```

Issue

Member must declare a body because it is not marked abstract or extern
(no fixes available)

Sample #2:



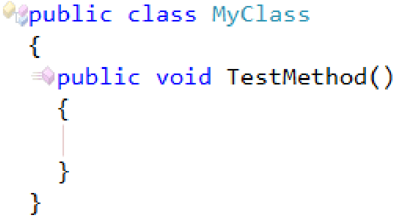
```
public class MyClass
{
  public event EventHandler EventName
  {
    add;
  }
}
```

Issue

Member must declare a body because it is not marked abstract or extern
(no fixes available)

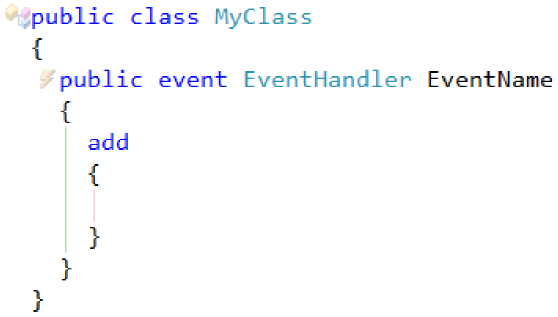
How to fix:

- Add the body of the method:



```
public class MyClass
{
  public void TestMethod()
  {

  }
}
```

- Add the body of the event accessor:



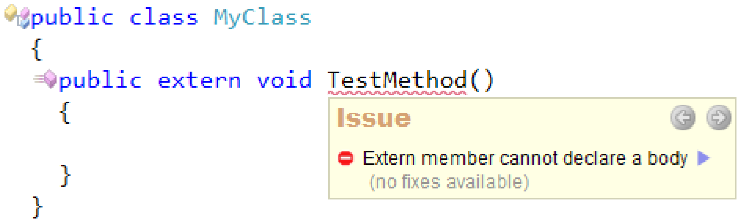
```
public class MyClass
{
  public event EventHandler EventName
  {
    add
    {

    }
  }
}
```

Extern member cannot declare a body

Cause:

This code issue is the opposite of the previous one. Highlights the member that is marked as an extern and that declares a body at the same time as an error.

Sample:



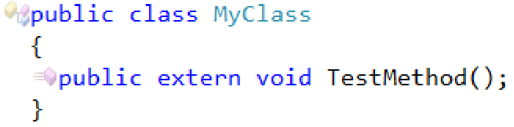
```
public class MyClass
{
  public extern void TestMethod()
  {

  }
}
```

Issue

Extern member cannot declare a body
(no fixes available)

How to fix:

- Remove the body of the extern member:



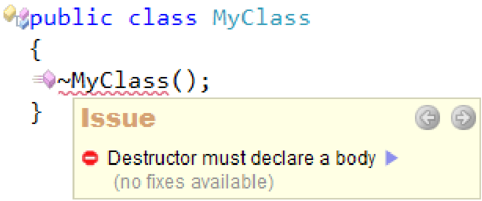
```
public class MyClass
{
  public extern void TestMethod();
}
```

Destructor must declare a body

Cause:

Destructors are a special type of members, so they have a dedicated code issue of the same error type. Destructors, like non-abstract and non-extern members, are also must declare a body.

Sample:



```
public class MyClass
{
  ~MyClass();
}
```

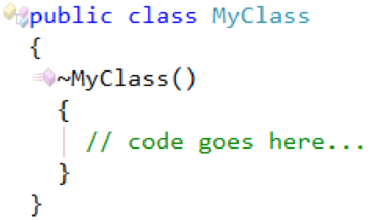
Issue

Destructor must declare a body

(no fixes available)

How to fix:

- Implement the desctructor:



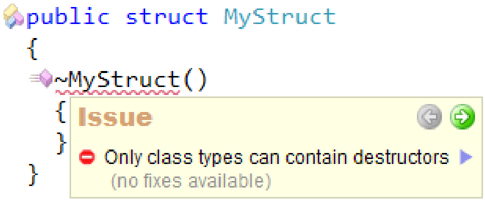
```
public class MyClass
{
  ~MyClass()
  {
    // code goes here...
  }
}
```

Only class types can contain destructors

Cause:

It is not allowed to declare destructors in a type different from a class, such as a structure or an interface. This code issue highlights destructors declared in the wrong type with an error.

Sample:



```
public struct MyStruct
{
  ~MyStruct()
  {
  }
}
```

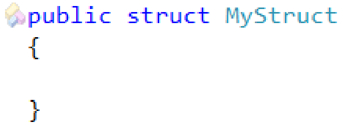
Issue

Only class types can contain destructors

(no fixes available)

How to fix:

- Remove the descructor:



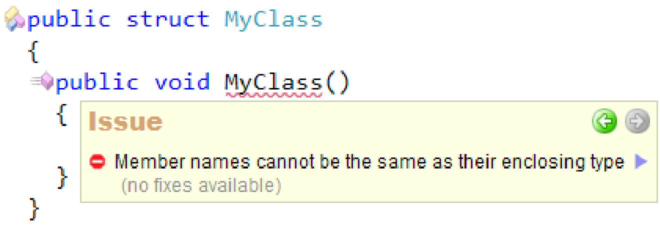
```
public struct MyStruct
{

}
```

Member names cannot be the same as their enclosing type

Cause:

The code issue of the error type highlights members with the same name as their enclosing type, because member names must not match the name of their enclosing type. Only type constructors can be named the same as their enclosing type.

Sample:



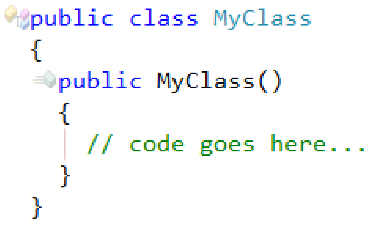
```
public struct MyClass
{
  public void MyClass()
  {
  }
}
```

Issue

Member names cannot be the same as their enclosing type
(no fixes available)

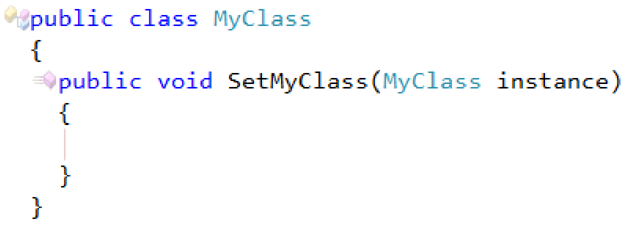
How to fix:

- Remove the member type to make it a constructor:



```
public class MyClass
{
  public MyClass()
  {
    // code goes here...
  }
}
```

- Rename the member:



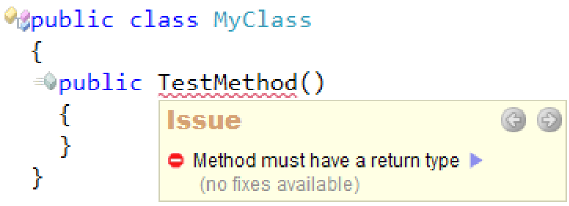
```
public class MyClass
{
  public void SetMyClass(MyClass instance)
  {

  }
}
```

Method must have a return type

Cause:

Methods can not be declared without a return type. For example, procedures that do not return a value should have the ‘void’ return type. If the method does not have a return type and it is not a constructor, the method is highlighted as erroneous. Only constructors should be declared without a return type specified.

Sample:



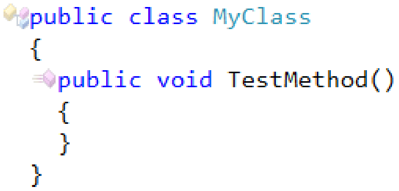
```
public class MyClass
{
  public TestMethod()
  {
  }
}
```

Issue

Method must have a return type
(no fixes available)

How to fix:

- Specify the member type:



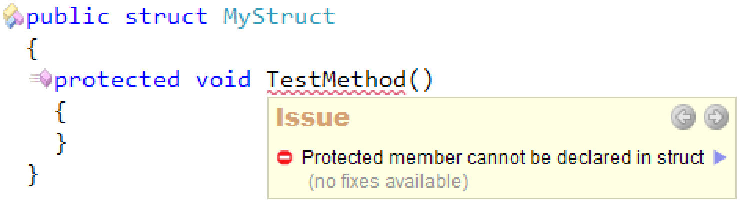
```
public class MyClass
{
  public void TestMethod()
  {
  }
}
```

Protected member cannot be declared in struct

Cause:

Protected members are not allowed inside structs because you can not inherit from a structure. This code issue has an error type:

Sample:



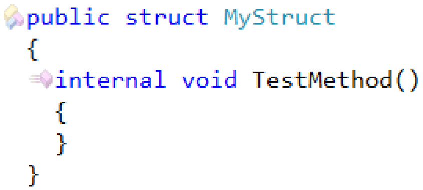
```
public struct MyStruct
{
  protected void TestMethod()
  {
  }
}
```

Issue

Protected member cannot be declared in struct
(no fixes available)

How to fix:

- Change the member access modifier, e.g.,:



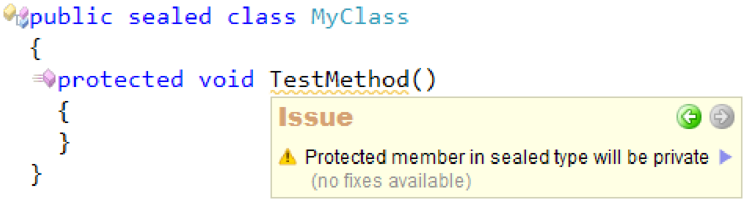
```
public struct MyStruct
{
  internal void TestMethod()
  {
  }
}
```

Protected member in sealed type will be private

Cause:

Sealed classes can not be inherited, so any protected member is treated as a private.

Sample:



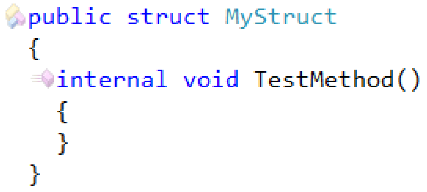
```
public sealed class MyClass
{
  protected void TestMethod()
  {
  }
}
```

Issue

Protected member in sealed type will be private
(no fixes available)

How to fix:

- Change the member access modifier, e.g.,:



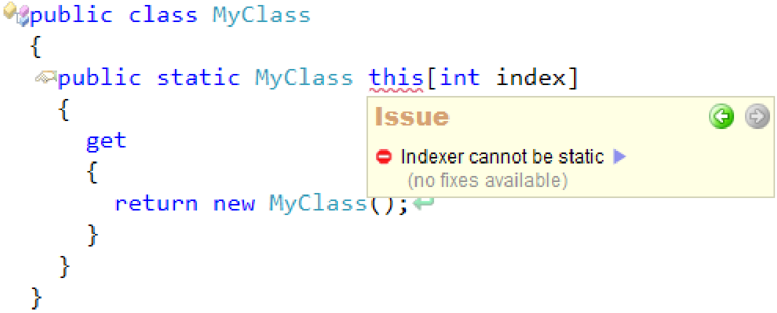
```
public struct MyStruct
{
  internal void TestMethod()
  {
  }
}
```

Indexer cannot be static

Cause:

Indexed properties are not allowed to be static. This code issue has an error type.

Sample:



```
public class MyClass
{
  public static MyClass this[int index]
  {
    get
    {
      return new MyClass();
    }
  }
}
```

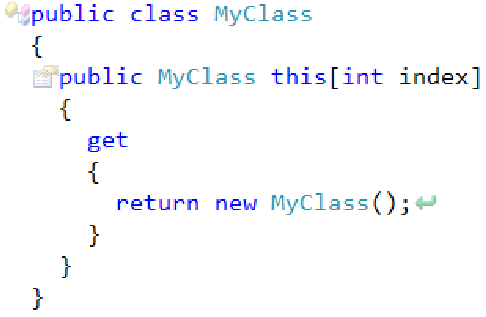
Issue

Indexer cannot be static

(no fixes available)

How to fix:

- Remove the static keyword:



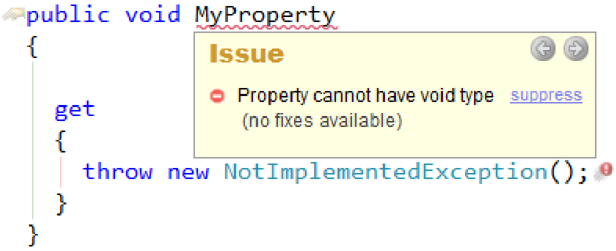
```
public class MyClass
{
  public MyClass this[int index]
  {
    get
    {
      return new MyClass();
    }
  }
}
```

Property cannot have void type

Cause:

A property is a member that provides a flexible mechanism to read, write, or compute the value of a field, thus it cannot have a void type. Properties are actually special methods called accessors (getter or setter) and they must have a specific non-void type according to most language specifications.

Sample:



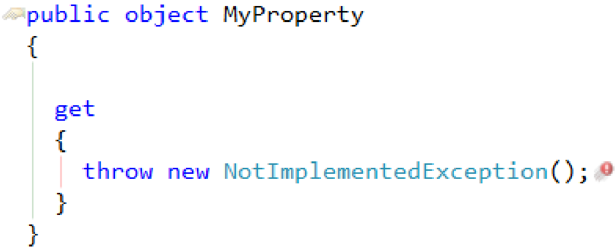
```
public void MyProperty
{

  get
  {
    throw new NotImplementedException();
  }
}
```

Issue

Property cannot have void type
(no fixes available)

How to fix:

Change the type of the property, e.g.,:



```
public object MyProperty
{

  get
  {
    throw new NotImplementedException();
  }
}
```

Code issues specific to constructors

A constructor is a special class member that is executed when a new object is created. There are two types of constructors: instance constructors and static constructors. Instance constructors are used to create and initialize instances of classes or structures. A static constructor is used to initialize a class itself. A static constructor is called automatically to initialize the class before the first instance is created or any static members are invoked.

All instance constructors implicitly include an invocation of another instance constructor immediately before the constructor-body if no constructor initializer is specified. There are two types of constructor initializers: a form base(), which calls a base class constructor and a form this(), which calls another constructor in the current class. There must always be a chain of constructors which invoke other constructors all the way up the class hierarchy. If an instance constructor has no constructor initializer, a constructor initializer of the form base() is implicitly provided.

DevExpress CodeRush provides several code issues specific for constructors. Most of them are hints (or suggestions) and others are errors. Code issues of an error type help you to avoid mistakes while declaring constructors inside classes or structures. Code issues of a suggestion (hint) type may help you to make your code more efficient and readable.

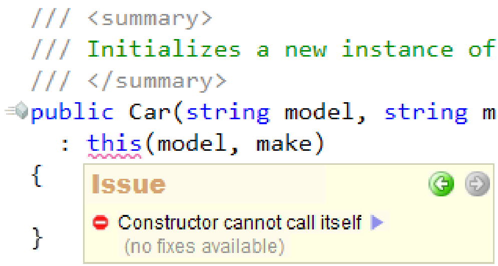
Here they are:

Constructor cannot call itself

Cause:

As I've said earlier, there must be a chain of constructors which invokes other constructors in the current class and/or class hierarchy. It is meaningless for a constructor to call itself, because there is no need to call the same instance initialization code. In fact, it would result in an infinite recursion if permitted. That is why a constructor cannot call itself.

Sample:



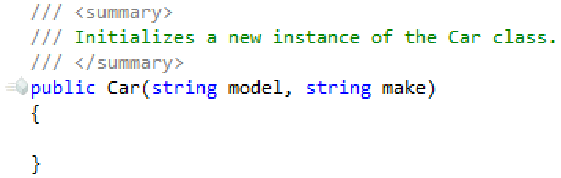
```
/// <summary>
/// Initializes a new instance of the Car class.
/// </summary>
public Car(string model, string make)
   : this(model, make)
{
}
```

Issue

Constructor cannot call itself
(no fixes available)

How to fix:

- Remove the call to the same constructor:



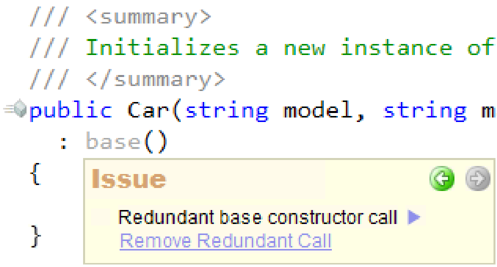
```
/// <summary>
/// Initializes a new instance of the Car class.
/// </summary>
public Car(string model, string make)
{

}
```

Redundant base constructor call

Cause:

We already know that if no constructor initializer is explicitly specified, then a constructor of the form base() (parameterless) is implicitly provided. There is no need to explicitly specify the parameterless base constructor call. That is why it is redundant and can be safely removed. Removing the redundant base constructor call may improve code readability.

Sample:



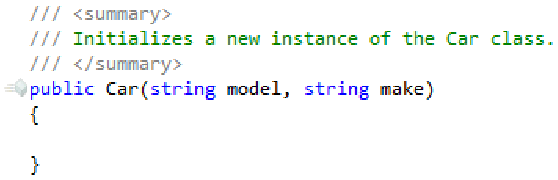
```
/// <summary>
/// Initializes a new instance of the Car class.
/// </summary>
public Car(string model, string make)
   : base()
{
}
```

Issue

Redundant base constructor call
Remove Redundant Call

How to fix:

- Apply the **Remove Redundant Call** refactoring:



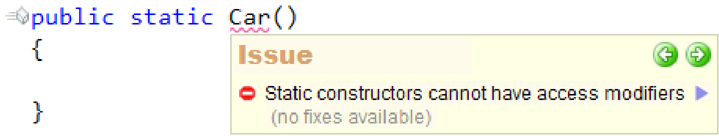
```
/// <summary>
/// Initializes a new instance of the Car class.
/// </summary>
public Car(string model, string make)
{

}
```

Static constructors cannot have access modifiers

Cause:

Static constructors cannot be called directly, they are called automatically as written previously. That is why you cannot include an access modifier (e.g. public, protected) when defining a static constructor. An access modifier is simply redundant and not allowed by the CSharp language specification, for example. You will see an error code issue inside the code editor when a static constructor has access modifiers.

Sample:



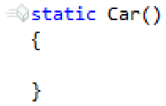
```
public static Car()
{

}
```

Issue

Static constructors cannot have access modifiers
(no fixes available)

How to fix:

- Remove the access modifiers from the static constructor:



```
static Car()
{

}
```

Struct cannot contain parameterless constructor

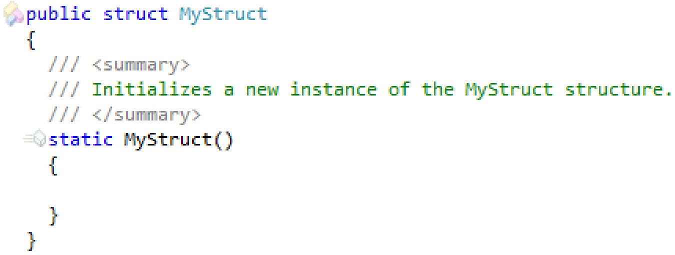
Cause:

Structs can have constructors just like classes. However, constructors and structs behave a bit differently from classes. Structs can have instance constructors with one notable exception – they cannot have a user-defined default parameterless constructor.

Parameterless constructors are not necessary for value types, since the compiler, by default, neither generates a default constructor, nor generates a call to the default constructor, which simply sets the fields of the value type to their default (zero) values. When an instance that has all of its fields zeroed is created, a user can, in some cases, then use this instance without further initialization.

It is important to make sure that the all-zeroed state is a valid initial state for the value types. A default parameterless constructor for a struct could set different values than the all-zeroed state which would be an unexpected behavior. The .NET Framework therefore prohibits default constructors for struct. An error is shown when you have such a constructor.

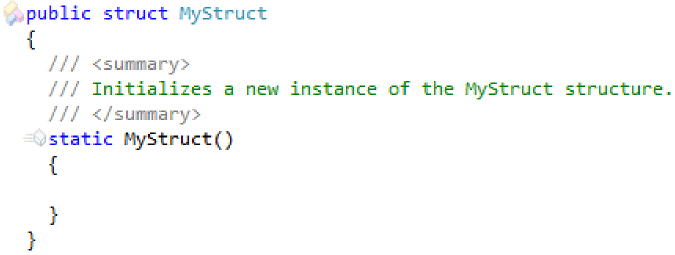
Sample:



```
public struct MyStruct
{
    /// <summary>
    /// Initializes a new instance of the MyStruct structure.
    /// </summary>
    static MyStruct()
    {

    }
}
```

How to fix:

- Mark the constructor ‘static’:



```
public struct MyStruct
{
    /// <summary>
    /// Initializes a new instance of the MyStruct structure.
    /// </summary>
    static MyStruct()
    {

    }
}
```

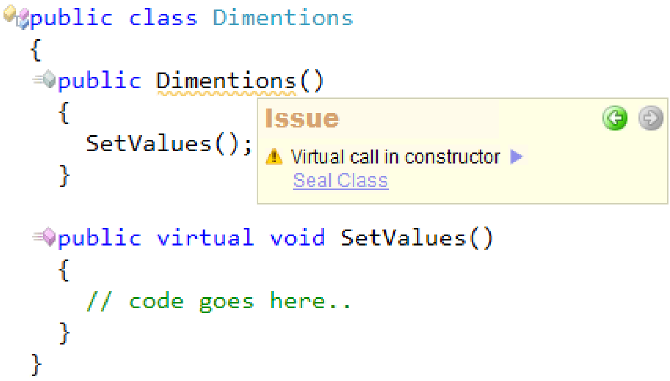
Virtual member call in constructor

Cause:

The order in which constructors are called is different from the order in which virtual methods are called. Calling virtual methods during construction will cause the most derived override to be called, even though the most derived constructor has not been completely run yet.

When an object is constructed, the object initializers run in an order from the most derived class to the top base class, and then constructors run in an order from the top base class to the most derived class. Objects being constructed do not change their type during initialization, but start out as the most derived type, with the method table being generated for the most derived type. This means that virtual method calls always run on the most derived type. If you invoke a virtual method in a constructor, and it is not the most derived type in its inheritance hierarchy, then it will be called on a class whose constructor has not been run yet, and therefore may not be in the correct state to have that method called. That is why calling a virtual method can produce unexpected results.

Sample:

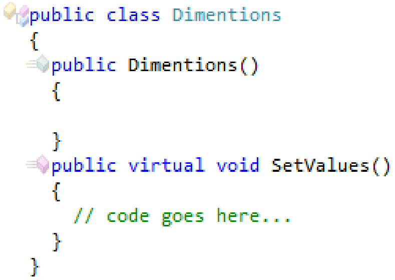


```
public class Dimentions
{
  public Dimentions()
  {
    SetValues();
  }

  public virtual void SetValues()
  {
    // code goes here..
  }
}
```

Virtual call in constructor

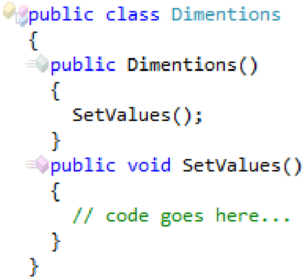
How to fix:

- Remove the virtual member call from the constructor:



```
public class Dimentions
{
  public Dimentions()
  {

  }
  public virtual void SetValues()
  {
    // code goes here...
  }
}
```

- Do not mark the invoked member ‘virtual’:



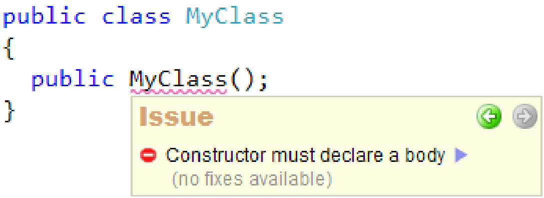
```
public class Dimentions
{
  public Dimentions()
  {
    SetValues();
  }
  public void SetValues()
  {
    // code goes here...
  }
}
```

Constructor must declare a body

Cause:

The **Constructor must declare a body** code issue shows an error if there is a constructor without a body. Constructors are similar to usual methods and whenever a class or struct is created, its constructor is called. Constructors should always declare its body to properly initialize a class or struct.

Sample:



```
public class MyClass
{
   public MyClass();
}
```

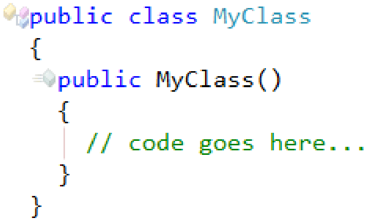
Issue

Constructor must declare a body

(no fixes available)

How to fix:

- Declare the body of the constructor:



```
public class MyClass
  {
    public MyClass()
    {
      // code goes here...
    }
  }
```

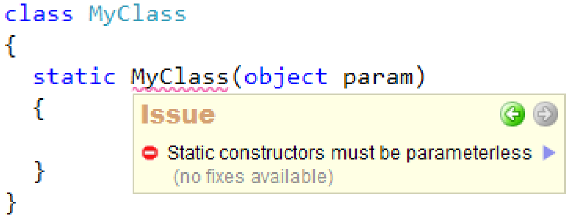
See the full list of code issues specific to constructors.

Static constructors must be parameterless

Cause:

A static constructor is used to initialize any static data, or to perform a particular action that needs performed once only. Parameters are not allowed for static constructors.

Sample:



```
class MyClass
{
   static MyClass(object param)
   {

   }
}
```

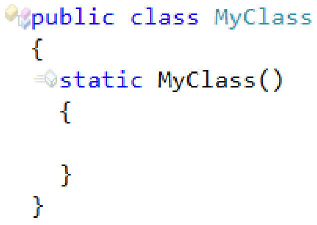
Issue

Static constructors must be parameterless

(no fixes available)

How to fix:

Remove the redundant parameter:



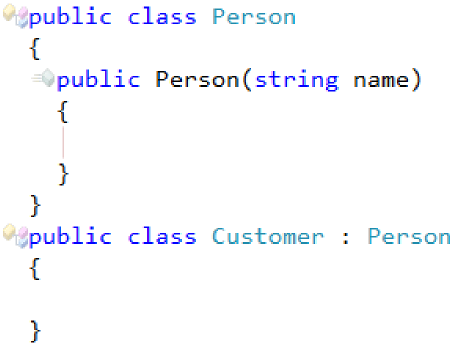
```
public class MyClass
  {
    static MyClass()
    {

    }
  }
```

See the full list of code issues specific to constructors.

Base type constructors are not implemented

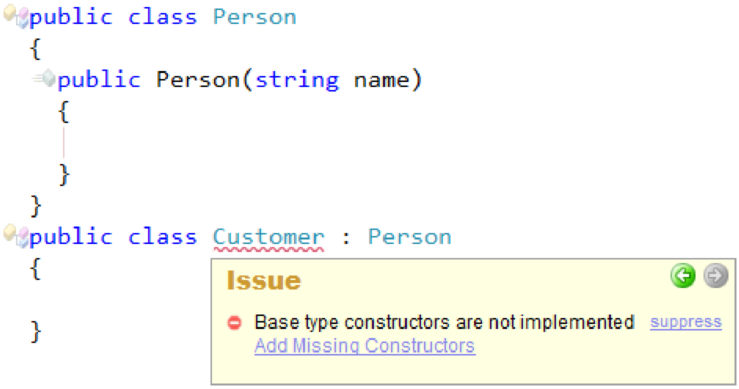
This code issue is an alternative code issue to the "Can implement base type constructors". The different between these two is in their type – one is an error and another one is a hint. When the current type doesn't implement non-default constructors from a base type the "Base type constructors are not implemented" error is shown.

Consider the following example, where you can see this code issue:



```
public class Person
{
    public Person(string name)
    {

    }
}
public class Customer : Person
{

}
```

Here we have two classes – *Person* and *Customer*. The *Person* class has a single constructor that takes one parameter of type string. The *Customer* class, on the other hand, doesn't have any constructors, and the compiler will generate the default one – a public constructor with no parameters. This default constructor will call the default constructor of the base class, which simply doesn't exist, and the source code won't compile in this case. Thus, we have to implement at least missing constructors from the base class – the code issue is shown in this case:



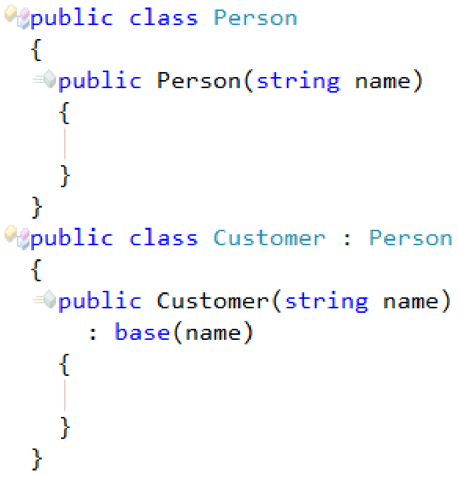
```
public class Person
{
    public Person(string name)
    {

    }
}
public class Customer : Person
{

}
```

Issue

Base type constructors are not implemented

How to fix:

- Apply the Add Missing Constructors code provider:



```
public class Person
{
    public Person(string name)
    {

    }
}
public class Customer : Person
{
    public Customer(string name)
        : base(name)
    {

    }
}
```

See the full list of code issues specific to constructors.

Code Issues specific to partial methods

A partial method declaration has two parts: the declaration itself and an implementation. Both parts of a partial method can be located in a single class or in different parts of a partial class. You can use partial methods in the code and implement them later if required. If you do not supply an implementation for a partial method, its signature is removed by the compiler.

There are several conditions that are applied to partial methods, such as:

- Partial methods must be void
- Signatures of both parts of partial methods must match
- Access modifiers are not allowed for partial methods
- Partial methods must be declared in partial classes
- etc

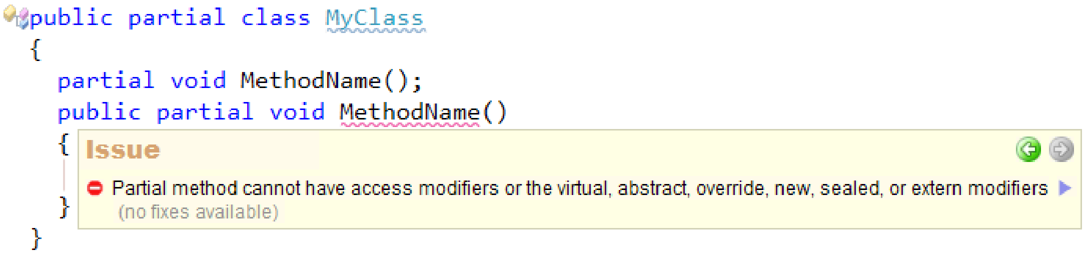
You don't have to remember all these conditions if you have the CodeRush code issues feature turned on. When the rule of the partial method declaration is violated, code issues will show you an error or a hint, and you can fix it before you compile the code. These code issues are:

Partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers

Cause:

If a partial method has an invalid modifier (e.g., virtual, abstract, override, new, sealed, extern, or an access modifier) you will see an error in the code editor, because partial methods cannot have these type of modifiers.

Sample #1:

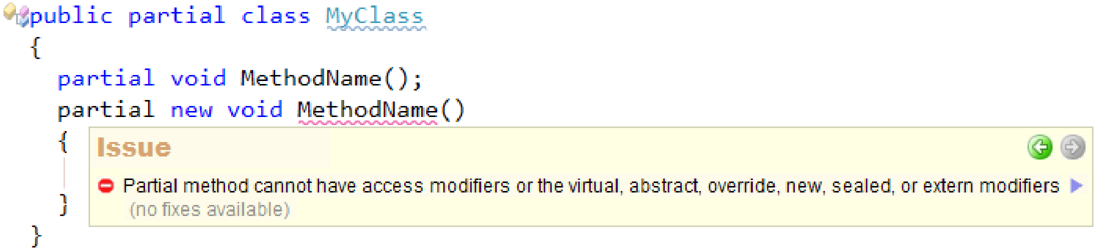


```
public partial class MyClass
{
    partial void MethodName();
    public partial void MethodName()
    {

    }
}
```

Issue

Partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers
(no fixes available)

Sample #2:



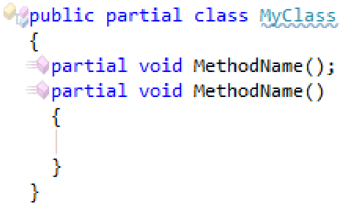
```
public partial class MyClass
{
    partial void MethodName();
    partial new void MethodName()
    {

    }
}
```

Issue

Partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers
(no fixes available)

How to fix:

- Remove the incorrect modifiers:



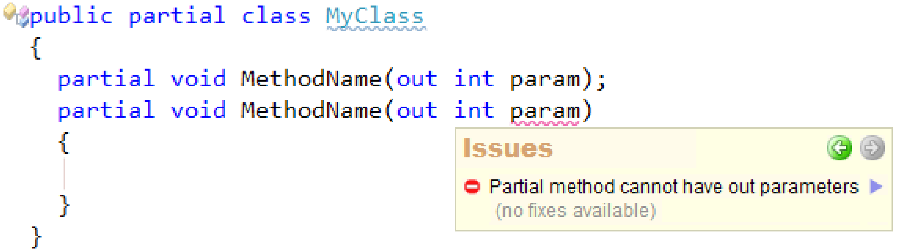
```
public partial class MyClass
{
    partial void MethodName();
    partial void MethodName()
    {

    }
}
```

Partial method cannot have out parameters

Cause:

Out parameters are not allowed for partial methods. If a partial method is declared with ‘out’ parameters, you will see an error.

Sample:



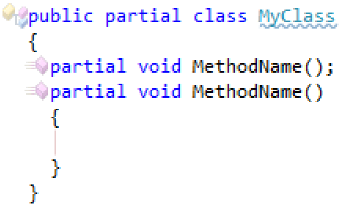
```
public partial class MyClass
{
    partial void MethodName(out int param);
    partial void MethodName(out int param)
    {

    }
}
```

Issues

Partial method cannot have out parameters
(no fixes available)

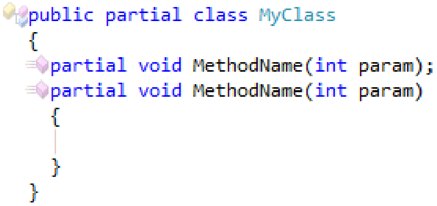
How to fix:

- Completely remove the ‘out’ parameter:



```
public partial class MyClass
{
    partial void MethodName();
    partial void MethodName()
    {

    }
}
```

- Do not mark the parameter as ‘out’:



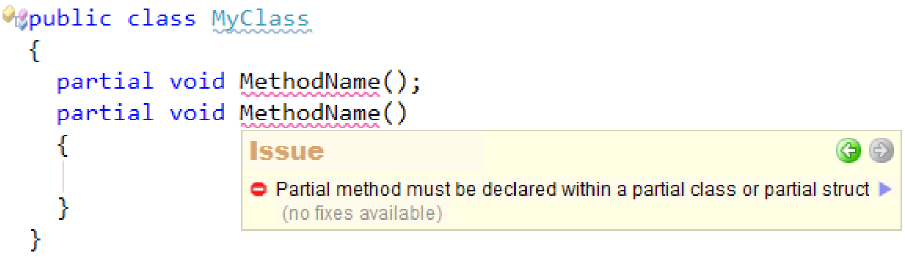
```
public partial class MyClass
{
    partial void MethodName(int param);
    partial void MethodName(int param)
    {

    }
}
```

Partial method must be declared within a partial class or partial struct

Cause:

Partial methods can only reside inside a partial class or a partial structure, otherwise, an error is shown.

Sample:



```
public class MyClass
{
    partial void MethodName();
    partial void MethodName()
    {

    }
}
```

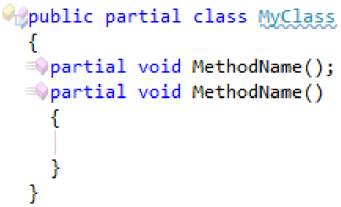
Issue

Partial method must be declared within a partial class or partial struct

(no fixes available)

How to fix:

- Mark the class ‘partial’:



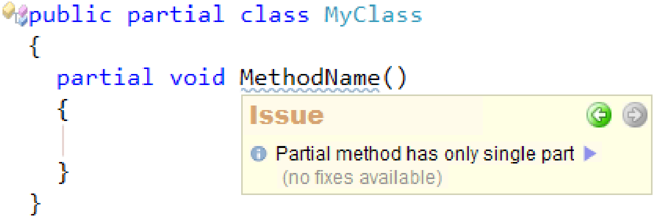
```
public partial class MyClass
{
    partial void MethodName();
    partial void MethodName()
    {

    }
}
```

Partial method has only single part

Cause:

When the partial method has only a single part without a declaration, it does not need to be declared as partial. This code issue has a hint type.

Sample:



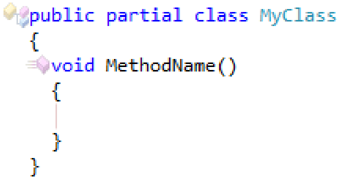
```
public partial class MyClass
{
  partial void MethodName()
  {

  }
}
```

Issue

Partial method has only single part ▸
(no fixes available)

How to fix:

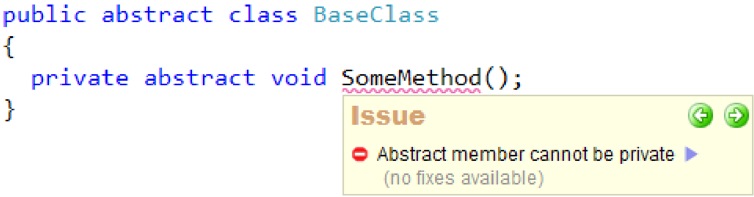
- Remove the partial keyword:



```
public partial class MyClass
{
  void MethodName()
  {

  }
}
```

Abstract member cannot be private

This code issue shows an error when an abstract member with the private visibility keyword is found. Abstract memdbers are incomplete, and must be implemented in a derived class. That's why abstract members can not be private – they must be visible in a derived class.



```
public abstract class BaseClass
{
  private abstract void SomeMethod();
}
```

Issue

Abstract member cannot be private ▸
(no fixes available)

Code issues for virtual members

As you probably know, if a member is declared with the '*virtual*' keyword, derived classes can override the implementation of this member. In a virtual member invocation, the run-time type of the instance for which that invocation takes place determines the actual member implementation to invoke: whether it is a base virtual member or an overridden member from a derived class. The virtual member is declared like an instance member with addition of a '*virtual*' keyword to its declaration.

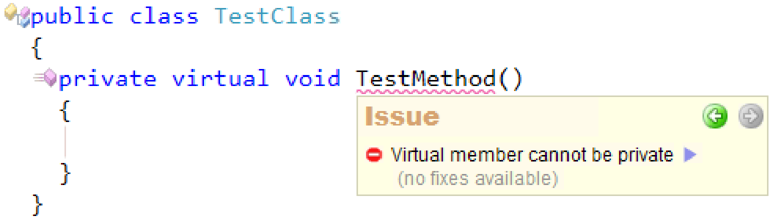
The peculiarity of virtual methods defines several other rules for their declaration. If you break these rules, you will see the error when you compile your source code. To help you declare the virtual methods correctly, CodeRush sugtgests several code issues that allow you to write the code for virtual member declaration without making mistakes. Here they are:

Virtual member cannot be private

Cause:

Virtual members must have a visibility greater than ‘private’ because they may be overridden in derived classes. If a member has a ‘*private*‘ modifier, it is not visible to descendants. You will see the error when a member is marked as ‘*virtual*‘ and has a ‘*private*‘ visibility keyword at the same time.

Sample:



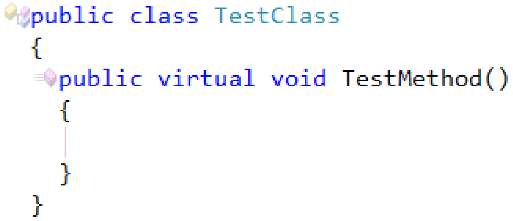
```
public class TestClass
{
  private virtual void TestMethod()
  {

  }
}
```

Issue

Virtual member cannot be private
(no fixes available)

How to fix:

- Change the ‘private’ visibility to ‘public’, for example:



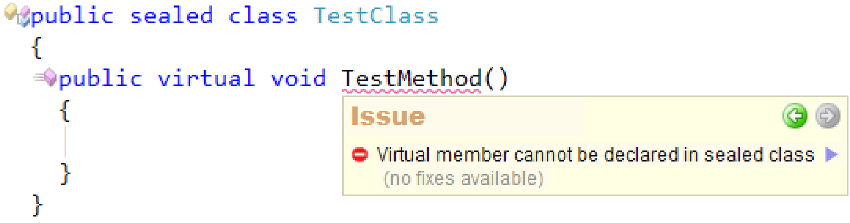
```
public class TestClass
{
  public virtual void TestMethod()
  {

  }
}
```

Virtual member cannot be declared in sealed class

Cause:

When the class is sealed, it cannot contain virtual members, because sealed classes cannot be inherited by other classes. Methods marked as ‘*virtual*‘ inside the sealed classes will show as an error.

Sample:



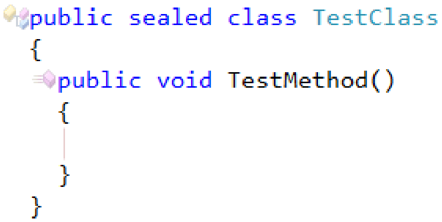
```
public sealed class TestClass
{
  public virtual void TestMethod()
  {

  }
}
```

Issue

Virtual member cannot be declared in sealed class
(no fixes available)

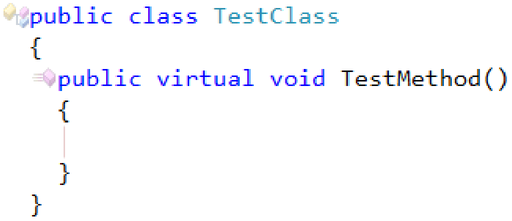
How to fix:

- Remove the ‘virtual’ modifier from the member, if appropriate:



```
public sealed class TestClass
{
  public void TestMethod()
  {

  }
}
```

- Remove the ‘sealed’ modifier from the class, if appropriate:



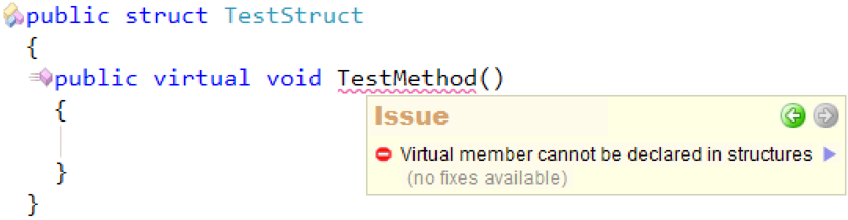
```
public class TestClass
{
  public virtual void TestMethod()
  {

  }
}
```

Virtual member cannot be declared in structures

Cause:

Virtual members are only allowed in types that can be descended from. Because you cannot inherit from a structure, you cannot declare a virtual member inside of it.

Sample:



```
public struct TestStruct
{
  public virtual void TestMethod()
  {

  }
}
```

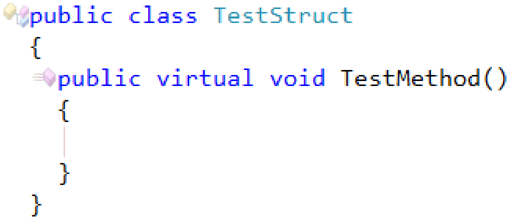
Issue

Virtual member cannot be declared in structures

(no fixes available)

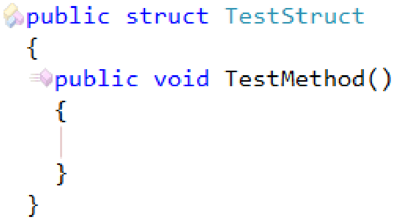
How to fix:

- Convert the class to a structure, if appropriate:



```
public class TestStruct
{
  public virtual void TestMethod()
  {

  }
}
```

- Do not mark the member as ‘virtual’, if appropriate:



```
public struct TestStruct
  {
   public void TestMethod()
     {
 
     }
  }
```

In addition to the code issues for members marked as ‘*virtual*‘, there are similar code issues for members marked as ‘*abstract*‘ and ‘*override*‘.

Code issues for overridden members

Overridden members (that include an ‘*override*‘ keyword) provide a new implementation of a virtual or an abkstract member with the same signature. Compile-time errors occur when member override rules are violated. To declare an overridden member according to the language specification without violating its rules, CodeRush suggests several overridden-specific code issues.

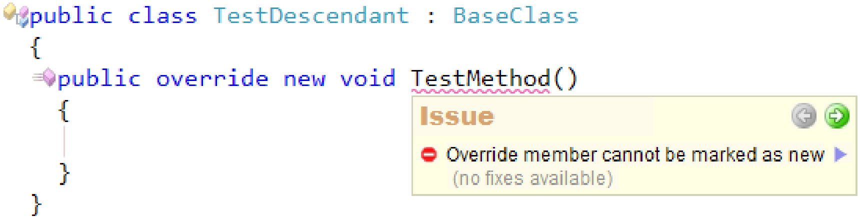
Here they are:

Override member cannot be marked as new

Cause:

An overridden member cannot be also marked as ‘*new*‘. Members marked with a ‘*new*‘ keyword, explicitly hide a member inherited from a base class. This is incorrect, because overridden members provide their own implementation of a member inherited from a base class and replace the base-class version. An error is shown for members marked as an ‘*override*‘ and ‘*new*‘ at the same time.

Sample:



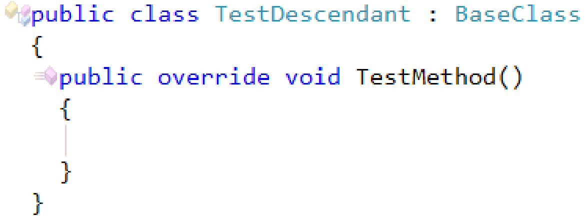
```
public class TestDescendant : BaseClass
  {
   public override new void TestMethod()
     {
 
     }
  }
```

Issue

Override member cannot be marked as new
(no fixes available)

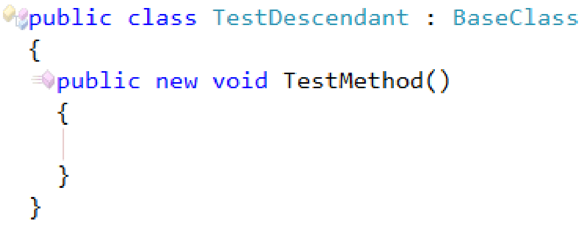
How to fix:

- Remove the ‘new’ keyword, if appropriate:



```
public class TestDescendant : BaseClass
  {
   public override void TestMethod()
     {
 
     }
  }
```

- Remove the ‘override’ keyword, if appropriate:



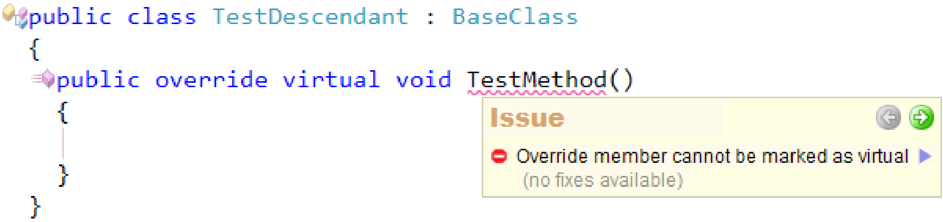
```
public class TestDescendant : BaseClass
  {
    public new void TestMethod()
    {

    }
  }
```

Override member cannot be marked as virtual

Cause:

An overridden member cannot also be marked as ‘*virtual*‘, because you can override the member marked as ‘*override*‘ in derived classes the same way as virtual members. So, the ‘*virtual*‘ modifier is redundant in this case. You will see an error for the overridden virtual member.

Sample:



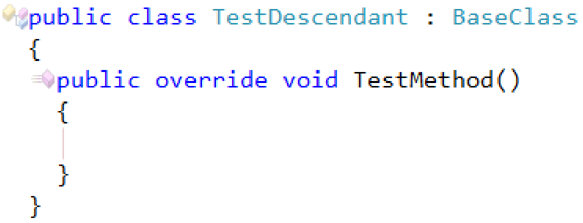
```
public class TestDescendant : BaseClass
  {
    public override virtual void TestMethod()
    {

    }
  }
```

Issue

Override member cannot be marked as virtual
(no fixes available)

How to fix:

- Do not mark the overridden member as virtual:



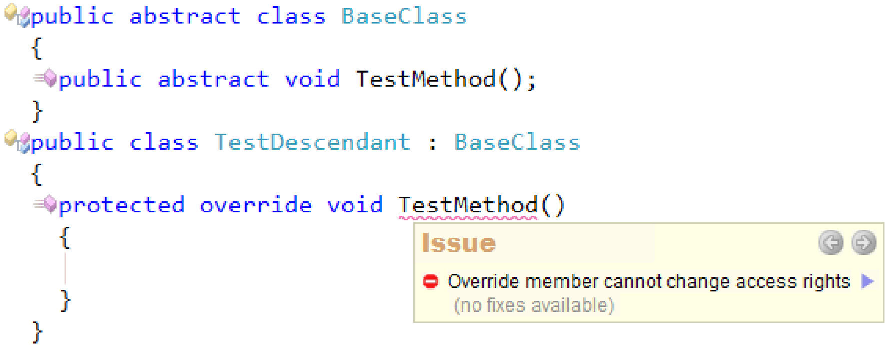
```
public class TestDescendant : BaseClass
  {
    public override void TestMethod()
    {

    }
  }
```

Override member cannot change access rights

Cause:

The visibility of overridden members must match inherited visibility of the base member that is being overridden. In other words, an overridden member cannot change the accessibility of the base member. However, if the overridden base member is ‘protected internal’ and it is declared in a different assembly than the assembly containing the override member, then the override method’s declared accessibility must be protected. An error is shown when different visibility modifiers are set for an overridden member and its base member.

Sample:



```
public abstract class BaseClass
  {
    public abstract void TestMethod();
  }
public class TestDescendant : BaseClass
  {
    protected override void TestMethod()
      {

      }
  }
```

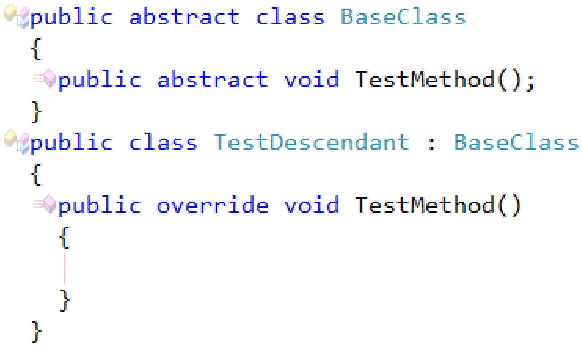
Issue

Override member cannot change access rights

(no fixes available)

How to fix:

- Do not change the access rights of the overridden member:



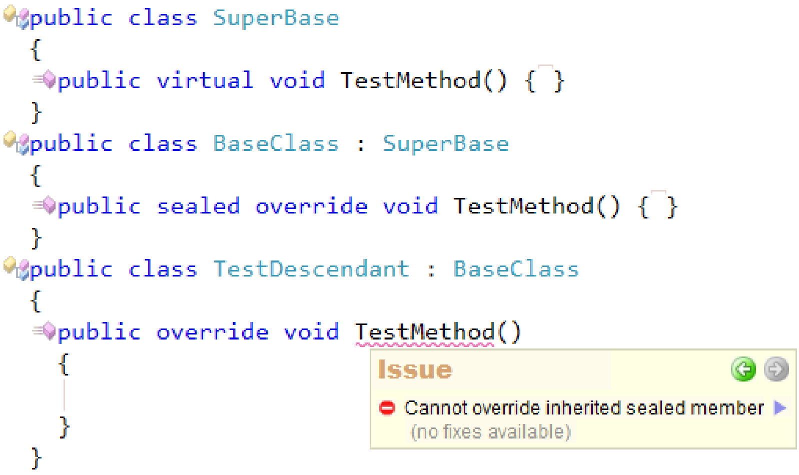
```
public abstract class BaseClass
  {
    public abstract void TestMethod();
  }
public class TestDescendant : BaseClass
  {
    public override void TestMethod()
      {

      }
  }
```

Cannot override an inherited sealed member

Cause:

The sealed member cannot be overridden according to the language specification.

Sample:



```
public class SuperBase
{
    public virtual void TestMethod() { }
}
public class BaseClass : SuperBase
{
    public sealed override void TestMethod() { }
}
public class TestDescendant : BaseClass
{
    public override void TestMethod()
    {

    }
}
```

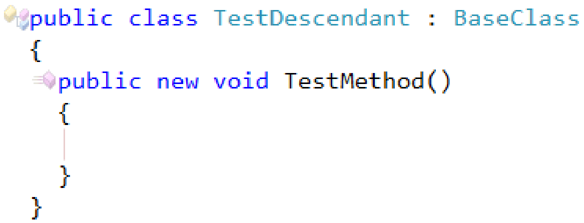
Issue

Cannot override inherited sealed member

(no fixes available)

How to fix:

- Mark the member as ‘new’ instead of ‘override’:



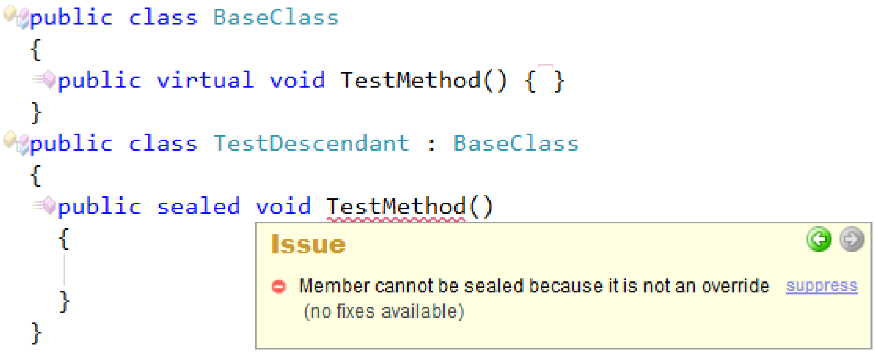
```
public class TestDescendant : BaseClass
{
    public new void TestMethod()
    {

    }
}
```

Member cannot be sealed because it is not an override

Cause:

The ‘sealed’ keyword is only permitted on overridden members. An error us shown for members marked as ‘*sealed*‘ but not overridden.

Sample:



```
public class BaseClass
{
    public virtual void TestMethod() { }
}
public class TestDescendant : BaseClass
{
    public sealed void TestMethod()
    {

    }
}
```

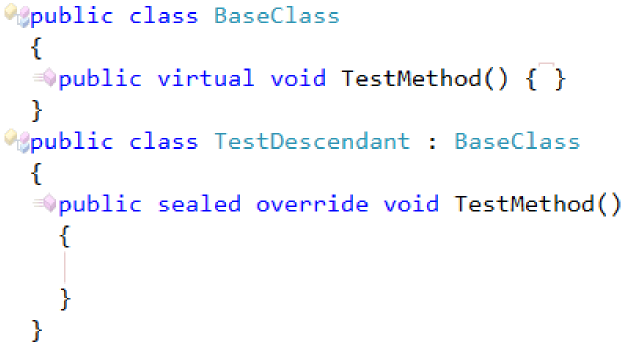
Issue

Member cannot be sealed because it is not an override suppress

(no fixes available)

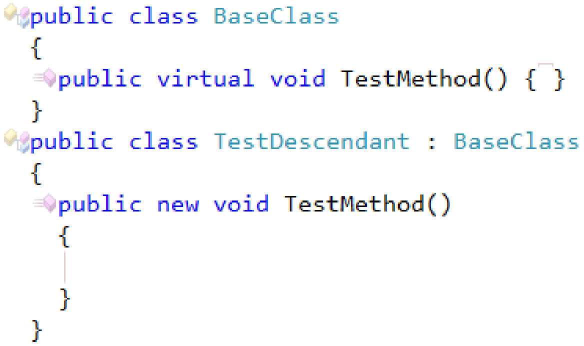
How to fix:

- Mark the sealed member as ‘override’ as well:



```
public class BaseClass
{
  public virtual void TestMethod() { }
}
public class TestDescendant : BaseClass
{
  public sealed override void TestMethod()
  {

  }
}
```

- Mark the member as ‘new’ instead of ‘sealed’:



```
public class BaseClass
{
  public virtual void TestMethod() { }
}
public class TestDescendant : BaseClass
{
  public new void TestMethod()
  {

  }
}
```

Code Issues specific to extension methods

Extension methods allow developers to expand existing types without having to sub-class, recompile or modify the original type. They were introduced as a feature of CSharp version 3.0 and Visual Basic version 9.0. Such methods are just like static methods invoked by using instance method syntax.

You can declare extension methods only in public static classes (C#) or Modules (VB). To declare an extension method in C#, you specify the keyword this as the first parameter of the public static method, for example:

```
public static class Extensions
{
  public static int ToInt32 (this string s)
  {
    return Int32.Parse(s);
  }
}
```

In Visual Basic you have to specify the *Extension*() attribute from the *System.Runtime.CompilerServices* namespace. The first parameter in an extension method definition specifies which data type the method extends, for example:

```
Public Module Extensions
  <Extension()>
  Public Function ToInt32(s As String) As Integer
    Return Int32.Parse(s)
  End Function
End Module
```

There are additional rules for syntax of declaring extension methods. CodeRush suggests a few code issues that ale lows you to correctly write them and check its syntax before compiling the code. All code issues are of the error type, because your code won't compile if you see them.

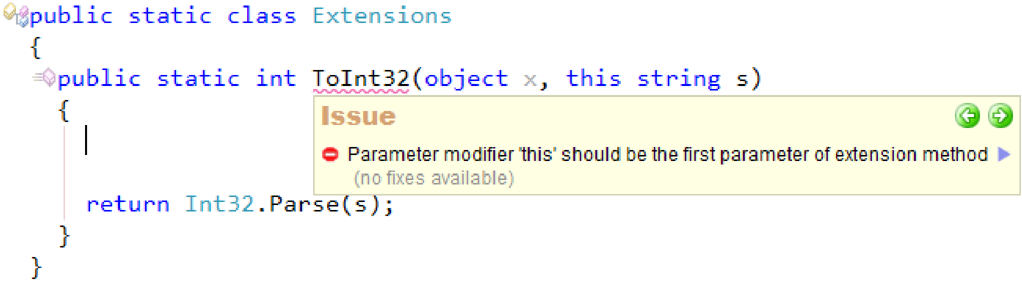
Here they are:

Parameter modifier 'this' should be the first parameter of extension method

Cause:

An extension method's first parameter must include 'this' modifier in CSharp.

Sample:



```
public static class Extensions
{
  public static int ToInt32(object x, this string s)
  {
    |
    return Int32.Parse(s);
  }
}
```

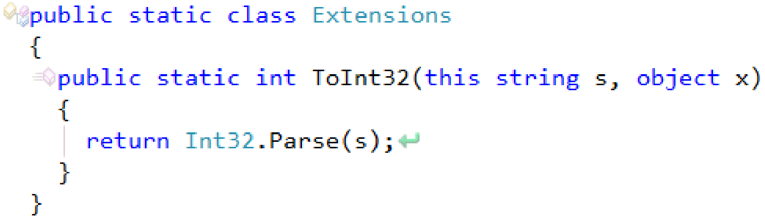
Issue

Parameter modifier 'this' should be the first parameter of extension method

(no fixes available)

How to fix:

- Make sure that the parameter with 'this' modifier is declared as the first parameter of the extension method:



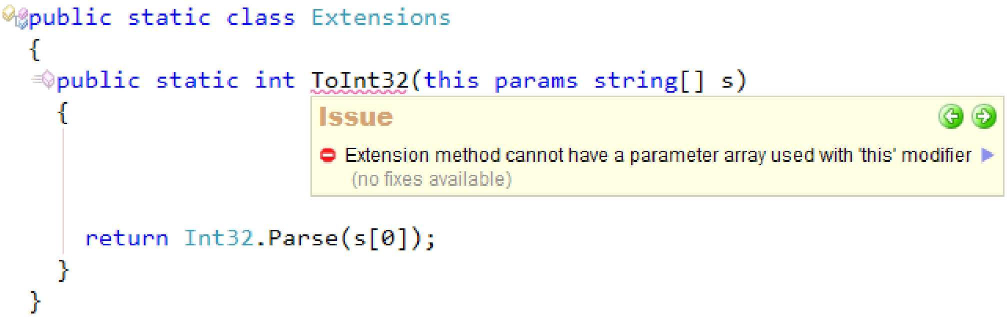
```
public static class Extensions
{
  public static int ToInt32(this string s, object x)
  {
    return Int32.Parse(s);
  }
}
```

Extension method cannot have a parameter array used with ‘this’ modifier

Cause:

The first parameter to an extension method cannot be a parameter array.

Sample:



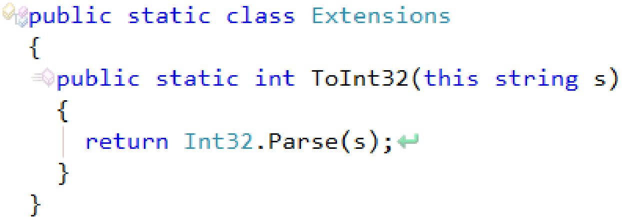
```
public static class Extensions
{
    public static int ToInt32(this params string[] s)
    {


        return Int32.Parse(s[0]);
    }
}
```

Issue

Extension method cannot have a parameter array used with 'this' modifier
(no fixes available)

How to fix:

- Make sure that the parameter with ‘this’ modifier is not declared as a param array:



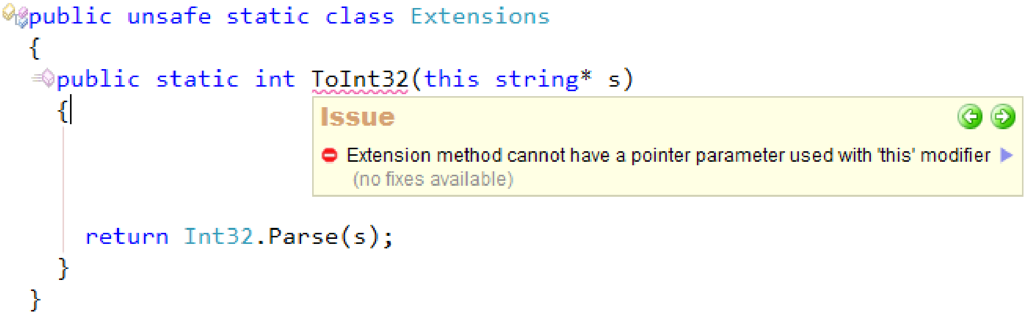
```
public static class Extensions
{
    public static int ToInt32(this string s)
    {
        return Int32.Parse(s);
    }
}
```

Extension method cannot have a pointer parameter used with ‘this’ modifier

Cause:

The first parameter to an extension method must not be of pointer type.

Sample:



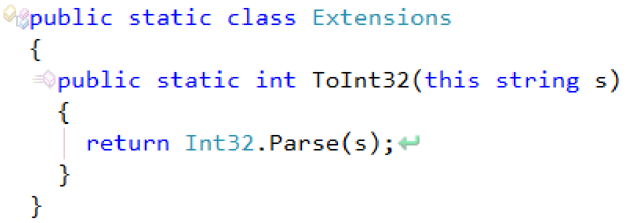
```
public unsafe static class Extensions
{
    public static int ToInt32(this string* s)
    {


        return Int32.Parse(s);
    }
}
```

Issue

Extension method cannot have a pointer parameter used with 'this' modifier
(no fixes available)

How to fix:

- Remove the pointer specifier from the parameter with ‘this’ modifier:



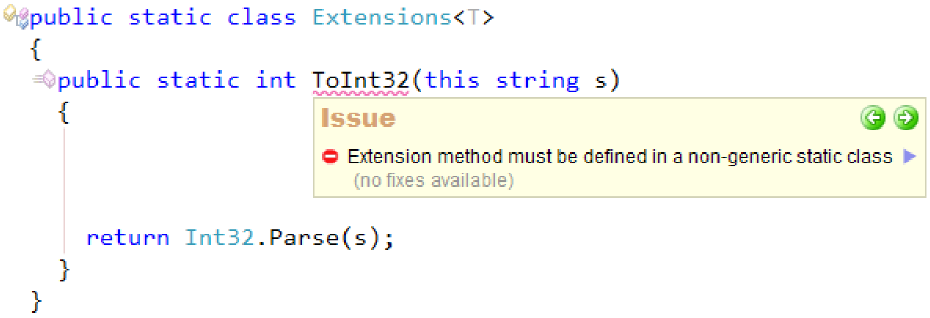
```
public static class Extensions
{
  public static int ToInt32(this string s)
  {
    return Int32.Parse(s);
  }
}
```

Extension method must be defined in a non-generic static class

Cause:

The class holding extension methods must be static and must not be generic.

Sample:



```
public static class Extensions<T>
{
  public static int ToInt32(this string s)
  {
    return Int32.Parse(s);
  }
}
```

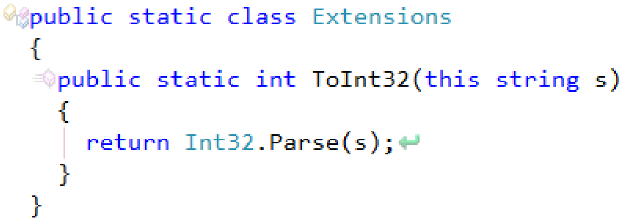
Issue

Extension method must be defined in a non-generic static class

(no fixes available)

How to fix:

- Make the class non-generic, if appropriate:



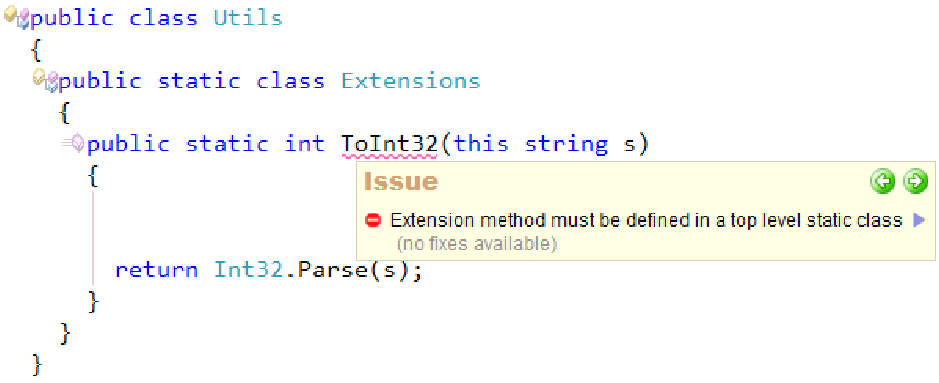
```
public static class Extensions
{
  public static int ToInt32(this string s)
  {
    return Int32.Parse(s);
  }
}
```

Extension method must be defined in a top level static class

Cause:

The class holding extension methods must be top level and cannot be nested.

Sample:



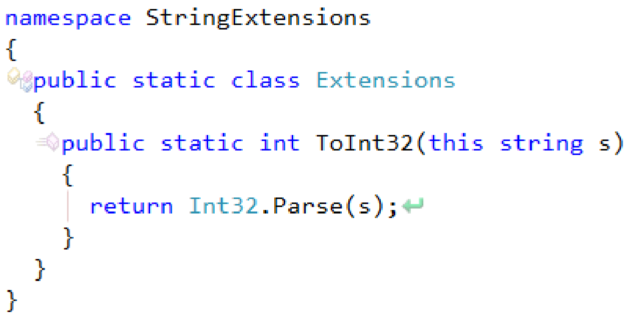
```
public class Utils
  {
    public static class Extensions
    {
      public static int ToInt32(this string s)
      {


        return Int32.Parse(s);
      }
    }
  }
```

Issue

Extension method must be defined in a top level static class
(no fixes available)

How to fix:

- Define the class that contains the extension method as a top-level class, e.g.,:



```
namespace StringExtensions
{
public static class Extensions
  {
  public static int ToInt32(this string s)
    {
    return Int32.Parse(s);
    }
  }
}
```

There are could be other code issues, such as “Extension methods can be called only on instances values” or “A reference to System.Core assembly is missing for declaring an extension method”. However, it is very easy to write your own code issue by creating a new plug-in for DXCore. Read about creating your own code issue using the **CodeIssue** provider DXCore control to learn more.

Code Issues specific to operators

An operator is a member that defines the meaning of applying a particular expression operator to instances of a class or structure. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators. All operators have must be declared according to the language specification.

For example, C# allows you to overload operators by defining static member functions using the operator keyword. Not all operators can be overloaded, and others have restrictions. For example, the signature of an operator should consist of the name of the operator and the type of each of its formal parameters, considered in left to right order. The signature of an operator specifically does not include the result type.

CodeRush includes several code issues that make it possible to declare operators without violating the operator declaration rules at the coding stage.

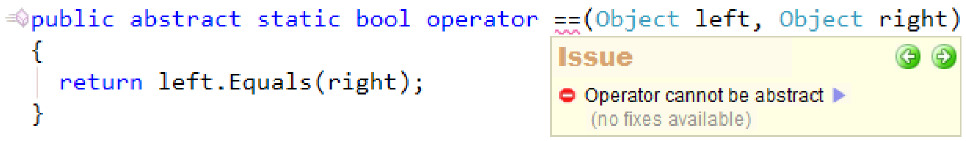
Here they are:

Operator cannot be abstract

Cause:

Operators must always be implemented. It is now allowed to mark operators with an ‘*abstract*‘ keyword. If an operator is marked as an ‘*abstract*‘, the error is shown.

Sample:



```
public abstract static bool operator ==(Object left, Object right)
{
  return left.Equals(right);
}
```

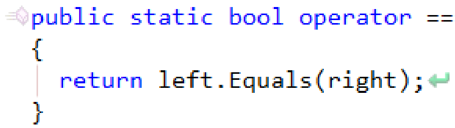
Issue

Operator cannot be abstract

(no fixes available)

How to fix:

- Remove the ‘abstract’ modifier:



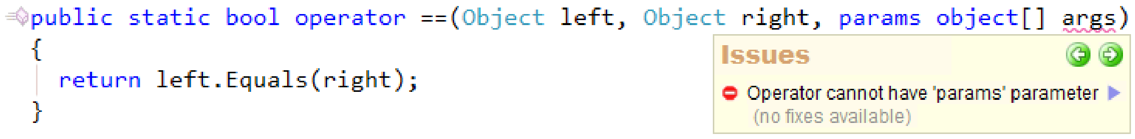
```
public static bool operator ==(Object left, Object right)
{
  return left.Equals(right);
}
```

Operator cannot have the ‘params’ parameter

Cause:

Operators must have an unchanging number of parameters. You cannot have an undefined number of parameters for an operator by using the ‘*params*‘ keyword.

Sample:



```
public static bool operator ==(Object left, Object right, params object[] args)
{
  return left.Equals(right);
}
```

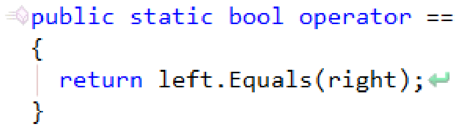
Issues

Operator cannot have 'params' parameter

(no fixes available)

How to fix:

- Remove the ‘params’ parameter:



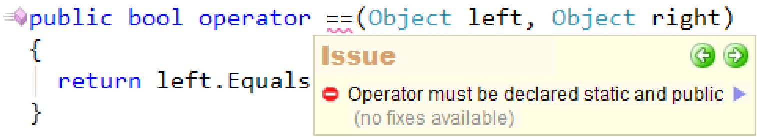
```
public static bool operator ==(Object left, Object right)
{
  return left.Equals(right);
}
```

Operator must be declared static and public

Cause:

According to the language specification, operators are declared as a static member with a ‘*public*‘ visibility modifier.

Sample:



```
public bool operator ==(Object left, Object right)
  {
    return left.Equals
  }
```

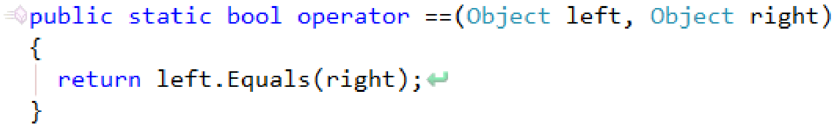
Issue

Operator must be declared static and public

(no fixes available)

How to fix:

- Mark the operator member as static and public:



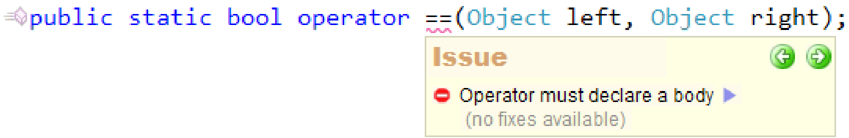
```
public static bool operator ==(Object left, Object right)
  {
    return left.Equals(right);
  }
```

Operator must declare a body

Cause:

Operators not marked with the external modifier must declare a member body. Operators without a code body will be highlighted as an error.

Sample:



```
public static bool operator ==(Object left, Object right);
```

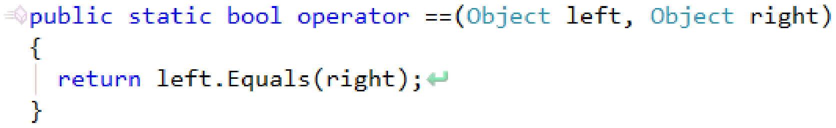
Issue

Operator must declare a body

(no fixes available)

How to fix:

- Declare the body of the operator:



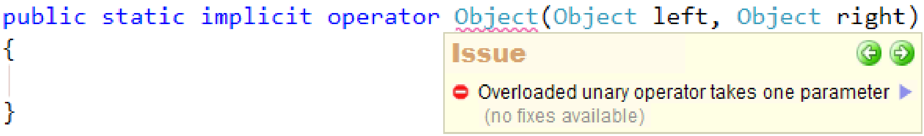
```
public static bool operator ==(Object left, Object right)
  {
    return left.Equals(right);
  }
```

Overloaded unary operator takes one parameter

Cause:

Implicit and explicit unary operators must accept a single parameter. If unary operator takes more than one parameter, an error is shown.

Sample:



```
public static implicit operator Object(Object left, Object right)
{

}
```

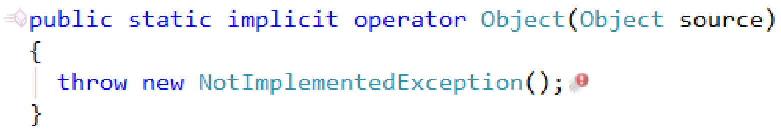
Issue

Overloaded unary operator takes one parameter

(no fixes available)

How to fix:

- Remove the redundant parameter, e.g.,:



```
public static implicit operator Object(Object source)
{
  throw new NotImplementedException();
}
```

Code Issues specific to C# and VB language keywords

We are going to review the CodeRush code issues dedicated to C# language limitations for the following keyewords: ‘base’, ‘this’, ‘yield’, ‘params’ and ‘Me’, ‘ParamArray’ keywords in Visual Basic. Here’s a brief overview of keywords:

The ‘this’ (C#) or ‘Me’ (VB) keyword is used within a class’ code to refer to the current instantiated object.

By using the ‘base’ keyword, a derived class can access members of the base class. Calling the base class members from within an overridden member can be used to combine functionality.

The ‘yield return’ and ‘yield break’ keywords are used in an iterator block to yield a value to the enumerator object or enumerable object of an iterator or to signal the end of the iteration.

The ‘params’ (C#) or ‘ParamArray’ (VB) keywords specify a parameter array declaration which indicates that any number of parameters of the indicated type may be used in the method call, allowing for optional parameters.

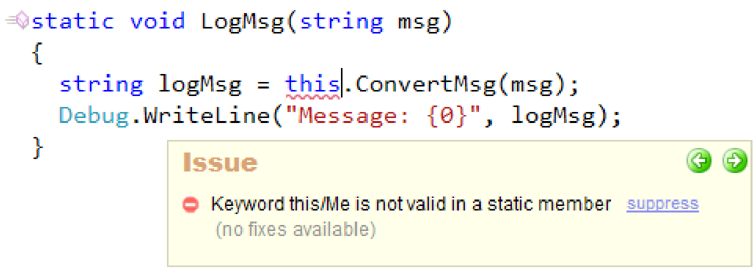
Here are the code issues provided by CodeRush specific to the keywords above.

Keyword this (Me) is not valid in a static member

Cause:

The ‘this’ keyword refers to an object which is an instance of a type. Since static members are independent of any instance of the containing class, the ‘this’ keyword is meaningless and therefore is not allowed.

Sample:



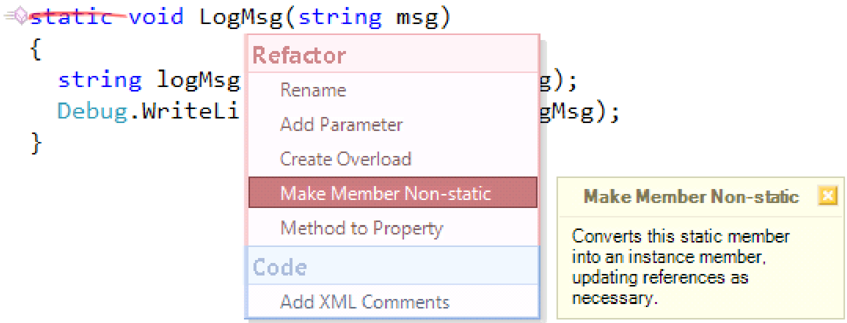
```
static void LogMsg(string msg)
{
  string logMsg = this.ConvertMsg(msg);
  Debug.WriteLine("Message: {0}", logMsg);
}
```

Issue

Keyword this/Me is not valid in a static member suppress
(no fixes available)

How to fix:

- make the initial member non-static by applying the Make Member Non-static refactoring:

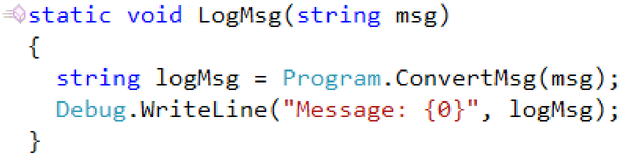


```
~~static~~ void LogMsg(string msg)
{
  string logMsg                       g);
  Debug.WriteLi                       gMsg);
}
```

Make Member Non-static

Converts this static member into an instance member, updating references as necessary.

- remove the ‘this’ keyword and use the name of the current class instead:



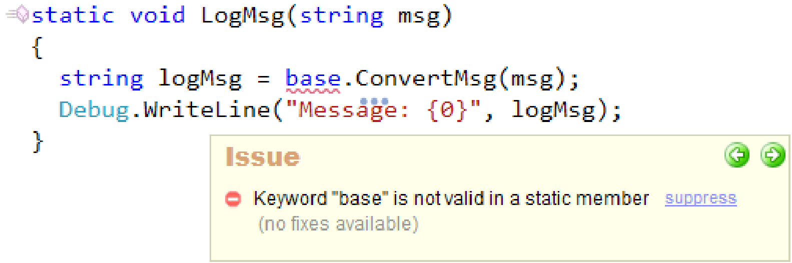
```
static void LogMsg(string msg)
{
  string logMsg = Program.ConvertMsg(msg);
  Debug.WriteLine("Message: {0}", logMsg);
}
```

Keyword base is not valid in a static member

Cause:

This is a similar code issue as above, when static members may not reference members of the base class.

Sample:

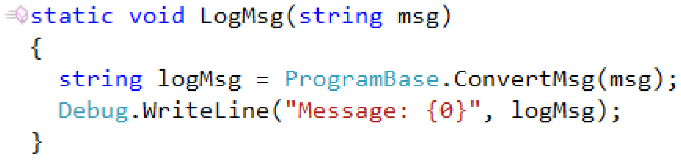


```
static void LogMsg(string msg)
{
  string logMsg = base.ConvertMsg(msg);
  Debug.WriteLine("Message: {0}", logMsg);
}
```

Keyword "base" is not valid in a static member

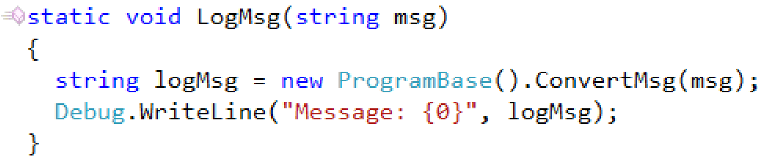
How to fix:

- make the initial member non-static by applying the Make Member Non-static refactoring.
- remove the ‘base’ keyword and use the name of the base class instead if appropriate:



```
static void LogMsg(string msg)
{
  string logMsg = ProgramBase.ConvertMsg(msg);
  Debug.WriteLine("Message: {0}", logMsg);
}
```

- create an instance of the base type and call the required member if possible:



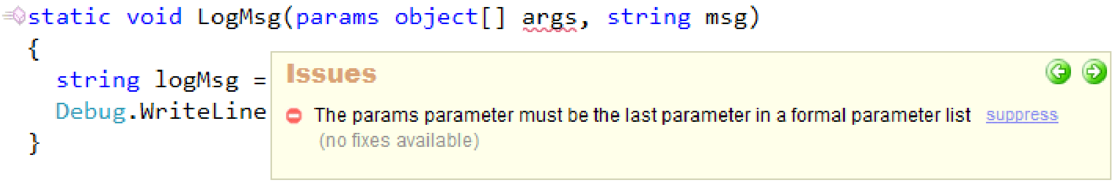
```
static void LogMsg(string msg)
{
  string logMsg = new ProgramBase().ConvertMsg(msg);
  Debug.WriteLine("Message: {0}", logMsg);
}
```

The params parameter must be the last parameter in a formal parameter list

Cause:

The ‘params’ keyword defines an optional array of a variable number of parameters (arguments). There can be only one argument with the ‘params’ keyword, and it must appear at the end of the argument list.

Sample:



```
static void LogMsg(params object[] args, string msg)
{
  string logMsg =
  Debug.WriteLine
}
```

Issues

The params parameter must be the last parameter in a formal parameter list suppress
(no fixes available)

How to fix:

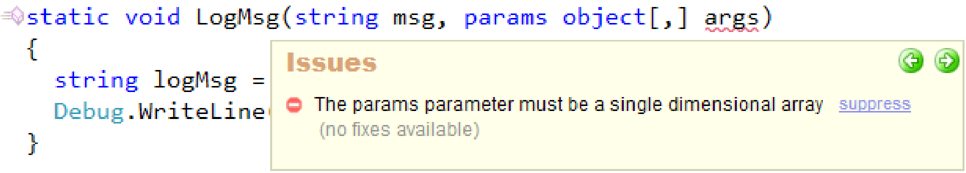
- move the ‘params’ parameter to the end of the parameters list;
- remove the ‘params’ parameter modifier.

The params parameter must be a single dimensional array

Cause:

The ‘params’ keyword defines an optional array of a variable number of parameters. The ‘params’ keyword must describe a single-dimension array according to the language specification.

Sample:



```
static void LogMsg(string msg, params object[,] args)
{
  string logMsg =
  Debug.WriteLine(
}
```

Issues

The params parameter must be a single dimensional array suppress
(no fixes available)

How to fix:

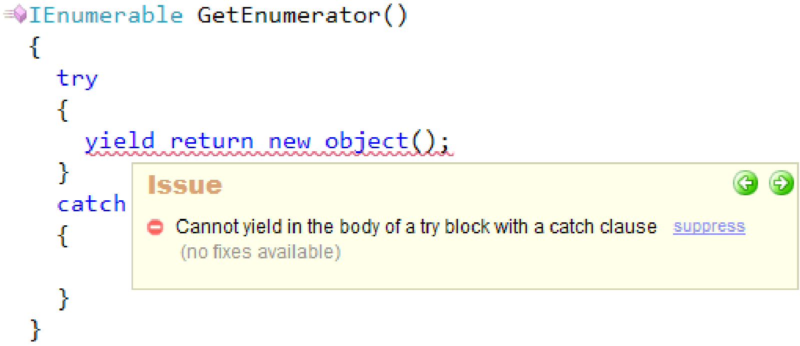
- make the ‘params’ parameter a single-dimension array;
- remove the ‘params’ keyword specifier.

Cannot yield in the body of a try block with a catch clause

Cause:

Yield statements are not permitted inside a try block with a clause. This occurs because there are problems implementing the correct behavior of the iterators in the compiler.

Sample:



```
IEnumerable GetEnumerator()
{
  try
  {
    yield return new object();
  }
  catch
  {

  }
}
```

Cannot yield in the body of a try block with a catch clause
(no fixes available)

How to fix:

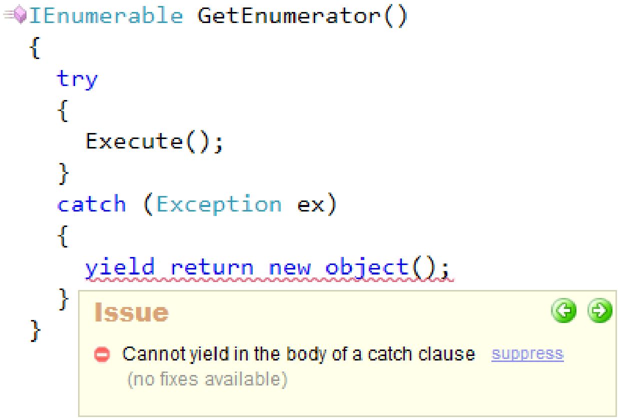
- move the yield statement outside of the try block.

Cannot yield in the body of a catch clause

Cause:

Yield statements are not permitted inside a catch clause body.

Sample:



```
IEnumerable GetEnumerator()
{
  try
  {
    Execute();
  }
  catch (Exception ex)
  {
    yield return new object();
  }
}
```

Cannot yield in the body of a catch clause
(no fixes available)

How to fix:

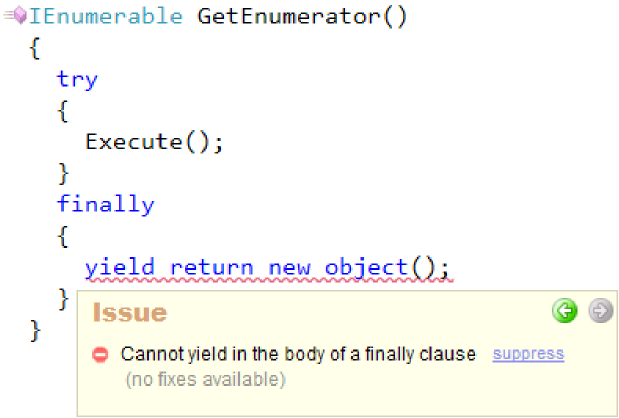
- move the yield statement outside of the catch clause body.

Cannot yield in the body of a finally clause

Cause:

Yield statements are not permitted inside a finally clause body.

Sample:



```
IEnumerable GetEnumerator()
  {
    try
    {
      Execute();
    }
    finally
    {
      yield return new object();
    }
  }
```

Cannot yield in the body of a finally clause

(no fixes available)

How to fix:

- move the yield statement outside of the finally clause body.

Try statement without catch or finally

Cause:

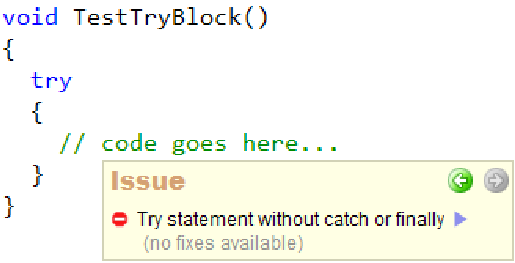
This code issue shows an error if a ‘try’ statement doesn’t have a ‘catch’ or ‘finally’ statement. The try block must be completed, whether with catch or finally blocks.

The ***try*** block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully.

The ***catch*** block is used to take a control when any or a specific exception has been thrown.

The ***finally*** block is useful for cleaning up any resources allocated in the try block, as well as running any code that must execute even if there is an exception. Control is always passed to the finally block regardless of how the try block exits.

Sample:



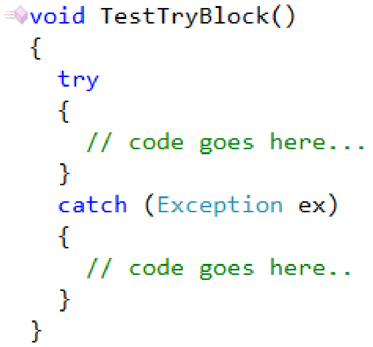
```
void TestTryBlock()
{
  try
  {
    // code goes here...
  }
}
```

Try statement without catch or finally

(no fixes available)

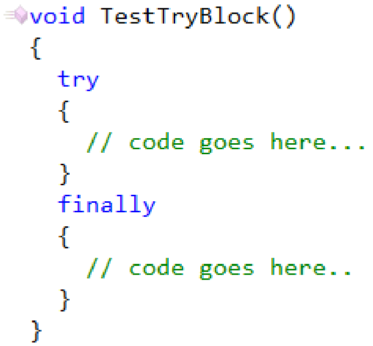
How to fix:

- Add the missing catch block:



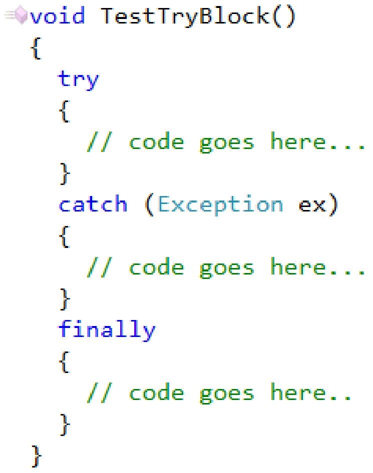
```
void TestTryBlock()
{
  try
  {
    // code goes here...
  }
  catch (Exception ex)
  {
    // code goes here..
  }
}
```

- Add the missing finally block:



```
void TestTryBlock()
{
  try
  {
    // code goes here...
  }
  finally
  {
    // code goes here..
  }
}
```

- Add the missing catch and finally blocks:



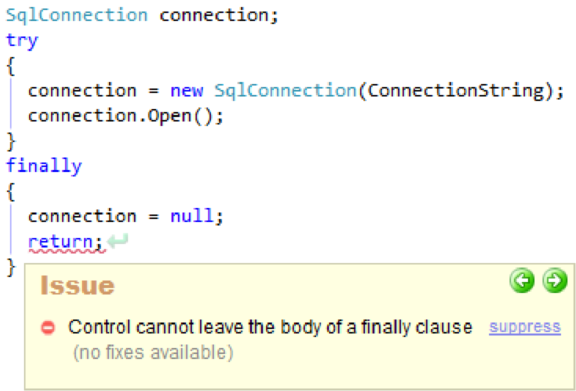
```
void TestTryBlock()
{
  try
  {
    // code goes here...
  }
  catch (Exception ex)
  {
    // code goes here...
  }
  finally
  {
    // code goes here..
  }
}
```

Control cannot leave the body of a finally clause

Cause:

Returns are forbidden in finally clauses by the compiler. A compiler error will occur in case of a return statement inside a finally block, and refuse to let you write such potentially ambiguous and confusing code. The purpose of a finally statement is to ensure that the necessary cleanup of objects happens immediately and always. In other words, it is designed for releasing all locks and resources and cannot leave the block until finishing the cleanup task.

Sample:



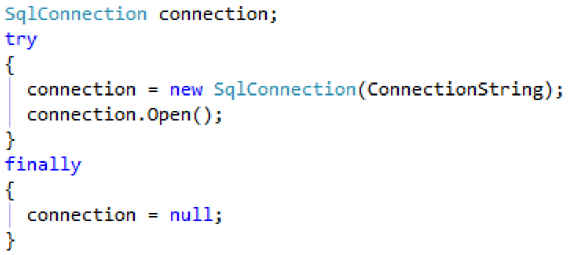
```
SqlConnection connection;
try
{
    connection = new SqlConnection(ConnectionString);
    connection.Open();
}
finally
{
    connection = null;
    return;
}
```

Issue

Control cannot leave the body of a finally clause suppress
(no fixes available)

How to Fix:

- Remove the return statement from a finally block:



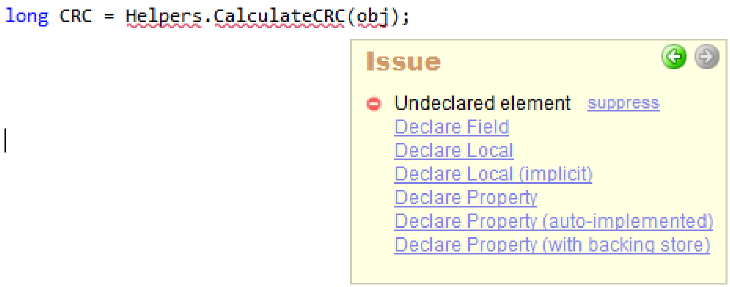
```
SqlConnection connection;
try
{
    connection = new SqlConnection(ConnectionString);
    connection.Open();
}
finally
{
    connection = null;
}
```

Undeclared element

Cause:

Highlights identifier references to local variables, fields, method, properties, classes, structures, interfaces and everything that is not yet declared.

Sample:



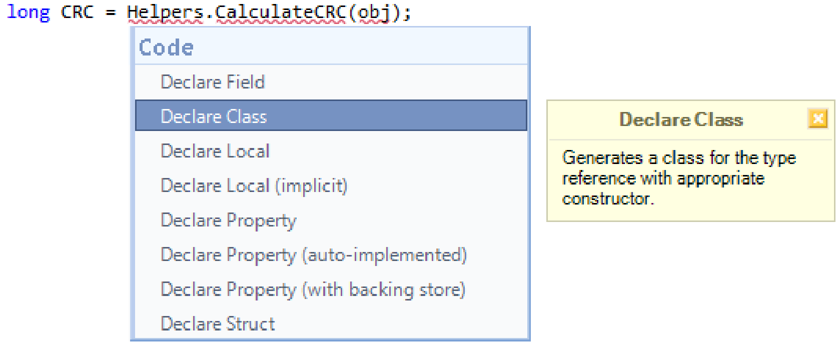
```
long CRC = Helpers.CalculateCRC(obj);
```

Issue

Undeclared element

How to Fix:

- Apply the Declare XXX code providers, e.g.:



```
long CRC = Helpers.CalculateCRC(obj);
```

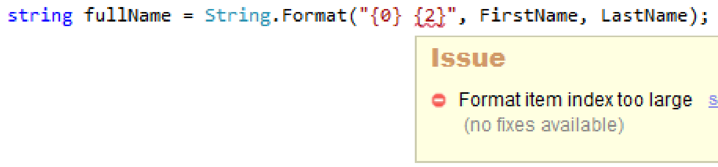
Generates a class for the type reference with appropriate constructor.

Format item index too large

Cause:

In the String.Format call you can specify any number of arguments and the corresponding format items, where those argument values will be inlined. When the corresponding argument in the call is absent, the FormatException will be thrown (Index (zero based) must be greater than or equal to zero and less than the size of the argument list). The compiler does not validate the format items, so this may lead to errors in your application. This code issue allows you to avoid such errors by highlighting format items with an incorrect index.

Sample:



```
string fullName = String.Format("{0} {2}", FirstName, LastName);
```

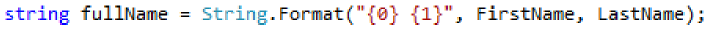
Issue

Format item index too large

(no fixes available)

How to Fix:

- Remove or correct the index of the format item:



```
string fullName = String.Format("{0} {1}", FirstName, LastName);
```

Chapter 3. Code-generation templates

CodeRush Code Templates overview

CodeRush code templates allow you to generate large code blocks on the fly with just a few keystrokes. The use of code templates dramatically decreases the code writing time, because it's not necessary to type the entire block of code manually. The Templates library shipped with CodeRush contain lots of code templates for most code blocks and regular coding structures including regions and comments.

There are several basic things to know about templates:

Template expansion

Template expansion is the resulting code or text generated after a particular template is expanded. The expansion can be of any-size (e.g. thousands lines of code) and can be interactive (e.g. you can interactively change specific parts and navigate between them in the expansion).

You can think of a template expansion as a ready code block like a member (e.g. a method or property), a statement (e.g. if, for, foreach, switch, etc) or an expression (e.g. an equality comparison, a value or property assignment). The Templates library contains hundreds of code templates for regular code blocks for all supported programming languages. Of course, you are able to create your own templates with ease.

Due to the context-sensitive nature of templates, a single template may be expanded differently depending on the current environment and conditions (context).

Template name

The abbreviation (name) used to identify the template and its expected expansion. CodeRush has its own templates language that is pretty easy to learn and understand. When you know the templates language and expand templates for code generation, your coding speed is increased significantly. The name of the template can contain one or more characters.

Template key

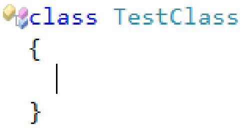
The keyboard shortcut that is used to expand the template. When you type the template name in the code editor and press the Template key, the template name is replaced with its expansion.

The Space bar is used by CodeRush for the maximum coding speed by default. Some templates may use an alternative key that is a Shift+Space. Both template keys are configurable. You can setup your own template key (like Tab, for example) if any templates interfere with your coding habits.

Code Templates samples

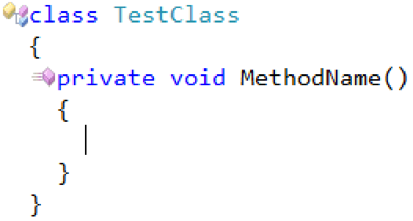
Here are example of the simplest code templates:

m – expands into a method inside a body of a class:



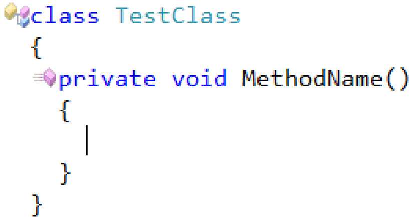
```
class TestClass
  {
    |
  }
```

f – expands into a for loop inside a body of a member:



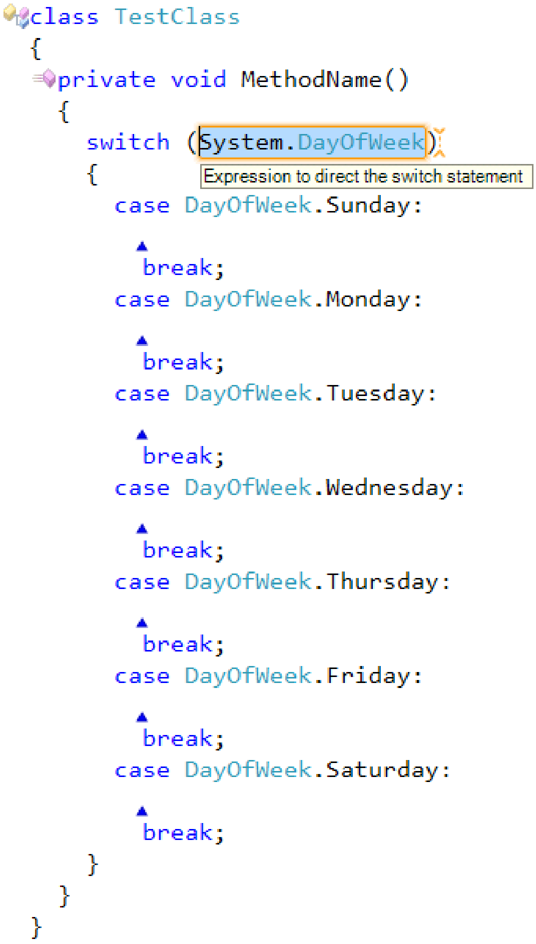
```
class TestClass
  {
    private void MethodName()
      {

      }
  }
```

sw – expands into a switch statement inside a body of a member:



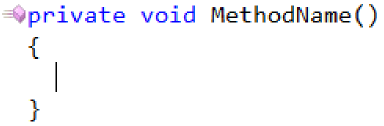
```
class TestClass
  {
    private void MethodName()
      {

      }
  }
```

Due to the context-sensitive nature of templates, the ‘sw’ template can be expanded differently. For example, if the Clipboard contains a variable name of the Enumeration type (e.g. *System.DayOfWeek*), you will get the following code generated:



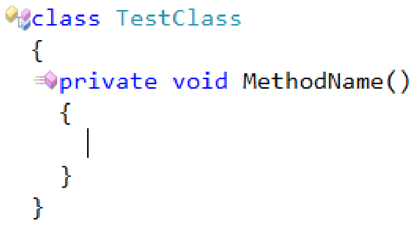
```
class TestClass
{
  private void MethodName()
  {
    switch (System.DayOfWeek)
    {
      case DayOfWeek.Sunday:

        break;
      case DayOfWeek.Monday:

        break;
      case DayOfWeek.Tuesday:

        break;
      case DayOfWeek.Wednesday:

        break;
      case DayOfWeek.Thursday:

        break;
      case DayOfWeek.Friday:

        break;
      case DayOfWeek.Saturday:

        break;
    }
  }
}
```

Most of the code templates from the CodeRush Templates Library have short names (m, f, sw) and must be protected from an unexpected expansion with a correct context set. For example, if you prefer to use a single-letter variable name when you declare it and then press the Space bar, the template with the same name will not be expanded:



```
private void MethodName()
{
  |
}
```

However, in other situations (with a different context) the template will be expanded as expected:

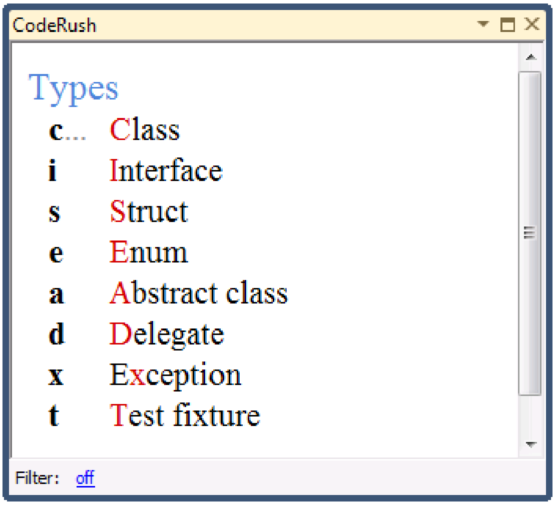


```
class TestClass
{
  private void MethodName()
  {
    |
  }
}
```

If you see unexpected template expansions, you might refer to the “Resolving templates conflicts” topic where you can learn how to avoid unexpected template expansions.

Templates language and learning basics

CodeRush code templates are easy to understand and learn. One of the easiest ways is to dock the CodeRush Training window inside the Visual Studio IDE. This window shows you the available templates for the current context. For example, if the editor caret is inside a namespace, the window will show you the following templates suggestion:

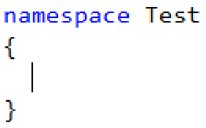


Types

c...	Class
i	Interface
s	Struct
e	Enum
a	Abstract class
d	Delegate
x	Exception
t	Test fixture

Filter: off

As you see, the first letter (highlighted in red) of the type declaration is used as a template name. If you’d like to declare an interface, you need to type the letter ‘i’ inside a namespace:



```
namespace Test
{
  |
}
```

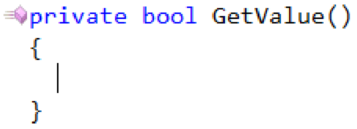
This is some kind of a rule for template naming – use abbreviations or mnemonics. In our sample, the first letter of the type is used. If the letter is already used for some template (e.g. “Enum” and “Exception”), the second letter is used to name a template instead.

Here are advanced examples of template names:

- **ea** – **E**vent**A**rgs
- **eh** – **E**vent**H**andler
- **os** – **o**bject **s**ender

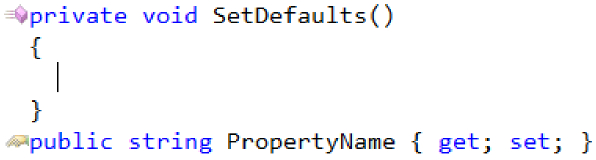
The template name usually sounds like the template expansion. Consider the following examples:

- **rt** – **r**eturn **t**rue:



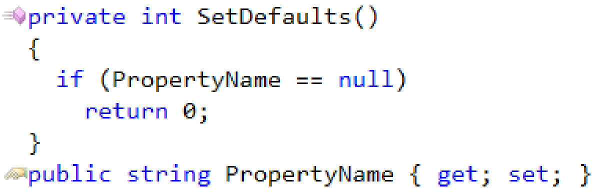
```
private bool GetValue()
{

}
```

- **sn** – **s**et **n**ull:



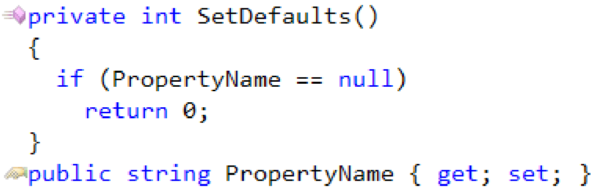
```
private void SetDefaults()
{

}
public string PropertyName { get; set; }
```

- **inrf** – **i**f **n**ull, **r**eturn **f**alse:



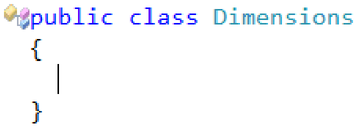
```
private int SetDefaults()
{
  if (PropertyName == null)
    return 0;
}
public string PropertyName { get; set; }
```

- **inr0**- **i**f **n**ull, **r**eturn **z**ero:

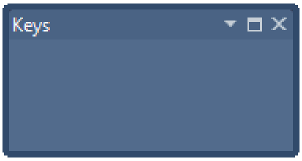


```
private int SetDefaults()
{
  if (PropertyName == null)
    return 0;
}
public string PropertyName { get; set; }
```

- **cc** – **C**reate **C**onstructor



```
public class Dimensions
{
  |
}
```



These samples show that the following grammatical structures are common in CodeRush code templates:

- noun
- verb noun

Once some templates are used several times, you may acquire a habit to use those ones, because it is one of the fastest ways to write different kind of code blocks. And when you write code fast, you safe your time.

Dynamic templates and dynamic lists

Template **dynamic lists** make a single code template universal and dynamic. For instance, a template for a variable declaration can be configurable on the fly by specifying its type. In other words, a template can be expanded differently depending on the selected dynamic list. A dynamic list contains a list of possible template alternative expansions (types when used with a variable template) for a single template.

Here are some of the standard dynamic lists:

- System .NET types (e.g., System.Boolean, System.Char, System.Int32, System.DateTime, System.Exception, System.IO.StreamReader, etc.)
- Generic Types with one param (e.g., System.Collections.Generic.List)
- Generic Types with two params (e.g., System.Collections.Generic.Dictionary)
- WPF Types (e.g., System.Windows.Media.Color, System.Windows.DependencyObject)
- etc

Each dynamic list has a caption, variable name and a comment. The variable name identifies a dynamic list and is specified for a static template to make it dynamic. When you create a dynamic template, a variable name of the dynamic list is used in combination with the static template mnemonic as follows:

TM + DLVN

where:

- TM – Template Mnemonic
- DLVN – Dynamic List Variable Name

Several dynamic lists may have the same variable name for identification but different captions for better dynamic list organization on the Dynamic Lists option page in the Options Dialog.

A dynamic list contains entries with Key and Value fields. A Key specifies a dynamic entry mnemonic name used in combination with the static template mnemonic. To expand a dynamic template, use the Template Mnemonic (TM) with a dynamic entry Key name. A Value is the entry value used in the template expansion. Consider the following dynamic list entries:

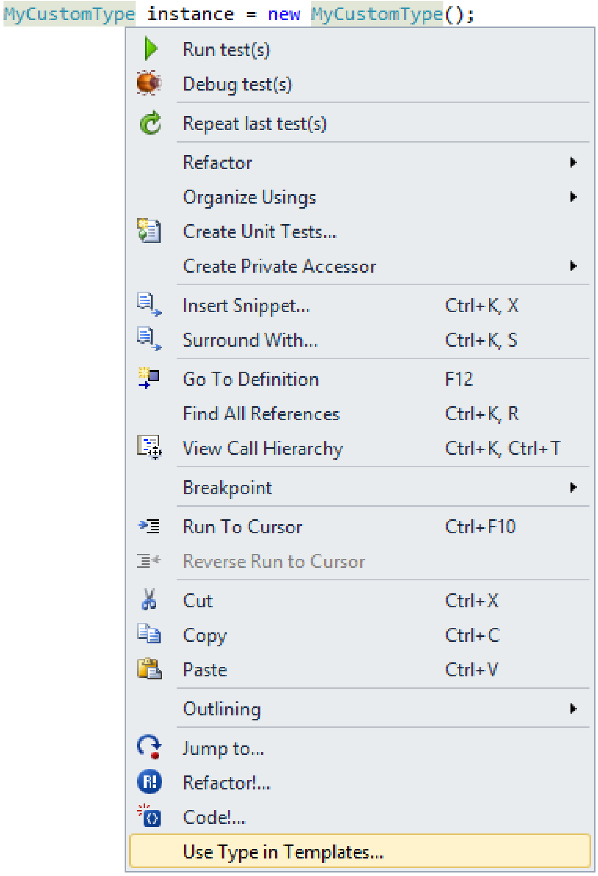
Key	Value
c	System.Char
i	System.Int32
s	System.String
\	Paste the Clipboard content
/	Insert the current type name
?	Insert the current method return type

Now, remember the 'm' template for methods. Combining the 'm' mnemonic with the Key dynamic list entries, we can expand the following methods:

mc – a method returning a value of the char type;
mi – a method returning a value of the integer type;
ms – a method returning a value of the string type;
m\ – a method returning a value of the type on the Clipboard;
m/ – a method returning a value of the current type;
etc.

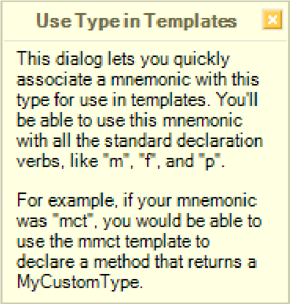
Here, a single template 'm' can have as many different expansions as dynamic list entries that exist. Imagine that other templates may reuse this dynamic list and dynamic list entries are configurable and extensible. This makes the CodeRush templates collection grow dramatically reflecting the number of available code templates.

Don't forget that you can add dynamic lists not only on the Dynamic Lists option page, but also right from the code editor window. Right-click the type declaration name or a type reference and select the **Use Type In Templates…** menu item:



```
MyCustomType instance = new MyCustomType();
```

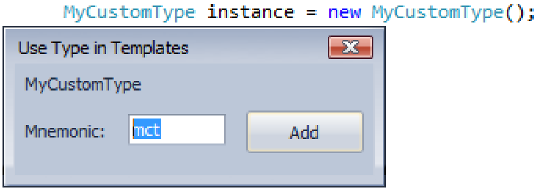
Then, specify the dynamic list entry mnemonic:



Use Type in Templates

This dialog lets you quickly associate a mnemonic with this type for use in templates. You'll be able to use this mnemonic with all the standard declaration verbs, like "m", "f", and "p".

For example, if your mnemonic was "mct", you would be able to use the mmct template to declare a method that returns a MyCustomType.



```
MyCustomType instance = new MyCustomType();
```

Use Type in Templates

MyCustomType

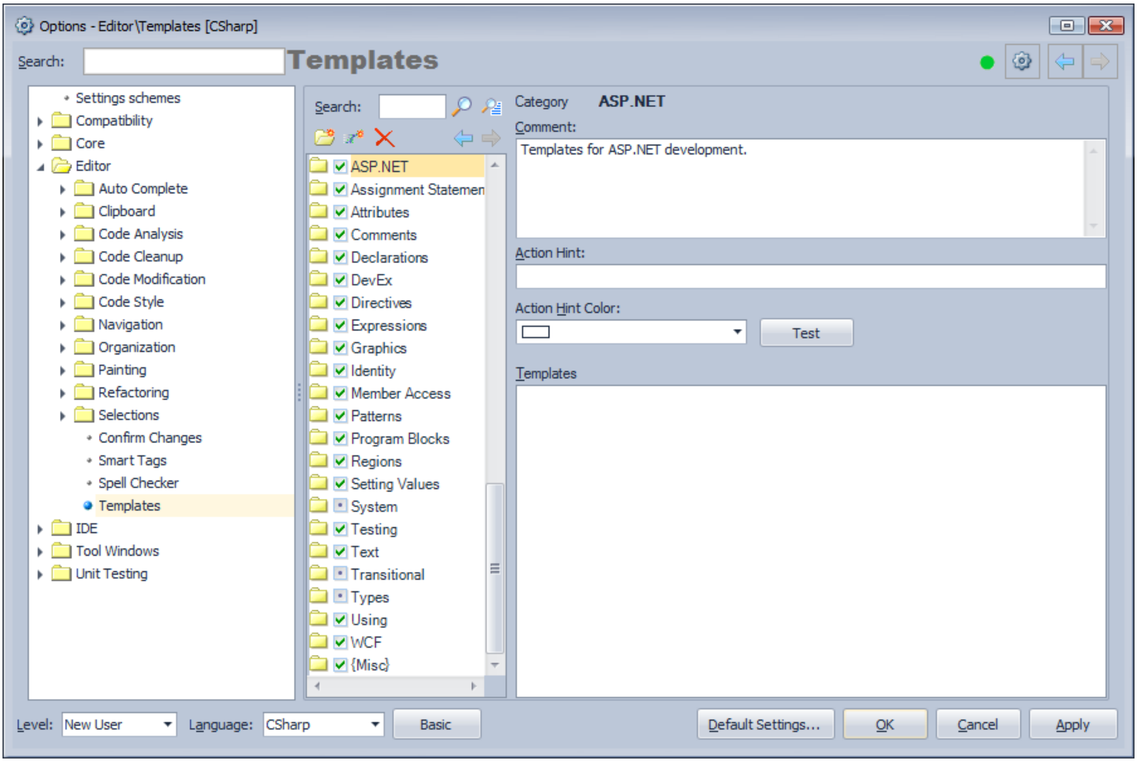
Mnemonic: mct

And you are done! Now, you can use this mnemonic in combination with all other existing standard and custom code templates. Custom mnemonics are added into the corresponding Custom Types dynamic list.

A great helper to remember static and dynamic template is to dock the CodeRush Training Window, which lists all available code templates in the current context.

Templates option page

The **Templates** option page is required to control your collection of built-in and custom CodeRush code templates. When you open the Templates option page, CodeRush will locate the last template you used by default. This is an excellent way to explore and learn how templates are built. When a template expands, if you want to see how it is built, just bring up the Template options page and the last template expanded will be displayed. Here is what it looks like:



Options - Editor\Templates [CSharp]

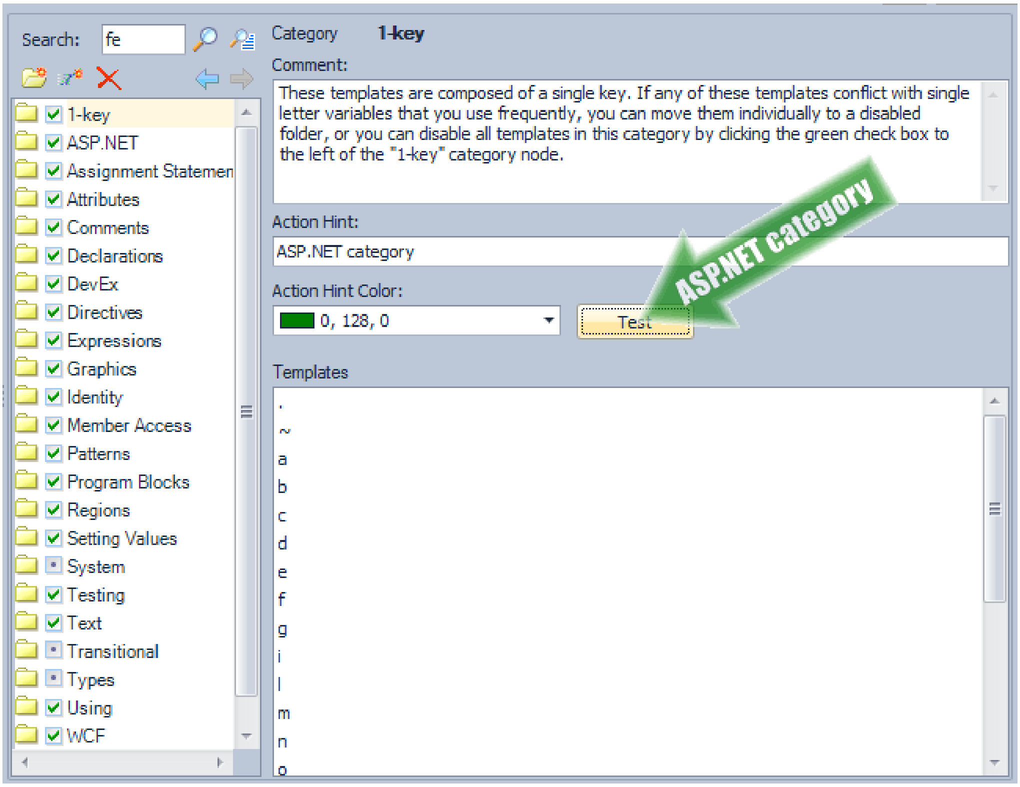
Category ASP.NET

Comment: Templates for ASP.NET development.

Level: New User

Language: CSharp

Templates are stored in folders (categories). Each category has a configurable name and a comment. You can also specify the Action Hint caption and its color for each particular template category. Click the Test button to see what it will look like:



Search: fe

Category 1-key

Comment:

These templates are composed of a single key. If any of these templates conflict with single letter variables that you use frequently, you can move them individually to a disabled folder, or you can disable all templates in this category by clicking the green check box to the left of the "1-key" category node.

Action Hint: ASP.NET category

Action Hint Color: 🟩 0, 128, 0

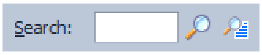
Templates

.
~
a
b
c
d
e
f
g
i
l
m
n
o

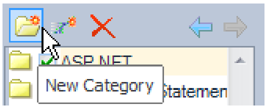
The Templates list shows a plain list of templates included into the selected folder and sub-folders.

Near the category name you can find a checkbox which enables or disables the entire folder with templates. To rename a category or template, double-click it in the main tree. A folder may contain nested folders for better template organization.

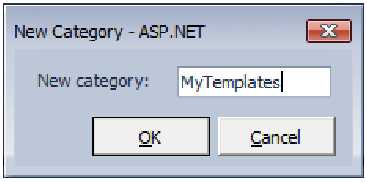
On the top of the option page there is a Search text box which allows you to find templates by name (abbreviation) or by its content (expansion). Click one of the two buttons near the text box to search in the preferred way:



Under the Search text box there are tool buttons to create a new nested folder, to create a new code template, or delete a folder or template. Plus, there are two navigation button for navigating by the history of visited templates and folders backwards and forwards. All buttons have tooltips to learn more about their purpose:

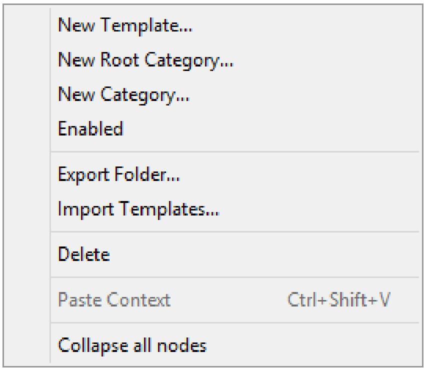


Clicking the New Category button will show you a dialog where you can type the name of the new nested category that will be added inside the selected category:

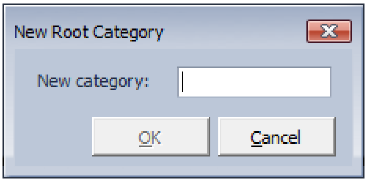


New category: MyTemplates

If you'd like to create a non-nested Category in the root, use the context menu for this task. A context menu on a folder looks like this:



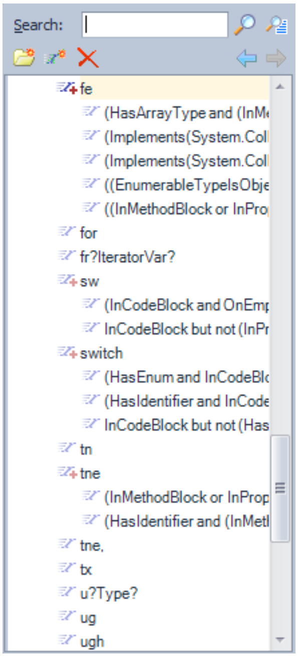
Clicking the New Root Category shows a similar dialog as a New Category:



The other context menu items are:

Menu item	Description
New Template…	Shows a New Template dialog allowing you to create a new template with the desired name.
New Root Category…	Shows a New Root Category dialog allowing you to create a new template with the desired name.
New Category…	Shows a New Category dialog allowing you to create a nested category for the currently selected folder.
Enabled	Toggles the Enabled state for the selected folder.
Move to Root	Makes the selected nested folder the root folder.
Export Folder…	Allows you to export the selected folder.
Import Templates…	Allows you to import previously exported templates.
Delete	Removes the selected category.
Paste Context	Sets the previously copied Context to all templates inside the selected folder.
Collapse all nodes	Collapses all folder nodes.

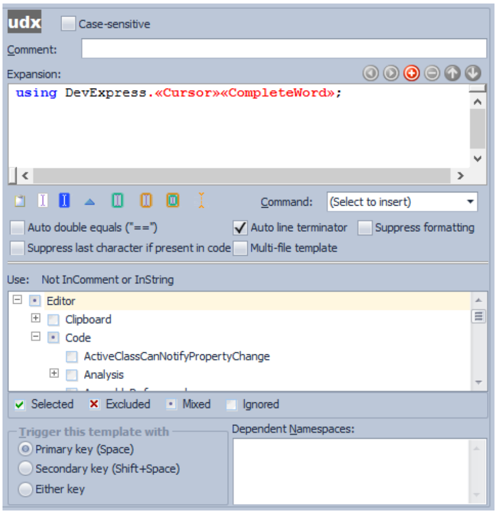
Of course, code templates are stored inside the folders. A template has a configurable name, a comment, an expansion, a context, and numerous options. Often, templates have alternate expansions (different versions of the template bound to the same character sequence). Alternate expansions are indicated by a special icon in the list and are differentiated by their context which appears in the tree view:



```
fe
    (HasArrayType and (InM
    (Implements(System.Col
    (Implements(System.Col
    ((EnumerableTypeIsObje
    ((InMethodBlock or InPro
for
fr?IteratorVar?
sw
    (InCodeBlock and OnEmp
    InCodeBlock but not (InPr
switch
    (HasEnum and InCodeBl
    (HasIdentifier and InCode
    InCodeBlock but not (Has
tn
tne
    (InMethodBlock or InProp
    (HasIdentifier and (InMetl
tne,
tx
u?Type?
ug
ugh
```

If you see alternate expansions, look at the contexts to see which one of those expansions will be expanded. Alternate expansion contexts are evaluated in the order you see them in the list, and often a list of alternate expansions will end with a template that has no context defined. That means that it will expand if none of the previous alternate expansion contexts were satisfied.

Here is what the templates configuration page looks like when a particular template is selected:


udx □ Case-sensitive

Comment:

Expansion:

```
using DevExpress.«Cursor»«CompleteWord»;
```

Command: (Select to insert)

□ Auto double equals ("==")   ☑ Auto line terminator   □ Suppress formatting

□ Suppress last character if present in code   □ Multi-file template

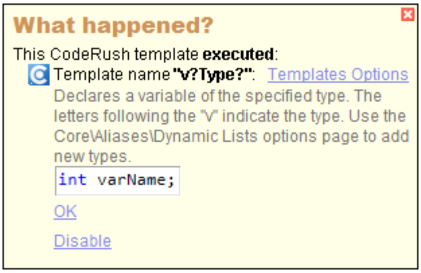
Use: Not InComment or InString

```
⊟ ▪ Editor
    ⊞ □ Clipboard
    ⊟ ▪ Code
        □ ActiveClassCanNotifyPropertyChange
        ⊞ □ Analysis
```

Trigger this template with
● Primary key (Space)
○ Secondary key (Shift+Space)
○ Either key

Dependent Namespaces:

On the top of the template configuration page, there is a Case-sensitive option which specifies whether or not the name of the template is case-sensitive. A comment is used to identify and describe the template. For instance, the What Happened window uses this comment when a template is expanded:



What happened?

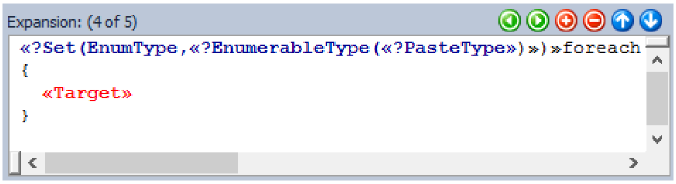
This CodeRush template **executed**:

Template name **"v?Type?"**: Templates Options

Declares a variable of the specified type. The letters following the "V" indicate the type. Use the Core\Aliases\Dynamic Lists options page to add new types.

```
int varName;
```

The Expansion text edit contains the primary template content. The result of a template is composed in this text area by adding the usual text, text commands and/or string providers. It also has several tool buttons to navigate between alternate expansions of a template, to add or remove alternate expansions, or to set their priority by moving an alternate expansion up or down:

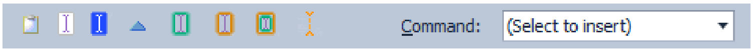


Expansion: (4 of 5)

```
«?Set(EnumType,«?EnumerableType(«?PasteType»)»)»foreach
{
  «Target»
}
```

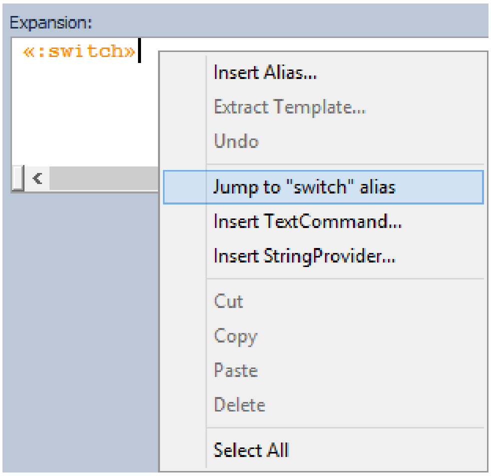
The Expansion is the most important part of a template. The result of a template is composed in this text area by adding the usual text, text commands and/or string providers.

The next set of tool buttons provides a quick way to insert most important text commands. Other commands are listed in the Command combobox:



Command: (Select to insert)

If you see an orange text inside a template expansion, this indicates a call to another template (known as a template alias). You can right-click the template editor to display a menu to jump to any template aliases referenced inside the active template:



Expansion:

```
«:switch»
```

An expansion context menu also contain the following menu items:

Menu item	Description
Insert Alias…	Shows the Select Template dialog which allows you to choose a template that will be used as an alias.
Extract Template…	Allows you to select a part of the template expansion and extract it as a new template. This option will show you a small dialog where it is required to enter a new name for the template.
Undo	Performs a simple undo operation which reverts back the last changes to the template expansion.
Jump to XXX alias	Navigates to the alias in question.
Insert TextCommand…	Shows the Select TextCommand dialog where all text commands are listed and can be filtered.
Insert StringProvider…	Shows the Select StringProvider dialog where all string providers are listed and can be filtered.
Cut	Cuts the selected text from the template expansion text edit.
Copy	Copies the selected text from the template expansion text edit.
Paste	Inserts the text from the Clipboard into the template expansion text edit.
Delete	Removes the selected text from the template expansion text edit.
Select All	Selects the entire context of the template expansion text edit.

Under the template Expansion text edit, you can specify the advanced options for a template:

[ ] Auto double equals (“==”)

Specifies whether or not to insert an additional equals ‘=’ character. This might be useful for languages like CSharp where the double equals operation is used to compare values.

[ ] Auto line terminator

Specifies whether or not to append a line terminator (a semicolon in C#, C++) to the resulting expansion.

[ ] Suppress formatting

Specifies whether or not to disable the automatic code formatting for this template, so it will be expanded “as is”.

[ ] Suppress the last character if present in code

Specifies whether or not to suppress the last character if the same character exists in code.

[ ] Multi-file template

Specifies whether or not a template will be expanded into multiple files, so a special multi-file undo will be created in order to undo those multi-file expansions.

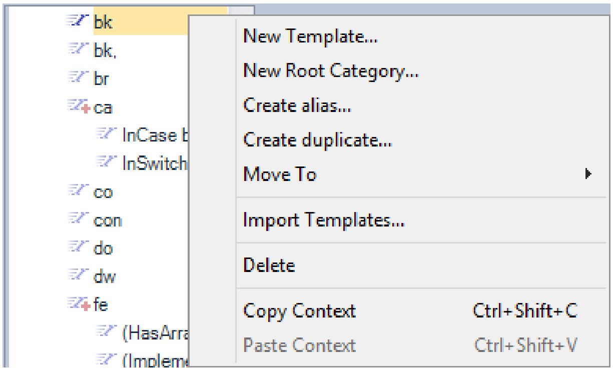
Another important part of a template is a context. A context specifies the conditions when a particular template is available and can be expanded.

Additional options include:

- A template trigger key indicating whether it is a primary key, a secondary key or either.

- Dependent Namespaces. The list of namespace references (e.g., System.Windows.Forms) which is required for the types from the template expansion. Namespace references will automatically be added to the corresponding usings/Imports section of the file where a template has been expanded.

When you add new templates or alternate expansions, or organize your custom templates, you can use the templates context menu and find a few useful options there:



Some of the menu items are the same as the menu items inside the Category context menu, but there are also additional items specific to templates:

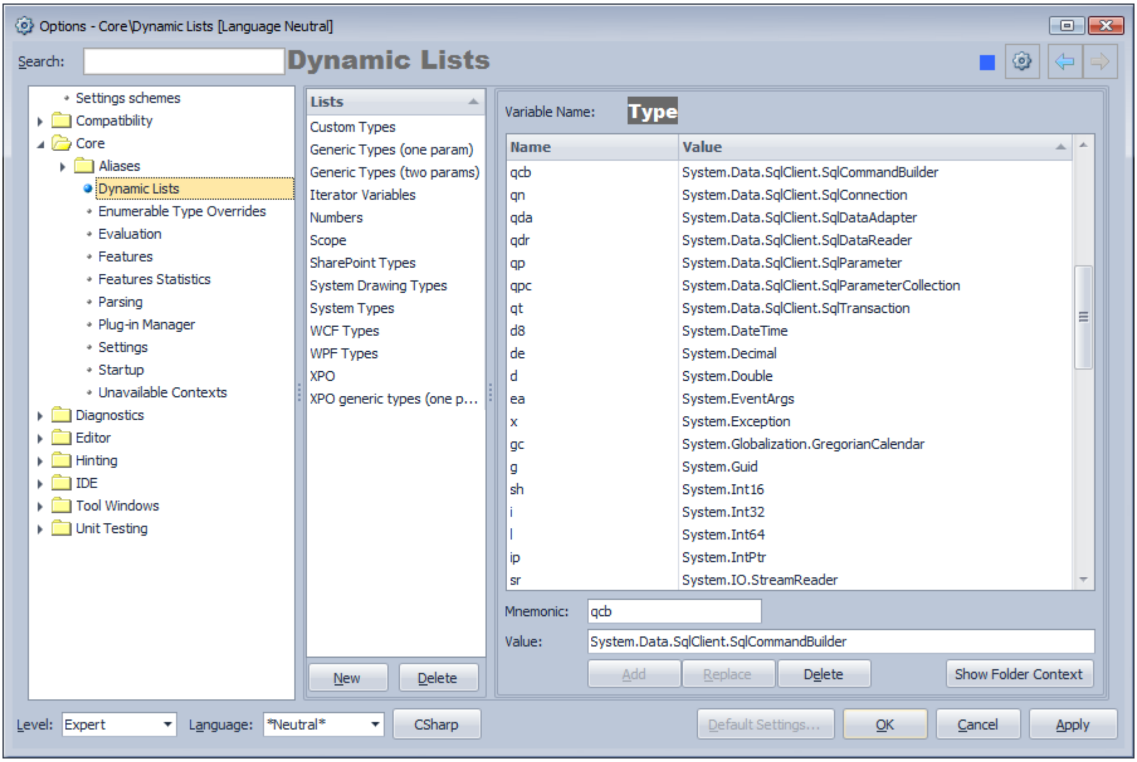
Menu item	Description
Create alias…	Creates a new alias for the selected template. Shows a dialog where an alias name should be specified.
Create duplicate…	Creates a copy of the selected template with the specified name.
Move to ->	Moves the selected template to a different folder chosen from the popup menu.
Copy Context	Copies the context of the selected template for further insertion in another template or a category.

The templates and categories are stored inside their own XML files with a specific file-format. These files are separated by languages, e.g.:

Templates.CSharp.xml
Templates.Basic.xml
Templates.JavaScript.xml

Dynamic Lists option page

Dynamic lists are configurable on the **Dynamic Lists** option page. The page is language-dependent, so you can create and use language-specific dynamic lists. Here is what it looks like:



Options - Core\Dynamic Lists [Language Neutral]

Search:

Dynamic Lists

- Settings schemes
- Compatibility
- Core
  - Aliases
  - Dynamic Lists
  - Enumerable Type Overrides
  - Evaluation
  - Features
  - Features Statistics
  - Parsing
  - Plug-in Manager
  - Settings
  - Startup
  - Unavailable Contexts
- Diagnostics
- Editor
- Hinting
- IDE
- Tool Windows
- Unit Testing

Lists

- Custom Types
- Generic Types (one param)
- Generic Types (two params)
- Iterator Variables
- Numbers
- Scope
- SharePoint Types
- System Drawing Types
- System Types
- WCF Types
- WPF Types
- XPO
- XPO generic types (one p…

Variable Name: Type

Name	Value
qcb	System.Data.SqlClient.SqlCommandBuilder
qn	System.Data.SqlClient.SqlConnection
qda	System.Data.SqlClient.SqlDataAdapter
qdr	System.Data.SqlClient.SqlDataReader
qp	System.Data.SqlClient.SqlParameter
qpc	System.Data.SqlClient.SqlParameterCollection
qt	System.Data.SqlClient.SqlTransaction
d8	System.DateTime
de	System.Decimal
d	System.Double
ea	System.EventArgs
x	System.Exception
gc	System.Globalization.GregorianCalendar
g	System.Guid
sh	System.Int16
i	System.Int32
l	System.Int64
ip	System.IntPtr
sr	System.IO.StreamReader

Mnemonic: qcb

Value: System.Data.SqlClient.SqlCommandBuilder

Level: Expert

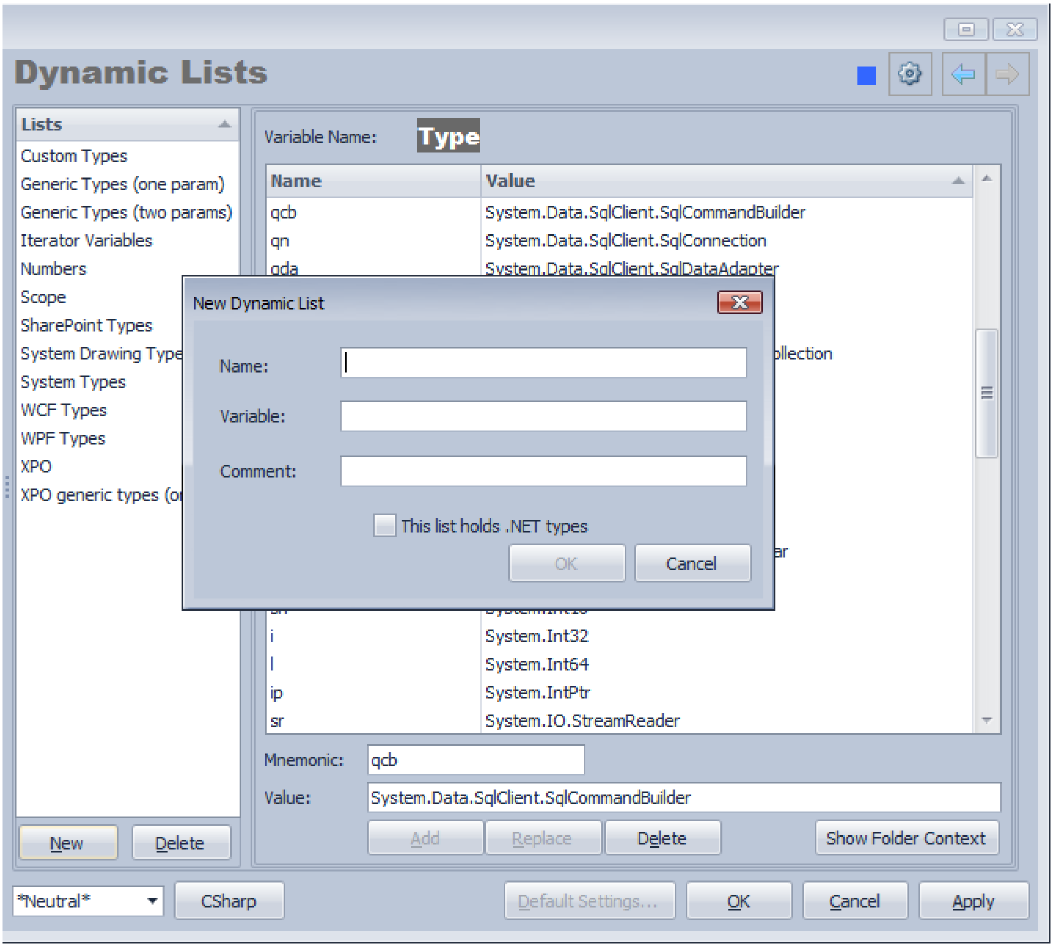
Language: *Neutral*

The option page composed of two parts:

- The list of available dynamic lists for the selected language.
- The list of dynamic list entries for the selected dynamic list.

Most of the available built-in dynamic lists are common for all languages. So, they are listed under the *Neutral* language.

You can add a new dynamic list by clicking the New button under the list of dynamic lists. The following dialog appears:



Lists
- Custom Types
- Generic Types (one param)
- Generic Types (two params)
- Iterator Variables
- Numbers
- Scope
- SharePoint Types
- System Drawing Type
- System Types
- WCF Types
- WPF Types
- XPO
- XPO generic types (o

Variable Name: Type

Name	Value
qcb	System.Data.SqlClient.SqlCommandBuilder
qn	System.Data.SqlClient.SqlConnection
qda	System.Data.SqlClient.SqlDataAdapter
i	System.Int32
l	System.Int64
ip	System.IntPtr
sr	System.IO.StreamReader

Mnemonic: qcb

Value: System.Data.SqlClient.SqlCommandBuilder

New Dynamic List

Name:

Variable:

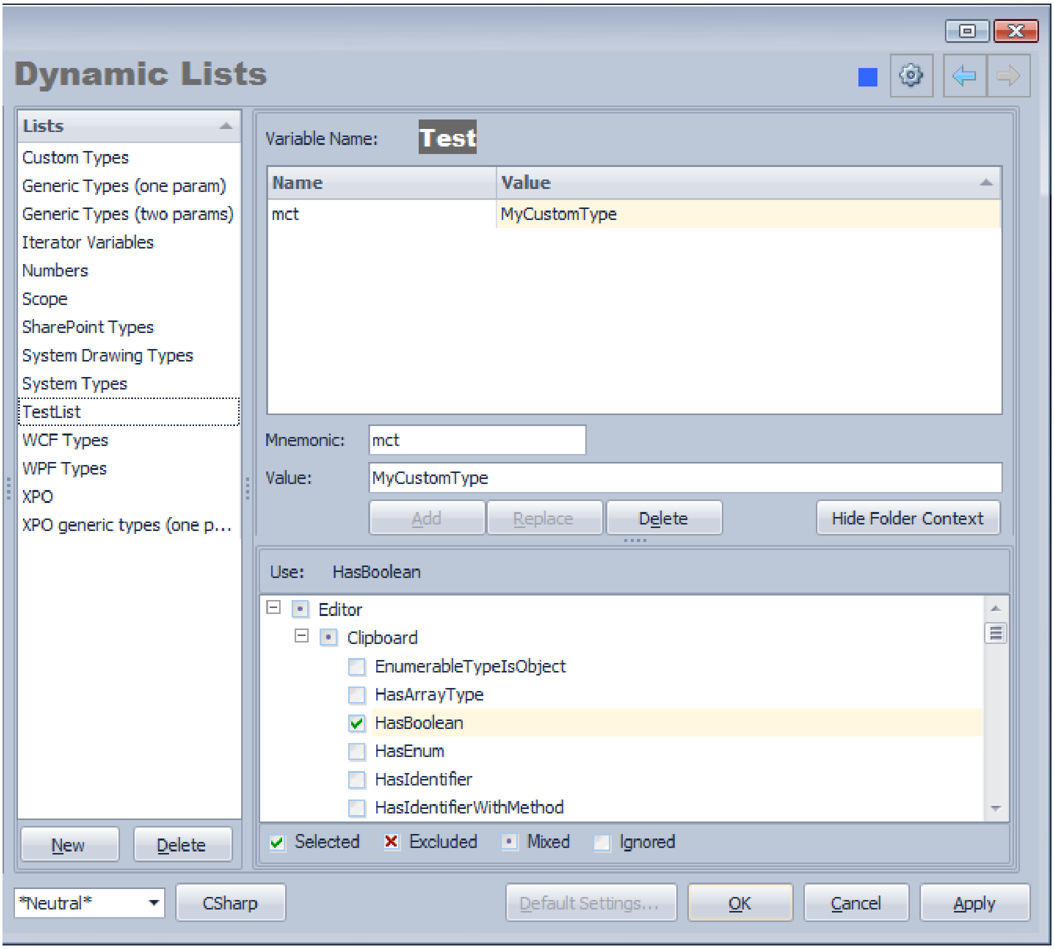
Comment:

□ This list holds .NET types

Dynamic list has a name (caption), variable name (identifier) and a comment (description). The 'This list holds .NET types' options specifies whether or not the new dynamic list contains system .NET types. If checked, the resulting types, once expanded, will be simplified by removing an extra namespace name.

Once a new dynamic list is created, you can add new dynamic list entries to it. An entry has a Key (mnemonic) and Value fields. Specify both values and click the Add button. The Replace button allows you to modify an existing entry. The Delete button removes the selected entry.

A dynamic list can also be context sensitive. Click the Show Folder Context button to specify the context for the current dynamic list. The standard context picker UI will be shown:



Variable Name: Test

Name	Value
mct	MyCustomType

Mnemonic: mct

Value: MyCustomType

Use: HasBoolean

- □ EnumerableTypeIsObject
- □ HasArrayType
- ☑ HasBoolean
- □ HasEnum
- □ HasIdentifier
- □ HasIdentifierWithMethod

Neutral CSharp

Of course, you can modify and extend any existing dynamic lists on this option page as well.

Resolving code template conflicts

Sometimes, you may find unexpected code template expansions when you typing code. This may happen when the template expansion key is bound to the Space Bar. Because this is a key used very often and there are tens of thousands of code templates; a complex contexts system, something may go wrong, especially if you prefer one or two key identifier names which may correspond to existing template abbreviations.

There are several approaches to resolve the unexpected template expansions:

- Change the Expansion Key to Tab instead of Space Bar (use FrictionFree settings scheme) on the Shortcuts option page. Bear in mind that, in this case, you will lose numerous automatic and essential code template expansions like “if”, “while”, “for”, “switch”, etc. These code blocks are automatically completed with parens and braces with the code templates expanded by the Space Bar.

- Enable the What Happened (Feature UI) window which will alert you on each code template expansion. This will allow you to choose what templates must be always available and what templates must never be expanded (disabled).

- Once a code template is expanded, open the Templates option page. The last expanded template will be selected. Here, you can:

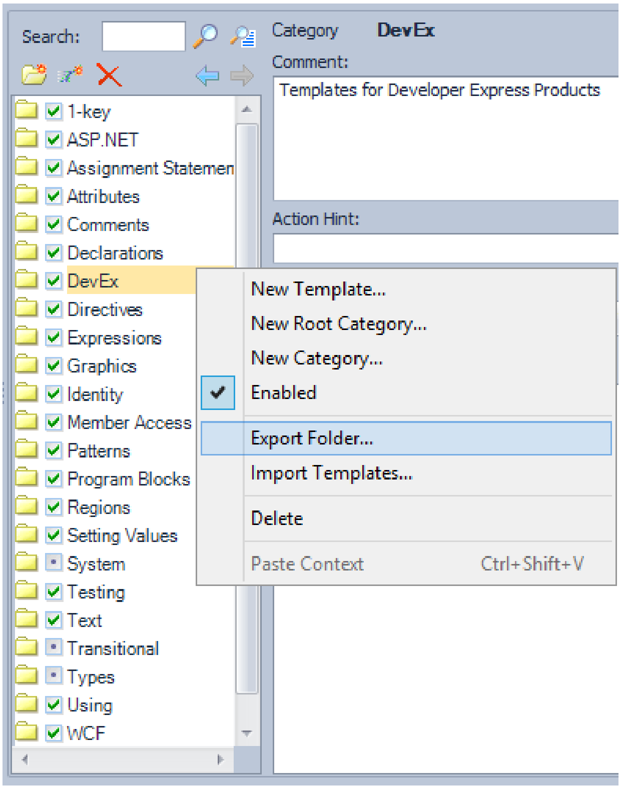
○ Delete the code template if you don’t need it;

○ Change the trigger key to Secondary key (e.g., Shift+Space);

○ Disable the entire template category by unchecking the folder where a template resides.

Importing and Exporting code templates

In case you want to share your custom code templates with others, you can easily export them. As recommended, custom templates should be stored in a separate folder. You can export the entire folder. Simply right-click the folder you wish to export on the **Templates** option page and select the corresponding item:



Category **DevEx**

Comment:

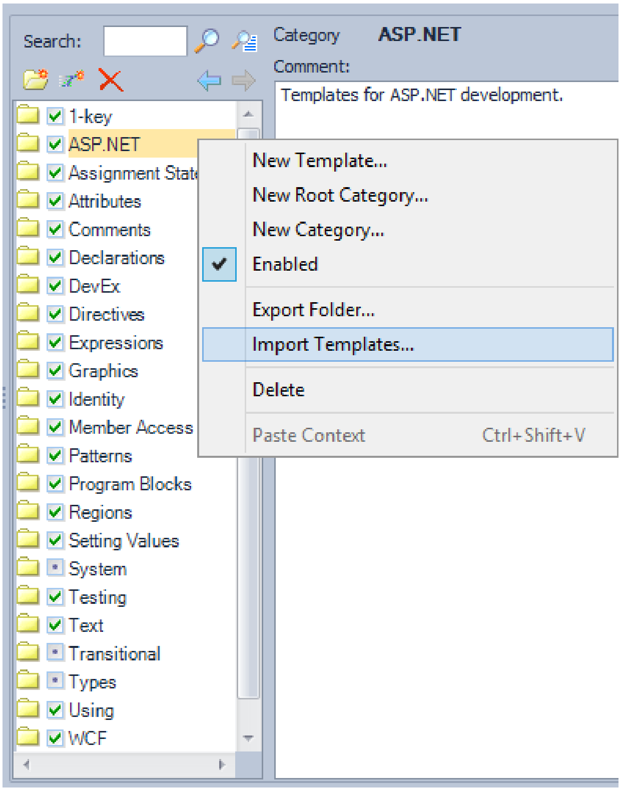
Templates for Developer Express Products

Action Hint:

The standard file saving dialog will appear allowing you to select the target location and the file name. The file name will be automatically composed from the active language and the parent folder names. The extension of the exported file will be “.xml”, e.g.:

CSharp_Declarations_Variables_System.xml

To import templates on a destination computer, open up the Templates option page, right-click any item in the list of templates and choose the “Import Templates…”:

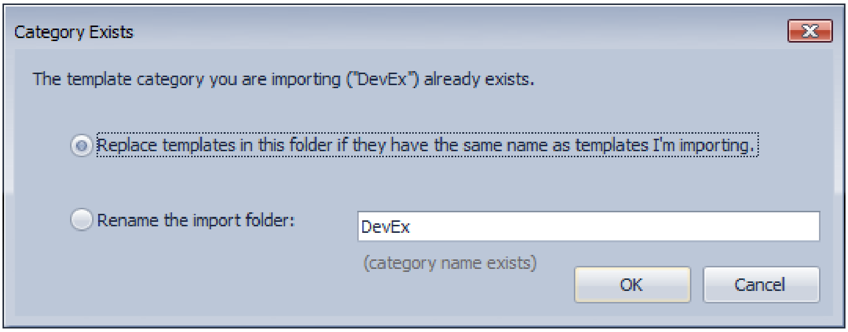


Category ASP.NET

Comment: Templates for ASP.NET development.

Then, choose the previously exported templates file.

When you import the templates from a file, the following dialog may appear:



Category Exists

The template category you are importing ("DevEx") already exists.

● Replace templates in this folder if they have the same name as templates I'm importing.

○ Rename the import folder: DevEx

(category name exists)

This means that the importing templates category with the same name already exists. You have two choices:

- replace the existing templates if they exist in the imported category;
- rename the imported category name.

Choose your option and click OK to import templates. Then, click OK or Apply on the Options Dialog to save the templates.

Standard Templates Library

The **standard templates library** contains more than one thousand static code templates designed to help you realize any coding requirements. Templates are organized by categories. For instance, there are over ٢٥ root categories for CSharp language on the Templates option page, such as:

- ASP.NET
- Attributes
- Comments
- Declarations
- Directives
- Expressions
- Graphics
- Patterns
- Program Blocks
- Regions
- Testing
- Text
- WCF
- etc

Each programming language has its own set of categories and templates. You can import and export the template categories, and add your own categories with custom templates.

With the dynamic lists capability, the number of code templates can total in the tens of thousands. Having learned the template language, it is easy to remember and write code using templates very quickly.

Here are some basic code templates from the library you can start with.

Type Declarations

Type declaration templates make it easy to generate classes, structures, interfaces, enumerations, etc. They are available in the corresponding context (condition), e.g., on an empty line inside a namespace.

Mnemonic	Declares
c	Class
i	Interface
s	Struct
d	Delegate
e	Enumeration

a	Abstract class
x	Exception class

Member Declarations

Member declaration templates allow you generate new methods, properties, fields and events.

Mnemonic	Declares
m	Method
p	Property
v	Variable
ev	Event

Note that the ‘v’ template may declare not only fields, but also local variables and parameters depending on the current context. In combination with the dynamic list of standard types, you can specify the desired type of a member using an additional type specification mnemonic:

Mnemonic	Type
o	object
b	Boolean
c	Char
i	Int32
d	Double
s	string
u	UInt32
…etc	

Loops and blocks

Code templates for quick generation of loops such as ‘for’, ‘foreach’, ‘while’, ‘do’ and other code blocks like ‘if’, ‘switch’, ‘try/catch’ etc.

Mnemonic	Loop or block
f	The ‘for’ loop
fe	The ‘foreach’ loop
w	The ‘while’ loop
dw	The ‘do/while’ loop
if	The ‘if’ block
sw	The ‘switch’ block
tc	The ‘try/catch’ block
tcf	The ‘try/catch/finally’ block
…etc	

Returning values

Code templates for generating a return statement with the desired return value.

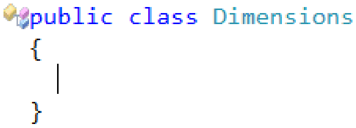
Mnemonic	Return value
rn	null
rt	true
rf	false
r0	zero
r1	one
r-1	-1
rth	this
‘r	single-quoted character
“r	double-quoted string

Note that the ‘r’ template will execute the **Smart Return** feature. This feature allows you to quickly select the return value from the suggested values (e.g., local variables).

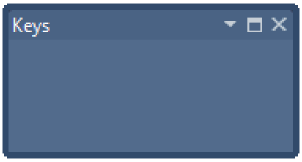
You can see the full collection of code templates for each language on the Templates option page.

Smart Constructor – Generating advanced type constructors in three keystrokes

The **Smart Constructor** CodeRush feature allows you to add constructors to the current class or structure, and pass type members, like fields and properties for initialization through its parameters. Creating a constructor is as simple as just typing three keys: “**cc & Space”:**

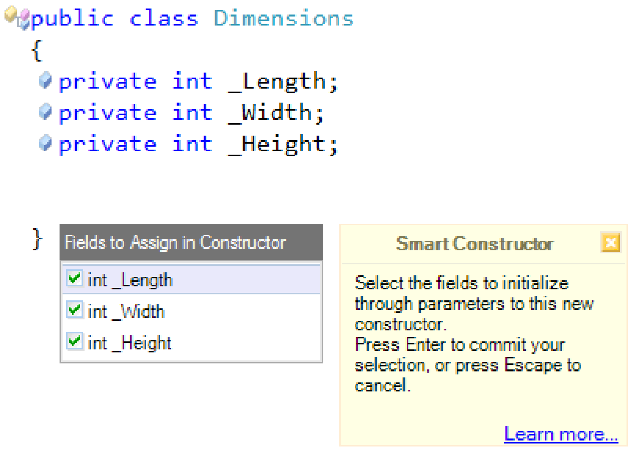


```
public class Dimensions
  {
    |
  }
```



The **Smart Constructor** feature is based on the **CodeRush Templates Engine**, that’s why it is so simple to execute. The “cc” code template is specially dedicated to this feature, and the *Space* key simply expands the template, by default. The template mnemonic (cc) is easy to learn, when you know that it means the “**C**reate **C**onstructor” (cc).

If the active type has members, such as fields and/or auto-implemented properties, the Smart Constructor allows you to choose any of them for initialization inside the constructor, using the “Parameters to the constructor” dialog, for example:



```
public class Dimensions
{
  private int _Length;
  private int _Width;
  private int _Height;


}
```

Fields to Assign in Constructor

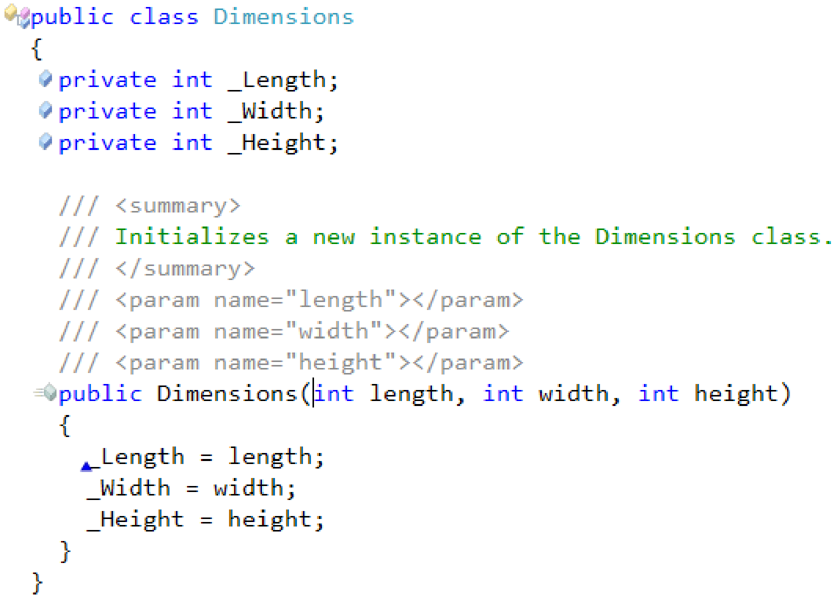
☑ int _Length
☑ int _Width
☑ int _Height

Smart Constructor

Select the fields to initialize through parameters to this new constructor.
Press Enter to commit your selection, or press Escape to cancel.

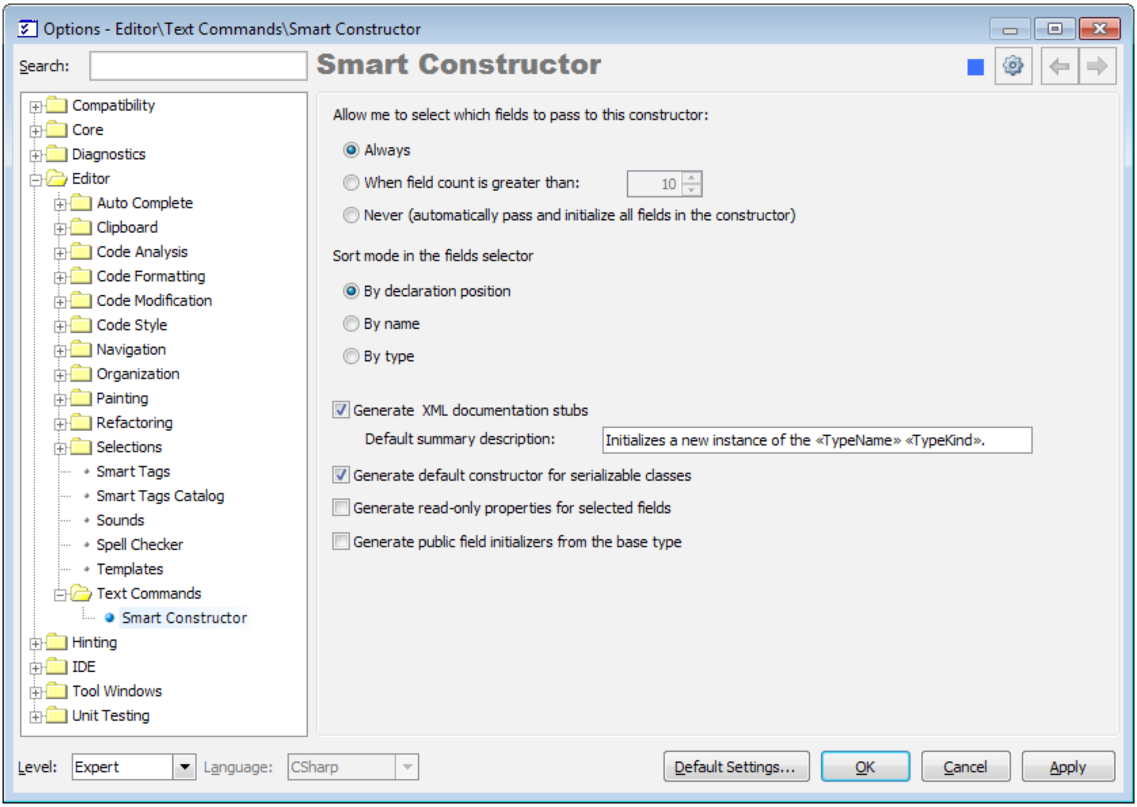
Learn more...

You can use the mouse cursor or the Space key to check or uncheck the object in the list. When you are done with selecting members, press the *Enter* key to commit your changes or the *Esc* key, if you changed you mind on creating a new constructor. Those items that have a check will be initialized through parameters to the constructor when committed, for example:



```
public class Dimensions
{
  private int _Length;
  private int _Width;
  private int _Height;

   /// <summary>
   /// Initializes a new instance of the Dimensions class.
   /// </summary>
   /// <param name="length"></param>
   /// <param name="width"></param>
   /// <param name="height"></param>
  public Dimensions(int length, int width, int height)
   {
     _Length = length;
     _Width = width;
     _Height = height;
   }
}
```

By default, **Smart Constructor** generates an XML doc comment with a standard summary description, which is configurable, of course. All settings for the feature are available on the **Editor** | **Text Commands** | **Smart Constructor** options page in the IDE Tools Options Dialog:



Smart Constructor

Allow me to select which fields to pass to this constructor:

● Always

○ When field count is greater than: 10

○ Never (automatically pass and initialize all fields in the constructor)

Sort mode in the fields selector

● By declaration position

○ By name

○ By type

☑ Generate XML documentation stubs

Default summary description: Initializes a new instance of the «TypeName» «TypeKind».

☑ Generate default constructor for serializable classes

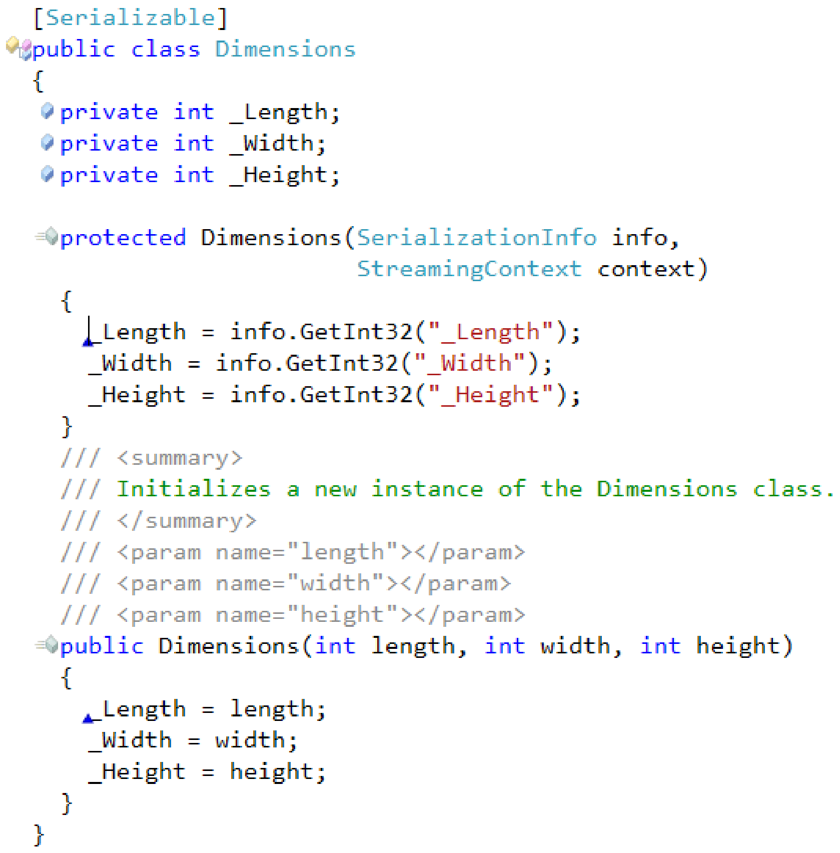
☐ Generate read-only properties for selected fields

☐ Generate public field initializers from the base type

Level: Expert Language: CSharp

Here, on this page, you can specify the following options:

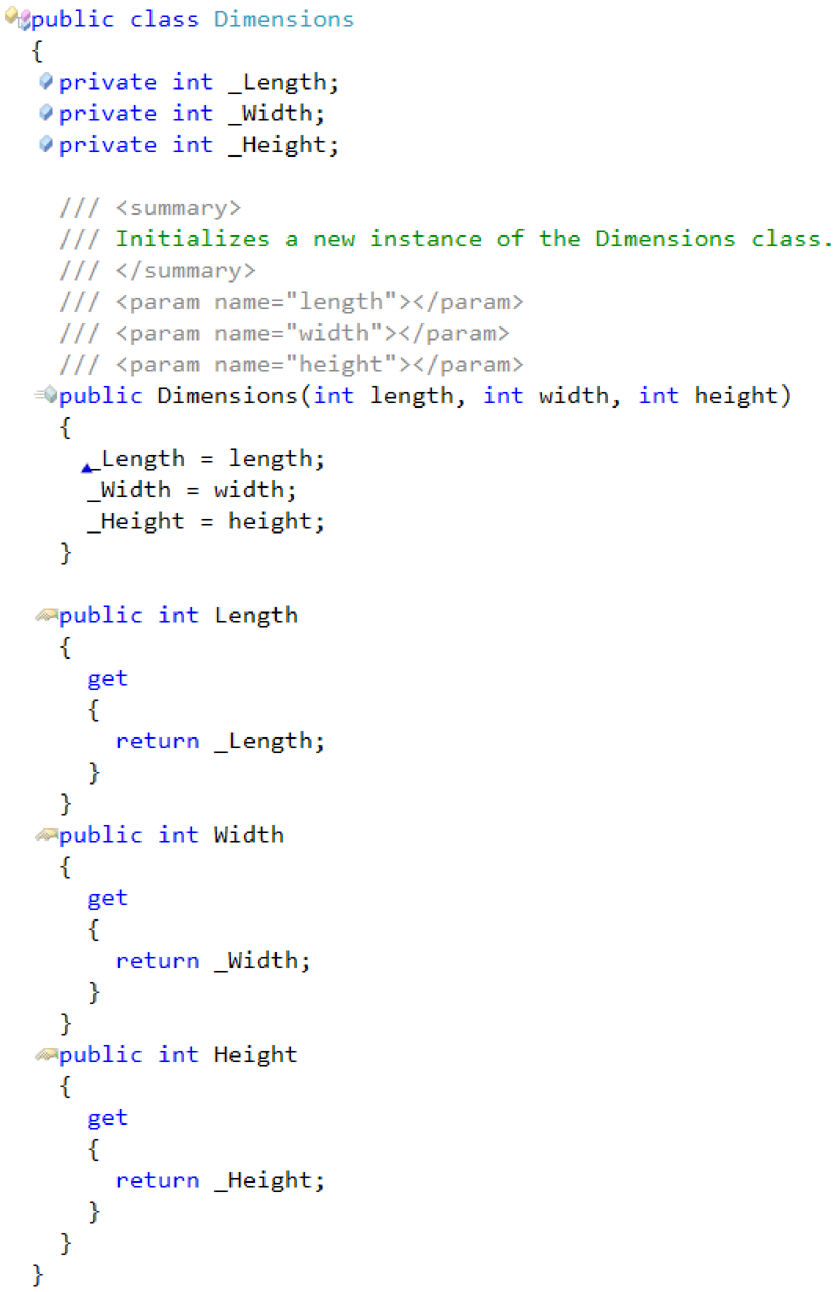
- Whether to show the “Parameters to the constructor” dialog
  - Show always
  - Show only when there are more than [X] members, where you can choose X number
  - Never show
- The sort mode inside the “Parameters to the constructor” dialog:
  - By declaration position will list member in the order as they appear in the code
  - By name
  - By type will sort member by its type (e.g. String, Int32, etc)
- Whether to generate XML doc comment and its default summary description
- Whether to generate default constructor for serializable types, for example:



```
[Serializable]
public class Dimensions
{
  private int _Length;
  private int _Width;
  private int _Height;

  protected Dimensions(SerializationInfo info,
                       StreamingContext context)
  {
    _Length = info.GetInt32("_Length");
    _Width = info.GetInt32("_Width");
    _Height = info.GetInt32("_Height");
  }
  /// <summary>
  /// Initializes a new instance of the Dimensions class.
  /// </summary>
  /// <param name="length"></param>
  /// <param name="width"></param>
  /// <param name="height"></param>
  public Dimensions(int length, int width, int height)
  {
    _Length = length;
    _Width = width;
    _Height = height;
  }
}
```

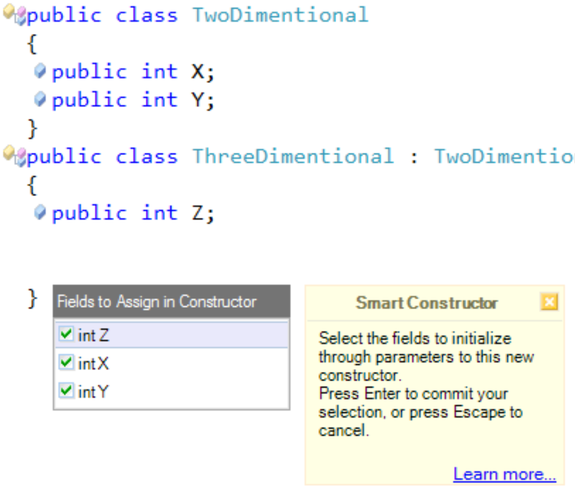
Show Visual Basic code… »

- Whether to generate read-only properties for the selected private fields, for example:



```
public class Dimensions
{
  private int _Length;
  private int _Width;
  private int _Height;

  /// <summary>
  /// Initializes a new instance of the Dimensions class.
  /// </summary>
  /// <param name="length"></param>
  /// <param name="width"></param>
  /// <param name="height"></param>
  public Dimensions(int length, int width, int height)
  {
    _Length = length;
    _Width = width;
    _Height = height;
  }

  public int Length
  {
    get
    {
      return _Length;
    }
  }
  public int Width
  {
    get
    {
      return _Width;
    }
  }
  public int Height
  {
    get
    {
      return _Height;
    }
  }
}
```

- Whether public fields from the base type should be listed in the “Parameters to the constructor”, for example:



```
public class TwoDimentional
  {
    public int X;
    public int Y;
  }
public class ThreeDimentional : TwoDimentional
  {
    public int Z;


  }
```

Fields to Assign in Constructor

- ☑ int Z
- ☑ int X
- ☑ int Y

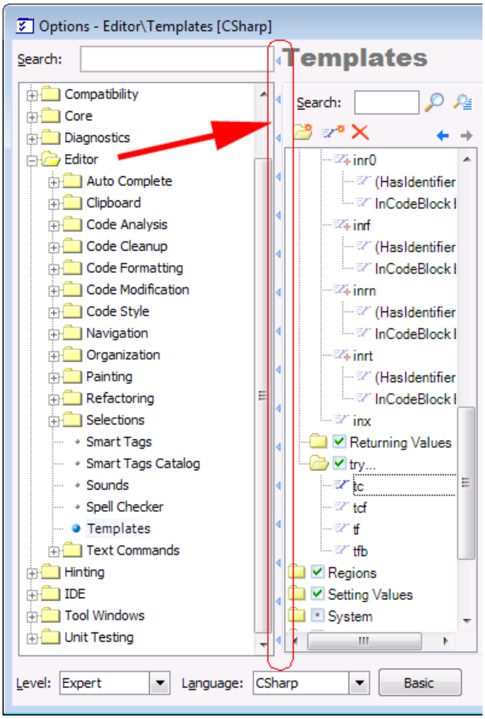
Smart Constructor

Select the fields to initialize through parameters to this new constructor.
Press Enter to commit your selection, or press Escape to cancel.

Learn more...

Creating code templates overview

Creating CodeRush code templates is easy. They are being created on the Templates options page in the Options Dialog. The Templates options page is an excellent way to explore and learn how templates are built. When a template expands, and you want to see how it is built, just bring up the Template options page and the last template expanded will be displayed. I recommend maximizing the Options dialog and collapse the options tree view on the left by clicking those tiny blue rectangles that appear when you hover over the area between the tree view and options page area:

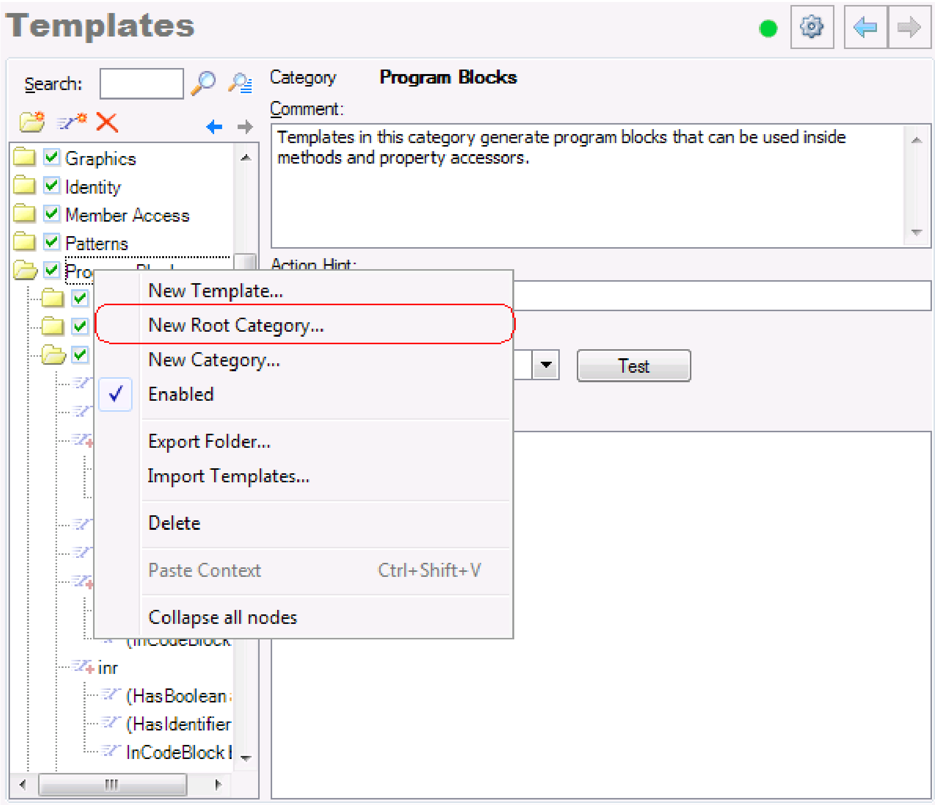


Options - Editor\Templates [CSharp]

Level: Expert   Language: CSharp

Templates are organized into categories. There are dozens of built-in categories in the **CodeRush** templates library, so it is recommended to create your own root folder for newly created templates. If you create your own folder, you can easily share it among other developers or other computers after its export. Categories can also be enabled or disabled by clicking the checkbox to the left of the category name.

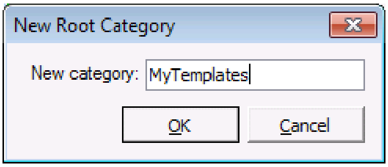
Before you start creating templates, make sure you have chosen the correct language on the bottom of the Options dialog. Remember, that templates are language specific – so it is important to choose the correct language. Then, let's create our root category, where all new templates will be stored. To create a new root category, right-click anywhere on the templates categories list, and choose "New Root Category…":



Category **Program Blocks**

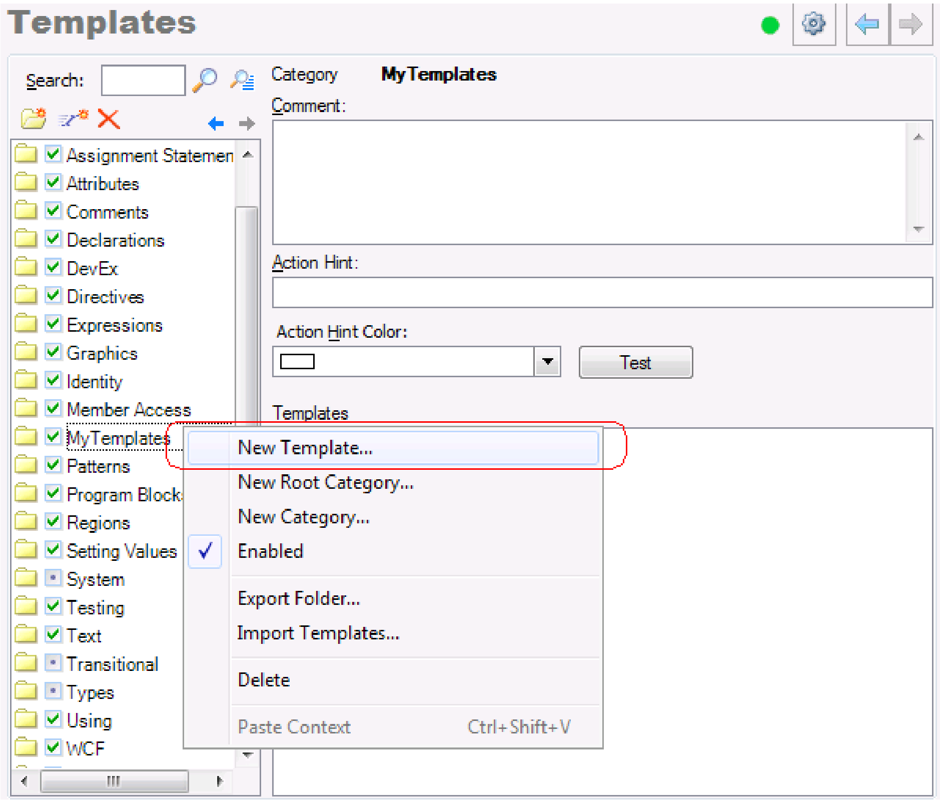
Comment: Templates in this category generate program blocks that can be used inside methods and property accessors.

Type the name of the category and hit Enter or click OK:



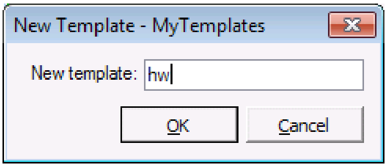
New category: MyTemplates

Once the root category is created, we can add new templates in this category by clicking the corresponding button on the tool bar or right-clicking the category and choosing "New Template":



Category MyTemplates

Type the name of the template, preferably according to the Templates language or something unique:



New template: hw

Now, it’s time to write a template expansion. An expansion can contain the following:

- **Plain text.** A plain text is expanded as is. The text typed in the Expansion area is inserted without modification.

- **String providers.** String providers are used like string functions returning some calculated or hard-coded string value like current date, current file name, new guid, etc.

- **Text commands**. Text commands are like inline procedures that execute some code performing an action like moving the caret, dropping a marker, creating a new file, adding an assembly reference, opening a dialog – everything you want. The standard text commands library contain a lot of useful commands which can be expanded by creating your own text commands via the special DXCore component.

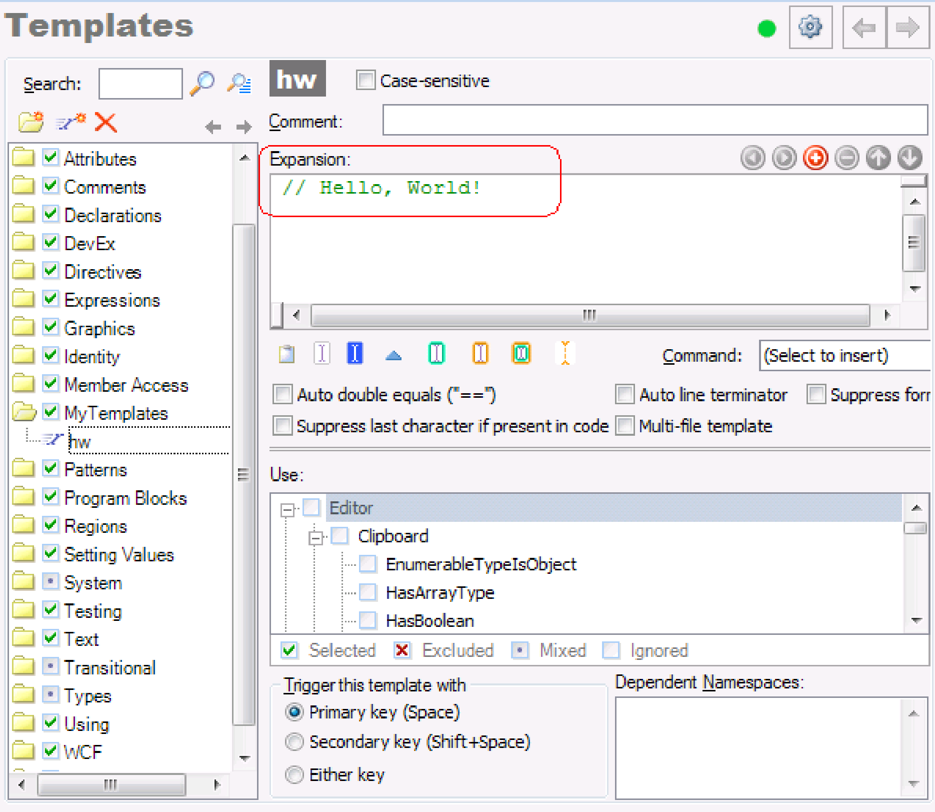
To learn more about creating your own templates, refer to the following articles:

- Creating simple templates step by step, where the simple templates creation process is described. Simple templates have an easy text expansion containing simplest string providers and text commands.

- Creating advanced code templates step by step, where complex templates are created. Such templates may have alternate expansions and use advanced text commands like ForEach, Get, Set, etc.

Creating simple templates step by step

Once you know recommendations for creating new templates and have the Templates options page open, you can start writing a template expansion starting from plain text. Any plain text typed or pasted into the Expansion area will be expanded as is, for example:



Templates

Search:

hw

□ Case-sensitive

Comment:

Expansion:

```
// Hello, World!
```

Command: (Select to insert)

□ Auto double equals ("==")
□ Auto line terminator
□ Suppress forr
□ Suppress last character if present in code
□ Multi-file template

Use:

- □ Editor
  - □ Clipboard
    - □ EnumerableTypeIsObject
    - □ HasArrayType
    - □ HasBoolean

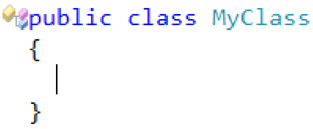
☑ Selected ☒ Excluded ▣ Mixed □ Ignored

Trigger this template with

- ● Primary key (Space)
- ○ Secondary key (Shift+Space)
- ○ Either key

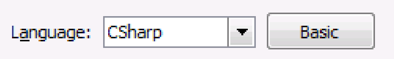
Dependent Namespaces:

the template name will be replaced with the expansion text:



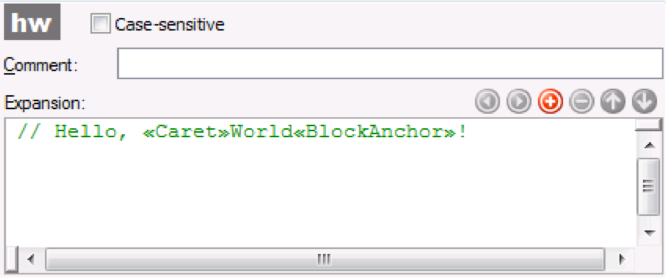
```
public class MyClass
{
  |
}
```

The text in the Expansion area is highlighted with the syntax of the chosen language at the bottom of the page, e.g.:



Language: CSharp

After a template expands, the editor caret is positioned at the end of the expansion, by default. We can bring more intelligence into our template expansion by inserting simple text commands for positioning an editor caret and selecting the resulting text, for example:



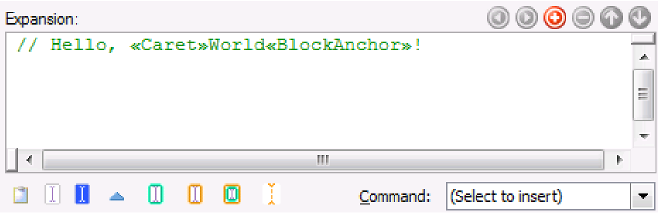
□ Case-sensitive

Comment:

Expansion:

```
// Hello, «Caret»World«BlockAnchor»!
```

The text commands are wrapped into special tags (« and »), so the CodeRush Templates engine can find commands in the template expansion, parse the name of a text command and then execute one, to get its results. The most common text commands are placed on the commands tool bar:

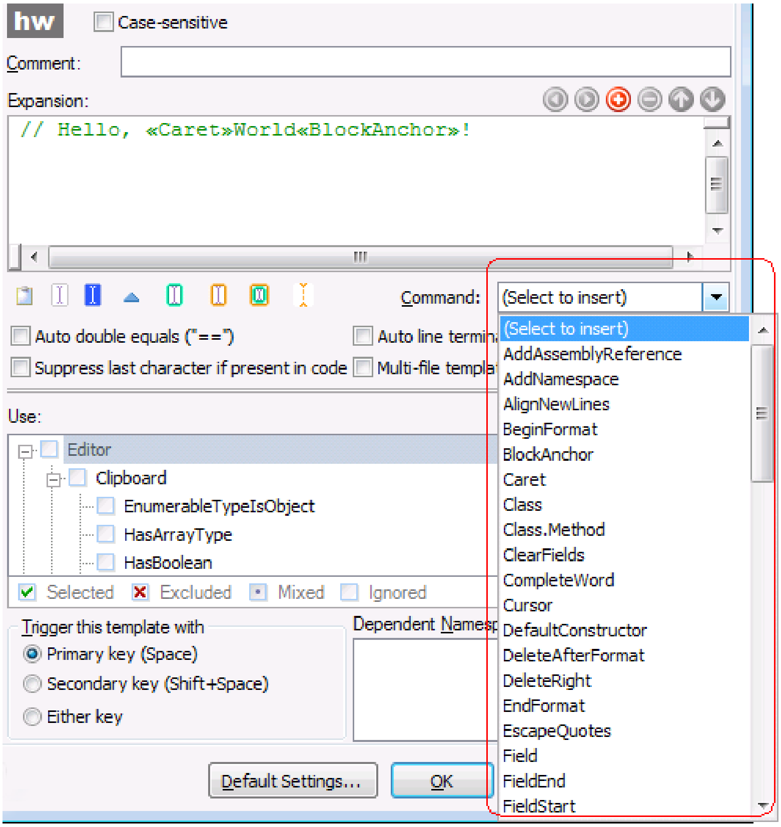


Expansion:

```
// Hello, «Caret»World«BlockAnchor»!
```

Command: (Select to insert)

All other commands can be inserted via the Command combo box:



□ Case-sensitive

Comment:

Expansion:

```
// Hello, «Caret»World«BlockAnchor»!
```

Command: (Select to insert)

□ Auto double equals ("==")
□ Auto line termin
□ Suppress last character if present in code
□ Multi-file templa

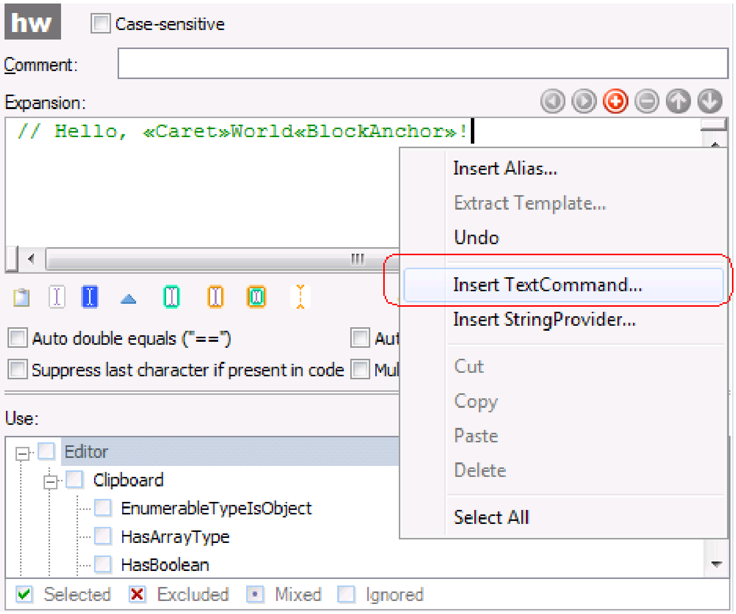
Use:

☑ Selected ☒ Excluded ▣ Mixed □ Ignored

Trigger this template with
● Primary key (Space)
○ Secondary key (Shift+Space)
○ Either key

Dependent Namesp

or by right-clicking the Expansion area and choosing the “Insert TextCommand…” menu item:



hw □ Case-sensitive

Comment:

Expansion:

```
// Hello, «Caret»World«BlockAnchor»!
```

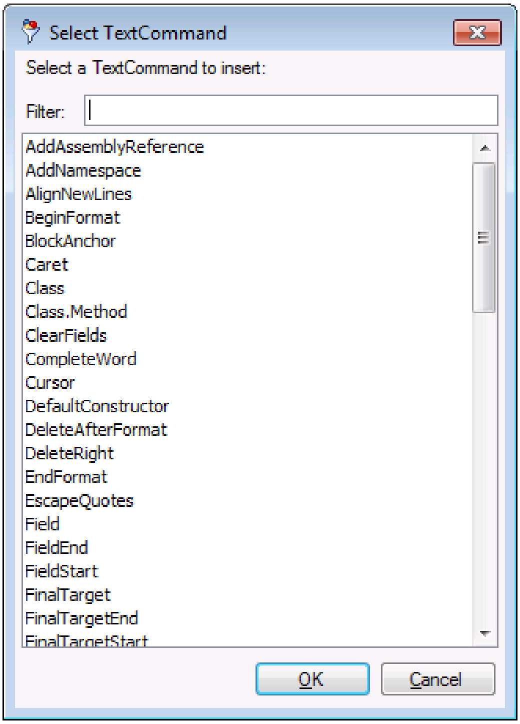
□ Auto double equals ("==")

□ Suppress last character if present in code

Use:

- □ Editor
  - □ Clipboard
    - □ EnumerableTypeIsObject
    - □ HasArrayType
    - □ HasBoolean

The Select TextCommand dialog will appear, where you can find the required text command by typing its name in the Filter box or by choosing it from the list:



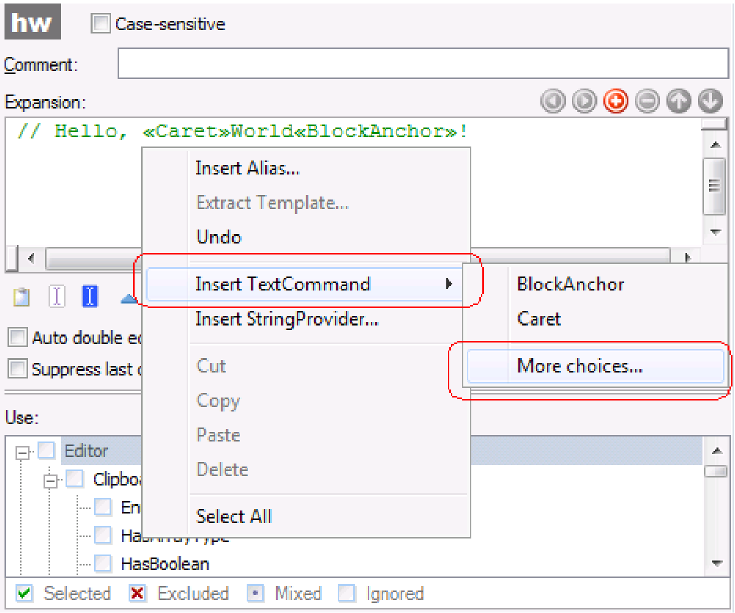
Select TextCommand

Select a TextCommand to insert:

Filter:

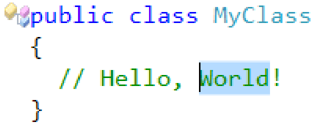
AddAssemblyReference
AddNamespace
AlignNewLines
BeginFormat
BlockAnchor
Caret
Class
Class.Method
ClearFields
CompleteWord
Cursor
DefaultConstructor
DeleteAfterFormat
DeleteRight
EndFormat
EscapeQuotes
Field
FieldEnd
FieldStart
FinalTarget
FinalTargetEnd
FinalTargetStart

The menu item will store previously chosen text commands, if any:



```
// Hello, «Caret»World«BlockAnchor»!
```

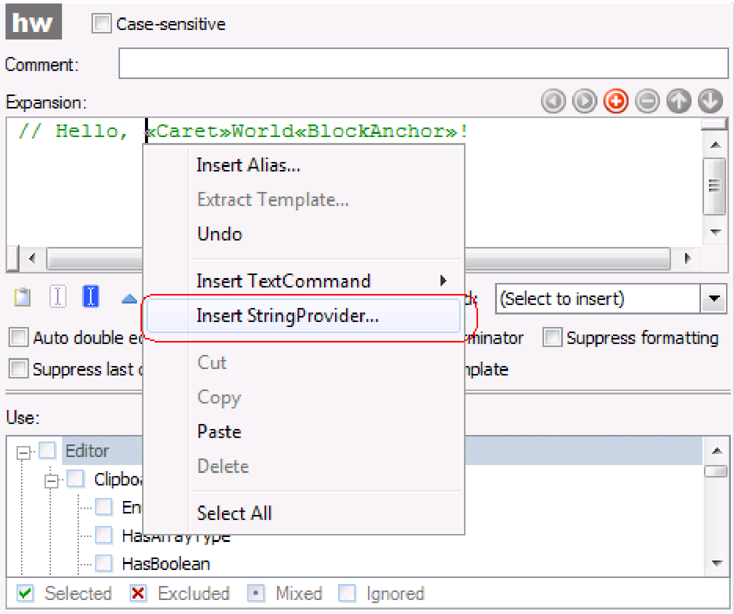
The Caret and Anchor text commands are usually used as a pair. The Caret is used to position the caret in the resulting template expansion. The Anchor text command is used to mark the end of the selection that starts from the Caret position. If there's no Anchor text command used in the expansion, there will be no text selection. However, if there's no Caret text command used, the current editor caret position is used. If there are several Caret or Anchor text commands, the only one last text command is valid. The resulting expansion looks as follows:



```
public class MyClass
{
   // Hello, World!
}
```

If you need to create templates with more than one entry point, you can drop markers in your template. Markers are useful when a template requires more than one point where you are going to modify or add code after an expansion is completed.

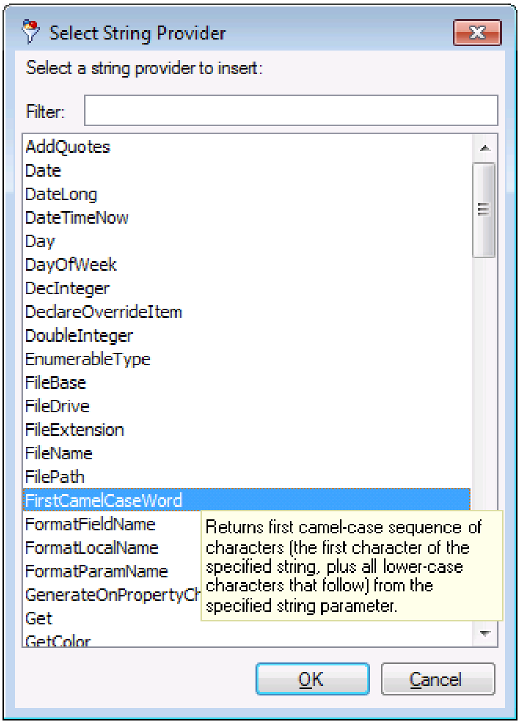
Now, let's add more intelligence by inserting several string providers. String providers simply return string values. You can insert them by right-clicking the Expansion area and choosing the "Insert StringProvider…" menu item:



Expansion:

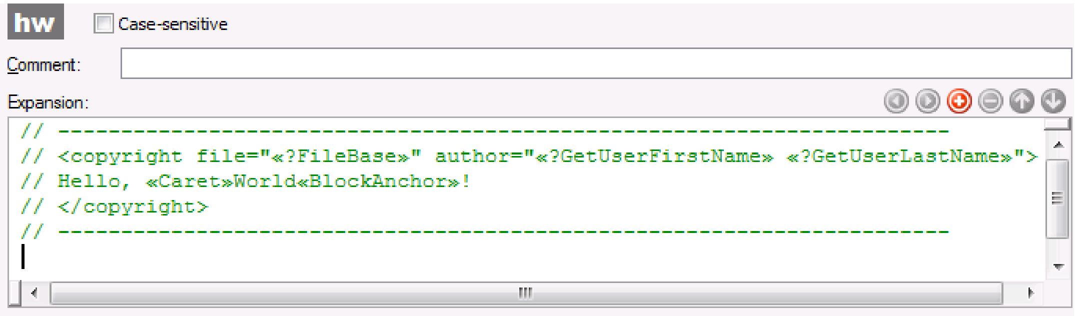
```
// Hello, «Caret»World«BlockAnchor»!
```

To learn more about a specific string provider, hover over its name in the Select String Provider dialog:



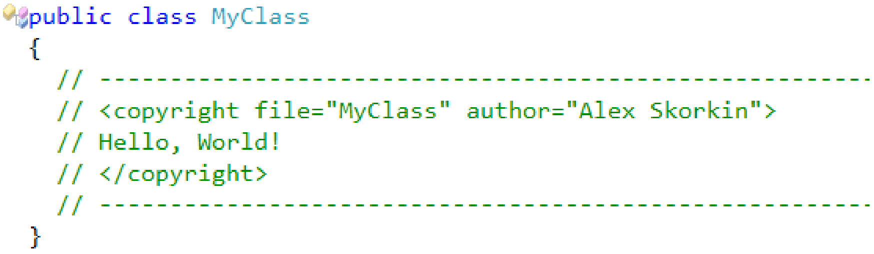
Returns first camel-case sequence of characters (the first character of the specified string, plus all lower-case characters that follow) from the specified string parameter.

Let's use the string providers like FileName, GetUserFirstName and GetUserLastName and modify the template expansion as follows:



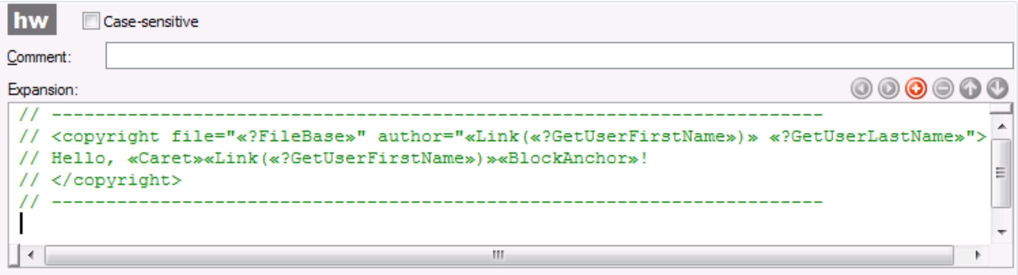
```
// ----------------------------------------------------------------------
// <copyright file="«?FileBase»" author="«?GetUserFirstName» «?GetUserLastName»">
// Hello, «Caret»World«BlockAnchor»!
// </copyright>
// ----------------------------------------------------------------------
```

Once this template saved and expanded, the following result is produced:



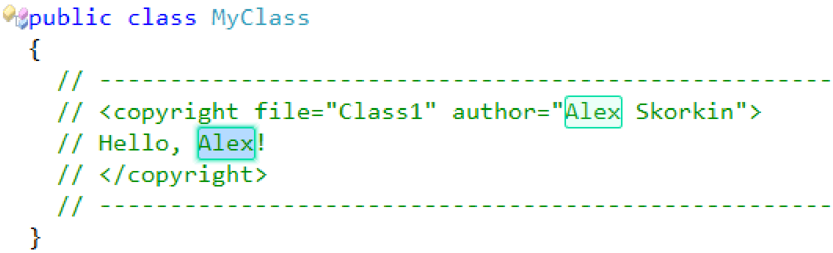
```
public class MyClass
{
  // ------------------------------------------------------
  // <copyright file="MyClass" author="Alex Skorkin">
  // Hello, World!
  // </copyright>
  // ------------------------------------------------------
}
```

Let's change the expansion further by adding the linked identifiers text commands which allow you to change a piece of text in several locations at once. For example, let's add linked identifiers for the first name of a user. The Link text command is used to add linked identifiers:



```
// ----------------------------------------------------------------------
// <copyright file="«?FileBase»" author="«Link(«?GetUserFirstName»)» «?GetUserLastName»">
// Hello, «Caret»«Link(«?GetUserFirstName»)»«BlockAnchor»!
// </copyright>
// ----------------------------------------------------------------------
```

Result:



```
public class MyClass
{
  // ---------------------------------------------------
  // <copyright file="Class1" author="Alex Skorkin">
  // Hello, Alex!
  // </copyright>
  // ---------------------------------------------------
}
```

This is how simple templates are created. The more complex templates use an advanced text command which may iterate members of the current class, add dependent namespaces or paste text into multiple locations as well as multiple files. See how complex templates are created in the corresponding post (coming soon).

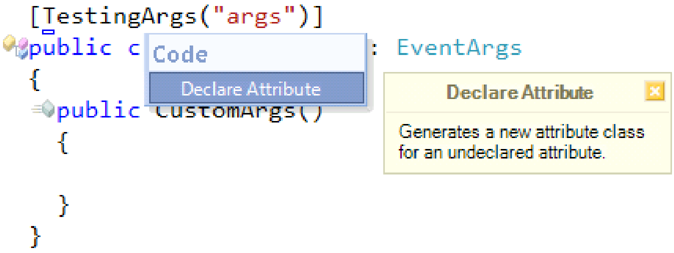
Chapter 4. Consume-first code declaration providers

Declaring type declarations

Declare Attribute CodeRush code provider

Attributes provide a powerful method of associating declarative information by decorating elements of the code, such as types, methods, properties, parameters and assemblies. Once an attribute is associated with a program entity, it can be queried at run time and used in various cases, for example, associating a help document with program entities (via the Help attribute), or marking an item as out of date (via the Obsolete attribute).

As the name says, the **Declare Attribute** code provider generates a new attribute class for an undeclared attribute reference:

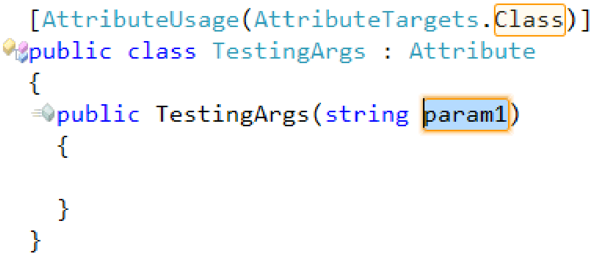


```
[TestingArgs("args")]
public c                             : EventArgs
{
  public CustomArgs()
  {

  }
}
```

Declare Attribute

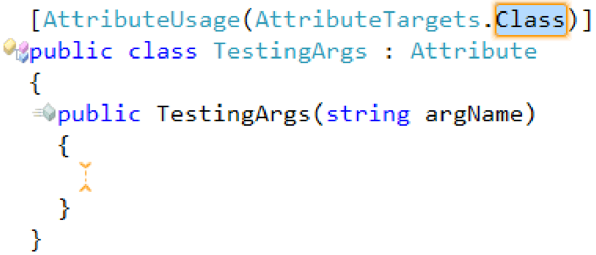
Generates a new attribute class for an undeclared attribute.

It adds the appropriate AttributeUsage attribute for the newly declared attribute with the corresponding attribute target, which can be easily modified using the Text Fields CodeRush feature:



```
[AttributeUsage(AttributeTargets.Class)]
public class TestingArgs : Attribute
{
  public TestingArgs(string param1)
  {

  }
}
```

Simply press Enter when you are done with the name of the attribute and you will be navigated to the attribute target for further modification:



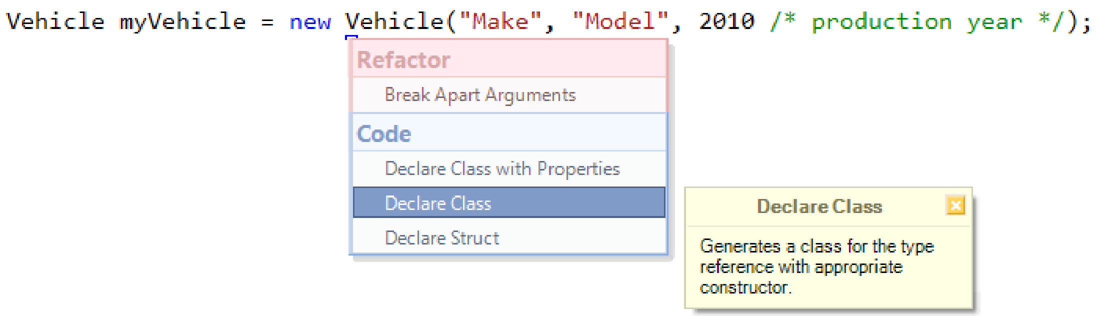
```
[AttributeUsage(AttributeTargets.Class)]
public class TestingArgs : Attribute
{
  public TestingArgs(string argName)
  {

  }
}
```

If the attribute target is valid, pressing the Enter key for the second time will move you inside the constructor of the class.

Declare Class

Declare Class code provider generates a class for the current type reference to a non-existent type. If the type reference on the editor caret creates a new instance of a non-existent type that takes some arguments, the appropriate constructor is generated for the new class.

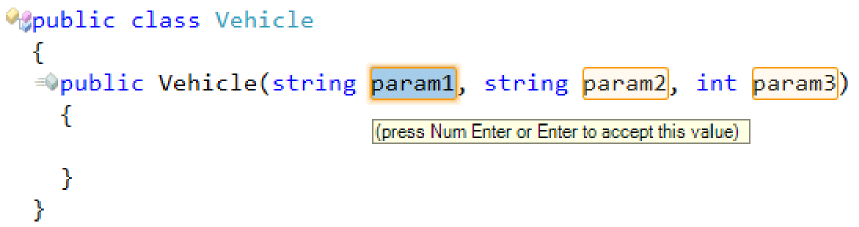
CSharp:



```
Vehicle myVehicle = new Vehicle("Make", "Model", 2010 /* production year */);
```

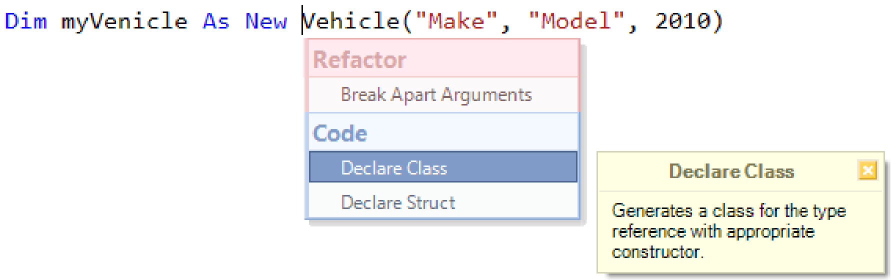
Refactor
Break Apart Arguments
Code
Declare Class with Properties
Declare Class
Declare Struct
Declare Class
Generates a class for the type reference with appropriate constructor.

Result:



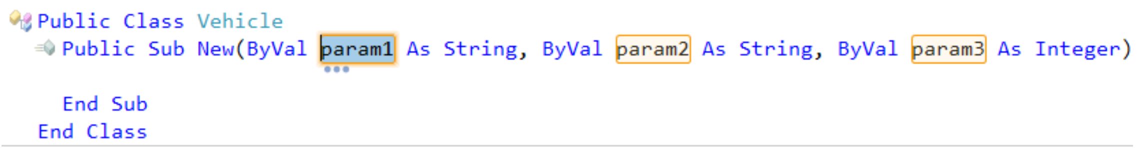
```
public class Vehicle
 {
   public Vehicle(string param1, string param2, int param3)
   {

   }
 }
```

Visual Basic:



```
Dim myVenicle As New Vehicle("Make", "Model", 2010)
```

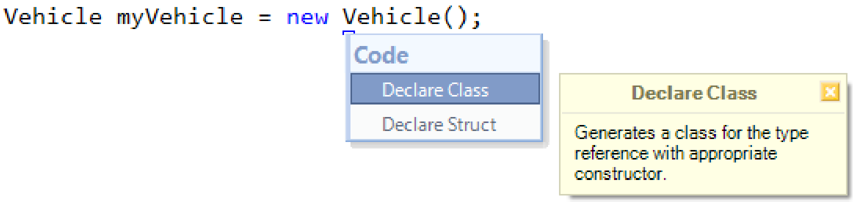
Result:



```
Public Class Vehicle
  Public Sub New(ByVal param1 As String, ByVal param2 As String, ByVal param3 As Integer)

  End Sub
End Class
```

Otherwise, if there are no arguments passed, the default public parameterless constructor is declared:

CSharp:

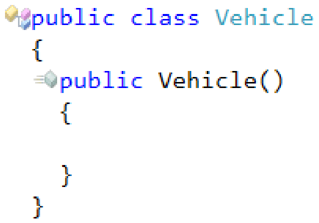


```
Vehicle myVehicle = new Vehicle();
```

Declare Class

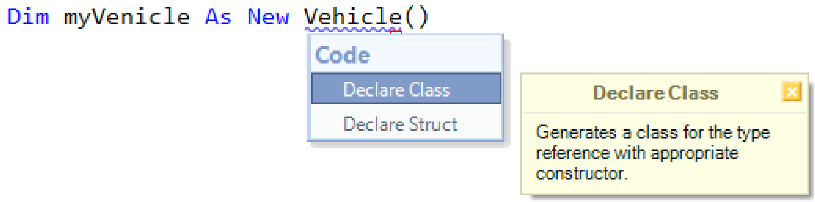
Generates a class for the type reference with appropriate constructor.

Result:



```
public class Vehicle
  {
    public Vehicle()
    {

    }
  }
```

Visual Basic:

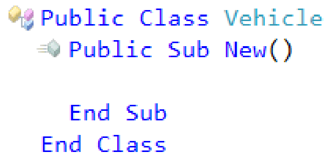


```
Dim myVenicle As New Vehicle()
```

Declare Class

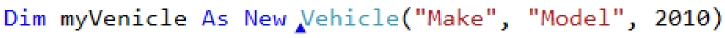
Generates a class for the type reference with appropriate constructor.

Result:



```
Public Class Vehicle
  Public Sub New()

    End Sub
  End Class
```

A marker will be dropped after **Declare Class** is performed at the source text caret position to easily get back and continue code editing:



```
Dim myVenicle As New Vehicle("Make", "Model", 2010)
```

Also, text fields will be created for the constructor parameters, if any, to easily rename them, according to your preference (see the sample above).

A similar code provider is Declare Struct.

Specific code providers for declaring new classes

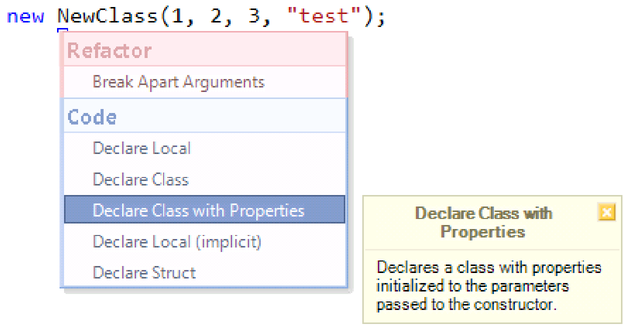
CodeRush provides code providers for declaring classes, such as:

- Declare Class to declare a simple class without members,
- Declare Struct to declare a structure without members,
- Declare Attribute to declare an attribute class,
- Create Descendant and Create Descendant (with virtual overrides) to declare descendant classes,
- Create Ancestor to declare base classes,
- Create Implementer to declare a class that implements an interface.

There are two additional specific code providers that declare classes:

- **Declare Class with Properties**
- **Declare EventArgs Descendant**

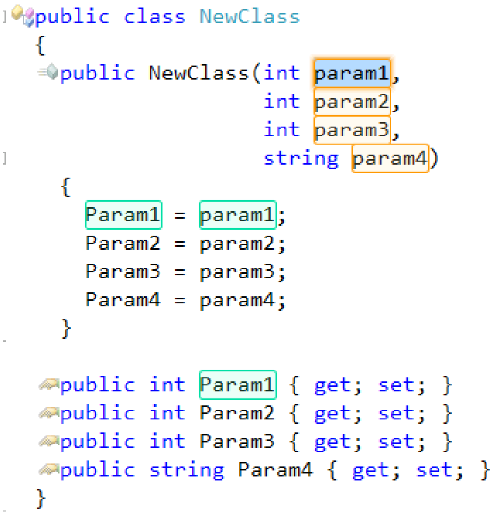
The **Declare Class with Properties** declares a class with properties (auto-implemented) initialized to the parameters passes to the constructor. That is, if you type a constructor call to an undeclared class that takes several parameters, for example:



```
new NewClass(1, 2, 3, "test");
```

Declares a class with properties initialized to the parameters passed to the constructor.

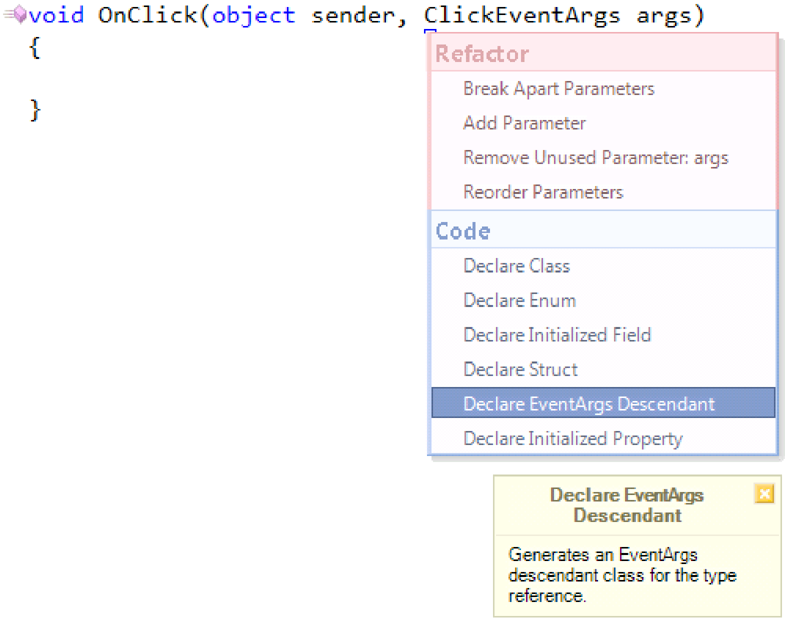
the code provider will produce the following class after it is applied:



```
public class NewClass
{
  public NewClass(int param1,
                  int param2,
                  int param3,
                  string param4)
  {
    Param1 = param1;
    Param2 = param2;
    Param3 = param3;
    Param4 = param4;
  }

  public int Param1 { get; set; }
  public int Param2 { get; set; }
  public int Param3 { get; set; }
  public string Param4 { get; set; }
}
```

All parameters are linked together for fast renaming. Properties are renamed in sync with parameters.

The **Declare EventArgs Descendant** is a dedicated code provider to declare classes that descend from the System. EventArgs class. It is available on the type reference to an undeclared class, whose name ends with the “EventArgs”:

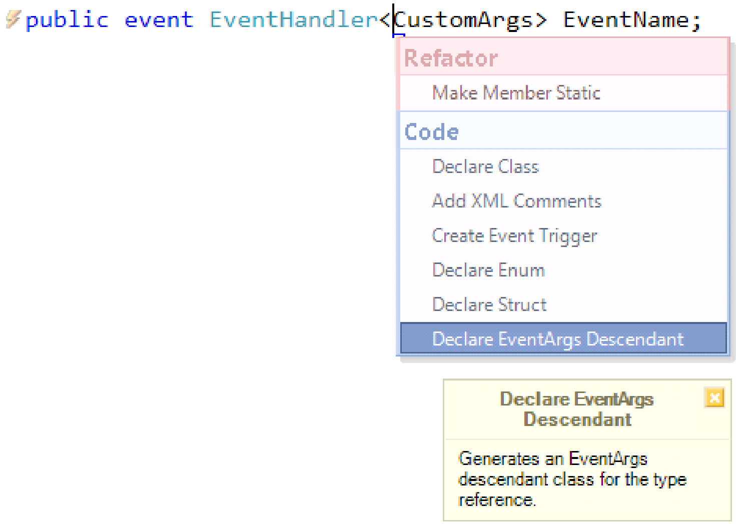


```
void OnClick(object sender, ClickEventArgs args)
{

}
```

Refactor
Break Apart Parameters
Add Parameter
Remove Unused Parameter: args
Reorder Parameters
Code
Declare Class
Declare Enum
Declare Initialized Field
Declare Struct
Declare EventArgs Descendant
Declare Initialized Property
Declare EventArgs Descendant

Generates an EventArgs descendant class for the type reference.

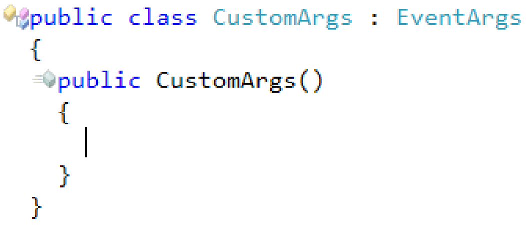
or on the type argument that must be an EventArgs descendant, for example:



```
public event EventHandler<CustomArgs> EventName;
```

Generates an EventArgs descendant class for the type reference.

The following result will be produced in the last case:

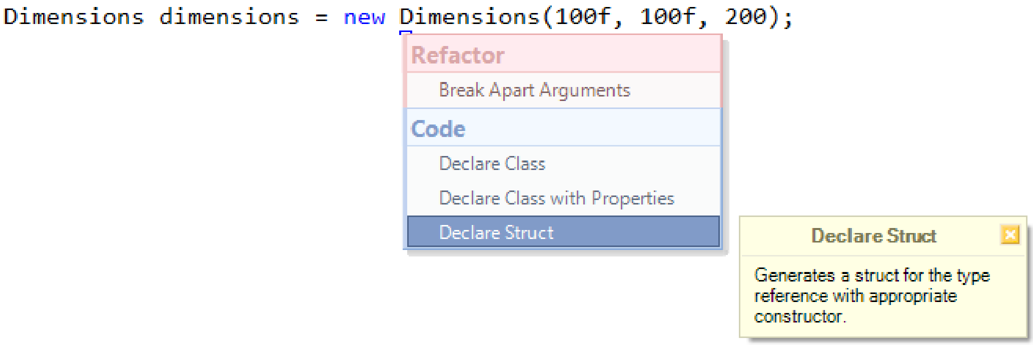


```
public class CustomArgs : EventArgs
{
  public CustomArgs()
  {
    |
  }
}
```

Declare Struct

Declare Struct code provider generates a structure for the current type reference to a non-existent type. If the type reference on the editor caret creates a new instance of a non-existent type that takes some arguments, the appropriate constructor is generated for the new structure:

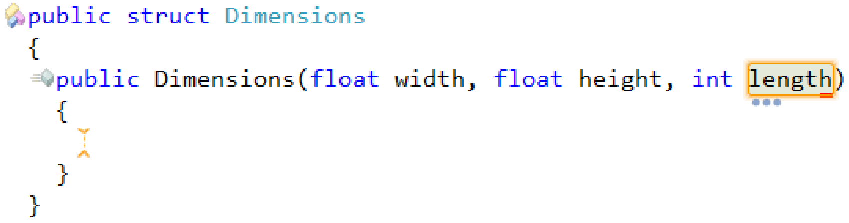
CSharp:



```
Dimensions dimensions = new Dimensions(100f, 100f, 200);
```

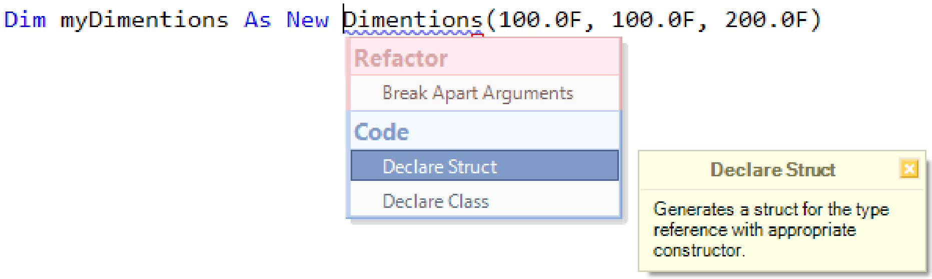
Refactor
Break Apart Arguments
Code
Declare Class
Declare Class with Properties
Declare Struct
Declare Struct
Generates a struct for the type reference with appropriate constructor.

Result:



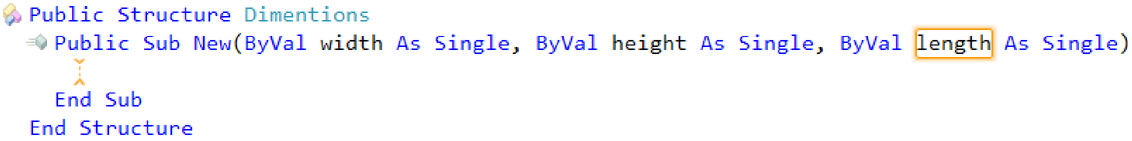
```
public struct Dimensions
{
  public Dimensions(float width, float height, int length)
  {

  }
}
```

Visual Basic:



```
Dim myDimentions As New Dimentions(100.0F, 100.0F, 200.0F)
```

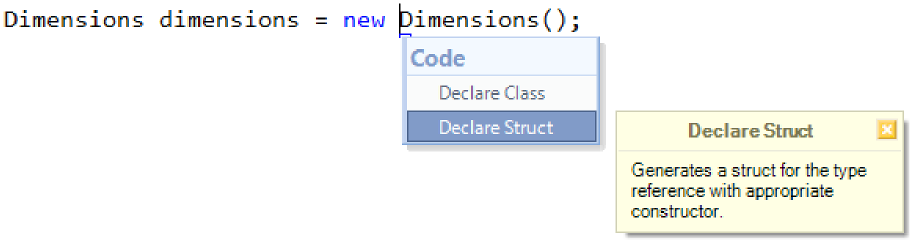
Result:



```
Public Structure Dimentions
    Public Sub New(ByVal width As Single, ByVal height As Single, ByVal length As Single)

    End Sub
End Structure
```

Otherwise, if there are no arguments passed, the default public parameterless constructor is declared:

CSharp:

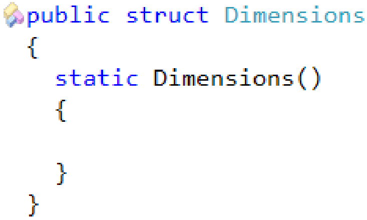


```
Dimensions dimensions = new Dimensions();
```

Declare Struct

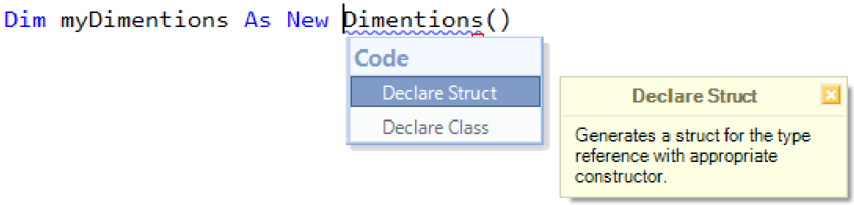
Generates a struct for the type reference with appropriate constructor.

Result:



```
public struct Dimensions
{
    static Dimensions()
    {

    }
}
```

Visual Basic:

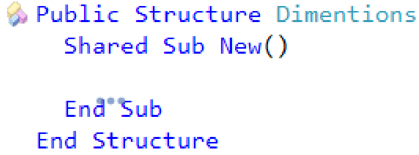


```
Dim myDimentions As New Dimentions()
```

Declare Struct

Generates a struct for the type reference with appropriate constructor.

Result:



```
Public Structure Dimentions
    Shared Sub New()

    End Sub
End Structure
```

A marker will be dropped after **Declare Struct** is performed at the source text caret position to easily get back and continue code editing:

```
Dimensions dimensions = new Dimensions(100f, 100f, 200);
```

Also, text fields will be created for the constructor parameters, if any, to easily rename them, according to your preference (see the sample above).

A similar code provider is Declare Class.

Declare Interface

The **Declare Interface** code provider generates a new definition of an interface and adds interface referenced members to it, if any. The declaring provider is available on an undeclared type reference, that starts with an upeper-case letter I, e.g. *ILogger*.

For example, declaring an Interface from the *ILogger* reference in this code:

```
class Log
{
  private static ILogger logger = new Logger();

  public static void SendMessage(string msg)
  {
    logger.SendMessage(msg);
  }
  public static string GetMessage(int index)
  {
    return logger.GetMessage(index);
  }
}
```

Will result in the following Interface declaration:

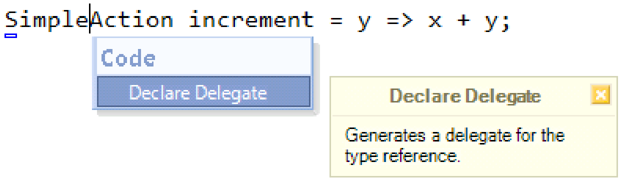
```
public interface ILogger
{
  void GetMessage(int index);
  void SendMessage(string msg);
}
```

The interface declaration is placed above the current type in the same file, and the marker is dropped at the source starting position over the interface reference.

Declare Delegate

A delegate is a special kind of object that holds a reference to a method. Once a delegate is assigned a method, it behaves exactly like that method. It can have parameters and a return value. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object.

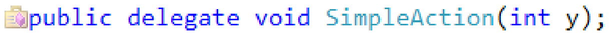
CodeRush provides the **Declare Delegate** code provider that is available on references that refer a delegate which is not declared yet, for example:



```
SimpleAction increment = y => x + y;
```

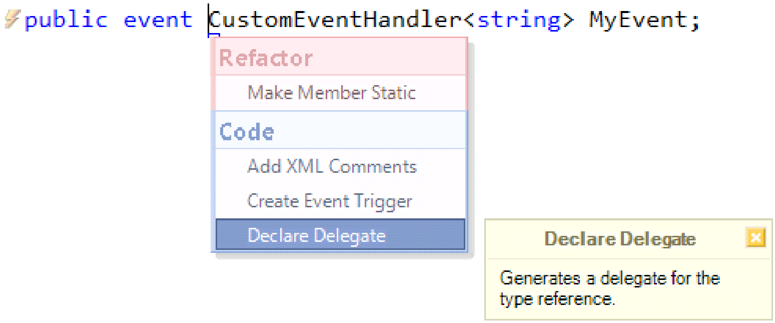
Generates a delegate for the type reference.

the following result will be produced after the code provider is applied:



```
public delegate void SimpleAction(int y);
```

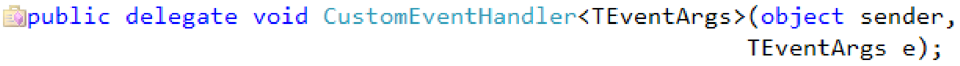
Another example:



```
public event CustomEventHandler<string> MyEvent;
```

Generates a delegate for the type reference.

the following delegate will be generated:



```
public delegate void CustomEventHandler<TEventArgs>(object sender,
                                                    TEventArgs e);
```

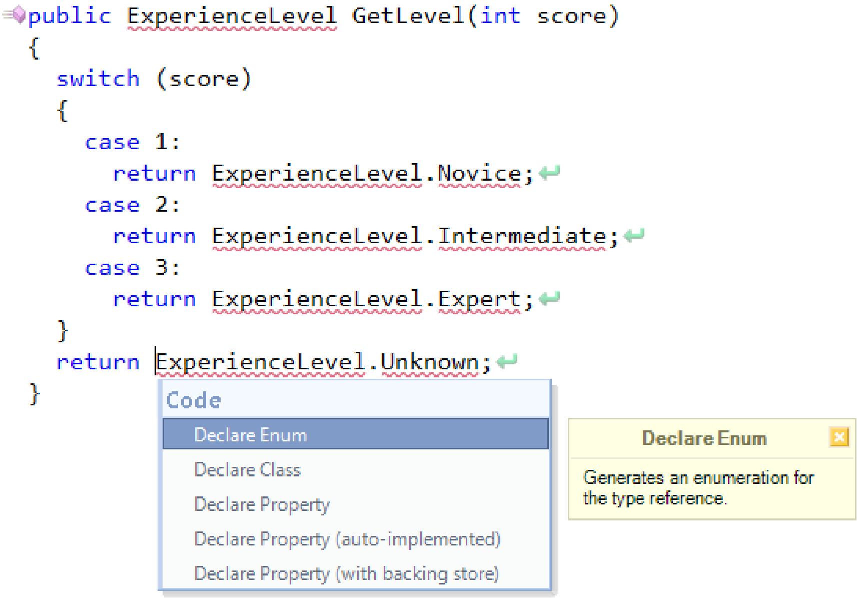
Declaring enumeration types and its elements

An enumeration type is a special set of related constants, each with an integer value. Enumerations are useful for defining states and sequences, particularly when there is a natural progression through those states. Each constant in the enumeration list can be compared and formatted using either its name or value. For example, assume that you have to define a variable whose value will represent a day of the week. There are only seven meaningful values which that variable will ever store. To define those values, you can use an enumeration type.

CodeRush helps you to declare enumeration types and its values easily and quickly. Type the code as if you already had the resulting enumeration type declared, e.g.:

```
public ExperienceLevel  GetLevel(int score)
{
  switch (score)
  {
    case 1:
      return ExperienceLevel.Novice;
    case 2:
      return ExperienceLevel.Intermediate;
    case 3:
      return ExperienceLevel.Expert;
  }
  return ExperienceLevel.Unknown;
}
```

However, the *ExperienceLevel* enumeration is not yet actually declared. Move the editor caret to the *ExperienceLevel* reference and execute the **Declare Enum** code provider:



```
public ExperienceLevel GetLevel(int score)
{
  switch (score)
  {
    case 1:
      return ExperienceLevel.Novice;
    case 2:
      return ExperienceLevel.Intermediate;
    case 3:
      return ExperienceLevel.Expert;
  }
  return ExperienceLevel.Unknown;
}
```

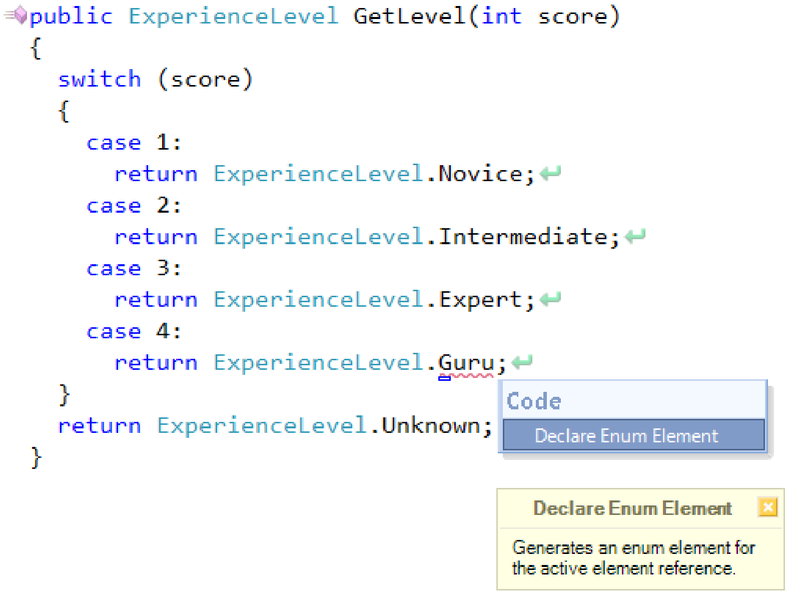
Code
Declare Enum
Declare Class
Declare Property
Declare Property (auto-implemented)
Declare Property (with backing store)
Declare Enum

Generates an enumeration for the type reference.

The resulting code will look like this:

```
public enum ExperienceLevel
{
   Unknown,
   Novice,
   Intermediate,
   Expert
}
```

If you didn’t declare all elements, and are considering adding other, you are able to continue to enter them and then use the **Declare Enum Element** code provider for each new element you would like to add into the enumeration type declaration:



```
public ExperienceLevel GetLevel(int score)
{
  switch (score)
  {
    case 1:
      return ExperienceLevel.Novice;
    case 2:
      return ExperienceLevel.Intermediate;
    case 3:
      return ExperienceLevel.Expert;
    case 4:
      return ExperienceLevel.Guru;
  }
  return ExperienceLevel.Unknown;
}
```

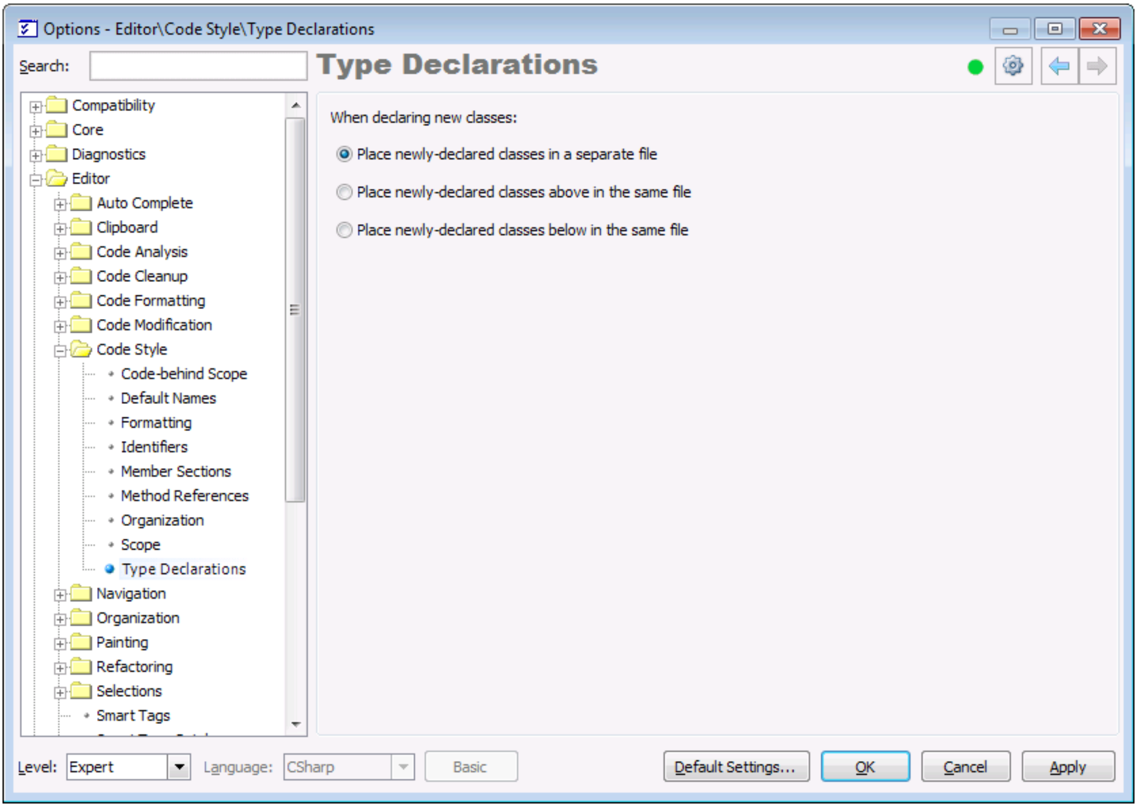
Generates an enum element for the active element reference.

Executing the **Declare Enum Element** will extend the target enumeration type as follows:

```
public enum ExperienceLevel
{
  Unknown,
  Novice,
  Intermediate,
  Expert,
  Guru
}
```

An option for positioning newly declared types

You can change the target position for classes, structures and interfaces created by CodeRush code generation code providers:



Options - Editor\Code Style\Type Declarations

Type Declarations

When declaring new classes:

● Place newly-declared classes in a separate file

○ Place newly-declared classes above in the same file

○ Place newly-declared classes below in the same file

Level: Expert

Language: CSharp

The following code providers are effected on this setting:

- Create Ancestor
- Create Descendant

- Create Descendant (with virtual overrides)
- Create Implementer (implicit)
- Create Implementer (explicit)
- Declare Attribute
- Declare Class
- Declare Class with Properties
- Declare Delegate
- Declare Enum
- Declare EventArgs Descendant
- Declare Interface
- Declare Struct

Declaring members

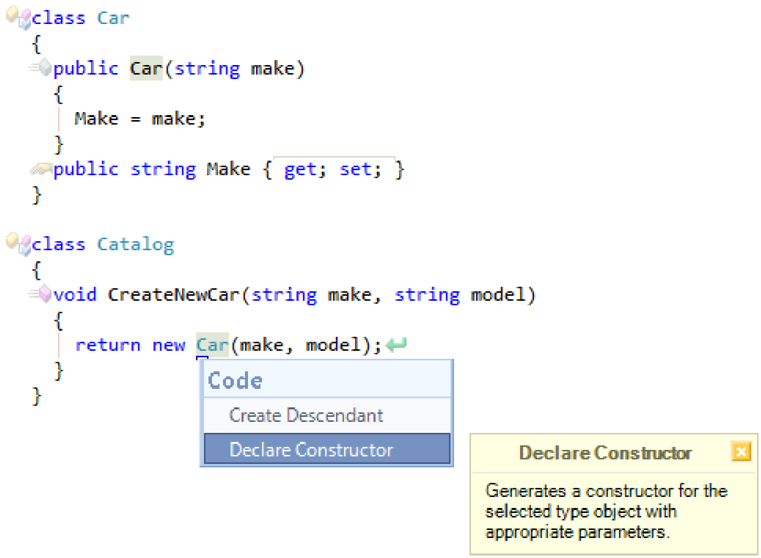
Declaring methods and constructors

When writing a new code, it usually implies writing a not-declared code like a method invocation, property calls, etc. To help you quickly create the required declarations, there are a lot of code generation providers for declaring methods, properties, fields, locals and everything else. Let’s review the method and constructor code declaration providers:

- **Declare Constructor**
- **Declare Method**
- **Declare Method (abstract)**

When instantiating a new class with the ‘new’ operator, we may realize that we need additional or a different set of arguments passed to a class constructor. In this case a constructor does not exist yet, of course. The **Declare Constructor** code provider allows you to create a new constructor at the preferred position, fill out the new parameter names automatically, and allow you to rename them if required.

After the code provider is applied on the following undeclared constructor instantiation:

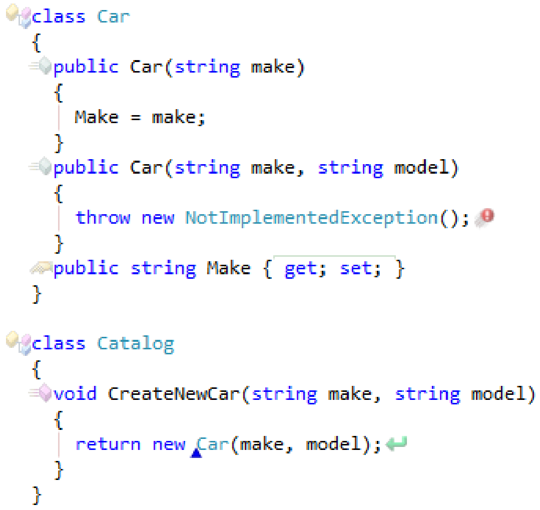


```
class Car
{
  public Car(string make)
  {
    Make = make;
  }
  public string Make { get; set; }
}

class Catalog
{
  void CreateNewCar(string make, string model)
  {
    return new Car(make, model);
  }
}
```

Declare Constructor

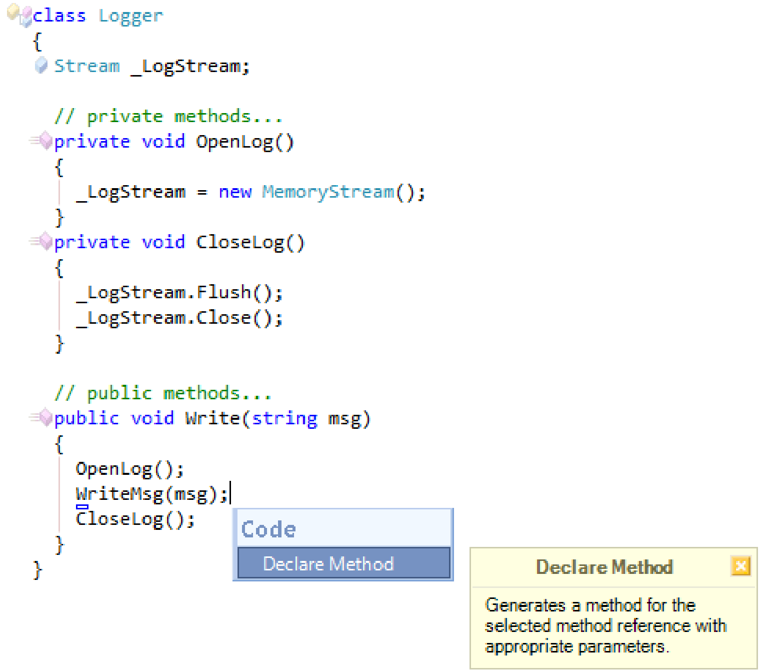
Generates a constructor for the selected type object with appropriate parameters.

it will produce the following code:



```
class Car
{
  public Car(string make)
  {
    Make = make;
  }
  public Car(string make, string model)
  {
    throw new NotImplementedException();
  }
  public string Make { get; set; }
}

class Catalog
{
  void CreateNewCar(string make, string model)
  {
    return new Car(make, model);
  }
}
```

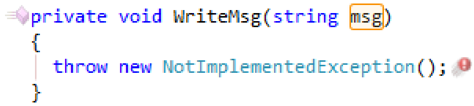
We can do the same with usual the methods, whether they are static or instance. Just write down the method invocation expression and specify the required arguments for the new method. Then, apply the **Declare Method** code provider:



```
class Logger
{
  Stream _LogStream;

   // private methods...
  private void OpenLog()
   {
     _LogStream = new MemoryStream();
   }
  private void CloseLog()
   {
     _LogStream.Flush();
     _LogStream.Close();
   }

   // public methods...
  public void Write(string msg)
   {
     OpenLog();
     WriteMsg(msg);
     CloseLog();
   }
}
```

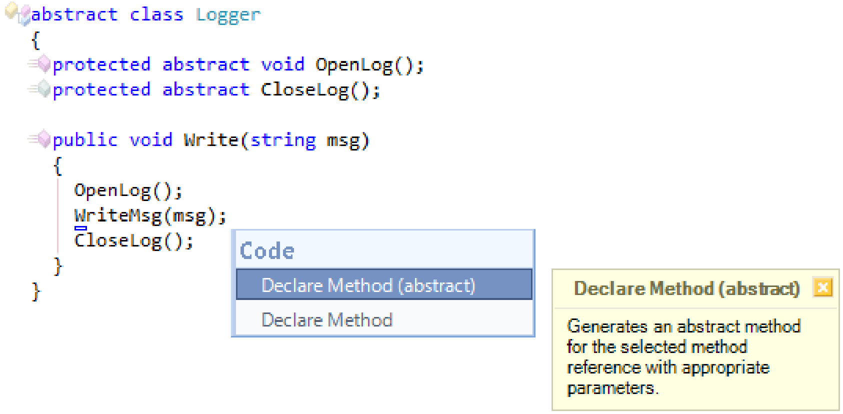
Generates a method for the selected method reference with appropriate parameters.

and get the method declared automatically:



```
private void WriteMsg(string msg)
{
  throw new NotImplementedException();
}
```

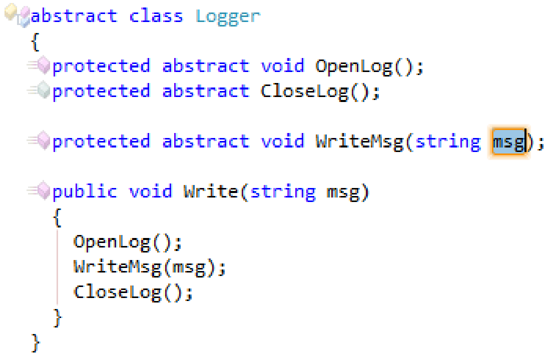
Another version of the code provider called **Declare Method (abstract)** does almost the same thing, but creates an abstract method if you are inside an abstract class:



```
abstract class Logger
{
  protected abstract void OpenLog();
  protected abstract CloseLog();

  public void Write(string msg)
   {
     OpenLog();
     WriteMsg(msg);
     CloseLog();
   }
}
```

Generates an abstract method for the selected method reference with appropriate parameters.

resulting in an abstract method declaration:



```
abstract class Logger
{
  protected abstract void OpenLog();
  protected abstract CloseLog();

  protected abstract void WriteMsg(string msg);

  public void Write(string msg)
  {
    OpenLog();
    WriteMsg(msg);
    CloseLog();
  }
}
```

The difference with the Visual Studio method code generation feature is that CodeRush code providers have the following benefits:

- choose a destination position for a new method
- specify the name of each parameter easily
- toggle the default method options, e.g., scope (public, private, protected, etc)
- quickly navigate inside the new method and get back if required using Markers

Declaring various properties

The consume-first declaration features of CodeRush are a quick way to generate the required code without typing the entire declaration's code by hand. Once you have a reference to an undeclared member, pressing the CodeRush key allows you to choose a member you would like to declare. Let's take a look at the property-declaring code generation providers you can use.

Here they are:

Provider Name	Description
Declare Property	Generates a property for the selected element reference.
Declare Property (auto-implemented)	Generates an auto-implemented property for the selected element reference.
Declare Property (with backing field)	Generates a property with backing store for the selected element reference.
Declare Properties	Generates auto-implemented properties for this object initializer expression.
Declare Initialized Property	Adds an auto-implemented property and initializes it to the parameter at the caret.
Declare Initialized Properties	Adds auto-implemented properties and initializes them to each parameter of this constructor or member.
Declare Getter	Generates a getter of the target Property for the selected element reference.
Declare Setter	Generates a setter of the target Property for the selected element reference.

The first three code generation providers (**Declare Property**, **Declare Property** (**auto-implemented**), **Declare Property** (**with backing field**)) generate the usual property at the target location: inside the current class, or inside the referenced class, if the reference has a type qualifier. In both cases, the final position of the property declaration is being chosen using the target picker.

Consider the following code sample:

```
public class Circle
{
  public static Circle Create()
  {
    Circle circle = new Circle();
    circle.Radius = ١٠;
    return circle;
  }
}
```

The *Radius* identifier references an undeclared element declaration, so we can declare a property for it. Performing the **Declare Property** code provider will result in the following declaration:

```
public int Radius
{
  get
  {
    throw new NotImplementedException();
  }
  set
  {
    throw snew NotImplementedException();
  }
}
```

Choosing the **Declare Property** (**with backing field**) will produce the following code:

```
private int _Radius;
public int Radius
{
  get
  {
    return _Radius;
  }
  set
  {
    _Radius = value;
  }
}
```

And, finally, **Declare Property** (**auto-implemented**) will create a simple property declaration:

```
public int Radius { get; set; }
```

Note that if the reference is not assigned to any value, the **Declare Property** and **Declare Property** (**with backing field**) code provider will produce a read-only property. For example, declaring a property for the Radius reference from this code:

```
public class Circle
{
  public static Circle Create()
  {
    Circle circle = new Circle();
    circle.Diameter = circle.Radius * ٢;
    return circle;
  }
  public int Diameter { get; set; }
}
```

Using the **Declare Property** (**with backing field**) will produce a read-only property with the getter only, because the identifier is not assigned and the value is only retrieved in the code:

```
private int _Radius;
public int Radius
{
  get
  {
    return _Radius;
  }
}
```

Note that the auto-implemented properties should always declare its setter and getter by the language specification. So, declaring a property from a read-only reference, using the Declare Property (auto-implemented) code provider will always generate a read-write property.

After you have a read-only or write-only property declared, and then write a reference to utilize another property's accessor, the **Declare Setter** and **Declare Getter** code providers may help you create one for you. In the sample above, we declared a read-only property. Here's the code:

```
public class Circle
{
  public static Circle Create()
  {
    Circle circle = new Circle();
    circle.Diameter = circle.Radius * ٢;
    return circle;
  }
  private int _Radius;
  public int Radius
  {
    get
    {
      return _Radius;
    }
  }
  public int Diameter { get; set; }
}
```

If you would like to initialize the *Radius* like this:

```
public static Circle Create()
{
  Circle circle = new Circle();
  circle.Radius = ١٠;
  circle.Diameter = circle.Radius * ٢;
  return circle;
}
```

You have to declare a setter for the property. Executing the **Declare Setter** will extend the Radius read-only property with the getter accessor and the property now has read-write access:

```
public int Radius
{
  get
  {
    return _Radius;
  }
  set
  {
    _Radius = value;
  }
}
```

Notice that the setter's code logic was automatically implemented for you as well. The **Declare Getter** generates a getter accessor for the write-only property and converts the property into a read-write property.

Now let's review the **Declare Initialized Property** and **Declare Initialized Properties** code providers. The first one declares an auto-implemented property and initializes it to the parameter at the editor caret. Once your editor caret position is at a parameter to a method or constructor, e.g. 'radius' in the following sample:

```
public class Circle
{
  /// <summary>
  /// Initializes a new instance of the Circle class.
  /// </summary>
  /// <param name=»radius»>The radius of the Circle.</param>
  public Circle(int radius)
  {
  }
}
```

the **Declare Initialized Property** becomes available, and when you perform it, it will produce the following result:

```
public class Circle
{
  /// <summary>
  /// Initializes a new instance of the Circle class.
  /// </summary>
  /// <param name=»radius»>The radius of the Circle.</param>
  public Circle(int radius)
  {
    Radius = radius;
  }
  public int Radius { get; private set; }
}
```

The **Declare Initialized Properties** does absolutely the same thing, but for several parameters. It is available on the name of the method or constructor. If there are at least two parameters that do not have the corresponding properties, it will declare properties for all parameters that do not have an assignment to a property. For example, having this code:

```
public class Circle
{
  /// <summary>
  /// Initializes a new instance of the Circle class.
  /// </summary>
  /// <param name=»radius»>The radius of the Circle.</param>
  /// <param name=»Color»>The color of the Circle.</param>
  public Circle(int radius, Color color)
  {

  }
}
```

and applying the **Declare Initialized Properties** will extend your code to the following one:

```
public class Circle
{
  /// <summary>
  /// Initializes a new instance of the Circle class.
  /// </summary>
  /// <param name=»radius»>The radius of the Circle.</param>
  /// <param name=»Color»>The color of the Circle.</param>
  public Circle(int radius, Color color)
  {
    Radius = radius;
    Color = color;
  }
  public int Radius { get; private set; }
  public Color Color { get; private set; }
}
```

And finally, the **Declare Properties** code provider. This one, as mentioned earlier, generates auto-implemented properties for an object initializer expression. Object initializer expression is constructed like this:

```
public class Circle
{
  public static Circle Create(int radius, Color color)
  {
    return new Circle { Radius = radius, Color = color };
  }
}
```

Notice that the *Radius* and *Color* properties are undefined. Move the editor caret to one of these identifiers and apply the **Declare Properties** code provider. It will result in the following code:

```
public class Circle
{
  public static Circle Create(int radius, Color color)
  {
    return new Circle { Radius = radius, Color = color
};
  }
  public int Radius { get; set; }
  public Color Color { get; set; }
}
```

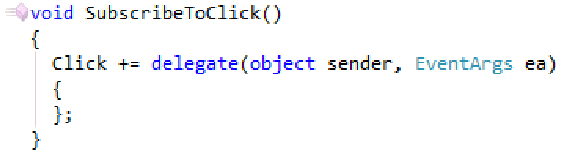
That's all for property generation code providers. See other code-declaring code providers that are shipped in CodeRush Pro. By the way, some of these providers are available in the free CodeRush Xpress version.

Declaring events and event handlers

With the help of CodeRush we can declare everything starting from undeclared local variables and members to undeclared objects like classes and structures. Here, we will take a look at the code declaration providers for declaring events and event handlers:

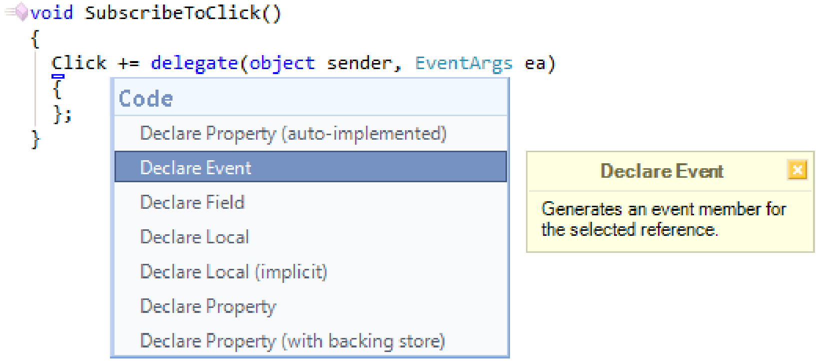
- **Declare Event**
- **Declare Event Handler**

The **Declare Event** code provider creates an event for an undeclared event reference, e.g:



```
void SubscribeToClick()
{
  Click += delegate(object sender, EventArgs ea)
  {
  };
}
```

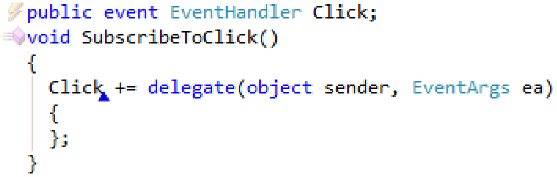
If the Click event is not yet declared, you can apply the **Declare Event** code provider for the event reference:



```
void SubscribeToClick()
{
  Click += delegate(object sender, EventArgs ea)
  {
  };
}
```

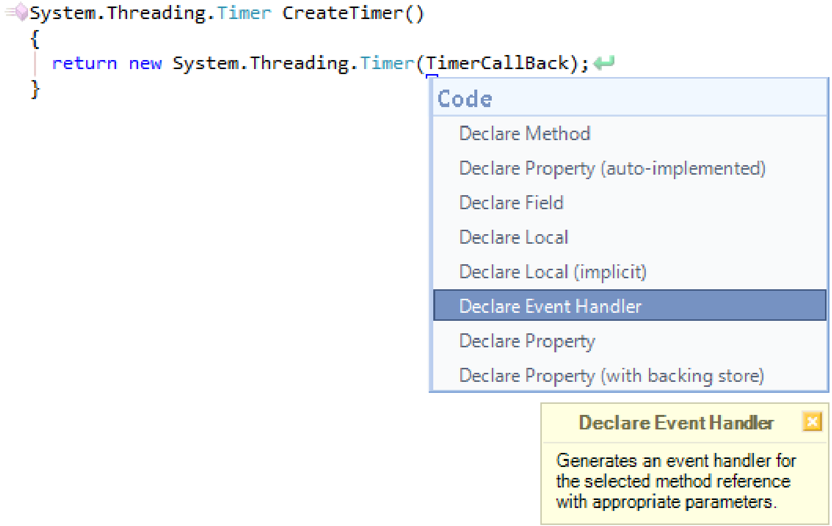
Generates an event member for the selected reference.

The code provider will determine the resulting delegate type automatically:



```
public event EventHandler Click;
void SubscribeToClick()
{
  Click += delegate(object sender, EventArgs ea)
  {
  };
}
```

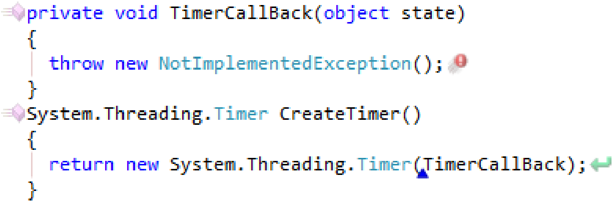
As a complementary addition, there is another code provider which allows you to declare event handlers. The **Declare Event Handler** code provider is available on the reference to an undeclared event handler, e.g.:



```
System.Threading.Timer CreateTimer()
{
  return new System.Threading.Timer(TimerCallBack);
}
```

Generates an event handler for the selected method reference with appropriate parameters.

The Timer object in this sample takes a single argument of the System.Threading.TimerCallBack type that represents a callback method to be executed. Applying the **Declare Event Handler** will generate the corresponding method with the signature of the TimerCallBack delegate:



```
private void TimerCallBack(object state)
{
  throw new NotImplementedException();
}
System.Threading.Timer CreateTimer()
{
  return new System.Threading.Timer(TimerCallBack);
}
```

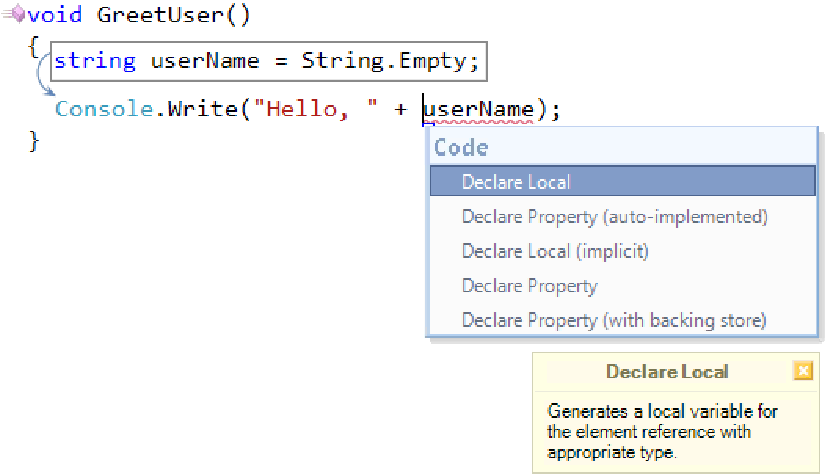
Declaring locals

Local variable declarations are the most often used types of declarations in code. A local variable is a type of variable declared by local variable declaration inside a block the variable is intended to be local to. The local variable declaration explicitly defines the type of the variable that has been declared along with the identifier that names the variable. You can also declare implicitly typed local variables, whose type is inferred by the compiler from the expression.

With the help of **CodeRush** consume-first declaration features, you don't have to type the definition of a local variable declaration by hand. Any local reference to an undeclared element, or a statement that returns a value, but does not assign it to anything, can be automatically declared using consume-first declaration features specific to local variable declarations: **Declare Local** and **Declare Local** (**implicit**).

Before you apply any of the code providers, you can see the resulting code that is being inserted.

The **Declare Local** code provider creates an explicitly-typed local variable declaration, and the **Declare Local** (**implicit**) declares an implicitly-typed local variable declaration. The Declare Local defines the correct type of the new variable declaration from the code:



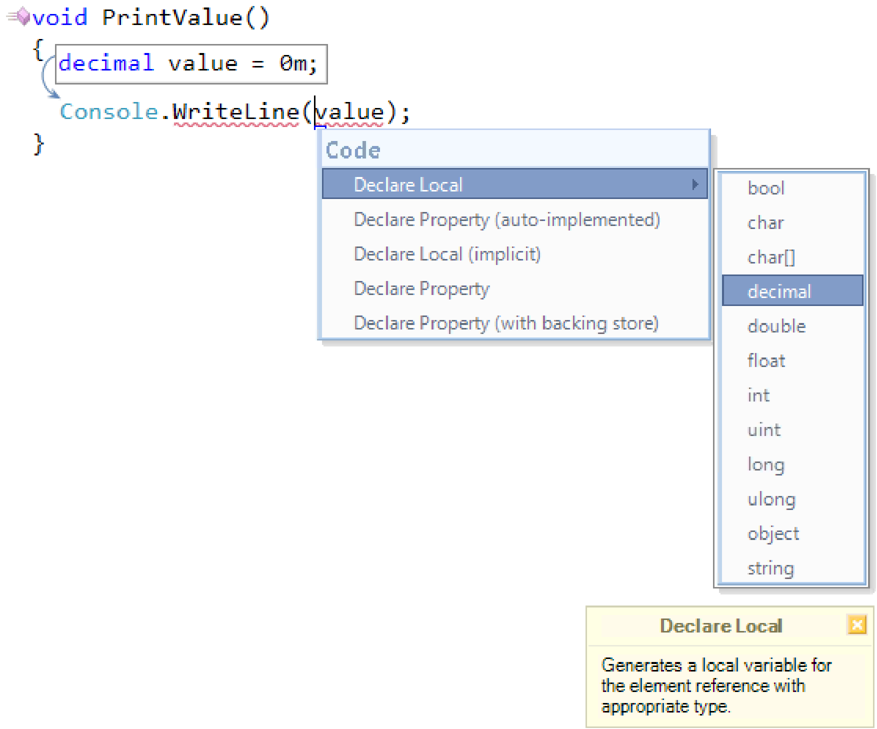
```
void GreetUser()
{
  string userName = String.Empty;
  Console.Write("Hello, " + userName);
}
```

Code
Declare Local
Declare Property (auto-implemented)
Declare Local (implicit)
Declare Property
Declare Property (with backing store)

Declare Local

Generates a local variable for the element reference with appropriate type.

If there are several possible types, you can choose the type from a sub menu:

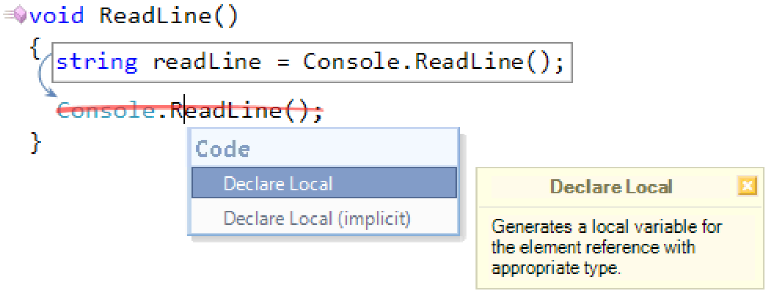


```
void PrintValue()
{
  decimal value = 0m;
  Console.WriteLine(value);
}
```

Declare Local

Generates a local variable for the element reference with appropriate type.

If the **Declare Local** is performed on a variable reference, it declares a named variable on the scope where it is visible for all other variable references. If the code provider is executed on a statement, it's value is assigned to a variable declaration:

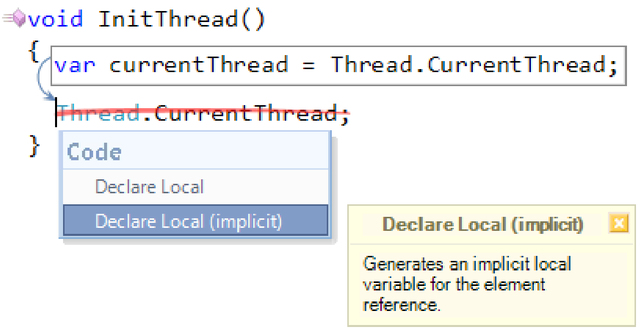


```
void ReadLine()
{
  string readLine = Console.ReadLine();
  ~~Console.ReadLine();~~
}
```

Declare Local

Generates a local variable for the element reference with appropriate type.

The **Declare Local (implicit)** is similar to the **Declare Local** code provider, but it creates an implicitly-typed local variable declaration in case you don't care about variable's type:



```
void InitThread()
{
  var currentThread = Thread.CurrentThread;
  ~~Thread.CurrentThread;~~
}
```

Generates an implicit local variable for the element reference.

See also: consume-first declaration features for field variables.

Declaring fields

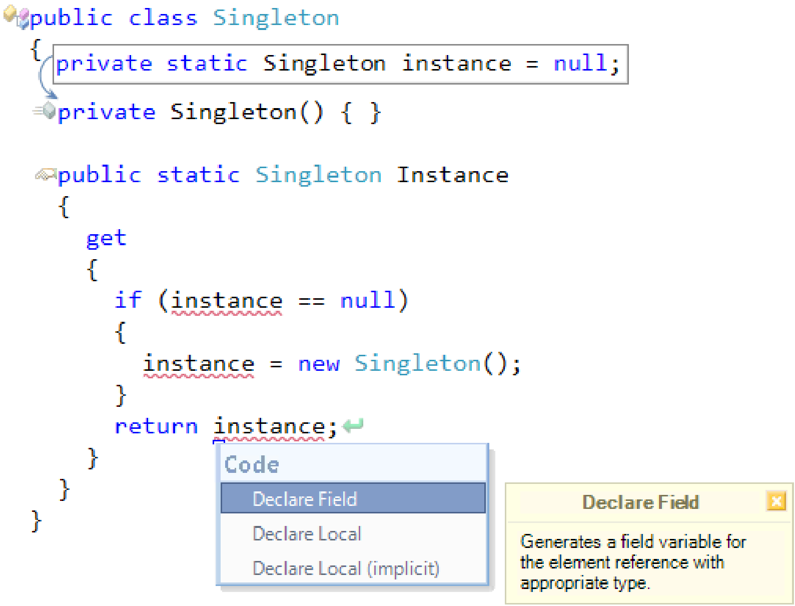
Unlike the local variable declarations, fields are simply variables that are declared directly within the code block of a class or a structure. Fields are declared in the type block by specifying the access level of the field, the type of the field and the name of the field. A field may also include a readonly modifier. This type of field can only have its value set directly in its declaration or from within a constructor of its containing class.

With the help of **CodeRush** consume-first declaration features, you don't have to type the definition of a field variable declaration by hand. Any reference to an undeclared element can be declared automatically for you in a single step. You can also declare a field variable and assign values of the method or constructor parameters. Here are useful code providers specific to field declarations:

- **Declare Field**
- **Declare Field (read-only)**
- **Declare Initialized Field**
- **Declare Initialized Fields**

Before you apply any of the code providers, you can see the resulting code that is being inserted.

The **Declare Field** code provider declares a new private field declaration from an undeclared reference inside the current type declaration:

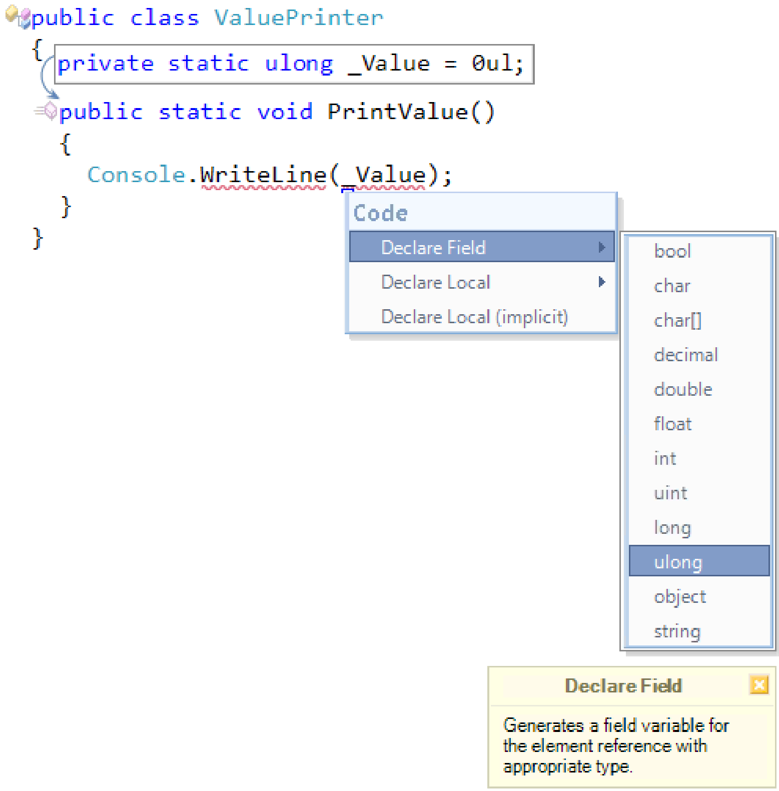


```
public class Singleton
{
  private static Singleton instance = null;
  private Singleton() { }

  public static Singleton Instance
  {
    get
    {
      if (instance == null)
      {
        instance = new Singleton();
      }
      return instance;
    }
  }
}
```

Declare Field

Generates a field variable for the element reference with appropriate type.

If there are several possible types, you can choose the type from a sub menu:



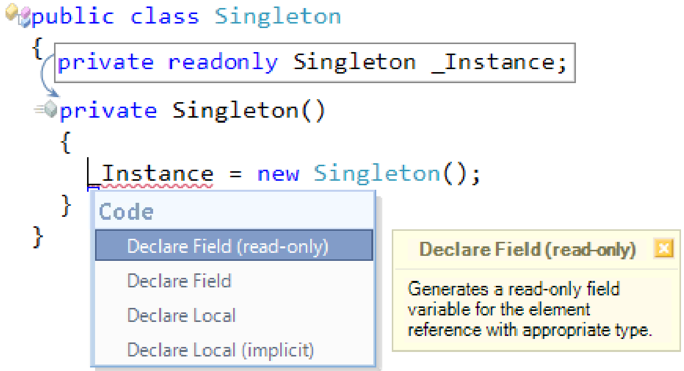
```
public class ValuePrinter
{
  private static ulong _Value = 0ul;
  public static void PrintValue()
  {
    Console.WriteLine(_Value);
  }
}
```

Declare Field

Generates a field variable for the element reference with appropriate type.

The newly created declaration is placed near the other field declaration or at the top of the current type if there are no fields in the class or struct.

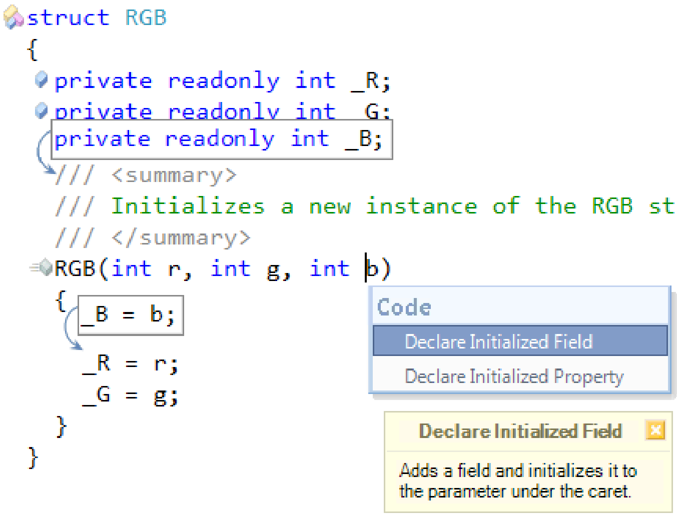
The **Declare Field** (**read-only**) allows you to declare a private read-only field declaration if the reference inside a constructor to an undeclared element that is not assigned:



```
public class Singleton
  {
    private readonly Singleton _Instance;
    private Singleton()
    {
      _Instance = new Singleton();
    }
  }
```

Generates a read-only field variable for the element reference with appropriate type.

The **Declare Initialized Field** and **Declare Initialized Fields** code providers both work with the parameters of a method or a constructor. They declare a field declaration and assign the value of a parameter to the newly created field declaration. The difference between these two code providers is that the Declare Initialized Field is available only for a single method parameter:

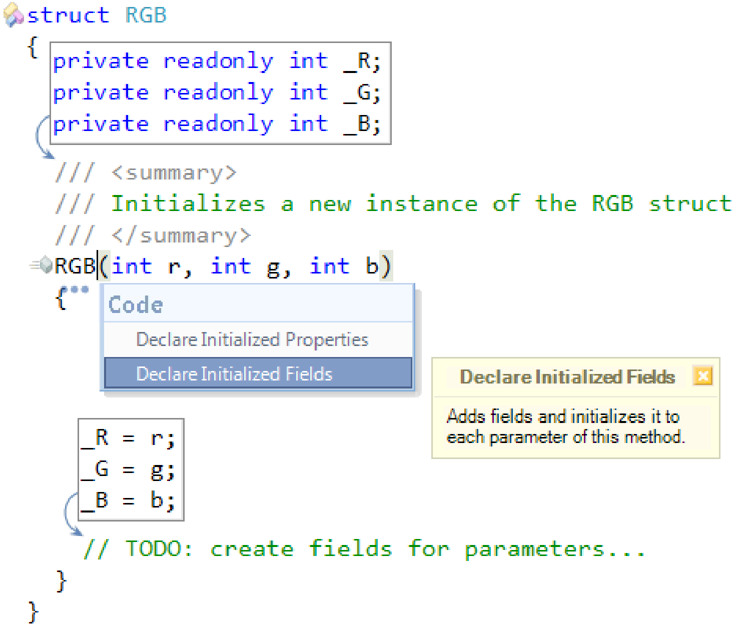


```
struct RGB
  {
    private readonly int _R;
    private readonly int _G;
    private readonly int _B;
    /// <summary>
    /// Initializes a new instance of the RGB structure.
    /// </summary>
    RGB(int r, int g, int b)
    {
      _B = b;
      _R = r;
      _G = g;
    }
  }
```

Code
Declare Initialized Field
Declare Initialized Property
Declare Initialized Field

Adds a field and initializes it to the parameter under the caret.

and the **Declare Initialized Fields** allows you to declare fields for all parameters that are not yet assigned to any field when the editor caret is at the name of the method/constructor:



```
struct RGB
{
  private readonly int _R;
  private readonly int _G;
  private readonly int _B;

  /// <summary>
  /// Initializes a new instance of the RGB structure.
  /// </summary>
  RGB(int r, int g, int b)
  {
    _R = r;
    _G = g;
    _B = b;
    // TODO: create fields for parameters...
  }
}
```

Declare Initialized Fields

Adds fields and initializes it to each parameter of this method.

Both providers add linked identifiers that allow you to immediately rename the field declaration and the assignment reference. Also, a marker is dropped to be able to quickly navigate back to the starting source location.

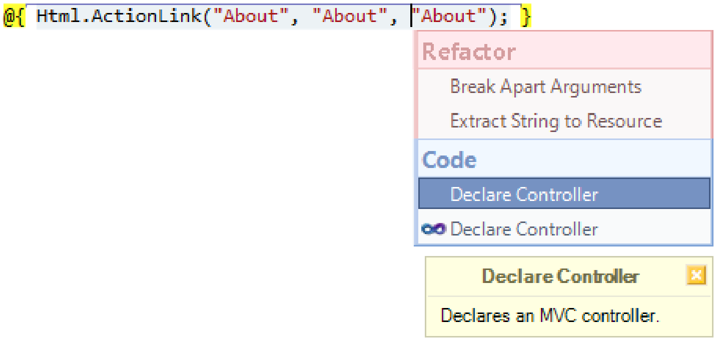
See also: consume-first declaration features for local variables.

MVC-specific providers

There are three code generation providers specific to MVC. They allow you to declare MVC controllers, MVC actions and MVC views correspondingly:

- **Declare Controller**
- **Declare Action**
- **Declare View**

The **Declare Controller** code provider creates a new MVC controller if one is undeclared. The code provider is available on the corresponding controller name argument of the Action, ActionLink, or RenderAction method invocation, e.g.:

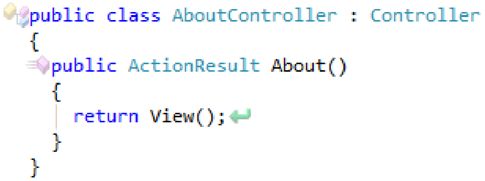


```
@{ Html.ActionLink("About", "About", "About"); }
```

Declare Controller

Declares an MVC controller.

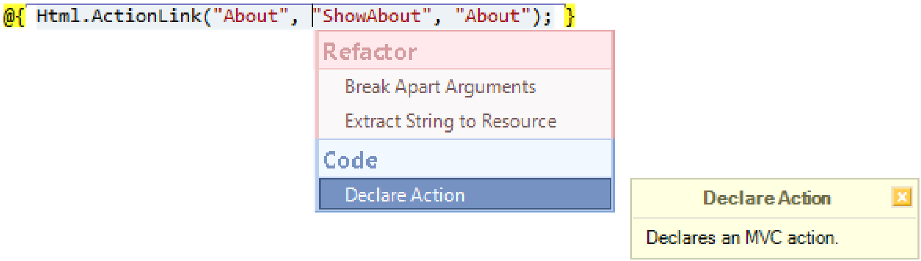
The referenced action is automatically added to the generated controller class:



```
public class AboutController : Controller
{
  public ActionResult About()
  {
    return View();
  }
}
```

There is also a second version of the code provider which simply calls the Visual Studio built-in dialog for declaring a new controller. You can differentiate between the two code provider by using a special icon over it in the Popup menu.

The **Declare Action** code provider creates a new MVC action from the reference to an action which is not yet declared. The code provider is available at the corresponding action name argument of the Action, ActionList, or RenderAction method invocation when the appropriate controller from the same method invocation exist:

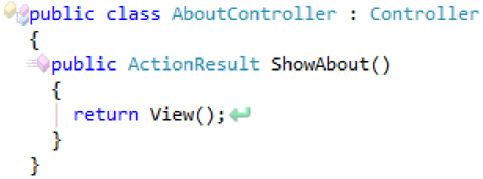


```
@{ Html.ActionLink("About", "ShowAbout", "About"); }
```

Declare Action

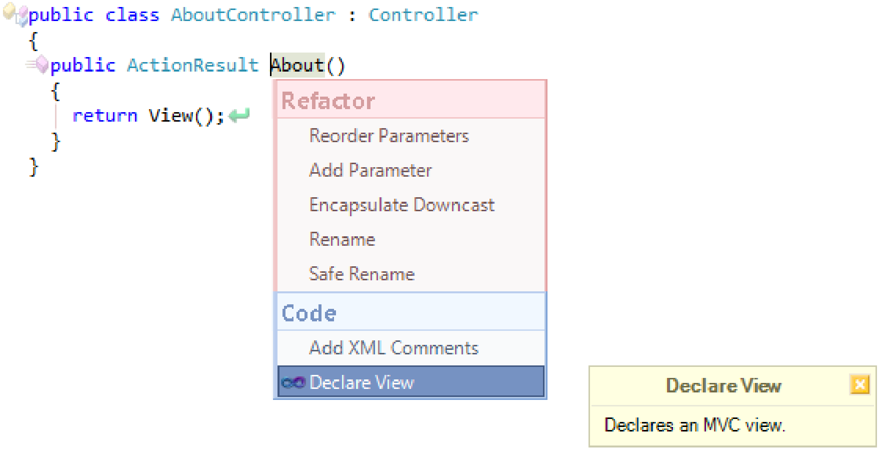
Declares an MVC action.

The result will be declared in the appropriate MVC controller:



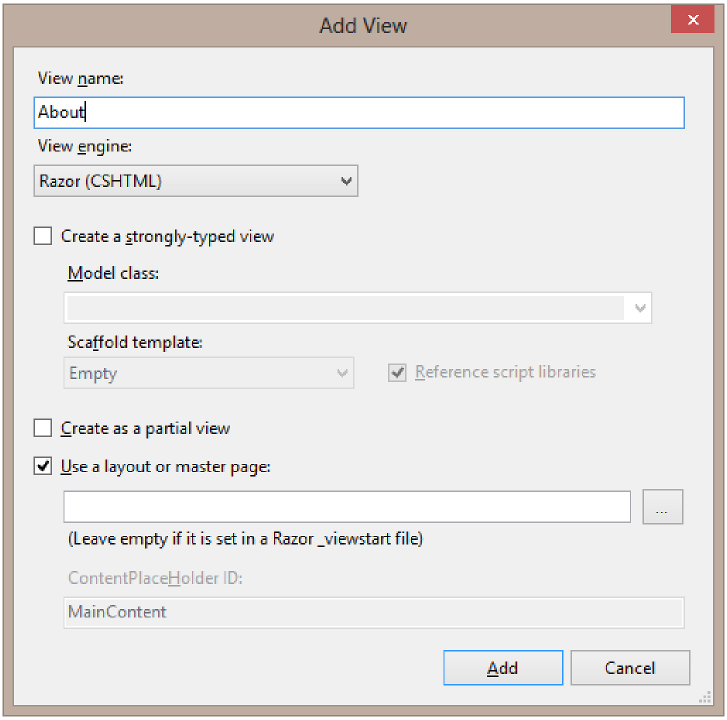
```
public class AboutController : Controller
{
  public ActionResult ShowAbout()
  {
    return View();
  }
}
```

The **Declare View** code provider invokes the Visual Studio built-in Add View code generation feature on the reference to non-existent View:



```
public class AboutController : Controller
{
  public ActionResult About()
  {
    return View();
  }
}
```

Once applied, the Add View dialog appears, where you can specify the required parameters for the new MVC View:



Add View

View name:
About

View engine:
Razor (CSHTML)

□ Create a strongly-typed view

Model class:

Scaffold template:
Empty

☑ Reference script libraries

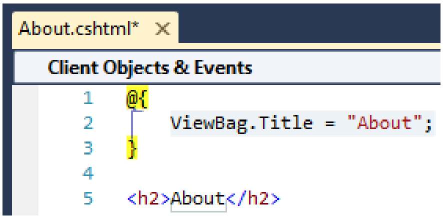
□ Create as a partial view

☑ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Once confirmed, the new MVC view will be added:



```
@{
    ViewBag.Title = "About";
}

<h2>About</h2>
```

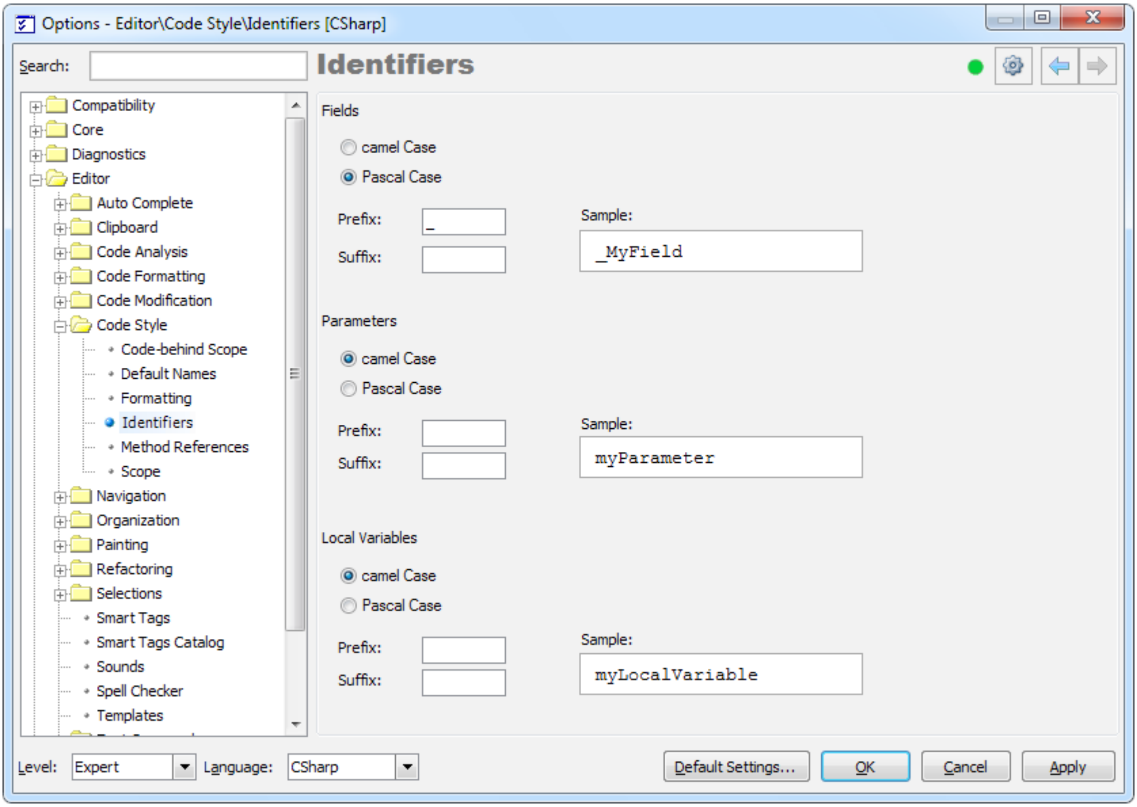
Chapter 5. Coding helpers and assistants

Auto Declare

Auto Declare allows you to quickly declare a variable (field, local or parameter) based on its type. To apply **Auto Declare**, type the class name and press the **Ctrl+A** shortcut. The variable will be declared with an appropriate name, and an action hint will be shown:

```
System.IFormatProvider
```

As you see, the identifier name is inherited from its type. The name of a variable is formatted according to your identifier code style settings. The style settings can be set for fields, local variables and parameters separately on the "**Editor** | **Code Style** | **Identifiers**" options page in the Options Dialog:



Options - Editor\Code Style\Identifiers [CSharp]

Search:

Identifiers

Fields

○ camel Case
● Pascal Case

Prefix: _
Suffix:

Sample: _MyField

Parameters

● camel Case
○ Pascal Case

Prefix:
Suffix:

Sample: myParameter

Local Variables

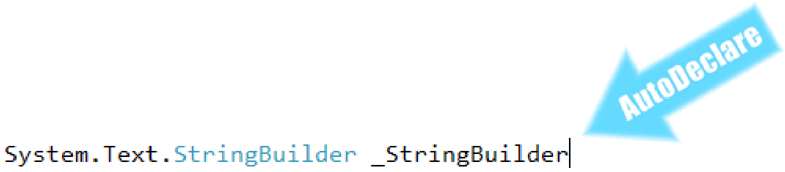
● camel Case
○ Pascal Case

Prefix:
Suffix:

Sample: myLocalVariable

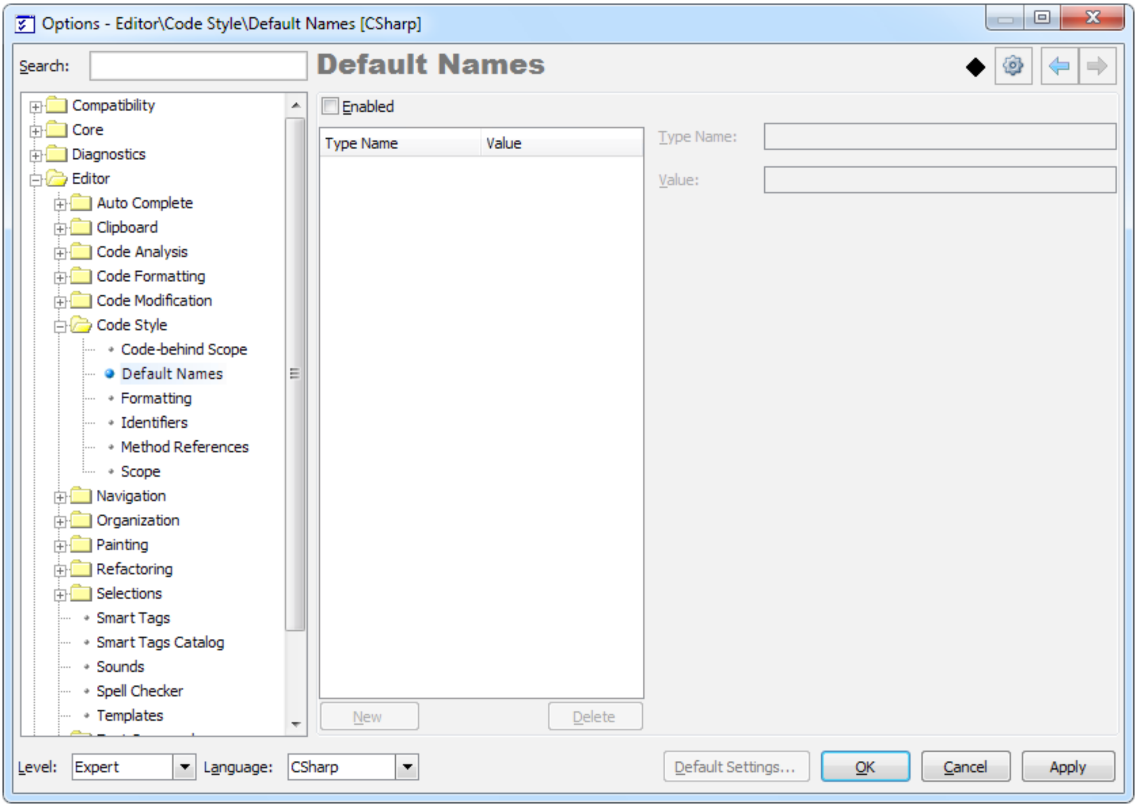
Level: Expert
Language: CSharp

For example, the field of the **StringBuilder** type will be named **_StringBuilder**:



```
System.Text.StringBuilder _StringBuilder
```

You can also specify the default names for identifiers of a certain class on the “Editor | Code Style | Default Names” options page:



Options - Editor\Code Style\Default Names [CSharp]

Default Names

☐ Enabled

Type Name	Value

Type Name:

Value:

Level: Expert

Language: CSharp

The *Enabled* option specifies whether the default names are used by **Auto Declare**. If the option is turned off, the declared identifier inherits the name from its type. Otherwise, if the *Enabled* option is turned on, **AutoDeclare** will use default names from this options page.

To add or delete a default name for a specific type, use the *New* and *Delete* buttons appropriately. For example, let’s specify the default name for the StringBuilder type to be just an “sb”.

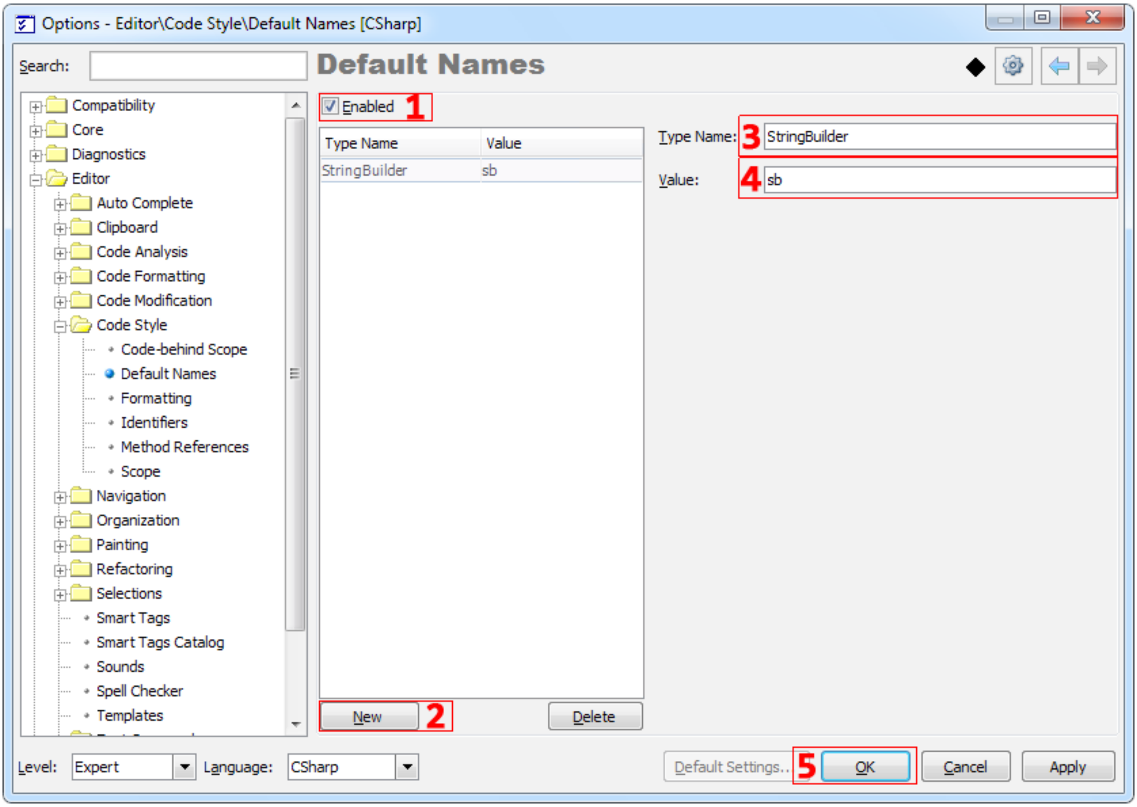
1. Click on the Enabled checkbox if it is not checked

2. Click on the New button

3. Fill the TypaName field

4. Fill the Value field

5. Click the OK button or press the Enter key to save your changes:



Options - Editor\Code Style\Default Names [CSharp]

Default Names

☑ Enabled

Type Name	Value
StringBuilder	sb

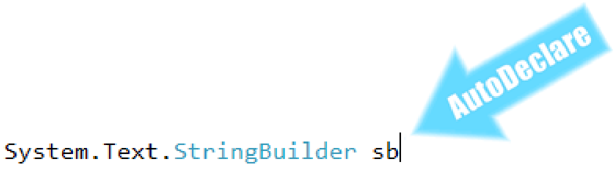
Type Name: StringBuilder

Value: sb

Level: Expert

Language: CSharp

Now, when we declare a local variable of the **StringBuilder** type, we will get the following result:



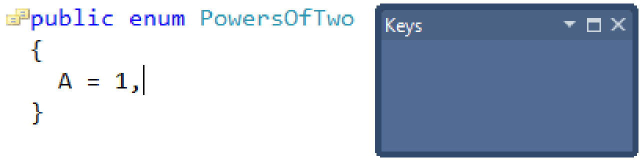
```
System.Text.StringBuilder sb
```

Intelligent code lines enhancement and modification using the Duplicate Line feature

The **Duplicate Line** CodeRush feature intelligently creates a copy of an existing line of code, allowing you to easily modify it. If the code line is recognized as a predefined pattern, the line is copied and a portion of it is selected for further modification. Note that some code line bits can be modified automatically for you as well.

The predefined duplication patterns are configurable. They are stored and presented as regular expressions that contain one or several DXCore built-in reg-ex aliases for easy understanding. If the line doesn't match any of the existing patterns – nothing happens, by default. However, there an option to copy-paste the entire line of the text if no matches are found when the feature is performed. All available options are listed later.

To perform the **Duplicate Line**, press the **Shift+Enter** key anywhere on the line:



```
public enum PowersOfTwo
  {
    A = 1,
  }
```

You can duplicate variable declarations, constants, events, single-line interface members, using statements, method calls, and other code pieces. Here's the list of predefined duplication patterns shipped with CodeRush:

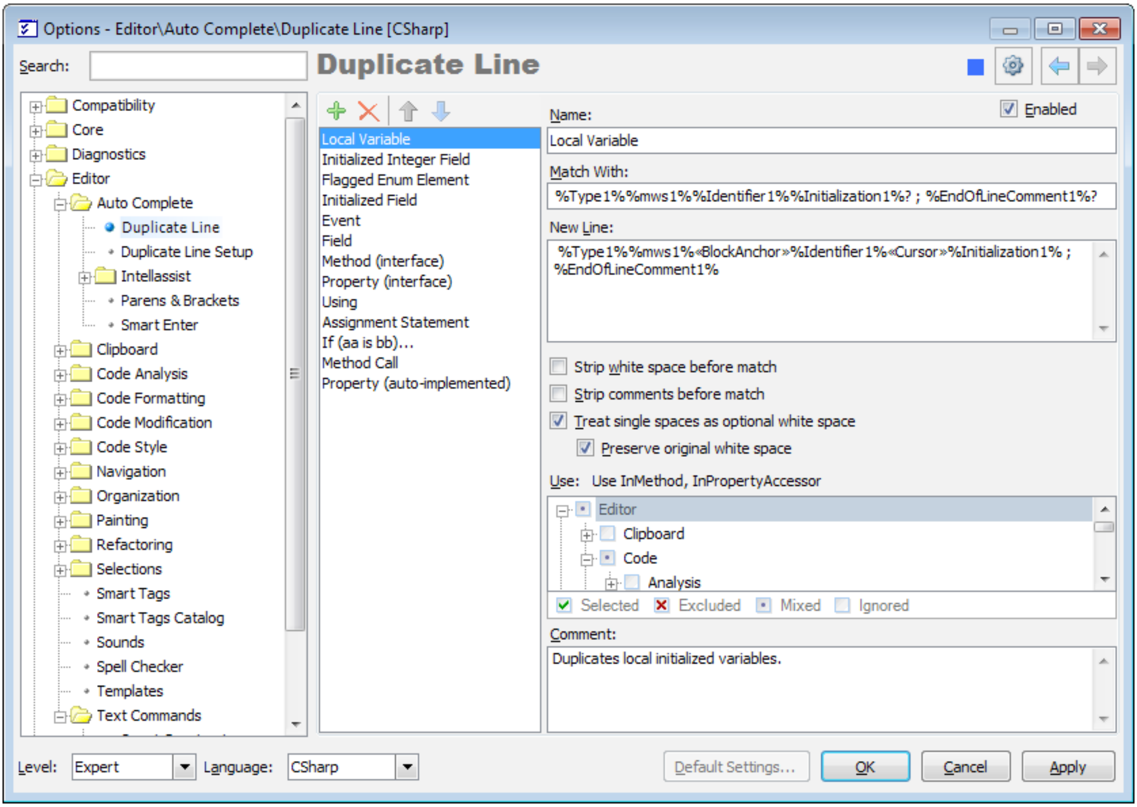
Pattern Name	Description	Sample code

Local Variable	Duplicates local variables, allowing you to change its name.	C#: string varName;
		VB: Dim varName As String;

Field	Duplicates field declarations.	C#: private string fieldName; VB: Private fieldName As String
Event	Duplicates event declarations.	C#: public event EventHandler MyEvent; VB: Public Event MyEvent As EventHandler
Property (auto-implemented)	Duplicates single-line auto-implemented property declarations.	C#: public string PropertyName {get; set;} VB: Public Property PropertyName As String
Using/Imports	Duplicates using (Imports) statements.	C#: using System.Drawing; VB: imports System.Drawing
Initialized Field	Duplicates an initialized field declaration (bringing the initialization to the new line).	C#: private bool fieldName = true; VB: Private fieldName As Boolean = True
Initialized Integer Field	Duplicates an initialized field of type integer. The initialization value is incremented by one against the source line value.	C#: private int fieldName = 1; VB: Private fieldName As Integer = 1
Method (Interface)	Duplicates interface method declarations.	C#: void MethodName(); VB: Sub MethodName()
Property (Interface)	Duplicates interface property declarations.	C#: String PropertyName { get {} set {} } VB: Property PropertyName() As String
Assignment Statement	Duplicates an assignment statement, selecting the left side of the statement.	C#: Identifier = value; VB: Identifier = value
if (aa is bb)	Matches if-expressions checking to see if object (O) is of type T. Declares a new local variable of type T, initialized to O (casts it to T type).	C#: if (myObject is SomeType) VB: If (myObject Is SomeType)

Flagged Enum Element	Duplicates an initialized element of an enumeration, doubling the initialization value on the next line.	C#: Identifier = 1, VB: Identifier = 1

Because **Duplicate Line** is based on the regular expression engine (similar to the Intelligent Paste feature), you can use it in all CodeRush supported languages. All duplication patterns are configurable on the **Editor** | **Auto Complete** | **Duplicate Line** options page in the Options Dialog:



Local Variable
Initialized Integer Field
Flagged Enum Element
Initialized Field
Event
Field
Method (interface)
Property (interface)
Using
Assignment Statement
If (aa is bb)…
Method Call
Property (auto-implemented)

Name: Local Variable ☑ Enabled

Match With:

```
%Type1%%mws1%%Identifier1%%Initialization1%? ; %EndOfLineComment1%?
```

New Line:

```
%Type1%%mws1%«BlockAnchor»%Identifier1%«Cursor»%Initialization1% ;
%EndOfLineComment1%
```

□ Strip white space before match
□ Strip comments before match
☑ Treat single spaces as optional white space
☑ Preserve original white space

Use: Use InMethod, InPropertyAccessor

Comment: Duplicates local initialized variables.

Here you are able to change any existing patterns, or create your own, using the appropriate buttons. On the left, you can observe the list of duplication patterns and change their order. The order is important, because patterns are evaluated in the order that they appear in the list. You can move items of the list with the arrow buttons provided or via the right-clicking context menu. More specific duplications should be listed before more general duplications.

To create a new duplication pattern, click the *New* button or press the *Insert* key. You can also create a new pattern based on an existing one, by right-clicking on the pattern from the list, and choosing the "*Duplicate*" menu item in the context menu. Note that it is possible to create patterns that match "*subsets*" of other patterns. For example, a field that is initialized to an integer ("*Initialized Integer Field*") is a subset of "*Initialized Field*" pattern. To remove the pattern, click the *Delete* button or press the *Ctrl+Delete* shortcut.

On the right side of the options page you can modify options for a particular duplication pattern, selected on the left side. Available options are:

Option Name	Description
Name	The name of the duplication pattern. Appears in the Feature UI hint.
Match With	The regular expression pattern of the code line where this pattern is available.

New Line
The regular expression pattern of the resulting duplicated code line.

Strip white space before match
Specifies whether the preceding whitespace in the source line is ignored.

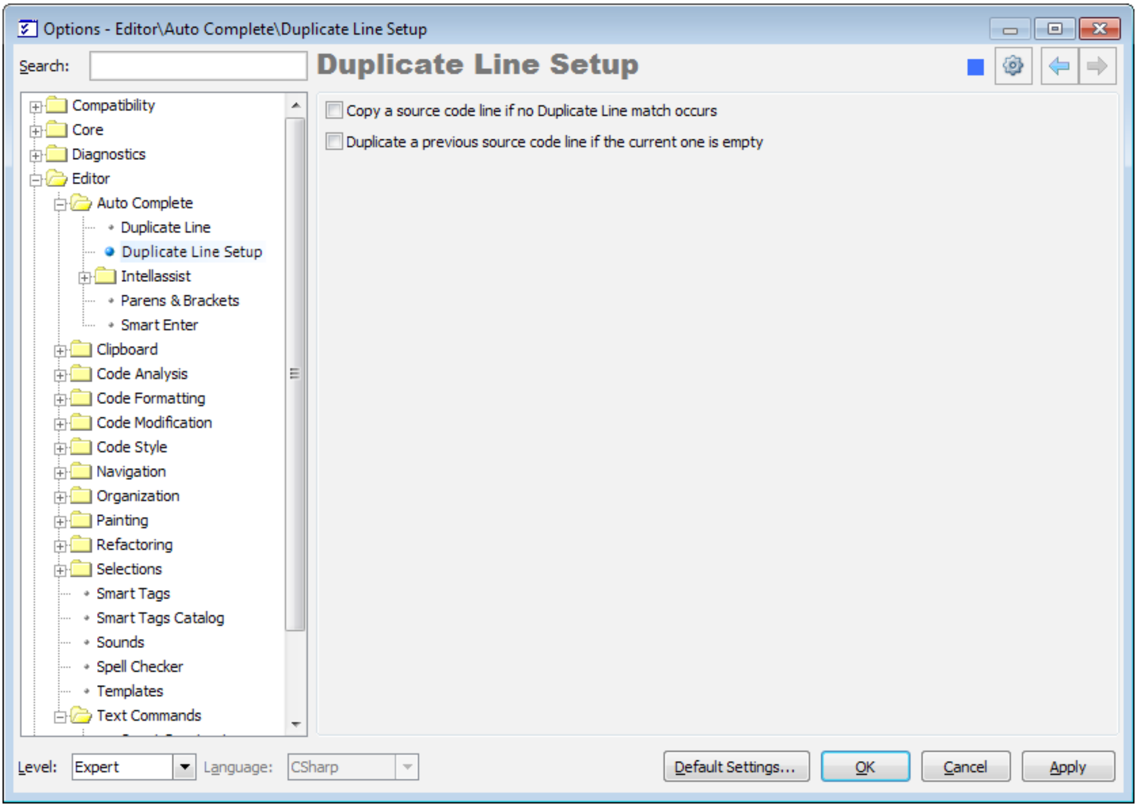
Strip comments before match
Specifies whether the preceding comments in the source line are ignored.

Treat single spaces as optional white space
If this option is on, **Duplicate Line** treats single spaces as an optional whitespace from the **Match With** and **New Line** values. This allows you to enter these values easily, which makes them more flexible in regard to whitespaces (tabs, spaces, several spaces in a row, etc).

Use
The context for the duplication pattern where it is available.

Comment
The comment for the duplication pattern.

Two additional options are available on another options page: **Editor** | **Auto Complete** | **Duplicate Line Setup**:



Options - Editor\Auto Complete\Duplicate Line Setup

Search:

Duplicate Line Setup

☐ Copy a source code line if no Duplicate Line match occurs

☐ Duplicate a previous source code line if the current one is empty

Level: Expert

Language: CSharp

- Copy a source code line if no Duplicate Line match occurs

The options specifies whether the entire line of text should be copy-pasted, if no duplication patterns match the current line.

- Duplicate a previous source code line if the current one is empty

The option specifies whether you are able to perform the **Duplicate Line** feature on an empty line, followed by the line you were going to duplicate. This is useful when you moved the editor caret to the next line (e.g. by pressing the *Enter* key at the end of line) and realized that you can actually duplicate the previous line with manual typing.

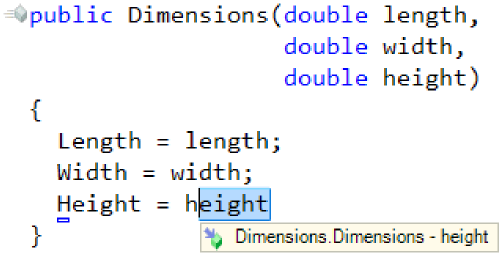
As always, any shortcuts like *Shift+Enter* for the **Duplicate Line** are configurable on the IDE | Shortcuts options page in the IDE Tools Options Dialog.

Note that the **Duplicate Line** feature is also available in the free CodeRush Xpress version.

Code editor auto-complete features – Intellassist

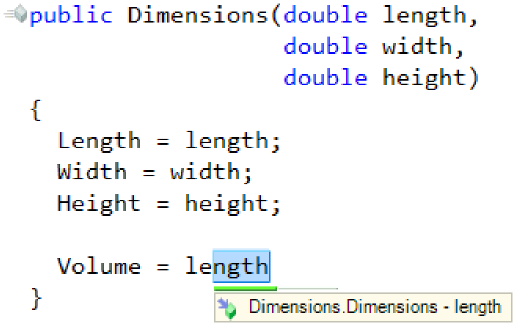
The **Instellassist** feature of DevExpress CodeRush Pro is similar to the built-in Intellisence feature of Visual Studio. It provides additional hints that pop up as you type the code and completes the edited code with an in-scope identifier or other suggestions. These hints help you select code elements or complete the text you need in the context of what you are working on without retyping the entire text.

To use **Intellassist**, you should simply type a text as you would normally. Intellassist activates automatically once you type a couple of characters and it has at least a single suggestion. You can tweak the Intellassist activation for different contexts such as a code, a string or a comment. When Intellassist senses one or more suggestions matching the code you have typed so far, the best suggestion will be displayed to the right of the editor caret as a selected text:



```
public Dimensions(double length,
                  double width,
                  double height)
  {
    Length = length;
    Width = width;
    Height = height
  }
```

If there are multiple suggestions, you can see the number of suggestions in a small bar under the current line and select any by cycling them as required:



```
public Dimensions(double length,
                  double width,
                  double height)
  {
    Length = length;
    Width = width;
    Height = height;

    Volume = length
  }
```

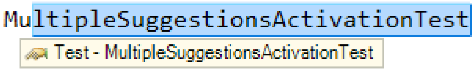
The suggested text does not actually exist in the code editor, so, you can simply ignore a suggestion and continue typing without paying attention to it or accept a suggestion.

Available Actions

The following actions may be performed when an **Intellassist** hint is active:

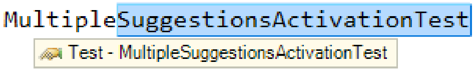
- Press the Enter key to accept the active suggestion.
- Press the Shift+Enter key to accept a portion of the active suggestion.

- Press the Shift+Enter to accept a part of a suggestion starting from the editor caret to the character preceding the next uppercase letter in the suggestion. For example, if “*MultipleSuggestionsActivationTest*” was the suggestion, and the first two letters (“*mu*“) had been typed:

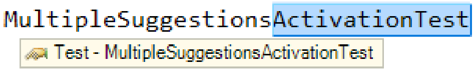


```
MultipleSuggestionsActivationTest
```

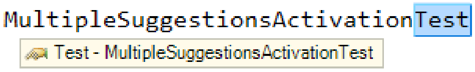
pressing Shift+Enter successive times would cause the selection to shift as follows (each picture illustrates the next Shift+Enter press):



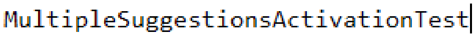
```
MultipleSuggestionsActivationTest
```



```
MultipleSuggestionsActivationTest
```



```
MultipleSuggestionsActivationTest
```



```
MultipleSuggestionsActivationTest
```

Accepting a part of a suggestion is useful when you need to create a new identifier name that is similar to an existing identifier, or when you want to quickly access a different but similarly-named suggestion. You can press Shift+Enter to accept a part of the suggestion, and then start typing different characters to get other suggestions.

- Press the Tab and Shift+Tab shortcuts to cycle forward and backward through the list of multiple suggestions.

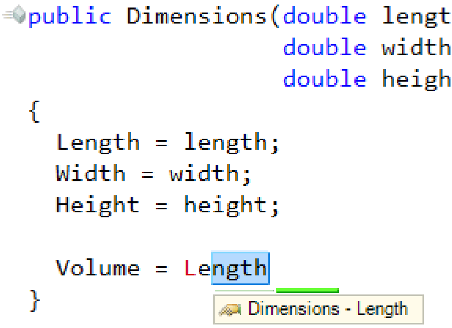
- Press the Delete key to cancel the active suggestion.

- Wait for a few moments and Intellassist will automatically hide the active suggestion.

- Continue typing the text completely ignoring the suggestion or narrowing down the suggestion list for further acceptance.

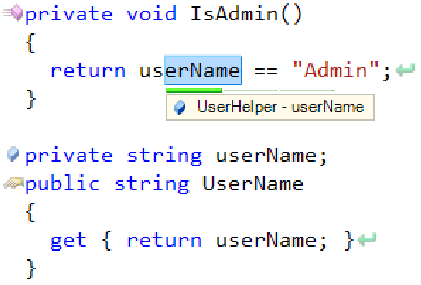
Case Sensitivity

Intellassist suggestions are case-insensitive by default. If Intellassist suggests an identifier that is a case-insensitive match to the text you have typed, it will highlight the mismatched characters:



```
public Dimensions(double length,
                  double width,
                  double height)
{
  Length = length;
  Width = width;
  Height = height;

  Volume = Length
}
```

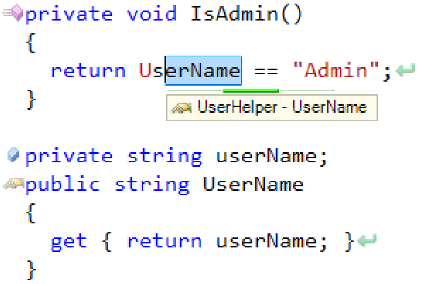
This is useful if you are working in a case-sensitive language such as C# or C++. For example, in the following code there are two suggestions are available for the text “us” typed:



```
private void IsAdmin()
{
  return userName == "Admin";
}

private string userName;
public string UserName
{
  get { return userName; }
}
```

One of the suggestions is a “userName”, which is a private field variable, and the other one is “UserName”, which is a property. The first suggestion starts with “us”. This is an exact match, so no characters are highlighted.

However, the second suggestion starts with “Us”, which is a case-insensitive match. So, an uppercase “U” is drawn over the “u” you typed, indicating changes that will occur to the code you have entered when you accept this suggestion:



```
private void IsAdmin()
{
  return UserName == "Admin";
}

private string userName;
public string UserName
{
  get { return userName; }
}
```

Note that the icon in the hint allows you to differentiate the suggested member names.

When the case-sensitivity option is turned off, there is another option to suggest only case-sensitive matches when the text you typed contains any uppercase letters. So in the example above, entering “Us” would result in only a single suggestion – “UserName”.

Configuration

Intellassist is highly configurable. On the Intelliassist Activation options page you can specify the number of characters to match before Intellassist is activated. On the Intelliassist Setup options page you can specify whether case-insensitive suggestions can be made, as well as the auto-hide and sorting options. To learn more about Intellassist options, read the appropriate topic.

Extensibility

Intellassist is also highly extensible. DXCore provides a special component called Intellassist Extension, which can extend the default suggestions with additional hints. Intellassist uses several extensions by default:

- Identifier Assist – the main suggestions provider, which suggests hints for the in-scope identifiers.

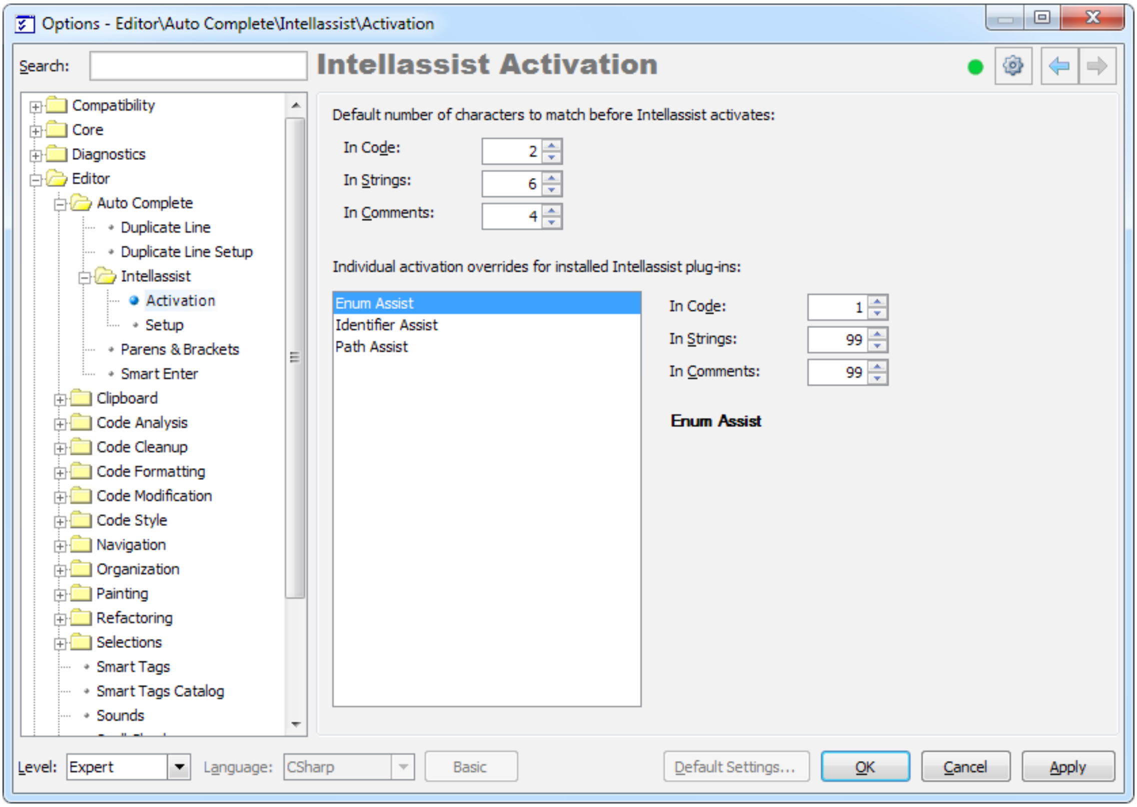
- Enum Assist – suggests elements of an enumeration.

- Path Assist – suggests phisical paths to the file on a disk.

Intellassist configuration and options

Intellassist feature of CodeRush has two options pages in the Options Dialog. Both of them configure this auto-comg plete feature and its extensions.

Activation options page



Intellassist Activation

Default number of characters to match before Intellassist activates:

In Code: 2
In Strings: 6
In Comments: 4

Individual activation overrides for installed Intellassist plug-ins:

Enum Assist
Identifier Assist
Path Assist

In Code: 1
In Strings: 99
In Comments: 99

Enum Assist

Level: Expert
Language: CSharp

This page allows you to set the number of characters to match before Intellassist activates in three different contexts. The contexts are:

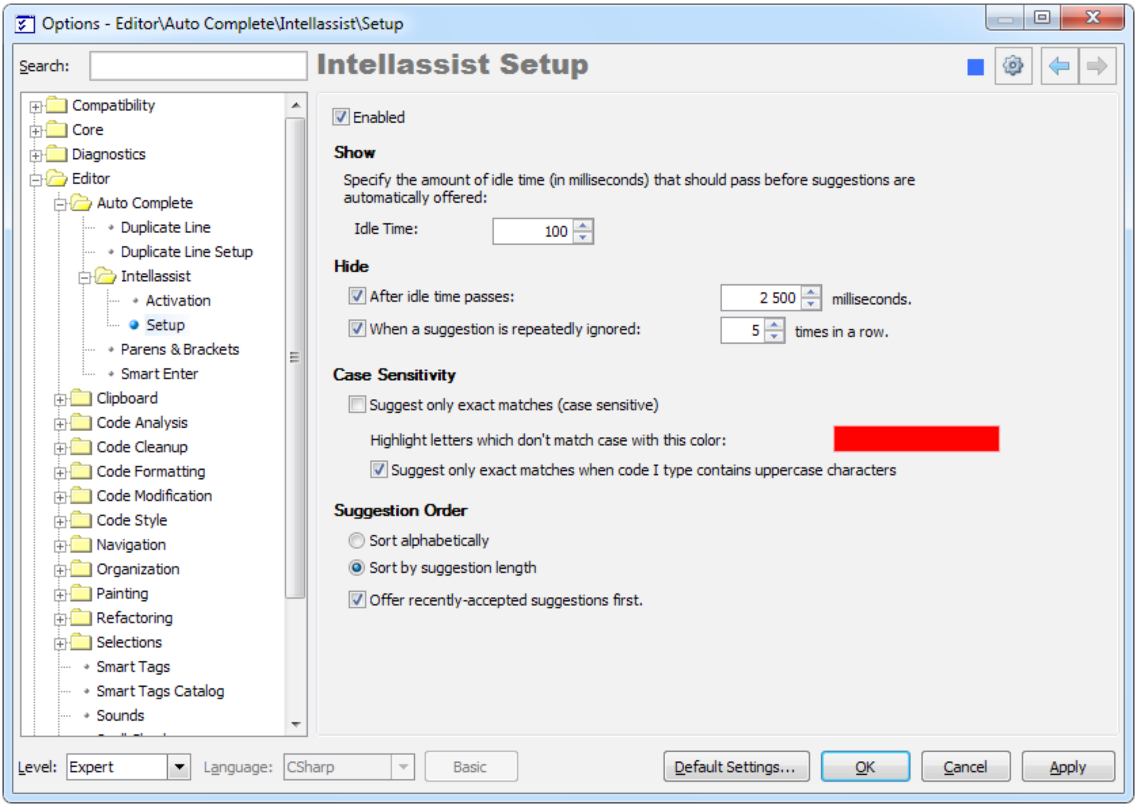
- In Code – when you type a text inside the source code

- In Strings – when you type a text inside the quotes of the string expression

- In Comments – when you type a text inside a comment or an xml doc comment

You can set the default number of characters and individually for each **Intellassist Extension** available. The available providers are listed in the installed Intellassist plug-ins area. If you create your own extension using the corresponding Intellassist Extension **DXCore** control, you will see your extension in the list automatically if your plug-in is loaded.

Setup options page



Options - Editor\Auto Complete\Intellassist\Setup

Intellassist Setup

☑ Enabled

Show

Specify the amount of idle time (in milliseconds) that should pass before suggestions are automatically offered:

Idle Time: 100

Hide

☑ After idle time passes: 2 500 milliseconds.

☑ When a suggestion is repeatedly ignored: 5 times in a row.

Case Sensitivity

☐ Suggest only exact matches (case sensitive)

Highlight letters which don't match case with this color: 🟥

☑ Suggest only exact matches when code I type contains uppercase characters

Suggestion Order

○ Sort alphabetically

● Sort by suggestion length

☑ Offer recently-accepted suggestions first.

Level: Expert

Language: CSharp

This options page allows you to tweak the following options:

- **Enabled**

Specifies whether or not Intellassist can provide any suggestions. If unchecked, you will not see the Intellassist functionality.

- **Show**

Specifies the amount of idle time (in milliseconds) that should pass before suggestions are automatically offered.

- **Hide**

Specifies two options when Intellassist suggestions should be automatically hidden. You can set the amount of idle time (in milliseconds) that should pass before suggestions are hidden, or the number of repeatedly ignored suggestions in a row after which Intellassist won't show the suggestion any longer.

- **Case Sensitivity**

Specifies the two case-sensitivity options when only exact case-sensitive matches should be offered: always or when you type the uppercase characters. Also, you can choose the color that is used to highlight the characters that do not match the case in which they have been typed, if case-sensitivity is allowed (checked).

- **Suggestion order and sorting**

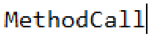
Specifies the sorting options – whether the suggestions are sorted alphabetically or by length. Also, you can move the recently-accepted suggestions to the top, so you will see them first in the list.

Smart Parentheses and Smart Brackets

Let's talk about two similar features – the **Smart Parentheses** (also known as **Smart Parens**) and **Smart Brackets**. These two features allow you to easily type parens and brackets without having to close them with the corresponding closing paren or bracket, because they will be inserted automatically.

In this article the 'paren' means the opening or closing parenthesis '(' or ')', and the bracket means the '[' (opening) and ']' (closing) characters.

Once you type an opening paren (or bracket), the closing paren (or bracket) appears immediately:



```
MethodCall
```

```
collection
```

When the feature is performed, the text fields (an orange box) are added with the capability of a quick leaving the pawrens (brackets) after the *Enter* key is pressed. If you type something inside parens or brackets and press the *Enter* key to commit your change – the editor caret will move after the closing paren or bracket, allowing you to continue typing without using arrow keys to move the caret. Such behavior with the text fields will likely seem to be unexpected for the most users, but after you get accustomed to it, I'm sure you will love it.

Both of these features, apart from the automatic completion of parens (brackets), also include the following:

- easy deletion of empty parentheses (brackets)

If you don't need parens (e.g. mistyped it), you can press the Backspace key and both parens (or brackets) will be removed.

- smart semi-colon when typed inside parentheses (brackets)

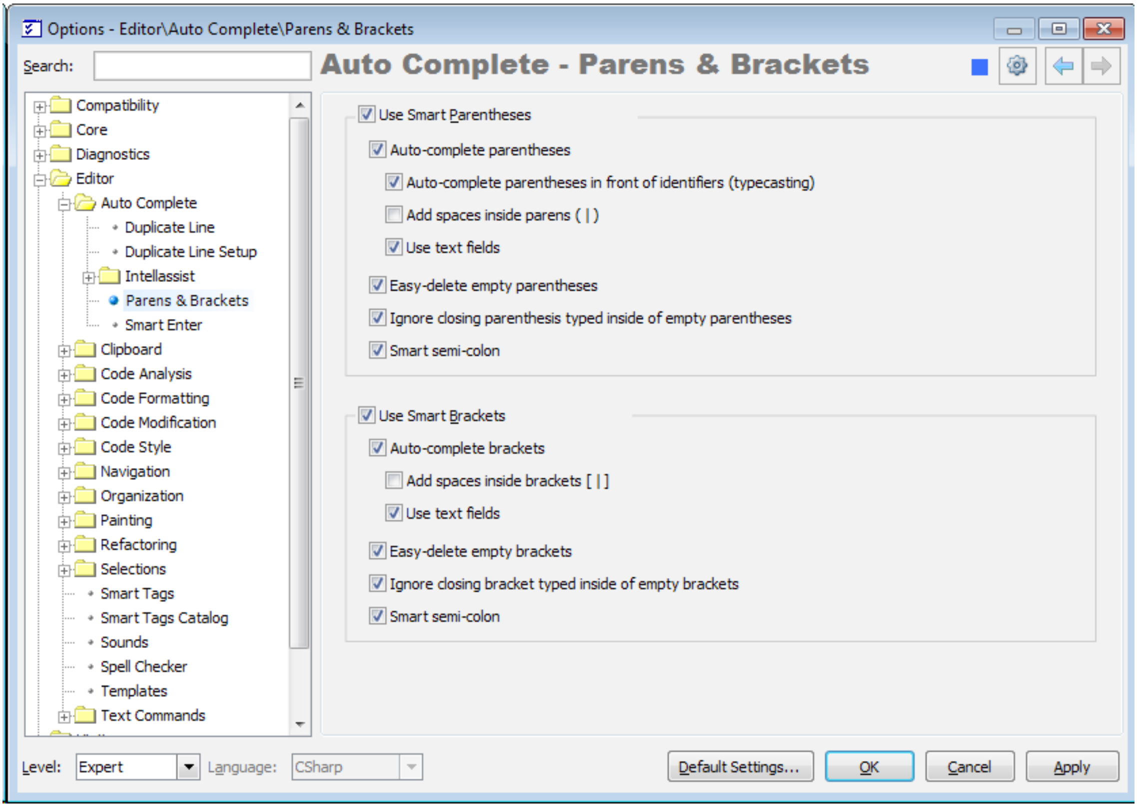
As an alternative to the *Enter* key to continue typing after the closing paren or bracket is inserted, you can type the semicolon ';', if appropriate. This is useful mostly in *C#* and *C++* languages, for code constructions which need to invoke a line terminator, e.g. method calls:

- ignoring closing paren (or bracket) if manually typed

Note, if you have a habit of typing the closing paren or bracket immediately after an opening paren or bracket, CodeRush will put only a single closing character instead of two (one automatically added and the other one –

manually typed), so you won't end with two closing parens or brackets.

Both **Smart Parentheses** and **Smart Brackets** features are fully customizable. Their settings are available on a single options page (*Editor* | *Auto Complete* | *Parens & Brackets*) in the Options Dialog:



Options - Editor\Auto Complete\Parens & Brackets

Auto Complete - Parens & Brackets

- ☑ Use Smart Parentheses
  - ☑ Auto-complete parentheses
    - ☑ Auto-complete parentheses in front of identifiers (typecasting)
    - □ Add spaces inside parens (|)
    - ☑ Use text fields
  - ☑ Easy-delete empty parentheses
  - ☑ Ignore closing parenthesis typed inside of empty parentheses
  - ☑ Smart semi-colon
- ☑ Use Smart Brackets
  - ☑ Auto-complete brackets
    - □ Add spaces inside brackets [|]
    - ☑ Use text fields
  - ☑ Easy-delete empty brackets
  - ☑ Ignore closing bracket typed inside of empty brackets
  - ☑ Smart semi-colon

Level: Expert

Language: CSharp

The settings are split into two groups separately for each feature. Available **Smart Parens** options are:

Option Name	Description
Use Smart Parentheses	Toggles the availability of the Smart Parentheses feature.
Auto-complete parentheses	Enables or disables parentheses auto completion (automatic insertion of the closing paren after the opening parent is typed).
Auto-complete parentheses in front of identifiers (typecasting)	Enables or disables parentheses auto completion when typed in front of an identifier (e.g. for a casting in C#).
Add spaces inside parens (|)	If this option is turned on, a space is added between parentheses after auto-complete parentheses is performed.
Use text fields	Enables or disables the text fields insertion after auto-complete parentheses is performed.
Easy-delete empty parentheses	Enables or disables the capability to easily delete empty parentheses with a single Backspace keystoke inside empty parentheses.
Ignore closing parenthesis typed inside empty parentheses	Enables or disables the capability to ignore the manually typed closing parenthesis after auto-complete parentheses is performed.

Smart semi-colon	Enables or disables the capability to type the semi-colon inside the parentheses (before the closing paren) and automatically move it to the appropriate location (after the closing paren).

The same options are available for **Smart Brackets** separately:

Option Name	Description
Use Smart Brackets	Toggles the availability of the Smart Brackets feature.
Auto-complete brackets	Enables or disables brackets auto completion (automatic insertion of the closing bracket after the opening bracket is typed).
Add spaces inside brackets [|]	If this option is turned on, a space is added between brackets after auto-complete brackets is performed.
Use text fields	Enables or disables the text fields insertion after auto-complete parentheses is performed.
Easy-delete empty brackets	Enables or disables the capability to easily delete empty brackets with a single Backspace keystoke inside empty brackets.
Ignore closing bracket typed inside empty brackets	Enables or disables the capability to ignore the manually typed closing bracket after auto-complete brackets is performed.
Smart semi-colon	Enables or disables the capability to type the semi-colon inside the brackets (before the closing bracket) and automatically move it to the appropriate location (after the closing bracket).

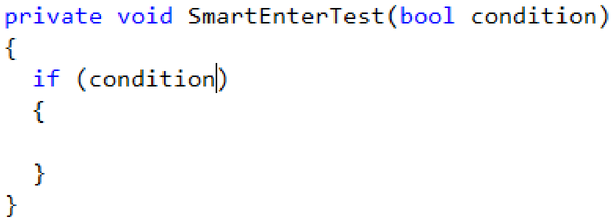
Smart Enter

Smart Enter is intended to improve the usability of the **Enter** key while coding in the editor. If the feature is enabled and the **Enter** key is pressed, it moves the editor text caret to the next code line (starting it with the same amount of white space on the current line and adding a line statement terminator to the end of the current line if it doesn't exist (.e.g semi-colon in *CSharp*) or inside of a code block, leaving the characters to the right of the source caret location at their initial position. It is activated if the character to the right of the current caret position is one of the following:

- **]** (closing bracket)
- **)** (closing paren)
- **>** (closing angle bracket)
- **;** (semi-colon)

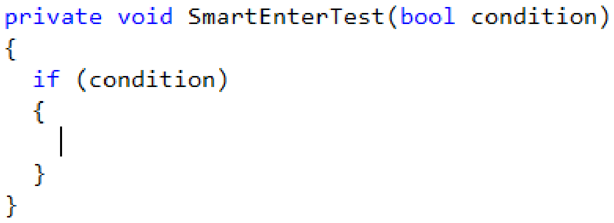
Here are the *CSharp* samples where **Smart Enter** feature may be useful:

1) Before **Enter** is pressed:



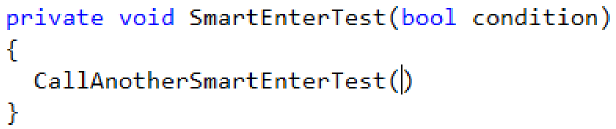
```
private void SmartEnterTest(bool condition)
{
  if (condition)
  {

  }
}
```

After **Enter** is pressed (notice the caret is moved inside of a code block):



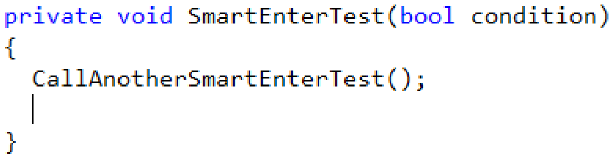
```
private void SmartEnterTest(bool condition)
{
  if (condition)
  {
    |
  }
}
```

2) Before **Enter** is pressed:



```
private void SmartEnterTest(bool condition)
{
  CallAnotherSmartEnterTest(|)
}
```

After **Enter** is pressed (notice the caret is moved to the next line, and a semi-colon is added to the end of the staring line):



```
private void SmartEnterTest(bool condition)
{
  CallAnotherSmartEnterTest();
  |
}
```

The feature doesn't depend on the current language used and should work everywhere when it is available. The most preferred language for this feature is *CSharp*, however. By default, it is disabled. To enable it, follow these steps to get to the **Smart Enter** options page:

1. From the DevExpress menu, select "***Options…***" to open the Options Dialog.

2. In the tree view on the left, navigate to this folder: "***Editor\Auto Complete***".

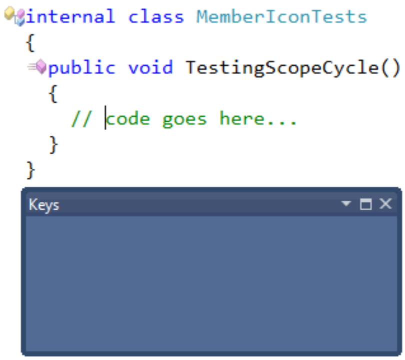
3. Select the "***Smart Enter***" options page.

4. Check the "***Enabled***" checkbox.

There are no other options on this options page.

Scope Cycle

The **Scope** (**Visibility**) **Cycle** feature adds the capability to quickly change the visibility modifier of the active member using a single shortcut. In a source code file, if your text caret is anywhere within or on a member (method, function, property, class, etc) definition, you can use the **Alt** key with the **Up** and**Down** arrow keys, to cycle through the legal visibility modifiers:

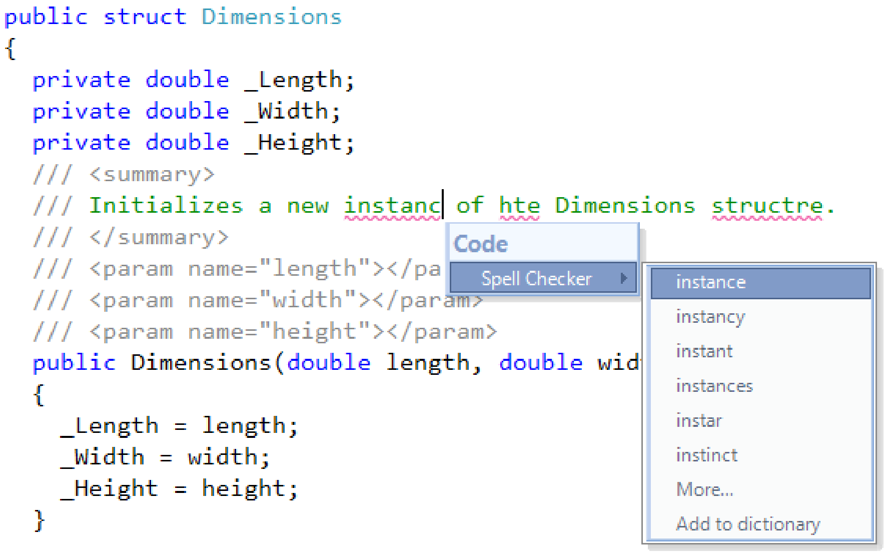


```
internal class MemberIconTests
{
  public void TestingScopeCycle()
  {
    // code goes here...
  }
}
```

You can also use another feature to change a member visibility modifier by clicking the Member Icons using the mouse.

Spell Checker

Spell Checker underlines the misspelled words inside the code editor. It can check spelling in strings, comments, XML comments, and HTML elements against a built-in dictionary. **Spell Checker** provides an easy way to fix the error using the Refactor! popup menu or a smart tag:



```
public struct Dimensions
{
  private double _Length;
  private double _Width;
  private double _Height;
  /// <summary>
  /// Initializes a new instanc of hte Dimensions structre.
  /// </summary>
  /// <param name="length"></pa
  /// <param name="width"></param>
  /// <param name="height"></param>
  public Dimensions(double length, double wid
  {
    _Length = length;
    _Width = width;
    _Height = height;
  }
```

Spell Checker is based on the XtraSpellChecker™ component from DevExpress. Thus, integrated intoVisual Studio **Spell Checker** has similar features, such as:

- Built-in support for ***OpenOffice*** dictionaries.

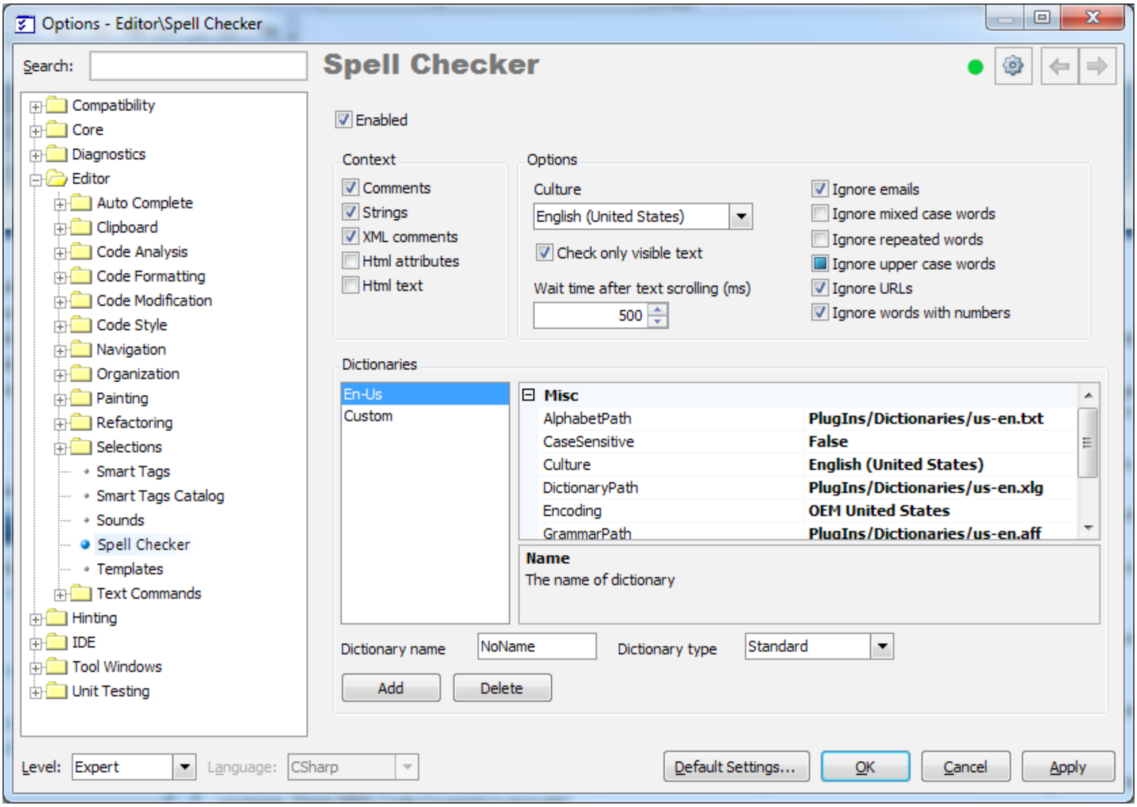
- ***Plain text*** format custom dictionary support.

- ***Automatic spell-checking*** as you type, word by word.

- Ability to ***add unknown words*** to the dictionary.

- ***Options*** allow you to ***ignore*** e-mails, URLs, mixed case/upper-case words, repeated words and words with numbers within them.

Here is the **Options Page** of the **Spell Checker**:



Spell Checker

☑ Enabled

Context

☑ Comments
☑ Strings
☑ XML comments
□ Html attributes
□ Html text

Options

Culture: English (United States)
☑ Check only visible text
Wait time after text scrolling (ms): 500

☑ Ignore emails
□ Ignore mixed case words
□ Ignore repeated words
■ Ignore upper case words
☑ Ignore URLs
☑ Ignore words with numbers

Dictionaries

En-Us
Custom

⊟ Misc	
AlphabetPath	PlugIns/Dictionaries/us-en.txt
CaseSensitive	False
Culture	English (United States)
DictionaryPath	PlugIns/Dictionaries/us-en.xlg
Encoding	OEM United States
GrammarPath	PlugIns/Dictionaries/us-en.aff

Name
The name of dictionary

Dictionary name: NoName
Dictionary type: Standard

Level: Expert
Language: CSharp

Here you can do the following:

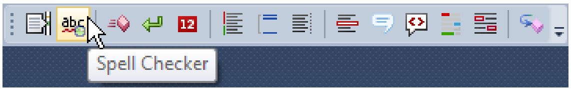
- Enable or disable the **Spell Checker**.

- Change the context **Spell Checker** is available for, e.g. "Strings", "Comments", "XML Comments", etc

- Change default options for the base **Spell Checker** component

- Manage dictionaries

To quickly turn **Spell Checker** on or off, use the appropriate icon on the DXCore Visualize toolbar:



To add a custom dictionary, refer to the following topic: How to add an Open Office dictionary for the Spell Checker.

How to add an Open Office dictionary for the Spell Checker

Follow these steps if you'd like to add a dictionary with a language other than *English* (*United States*) to the CodeRush Spell Checker:

1. Download the required spelling dictionary from the openoffice.org (e.g. *English* (*Canada*)).

2. Create a new folder (e.g., "En-CA") for your new dictionary. You can create it under the folder where default Spell Checker dictionaries are stored, for example:

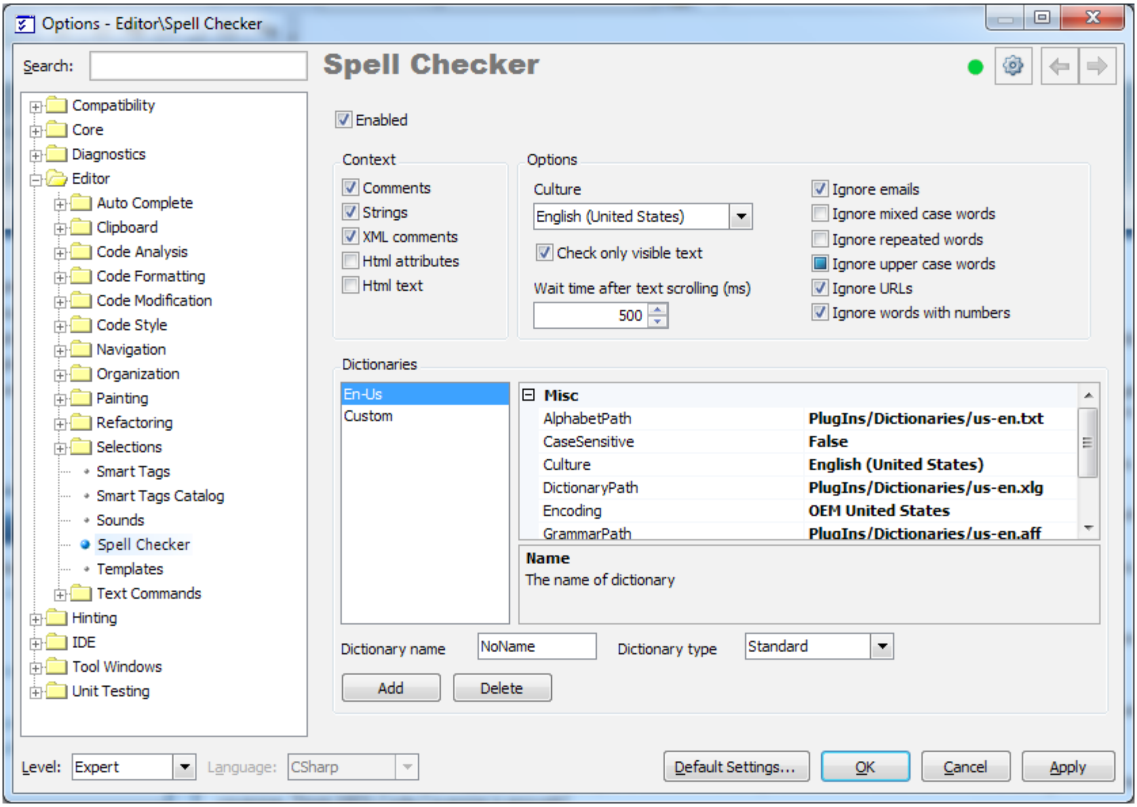
"*C:\Program Files\DevExpress 2010.1\IDETools\System\CodeRush\BIN\PLUGINS\Dictionaries\En-Ca*"

3. Extract the archive file (.zip) you downloaded in step #1 to the folder you created in step #2.

4. Inside Visual Studio, from the DevExpress menu, select "Options…" to open the Options Dialog.

5. In the tree view on the left, navigate to the "Editor" folder.

6. Select the "Spell Checker" options page:



☑ Enabled

Context

☑ Comments
☑ Strings
☑ XML comments
☐ Html attributes
☐ Html text

Options

Culture: English (United States)

☑ Check only visible text

Wait time after text scrolling (ms)
500

☑ Ignore emails
☐ Ignore mixed case words
☐ Ignore repeated words
▣ Ignore upper case words
☑ Ignore URLs
☑ Ignore words with numbers

Dictionaries

En-Us
Custom

Misc	
AlphabetPath	**PlugIns/Dictionaries/us-en.txt**
CaseSensitive	**False**
Culture	**English (United States)**
DictionaryPath	**PlugIns/Dictionaries/us-en.xlg**
Encoding	**OEM United States**
GrammarPath	**PlugIns/Dictionaries/us-en.aff**

Name
The name of dictionary

Dictionary name: NoName
Dictionary type: Standard

Level: Expert
Language: CSharp

7. At the bottom of the options page, enter the Dictionary name (e.g., "En-CA").

8. Choose the "*Open Office*" dictionary type in the corresponding combo box.

9. Click the Add button.

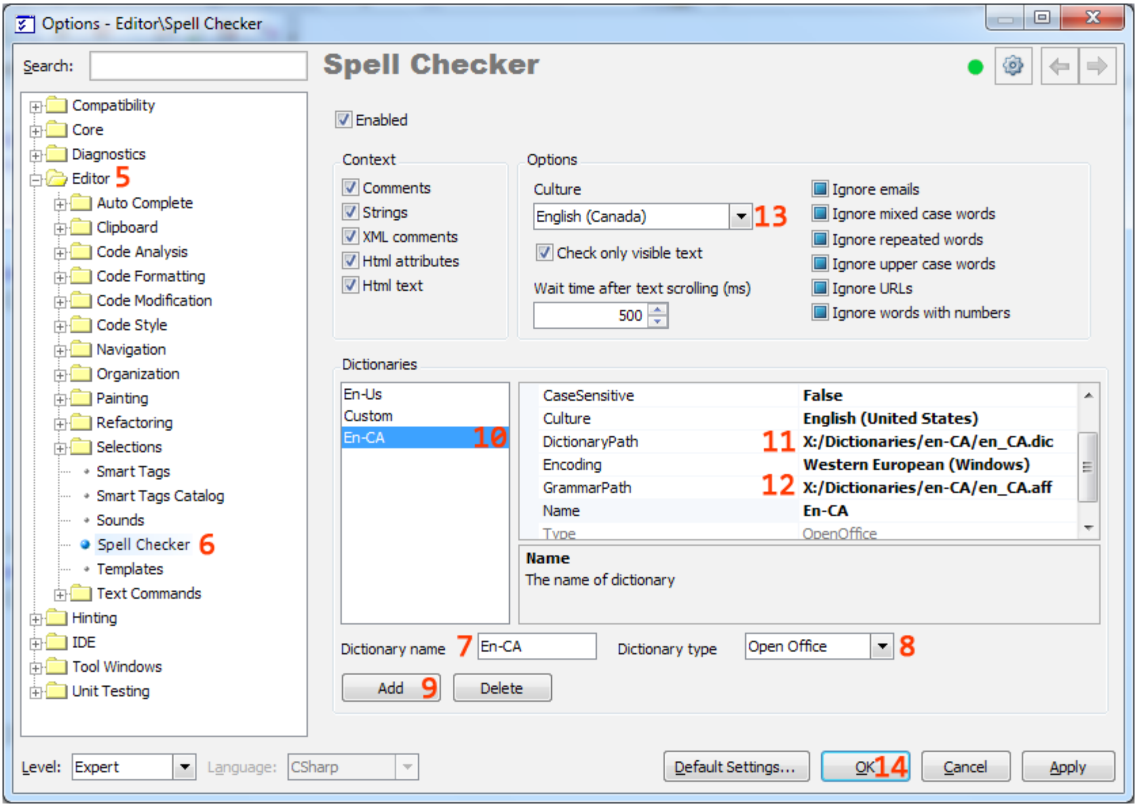
10. In the Dictionaries list, select the newly added dictionary.

11. Specify the path to the downloaded dictionary in the DictionaryPath field.

12. Specify the path to the grammar file (.aff) in the GrammarPath field.

13. Specify the Culture in the Options section on top of the options page.

14. Click OK to apply your new dictionary and close the Options Dialog.



Options - Editor\Spell Checker

Spell Checker

☑ Enabled

Context

☑ Comments
☑ Strings
☑ XML comments
☑ Html attributes
☑ Html text

Options

Culture: English (Canada) 13

☑ Check only visible text

Wait time after text scrolling (ms)
500

□ Ignore emails
□ Ignore mixed case words
□ Ignore repeated words
□ Ignore upper case words
□ Ignore URLs
□ Ignore words with numbers

Dictionaries

En-Us
Custom
En-CA 10

CaseSensitive	**False**
Culture	**English (United States)**
DictionaryPath 11	**X:/Dictionaries/en-CA/en_CA.dic**
Encoding	**Western European (Windows)**
GrammarPath 12	**X:/Dictionaries/en-CA/en_CA.aff**
Name	**En-CA**
Type	OpenOffice

Name
The name of dictionary

Dictionary name 7: En-CA

Dictionary type: Open Office 8

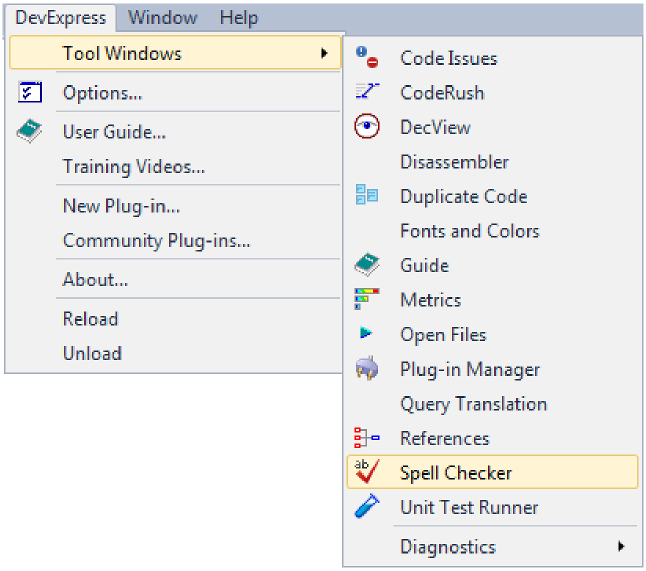
Level: Expert

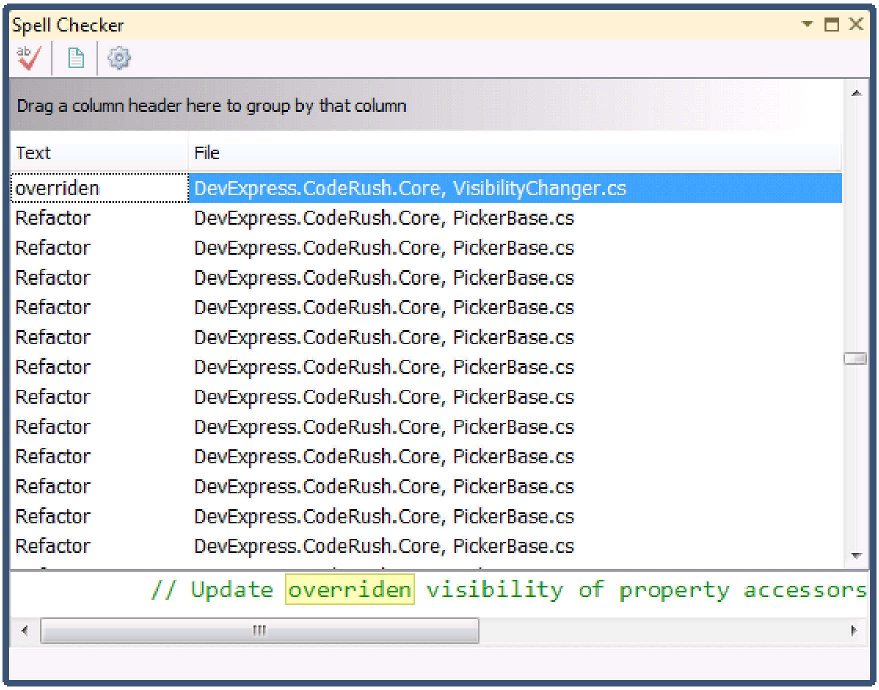
Language: CSharp

Now, the Spell Checker should use the new language dictionary to check spelling inside the code editor. You can change additional settings later, if required.

Spell checking the entire solution

In addition to the usual Spell Checker CodeRush feature, you can find and review spelling errors from the entire solution in the dedicated **Spell Checker** tool window:





Text	File
overriden	DevExpress.CodeRush.Core, VisibilityChanger.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs
Refactor	DevExpress.CodeRush.Core, PickerBase.cs

```
// Update overriden visibility of property accessors
```

Note that the **Spell Checker** feature must be enabled to perform a solution spell checking operation (enabled on the Spell Checker options page in the Options Dialog).

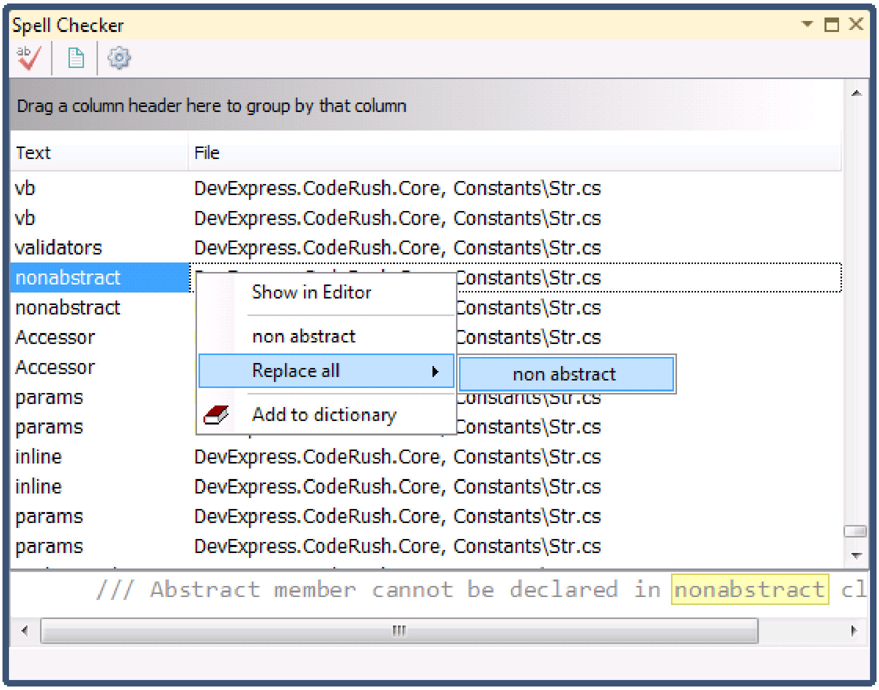
The tool window contains three toolbar items:

Button caption	Description
Spell check solution	Runs a spell check analysis for the entire solution

Edit custom dictionary	Opens a custom dictionary where you can specify custom words.
Options	Opens the Spell Checker options page in the Options Dialog.

The other part of the window shows a grid with the spelling errors: the text with an error and the file containing one. At the bottom of the tool window, you can see the preview code where the misspelled word was discovered.

Via the right click context menu, you are able to fix all similar misspelled words at once, fix a particular typo with the suggested correction, navigate to a misspelled word or add it to a dictionary:



Text	File
vb	DevExpress.CodeRush.Core, Constants\Str.cs
vb	DevExpress.CodeRush.Core, Constants\Str.cs
validators	DevExpress.CodeRush.Core, Constants\Str.cs
nonabstract	Constants\Str.cs
nonabstract	Constants\Str.cs
Accessor	Constants\Str.cs
Accessor	
params	Constants\Str.cs
params	Constants\Str.cs
inline	DevExpress.CodeRush.Core, Constants\Str.cs
inline	DevExpress.CodeRush.Core, Constants\Str.cs
params	DevExpress.CodeRush.Core, Constants\Str.cs
params	DevExpress.CodeRush.Core, Constants\Str.cs

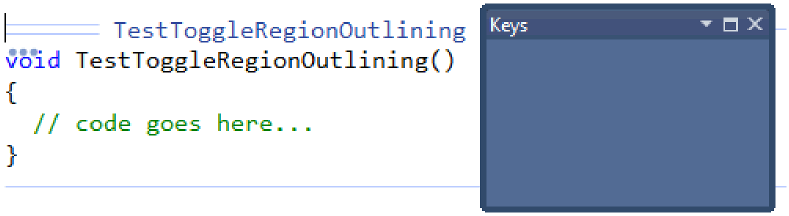
```
    /// Abstract member cannot be declared in nonabstract cl
```

Toggle Region

The **ToggleRegion** action changes the collapsed state of the currently active region. This functionality is very similar to the existing “Toggle Regions Outlining” shortcut, provided by CodeRush. The only difference is that you can collapse the active region using the **ToggleRegion** action when the editor caret is located anywhere inside of a region, while the “**Toggle Region Outlining**” shortcut is available only on the line with a region directive. The action is not bound to any key, so you might want to do it via the Shortcuts options page in the Options Dialog.

Toggle Regions Outlining

When the editor caret is on the line with a region directive, you can press the **Space** key to collapse or expand the outline of a region. There are built-in shortcuts for toggling outline expansion, but they are probably more tricky to press and remember, e.g. “**Ctrl+M**, **M**” (hold the **Control** key and press the **M**key twice in the Visual C# keyboard mapping scheme).

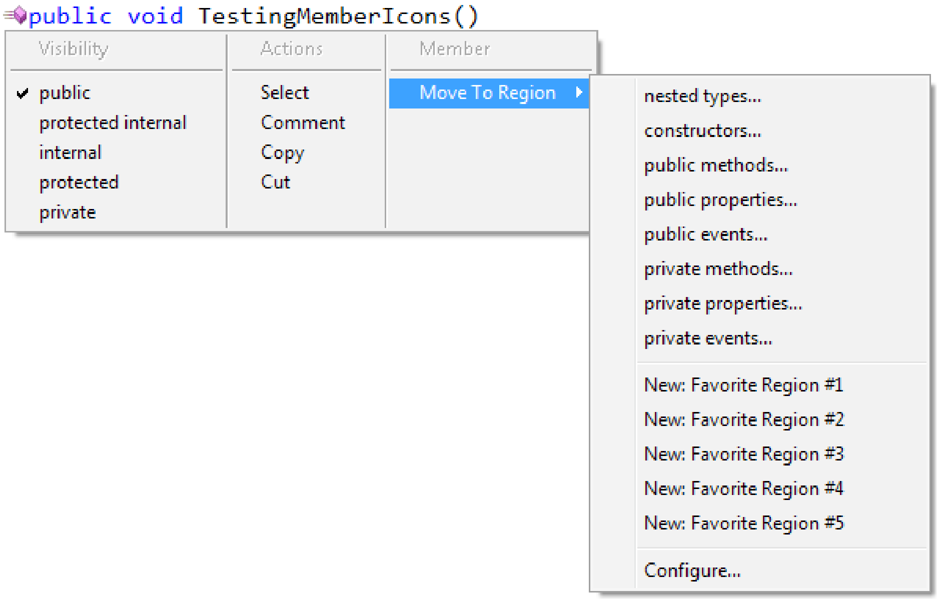


```
        TestToggleRegionOutlining
void TestToggleRegionOutlining()
{
  // code goes here...
}
```

This ability is provided by a single shortcut, registered on the Shortcuts options page in the Options Dialog, which just calls the “**Edit.ToggleOutliningExpansion**” Visual Studio command. There also exists the ToggleRegion action that provides the same functionality (but available anywhere inside of a member, not only on directive lines), but it is not bound to any key.

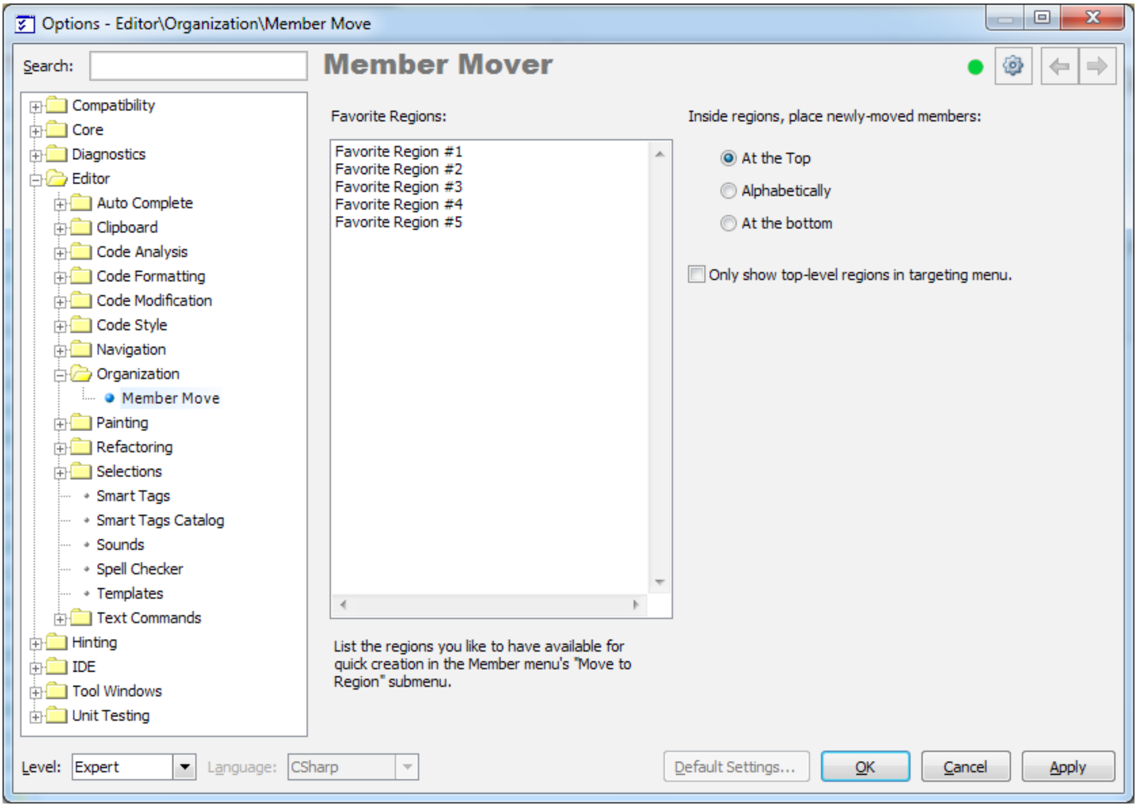
Move to Region

The **Move to Region** (also known as **Member Mover**) **CodeRush** feature extends the Member drop down menu with a menu item that allows you to move the active member to a region according to your preference:



```
public void TestingMemberIcons()
```

The list of regions contains regions that already exist with the current source code file. In addition, it also contains the list of favorite regions that you may specify on the **Move to Region** options page in the Options Dialog:



Options - Editor\Organization\Member Move

Search:

Member Mover

Favorite Regions:

Favorite Region #1
Favorite Region #2
Favorite Region #3
Favorite Region #4
Favorite Region #5

List the regions you like to have available for quick creation in the Member menu's "Move to Region" submenu.

Inside regions, place newly-moved members:

● At the Top
○ Alphabetically
○ At the bottom

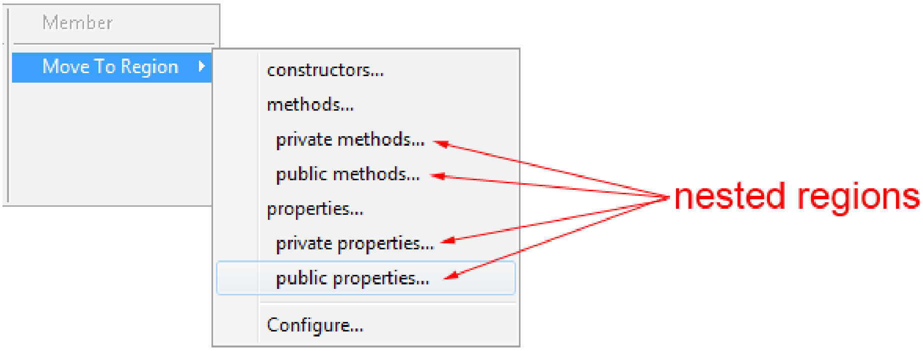
□ Only show top-level regions in targeting menu.

Level: Expert
Language: CSharp

This options page allows you to pre-configure an additional list of regions which will appear in the drop-down menu, regardless of whether or not they already exist in the source code file. Here, you can also specify the position of a member inside target regions:

- At the top
- Alphabetically
- At the bottom

The "*Only show top-level regions in targeting menu*" option deprecates nested regions showing inside of the **Member** drop-down menu. Nested regions are shown with a small offset to the right in the list:



The “**CR_MemberMover**” plug-in, containing this feature, is an open source sample plug-in from the CodeRush Shared Source solution. You are welcome to tweak it for your specific needs as you’d like.

Organizing source code and sorting class members

Reorganization of the code is a great practice to improve code readability. If you prefer to sort your type members like this:

```
class {
private fields
constructors
public properties
public methods
private methods
}
```

or any other order, you can use the **Member Organizer** CodeRush plug-in for this purpose.

The plug-in is an open source DXCore plug-in from the CodeRush Shared Source solution. It contains only a single action which is not bound to any key, by default. To assign a key, see the appropriate topic – the command name is the “*Organize Members*“.

Because the plug-in has the source code, you are welcome to tweak it in any way you’d like. By default, when you press the shortcut key bound to the “*Organize Members*” command, it will take the current type declaration, where the editor caret is located, and organize its members using the following structure:

- First, it sorts all members into groups by its visibility:
  - public
  - protected
  - protected internal
  - internal
  - private

Each one of these groups is automatically wrapped into a region (optional) with the corresponding name, for example:

#region Public members…
#endregion

or

#region Private members…
#endregion

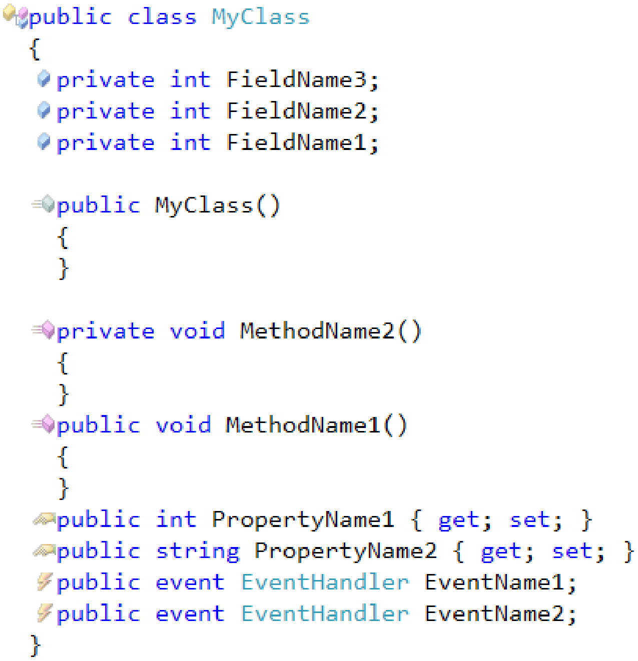
- Then, inside each visibility group, **Member Organizer**will sort them by member type in the following order:
  - methods
  - properties

- events
- fields

Members inside such groups are preceded with a corresponding comment (optional), such as:

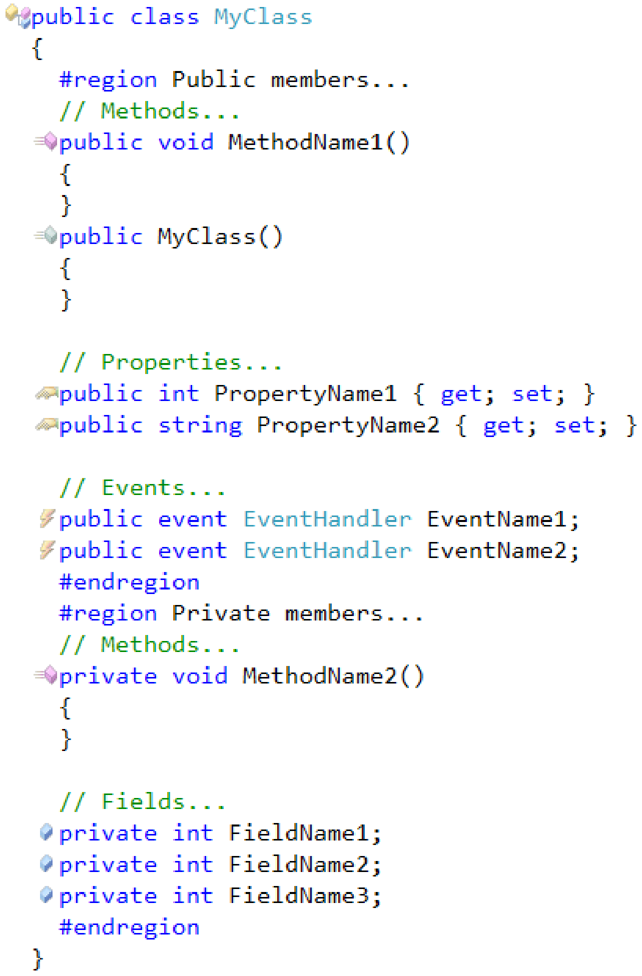
// Properties…
// Methods…

For example, having a code like this:



```
public class MyClass
{
  private int FieldName3;
  private int FieldName2;
  private int FieldName1;

  public MyClass()
  {
  }

  private void MethodName2()
  {
  }
  public void MethodName1()
  {
  }
  public int PropertyName1 { get; set; }
  public string PropertyName2 { get; set; }
  public event EventHandler EventName1;
  public event EventHandler EventName2;
}
```

Member Organizer with default settings will reorganize the code into this:



```
public class MyClass
{
  #region Public members...
  // Methods...
  public void MethodName1()
  {
  }
  public MyClass()
  {
  }

  // Properties...
  public int PropertyName1 { get; set; }
  public string PropertyName2 { get; set; }

  // Events...
  public event EventHandler EventName1;
  public event EventHandler EventName2;
  #endregion
  #region Private members...
  // Methods...
  private void MethodName2()
  {
  }

  // Fields...
  private int FieldName1;
  private int FieldName2;
  private int FieldName3;
  #endregion
}
```

To modify the order of members, you have to modify the source code of the plug-in and then recompile it. Open up the "*MemberOrgPlugIn.cs*" file, and go to the "*MemberOrgPlugIn*" class constructor. Here's a part of its code:

```
_Filters = new FilterCollection();
 AddMainVisibiltyGroup(MemberVisibility.Public);
 AddMainVisibiltyGroup(MemberVisibility.Protected);
 AddMainVisibiltyGroup(MemberVisibility.ProtectedInternal);
 AddMainVisibiltyGroup(MemberVisibility.Internal);
 AddMainVisibiltyGroup(MemberVisibility.Private);
```

The *AddMainVisibiltyGroup* method looks like this:

```
private void AddMainVisibiltyGroup(MemberVisibility visibility)
{
  VisibilityFilter newFilter = new VisibilityFilter(visibility);
  newFilter.Wrapper = WrapperMode.Region;
  AddSubFilter(newFilter, LanguageElementType.Method);
  AddSubFilter(newFilter, LanguageElementType.Property);
  AddSubFilter(newFilter, LanguageElementType.Event);
  AddSubFilter(newFilter, LanguageElementType.Variable);
  _Filters.Add(newFilter);
}
```

And, finally, the *AddSubFilter* method:

```
private void AddSubFilter(VisibilityFilter newFilter, LanguageElementType elementType)
{
  MemberTypeFilter subFilter = new MemberTypeFilter(elementType);
  subFilter.Wrapper = WrapperMode.Comment;
  newFilter.Add(subFilter);
}
```

As you can see, it is easy to modify the algorithm of the method sort. You can change positions of the members and specify the optional *Wrapper* mode, which can have three self-explained states:

- None
- Region
- Comment

The latest version of the plug-in is attached with a few minor fixes (such as comment names). Don’t forget to specify the correct build path in the Project options of the plug-in.

Chapter 6. Code selection and clipboard tools

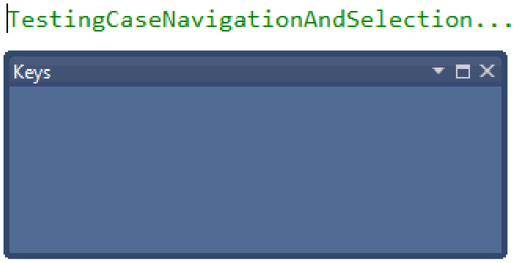
Camel Case Navigation and Selection

Camel Case Navigation (also known as **Camel Case Nav**) moves the code editor text caret to the next or previous lower case to the upper case transition of the current word. **Camel Case Nav** is built using DXCore action components, so it is activated via shortcuts.

The following shortcuts are available:

Shortcut	Action name	Description
Alt+Left	CamelCaseLeft	Moves the text caret to the next lower case to upper case transition, to the left of the text caret.
Alt+Right	CamelCaseRight	Moves the text caret to the next lower case to upper case transition, to the right of the caret.
Alt+Shift+Left	CamelCaseLeft	Extends the selection to the next lower case to upper case transition, to the left of the caret.
Alt+Shift+Right	CamelCaseRight	Extends the selection to the next lower case to upper case transition, to the right of the caret.
Alt+Backspace	CamelCaseDeleteLeft	Deletes the camel-case word to the left of the caret.
Alt+Delete	CamelCaseDeleteRight	Deletes the camel-case word to the right of the caret.

Sample:



```
TestingCaseNavigationAndSelection...
```

You can customize shortcuts within the IDE | Shortcuts options page in the Options Dialog.

Selection Increase

Selection Increase is a command for a quick selection of continuous logical code blocks. For example, if the caret is inside an expression, you can quickly expand the selection so it entirely holds the expression. The ability to quickly define a selection around a logical block is useful for refactoring. For example, if you'd like to extract a piece of code into a method or property – the **Selection Increase** will help you define your code block for extraction very quickly. You can also use it to quickly select code you want to move to another location or select any member and/or type declarations. Just place the caret at the beginning of the declaration you want to select, and press the shortcut. This command is bound to **NUM+** key or **Ctrl**+**W** key. After increasing a selection, you can reduce it using the Selection Reduce command by pressing the **NUM-** key.

Increasing the Selection Further

If you want to add more code to the start of the selection (by logical code blocks), you may use the **SelectionInclude-PreviousElement** action (which is not bound to any key by default). This will extend the active point of the selected

block to include the start of the preceding language element.
If you want to add more code to the end of the selection (by logical code blocks), you may use the **SelectionIncludeNextElement** action (which is not bound to any key by default). This will extend the active point of the selected block to include the end of the next language element.

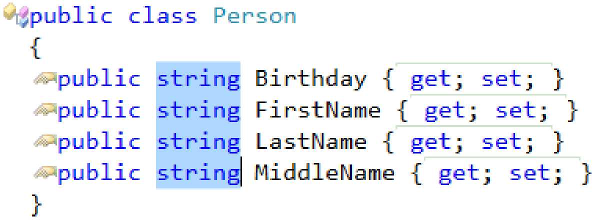
Note: If the caret is inside a white space (e.g., at the absolute beginning of a line), the parenting block that is first selected may be greater than you expect (for example, the first selection might be a method or an entire class rather than a block that starts later on the same line. For best results, place the caret somewhere on a text of the code block you want to select.

Selection Reduce

This command reduces the selection using the same logical blocks by which it was last increased via the Selection Increase feature. If you accidentally overshoot the block you want to select, you can reduce the selection using **NUM-** key or **CTRL**+**Shift**+**W** keys.

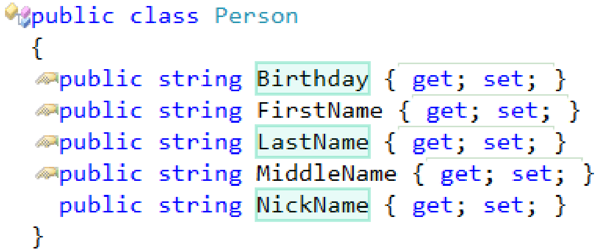
The support of multi-selection for Visual Studio

The Visual Studio IDE has a great box selection feature that allows you to select a rectangular region of text within the code editor by holding down the Alt key while selecting the text region with the mouse:



```
public class Person
 {
  public string Birthday { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string MiddleName { get; set; }
 }
```

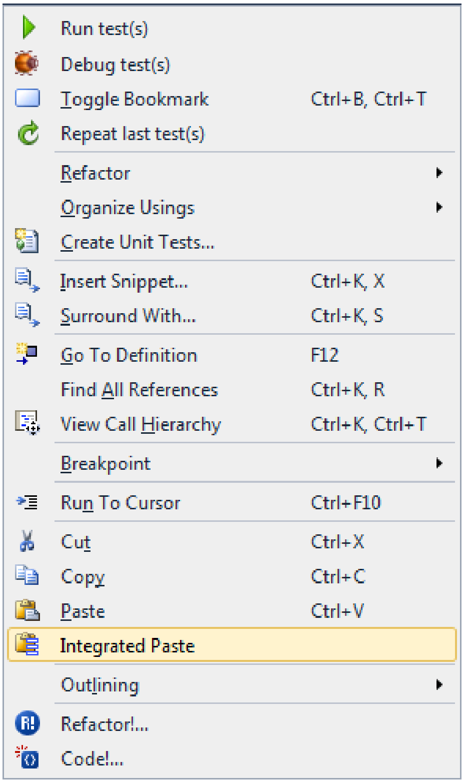
However, the selected region should represent a single continuous selection box. In other words, you can not select multiple parts of the source code. But with the help of the CodeRush **Multi-Select** feature you can do that easily. This feature allows you to create multiple selections in the source code or a typical text in the active text document as follows:



```
public class Person
 {
  public string Birthday { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string MiddleName { get; set; }
  public string NickName { get; set; }
 }
```

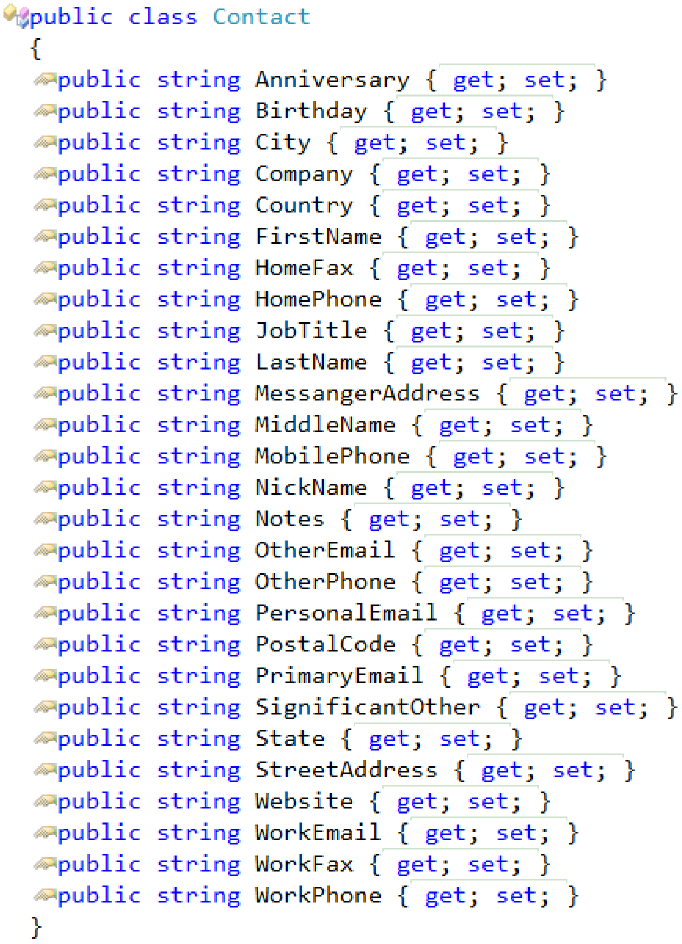
You can cut, copy, paste or simply delete the multi-selected text in a single action via default shortcuts (Ctrl+C, Ctrl+X, Ctrl+V). Text selections are contained within the Multi-Select stack. You can have any number of stacks for each opened text document in the IDE.

Furthermore, the Multi-Select feature provides an additional action called **Integrated Paste**. It becomes available in the right-click code editor context menu if you have copied several members of a class into Clipboard:



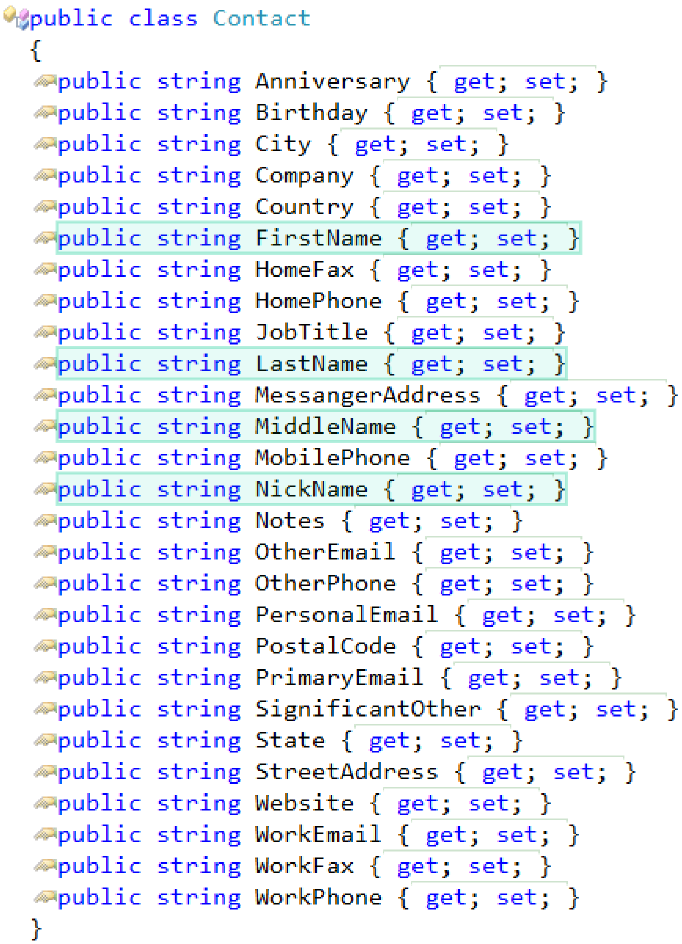
The integrated paste inserts copied members in a specific order: methods to method, properties to properties, fields to fields etc. In other words, when you paste fields and properties, the copied fields will be moved near existing fields and new properties will be pasted right over other properties of the destination class. This feature might be very useful to rearrange members of a class when moving them to a different class.

The main purpose of the Multi-Select feature is to simplify source code edits. Consider the following code:



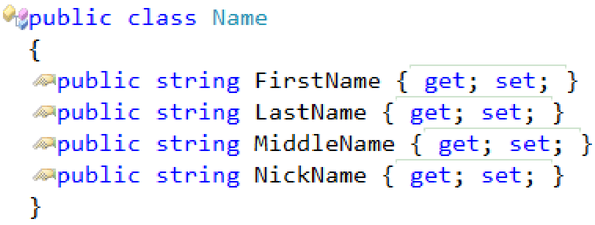
```
public class Contact
{
    public string Anniversary { get; set; }
    public string Birthday { get; set; }
    public string City { get; set; }
    public string Company { get; set; }
    public string Country { get; set; }
    public string FirstName { get; set; }
    public string HomeFax { get; set; }
    public string HomePhone { get; set; }
    public string JobTitle { get; set; }
    public string LastName { get; set; }
    public string MessangerAddress { get; set; }
    public string MiddleName { get; set; }
    public string MobilePhone { get; set; }
    public string NickName { get; set; }
    public string Notes { get; set; }
    public string OtherEmail { get; set; }
    public string OtherPhone { get; set; }
    public string PersonalEmail { get; set; }
    public string PostalCode { get; set; }
    public string PrimaryEmail { get; set; }
    public string SignificantOther { get; set; }
    public string State { get; set; }
    public string StreetAddress { get; set; }
    public string Website { get; set; }
    public string WorkEmail { get; set; }
    public string WorkFax { get; set; }
    public string WorkPhone { get; set; }
}
```

This class contains many properties. We can refactor this code by creating separate classes for the specific data. Let’s select the required properties and cut them:



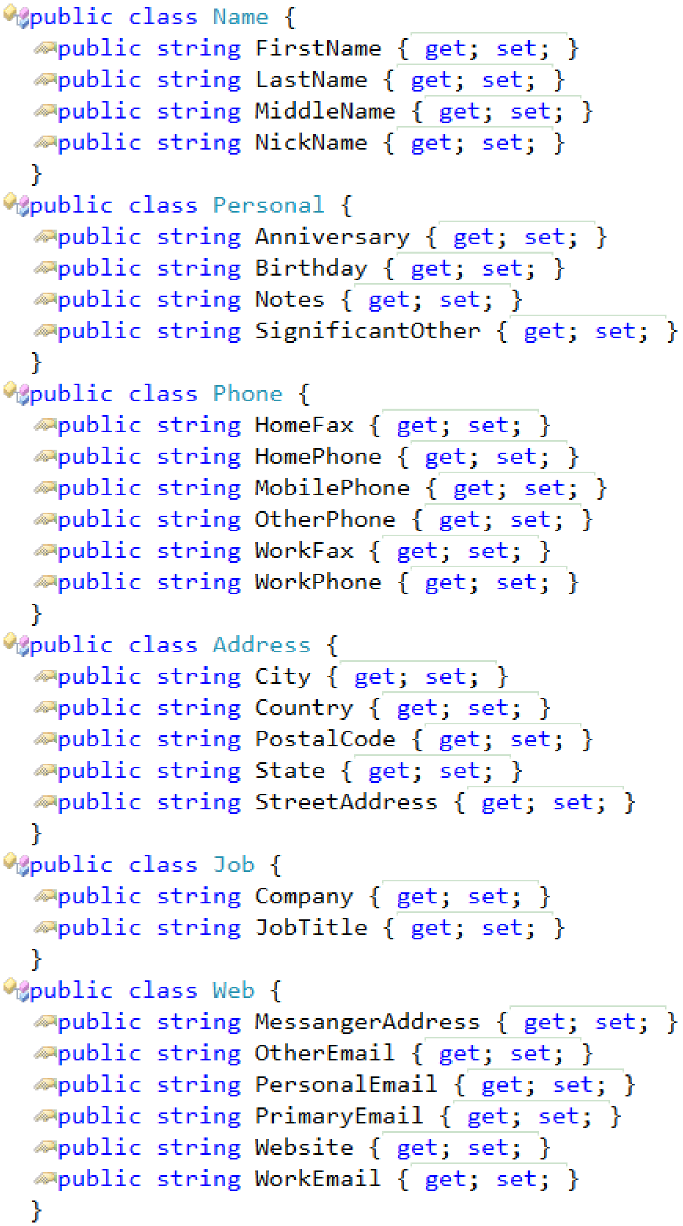
```
public class Contact
 {
  public string Anniversary { get; set; }
  public string Birthday { get; set; }
  public string City { get; set; }
  public string Company { get; set; }
  public string Country { get; set; }
  public string FirstName { get; set; }
  public string HomeFax { get; set; }
  public string HomePhone { get; set; }
  public string JobTitle { get; set; }
  public string LastName { get; set; }
  public string MessangerAddress { get; set; }
  public string MiddleName { get; set; }
  public string MobilePhone { get; set; }
  public string NickName { get; set; }
  public string Notes { get; set; }
  public string OtherEmail { get; set; }
  public string OtherPhone { get; set; }
  public string PersonalEmail { get; set; }
  public string PostalCode { get; set; }
  public string PrimaryEmail { get; set; }
  public string SignificantOther { get; set; }
  public string State { get; set; }
  public string StreetAddress { get; set; }
  public string Website { get; set; }
  public string WorkEmail { get; set; }
  public string WorkFax { get; set; }
  public string WorkPhone { get; set; }
 }
```

then, paste them into a separate class:



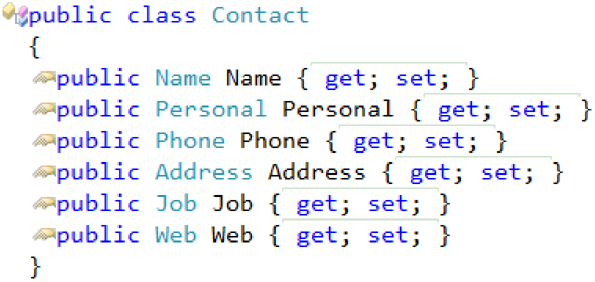
```
public class Name
 {
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string MiddleName { get; set; }
  public string NickName { get; set; }
 }
```

And apply the same technique to other data, so we end up with this:



```
public class Name {
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string MiddleName { get; set; }
   public string NickName { get; set; }
}
public class Personal {
   public string Anniversary { get; set; }
   public string Birthday { get; set; }
   public string Notes { get; set; }
   public string SignificantOther { get; set; }
}
public class Phone {
   public string HomeFax { get; set; }
   public string HomePhone { get; set; }
   public string MobilePhone { get; set; }
   public string OtherPhone { get; set; }
   public string WorkFax { get; set; }
   public string WorkPhone { get; set; }
}
public class Address {
   public string City { get; set; }
   public string Country { get; set; }
   public string PostalCode { get; set; }
   public string State { get; set; }
   public string StreetAddress { get; set; }
}
public class Job {
   public string Company { get; set; }
   public string JobTitle { get; set; }
}
public class Web {
   public string MessangerAddress { get; set; }
   public string OtherEmail { get; set; }
   public string PersonalEmail { get; set; }
   public string PrimaryEmail { get; set; }
   public string Website { get; set; }
   public string WorkEmail { get; set; }
}
```

The original code will look like this:



```
public class Contact
{
  public Name Name { get; set; }
  public Personal Personal { get; set; }
  public Phone Phone { get; set; }
  public Address Address { get; set; }
  public Job Job { get; set; }
  public Web Web { get; set; }
}
```

This code is now much easier to read and understand.

The following Multi-Select shortcuts are available by default:

Shortcut	Action	Description
Ctrl+Alt+Enter	MultiSelectAdd	Adds the current selection into the Multi-Select stack. If there is no selection, the current language element will be automatically selected.
Ctrl+Alt+Backspace	MultiSelectClear	Clears the current stack.
Ctrl+Alt+Num-	MultiSelectUndo	Removes the previous added selection from the stack.
Ctrl+Alt+Num+	MultiSelectRedo	The opposite of the MultiSelectUndo to restore the removed selection.

Note that if you don't have an active text selection, the Multi-Select feature will automatically expand the selection the same way as you would execute the Selection Increase command.

The **Multi-Select** feature is available out of the box. It was created as an open source plug-in in two CodeRush Feature Workshop webinars**.** You can see how it was built from scratch and learn more on how to create CodeRush plug-ins.

The feature has a single option page (Editor | Selections | MultiSelect) in the Options Dialog that allows you to tweak the default highlighting color of a multi-selection piece.

You can also extend this feature programmatically via the DXCore MultiSelect service.

Wrapping code blocks with the Code Embeddings feature

CodeRush **Code Embeddings** allow you to wrap the selected code block or any text into another predefined code block, such as: try/catch, try/finally, using and lock statements, while and other loops, region directives, etc. **Code Embeddings** are available via the predefined keyboard shortcuts, via the *Embed Selection* code editor context menu item, or using the Embed Selection code provider.

Embeddings are not context sensitive. This means that you can wrap any text into any code block, because the feature does not analyze selected code – you only add text before and/or after the selected text. However, depending on the selection type (e.g. whole line, multi-line, line fragment), different embeddings are available. So, it is up to you to decide whether to apply a specific code embedding or not.

The feature can also change the resulting editor caret position, drop markers, paste clipboard contents into the specified location and much more. This means that you have a full control over **Code Embeddings**, i.e. you are able to add string providers and text commands to participate in embeddings. Embeddings are customized on the same named Embeddings options page in the Options Dialog, where you can modify existing items or create custom ones.

Consider the following code:

```
string connectionString = Application.Properties.Settings.Default.ConnectionString;

SqlConnection connection = new SqlConnection(connectionString);

connection.Open();

SqlCommand command = new SqlCommand(«SELECT TOP 2 * FROM Main», connection);

SqlDataReader reader = command.ExecuteReader();

reader.Read();

Console.WriteLine(«{0} {1} {2}»,

reader.GetInt32(0), reader.GetString(1), reader.GetString(2));
```

This code creates a new SQL connection, executes an SQL command, reads its result and prints them on the Console. Using the **Code Embeddings** feature we can enhance the code to the following:

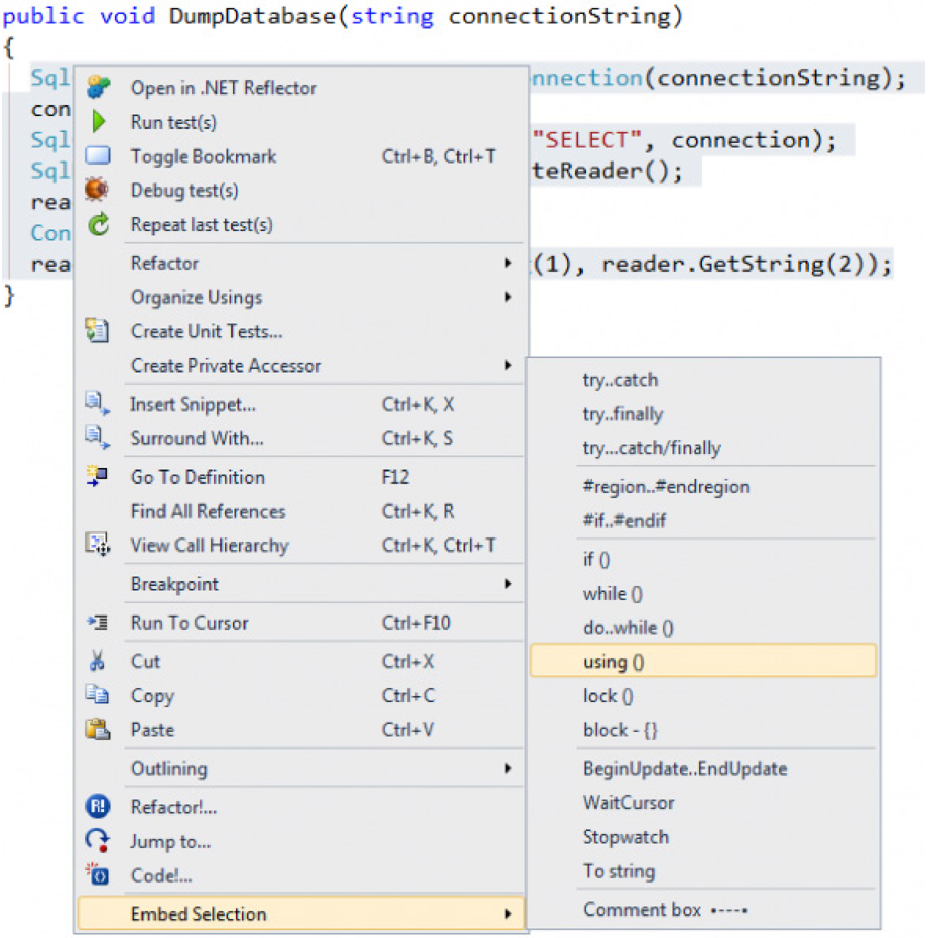
```
string connectionString = Application.Properties.Settings.Default.ConnectionString;

using (SqlConnection connection = new SqlConnection(connectionString))

{

  connection.Open();

  using (SqlCommand command = new SqlCommand("SELECT TOP 2 * FROM Main", connection))

  {

    using (SqlDataReader reader = command.ExecuteReader())

    {

      while (reader.Read())

      {

        Console.WriteLine(«{٢} {١} {٠}», reader.GetInt٠)٣٢), reader.GetString(١), reader.GetString(٢));

      }

    }

  }

}
```

Which is much more appropriate for using an SQL database access APIs.

To apply an embedding use one of the following ways:

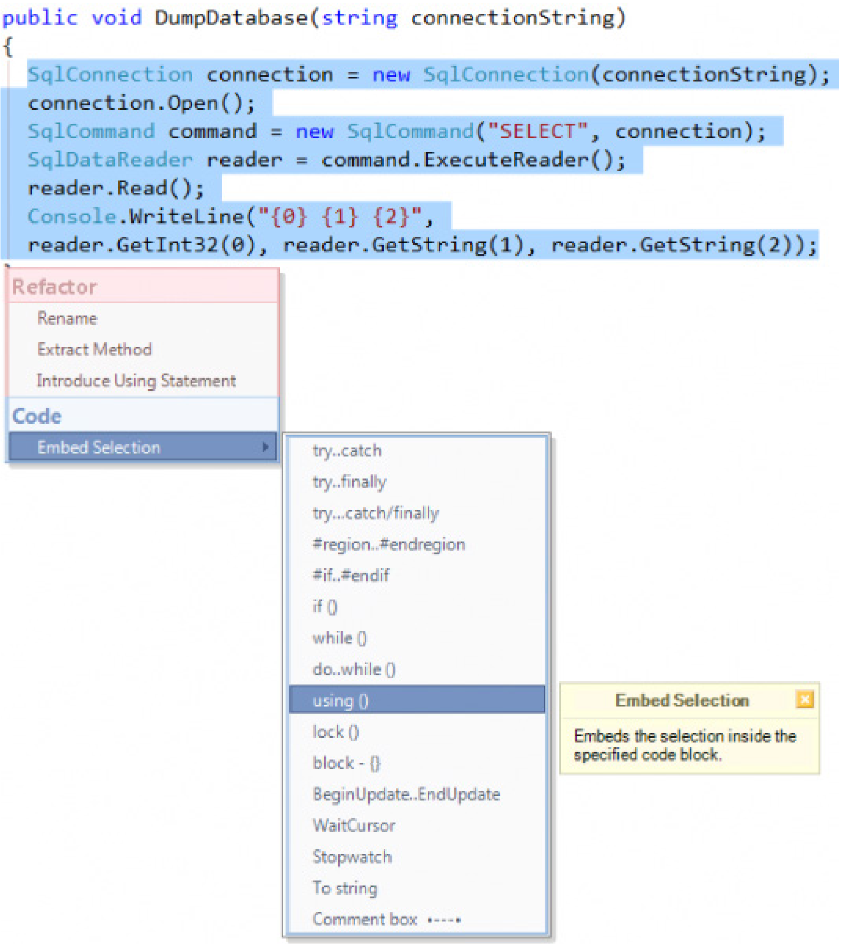
- Press the shortcut key. You can see all available shortcuts and create your own on the Shortcuts options page. To create a new shortcut, use the ***Embed*** action name.

- Select the appropriate text and right click the code editor. Then choose the **Embed Selection** menu item:



```
public void DumpDatabase(string connectionString)
{
  Sql                                                    nnection(connectionString);
  con
  Sql                                                    "SELECT", connection);
  Sql                                                    teReader();
  rea
  Con
  rea                                                    (1), reader.GetString(2));
}
```

- Select the appropriate text and perform the Embed Selection code provider:



```
public void DumpDatabase(string connectionString)
{
    SqlConnection connection = new SqlConnection(connectionString);
    connection.Open();
    SqlCommand command = new SqlCommand("SELECT", connection);
    SqlDataReader reader = command.ExecuteReader();
    reader.Read();
    Console.WriteLine("{0} {1} {2}",
    reader.GetInt32(0), reader.GetString(1), reader.GetString(2));
```

Embeds the selection inside the specified code block.

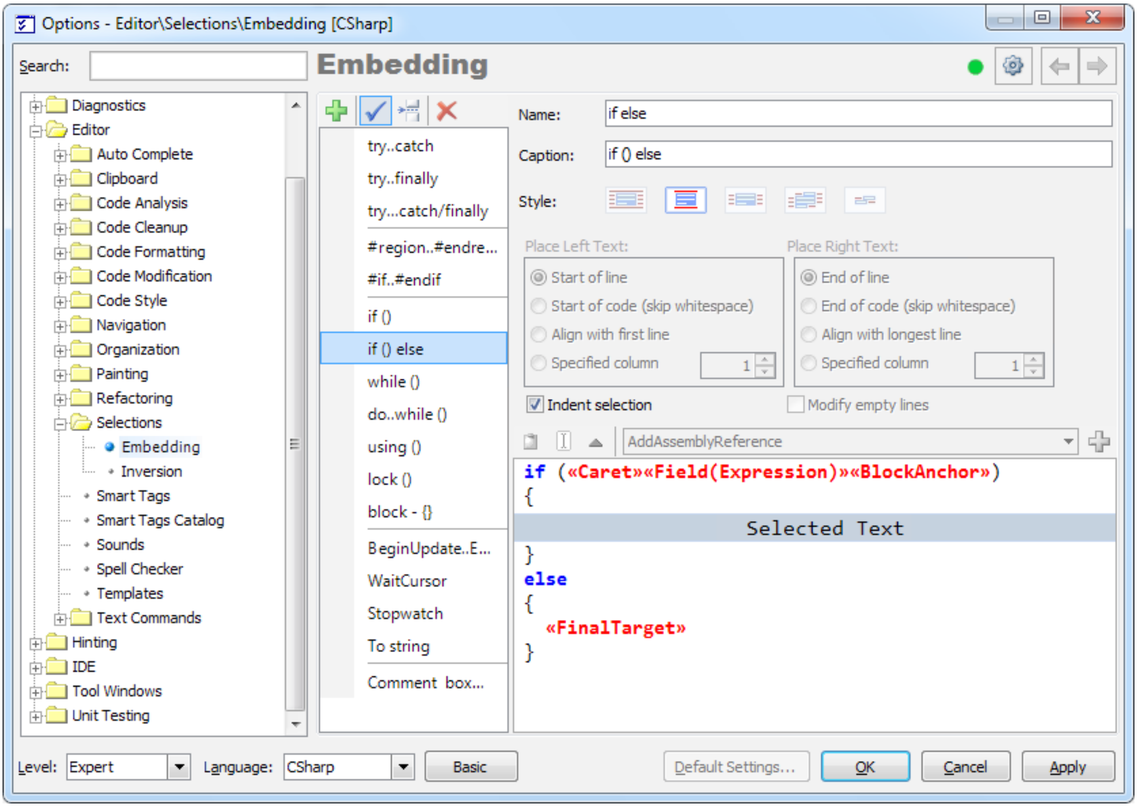
Here’s the list of available code embeddings for different languages.

CSharp:

Embedding	Shortcut	Description
try..catch	C	Embeds the selection into the *try{} catch{}* code block.
try..finally	F	Embeds the selection into the *try{} finally{}* code block.
try..catch/finally	T	Embeds the selection into the *try{} catch{} finally{}* code block.
#region.. #endregion	R, Ctrl+3	Embeds the selection between the *#region* and *#endregion* directives.
#if..#endif		Embeds the selection between the *#if* and *#endif* directives.
if ()		Embeds the selection into the *if(){}* code block.
if () else		Embeds the selection into the *if(){} else{}* code block.
while()		Embeds the selection into the *while(){}* code block.
do..while()		Embeds the selection into the *do{} while()* code block.
using()	U	Embeds the selection into the *using(){}* code block.
lock()	L	Embeds the selection into the *lock(){}* code block.
block – {}	B, Shift+[	Encloses the selection in *{}* braces.
BeginUpdate.. EndUpdate		Embeds the selection between the *BeginUpdate* and *EndUpdate* method calls of an object from the clipboard.
WaitCursor	W	Embeds the selection into the *try{} finally{}* block, provided that the *WaitCursor* will be used during the selected code lines' execution.
Stopwatch		Creates a new instance of the *System. Diagnostics. Stopwatch* class, embeds the selection into the *try{} finally{}* block between the *Stopwatch. Start()* and *Stopwatch. Stop()* method calls, and outputs the *Stopwatch. Elapsed* value to the console.
To string	Shift+'	Quotes the selected code section.
Comment box		Embeds the selection into the comment box.
Parentheses	Shift+0	Embeds the selection into the parentheses.
Not Parentheses	Shift+1	Embeds the selection into the parentheses and places the logical Not (!) operator before the parentheses.
Brackets	[	Embeds the selection into the brackets.

Code Embeddings Options

The **Embeddings** options page allows you to customize or remove the predefined CodeRush Code Embeddings and create your own for any registered language. The options page is located at the *Editor* | *Selections* | *Embeddings* path inside the CodeRush Options Dialog. Here is what is looks like:



Name: if else

Caption: if () else

Place Left Text:
● Start of line
○ Start of code (skip whitespace)
○ Align with first line
○ Specified column 1

Place Right Text:
● End of line
○ End of code (skip whitespace)
○ Align with longest line
○ Specified column 1

☑ Indent selection

□ Modify empty lines

```
if («Caret»«Field(Expression)»«BlockAnchor»)
{
                    Selected Text
}
else
{
  «FinalTarget»
}
```

Level: Expert

Language: CSharp

It consists of two parts:

1. The list of predefined code embeddings on the left with a toolbar at the top

2. Options for the selected code embedding on the right

In the list, you can see embeddings defined for the selected language. Items in the list can be grouped together by using a separator. Embeddings will also be grouped the same way in the code editor context menu, for example.

There are four buttons on a toolbar at the top of the list:

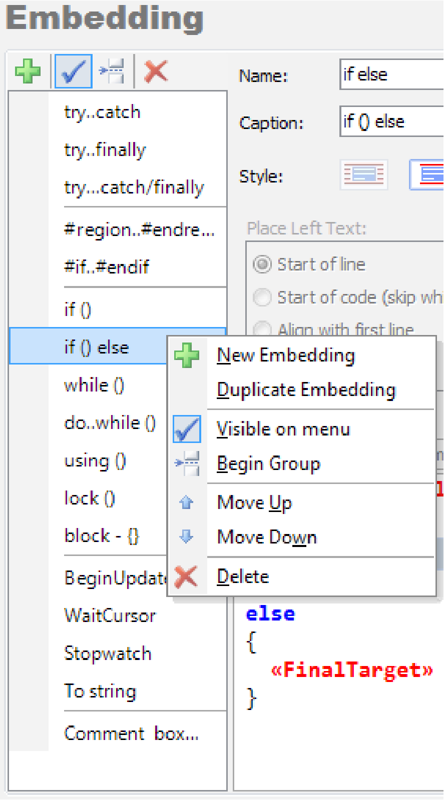
1. *New Embedding* – adds a new embedding item over the currently selected embedding in the list

2. *Visible on menu* – toggles the visibility of the embedding item in the editor context menu or the Embed Selection code provider. You can still perform hidden embedding using the keyboard shortcuts.

3. *Begin group* – begins grouping of the selected embedding item in the list. A group separator is inserted above the selected embedding item.

4. *Delete* – removes the embedding item from the list

The list of available code embeddings also has a context menu:



Name: if else

Caption: if () else

Style:

Place Left Text:

◉ Start of line

○ Start of code (skip wh

○ Align with first line

```
else
{
   «FinalTarget»
}
```

It has the same items as the toolbar, along with a few additional items:

- Duplicate Embedding – creates a copy of the currently selected embedding item in the list
- Move Up – moves the selected embedding item one position up in the list and user UI
- Move Down- moves the selected embedding item one position down in the list and user UI

Now, the embedding item parameters are on the right of the options page. You can change the following settings for the selected code embedding:

- *Name* – the name of the code embedding is used to access this specific embedding, e.g. when you create a new shortcut for it.
- *Caption* – the display name of the code embedding is used in the UI, such as: the editor context menu and the Embed Selection code provider.
- Style – the style of the code embeddings. The following styles are available:

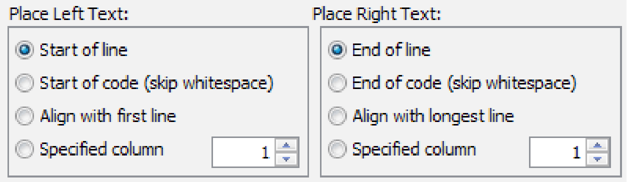
Icon	Description
▤	Surround the selected text. The text is placed at the top, bottom, and on the left and on the right of the selection.
▤	Top and/or bottom of the selected text. The text is placed above and below the selection.

Left and/or right of the selected text. The text is placed to the left and to the right of the selection (each line of the selection gains a prefix and a suffix).

Unique text is inserted at the start and end of the selected text, with repeating text added to the left and right to each line in the middle of the selection.

The text is placed in front of and immediately after the selection.

After the style settings, there are settings for the left and right text positions:



Place Left Text:

- ● Start of line
- ○ Start of code (skip whitespace)
- ○ Align with first line
- ○ Specified column 1

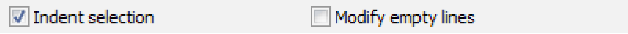
Place Right Text:

- ● End of line
- ○ End of code (skip whitespace)
- ○ Align with longest line
- ○ Specified column 1

Available settings are:

- *Start of line* – place the text at the start of a line
- *Start of code (skip whitespace)* – place the text right before another text (code) skipping the whitespace characters
- *Align with first line* – align the text with the text of the first selected line
- *Specified column [X]* – place the text at the specified column (starting from ١)

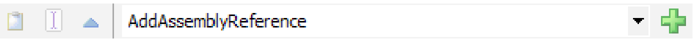
Next, there are two checkboxes:



☑ Indent selection □ Modify empty lines

- *Indent Selection* – specifies whether an indent is added before the embedded text
- *Modify empty lines* – specifies whether empty lines, as well as lines containing a text, are modified. If unchecked, empty lines are ignored and not wrapped at all.

The code embedding itself is edited in the corresponding area at the bottom of the right part of the options page. You can see the Selected Text bar and type the text above, below, on the left, or on the right of the bar, depending on the selected *Style* option. This area has a toolbar with several helper buttons and a combobox that allows you to choose and insert another text command:



AddAssemblyReference

The buttons on the toolbar insert widely-used text commands:

- *Paste from clipboard* – inserts the Paste text command into the embedding editing area that will paste the contents of the Clipboard when this code embedding is applied
- *Place caret* – inserts the Caret text command into the embedding editing area that moves the caret to the specified position when this code embedding is applied
- *Drop marker* – inserts the Marker text command into the embedding editing area that drops a blue marker at the specified location when this code embedding is applied

The combobox and the *Plus [+]* button to the right allow you to choose any other registered text command that will participate in the code embedding expansion once it is applied.

To learn more on how to add your own code embeddings, please read the appropriate topic.

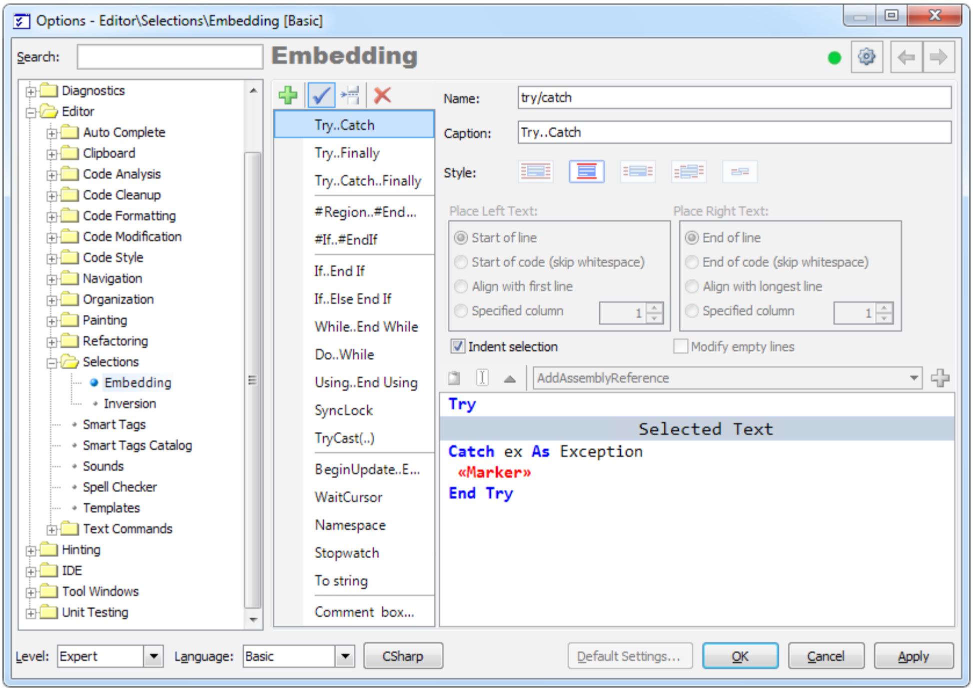
Creating custom Code Embeddings

CodeRush Code Embeddings are completely customizable. They are modified on the Embeddings options page in the Options Dialog, where you can create your own. Let’s add a new code embedding to surround the selected code with the logging capabilities.

We will use the DXCore *Log* class located in the *DevExpress.CodeRush.Diagnostics.General* namespace (in the *DevExpress.CodeRush.Common* assembly). The embedded code will look like this (C#):

```
Log.Enter(“Message”);
try
{
  // selected text goes here...
}
catch (Exception ex)
{
  Log.SendException(«An exception occurred», ex);
}
finally
{
  Log.Exit(«Message»);
}
```

1) Open up the Embedding options page:



Options - Editor\Selections\Embedding [Basic]

Embedding

Name: try/catch

Caption: Try..Catch

Place Left Text: ● Start of line

Place Right Text: ● End of line

☑ Indent selection

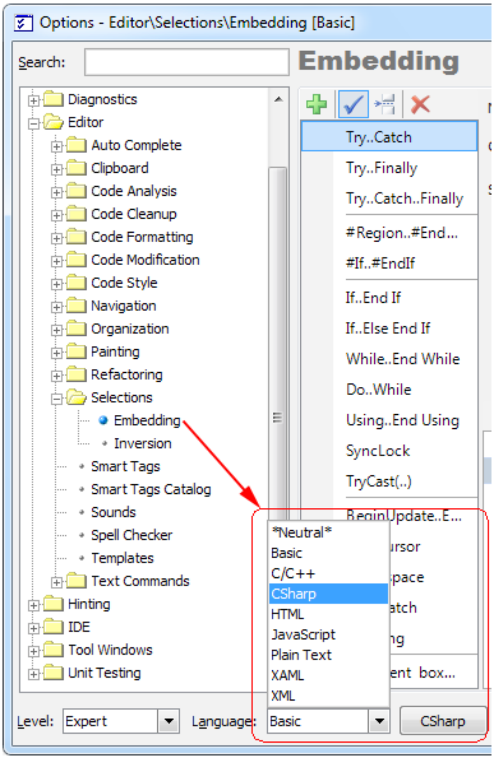
☐ Modify empty lines

```
Try
            Selected Text
Catch ex As Exception
 «Marker»
End Try
```

Level: Expert

Language: Basic

2) Switch to the C# language if it is not active:



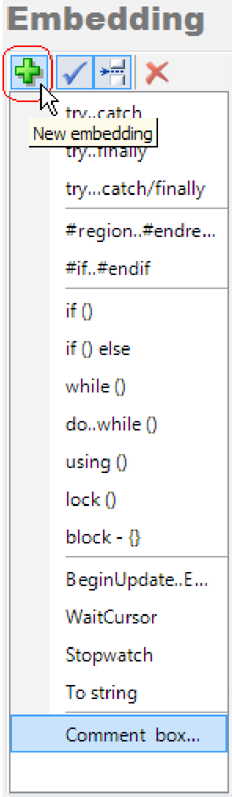
Options - Editor\Selections\Embedding [Basic]

Embedding

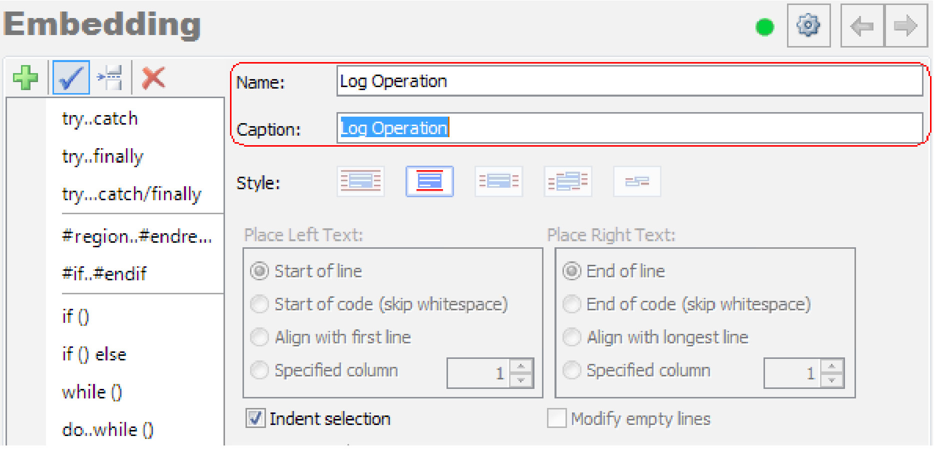
Level: Expert

Language: Basic

3) Click the *Add Embedding* button:



4) Type the *Name* and *Caption* of the new embedding:



Name: Log Operation

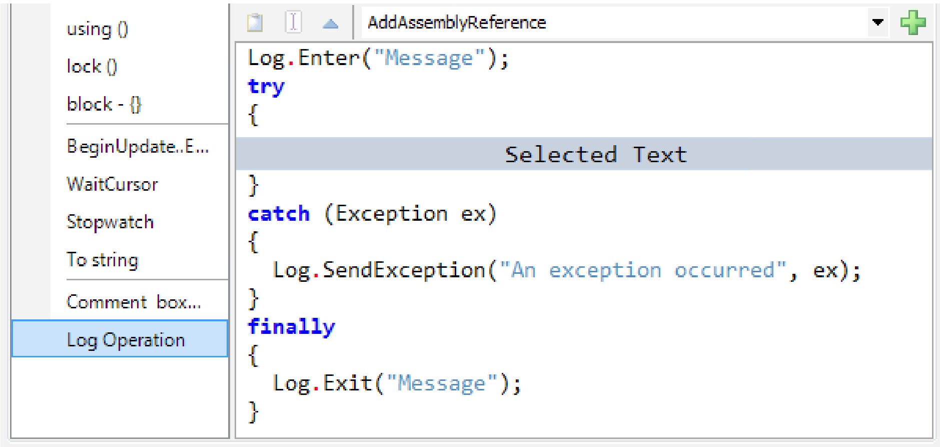
Caption: Log Operation

Place Left Text:
◉ Start of line
○ Start of code (skip whitespace)
○ Align with first line
○ Specified column 1

Place Right Text:
◉ End of line
○ End of code (skip whitespace)
○ Align with longest line
○ Specified column 1

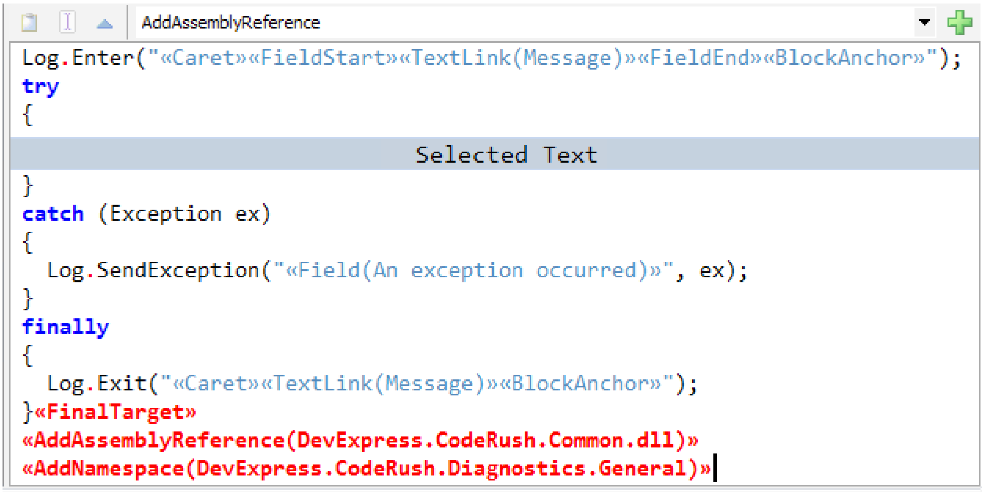
☑ Indent selection
☐ Modify empty lines

5) Leave the *Style* and other options as is, then fill the code embedding expansion area:



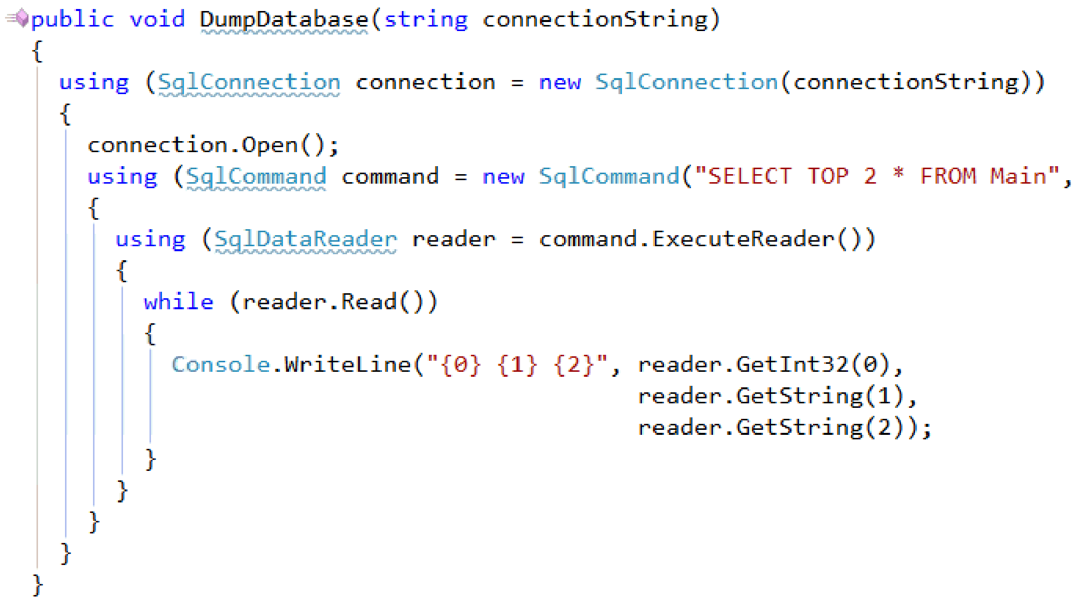
```
Log.Enter("Message");
try
{
                    Selected Text
}
catch (Exception ex)
{
  Log.SendException("An exception occurred", ex);
}
finally
{
  Log.Exit("Message");
}
```

6) Add the required text commands to the expansion:



```
Log.Enter("«Caret»«FieldStart»«TextLink(Message)»«FieldEnd»«BlockAnchor»");
try
{
                    Selected Text
}
catch (Exception ex)
{
  Log.SendException("«Field(An exception occurred)»", ex);
}
finally
{
  Log.Exit("«Caret»«TextLink(Message)»«BlockAnchor»");
}«FinalTarget»
«AddAssemblyReference(DevExpress.CodeRush.Common.dll)»
«AddNamespace(DevExpress.CodeRush.Diagnostics.General)»
```

That's it. Let's try to apply it (*click the animation to enlarge*):

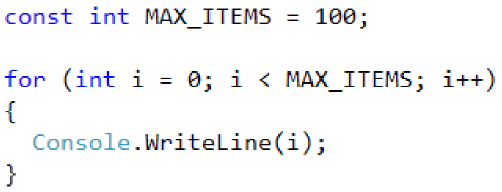


```
public void DumpDatabase(string connectionString)
{
  using (SqlConnection connection = new SqlConnection(connectionString))
  {
    connection.Open();
    using (SqlCommand command = new SqlCommand("SELECT TOP 2 * FROM Main",
    {
      using (SqlDataReader reader = command.ExecuteReader())
      {
        while (reader.Read())
        {
          Console.WriteLine("{0} {1} {2}", reader.GetInt32(0),
                                           reader.GetString(1),
                                           reader.GetString(2));
        }
      }
    }
  }
}
```

Inverting a code block into its opposite using Selection Inversion

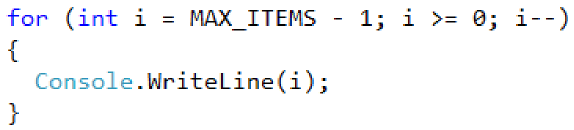
The **Selection Inversion** feature of CodeRush Pro allows you to invert the selected code block to the essential opposite. For example, it can invert assignments so its expression assigns the left side; for loops, so they are iterated in a different order; type cast expression, so an ‘as’ operator expression is converted into an explicit typecast, etc.

Selecting the ‘for’ loop in the following code:



```
const int MAX_ITEMS = 100;

for (int i = 0; i < MAX_ITEMS; i++)
{
  Console.WriteLine(i);
}
```

and applying **Selection Inversion** will yield into the following code:



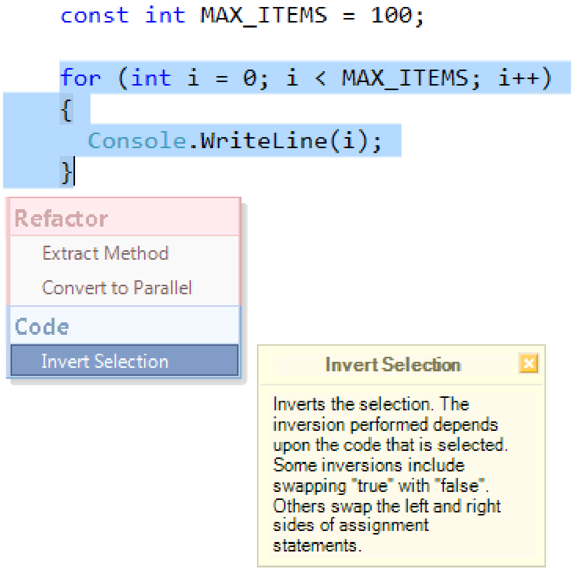
```
for (int i = MAX_ITEMS - 1; i >= 0; i--)
{
  Console.WriteLine(i);
}
```

The feature is based on the regular expressions and easily customizable. You can add your own inverted regular expressions on the Inversion options page. Also, there's is a special DXCore component used to extend the feature called SelectionInversionExtension that adds custom intelligent selection inversions.

To apply the **Selection Inversion**, you can utilize the following techniques:

- Use the default Ctrl+Shift+I shortcut: select a block of code that corresponds to one of the existing selection inversion expressions and press the shortcut to invert it.

- Use the Invert Selection code provider: select a block of code and perform the code provider to invert the selected text:



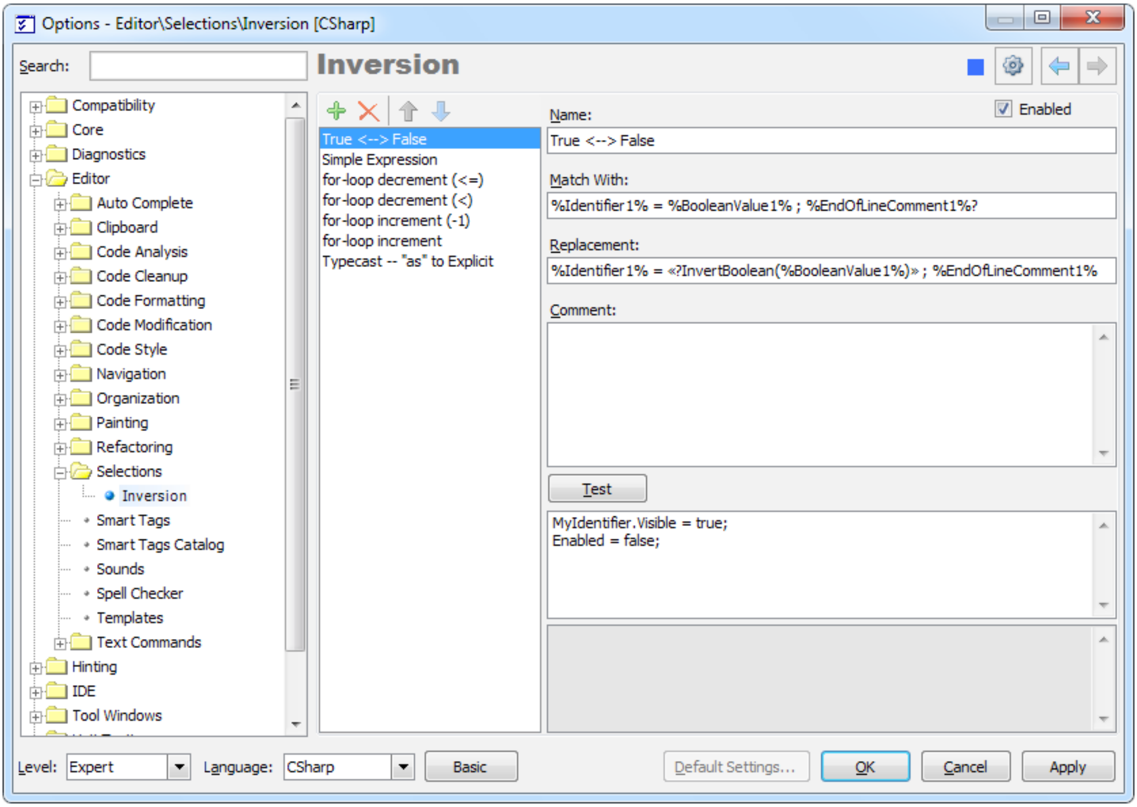
```
const int MAX_ITEMS = 100;

for (int i = 0; i < MAX_ITEMS; i++)
{
  Console.WriteLine(i);
}
```

Inverts the selection. The inversion performed depends upon the code that is selected. Some inversions include swapping "true" with "false". Others swap the left and right sides of assignment statements.

See how you can customize and define your own inversion items on the Selection Inversion options page.

Customizing Selection Inversion and its predefined templates

The **Inversion** options page allows you to add, modify or remove inversion regular expressions that participate in the Selection Inversion feature of CodeRush. Here is what it looks like:



Name: True <--> False

☑ Enabled

Match With: %Identifier1% = %BooleanValue1% ; %EndOfLineComment1%?

Replacement: %Identifier1% = «?InvertBoolean(%BooleanValue1%)» ; %EndOfLineComment1%

```
MyIdentifier.Visible = true;
Enabled = false;
```

Level: Expert

Language: CSharp

The page is multi-language, so you need to select your working programming language in the Options Dialog. The page consists of two parts:

1. The list of inversion items on the left

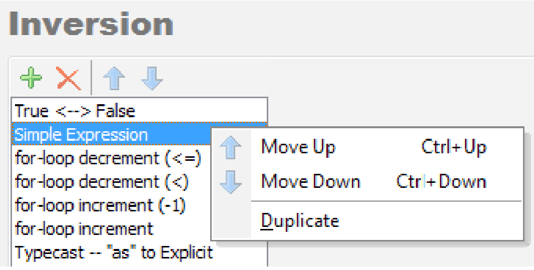
2. Parameters of the selected inversion on the right

The list on the left contains the available inversion items and a toolbar with the following buttons:

Button	Shortcut	Description
✚ Add	Insert	Adds a new inversion item to the list.
✕ Delete	Ctrl+Delete	Removes the selected inversion item from the list.
⇧ Move Up	Ctrl+Up	Moves the selected inversion item up in the list.
⇩ Move Down	Ctrl+Down	Moves the selected inversion item down in the list.

The order of items is important, because the first match will be used, so use the *Move Up* and *Move Down* items to rearrange inversion items. Items on the top of the list have a higher priority over items on the bottom.

You can use the context menu of the list to move items up or down. The context menu also contains the Duplicate menu item that allows you to copy and create a new inversion item based on the selected item:



On the right, you can specify the parameters for the selected inversion item, such as:

- Enabled - toggles the availability of the inversion item. If unchecked, the expression will not be used to match the selected code when the Insertion feature is applied.

- Name - the name of the inversion item.

- Match With - the regular expression based on the DXCore Regular Expression language that matches the selected text when the Inversion feature is applied. This text is used to compare and match with the selected text block.

- Replacement - the regular expression that will be used to convert selected text.

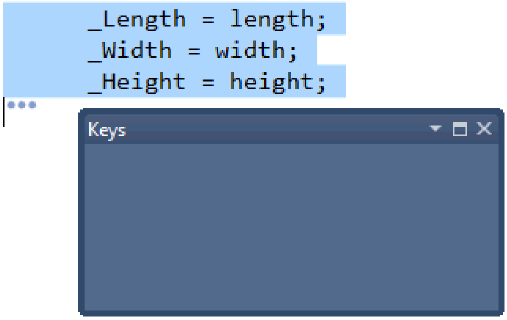
- Comment - an optional comment for the inversion item.

- Test Area – the area where you can test your inversion items and see the produced result. Enter the text you would like to test and click the Test button to see the resulting output replacement.

Quick Comment/Uncomment

The **Selection Comment** feature adds the capability to quickly comment or uncomment a multi-line block of code (or any text in the code editor) via a single shortcut. Only one shortcut is used to comment and uncomment of the current selection. The shortcut is easy to remember. For example, in Visual Basic it is (')(*Apostrophe*) and in CSharp (or C++) it is (/) (*Slash*).

Here's a **sample**:



```
      _Length = length;
      _Width = width;
      _Height = height;
```

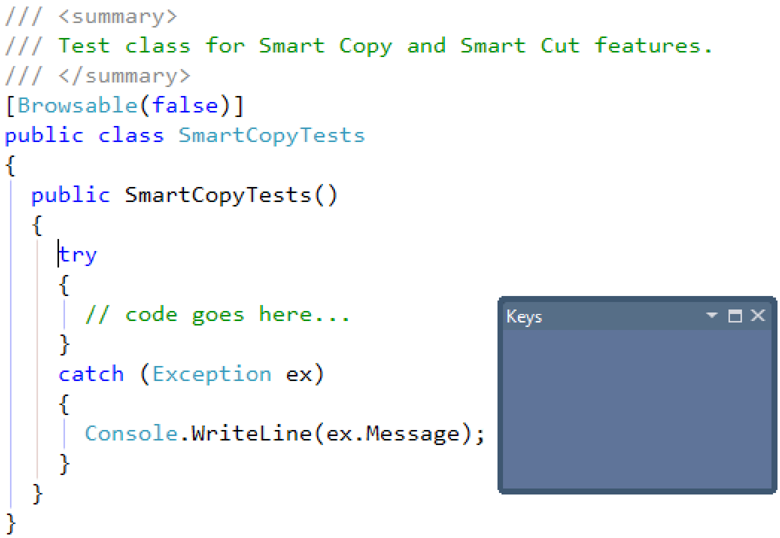
You can customize shortcuts within the IDE | Shortcuts options page in the Options Dialog.

Smart Cut and Smart Copy

These CodeRush clipboard features allow you to take an identifier or a block of code, and put it into clipboard without selecting it in advance. Just place the editor text caret at the identifier or at the start or end of a code block and press the copy key (**Ctrl**+**C** or **Ctrl**+**Insert**) or the cut key (**Ctrl**+**X** or**Shift**+**Delete**). This makes it easy to use cut and copy operations to move, duplicate or delete contiguous blocks of code, like methods, properties, conditional statements, loops, try/catch/finally blocks, comments, etc without selecting it first.

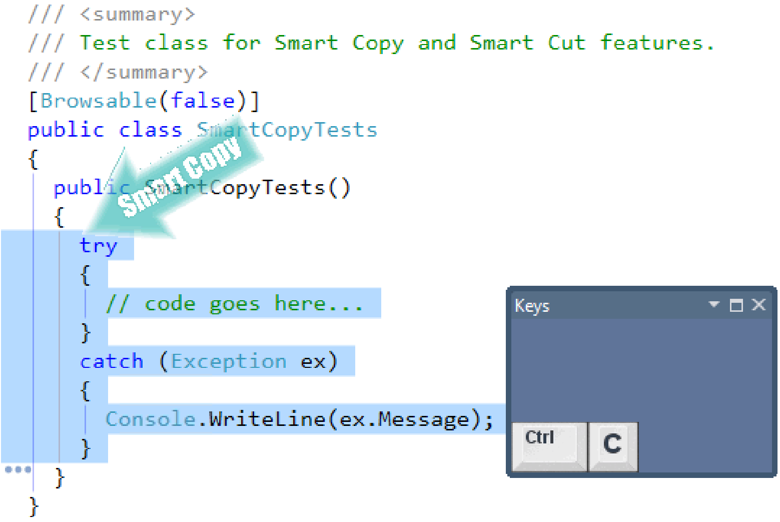
If the text caret is at the start or end of a code block, the entire block will be copied or cut (depending on the key you pressed) into clipboard. **CodeRush** detects that there is no selection and it will create the most intelligent selection for you:

Before:



```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
  public SmartCopyTests()
  {
    try
    {
      // code goes here...
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

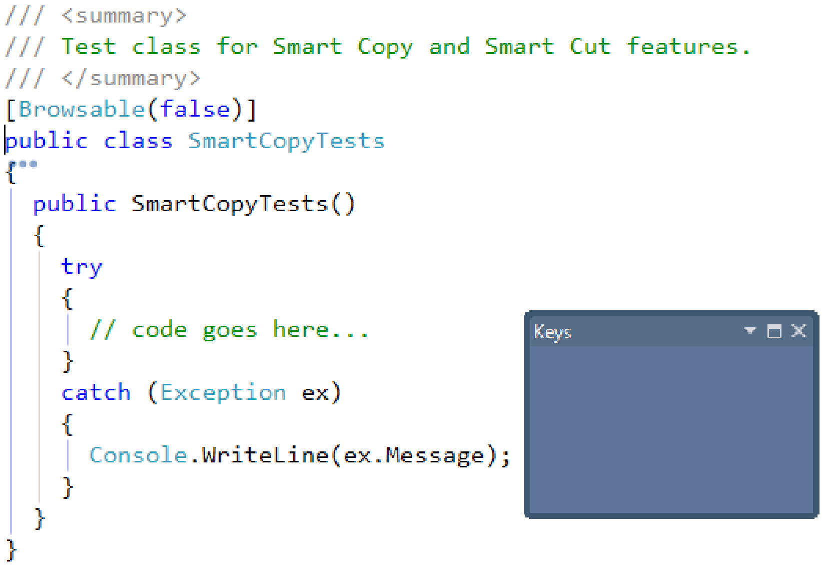
After:



```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
  public SmartCopyTests()
  {
    try
    {
      // code goes here...
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

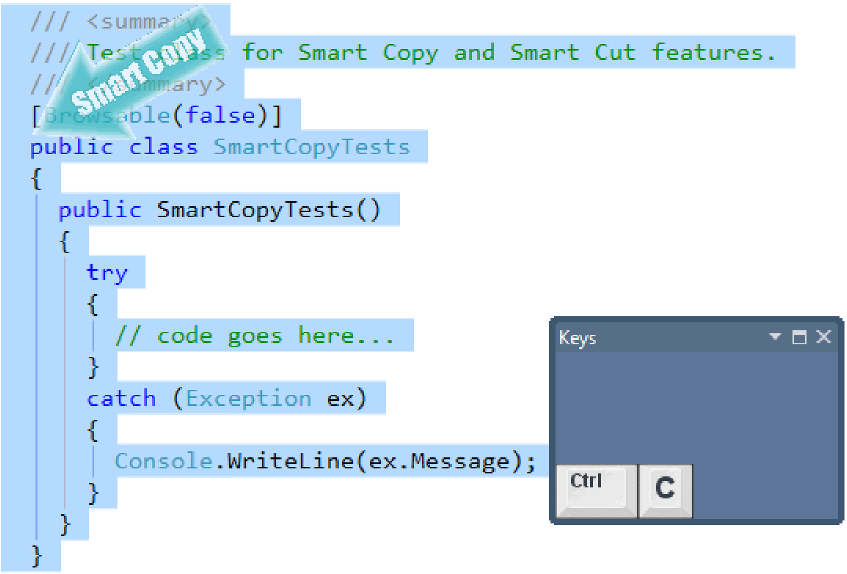
Note that the block of code is selected including the leading and trailing white space. This is the optional behavior (see below). In case of a member or type declarations, the selection will include all support elements such as attributes, xml doc comments as well:

Before:



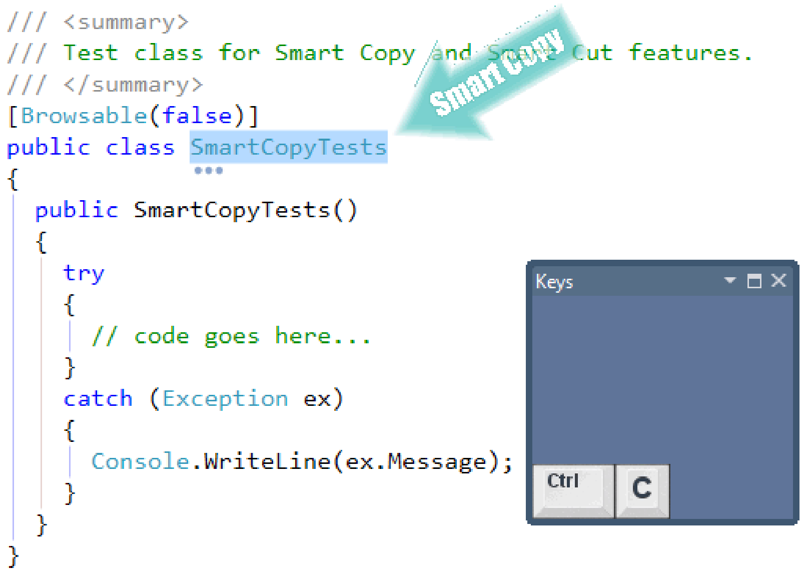
```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
    public SmartCopyTests()
    {
        try
        {
            // code goes here...
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

After:



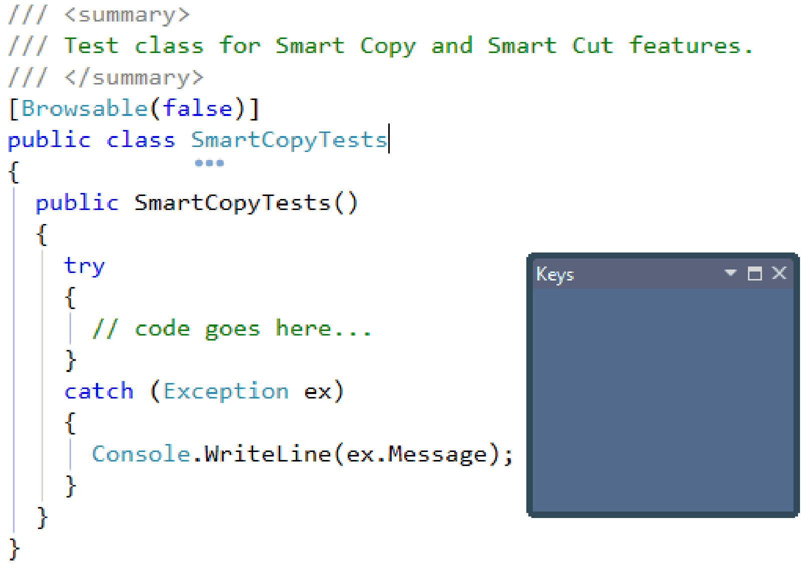
```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
    public SmartCopyTests()
    {
        try
        {
            // code goes here...
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Otherwise, if the caret is at the identifier (name or type) the only identifier is copied (or cut):



```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
  public SmartCopyTests()
  {
    try
    {
      // code goes here...
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

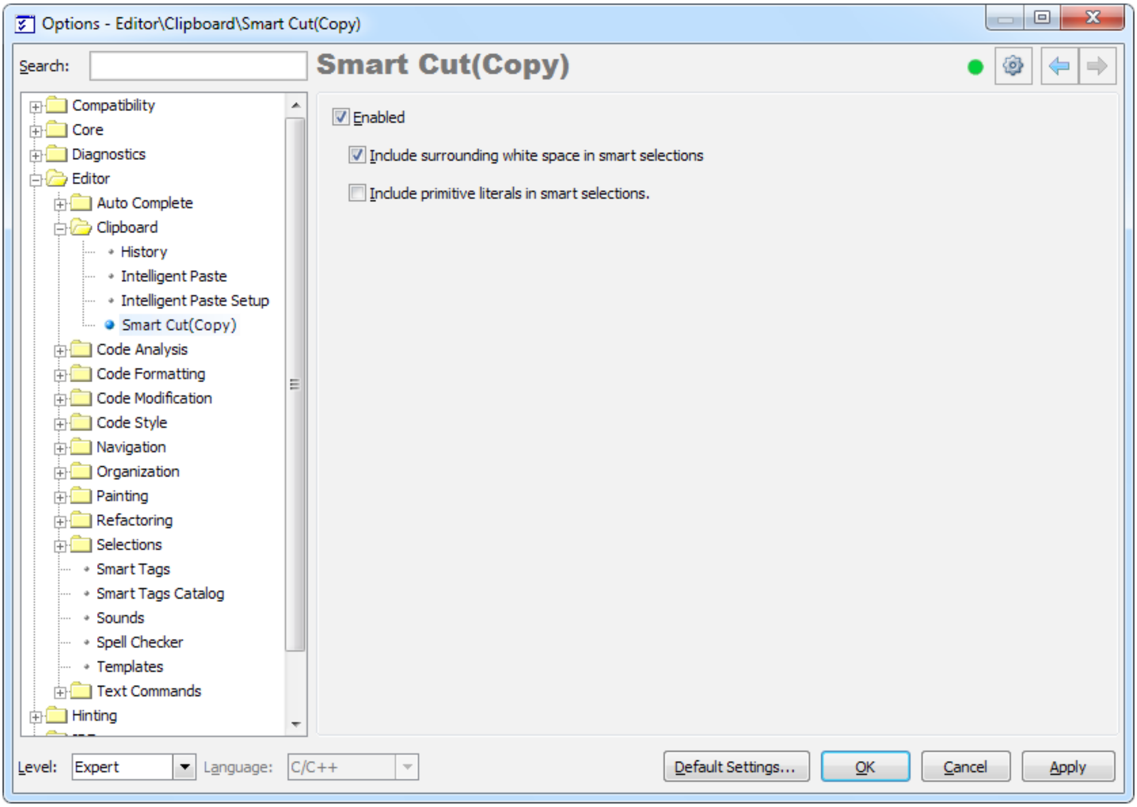
Once the identifier is on the clipboard, other **CodeRush** features, such as Code Templates become very useful. For example, if we copy the class name onto a clipboard, we can easily create a descendant for this class using the "**c**" code template:



```
/// <summary>
/// Test class for Smart Copy and Smart Cut features.
/// </summary>
[Browsable(false)]
public class SmartCopyTests
{
  public SmartCopyTests()
  {
    try
    {
      // code goes here...
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Note that if the selection exists and you press the cut or copy key, the default Visual Studio behavior is performed, i.e. the only selected text is being copied into clipboard as is. The same happens if the text caret is surrounded by white space – the current line will be placed on the clipboard, which is the default Visual Studio cut and copy behavior (optional, enabled by default).

You can disable or enable the **Smart Cut** and/or **Smart Copy** on the "**Editor | Clipboard | Smart Cut**(**Copy**)" option page in the Options Dialog:



Options - Editor\Clipboard\Smart Cut(Copy)

Smart Cut(Copy)

☑ Enabled

☑ Include surrounding white space in smart selections

☐ Include primitive literals in smart selections.

Level: Expert

Language: C/C++

There are two additional options available:

- Include surrounding white space in smart selections

If this option is turned on, a **Smart Cut** (**Copy**) selection includes the leading and trailing white space of an active block of code. Otherwise, it includes only the constructive parts of a code block.

- Include primitive literals in smart selections

This option affects the **Smart Copy**, **Smart Cut** and Paste Replace Word features when working with text strings. If the option is enabled, **CodeRush** copies (cuts) or replaces an entire string including quotes. Otherwise, it replaces only the word under the text caret inside of a text string expression.

Intelligent Paste

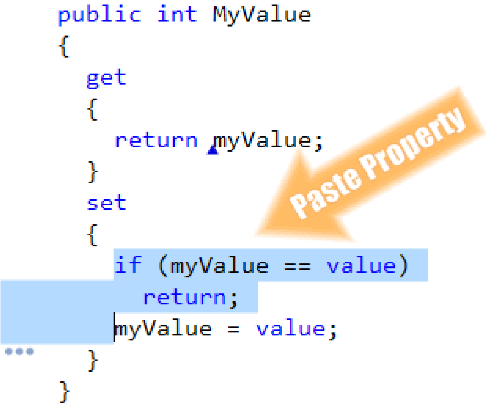
The CodeRush **Intelligent Paste** clipboard feature modifies the text from the clipboard before inserting it into the code editor. An action hint with the name of the expansion appears once the suitable expansion is triggered on the **Paste** (**Ctrl**+V) command.

For example, copying a field member into a clipboard and then pasting it on the next line will produce a read-write property for the field:

Before:

```
private int myValue;
```

After:



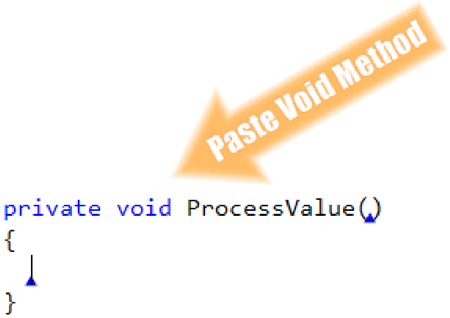
```
public int MyValue
{
  get
  {
    return myValue;
  }
  set
  {
    if (myValue == value)
      return;
    myValue = value;
  }
}
```

Another example is when copying simple method invoke expression (method call), the entire void method declaration will be pasted with markers:

Before:

```
ProcessMyValue();
```

After:



```
private void ProcessValue()
{

}
```

To execute **Intelligent Paste**, simply perform a usual **Paste** command bound to **Ctrl+V** or **Shift+Insert** shortcut s preceded by copying a piece of code with a structure defined as an **Intelligent Paste** expansion. The list of such expansions can be found on the **Editor** | **Clipboard** | Intelligent Paste options page in the Options Dialog. **Intelligent Paste** is fully extensible through the options page, and lets you specify the precise conditions that trigger the particular extension.

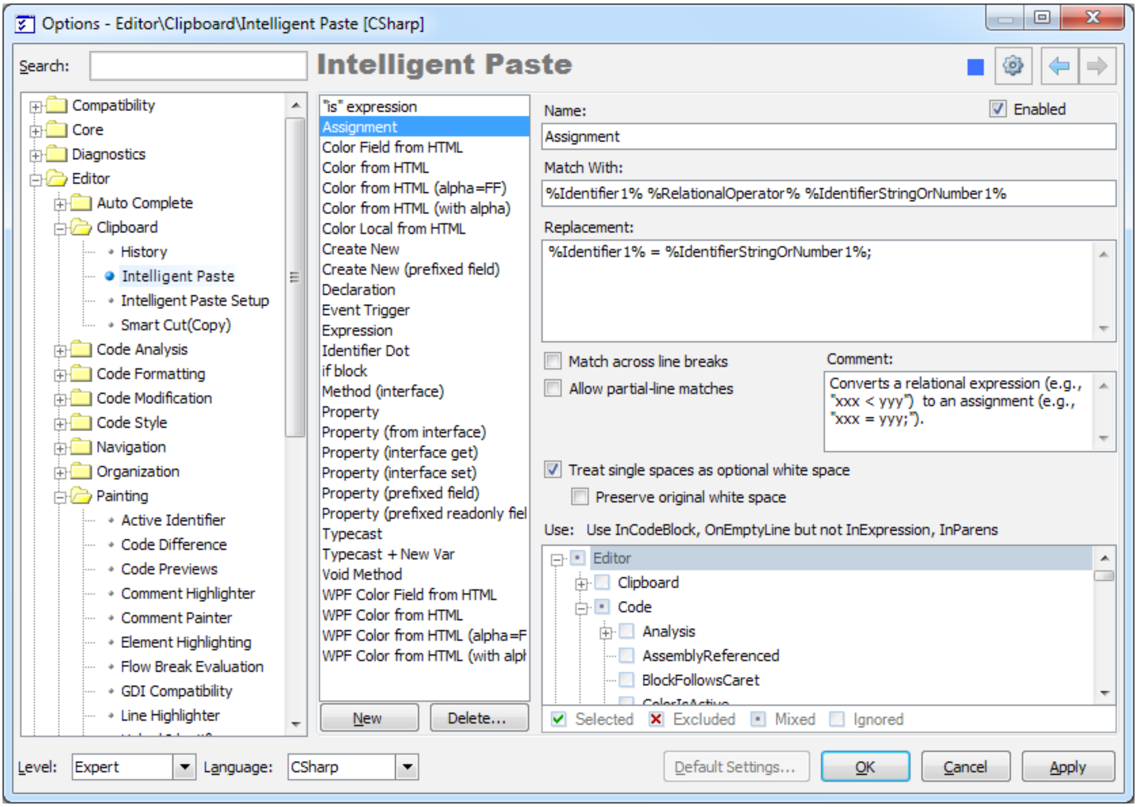
Here's the list of built-in **CodeRush Intelligent Paste** expansions for *CSharp* language. Most of these expansions are also defined for *Visual Basic* and *C++* languages. Of course, you can easily add new expansions to any supported language. Note that you can use DXCore text commands inside expansions.

Name	Description
Assignment	Converts a relational expression (e.g., “xxx < yyy”) to an assignment (e.g., “xxx = yyy”).
Color Field from HTML	Converts an HTML color string (e.g., “#FF0000″) to a new color instance (e.g., “Color.FromArgb(0xFF, 0×00, 0×00)”).
Color from HTML	Converts an HTML color string (e.g., “#FF0000″) to a new color instance (e.g., “Color.FromArgb(0xFF, 0×00, 0×00)”).
Color from HTML (alpha=FF)	Converts an HTML color string (e.g., “#FFAA0000″) to a new color instance (e.g., “Color.FromArgb(0xAA, 0×00, 0×00)”).
Color from HTML (with alpha)	Converts an HTML alpha-blended color string (e.g., “#80AA0000″) to a new color instance (e.g., “Color.FromArgb(0×80, 0xAA, 0×00, 0×00)”).
Color Local from HTML	Converts an HTML color string (e.g., “#FF0000″) to a new color instance (e.g., “Color.FromArgb(0xFF, 0×00, 0×00)”).
Create New	Converts a field declaration (e.g., “private Hashtable myTable;”) into code that constructs an instance of that identifier (e.g., “myTable = new Hashtable();”) when pasted on an empty line inside a method or property.
Create New (prefixed field)	Converts a field declaration (e.g., “private Hashtable _MyTable;”) into code that constructs an instance of that identifier (e.g., “_MyTable = new Hashtable();”) when pasted on an empty line inside a method or property.
Declaration	Converts a statement that constructs a new identifier into a declaration.
Event Trigger	Creates an event trigger for an event declaration.
Expression	Converts an assignment to an expression when pasted inside parens or following an assignment statement.
Identifier Dot	Invokes Intellisense when pasting identifiers followed by a “.” inside code blocks.
if block	Converts an if-statement with an expression to the extracted expression when pasted inside parens.
Is expression	Converts a typecast to an “is” expression. Note that this expansion is disabled by default.
Method (interface)	Creates a new method when a method declaration from an interface is pasted on an empty line inside a class or struct.
Property	Creates a property from a field declaration.
Property (from interface)	Creates a new property when a property declaration from an interface is pasted on an empty line inside a class or struct.
Property (interface get)	Creates a new property when a property declaration from an interface is pasted on an empty line inside a class or struct.
Property (interface set)	Creates a new property when a property declaration from an interface is pasted on an empty line inside a class or struct.
Property (prefixed field)	Creates a property from a declaration for a prefixed field (e.g., “_MyField”, or “m_MyField”).
Property (prefixed readonly field)	Creates a property from a declaration for a prefixed readonly field (e.g., “_MyField”, or “m_MyField”).
Typecast + New Var	Creates a new local variable of the type tested in the “is” expression, and initializes it to the identifier typecast as that type. Valid on empty lines in code blocks.
Void Method	Creates a new method declaration when a method call is pasted on an empty line outside a method.

WPF Color Field from HTML	Converts an HTML color string (e.g., “#FF0000″) to a new color instance (e.g., “Color.FromRgb(0xFF, 0×00, 0×00)”).
WPF Color from HTML	Converts an HTML color string (e.g., “#FF0000″) to a new color instance (e.g., “Color.FromRgb(0xFF, 0×00, 0×00)”).
WPF Color from HTML (alpha=FF)	Converts an HTML color string (e.g., “#FFAA0000″) to a new color instance (e.g., “Color.FromRgb(0xAA, 0×00, 0×00)”).
WPF Color from HTML (with alpha)	Converts an HTML alpha-blended color string (e.g., “#80AA0000″) to a new color instance (e.g., “Color.FromArgb(0×80, 0xAA, 0×00, 0×00)”).

Intelligent Paste Options

There are two options pages for the Intelligent Paste **CodeRush** clipboard feature inside IDETools Options Dialog. The first one defines the main expansions and the second one allows you to manage **Intelligent Paste** extensions. Note the difference between “expansions” and “extensions”; in other words, extensions provide expansions. For example, the “*Regular Expressions*” extension provides the main list of **Intelligent Paste** expansions from the first options page:


Options - Editor\Clipboard\Intelligent Paste [CSharp]

Intelligent Paste

Name: Assignment

☑ Enabled

Match With: %Identifier1% %RelationalOperator% %IdentifierStringOrNumber1%

Replacement: %Identifier1% = %IdentifierStringOrNumber1%;

☐ Match across line breaks

☐ Allow partial-line matches

Comment: Converts a relational expression (e.g., "xxx < yyy") to an assignment (e.g., "xxx = yyy;").

☑ Treat single spaces as optional white space

☐ Preserve original white space

Use: Use InCodeBlock, OnEmptyLine but not InExpression, InParens

Level: Expert

Language: CSharp

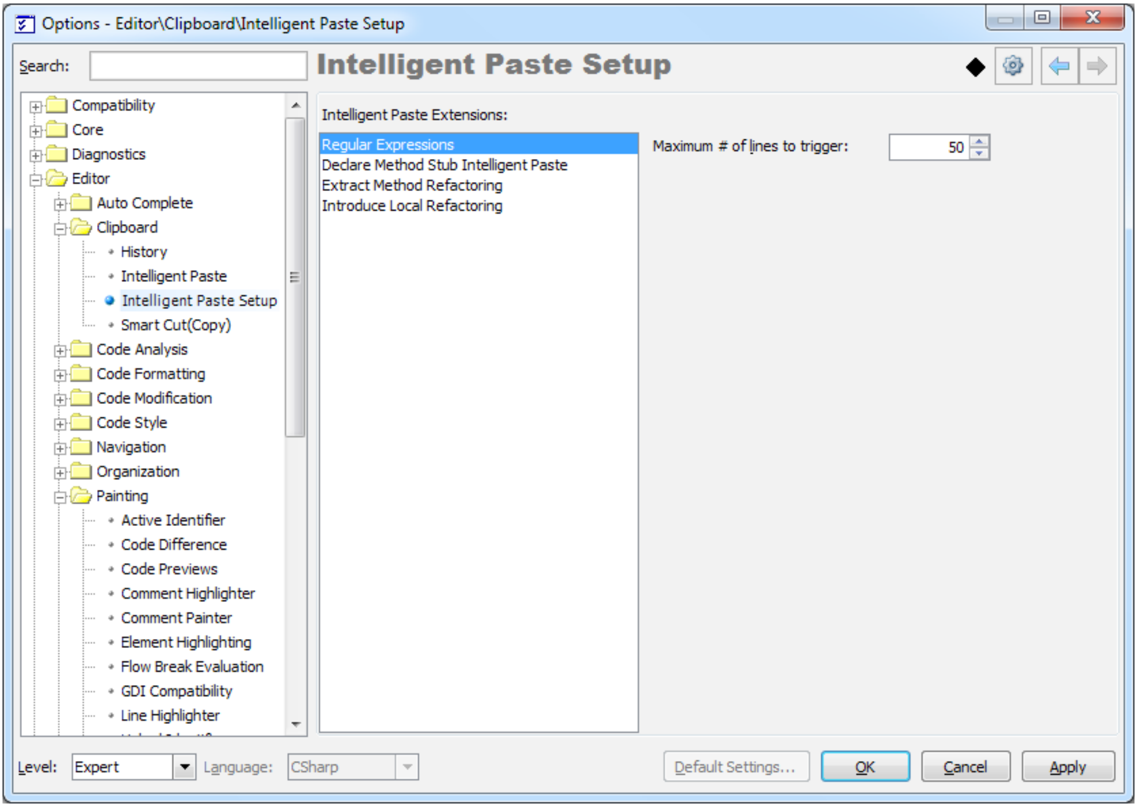
This options page lists the main **Intelligent Paste** expansions available, and enables you to customize options for each expansion.

The page consists of the expansions list on the left, and the options for the selected expansion on the right. Under the list with expansions on the left there are two buttons: “*New*” and “*Delete…*“. These buttons are used for adding and removing expansion items. As an alternative you can use the context menu available via right-clicking on the list and choosing the single "*Duplicate*" menu item. This way you can easily add a new item based on the existing one and modify its context, replacement and **Match With** values of the selected expansion.

Once you selected an expansion, the following options related to the expansion are available:

Option	Description
Enabled	If this option is on, the selected expansion under the **Name** label will be enabled and **CodeRush Intelligent Paste Engine** will try to perform a replacement conversion operation of the text on the clipboard, if appropriate.
Match With	Specifies the clipboard text template using the build-in DXCore reg-ex aliases engine. If the text from the clipboard matches this template, the expansion activates.
Replacement	Specifies the code block pasted by the expansion instead of the initial clipboard text content.
Comment	The additional information to describe the expansion. The value of this option does not affect the expansion functionality.
Match across line breaks	If this option is on, **Intelligent Paste Engine** compares each text line from the clipboard with the template, and activates the expansion for each matched line.
Allow partial-line matches	If this option is on, the expansion is activated, even if only a part of the clipboard content matches the template. In this case, the expansion changes only the part that matches the template.
Treat single spaces as optional whitespace	If this option is on, **Intelligent Paste Engine** treats single spaces as an optional whitespace from the **Match With** and **Replacement** values. This allows you to enter these values easily, which makes them more flexible in regard to whitespaces (tabs, spaces, several spaces in a row, etc).
Preserve original whitespace	Specifies whether whitespace from the clipboard is preserved in the replacement expansion.
Context	Specifies the context where the expansion is available.

On the second options page **Editor | Clipboard | Intelligent Paste Setup**, the extensions that provide **Intelligent Paste** expansions are listed:



Options - Editor\Clipboard\Intelligent Paste Setup

Intelligent Paste Setup

Intelligent Paste Extensions:

- Regular Expressions
- Declare Method Stub Intelligent Paste
- Extract Method Refactoring
- Introduce Local Refactoring

Maximum # of lines to trigger: 50

Level: Expert

Language: CSharp

There are four extensions shipped with CodeRush and Refactor!:

Name	Description
Regular Expressions	These expansions are listed on the first options page.
Declare Method Stub Intelligent Paste	Invokes the **Declare Method** code provider to construct the method with appropriate parameters when method invoke expression is copied and then pasted into the class or struct.
Extract Method Refactoring	Invokes the Extract Method refactoring for the selected block of code when it is pasted into a class or struct.
Introduce Local Refactoring	Invokes the **Introduce Local** refactoring for the selected code expression when it is pasted into a new line inside of the same member scope.

The **Maximum # of lines to trigger** option specifies the upper limit of the lines of the text on the clipboard for the selected **Intelligent Paste** extension. If the clipboard contains a larger text content than specified with this option, the extension is not triggered.

If you add a new **Intelligent Paste** extension using the DXCore Intelligent Paste Extension component, the new extension will appear on this options page.

Paste Replace Word

Once you have a word (identifier) on the clipboard, you can easily replace any different word (identifier) with the one on the clipboard using **Ctrl+B** shortcut. The **Paste Replace** CodeRush action will select the word (identifier) at the caret before pasting in the contents of the clipboard, so you don't have to select it first.

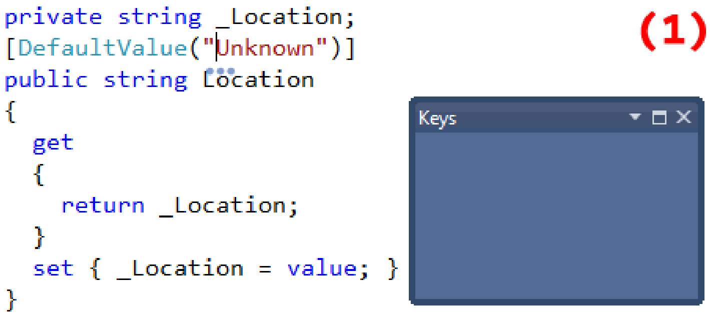
Here is an example. Consider that we have the Location property, which should be returning the default "*Unknown*" value, but currently it returns null. Let's see how the **Paste Replace Word** feature will be useful here:

1. Put the editor caret on the "*Unknown*" word

2. Press **Ctrl+C** to copy it into the Clipboard, using another Smart Copy clipboard feature

3. Put the editor caret on the "*null*" keyword

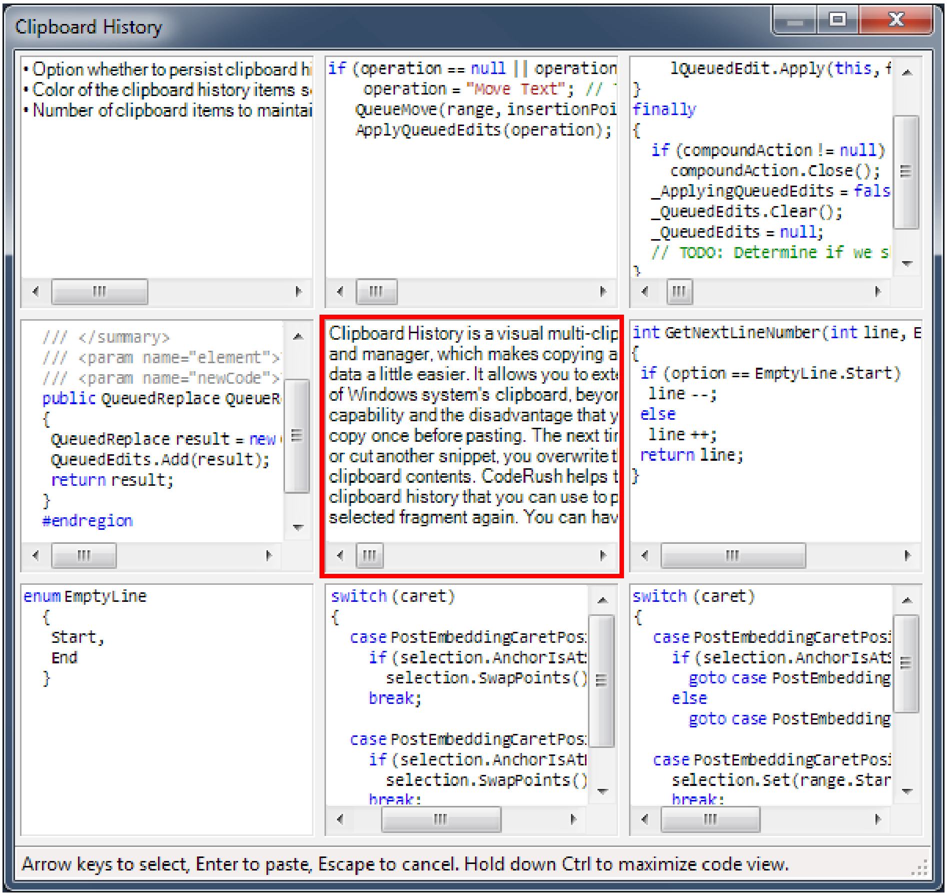
4. Press **Ctrl+B** to replace the keyword:



```
private string _Location;
[DefaultValue("Unknown")]
public string Location
{
  get
  {
    return _Location;
  }
  set { _Location = value; }
}
```

Clipboard History

Clipboard History is a visual multi-clipboard viewer and manager, which makes copying and pasting of data a little easier. It allows you to extend the facility of *Windows* system's clipboard, beyond its default capability and the disadvantage that you can only copy once before pasting. The next time you copy or cut another snippet, you overwrite the existing clipboard contents. CodeRush helps to keep the clipboard history that you can use to paste any selected fragment again. You can have up to ٦٤ independent fragments and work with each of them separately, persisting these fragments across Visual Studio sessions for future use. If you copy a code fragment, **Clipboard History** will maintain its syntax highlighting as well:



- Option whether to persist clipboard hi
- Color of the clipboard history items s
- Number of clipboard items to maintai

```
if (operation == null || operation
   operation = "Move Text"; //
  QueueMove(range, insertionPoi
  ApplyQueuedEdits(operation);
```

```
    lQueuedEdit.Apply(this, f
}
finally
{
  if (compoundAction != null)
    compoundAction.Close();
  _ApplyingQueuedEdits = fals
  _QueuedEdits.Clear();
  _QueuedEdits = null;
  // TODO: Determine if we s
}
```

```
  /// </summary>
  /// <param name="element">
  /// <param name="newCode">
  public QueuedReplace QueueR
  {
   QueuedReplace result = new
   QueuedEdits.Add(result);
   return result;
  }
  #endregion
```

Clipboard History is a visual multi-clip
and manager, which makes copying a
data a little easier. It allows you to ext
of Windows system's clipboard, beyor
capability and the disadvantage that y
copy once before pasting. The next tir
or cut another snippet, you overwrite t
clipboard contents. CodeRush helps t
clipboard history that you can use to p
selected fragment again. You can hav

```
int GetNextLineNumber(int line, E
{
 if (option == EmptyLine.Start)
  line --;
 else
  line ++;
 return line;
}
```

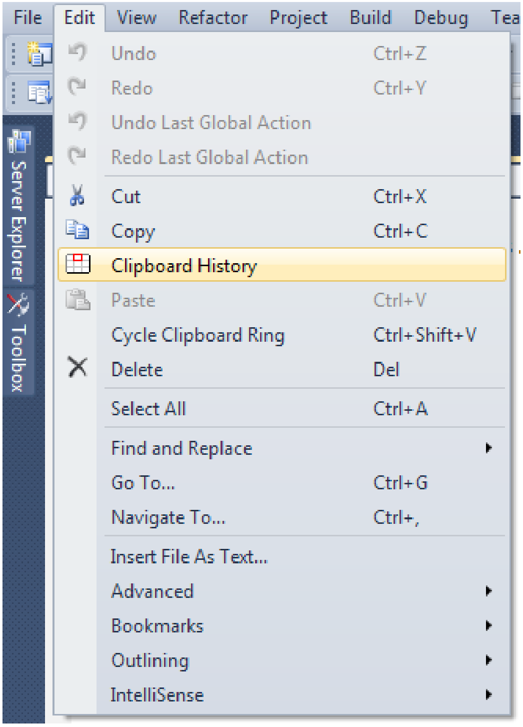
```
enum EmptyLine
  {
   Start,
   End
  }
```

```
switch (caret)
{
  case PostEmbeddingCaretPos
    if (selection.AnchorIsAt
      selection.SwapPoints()
    break;

  case PostEmbeddingCaretPos
    if (selection.AnchorIsAt
      selection.SwapPoints()
    break;
```

```
switch (caret)
{
  case PostEmbeddingCaretPosi
    if (selection.AnchorIsAt
      goto case PostEmbedding
    else
      goto case PostEmbedding

  case PostEmbeddingCaretPosi
    selection.Set(range.Star
    break;
```

Arrow keys to select, Enter to paste, Escape to cancel. Hold down Ctrl to maximize code view.

To access the **Clipboard History** – press **Ctrl+Shift+Insert** or choose it from the **Edit** menu:



On the other hand, Visual Studio has a built-in feature called **Clipboard Ring**. The **Clipboard Ring** keeps track of the last ٢٠ items you have either cut or copied so you can reuse them over and over again. Repeatedly pressing **Ctrl**+**Shift**+**V** will cycle through the entries in the **Clipboard Ring**. Each time you press **Ctrl**+**Shift**+**V**, the IDE editor replaces the last entry you pasted from the **Clipboard Ring** so that you end up with only the last one you selected.

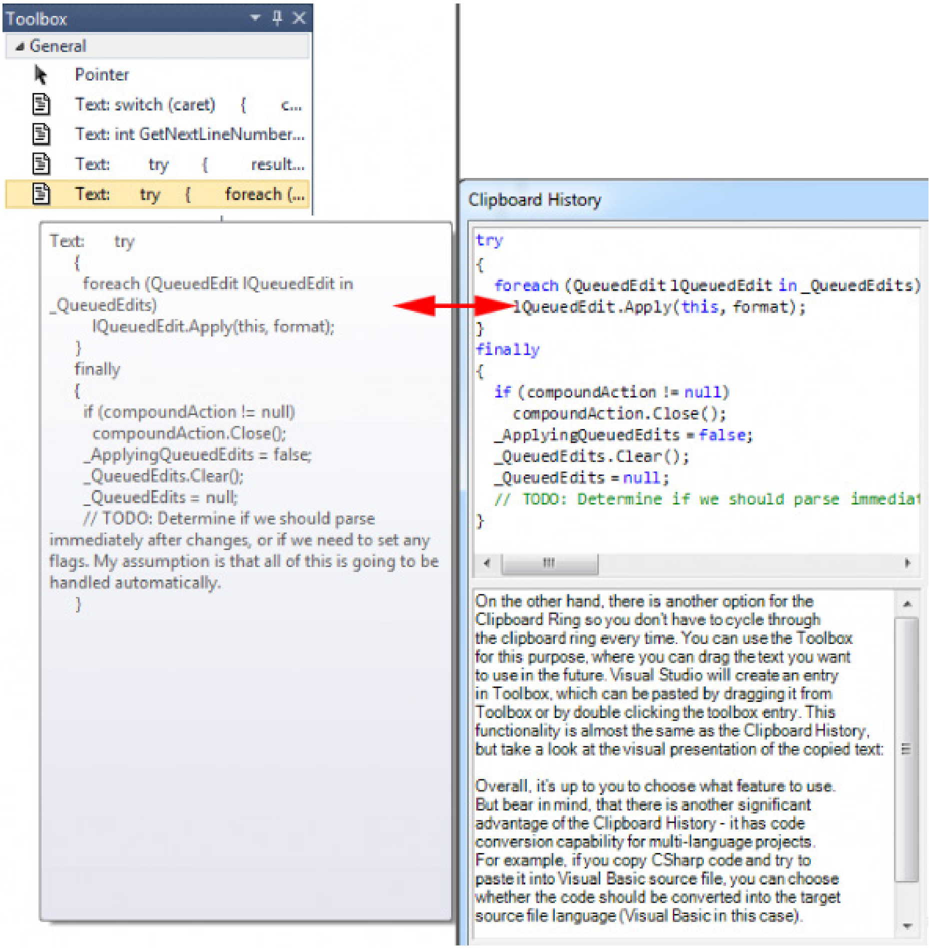
So, what's the difference between two? Let's compare:

- **Usability**

For example, if you wanted to paste the 10th item in the clipboard ring, you need to press **Ctrl-Shift-V 10** times. If you don't know what the position of the item in the ring, you have to press the shortcut numerous times, until you find the item you need. Using the **Clipboard History**, all you need is to click once on the item you need.

- **Visual presentation**

On the other hand, there is another option for the **Clipboard Ring** so you don't have to cycle through the clipboard ring every time. You can use the Toolbox for this purpose, where you can drag the text you want to use in the future. Visual Studio will create an entry in Toolbox, which can be pasted by dragging it from Toolbox or by double clicking the toolbox entry. This functionality is almost the same as the **Clipboard History**, but take a look at the visual presentation of the copied text:



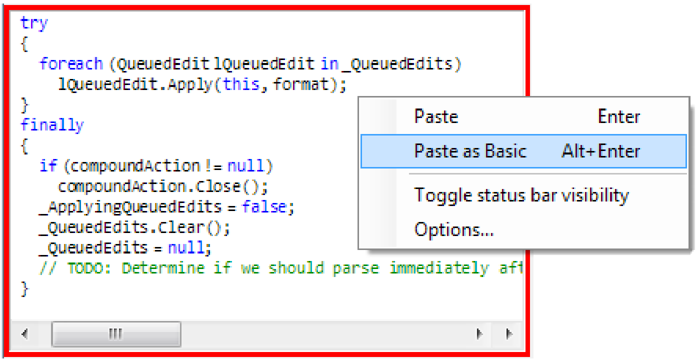
```
Text:     try
      {
        foreach (QueuedEdit lQueuedEdit in
_QueuedEdits)
          lQueuedEdit.Apply(this, format);
      }
      finally
      {
        if (compoundAction != null)
          compoundAction.Close();
        _ApplyingQueuedEdits = false;
        _QueuedEdits.Clear();
        _QueuedEdits = null;
        // TODO: Determine if we should parse
immediately after changes, or if we need to set any
flags. My assumption is that all of this is going to be
handled automatically.
      }
```

```
try
{
  foreach (QueuedEdit lQueuedEdit in _QueuedEdits)
    lQueuedEdit.Apply(this, format);
}
finally
{
  if (compoundAction != null)
    compoundAction.Close();
  _ApplyingQueuedEdits = false;
  _QueuedEdits.Clear();
  _QueuedEdits = null;
  // TODO: Determine if we should parse immediat
}
```

On the other hand, there is another option for the Clipboard Ring so you don't have to cycle through the clipboard ring every time. You can use the Toolbox for this purpose, where you can drag the text you want to use in the future. Visual Studio will create an entry in Toolbox, which can be pasted by dragging it from Toolbox or by double clicking the toolbox entry. This functionality is almost the same as the Clipboard History, but take a look at the visual presentation of the copied text:

Overall, it's up to you to choose what feature to use. But bear in mind, that there is another significant advantage of the Clipboard History - it has code conversion capability for multi-language projects. For example, if you copy CSharp code and try to paste it into Visual Basic source file, you can choose whether the code should be converted into the target source file language (Visual Basic in this case).

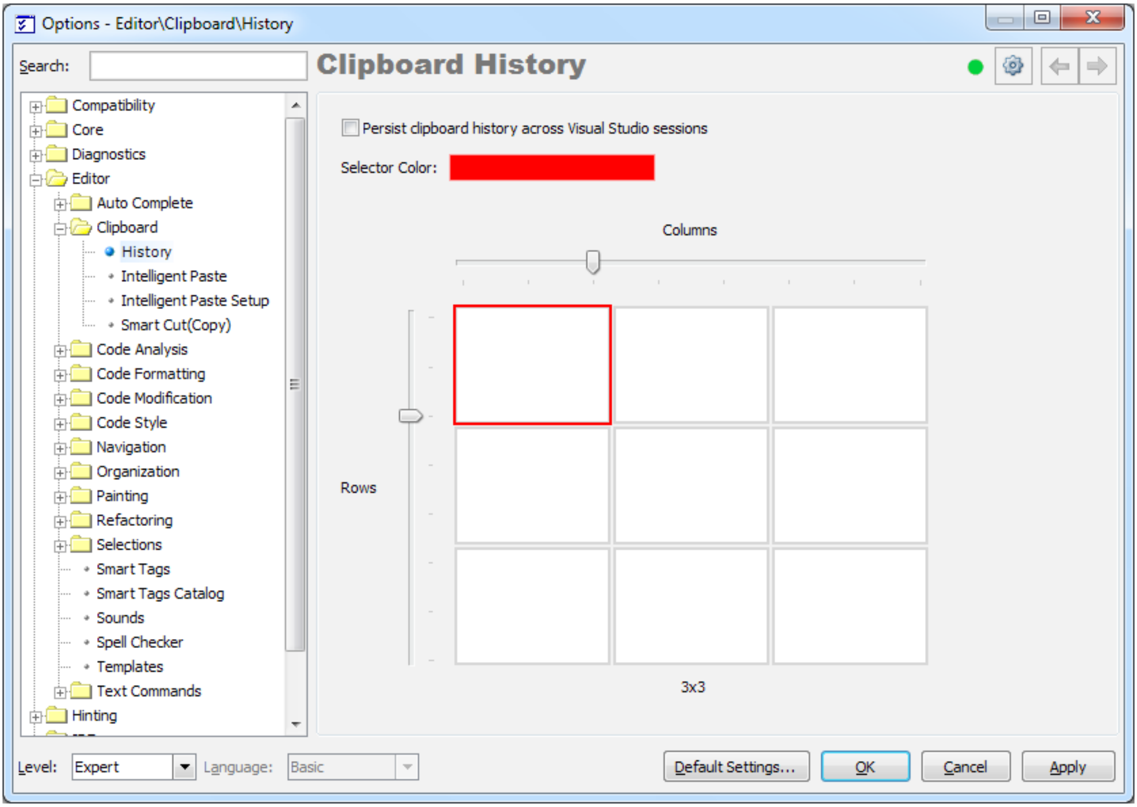
Overall, it's up to you to choose what feature to use. But bear in mind, that there is another significant advantage of the **Clipboard History** – it has code conversion capability for multi-language projects. For example, if you copy *CSharp* code and try to paste it into *Visual Basic* source file, you can choose whether the code should be converted into the target source file language (*Visual Basic* in this case):



```
try
{
  foreach (QueuedEdit lQueuedEdit in _QueuedEdits)
    lQueuedEdit.Apply(this, format);
}
finally
{
  if (compoundAction != null)
    compoundAction.Close();
  _ApplyingQueuedEdits = false;
  _QueuedEdits.Clear();
  _QueuedEdits = null;
  // TODO: Determine if we should parse immediately aft
}
```

On the **Clipboard History** options page in the Options Dialog, there are a few simple options available:

- Option whether to persist clipboard history across Visual Studio sessions
- Color of the clipboard history items selector
- Number of clipboard items to maintain (1 – 64) by choosing the rows and columns size


Options - Editor\Clipboard\History

Clipboard History

□ Persist clipboard history across Visual Studio sessions

Selector Color: 🟥

Columns

Rows

3x3

Level: Expert

Language: Basic

Chapter 7. Code modification providers

Create Descendant and Create Descendant (with virtual overrides)

The **Create Descendant** code provider shipped in CodeRush generates a descendant class for the active class, providing overrides for abstract members, if any. The second version of the code provider named **Create Descendant** (**with virtual overrides**), in addition to the **Create Descendant**, adds overrides for virtual members into a descendant class.

Applying the **Create Descendant** for the following abstract class:

```
public abstract class Logger
{
  public abstract void GetMessage(int index);
  public abstract void SendMessage(string msg);
}
```

Will generate the following descendant declaration:

```
public class LoggerDescendant : Logger
{
  public LoggerDescendant()
  {

  }
  public override void GetMessage(int index)
  {
    throw new NotImplementedException();
  }
  public override void SendMessage(string msg)
  {
    throw new NotImplementedException();
  }
}
```

If a class contains virtual members you can choose whether they should be overridden by applying the corresponding provider. Having the following class:

```
public class Calculator
{
  public virtual void Add()  {  }
  public virtual void Sub()  {  }
  public virtual void Mul()  {  }
  public virtual void Div()  {  }
}
```

Applying the **Create Descendant** (**with virtual overrides**) will result in generating the following class:

```
public class CalculatorDescendant : Calculator
{
  public CalculatorDescendant()
  {

  }
  public override void Add()
  {
    base.Add();
  }
  public override void Sub()
  {
    base.Sub();
  }
  public override void Mul()
  {
    base.Mul();
  }
  public override void Div()
  {
    base.Div();
  }
}
```

Both code providers add a default public parameterless constructor for the new class, and drop a marker at the starting source position. The new class is placed above the current class in the same source file.

Create Ancestor

The **Create Ancestor** code provider generates a new base class declaration for the active class. The generated base class will be declared above the active class and the active class becomes a descendant of the new base class. The new base class has the same visibility as the active class and contains the default public parameterless constructor. All identifiers and references of the base class are linked together for easy renaming.

For example, applying the **Create Ancestor** for the following class:

```
public class Logger
{
  public Logger() { }
}
```

Will result in the following result:

```
public class LoggerBase
{
  public LoggerBase()
  {

  }
}
public class Logger : LoggerBase
{
  public Logger() { }
}
```

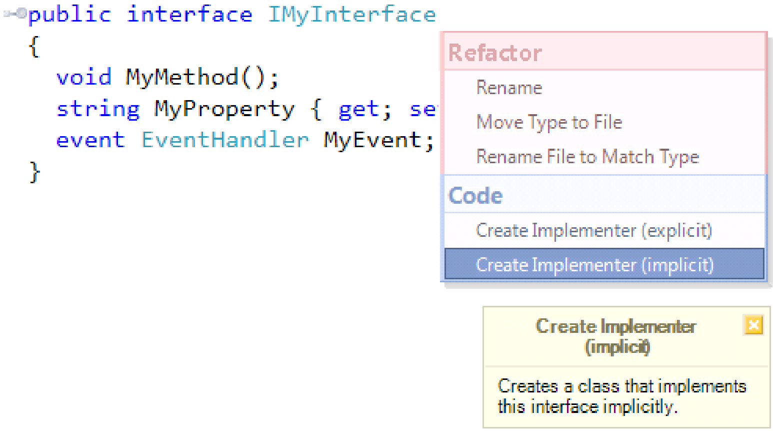
Note that the code provider won't be available when the active class already descends from another base class.

Creating interface implementers

A class that implements an interface should implement all members of that interface. Members of implemented interfaces can be declared in two ways: implicit or explicit. That is why there are two versions of the **Create Implementer** code provider in CodeRush:

- **Create Implementer (implicit)**
- **Create Implementer (explicit)**

The **Create Implementer** (**implicit**) code provider creates a class that implements an interface implicitly. You can apply this code provider on the interface itself by placing the editor caret at the name of the interface:

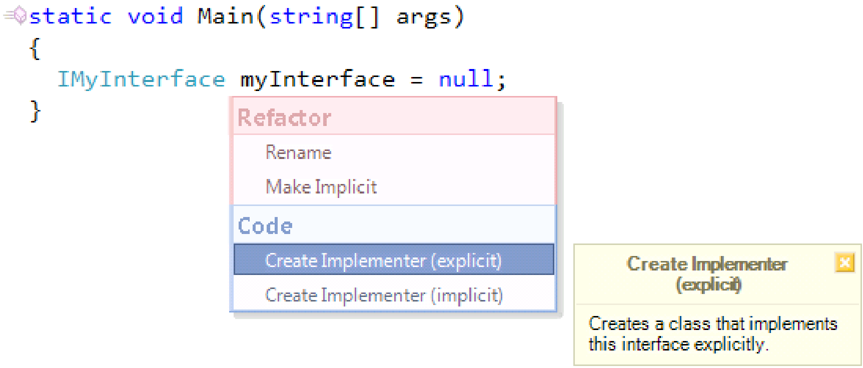


```
public interface IMyInterface
{
  void MyMethod();
  string MyProperty { get; se
  event EventHandler MyEvent;
}
```

Refactor
Rename
Move Type to File
Rename File to Match Type
Code
Create Implementer (explicit)
Create Implementer (implicit)
Create Implementer (implicit)

Creates a class that implements this interface implicitly.

or on the reference to the interface in the code:

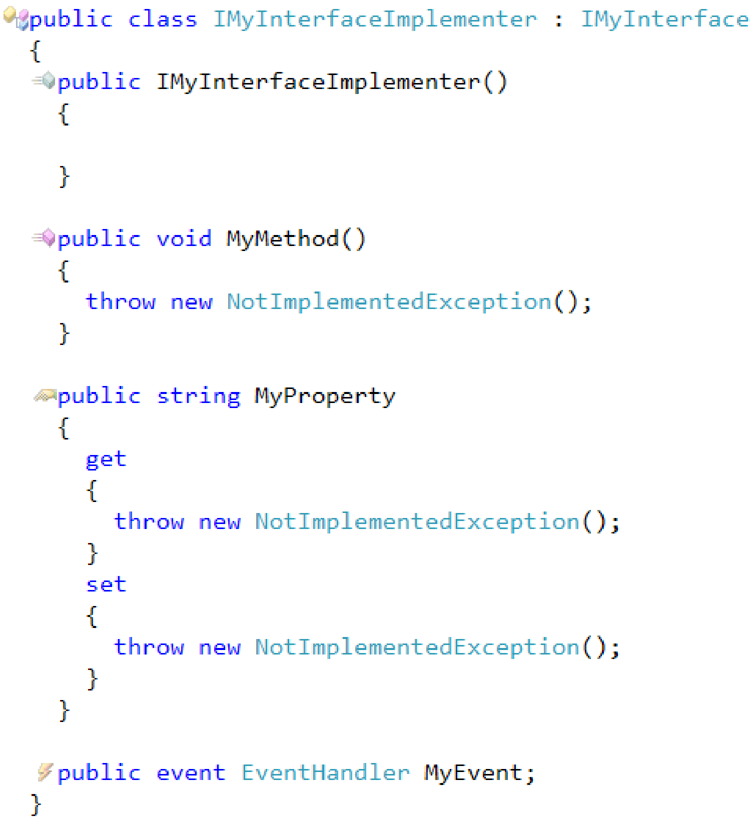


```
static void Main(string[] args)
{
  IMyInterface myInterface = null;
}
```

Refactor
Rename
Make Implicit
Code
Create Implementer (explicit)
Create Implementer (implicit)
Create Implementer (explicit)

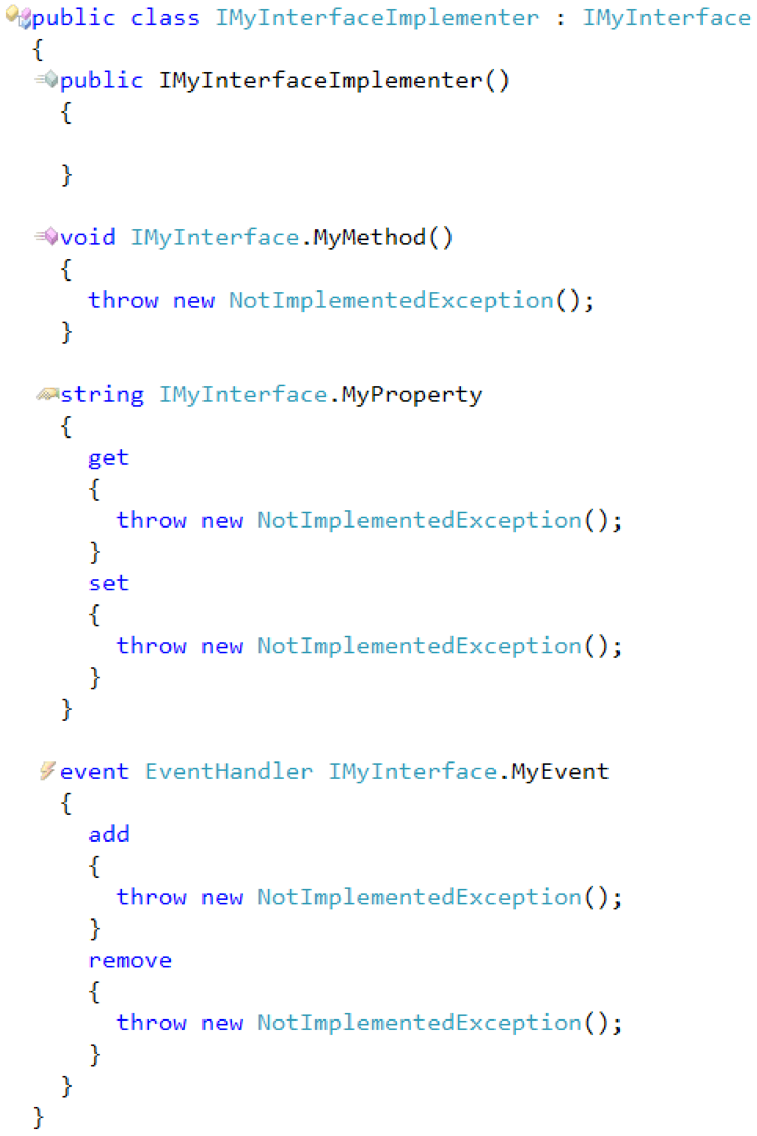
Creates a class that implements this interface explicitly.

The newly declared implementer is placed according to the Type Declarations code style settings. Here is the implementer code after code provider is performed (implicit version):



```
public class IMyInterfaceImplementer : IMyInterface
{
  public IMyInterfaceImplementer()
  {

  }

  public void MyMethod()
  {
    throw new NotImplementedException();
  }

  public string MyProperty
  {
    get
    {
      throw new NotImplementedException();
    }
    set
    {
      throw new NotImplementedException();
    }
  }

  public event EventHandler MyEvent;
}
```

The second version of the code providers does the same as the first version, but implements an interface members explicitly:

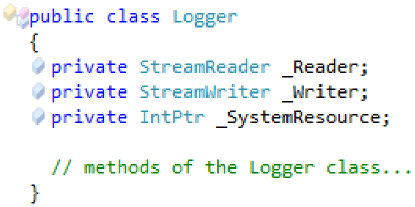


```
public class IMyInterfaceImplementer : IMyInterface
{
  public IMyInterfaceImplementer()
  {

  }

  void IMyInterface.MyMethod()
  {
    throw new NotImplementedException();
  }

  string IMyInterface.MyProperty
  {
    get
    {
      throw new NotImplementedException();
    }
    set
    {
      throw new NotImplementedException();
    }
  }

  event EventHandler IMyInterface.MyEvent
  {
    add
    {
      throw new NotImplementedException();
    }
    remove
    {
      throw new NotImplementedException();
    }
  }
}
```

Implementing the IDisposable pattern

The **IDisposable Pattern** is used to release and close unmanaged resources such as files, streams, and handles as well as references to managed objects that use unmanaged resources, held by an instance of the class. For classes that require a resource cleanup, we recommended following the IDisposable pattern, which provides a standard and unambiguous pattern. The pattern has been designed to ensure reliable, predictable cleanup, and to prevent temporary resource leaks.

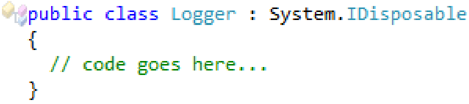
There are several options to consider when implementing the IDisposable pattern. Let’s take a look at the following code sample:



```
public class Logger
  {
    private StreamReader _Reader;
    private StreamWriter _Writer;
    private IntPtr _SystemResource;

    // methods of the Logger class...
  }
```

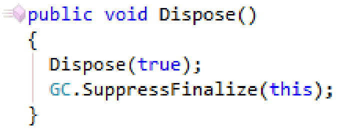
The problem with this class is that StreamReader and StreamWriter objects use unmanaged resources. If this class is instantiated, and the later is allowed to simply go out of scope, the objects may not be cleaned up immediately and resources will be locked unnecessarily.

To implement the IDisposable pattern, we should first specify the corresponding interface as one of those that our class implements:



```
public class Logger : System.IDisposable
  {
    // code goes here...
  }
```

The interface contains only a single Dispose method. This is what the public Dispose() method implementation looks like:



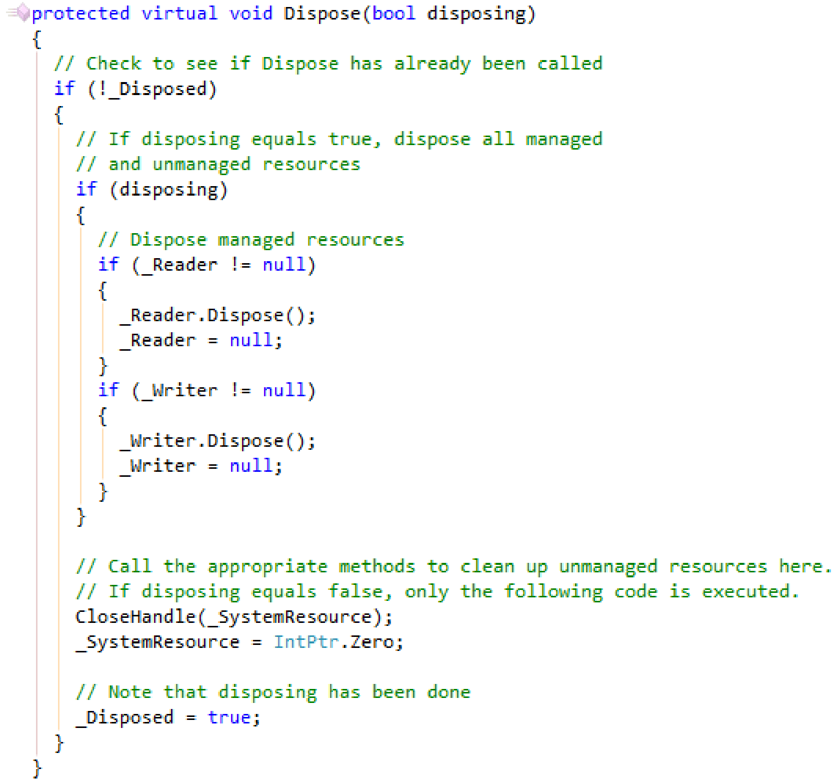
```
public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }
```

The method calls the second parameterized Dispose() method, passing 'true' as an argument to indicate that the method has been called manually. Then, it calls the SuppressFinalize() method of the garbage collector, which prevents the finalizer from executing and limits any associated performance degradation, because finalization increases the cost and duration of your object's lifetime since each finalizable object must be placed on the finalization queue when it is allocated.

The order of two calls inside the Dispose is important, as it ensures that the GC.SupressFinalize()call is only invoked if the Dispose operation is completed successfully. When Dispose calls Dispose(true), the call may fail while an exception is thrown, but when Finalize is called later, it calls Dispose(false). In reality, these are two different calls can execute different portions of the code, even though Dispose(true) fails, Dispose(false) may not.

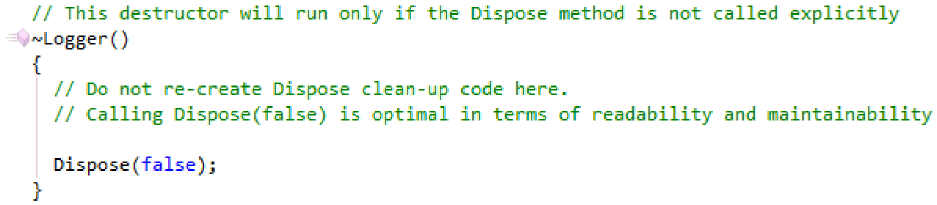
The second version of the Dispose(bool disposing) method accepts a Boolean parameter that indicates whether the code has been called manually or automatically. If called manually, all managed and unmanaged resources can be released. If the method was called automatically via the garbage collection process, the managed resources will already have been dealt with, so only the unmanaged resources are cleared.

When implementing the Dispose method, we must ensure that all held resources are freed by propagating the call through the containment hierarchy. To give the base classes a chance to cleanup resources, it is required to invoke the base class' Dispose(bool disposing) method as the last operation, if one is available. Make sure to pass the same value of 'disposing' to the base class' method.
So, the method implementation for the Logger class looks like this:



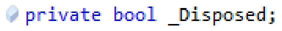
```
protected virtual void Dispose(bool disposing)
{
  // Check to see if Dispose has already been called
  if (!_Disposed)
  {
    // If disposing equals true, dispose all managed
    // and unmanaged resources
    if (disposing)
    {
      // Dispose managed resources
      if (_Reader != null)
      {
        _Reader.Dispose();
        _Reader = null;
      }
      if (_Writer != null)
      {
        _Writer.Dispose();
        _Writer = null;
      }
    }

    // Call the appropriate methods to clean up unmanaged resources here.
    // If disposing equals false, only the following code is executed.
    CloseHandle(_SystemResource);
    _SystemResource = IntPtr.Zero;

    // Note that disposing has been done
    _Disposed = true;
  }
}
```

Because the Dispose method must be called explicitly, objects that implement IDisposable must also add a finalizer, if a class contains unmanaged resources, to handle the release of those resources when Dispose is not called. The finalizer must utilize the Dispose method with the ‘disposing’ parameter set to ‘false’. The false value indicates that the method was not called manually:



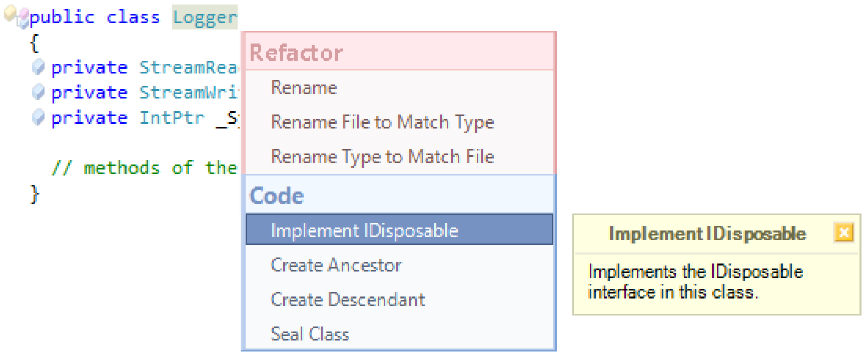
```
  // This destructor will run only if the Dispose method is not called explicitly
~Logger()
{
  // Do not re-create Dispose clean-up code here.
  // Calling Dispose(false) is optimal in terms of readability and maintainability

  Dispose(false);
}
```

Don’t forget to declare the _Disposed field as follows which indicates the state of the object whether it is disposed or not:



```
private bool _Disposed;
```

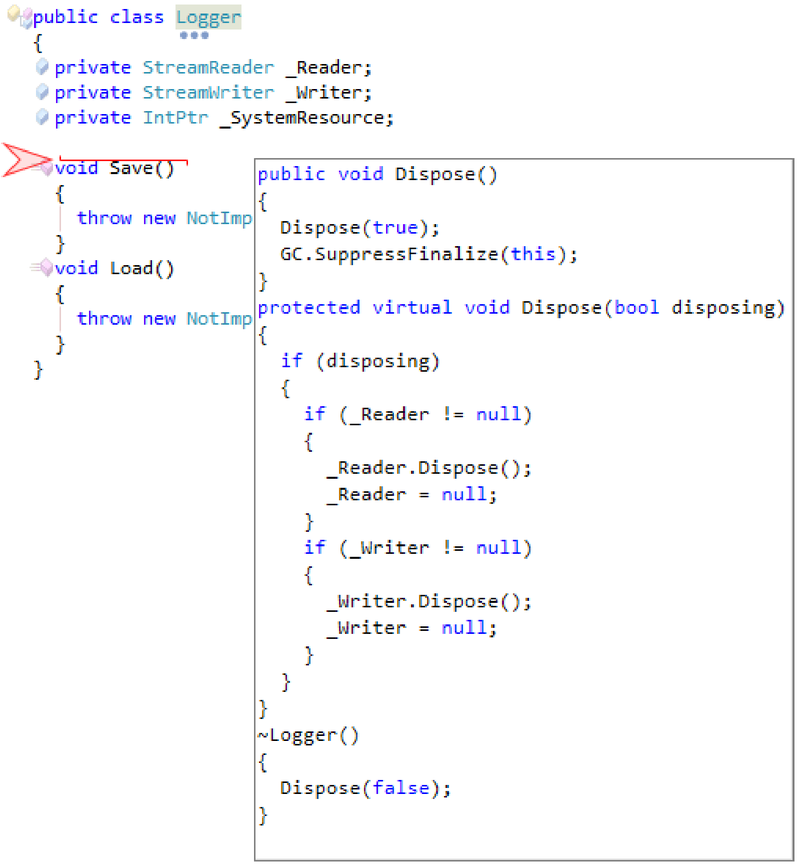
As always, CodeRush provides a couple of code providers that allow you to easily implement the IDisposable pattern without having to remember the entire pattern, using the corresponding **Implement IDisposable** code provider available on the class name:



```
public class Logger
{
  private StreamRea
  private StreamWri
  private IntPtr _S

    // methods of the
}
```

Implements the IDisposable interface in this class.

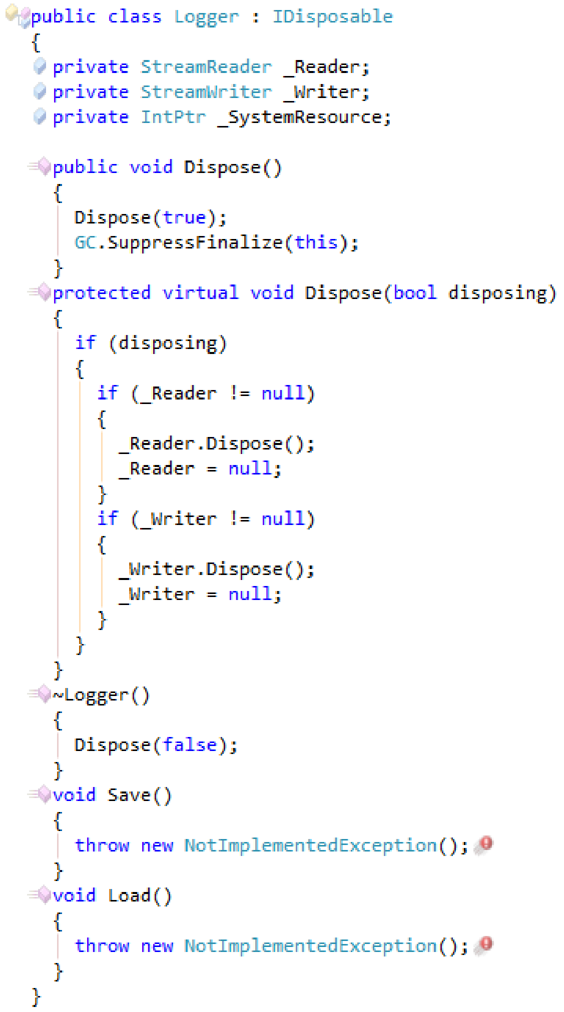
you can automatically generate the required code to release all unmanaged resources. Just choose the position where you would like to put the pattern's code:



```
public class Logger
{
  private StreamReader _Reader;
  private StreamWriter _Writer;
  private IntPtr _SystemResource;

  void Save()
  {
    throw new NotImp
  }
  void Load()
  {
    throw new NotImp
  }
}
```

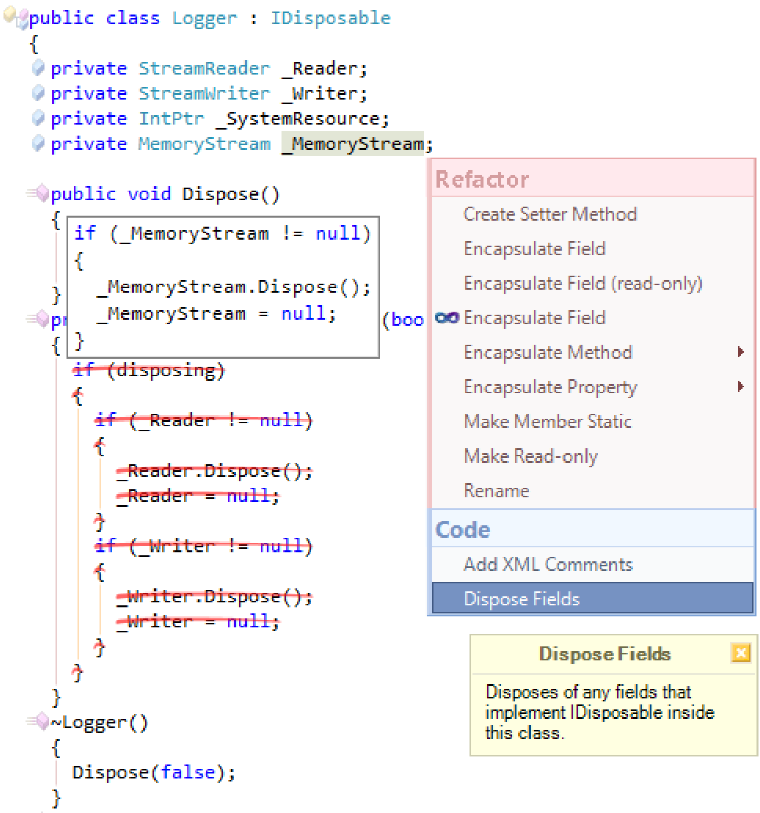
```
public void Dispose()
{
  Dispose(true);
  GC.SuppressFinalize(this);
}
protected virtual void Dispose(bool disposing)
{
  if (disposing)
  {
    if (_Reader != null)
    {
      _Reader.Dispose();
      _Reader = null;
    }
    if (_Writer != null)
    {
      _Writer.Dispose();
      _Writer = null;
    }
  }
}
~Logger()
{
  Dispose(false);
}
```

and you are done:



```
public class Logger : IDisposable
{
    private StreamReader _Reader;
    private StreamWriter _Writer;
    private IntPtr _SystemResource;

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_Reader != null)
            {
                _Reader.Dispose();
                _Reader = null;
            }
            if (_Writer != null)
            {
                _Writer.Dispose();
                _Writer = null;
            }
        }
    }
    ~Logger()
    {
        Dispose(false);
    }
    void Save()
    {
        throw new NotImplementedException();
    }
    void Load()
    {
        throw new NotImplementedException();
    }
}
```

If you later create new fields that should be disposed, you can utilize the **Dispose Fields** code provider:



```
public class Logger : IDisposable
{
  private StreamReader _Reader;
  private StreamWriter _Writer;
  private IntPtr _SystemResource;
  private MemoryStream _MemoryStream;

  public void Dispose()
  {
  }
  pr                                    (boo
  {
    ~~if (disposing)~~
    ~~{~~
      ~~if (_Reader != null)~~
      ~~{~~
        ~~_Reader.Dispose();~~
        ~~_Reader = null;~~
      ~~}~~
      ~~if (_Writer != null)~~
      ~~{~~
        ~~_Writer.Dispose();~~
        ~~_Writer = null;~~
      ~~}~~
    ~~}~~
  }
  ~Logger()
  {
    Dispose(false);
  }
```

```
if (_MemoryStream != null)
{
  _MemoryStream.Dispose();
  _MemoryStream = null;
}
```

Dispose Fields

Disposes of any fields that implement IDisposable inside this class.

As a result, the **Dispose Fields** code provider will add the required disposal code for the specified field as illustrated in the preview hint above.

Using Disposable Objects

Implementing the IDisposable pattern is very important when creating objects that hold disposable resources, but this pattern does not dictate you how to use an instance of the resulting class. Unfortunately, there is no way to force developers to call Dispose explicitly when they are done using it, or to force them to use the proper exception handling mechanism to make sure that the Dispose is called even when an exception is thrown.

So, when you use an object that accesses unmanaged resources, it is a good practice to create the instance with a using statement. The using statement automatically calls the Dispose method on the object when the code that is using it has completed. You can create a using statement by utilizing the Introduce Using Statement code provider. For instance, for this code:

```
Logger logger = new Logger();
logger.Write("Message");
```

the code provider will produce the following code:

```
using (Logger logger = new Logger())
{
  logger.Write("Message");
}
```

Once a class has been disposed, all of its resources should be considered as unreachable. Attempts to continue using the class may be invalid and should be prevented. If the object has been disposed, we can throw an ObjectDisposedException, providing the object name, to indicate the error. This check should be made in all public members of the class.

Converting methods to functions or procedures

The simplest type of method is one that performs a task without requiring any parameters and without returning any information – let's call them procedures (or void methods). Methods that use some type as the return value – are functions.

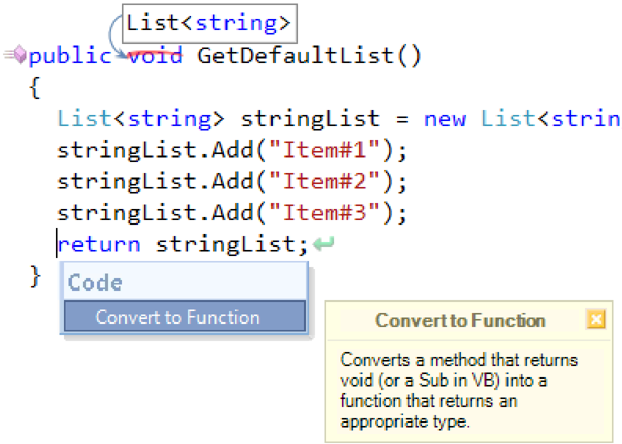
Sometimes, when we declare a method with a return value, we may realize that we don't actually need this return value, and instead, need a simple void method that does not return anything. On the other hand, we marked a method as void, but may require it to be a function, returning a value.

In this case, to help you in converting a void method into a function and vice versa, CodeRush Pro suggests the corresponding code providers:

- **Convert to Function**
- **Convert to Procedure**

Both are opposites of each other. Before you apply them – you can see the resulting preview code.

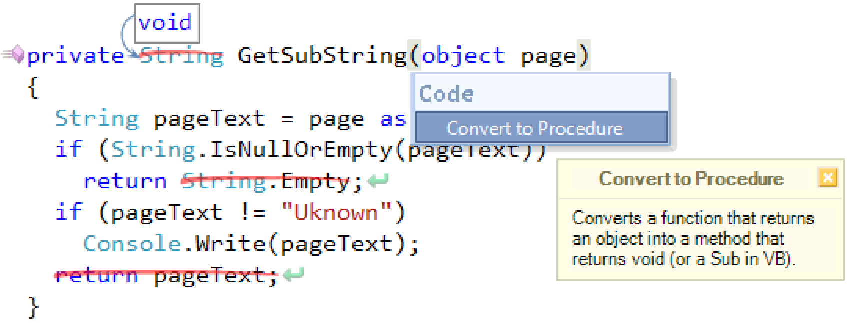
The **Convert to Function** code provider changes a method that returns void (or a Sub in VB) into a function that returns an appropriate type. The type is inferred from the returning value. This code provider will help you easily add a return type for a method and guess its correct type, for example:



```
        List<string>
public void GetDefaultList()
{
  List<string> stringList = new List<string>();
  stringList.Add("Item#1");
  stringList.Add("Item#2");
  stringList.Add("Item#3");
  return stringList;
}
```

Converts a method that returns void (or a Sub in VB) into a function that returns an appropriate type.

The **Convert to Procedure** code provider converts a private function that returns some value into a method that returns void (or a Sub in VB). It will also change all "return something" statements into simple returns, for example:



```
              void
private ~~String~~ GetSubString(object page)
{
  String pageText = page as
  if (String.IsNullOrEmpty(pageText))
    return ~~String~~.Empty;
  if (pageText != "Uknown")
    Console.Write(pageText);
  ~~return pageText;~~
}
```

Convert to Procedure

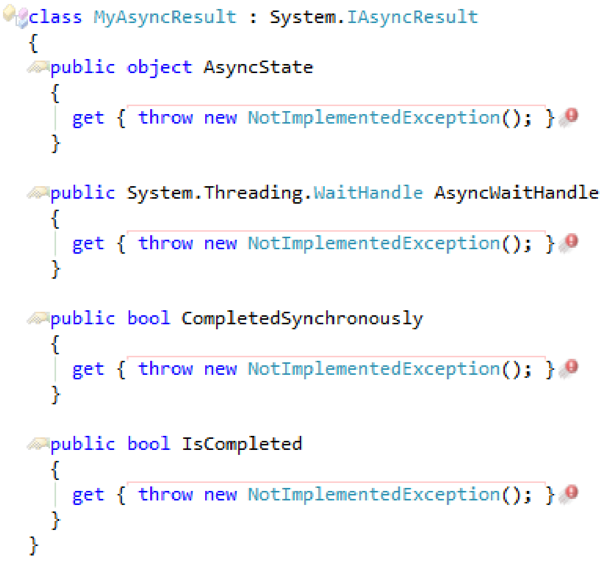
Converts a function that returns an object into a method that returns void (or a Sub in VB).

Converting unimplemented properties

CodeRush has two code providers that allow you to convert unimplemented properties into a property with backing store field or into an auto-implemented property quickly:

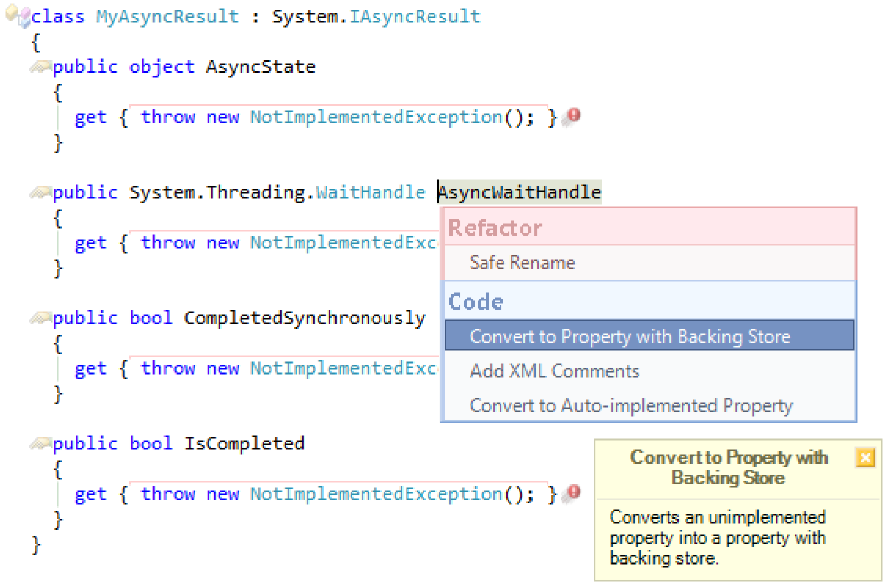
- **Convert to Property with Backing Store**
- **Convert to Auto-implemented Property**

The code providers are useful if you have implemented an interface, for instance:



```
class MyAsyncResult : System.IAsyncResult
{
  public object AsyncState
  {
    get { throw new NotImplementedException(); }
  }

  public System.Threading.WaitHandle AsyncWaitHandle
  {
    get { throw new NotImplementedException(); }
  }

  public bool CompletedSynchronously
  {
    get { throw new NotImplementedException(); }
  }

  public bool IsCompleted
  {
    get { throw new NotImplementedException(); }
  }
}
```

and you are going to provide implementation for properties of the interface:

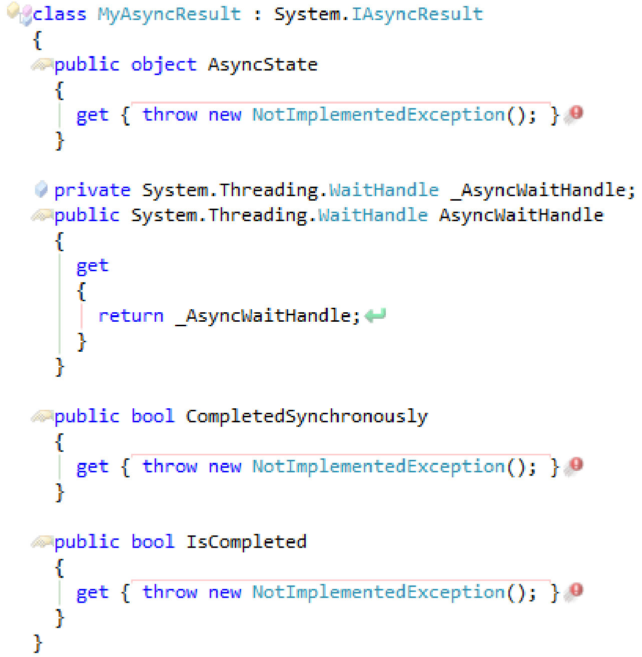


```
class MyAsyncResult : System.IAsyncResult
{
  public object AsyncState
  {
    get { throw new NotImplementedException(); }
  }

  public System.Threading.WaitHandle AsyncWaitHandle
  {
    get { throw new NotImplementedExc
  }

  public bool CompletedSynchronously
  {
    get { throw new NotImplementedExc
  }

  public bool IsCompleted
  {
    get { throw new NotImplementedException(); }
  }
}
```

Refactor
Safe Rename
Code
Convert to Property with Backing Store
Add XML Comments
Convert to Auto-implemented Property
Convert to Property with Backing Store

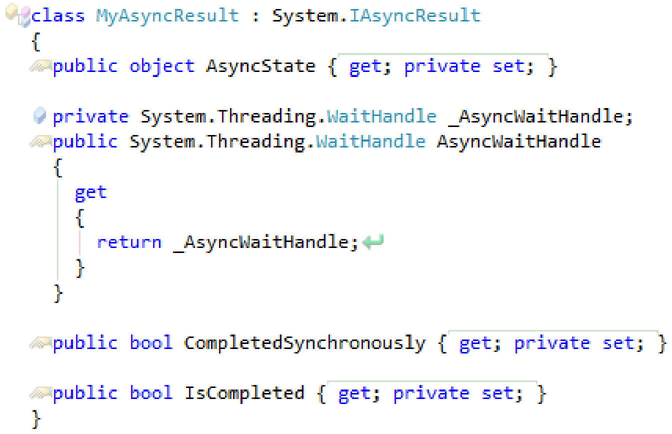
Converts an unimplemented property into a property with backing store.

Choose the **Convert to Property with Backing Store** code provider if you'd like to create a backing store field and remove it from the property:



```
class MyAsyncResult : System.IAsyncResult
{
  public object AsyncState
  {
    get { throw new NotImplementedException(); }
  }

  private System.Threading.WaitHandle _AsyncWaitHandle;
  public System.Threading.WaitHandle AsyncWaitHandle
  {
    get
    {
      return _AsyncWaitHandle;
    }
  }

  public bool CompletedSynchronously
  {
    get { throw new NotImplementedException(); }
  }

  public bool IsCompleted
  {
    get { throw new NotImplementedException(); }
  }
}
```

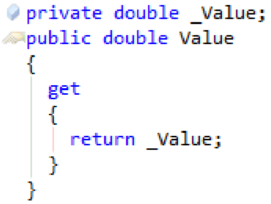
Choose the **Convert to Auto-implemented Property** code provider if you do not need a backing store field and would like to have an auto-implemented property instead. This is the resulting code after the code provider is applied on three other properties:



```
class MyAsyncResult : System.IAsyncResult
{
  public object AsyncState { get; private set; }

  private System.Threading.WaitHandle _AsyncWaitHandle;
  public System.Threading.WaitHandle AsyncWaitHandle
  {
    get
    {
      return _AsyncWaitHandle;
    }
  }

  public bool CompletedSynchronously { get; private set; }

  public bool IsCompleted { get; private set; }
}
```

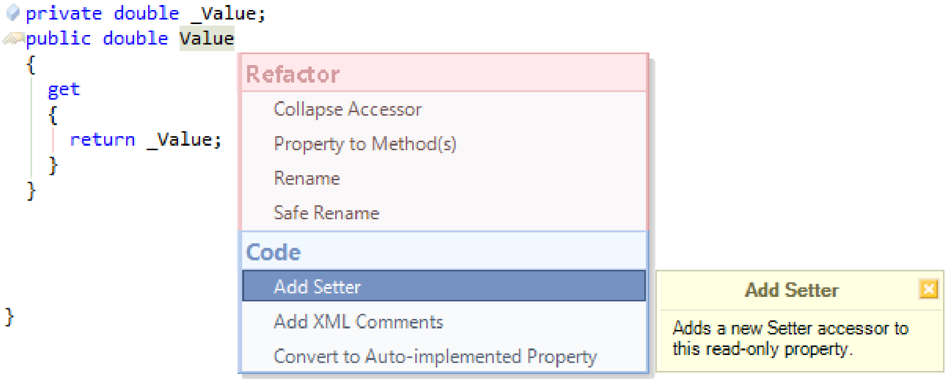
Quick ways to add and remove a set/get property accessor

A property without a ‘set’ accessor is considered read-only:



```
private double _Value;
public double Value
{
  get
  {
    return _Value;
  }
}
```

You can not assign a value to this property until you have a ‘set’ accessor to this property. The set accessor is similar to a void method with an implicit parameter called value, whose type is the type of the property. If a property does not yet have a set accessor, you can easily add it using the **Add Setter** code provider:

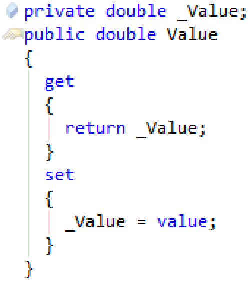


```
private double _Value;
public double Value
{
  get
  {
    return _Value;
  }
}


}
```

Add Setter

Adds a new Setter accessor to this read-only property.

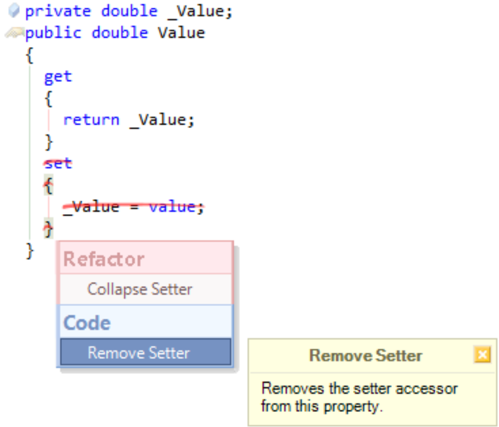
As a result, a property has a read-write access:



```
private double _Value;
public double Value
{
  get
  {
    return _Value;
  }
  set
  {
    _Value = value;
  }
}
```

The code provider will automatically determine the field reference used as a backing store and assign a value to it.

You can also perform the opposite operation – convert a read-write property to a read-only property by using the corresponding **Remove Setter** code provider. The code provider is available when a value is never assigned to a property. In other words, the code provider safely removes the set accessor if it is not referenced in the code:



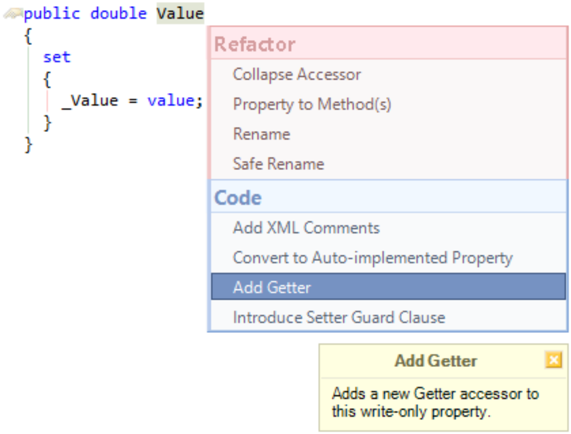
```
private double _Value;
public double Value
{
  get
  {
    return _Value;
  }
  set
  {
    ~~_Value = value;~~
  }
}
```

Remove Setter

Removes the setter accessor from this property.

The latter code provider is useful to refactor and cleanup your code, eliminating undesirable property use.

If the property is write-only, having only the setter accessor, you can use the **Add Getter** code provider, which is similar to the **Add Setter**, but adds the getter accessor instead:



```
public double Value
{
  set
  {
    _Value = value;
  }
}
```

Add Getter

Adds a new Getter accessor to this write-only property.

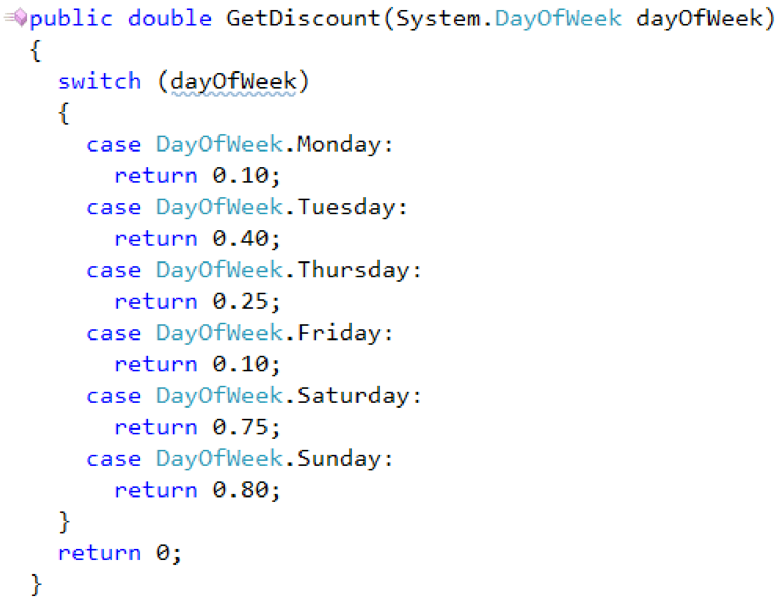
Code fixes for code issues for switch (select) statements

You might have already learned that CodeRush suggests several code issues that highlight the switch (Select in VB) statement with hints and warnings when it has suspicious code. For example, when the switch statement handles only a subset of the possible enumeration values it is checking for, this may be a sign of incomplete code.

Until recently, there were no code fixes of those code issues. Now, they appear:

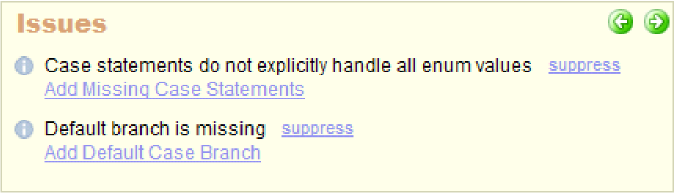
- **Add Missing Case Statements** – adds all missing case statements to the switch statement.
- **Add Default Case Branch** – adds the default case branch to the switch statement.

Consider the following code:



```
public double GetDiscount(System.DayOfWeek dayOfWeek)
{
  switch (dayOfWeek)
  {
    case DayOfWeek.Monday:
      return 0.10;
    case DayOfWeek.Tuesday:
      return 0.40;
    case DayOfWeek.Thursday:
      return 0.25;
    case DayOfWeek.Friday:
      return 0.10;
    case DayOfWeek.Saturday:
      return 0.75;
    case DayOfWeek.Sunday:
      return 0.80;
  }
  return 0;
}
```

The DayOfWeek enumeration contain seven enumeration elements for each day of a week. In the code above one of the days is missing (Wednesday). The expression of the switch statement is highlighted with a code issue of the suggestion type, and the following hint appears if you hover over it:



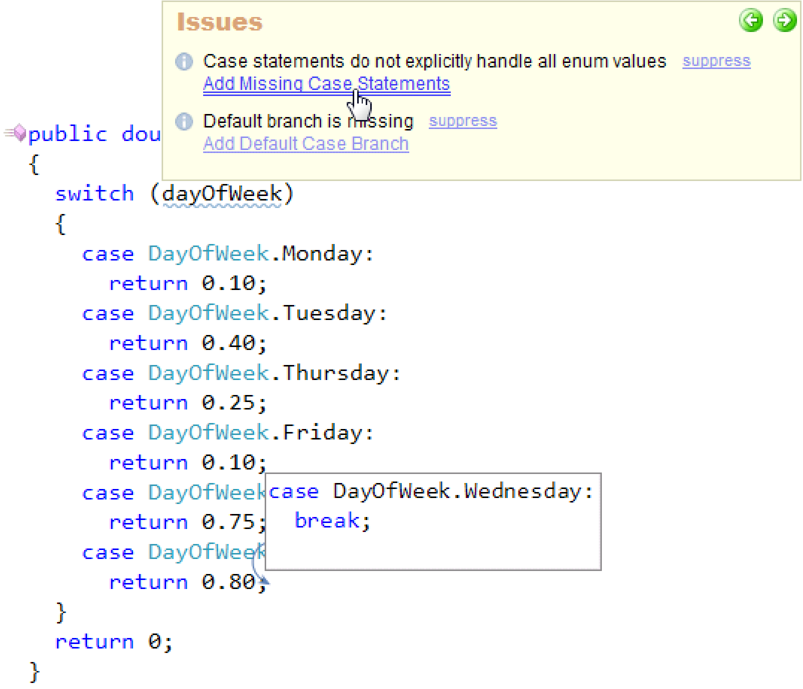
Issues

Case statements do not explicitly handle all enum values suppress
Add Missing Case Statements

Default branch is missing suppress
Add Default Case Branch

There are two code issues listed in the hint and both have the corresponding fixes which can be seen in the preview hint before applying:

Add Missing Case Statements



Issues

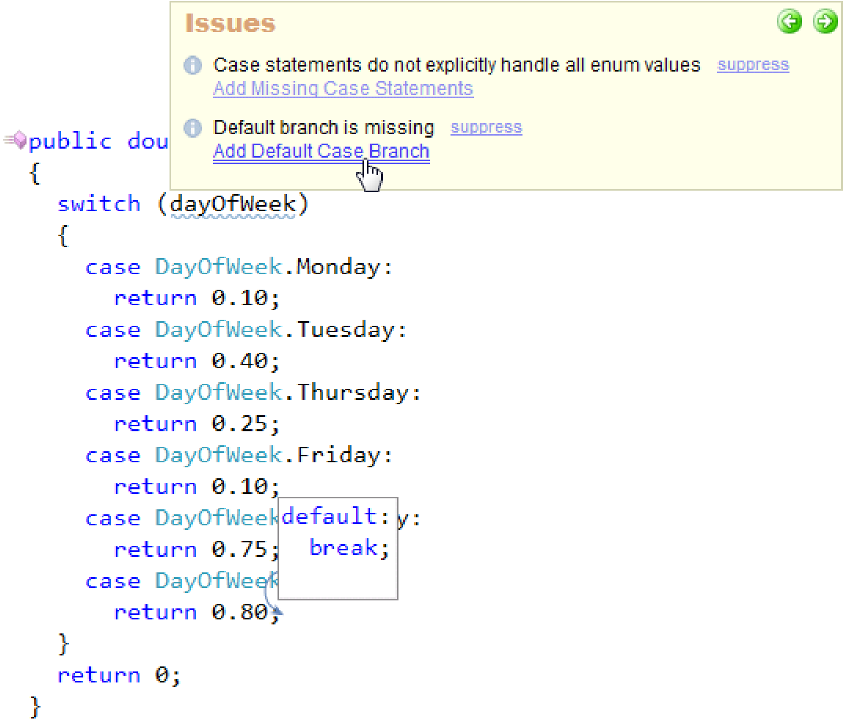
Case statements do not explicitly handle all enum values suppress
Add Missing Case Statements

Default branch is missing suppress
Add Default Case Branch

```
public dou
{
  switch (dayOfWeek)
  {
    case DayOfWeek.Monday:
      return 0.10;
    case DayOfWeek.Tuesday:
      return 0.40;
    case DayOfWeek.Thursday:
      return 0.25;
    case DayOfWeek.Friday:
      return 0.10;
    case DayOfWeek
      return 0.75;
    case DayOfWeek
      return 0.80;
  }
  return 0;
}
```

```
case DayOfWeek.Wednesday:
  break;
```

Add Default Case Branch



Issues

Case statements do not explicitly handle all enum values suppress
Add Missing Case Statements

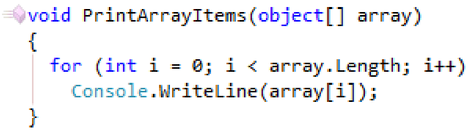
Default branch is missing suppress
Add Default Case Branch

```
public dou
{
  switch (dayOfWeek)
  {
    case DayOfWeek.Monday:
      return 0.10;
    case DayOfWeek.Tuesday:
      return 0.40;
    case DayOfWeek.Thursday:
      return 0.25;
    case DayOfWeek.Friday:
      return 0.10;
    case DayOfWeek        y:
      return 0.75;
    case DayOfWeek
      return 0.80;
  }
  return 0;
}
```

```
default:
  break;
```

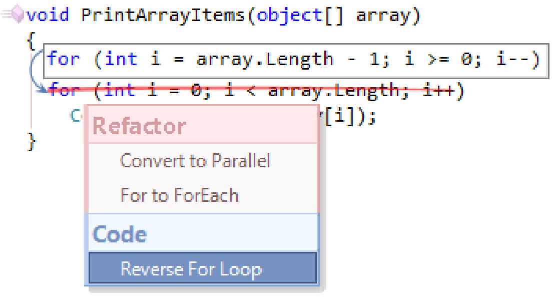
Reverse For Loop

When working with arrays and lists enumerating all of its items we usually create a for loop statement as follows:



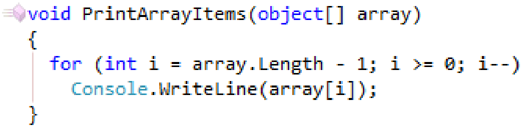
```
void PrintArrayItems(object[] array)
{
  for (int i = 0; i < array.Length; i++)
    Console.WriteLine(array[i]);
}
```

But what if we want array items to be listed in reverse? Sometimes, we want to create a loop that starts from a larger number and decrements the iterator variable's value after each iteration until zero is reached. In this case, we can easily reverse the loop using the **Reverse For Loop** code provider:



```
void PrintArrayItems(object[] array)
{
  for (int i = array.Length - 1; i >= 0; i--)
  ~~for (int i = 0; i < array.Length; i++)~~
    C                     [i]);
}
```

Once it is applied, it will change the order in which items are enumerated in the loop as illustrated in the preview hint:

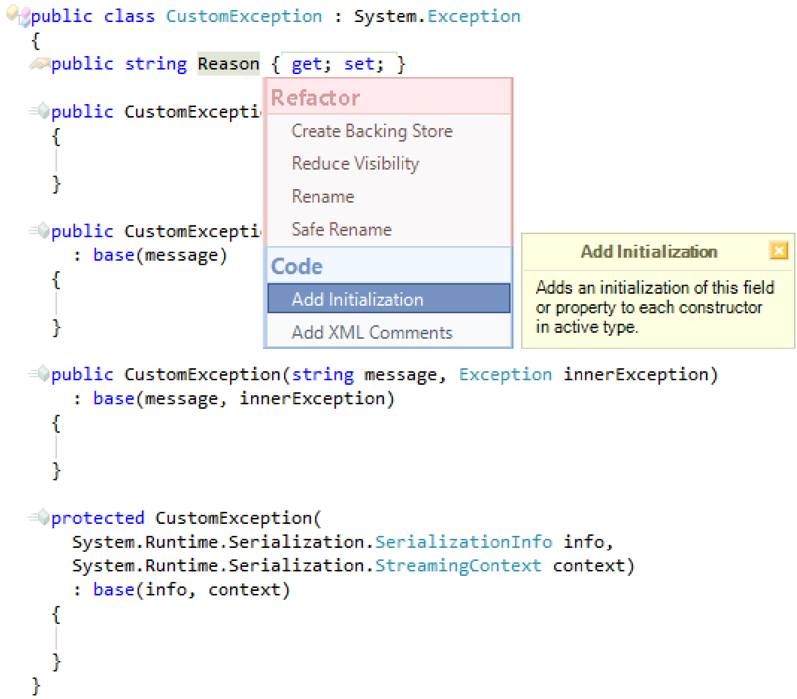


```
void PrintArrayItems(object[] array)
{
  for (int i = array.Length - 1; i >= 0; i--)
    Console.WriteLine(array[i]);
}
```

The code provider can invert loops written in different kinds of ways in any supported programming language.

Add Initialization

The **Add Initialization** code provider inserts an initialization of the selected field or auto-implemented property to each constructor of the active type. The code provider is useful when you want to initialize those objects with a different default value:

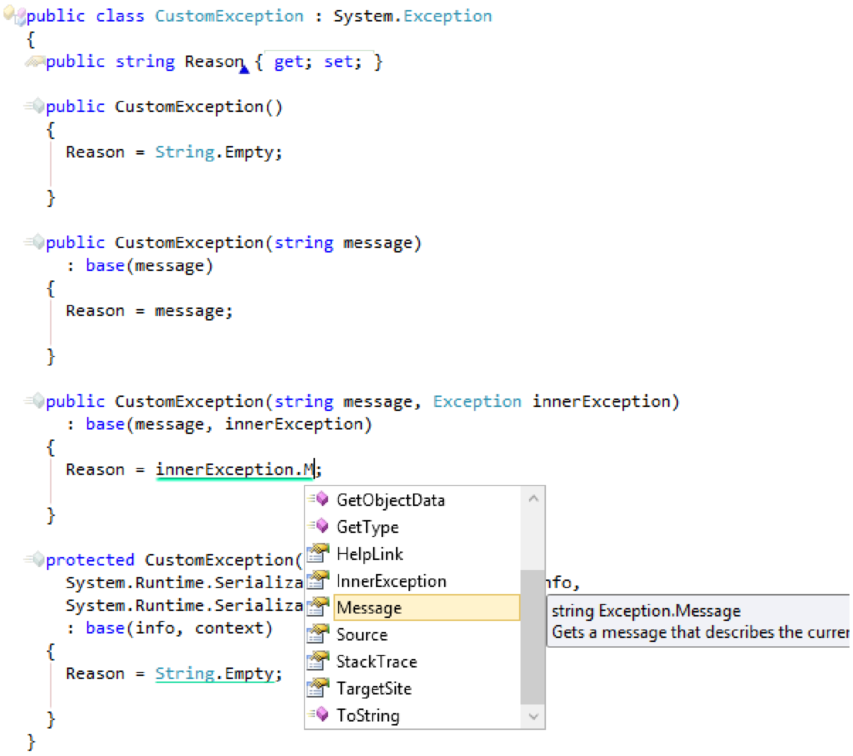


```
public class CustomException : System.Exception
{
  public string Reason { get; set; }

  public CustomExcepti
  {

  }

  public CustomExcepti
    : base(message)
  {

  }

  public CustomException(string message, Exception innerException)
    : base(message, innerException)
  {

  }

  protected CustomException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base(info, context)
  {

  }
}
```

Add Initialization

Adds an initialization of this field or property to each constructor in active type.

Once the code provider is applied, you will get the following initialization generated in each constructor:



```
public class CustomException : System.Exception
{
  public string Reason { get; set; }

  public CustomException()
  {
    Reason = String.Empty;

  }

  public CustomException(string message)
    : base(message)
  {
    Reason = message;

  }

  public CustomException(string message, Exception innerException)
    : base(message, innerException)
  {
    Reason = innerException.M;

  }

  protected CustomException(
    System.Runtime.Serializa                                   nfo,
    System.Runtime.Serializa
    : base(info, context)
  {
    Reason = String.Empty;

  }
}
```

string Exception.Message
Gets a message that describes the currer

Changing an explicit cast-expression to an 'as' operator and back

Objects can be converted from one type to another, assuming that the types are compatible. Often this is achieved using implicit conversion or explicitly with the cast operator. To perform an explicit casting, there are two approaches used:

- a cast-expression
- an 'as' operator

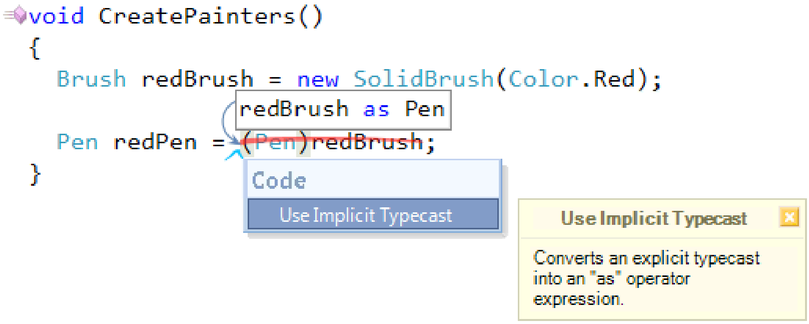
A cast-expression is used to explicitly convert an expression to a given type. The cast operation forces the conversion, possibly losing some data along the way. Care must be taken, as it is possible to cause a run-time error, or receive unpredictable results when the destination casting type is too small to receive a large value of the cast expression.

The 'as' operator is used to explicitly convert a value to a given reference type or nullable type. Unlike the cast expression, the 'as' operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is null.

To convert between explicit type cast approaches, you can use two code providers shipped in CodeRush Pro:

Use Implicit type cast

Converts an explicit type cast expression into an 'as' operator expression:



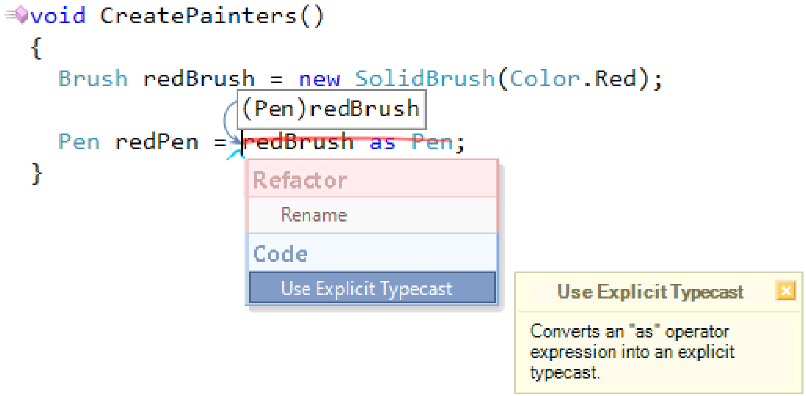
```
void CreatePainters()
{
  Brush redBrush = new SolidBrush(Color.Red);
                redBrush as Pen
  Pen redPen = (Pen)redBrush;
}
```

Use Implicit Typecast

Converts an explicit typecast into an "as" operator expression.

Use Explicit type cast

Converts an 'as' operator expression into an explicit type cast expression:



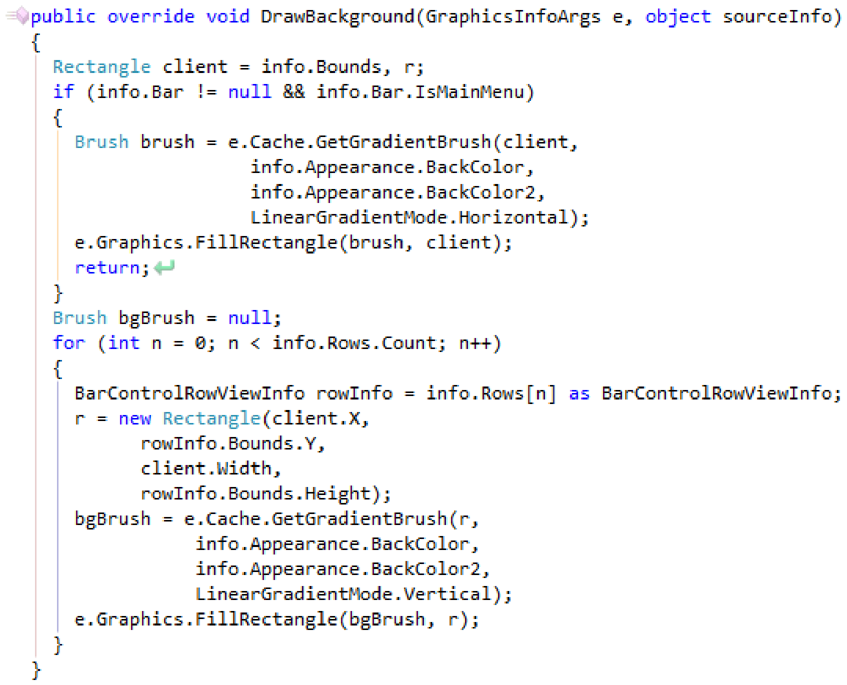
```
void CreatePainters()
{
  Brush redBrush = new SolidBrush(Color.Red);
                (Pen)redBrush
  Pen redPen = redBrush as Pen;
}
```

Use Explicit Typecast

Converts an "as" operator expression into an explicit typecast.

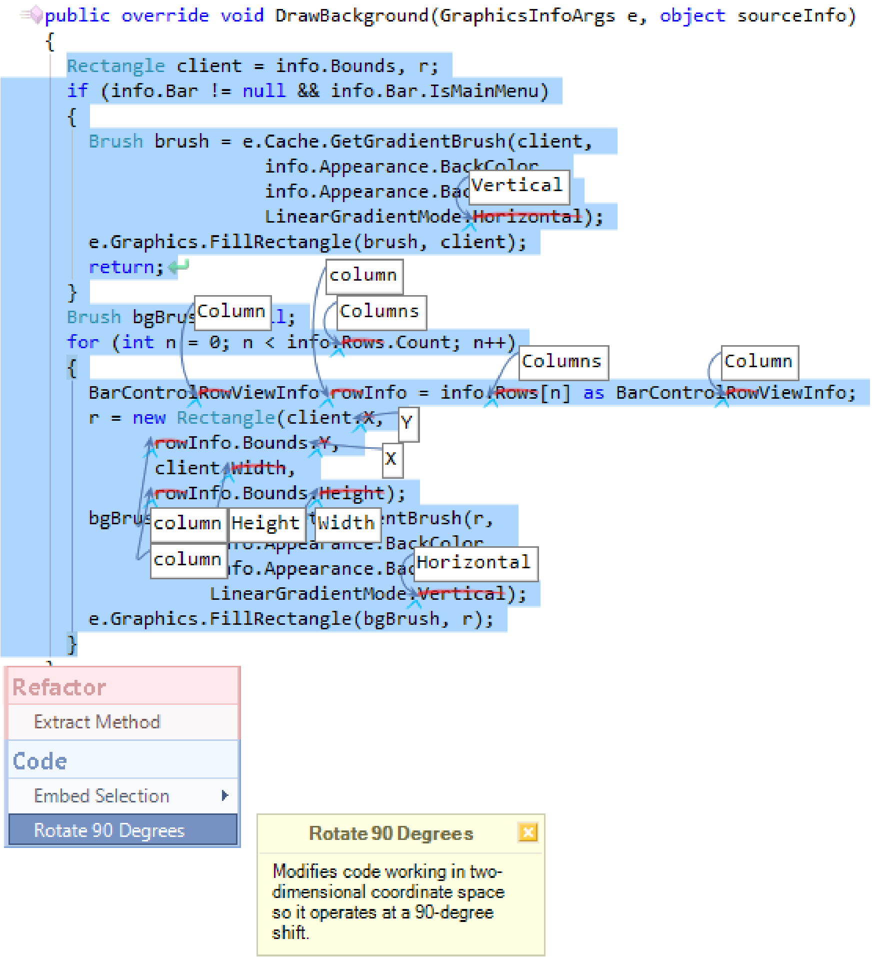
Working in two-dimensional space

There are two code providers that are useful when you have written the code to work in one dimension, but if you want to create a similar code to work in a two-dimensional space, here is an example:



```
public override void DrawBackground(GraphicsInfoArgs e, object sourceInfo)
{
  Rectangle client = info.Bounds, r;
  if (info.Bar != null && info.Bar.IsMainMenu)
  {
    Brush brush = e.Cache.GetGradientBrush(client,
                    info.Appearance.BackColor,
                    info.Appearance.BackColor2,
                    LinearGradientMode.Horizontal);
    e.Graphics.FillRectangle(brush, client);
    return;
  }
  Brush bgBrush = null;
  for (int n = 0; n < info.Rows.Count; n++)
  {
    BarControlRowViewInfo rowInfo = info.Rows[n] as BarControlRowViewInfo;
    r = new Rectangle(client.X,
          rowInfo.Bounds.Y,
          client.Width,
          rowInfo.Bounds.Height);
    bgBrush = e.Cache.GetGradientBrush(r,
               info.Appearance.BackColor,
               info.Appearance.BackColor2,
               LinearGradientMode.Vertical);
    e.Graphics.FillRectangle(bgBrush, r);
  }
}
```

The code is working with rows of a grid and draws the background for them. Consider working with columns instead. Here is what the **Rotate 90 Degrees** code provider will produce:



```
public override void DrawBackground(GraphicsInfoArgs e, object sourceInfo)
{
  Rectangle client = info.Bounds, r;
  if (info.Bar != null && info.Bar.IsMainMenu)
  {
    Brush brush = e.Cache.GetGradientBrush(client,
                    info.Appearance.BackColor,
                    info.Appearance.Ba
                    LinearGradientMode.~~Horizontal~~);
    e.Graphics.FillRectangle(brush, client);
    return;
  }
  Brush bgBrus                l;
  for (int n = 0; n < info.~~Rows~~.Count; n++)
  {
    BarControl~~Row~~ViewInfo ~~row~~Info = info.~~Rows~~[n] as BarControl~~Row~~ViewInfo;
    r = new Rectangle(client.~~X~~,
          ~~row~~Info.Bounds.~~Y~~,
          client.~~Width~~,
          ~~row~~Info.Bounds.~~Height~~);
    bgBrus                     ntBrush(r,
                                .BackColor
                fo.Appearance.Ba
                LinearGradientMode.~~Vertical~~);
    e.Graphics.FillRectangle(bgBrush, r);
  }
```

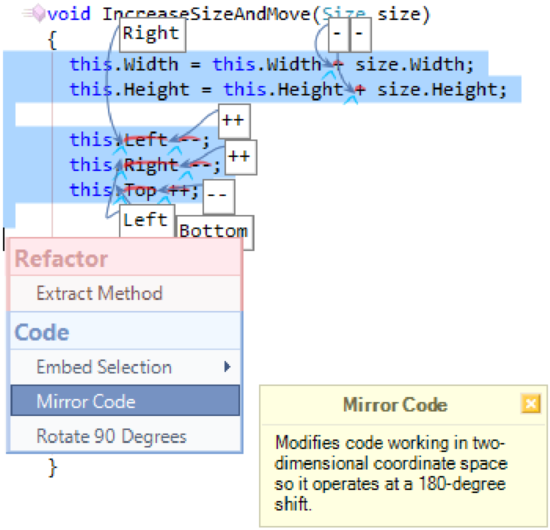
Rotate 90 Degrees

Modifies code working in two-dimensional coordinate space so it operates at a 90-degree shift.

The code provider modifies the code working in two-dimensional coordinate space, so it operates at a 90-degree shift. In other words, it changes the corresponding properties into neighborhood properties, such as:

- X -> Y, Y -> X;
- Height -> Width, Width -> Height;
- Horizontal -> Vertical, Vertical -> Horizontal;
- Left -> Top, Top -> Left, Right -> Bottom, Bottom -> Right;
- Column->Row, Row-> Column, Rows->Columns, Columns->Rows.

The **Mirror Code** code provider, in its turn, modifies the code working in two-dimensional coordinate space, so it operates at a -۱۸۰degree shift:



```
void IncreaseSizeAndMove(Size size)
{
    this.Width = this.Width + size.Width;
    this.Height = this.Height + size.Height;

    this.Left--;
    this.Right--;
    this.Top++;
}
```

Mirror Code

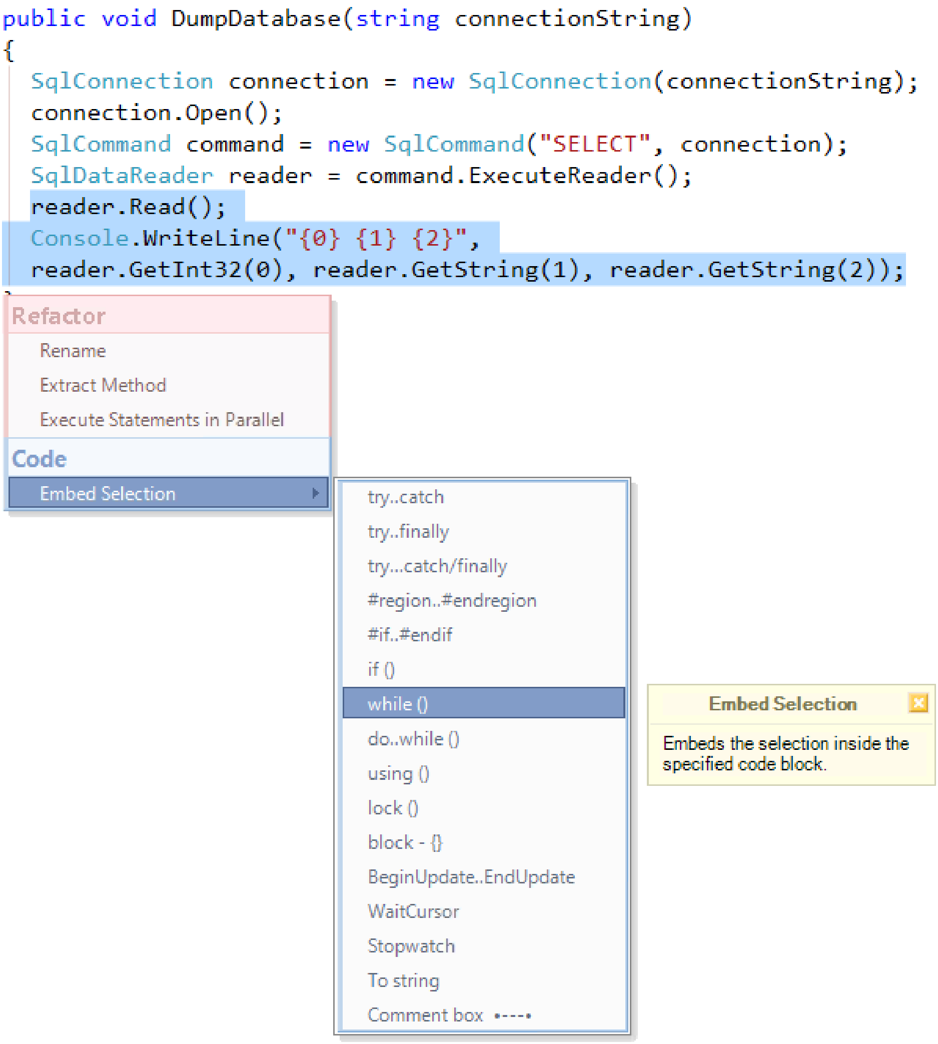
Modifies code working in two-dimensional coordinate space so it operates at a 180-degree shift.

The code providers changes + to -, ++ to — as well as – to + and — to ++. It also converts Left to Right, Top to Bottom the same way as the **Rotate 90 Degrees** code provider does.

Embed Selection

The **Embed Selection** code provider allows you to wrap the selection into the predefined code blocks from the Code Embedding feature of CodeRush. The code provider is available when a whole line or multiple lines are selected and at least one embedding is predefined for the current language.

To apply the code provider, select the text or a code block in the code editor, and choose one of the available embeddings from the **Embed Selection** menu:

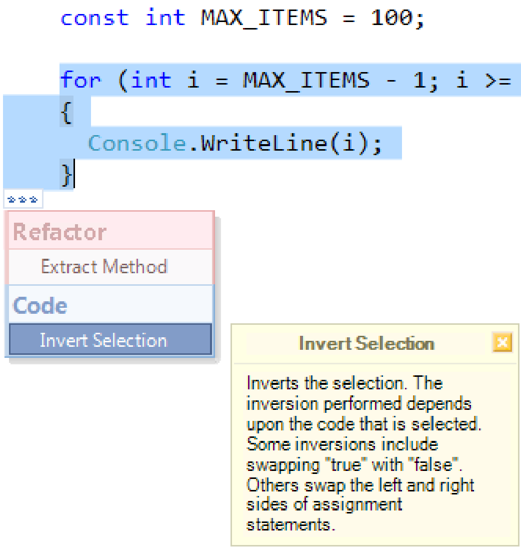


```
public void DumpDatabase(string connectionString)
{
  SqlConnection connection = new SqlConnection(connectionString);
  connection.Open();
  SqlCommand command = new SqlCommand("SELECT", connection);
  SqlDataReader reader = command.ExecuteReader();
  reader.Read();
  Console.WriteLine("{0} {1} {2}",
  reader.GetInt32(0), reader.GetString(1), reader.GetString(2));
```

Embeds the selection inside the specified code block.

Invert Selection

The **Invert Selection** code provider is used to apply the Selection Inversion feature of CodeRush. Once you select a block of text and there is a match with one of the predefined inversion items, the provider becomes available:



```
const int MAX_ITEMS = 100;

for (int i = MAX_ITEMS - 1; i >= 0; i--)
{
  Console.WriteLine(i);
}
```

Invert Selection

Inverts the selection. The inversion performed depends upon the code that is selected. Some inversions include swapping "true" with "false". Others swap the left and right sides of assignment statements.

After it is applied, the selected block is replaced with its opposite or any other block you manually created on the Inversion options page.

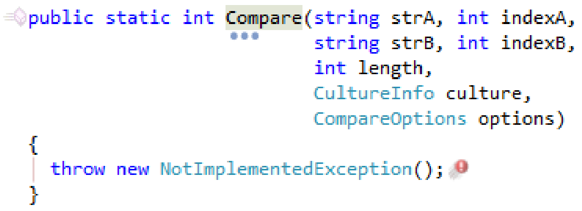
Add XML Comments

Programming languages such as C# and Visual Basic support the creation of XML Documentation Comments, allowing developers to quickly annotate and document their source code to keep the documentation in a standard format and to gain benefit from the information as you code. Documentation is important to ensure that developers can quickly learn and use the source code.

Developers may use XML Comments to document code on user-defined types, including classes, structures, delegates, and enumerations. In addition, XML Comments may be attached to members, including fields, events, properties, indexers, and methods. In other words, these are specially formatted comments that decorate elements of your code. When you build your project with the corresponding option enabled, the XML Comments are extracted and combined in an XML file that resides in the same folder as the compiled application.

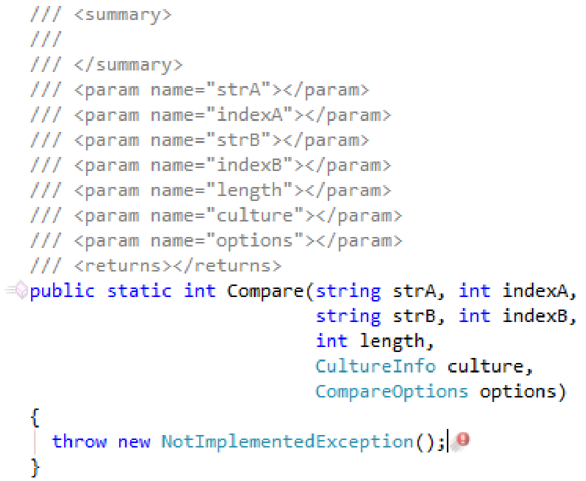
Documenting the source code should be an important part of the development process. If you get in the habit of documenting your source code as you write it, you will find that you can produce fully documented code much faster than if you write code and try and go back and create documentation later.

To help you document the source code, CodeRush provides the **Add XML Comments** code provider which may be very helpful for these purposes, if, for instance, you apply it to an existing method:



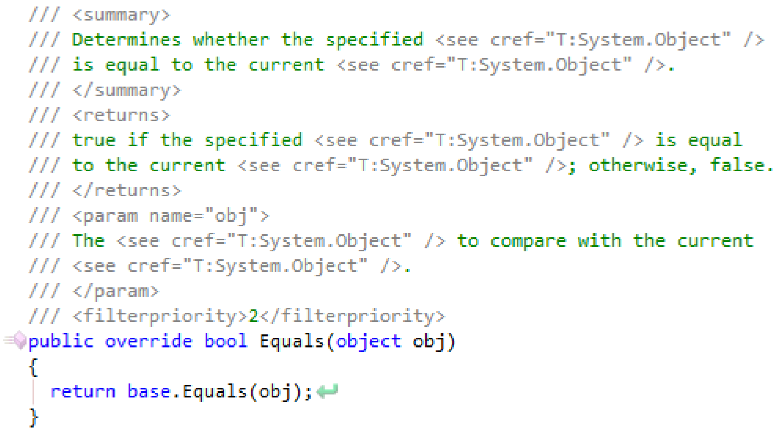
```
public static int Compare(string strA, int indexA,
                          string strB, int indexB,
                          int length,
                          CultureInfo culture,
                          CompareOptions options)
{
  throw new NotImplementedException();
}
```

The code provider will generate the XML Documentation with summary, param, and return value tags as follows:



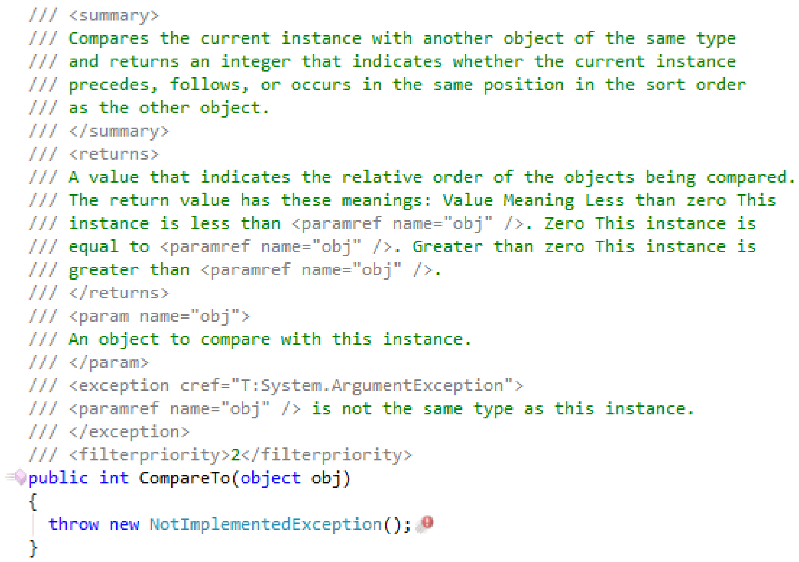
```
/// <summary>
///
/// </summary>
/// <param name="strA"></param>
/// <param name="indexA"></param>
/// <param name="strB"></param>
/// <param name="indexB"></param>
/// <param name="length"></param>
/// <param name="culture"></param>
/// <param name="options"></param>
/// <returns></returns>
public static int Compare(string strA, int indexA,
                          string strB, int indexB,
                          int length,
                          CultureInfo culture,
                          CompareOptions options)
{
    throw new NotImplementedException();
}
```

On the overridden members, the code provider will first look for the base member in the ancestor class. If the base member already provides XML documentation, the code provider will copy it to the member that overrides it:



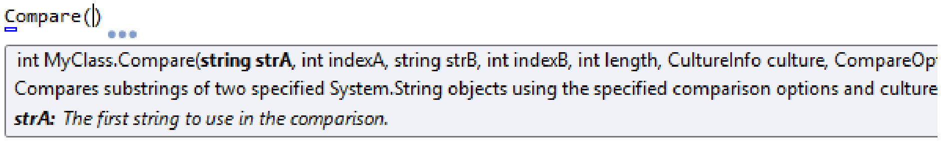
```
/// <summary>
/// Determines whether the specified <see cref="T:System.Object" />
/// is equal to the current <see cref="T:System.Object" />.
/// </summary>
/// <returns>
/// true if the specified <see cref="T:System.Object" /> is equal
/// to the current <see cref="T:System.Object" />; otherwise, false.
/// </returns>
/// <param name="obj">
/// The <see cref="T:System.Object" /> to compare with the current
/// <see cref="T:System.Object" />.
/// </param>
/// <filterpriority>2</filterpriority>
public override bool Equals(object obj)
{
    return base.Equals(obj);
}
```

Additionally, if a member implements the interface member from a referenced assembly, and the latter one provides the documentation in the corresponding XML file, the code provider will load it from the XML file for you, e.g.:



```
/// <summary>
/// Compares the current instance with another object of the same type
/// and returns an integer that indicates whether the current instance
/// precedes, follows, or occurs in the same position in the sort order
/// as the other object.
/// </summary>
/// <returns>
/// A value that indicates the relative order of the objects being compared.
/// The return value has these meanings: Value Meaning Less than zero This
/// instance is less than <paramref name="obj" />. Zero This instance is
/// equal to <paramref name="obj" />. Greater than zero This instance is
/// greater than <paramref name="obj" />.
/// </returns>
/// <param name="obj">
/// An object to compare with this instance.
/// </param>
/// <exception cref="T:System.ArgumentException">
/// <paramref name="obj" /> is not the same type as this instance.
/// </exception>
/// <filterpriority>2</filterpriority>
public int CompareTo(object obj)
{
  throw new NotImplementedException();
}
```

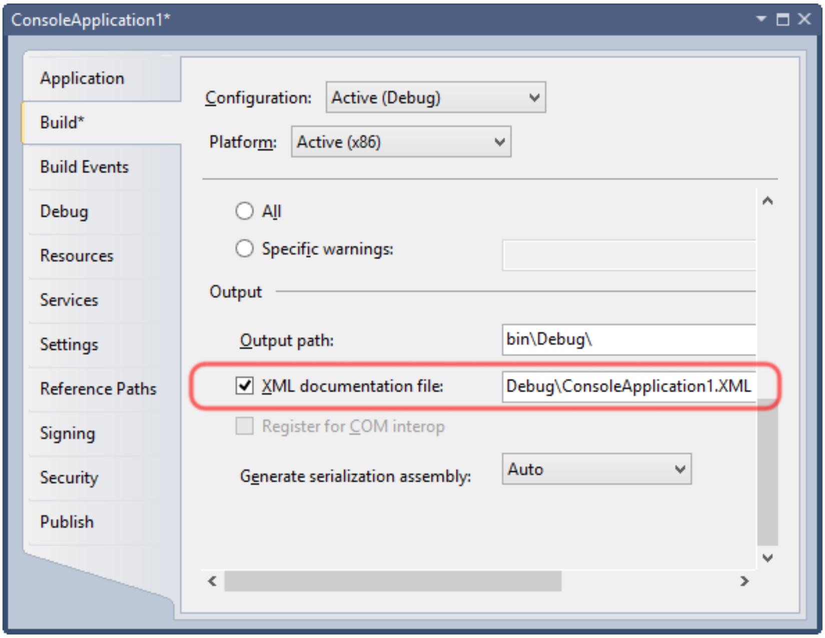
As a result, when you type this method call, you will see a nice hint with the corresponding documentation:



```
Compare(|)
```

int MyClass.Compare(**string strA**, int indexA, string strB, int indexB, int length, CultureInfo culture, CompareOp
Compares substrings of two specified System.String objects using the specified comparison options and culture
strA: *The first string to use in the comparison.*

Don’t forget to enable the appropriate option to generate the XML documentation file when you build the project in the project properties window:



Configuration: Active (Debug)
Platform: Active (x86)

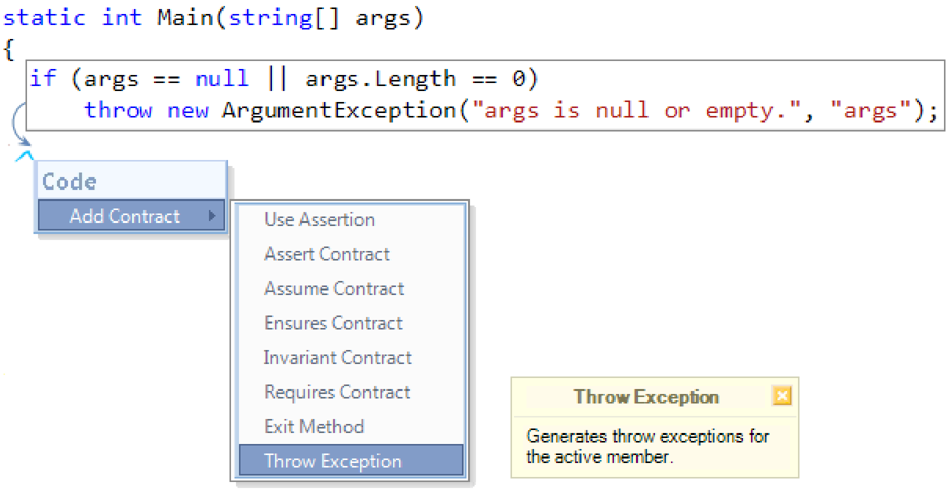
○ All
○ Specific warnings:

Output

Output path: bin\Debug\
☑ XML documentation file: Debug\ConsoleApplication1.XML
□ Register for COM interop
Generate serialization assembly: Auto

Add Contract

Add Contract code provider adds conditions (contracts or guard code) for a validation of the active method parameters. You can choose the preferred contract via the sub-menu of the **Add Contract**:



```
static int Main(string[] args)
{
 if (args == null || args.Length == 0)
     throw new ArgumentException("args is null or empty.", "args");
```

Contracts represent some kind of obligations for the method parameters and its return value. Consider this **Main** function written in **CSharp**:

```
public static int Main(string[] args)
{
  Console.WriteLine(«Amount of args passed: « + args.Length);
  return args.Length;
}
```

or in **Visual Basic**:

```
Public Shared Function Main(ByVal args As String()) As Integer
  Console.WriteLine(«Amount of args passed: « + args.Length)
  Return args.Length
End Function
```

The method accepts a string array and returns the integer as a result. These obligations can be treated as contracts:

- you have to pass a single array of strings as an argument to this method: you can’t pass additional arguments, and you can’t pass an array of different type, e.g. an array of integer type.

- you are not able to treat return result as a different type other than integer, for example, as a string type.

Now, additional questions can arise to this method that may represent other contracts:

- Can the method take ‘null’ (*CSharp*) or ‘Nothing’ (*Visual Basic*) as an argument?
- Can the method take an empty array that don’t contain elements at all?
- How many items in the array the method expects?
- Can item of an array argument be “nullness”?
- Can item of an array argument be an empty string?
- Can the return result of the method be ignored?

The **Add Contract** code provider is intended to help developers to add these additional contracts. Here’s the list of available contracts:

Contract name	Description
Assert Contract	Checks for a condition; if the condition is false, follows the escalation policy set by the analyzer and displays the specified message.
Assume Contract	Instructs code analysis tools to assume that a condition is true, even if it cannot be statically proven to always be true, and displays a message if the assumption fails.
Ensures Contract	Specifies a post condition contract for a provided exit condition and a message to display if the condition is false.
Invariant Contract	Specifies an invariant contract for the enclosing method or property, and displays a message if the condition for the contract fails.
Requires Contract	Specifies a precondition contract for the enclosing method or property, and displays a message if the condition for the contract fails.
Exit Method	Generates return statements to validate active method parameters in runtime.
Throw Exceptions	Generates throw exception statements to validate active method parameters in runtime.
Use Assertion	Generates debug assertions to validate active method parameters in runtime.

The first five contracts (*Assert*, *Assume*, *Ensures*, *Invariant*, *Requires*) are available in Visual Studio 2010 only (the project must target the .NET 4.0 (and higher) version as well). These contracts provide a language-agnostic way to express coding assumptions in programs. They take the form of pre-conditions, post-conditions, and object invariants, and are used to improve testing via runtime checking.

These contracts generate the following code for the array parameter correspondingly:

CSharp:

```
Contract.Assert(args != null && args.Length != 0, “args is null or empty.”);
Contract.Assume(args != null && args.Length != 0, “args is null or empty.”);
Contract.Ensures(args != null && args.Length != 0, “args is null or empty.”);
Contract.Invariant(args != null && args.Length != 0, “args is null or empty.”);
Contract.Requires(args != null && args.Length != 0, «args is null or empty.»);
```

Visual Basic:

```
Contract.Assert(args IsNot Nothing AndAlso args.Length <> 0, “args is nothing or empty.”)

Contract.Assume(args IsNot Nothing AndAlso args.Length <> 0, “args is nothing or empty.”)

Contract.Ensures(args IsNot Nothing AndAlso args.Length <> 0, “args is nothing or empty.”)

Contract.Invariant(args IsNot Nothing AndAlso args.Length <> 0, “args is nothing or empty.”)

Contract.Requires(args IsNot Nothing AndAlso args.Length <> 0, «args is nothing or empty.»)
```

The three other contracts (*Exit Method*, *Throw Exceptions*, *Use Assertion*) represent defensive statements called “guard code”. They act like a shield, keeping us from making obvious mistakes later in the method. For example, by checking that the array isn’t null and that it’s at least one element in length, we believe that the code in the remainder of the method will run smoothly or at least without exceptions. These tests will allow us to check that the array passed to the method is structured and populated just as we require.

These contracts generate the following code for the array parameter:

Exit Method:

CSharp:

```
int result = 0;

if (args == null || args.Length == 0)

return result;
```

Visual Basic:

```
Dim result As Integer = 0

If args Is Nothing OrElse args.Length = 0 Then

  Return result

End If
```

Throw Exceptions:

CSharp:

```
if (args == null || args.Length == 0)

throw new ArgumentException(«args is null or empty.», «args»);
```

Visual Basic:

```
If args Is Nothing OrElse args.Length = 0 Then
  Throw New ArgumentException(“args is nothing or empty.”, “args”)
End If
```

Use Assertion:

CSharp:

```
Debug.Assert(args != null && args.Length != 0, «args is null or empty.»);
```

Visual Basic:

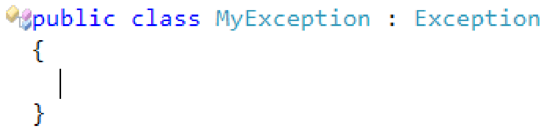
```
Debug.Assert(args IsNot Nothing AndAlso args.Length <> 0, «args is nothing
or empty.»)
```

You are able to extend the list of available contracts by adding your own using the ContractProvider **DX-Core** component via creating a new IDE Tools plug-in.

Add Missing Constructors code provider

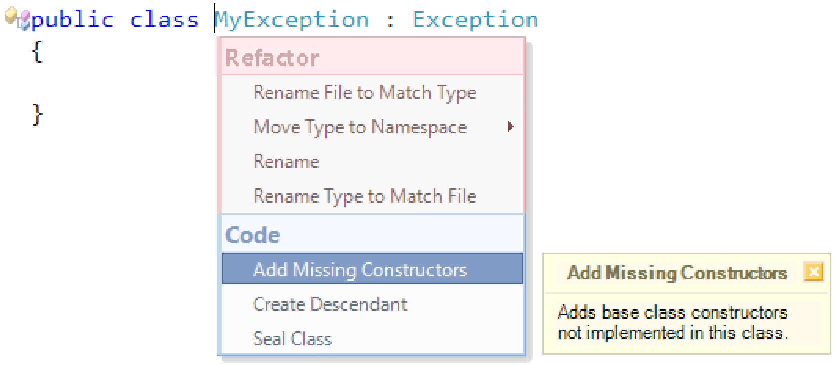
The **Add Missing Constructors** code provider from CodeRush allows you to add constructors from the ancestor class to the current class or structure, which are not implemented.

Imagine you are creating a descendant of the *System.Exception* class:



```
public class MyException : Exception
{
  |
}
```

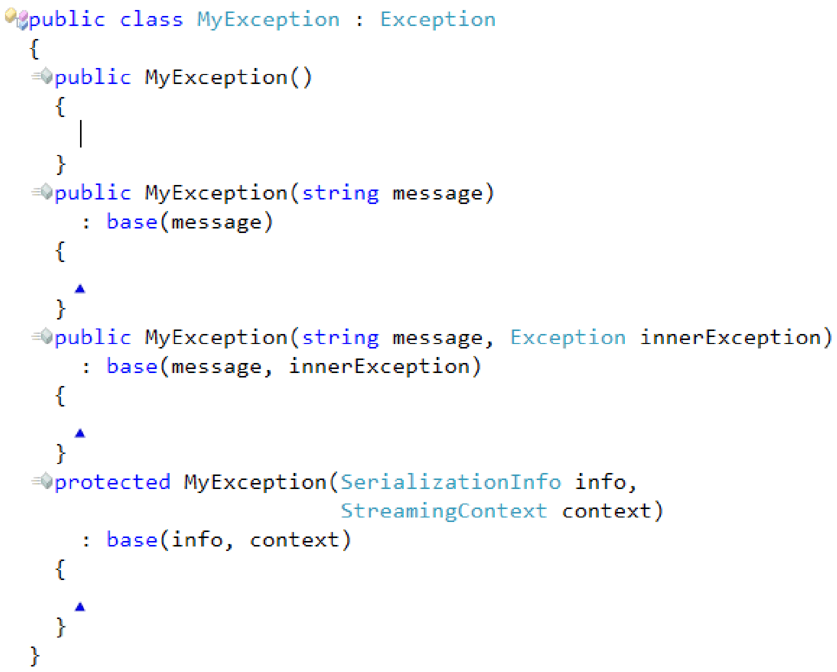
The **Add Missing Constructors** is available when the editor caret is located on the name of the class or its ancestor:



```
public class MyException : Exception
{

}
```

Adds base class constructors not implemented in this class.

After the code provider is applied, all constructors from the base class are added to your implementation:



```
public class MyException : Exception
{
  public MyException()
  {
    
  }
  public MyException(string message)
    : base(message)
  {

  }
  public MyException(string message, Exception innerException)
    : base(message, innerException)
  {

  }
  protected MyException(SerializationInfo info,
                        StreamingContext context)
    : base(info, context)
  {

  }
}
```

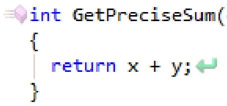
Constructors will call the appropriate base type constructors and have a marker inside of their bodies to navigate between them easily.

This feature is a code fix for the corresponding CodeRush code issues:

- Can implement base type constructors
- Base type constructors are not implemented

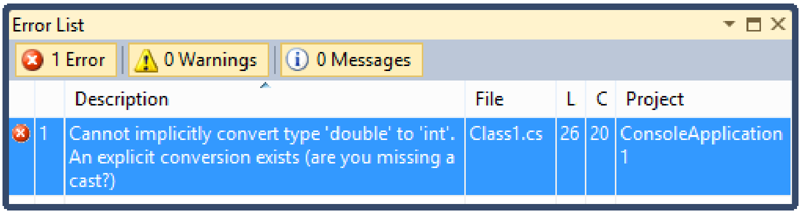
Convert to Integer

The **Convert to Integer** code provider allows you to wrap an expression returning a non-integer value to Math.Ceiling, Math.Floor, or Math.Round. Consider the following code sample:



```
int GetPreciseSum(double x, double y)
{
  return x + y;
}
```

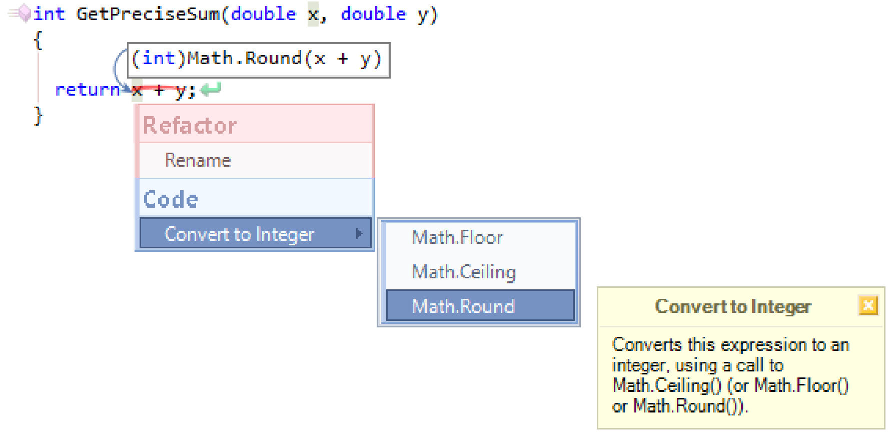
The method returns the sum of two input parameters as an integer. However, both input parameters are of the ‘double’ type. In this case, Visual Studio shows an error:



Error List

		Description	File	L	C	Project
⊗	1	Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)	Class1.cs	26	20	ConsoleApplication 1

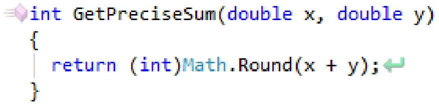
To fix this error, we can apply the **Convert to Integer** code provider and choose the rounding method:



```
int GetPreciseSum(double x, double y)
{
          (int)Math.Round(x + y)
  return ~~x + y~~;
}
```

Converts this expression to an integer, using a call to Math.Ceiling() (or Math.Floor() or Math.Round()).

Once the code provider is applied, the code is modified according to the preview hint:

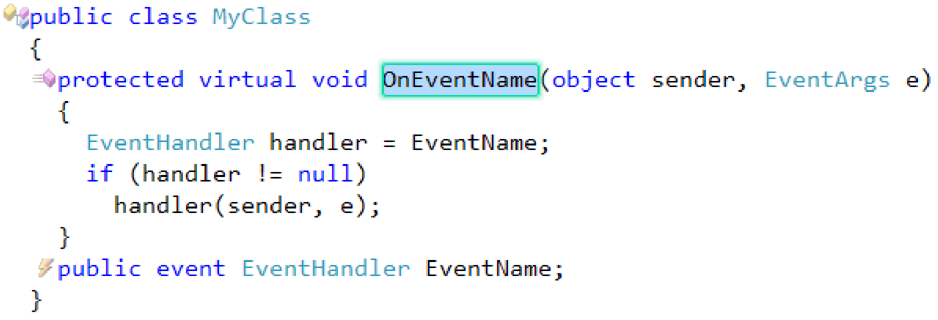


```
int GetPreciseSum(double x, double y)
{
  return (int)Math.Round(x + y);
}
```

Create Event Trigger

The **Create Event Trigger** CodeRush code provider generates an event trigger for the specified event. An event trigger is a single method that is used to raise an event. Although it is not strictly necessary to create such a trigger, it is useful, as it makes maintenance of the code simpler.

Once the code provider is applied, you can select the target position where the event trigger should be declared. The event trigger will be named *OnEventName* (where EventName is the name of the original event) and will be 'virtual', so derived classes can override the functionality and alter the manner in which events are fired:

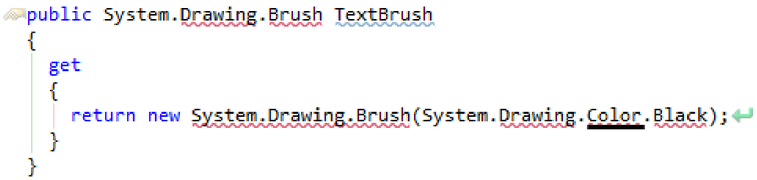


```
public class MyClass
{
  protected virtual void OnEventName(object sender, EventArgs e)
  {
    EventHandler handler = EventName;
    if (handler != null)
      handler(sender, e);
  }
  public event EventHandler EventName;
}
```

Note that the protected OnEventName method allows derived classes to override the event without attaching a delegate to it. A derived class must always call the OnEventName method of the base class to ensure that registered delegates receive the event.

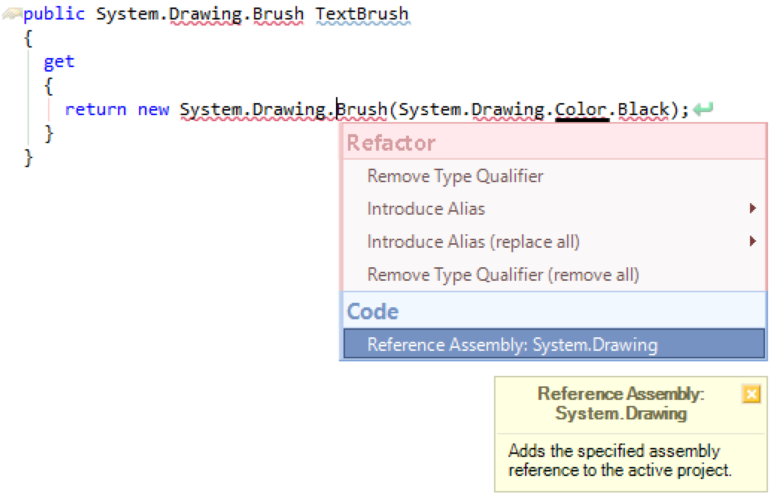
Reference Assembly

The **Reference Assembly** code provider is a simple coding helper which allows you to add an assembly reference if one does not exist for the type reference under the editor caret. The code provider is connected to the CodeRush Code Issues feature, so the editor caret should be located in the 'Undeclared element' code issue, for instance:



```
public System.Drawing.Brush TextBrush
{
  get
  {
    return new System.Drawing.Brush(System.Drawing.Color.Black);
  }
}
```

The code provider searches for the full type name under the editor caret and adds the required assembly reference once applied:



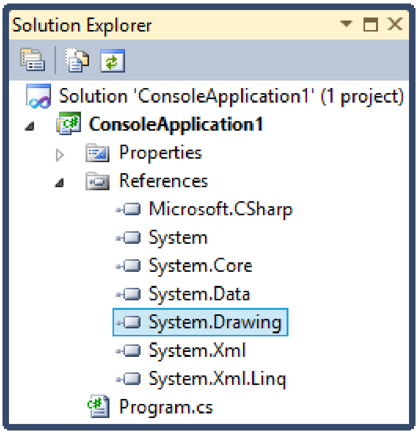
```
public System.Drawing.Brush TextBrush
{
  get
  {
    return new System.Drawing.Brush(System.Drawing.Color.Black);
  }
}
```

Refactor
Remove Type Qualifier
Introduce Alias
Introduce Alias (replace all)
Remove Type Qualifier (remove all)
Code
Reference Assembly: System.Drawing

Reference Assembly: System.Drawing

Adds the specified assembly reference to the active project.

so, the project no longer misses the assembly:



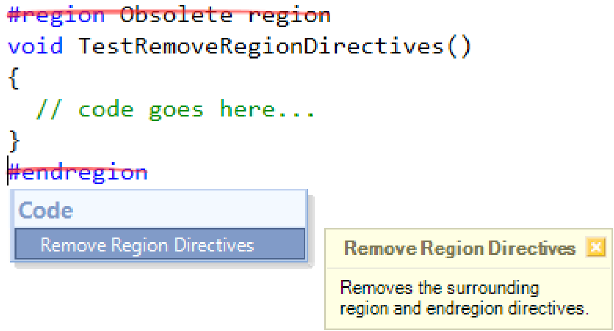
The code provider only searches in the several standard .NET assemblies.

Remove Region Directives

This simple code provider, as the name says, removes the two lines of region directives:

- #region and #endregion lines in **CSharp**
- #Region and #End Region lines in **Visual Basic**
- #pragma region and #pragma endregion lines in **C++**

It is available on any of the two directives – either starting or ending:



```
~~#region Obsolete region~~
void TestRemoveRegionDirectives()
{
  // code goes here...
}
~~#endregion~~
```

Removes the surrounding region and endregion directives.

The opposite of this code provider is the Auto Create Region feature.

Code provider to format the String.Format call output

The **String.Format** call is the best approach for outputting information for the user to read. It replaces the format items in the specified template string with the string representation of the corresponding objects. The template string contains text that is desired in the final string and one or more format item placeholders, which will be replaced with other values passed as arguments to the String.Format call as either variable references or other literals.

When inserting values into format item placeholders, the values can be formatted by adding numerous format specifiers within the placeholders. A format specifier is an optional string of formatting codes. There are standard and custom format specifiers for formatting numbers, dates and times, and enumerations. Here are some of the available format specifiers:

Numeric Format Specifiers

Character	Description	Usage	Example Output
c	Currency	{0:c}	$12,345
d	Decimal (whole number)	{0:d}	12345
e	Scientific	{0:e}	1.234500e+004
f	Fixed Point	{0:f}	12345
g	General	{0:g}	12345
n	Thousand Separator	{0:n}	12,345
r	Round Triple (decimal only)	{0:r}	System.FormatException
x	Hexadecimal (int only)	{0:x4}	3039

Date/Time Format Specifiers

Date and Time format specifiers are dependant on the user's locale, so the output may be different.

Character	Description	Usage	Example Output
d	Short date	{0:d}	04/09/2012
D	Long Date	{0:D}	04 September 2012
t	Short time	{0:t}	15:43
T	Long time	{0:T}	15:43:55
f	Full Date/Time	{0:f}	04 September 2012 15:43
F	Full Date/Time (long time)	{0:F}	04 September 2012 15:43:55
g	General Date/Time	{0:g}	04/09/2012 15:43
G	General Date/Time (long time)	{0:G}	04/09/2012 15:43:55
M or m	Month Day	{0:M}	04 September
R or r	RFC1123	{0:r}	Tue, 04 Sep 2012 15:43:55 GMT
s	SortableDate/Time	{0:s}	2012-09-04T15:43:55
u	Universal Full Date/Time	{0:u}	2012-09-04 15:43:55
U	Universal Full Date/Time (long time)	{0:U}	04 September 2012 15:43:55
Y or y	Year Month	{0:Y}	September 2012
O or o	Round-Trip Date/Time	{0:O}	2012-09-04T15:43:55.7770000

Custom Date/Time Format Specifiers

Format	Description	Usage	Example Output
dd	Day	{0:dd}	04
ddd	Day Name	{0:ddd}	Tue
dddd	Full Day Name	{0:dddd}	Tuesday
f, ff, …	Second Fractions	{0:fff}	777
gg, …	Era	{0:gg}	A.D.
hh	2 Digit Hour	{0:hh}	03
HH	2 Digit Hour, 24hr Format	{0:HH}	15
mm	Minute 00-59	{0:mm}	43
MM	Month 01-12	{0:MM}	09
MMM	Month Abbreviation	{0:MMM}	Sep
MMMM	Full Month Name	{0:MMMM}	September
ss	Seconds 00-59	{0:ss}	46
tt	AM or PM	{0:tt}	PM
yy	Year, 2 digits	{0:yy}	12
yyyy	Year	{0:yyyy}	2012
zz	Timezone offset, 2 digits	{0:zz}	-05
zzz	Full timezone offset	{0:zzz}	-05:00
:	Separator	{0:hh:mm:ss}	15:43:55

/	Separator	{0:dd/MM/yyyy}	04/09/2012

General Format Specifiers

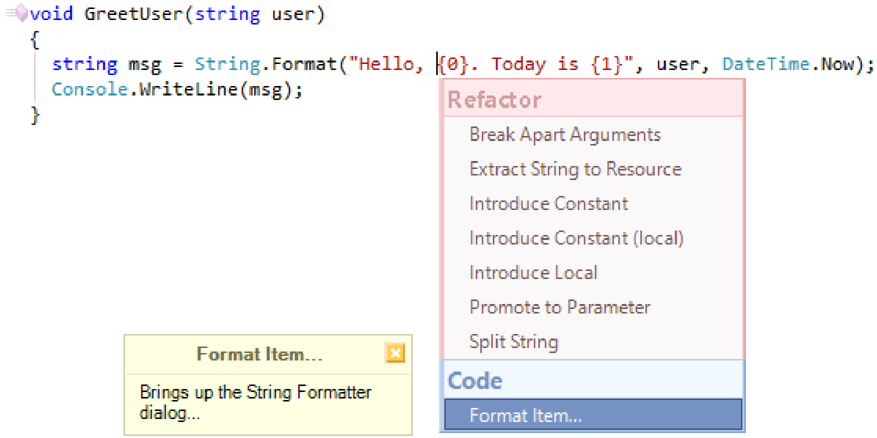
You can also use custom format strings, such as decimal placeholders, and leading and trailing characters. In these examples, we pass in 12345.12.

Character	Description	Usage	Example Output
0	Zero Placeholder	{0:00.0000}	12345.1200
#	Digit Placeholder	{0:(#).##}	(12345).12
.	Decimal Point	{0:0.0}	12345.12
,	Thousand Separator	{0:0,0}	12,345
%	Percent	{0:0%}	1234512%
e	Exponent Placeholder	{0:00e+0}	12e+3

Some useful examples

Format	Usage	Example Output
0:000-00-0000	SSN	012-34-5678
0:(###)###-####	US Phone Number	(555)555-5555
0:#-(###)###-####	US Phone Number	1-(555)555-5555
0:1-(###)###-####	US Phone Number	1-(555)555-5555

As you can see there are too many of format specifiers to remember. But CodeRush may help you to format the template string without necessity to remember any of the format specifiers. The **Format Item…** code provider is available on the format item placeholder:

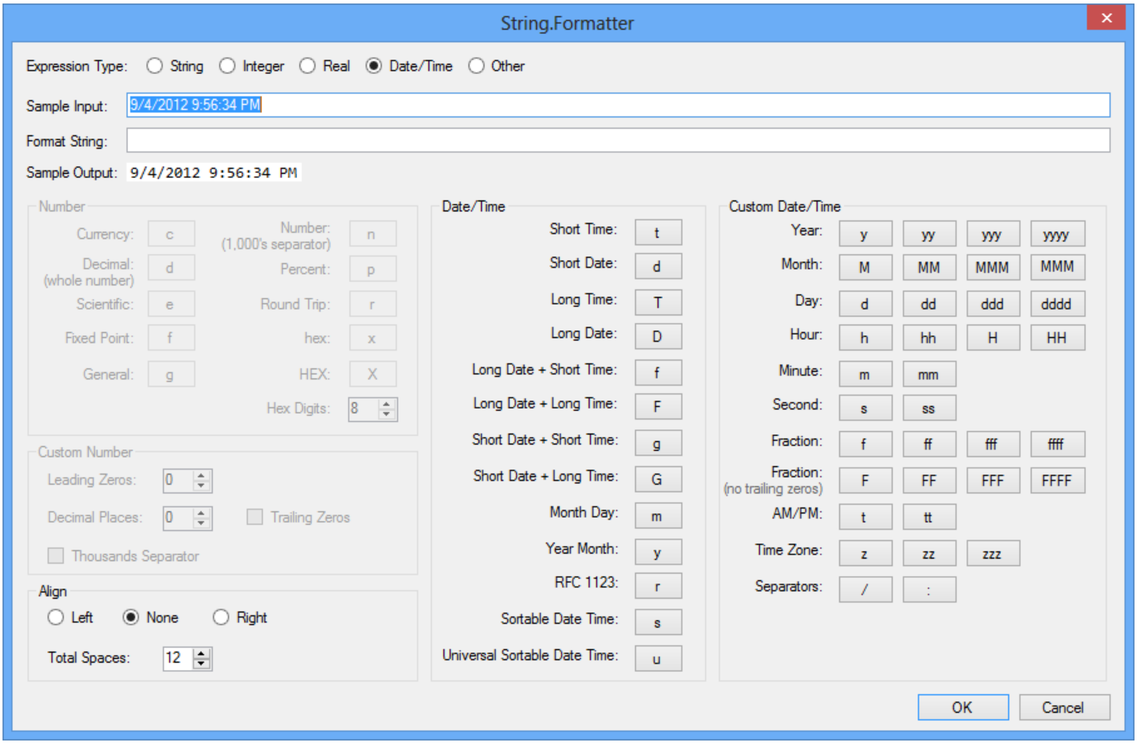


```
void GreetUser(string user)
{
  string msg = String.Format("Hello, {0}. Today is {1}", user, DateTime.Now);
  Console.WriteLine(msg);
}
```

Format Item...

Brings up the String Formatter dialog...

Once executed, it shows you the **String.Formatter** dialog where you can choose the format specifier you prefer and see the sample output before it is applied:



String.Formatter

Expression Type: ○ String ○ Integer ○ Real ● Date/Time ○ Other

Sample Input: 9/4/2012 9:56:34 PM

Format String:

Sample Output: 9/4/2012 9:56:34 PM

Leading Zeros: 0

Decimal Places: 0

Hex Digits: 8

Align: ○ Left ● None ○ Right

Total Spaces: 12

The dialog allows you to easily apply a format specifier to the template string. The corresponding format specifiers are enabled automatically based on the value type passed as argument into the String.Format call. The feature might be very helpful if you forgot any of the format specifiers and want to apply a custom string output.

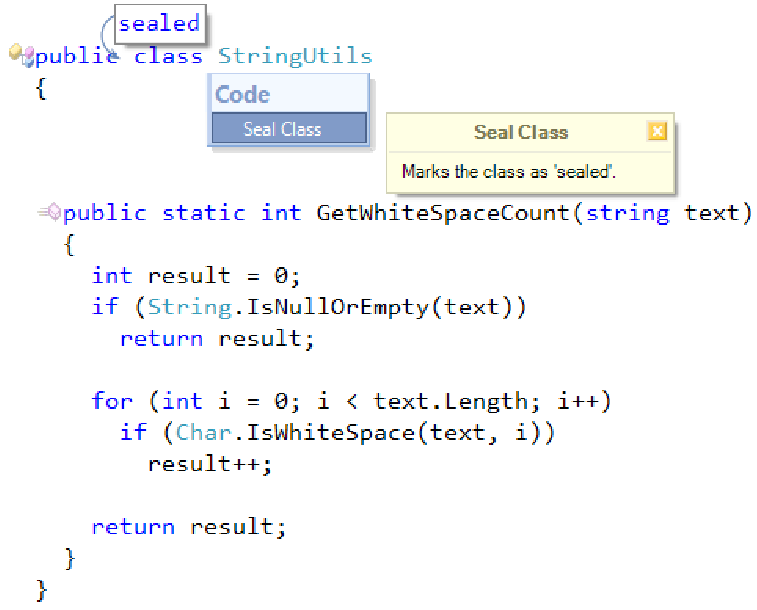
The String.Formatter has been built quickly in front of the users in the CodeRush Feature Workshop Webinar series. You might want to review them to learn more on how to build great features quickly. There are two parts:

- CodeRush Feature Workshop: Improving String.Format (Part 1)
- CodeRush Feature Workshop: Improving String.Format (Part 2)

Seal Class

This is one of the simplest code provider operations which simply marks the active class as '*sealed*' (CSharp) or '*NotInheritable*' (Visual Basic). The sealed modifier is used to prevent inheriting from a class. For example, this can be useful when creating 'helper' classes containing utility methods that should never be extended by overriding the existing functionality. Another benefit of this code provider is that it enables a hypothetical run-time optimization, because having a sealed class known to never have any derivations; it is possible for the compiler to transform virtual function member invocations into non-virtual which take less time to perform.

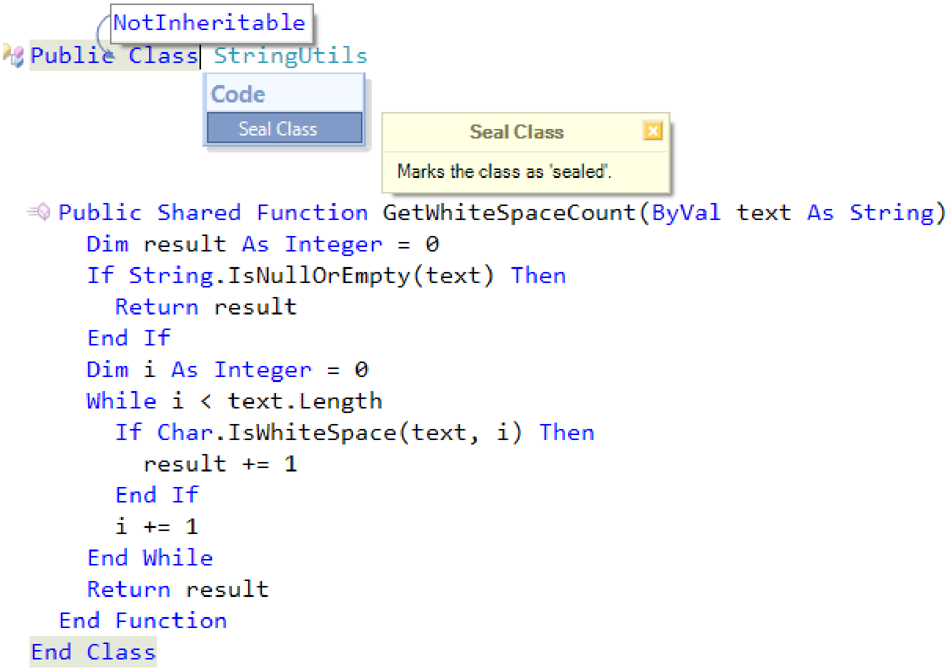
CSharp preview:



Marks the class as 'sealed'.

```
public class StringUtils
{
  public static int GetWhiteSpaceCount(string text)
  {
    int result = 0;
    if (String.IsNullOrEmpty(text))
      return result;

    for (int i = 0; i < text.Length; i++)
      if (Char.IsWhiteSpace(text, i))
        result++;

    return result;
  }
}
```

Visual Basic preview:



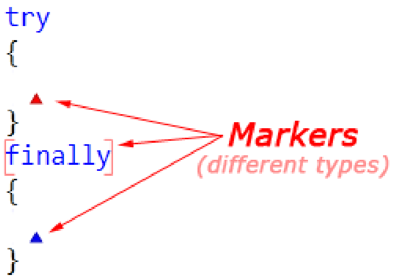
Marks the class as 'sealed'.

```
Public Class StringUtils
  Public Shared Function GetWhiteSpaceCount(ByVal text As String)
    Dim result As Integer = 0
    If String.IsNullOrEmpty(text) Then
      Return result
    End If
    Dim i As Integer = 0
    While i < text.Length
      If Char.IsWhiteSpace(text, i) Then
        result += 1
      End If
      i += 1
    End While
    Return result
  End Function
End Class
```

Chapter 8. Code navigation utilities

Markers

Markers are navigation placeholders that remember important locations inside your source code you'll need to move to in the future. In most cases, they look like little triangular glyphs in the IDE's code editor:



```
try
{

}
finally
{

}
```

Markers are stack-based. When you collect a marker, it pops off the stack. You can jump back at any time to the top marker on the stack by pressing **Esc** key (or **Alt**+**End**).

Markers can be of the following types:

- **Hard Marker** (red triangle) – is dropped manually onto the current caret position using the default **Alt**+**Home** key. They will remain until collected, regardless of any activity on the line they sit on. Currently, manual dropping and manipulating of markers is only available in CodeRush Pro. But there's awork-around for other products if you wish to apply this feature.

- **Soft Marker** (blue triangle) – is dropped automatically by **CodeRush** and/or Refactor! (e.g. through a template expansion, selection embedding, or whenever you apply a refactoring or code provider that takes you away from the original location, etc). This kind of marker will dissolve and vanish from the stack if the line on which they are resident is altered in any way.

- **Selection Marker** (color brackets) – is dropped manually onto the current selection. It completely restores the selection you had before, once you collect it.

Invisible Marker – is dropped onto the current position when you use navigation features such as Tab to Next Reference or **Quick Navigation**. This marker is not displayed in the code editor, and is used to restore your original location before you started using the navigation features.

Markers provide the following features:

Feature	Default shortcut	Description
Collect Marker	**Esc** (**Alt**+**End**)	Moves the ***caret**** to the topmost marker within the stack, and removes this marker.
Drop Marker	**Alt**+**Home**	Drops a marker at the current caret position.
Swap Marker	**Shift**+**Alt**+**Home**	Moves the topmost marker within the stack to the current cursor position, and moves the caret to the previous position of the marker.
Collect Marker and Paste	**Shift**+**ESC**	Moves the caret to the topmost marker within the stack, removes this marker and pastes the contents of the clipboard to the new caret position.
Select Marker	**Shift**+**Alt**+**Page Up**	Selects the code section from the current caret position to the topmost marker within the stack.

Collect Marker

When marker is collected, a circle (like the image above) appears, and starts narrowing until it finally hides. The animation is intended to catch your attention, so you see what happened, instead of looking around the screen where your caret ended up.

Collect Marker and Paste

If the clipboard contains code that you want to paste at a recently-dropped marker, you can jump back to the marker and paste in a single action by pressing **Shift**+**Esc**.

Swapping markers

You can even use markers to work in two places at once (alternating between two important locations with a single keystroke), and you can also use them as visual reminders of unfinished code. Often when creating new code, you want to reference another section of code while you work. Usually when this happens, the location where you want to build the new code is far from the source. **Markers** can solve this problem elegantly. Here’s how:

1. Press **Alt**+**Home** to drop a marker at the first location where you want to be. Copy anything to the clipboard from this location, if needed.

2. Navigate to the secondary location.

3. Do what you need at the secondary location until you need to return to the original location.

4. Press **Shift**+**Alt**+**Home** to execute the **Swap Marker** feature. This will exchange the caret with the topmost marker on the stack, replacing the topmost marker with a new one (positioned at the caret location where you just pressed **Shift**+**Alt**+**Home**).

5. Now, do what you need at the original location (e.g. copy something to the clipboard).

6. When you’re ready to return to secondary location, press **Shift**+**Alt**+**Home** again.

7. Repeat steps 3-6 as often as you like, to get the work done.

This is an area where the dynamic nature of markers really excels over static bookmarks. All you need to do is drop a marker at the primary location and press **Shift**+**Alt**+**Home** from the secondary location. As you work down through each respective section of code, the markers move with you, always dropping at the point where you stopped working.

Markers – Comparison with Visual Studio Bookmarks

If you are already proficient with Visual Studio bookmarks, you might be wondering if **CodeRush** markers offer any compelling benefits over bookmarks. Indeed, markers offer a number of improvements over the basic bookmarks included with Visual Studio. Here’s a quick overview of the differences between markers and Visual Studio bookmarks.

Feature	**Markers**	Visual Studio Bookmarks
Marks a line for later visitation	✓	✓(VS bookmarks are drawn in the widget column, and are prone to being obscured by breakpoints and other widgets)
Remembers column position	✓	✗ (the caret is always positioned at first column)
Remembers view position	✓	✗ (bookmark is always centered)

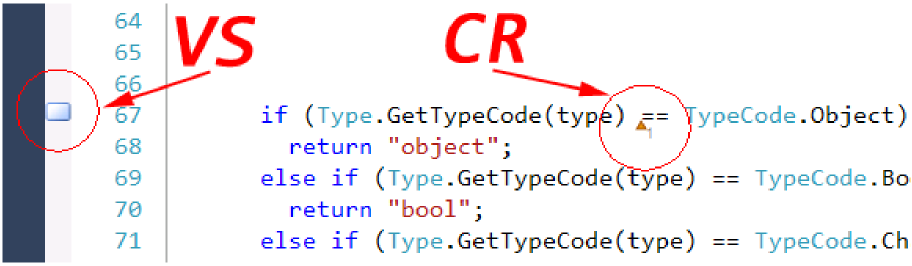
Stack-based, allowing tracing of steps	✓	✗ (bookmarks are navigated in line number order)
Move quickly between two locations	✓	✓ (only if the two locations are both in the active file (and there are only two bookmarks in that file)

Bookmarks

When you work on large projects you may need to revisit several areas of your code on a regular basis. CodeRush allows you to store such important locations in the code and move back to them in the future by using bookmarks. Visual Studio provides a similar bookmarks feature, which allows you to mark places in your code that you would want to come back to. Let's see what differences between the CodeRush and Visual Studio bookmarks are.

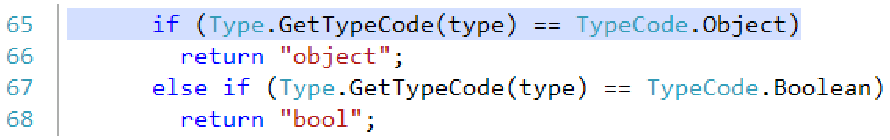
Appearance

This is what Visual Studio (VS) and CodeRush (CR) bookmarks look like:



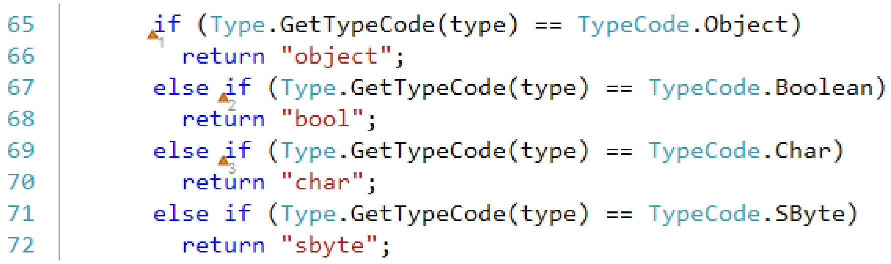
```
        if (Type.GetTypeCode(type) == TypeCode.Object)
          return "object";
        else if (Type.GetTypeCode(type) == TypeCode.Bo
          return "bool";
        else if (Type.GetTypeCode(type) == TypeCode.Ch
```

A Visual Studio bookmark appears as a square-shaped glyph in the margin at the left of the code. There is also an option (uncheck the Options | Text Editor | Indicator margin option, then go to Fonts and Colors and change the background color of the Bookmark display item) for VS bookmarks to highlight the entire marked line, so it might look like this:



```
        if (Type.GetTypeCode(type) == TypeCode.Object)
          return "object";
        else if (Type.GetTypeCode(type) == TypeCode.Boolean)
          return "bool";
```

CodeRush bookmarks have an index indicator starting from 1 to infinity. It allows you to visually remember several bookmarks and their associated position:



```
        if (Type.GetTypeCode(type) == TypeCode.Object)
          return "object";
        else if (Type.GetTypeCode(type) == TypeCode.Boolean)
          return "bool";
        else if (Type.GetTypeCode(type) == TypeCode.Char)
          return "char";
        else if (Type.GetTypeCode(type) == TypeCode.SByte)
          return "sbyte";
```

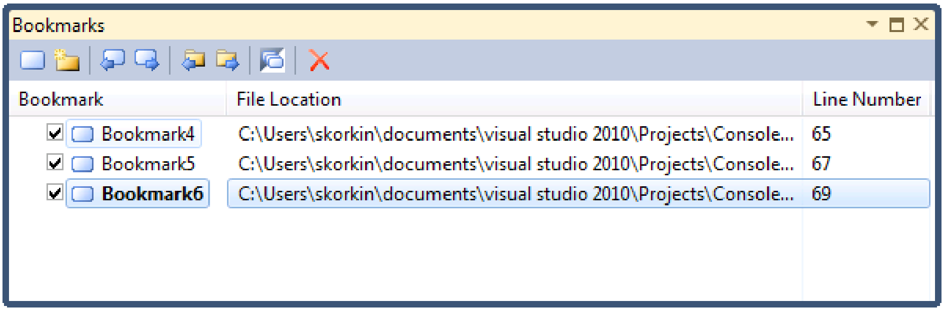
VS bookmarks allow you to bookmark a line in your code, where as CR bookmarks may store an exact position of the editor caret, so you don't have to move an editor caret later when you get back to a bookmark.

Toggle/Navigate

You can add and/or remove CR and VS bookmarks via a keyboard shortcut. Once a CR or VS bookmark is added, it is persisted between Visual Studio sessions.

CodeRush allows you to easily switch between the first ten (0..9) bookmarks at any time by hitting the specific shortcut. You can setup additional shortcuts to navigate to other particular bookmarks (10…~) on the Shortcuts options page in the Options Dialog.

VS bookmarks, on the other hand, do not have an index and have no associated shortcuts to navigate to the specific bookmark, but, instead, provide a tool window (View | Other Windows | Bookmark Window) that enumerates all VS bookmarks:

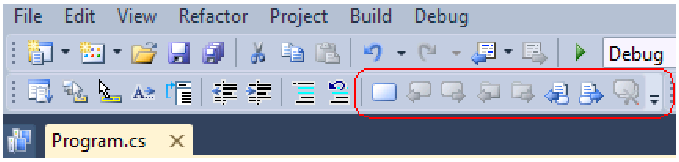


Bookmarks

Bookmark	File Location	Line Number
☑ Bookmark4	C:\Users\skorkin\documents\visual studio 2010\Projects\Console...	65
☑ Bookmark5	C:\Users\skorkin\documents\visual studio 2010\Projects\Console...	67
☑ Bookmark6	C:\Users\skorkin\documents\visual studio 2010\Projects\Console...	69

Hitting a particular shortcut might be faster in navigating to the specific location. VS bookmark feature only allows you to navigate between next and previous bookmarks using shortcuts.

VS bookmarks actions are also available on the Text Editor toolbar:



and the Edit | Bookmarks menu item.

Shortcuts

The following shortcuts are used for bookmarks:

Shortcut	CR	VS
Toggle bookmark	Ctrl + Plus	`Ctrl+K, Ctrl+K`
Toggle bookmark with a specific index (number)	Ctrl + Alt + Number	–
Remove last bookmark	Ctrl + Minus	–
Move to next bookmark	Ctrl + Alt + Right	`Ctrl+K, Ctrl+N`
Move to previous bookmark	Ctrl + Alt + Left	`Ctrl+K, Ctrl+P`
Move to specific bookmark	Alt+Number	-
Clear all bookmarks	-	`Ctrl+K, Ctrl+L`

Comparison

Here is the comparison table for mentioned bookmark capabilities:

Feature	CR	VS
Toggle bookmark via shortcut	✓	✓
Toggle bookmark via mouse click	✗	✓
Navigation to particular column	✓	✗
Navigation to particular bookmarks	✓	✗
Visual index indicator	✓	✗
Tool Window	✗	✓
Toolbar	✗	✓
Persistence between sessions	✓	✓

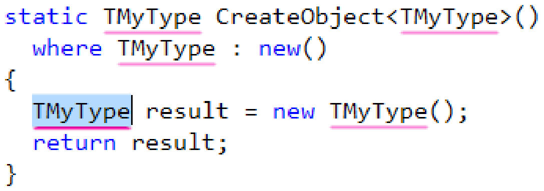
Conclusion

CodeRush bookmarks may be faster to use because they have shortcuts to navigate to the specific bookmarks and have more precise navigation. If you use bookmarks extensively, Visual Studio bookmarks can be better organized with an extra level of organization using folders inside the Bookmarks tool window.

Tab to Next Reference

The **Tab to Next Reference** feature is the simplest and one of the most powerful navigation features of CodeRush. It allows you immediately see all references and navigate among them with ease. As the name says, the single Tab shortcut is used to navigate to the next reference of an identifier or a type reference under the editor caret. Hitting the Tab key again will move you to the next reference cyclically, in other words, you can tab to the next reference over and over again, even if you come back to the first reference where you started.

The underline visual indicator is used to highlight navigation references inside the code editor. By default, it has a pink color (which is customizable):



```
static TMyType CreateObject<TMyType>()
  where TMyType : new()
{
  TMyType result = new TMyType();
  return result;
}
```

You can also use the Shift+Tab shortcut to navigate backward to the previous reference, or the Escape key to move back to the location where you started. The feature works solution wide, opening a file with a reference if necessary. It also has cross-language support, e.g. if your solution contains projects written in different languages (e.g. *C#, C++, VB*) and you Tab over a *System.Object* type reference you might visit all of those projects.

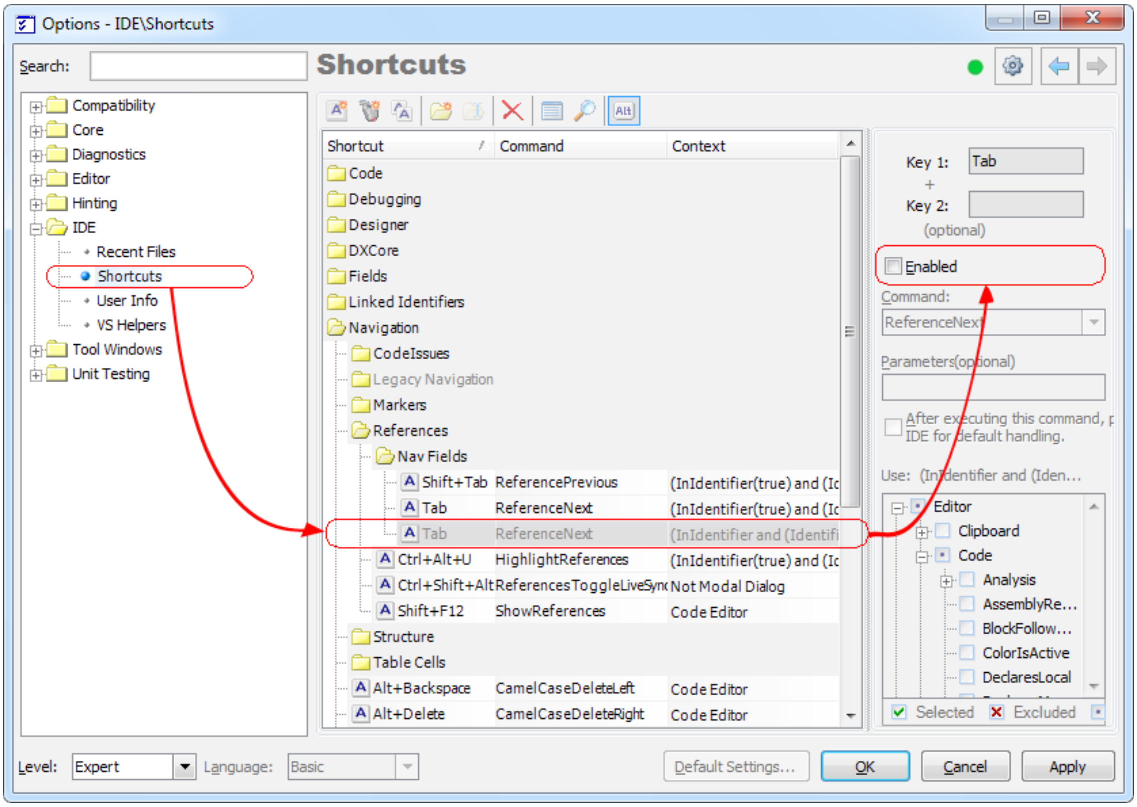
For those who prefer to align code using the white space, the **Tab to Next Reference** is not available at the very beginning of an identifier or a type references by default, e.g.:

```
void GetCount(ref int result)
{
  [caret]int count = 100;
  [caret]result += count;
}
```

“[caret]” is indicating the editor caret position.

In both places the feature will insert the regular tab whitespace character, because this character is often used for source code aligning. If you would like to change this behavior, there is an alternative Tab shortcut binding (disabled by default) that allows the Tab key to take you to the next reference when the caret at the beginning of an identifier or a type reference.

To enable this shortcut, go to the IDE | Shortcuts options page in the CodeRush Options Dialog, find an alternative shortcut binding (Navigation | References | Nav Fields) and check the *Enabled* check box:



Shortcut	Command	Context
Code		
Debugging		
Designer		
DXCore		
Fields		
Linked Identifiers		
Navigation		
CodeIssues		
Legacy Navigation		
Markers		
References		
Nav Fields		
Shift+Tab	ReferencePrevious	(InIdentifier(true) and (Ic
Tab	ReferenceNext	(InIdentifier(true) and (Ic
Tab	ReferenceNext	(InIdentifier and (Identifi
Ctrl+Alt+U	HighlightReferences	(InIdentifier(true) and (Ic
Ctrl+Shift+Alt	ReferencesToggleLiveSync	Not Modal Dialog
Shift+F12	ShowReferences	Code Editor
Structure		
Table Cells		
Alt+Backspace	CamelCaseDeleteLeft	Code Editor
Alt+Delete	CamelCaseDeleteRight	Code Editor

Key 1: Tab
+
Key 2:
(optional)

□ Enabled

Command: ReferenceNext

Parameters(optional)

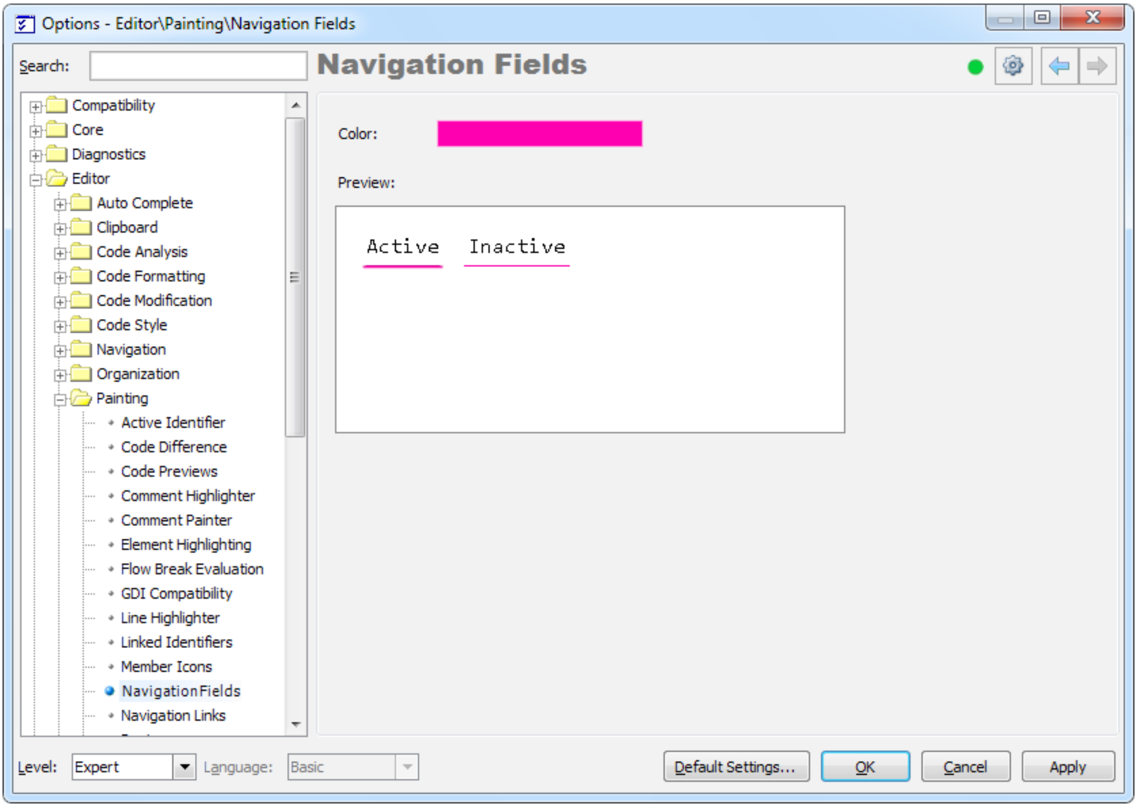
□ After executing this command, p IDE for default handling.

Use: (InIdentifier and (Iden...

Once it is enabled, to insert whitespace at the beginning of a line of code, you will need to use the Space bar instead of the Tab key, or move the editor caret to the left, so there’s at least one whitespace character to the right of the editor caret.

Color Options

To tweak the color of the underline highlighting, you can go to the *Editor* | *Painting* | *Navigation Fields* options page in the Options Dialog:



Options - Editor\Painting\Navigation Fields

Navigation Fields

Color: 🟥

Preview:

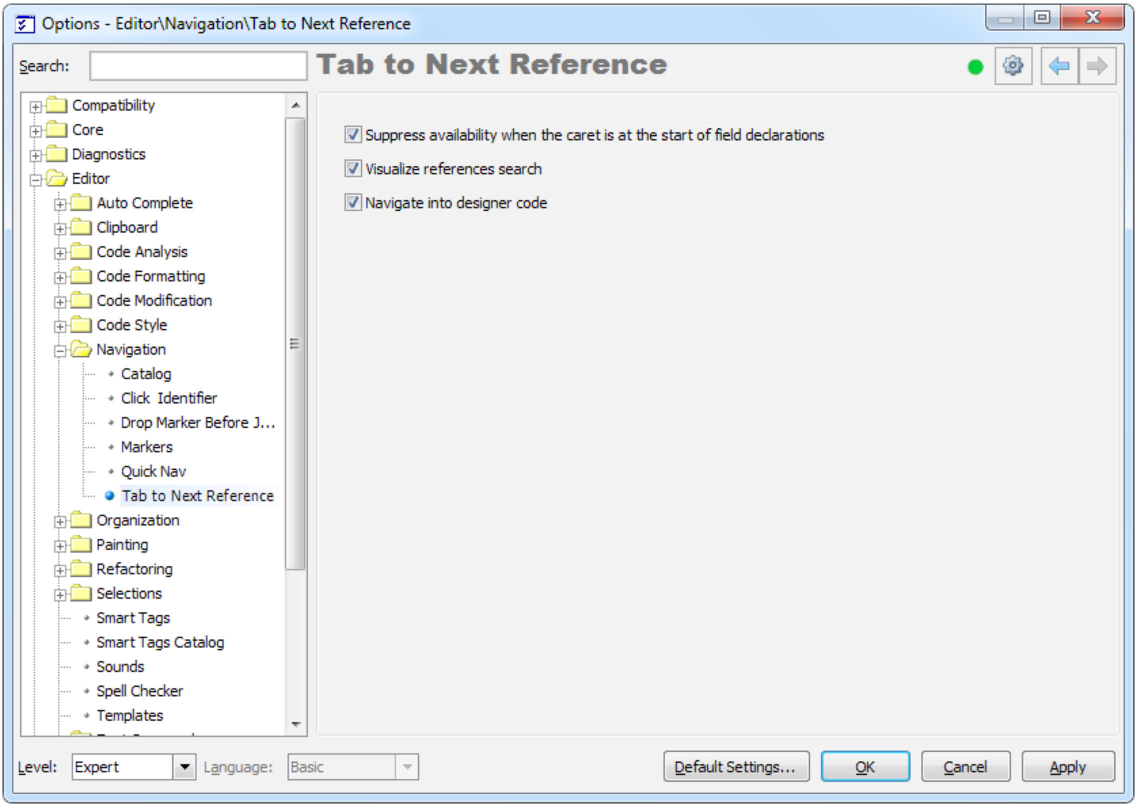
```
Active   Inactive
```

Level: Expert

Language: Basic

Behavior Options

The options specific to the **Tab to Next Reference** feature are located on the *Editor* | *Navigation* | *Tab to Next Reference* options page:



Options - Editor\Navigation\Tab to Next Reference

Tab to Next Reference

☑ Suppress availability when the caret is at the start of field declarations

☑ Visualize references search

☑ Navigate into designer code

Level: Expert

Language: Basic

Available options are:

- Suppress availability when the caret is at the start of field declarations

If checked, **Tab to Next Reference** won't be available at the start of a name of a field declaration, and hitting the Tab key will insert a regular whitespace character. Here's a sample code with a caret position which is implied in the option:

```
class Test
{
  int [caret]MyField;
}
```

This option is useful if you prefer to align your code in the following way:

```
class Test
{
  int    Height;
  int    Width;
  double  Weight;
  string   Name;
}
```

Visualize references search

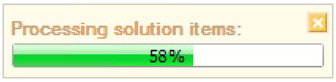
Shows a small hint with a progress bar while the references search is in progress. In most cases, you won't notice a hint, because references search is ultra-fast and takes only milliseconds.

- Navigate into designer code

If checked, **Tab to Next Reference** will skip references in designer-generated code inside "*.Designer.*" files.

Notes

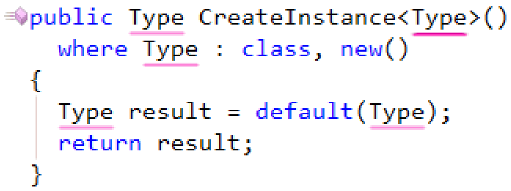
The **Tab to Next Reference** feature relies upon the DXCore's background parse, and may not be available for a few seconds after a solution is opened. In this case, if you press Tab on an identifier before the background parse is complete, you will get a regular tab whitespace character inserted at the editor caret position instead of the expected feature behavior. You can encounter such behavior only in rare cases for very large projects with thousands of files. When the background parsing is in progress, you can see the "Processing solution items…" hint:



If you don't see this hint, it is safe to use any **CodeRush** features as well as the **Tab to Next Reference**.

Highlighting an identifier and its references

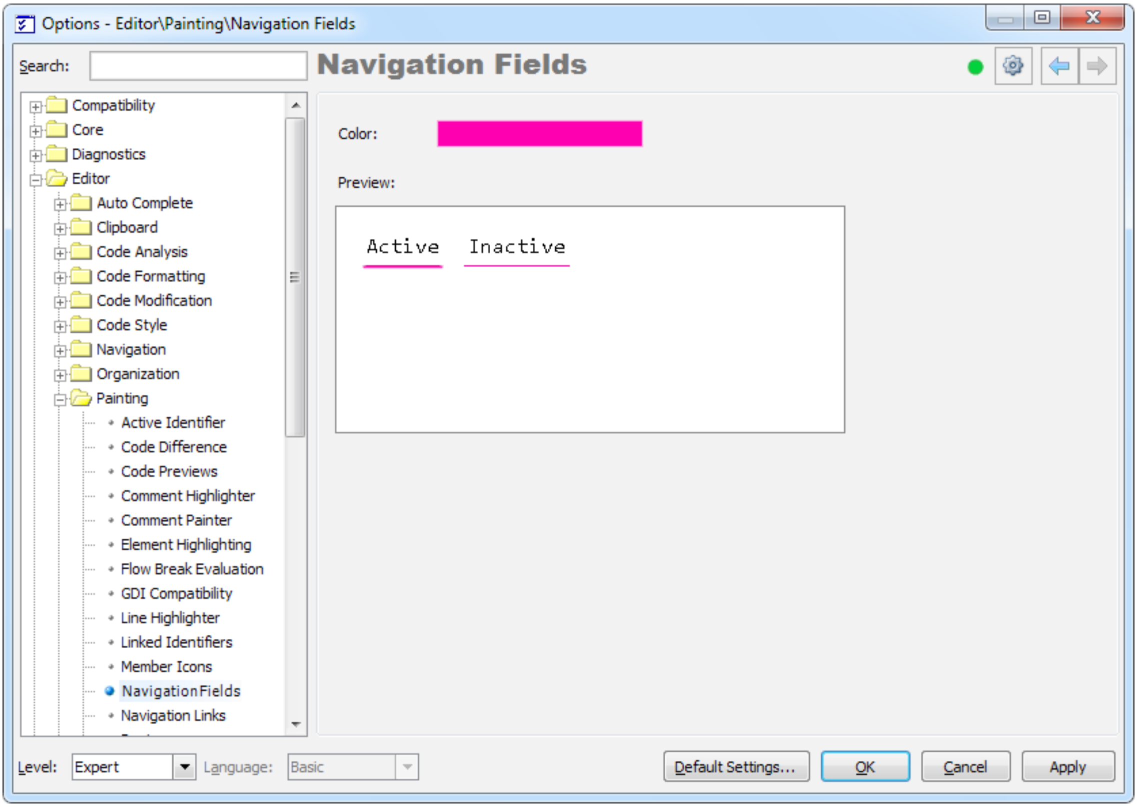
CodeRush has the **Highlighting All References** feature, based on the Tab to Next Reference feature, with the difference that you do not actually navigate between references. The feature simply highlights the current identifier and all its references. The default shortcut to apply the reference highlighting is Ctrl+Alt+U. Once performed on an identifier, you will see the identifier and its references highlighted in pink:



```
public Type CreateInstance<Type>()
  where Type : class, new()
{
  Type result = default(Type);
  return result;
}
```

If you move the caret outside of the highlight – it will automatically disappear. Or, you can press the Enter key to accept the link, so it will go away. Don't forget that you can perform the Tab to Next Reference feature if you would like to navigate between highlighted references.

Visual Studio 2010 has a similar highlighting of a word under the caret, however, a gray background is not as visually appealing as the highlight of this **CodeRush** feature. The color of the highlighting can be changed on the Editor | Painting | Navigation Fields options page in the Options Dialog:



Options - Editor\Painting\Navigation Fields

Search:

Navigation Fields

- Compatibility
- Core
- Diagnostics
- Editor
  - Auto Complete
  - Clipboard
  - Code Analysis
  - Code Formatting
  - Code Modification
  - Code Style
  - Navigation
  - Organization
  - Painting
    - Active Identifier
    - Code Difference
    - Code Previews
    - Comment Highlighter
    - Comment Painter
    - Element Highlighting
    - Flow Break Evaluation
    - GDI Compatibility
    - Line Highlighter
    - Linked Identifiers
    - Member Icons
    - NavigationFields
    - Navigation Links

Color: 🟥

Preview:

Active Inactive

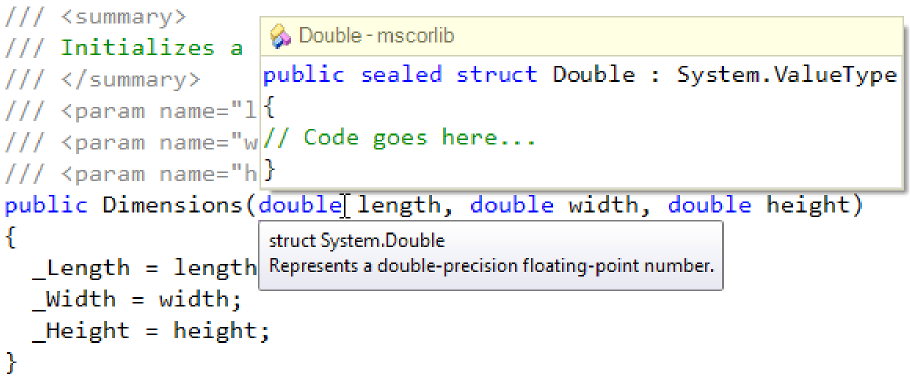
Level: Expert Language: Basic

Default Settings... OK Cancel Apply

As always, you can change or add a new shortcut to apply the feature on the Shortcuts options page, using the **HighlightReferences** action name.

Click Identifier

Click Identifier allows you to navigate to the declaration of the identifier under the mouse cursor by a single mouse left click when the **CTRL** key is held down. If the declaration is located inside your source code, the source file will be opened and the text caret will move to that declaration. Otherwise, the metadata for the target declaration will be shown (e.g. for "***System.Double***").



```
/// <summary>
/// Initializes a
/// </summary>
/// <param name="l
/// <param name="w
/// <param name="h
public Dimensions(double length, double width, double height)
{
   _Length = length
   _Width = width;
   _Height = height;
}
```

Double - mscorlib

```
public sealed struct Double : System.ValueType
{
// Code goes here...
}
```

struct System.Double
Represents a double-precision floating-point number.

Click Identifier feature can show a preview hint of the declaration when you hover over an identifier with the mouse pointer (while holding down the **CTRL** key) and drop a marker at the source location before navigating to it.

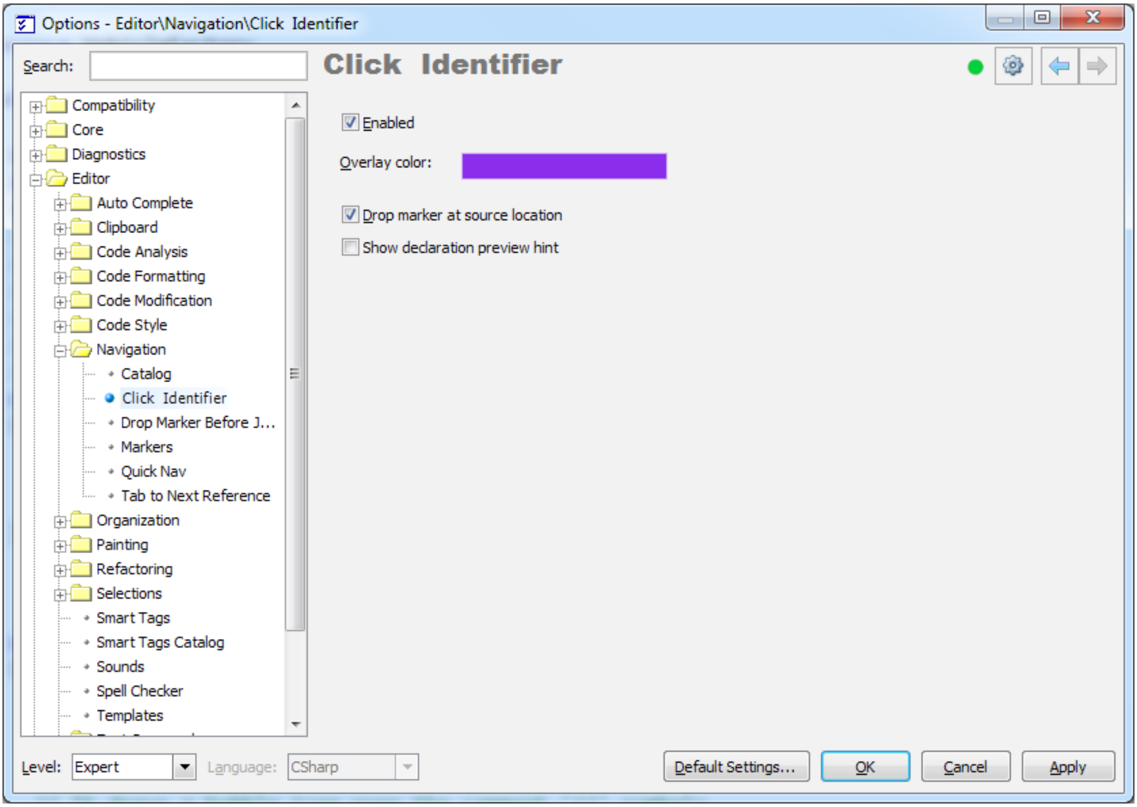
By default, this feature is disabled. To enable it, follow these steps to get to the **Click Identifier** options page:

1. From the DevExpress menu, select "Options…" to open the Options Dialog.

2. In the tree view on the left, navigate to this folder: "***Editor\Navigation***"

3. Select the "***Click Identifier***" options page.

4. Check the "***Enabled***" checkbox.



Options - Editor\Navigation\Click Identifier

Click Identifier

☑ Enabled

Overlay color: 🟪

☑ Drop marker at source location

☐ Show declaration preview hint

Level: Expert

Language: CSharp

There are also a few additional options available:

- Color of the line, which underlines the identifier under the mouse pointer

- Drop a marker at the source location before navigation is made

- Show target declaration preview hint.

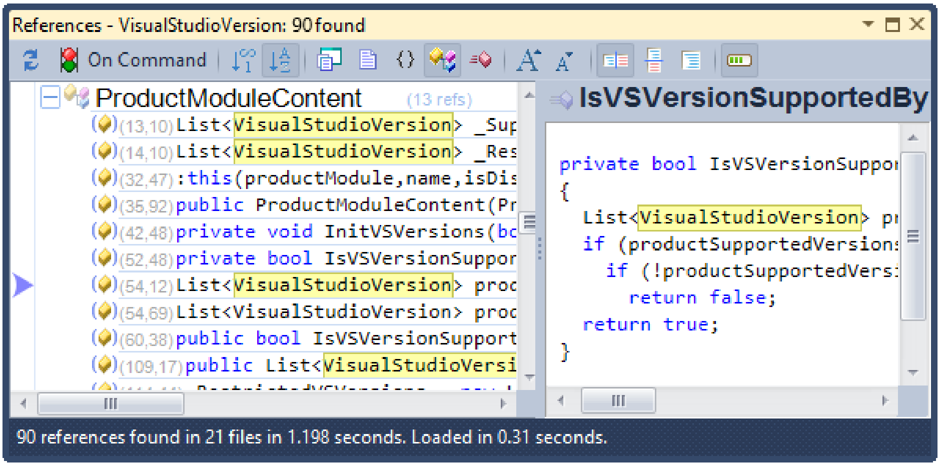
Drop Marker Before Jump

The "**Drop Marker Before Jump**" feature is intended to help you easily navigate back at the source location after the "***Go to Definition***" Visual Studio command is performed. It drops a marker at the text caret location, before you jump to the definition. After you finish navigation to the definitions, pressing the**Esc** key will move you back to each location you have visited.

The "**CR_DropMarkerBeforeJump**" plug-in containing this feature is installed, by default. However, the feature is disabled. You're able to configure it on the "***Editor\Navigation\Drop Marker Before Jump***" options page in the Options Dialog.

The References tool window also known as Find All References

The **References** tool window is designed to search, navigate and review the identifier references in the entire solution. This is what it looks like:



References - VisualStudioVersion: 90 found

```
ProductModuleContent      (13 refs)
  (13,10) List<VisualStudioVersion> _Sup
  (14,10) List<VisualStudioVersion> _Res
  (32,47) :this(productModule,name,isDis
  (35,92) public ProductModuleContent(Pr
  (42,48) private void InitVSVersions(bc
  (52,48) private bool IsVSVersionSuppor
  (54,12) List<VisualStudioVersion> proc
  (54,69) List<VisualStudioVersion> proc
  (60,38) public bool IsVSVersionSupport
  (109,17) public List<VisualStudioVersi
```

IsVSVersionSupportedBy

```
private bool IsVSVersionSuppor
{
  List<VisualStudioVersion> pr
  if (productSupportedVersion:
    if (!productSupportedVers:
      return false;
  return true;
}
```

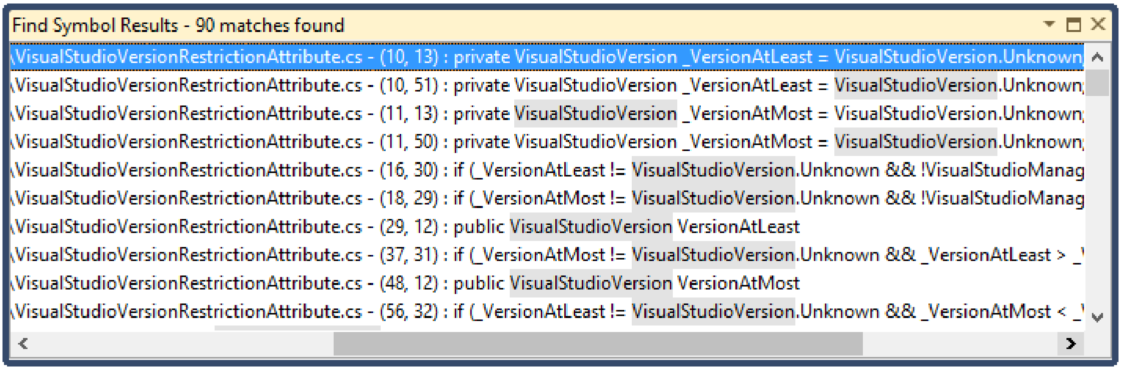
90 references found in 21 files in 1.198 seconds. Loaded in 0.31 seconds.

You can bring it up via the DevExpress | Tool Windows | References menu item or by pressing the Shift + F12 shortcut.

The main window contains a toolbar, the references list, and the code preview area. The layout of the window is configurable using the corresponding tool buttons (Landscape, Portrait, or References Only layouts).

Visual Studio also has a similar built-in feature called Find All Reference (FAR). What's the difference between the CodeRush References tool window and Visual Studio FAR feature?

First, the user interface. Let's compare what we will see in the Visual Studio FAR window while looking for the references of the same identifier as in the CodeRush References screenshot above:



Find Symbol Results - 90 matches found

```
\VisualStudioVersionRestrictionAttribute.cs - (10, 13) : private VisualStudioVersion _VersionAtLeast = VisualStudioVersion.Unknown;
\VisualStudioVersionRestrictionAttribute.cs - (10, 51) : private VisualStudioVersion _VersionAtLeast = VisualStudioVersion.Unknown;
\VisualStudioVersionRestrictionAttribute.cs - (11, 13) : private VisualStudioVersion _VersionAtMost = VisualStudioVersion.Unknown;
\VisualStudioVersionRestrictionAttribute.cs - (11, 50) : private VisualStudioVersion _VersionAtMost = VisualStudioVersion.Unknown;
\VisualStudioVersionRestrictionAttribute.cs - (16, 30) : if (_VersionAtLeast != VisualStudioVersion.Unknown && !VisualStudioManag
\VisualStudioVersionRestrictionAttribute.cs - (18, 29) : if (_VersionAtMost != VisualStudioVersion.Unknown && !VisualStudioManag
\VisualStudioVersionRestrictionAttribute.cs - (29, 12) : public VisualStudioVersion VersionAtLeast
\VisualStudioVersionRestrictionAttribute.cs - (37, 31) : if (_VersionAtMost != VisualStudioVersion.Unknown && _VersionAtLeast > _'
\VisualStudioVersionRestrictionAttribute.cs - (48, 12) : public VisualStudioVersion VersionAtMost
\VisualStudioVersionRestrictionAttribute.cs - (56, 32) : if (_VersionAtLeast != VisualStudioVersion.Unknown && _VersionAtMost < _'
```

Next, one of the most important differences is the speed of reference searching. Here are a couple of measures made inside Visual Studio 2010 on one of the main CodeRush core solutions:

References for:	Visual Studio FAR	CodeRush References
‘Int’ type (20,528 references)	47,305 sec	24,224 sec
‘String’ type (31,805 references)	76,258 sec	54,740 sec
‘TextDocument’ object (609 references)	7,802 sec	2,206 sec

The machine configuration is the following: Intel Core i5 CPU M560 @2.67Ghz, 8.0 Gb, Windows 7 x64.

It seems that the CodeRush References feature might be around 2 to 3 times faster than Visual Studio’s FAR on a medium solution (35 Mbytes of code).

Next, the results organization. In the CodeRush References window, you can do the following:

- group results by projects
- group results by source files
- group results by a namespace
- group results by a class
- group results by a member
- sort references alphabetically
- sort by reference count
- increase/decrease the font size
- see the reference including its parent member’s code preview

In Visual Studio you can:

- see the full file path and member signature of the reference found.

In addition, the CodeRush references window can search for references in two modes:

- Live Sync (automatic mode)
- On Command (manual mode).

If you are in the Live Sync mode, stepping on a reference will automatically look for other references automatically. In the On Command mode, you have to click the Refresh button (or press Shift+F12 again) to search for references. In the Visual Studio FAR feature, there is only a manual mode.

The search mode is toggled on the References window toolbar. Other tool buttons are:

Icon	Caption	Description
↓∞1	Sort by reference count	Sorts the found references by the count in descending order.
↓AZ	Sort alphabetically	Sorts the found references alphabetically in ascending order.
🗗	Show parenting project nodes	Allows you to see and group references by the containing project name.
🗎	Show parenting file nodes	Allows you to see and group references by the containing file name.
{}	Show parenting namespace nodes	Allows you to see and group references by the containing namespace name.
◆	Show parenting class nodes	Allows you to see and group references by the containing class name.
◆	Show parenting member nodes	Allows you to see and group references by the containing member name.
A▴	Increase font size	Makes the references list font bigger.
A▾	Decrease font size	Makes the references list font smaller.
☰	Landscape layout (code preview on right)	Arranges the references list on the left and the code preview area on the right of the window.
☰	Portrait layout (code preview below)	Arranges the references list at the top and the code preview area at the bottom of the window.
☰	References Only (no code preview)	Hides the code preview area.
▭	Visualize references search	Shows a progress indicator while searching for references.

In the reference list, you can double-click the reference to navigate to it inside the code editor. The blue arrow marker shows the currently selected reference. While navigating between references, a temporal white marker is dropped, so you can simply press Esc to navigate back to the previous locations.

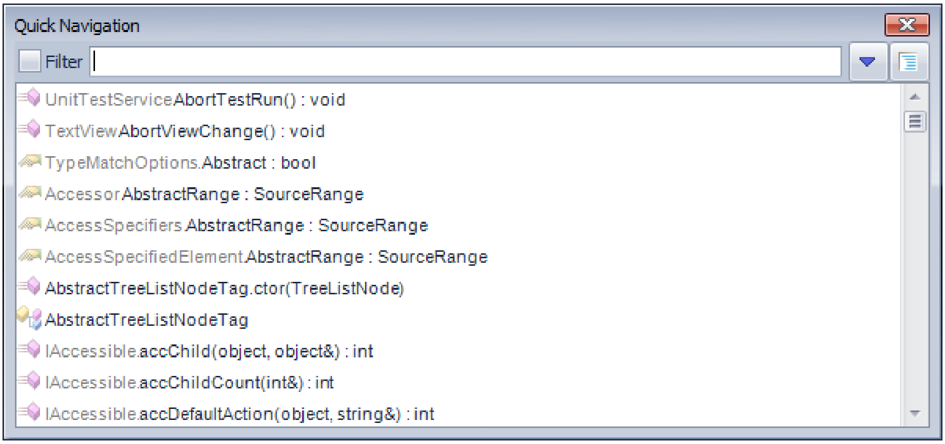
If you right-click the references list, you can access a context menu with the following menu items:

Menu item	Description
Collapse All	Collapses all items in the references list.
Collapse Children	Collapses children items of the selected item in the references list.
Expand All	Expands all items in the references list.
Reference Highlight Color…	Allows you to open the option page for configuring the code preview highlight color (Editor | Painting | Code Previews) in the Options Dialog.
Layout ->	Allows you to choose between window layouts (Landscape, Portrait, No Code Preview).

At the bottom of the window there is summary information about the count of references found and how much time it took to find them all.

Quick Navigation

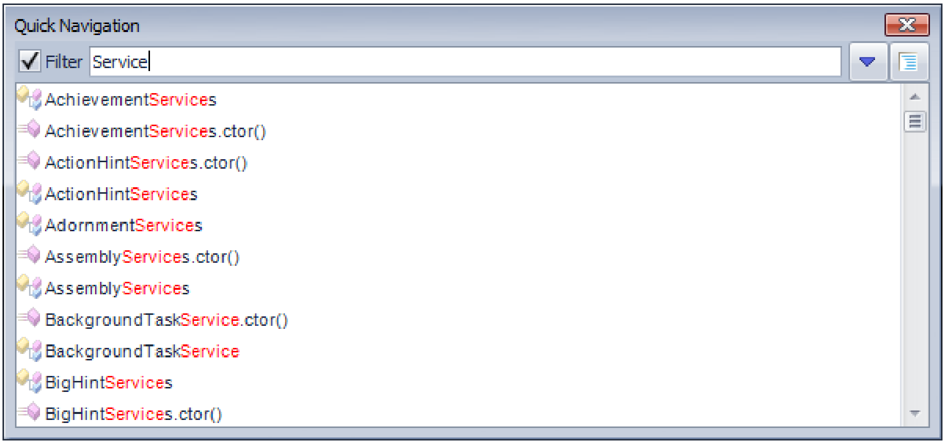
The **Quick Navigation** feature allows you to find any code members inside your solution, such as classes, interfaces, structures, enumerations, delegates, methods, properties, events, fields locals and parameters. Once you hit the Ctrl+Shift+Q shorcut, the simplified Quick Navigation window will appear by default:



```
UnitTestService.AbortTestRun() : void
TextView.AbortViewChange() : void
TypeMatchOptions.Abstract : bool
Accessor.AbstractRange : SourceRange
AccessSpecifiers.AbstractRange : SourceRange
AccessSpecifiedElement.AbstractRange : SourceRange
AbstractTreeListNodeTag.ctor(TreeListNode)
AbstractTreeListNodeTag
IAccessible.accChild(object, object&) : int
IAccessible.accChildCount(int&) : int
IAccessible.accDefaultAction(object, string&) : int
```

The window is not modal and is not dockable, so that you can quickly hide it when you have navigated to the desired member or the window focus is lost.

The **Quick Nav** window has a Filter checkbox that indicates that the content of the members list is filtered by the member name entered in the text box right after the filter:



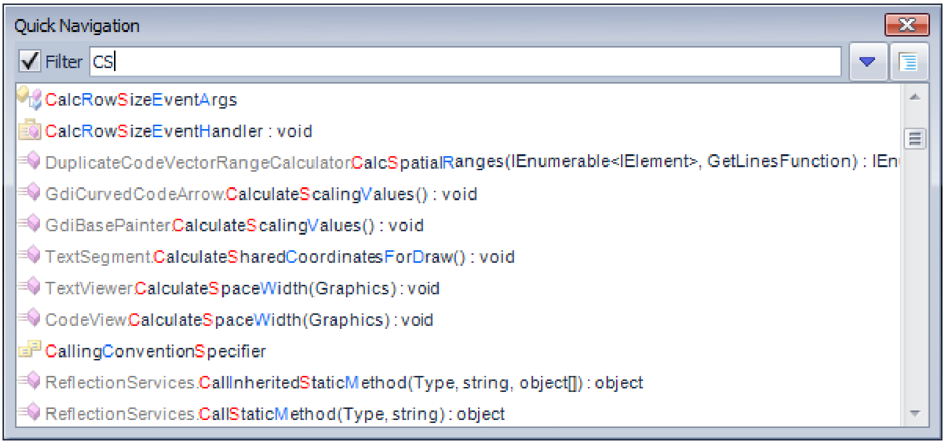
☑ Filter Service

```
AchievementServices
AchievementServices.ctor()
ActionHintServices.ctor()
ActionHintServices
AdornmentServices
AssemblyServices.ctor()
AssemblyServices
BackgroundTaskService.ctor()
BackgroundTaskService
BigHintServices
BigHintServices.ctor()
```

Near the filter text value there are two options:

- Show additional filter options;
- Preview target location (by holding down the Ctrl key).

The main area of the window is the members list. The initial members list consists of all members if it is not filtered by the name or member type. The filter by the name text box has advanced filtering capabilities, such camel-case filtering. If you type an uppercase letter, the corresponding member names will be shown, e.g.:

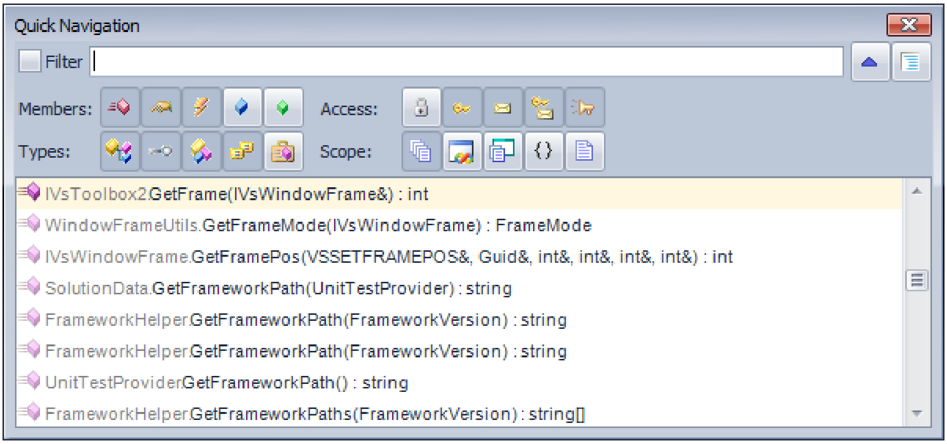


CS

CalcRowSizeEventArgs
CalcRowSizeEventHandler : void
DuplicateCodeVectorRangeCalculator.CalcSpatialRanges(IEnumerable<IElement>, GetLinesFunction) : IEn
GdiCurvedCodeArrow.CalculateScalingValues() : void
GdiBasePainter.CalculateScalingValues() : void
TextSegment.CalculateSharedCoordinatesForDraw() : void
TextViewer.CalculateSpaceWidth(Graphics) : void
CodeView.CalculateSpaceWidth(Graphics) : void
CallingConventionSpecifier
ReflectionServices.CallInheritedStaticMethod(Type, string, object[]) : object
ReflectionServices.CallStaticMethod(Type, string) : object

The uppercase letter entered in the text filter box are highlighted in red, and the other uppercase letters are highlighted in blue, so you can continue typing suggested letter candidates.

The advanced filtering options are accessible by pressing the ‘Show advanced filtering options’ button. There are four advanced filters:



IVsToolbox2.GetFrame(IVsWindowFrame&) : int
WindowFrameUtils.GetFrameMode(IVsWindowFrame) : FrameMode
IVsWindowFrame.GetFramePos(VSSETFRAMEPOS&, Guid&, int&, int&, int&, int&) : int
SolutionData.GetFrameworkPath(UnitTestProvider) : string
FrameworkHelper.GetFrameworkPath(FrameworkVersion) : string
FrameworkHelper.GetFrameworkPath(FrameworkVersion) : string
UnitTestProvider.GetFrameworkPath() : string
FrameworkHelper.GetFrameworkPaths(FrameworkVersion) : string[]

By members:

Icon	Filter	Shortcut
◆	Methods	Alt+M
✋	Properties	Alt+P
⚡	Events	Alt+V
◆	Fields	Alt+F
◆	Locals and parameters	Alt+R

By types:

Icon	Filter	Shortcut
	Classes	Alt+C
	Interfaces	Alt+I
	Structures	Alt+S
	Enumerations	Alt+E
	Delegates	Alt+D

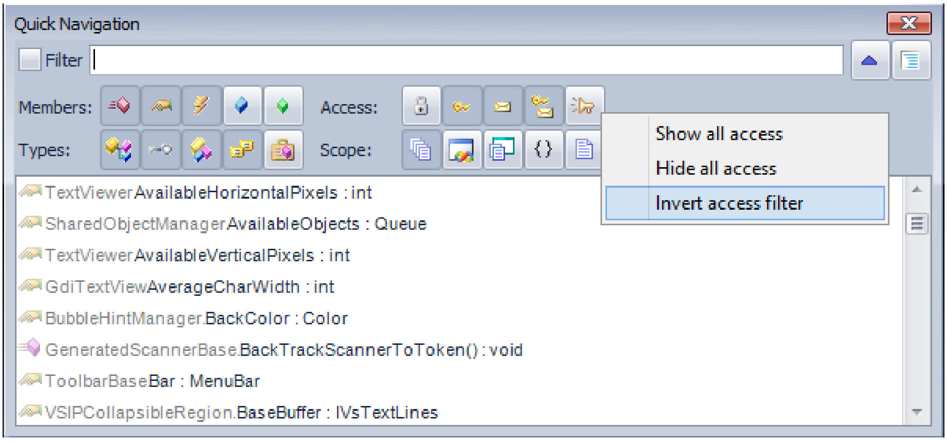
By access (visibility):

Icon	Filter	Shortcut
	Private	Alt+A
	Protected	Alt+O
	Internal	Alt+N
	Protected Internal	Alt+T
	Public	Alt+B

By scope:

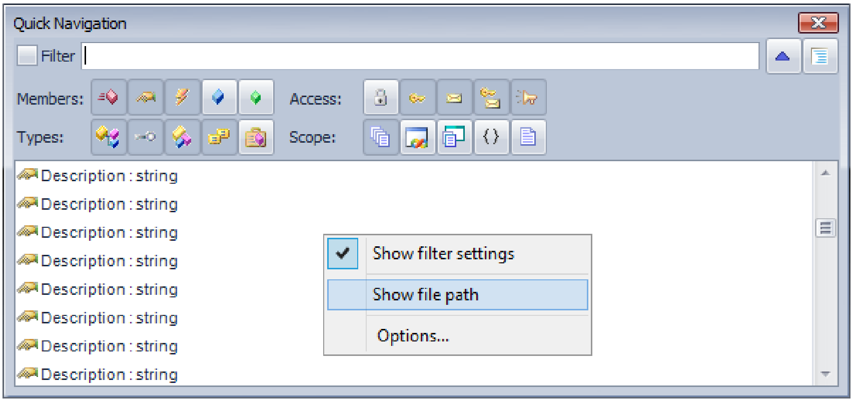
Icon	Filter	Shortcut
	All files (incl. (miscellaneous	Alt+Shift+A
	Current Solution	Alt+Shift+S
	Current Project	Alt+Shift+P
{ }	Current Namespace	Alt+Shift+N
	Current File	Alt+Shift+F

If you right-click the filters, you can change the state of the filters group by choosing the corresponding menu item:



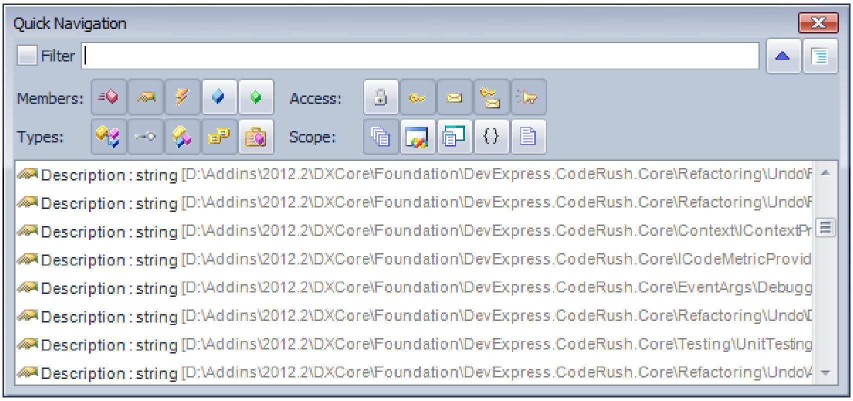
TextViewerAvailableHorizontalPixels : int
SharedObjectManagerAvailableObjects : Queue
TextViewerAvailableVerticalPixels : int
GdiTextViewAverageCharWidth : int
BubbleHintManager.BackColor : Color
GeneratedScannerBase.BackTrackScannerToToken() : void
ToolbarBaseBar : MenuBar
VSIPCollapsibleRegion.BaseBuffer : IVsTextLines

Once all filters are tweaked as required, the members list will contain corresponding members. If there are members with equal names, you can show a full file name by right-clicking the members list and choosing the “Show qualifiers…” menu item:



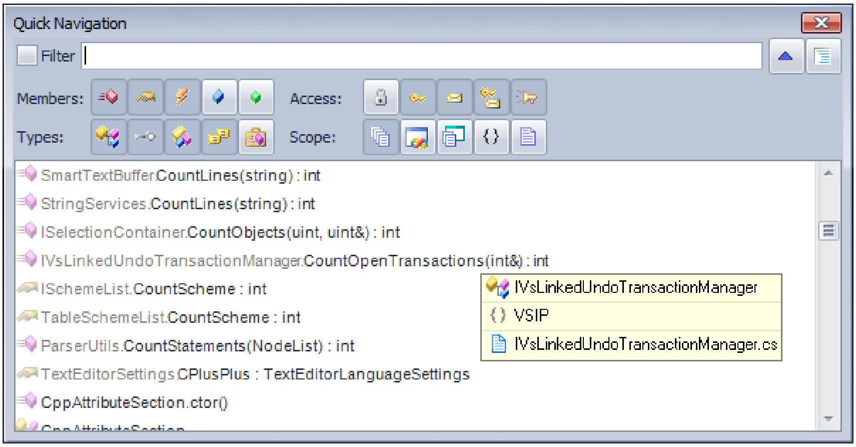
Description : string
Description : string
Description : string
Description : string
Description : string
Description : string
Description : string
Description : string

Enabling this option will display the full file paths to each member:



```
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Refactoring\Undo\F
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Refactoring\Undo\F
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Context\IContextPr
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\ICodeMetricProvid
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\EventArgs\Debugg
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Refactoring\Undo\I
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Testing\UnitTesting
Description : string [D:\Addins\2012.2\DXCore\Foundation\DevExpress.CodeRush.Core\Refactoring\Undo\/
```

Hovering over a member name in the list will also show you an additional hint that displays the location of the member (e.g., class, namespace, file):



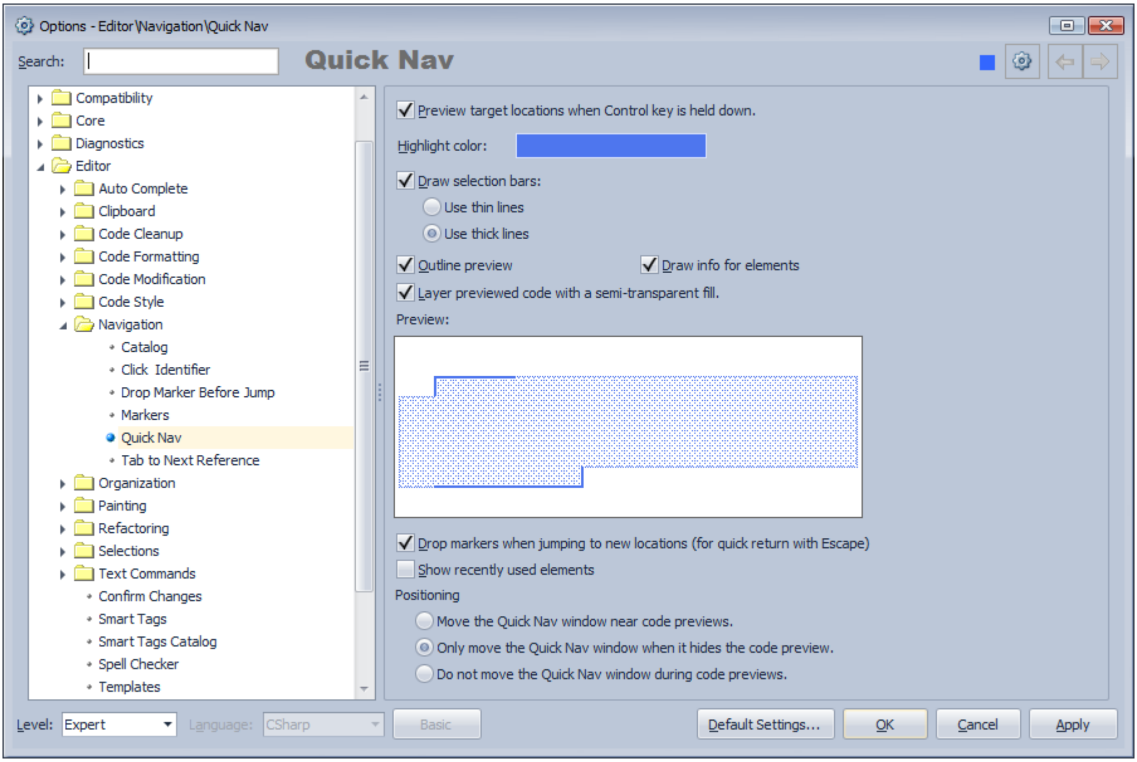
```
SmartTextBuffer.CountLines(string) : int
StringServices.CountLines(string) : int
ISelectionContainer.CountObjects(uint, uint&) : int
IVsLinkedUndoTransactionManager.CountOpenTransactions(int&) : int
ISchemeList.CountScheme : int
TableSchemeList.CountScheme : int
ParserUtils.CountStatements(NodeList) : int
TextEditorSettings.CPlusPlus : TextEditorLanguageSettings
CppAttributeSection.ctor()
CppAttributeSection
```

IVsLinkedUndoTransactionManager
VSIP
IVsLinkedUndoTransactionManager.cs

The second option on top near the name filter allows you to preview the chosen member by holding down the Ctrl key as follows:
The Quick Nav window and member preview are customizable. See the corresponding topic to learn more on how to customize the window and member preview, and create dedicated keyboard shortcuts with the predefined advanced filters.

Advanced CodeRush Quick Navigation customization

The Quick Navigation window is customizable on the Editor | Navigation | Quick Nav option page in the CodeRush Options Dialog accessible via the DevExpress menu:



☑ Preview target locations when Control key is held down.

Highlight color: 🟦

☑ Draw selection bars:

○ Use thin lines

● Use thick lines

☑ Outline preview ☑ Draw info for elements

☑ Layer previewed code with a semi-transparent fill.

Preview:

☑ Drop markers when jumping to new locations (for quick return with Escape)

☐ Show recently used elements

Positioning

○ Move the Quick Nav window near code previews.

● Only move the Quick Nav window when it hides the code preview.

○ Do not move the Quick Nav window during code previews.

Level: Expert Language: CSharp

Here are available options:

[X] Preview target locations when Control key is held down

This option specifies the default value for the member preview availability when holding down the Ctrl key on the active member inside the **Quick Nav** member list.

The 'Highlight color' color option is self-described.

[X] Draw selection bars

This option specifies the type of lines drawn at the top left and bottom-right corners of the preview whether they are thin or thick.

[X] Outline preview

This option specifies whether or not the border of the member preview should be highlighted.

[X] Draw qualifiers for elements

This option specifies wether or not to show a member full file path.

[X] Layer previewed code with a semi-transparent fill

This option specifies whether or not to fill a background of the member preview.

The preview area allows you to see the resulting member preview when changing the corresponding preview options.

[X] Drop markers when jumping to new locations (for quick return with Escape)

This option specifies whether or not to leave markers to be able to quickly get back to the starting position in code.

[X] Show recently used elements

The ‘Positioning’ options specify the starting location of the Quick Navigation window in relation to the member preview.

In addition to these options, you can create custom keyboard shortcuts to open Quick Navigation with the predefined advanced filters. This is configurable on the IDE | Shortcuts option page in the CodeRush Options Dialog.

You can create new shortcuts for the **QuickNav** command and specify a combination of the following parameters:

Type filter – specifies the ‘ by types’ filter with the following values: Classes, Interfaces, Structs, Enums, Delegates, AllTypes.

Member filter – specifies the ‘by members’ filter with the following values: Methods, Properties, Events, Fields, LocalsAndParams, AllMembers.

Access filter – specifies the ‘by access (visibility)’ filter with the following values: Private, Protected, Internal, ProtectedInternal, Public, AllVisibilities.

Scope filter – specifies the ‘by scope’ filter with the following values: AllFiles, CurrentSolution, CurrentProject, CurrentNamespace, CurrentFile.

You can combine each of the filters with the ‘and’ keyword. The advanced filter options are specified without quotes in the parameters of the shortcut and should be specified consequently, separated with a comma, for example:

Find everything everywhere: “AllTypes, AllMembers, AllVisibilities, AllFiles”.

Find global methods, properties and events in a solution: “, Methods and Properties and Events, Public and Internal, CurrentSolution”. Note that the type filter is skipped by specifying just a comma.

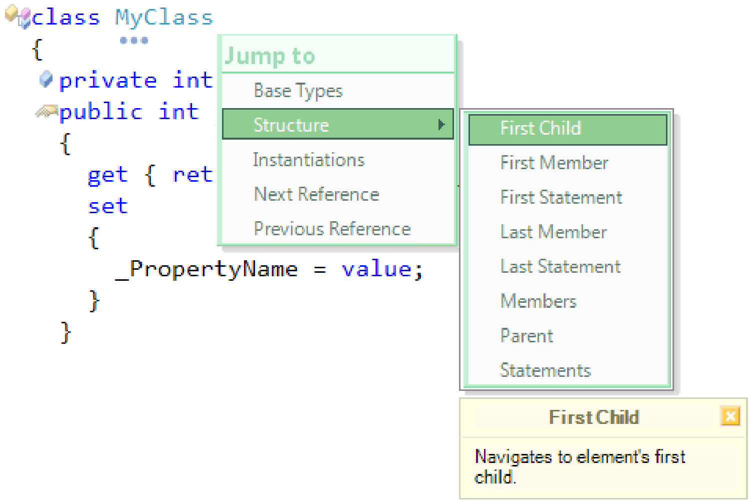
Find private fields in a project: “, Fields, Private, CurrentProject”. Note that in this case the type filter is skipped as well.

The Quick Nav feature will save the parameters, so you may call the window with the default shortcut with the same filter options as the previously opened window.

Navigation providers overview

CodeRush **Navigation Providers Engine** is an extensible architecture that allows you to easily navigate inside your code structure and particular code fragments.

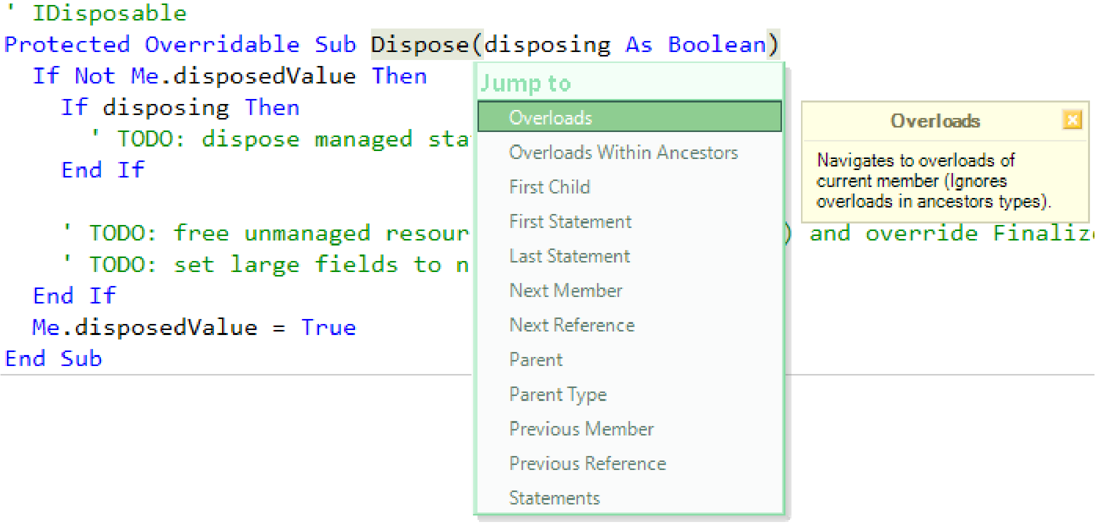
There are dozens of nav providers shipped with **CodeRush**. To see available navigation providers in the current context, press the **Ctrl**+**Alt**+**N** shortcut inside the code editor, and the “**Jump to**” popup menu appears:



```
class MyClass
  {
   private int
   public int
    {
      get { ret
      set
      {
       _PropertyName = value;
      }
    }
```

Navigates to element's first child.

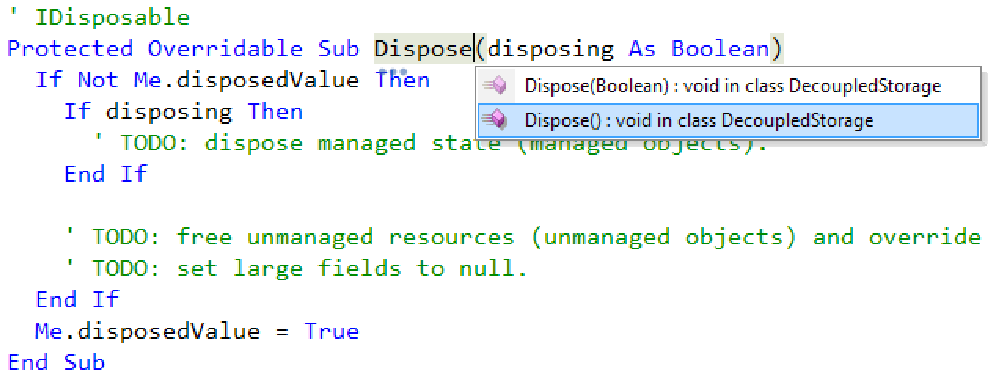
or



```
' IDisposable
Protected Overridable Sub Dispose(disposing As Boolean)
  If Not Me.disposedValue Then
    If disposing Then
      ' TODO: dispose managed sta
    End If

    ' TODO: free unmanaged resour                          ) and override Finaliz
    ' TODO: set large fields to n
  End If
  Me.disposedValue = True
End Sub
```

Navigates to overloads of current member (Ignores overloads in ancestors types).

After an item from the list is executed, in most cases, you will immediately navigate to the target position. But if there are several targets for navigation, the additional popup menu with the list of these targets will be shown:



```
' IDisposable
Protected Overridable Sub Dispose(disposing As Boolean)
  If Not Me.disposedValue Then
    If disposing Then
      ' TODO: dispose managed state (managed objects).
    End If

    ' TODO: free unmanaged resources (unmanaged objects) and override
    ' TODO: set large fields to null.
  End If
  Me.disposedValue = True
End Sub
```

Dispose(Boolean) : void in class DecoupledStorage
Dispose() : void in class DecoupledStorage

A marker is dropped at the source location, when you navigate to the specific target. You can press the **Esc** key to return to the initial position.

You can also bind your favorite navigation providers to a specific shortcut key using the **Navigate** command.

CodeRush Navigation providers list

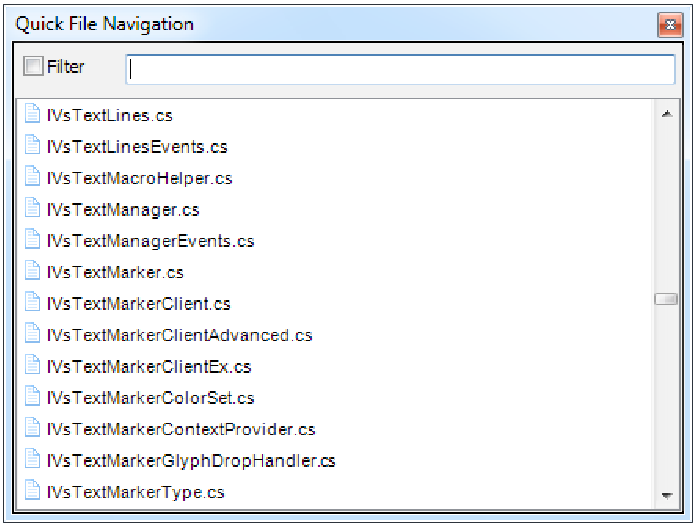
Here is the list of navigation providers shipped with the latest CodeRush Pro version. Press the **Ctrl+Alt+N** shortcut inside the code editor, to see all available navigation providers in the current context. You can use the provider name to bind it to a particular shortcut key using the **Navigate** command.

Name	Description
Abstract Member Ancestor	Navigates to the abstract member ancestor of the current member implementation.
Ancestor	Navigates to the ancestor type declaration.
Base Types	Navigates to base types of the current class (doesn't list base type of ancestors).
Base Types within Ancestors	Navigates to base types of the current class including base types of ancestors.
Declaration	Navigates to the declaration of the current identifier.
Descendants	Navigates to descendant type declarations.
Enum Element	Navigates to enum elements of the current enumeration declaration.
First Child	Navigates to the first child of the current language element.
First Enum Element	Navigates to the first enum element of the current enumeration declaration.
First Member	Navigates to the first member in the current type declaration.
First Statement	Navigates to the first statement in the current scope.
Flow Breaks	Navigates to flow breaks in the current scope.
Implementations	Navigates to method implementations for the current method reference.
Implementers	Navigates to implementers of the current interface declaration.
Last Enum Element	Navigates to the last enum element of the current enumeration declaration.
Last Member	Navigates to the last member in the current type declaration.
Last Statement	Navigates to the last statement in the current scope.

Members	Navigates to members of the current type declaration. Nested type declarations are ignored.
MVC View	Navigates to the corresponding MVC view.
MVC Controller	Navigates to the corresponding MVC controller.
MVC Action	Navigates to the corresponding MVC action.
Next Enum Element	Navigates to the next enum element of the current enumeration declaration.
Next Member	Navigates to the next member in the current type declaration.
Next Reference	Navigates to the next references of the current identifier.
Next Statement	Navigates to the next statement in the current scope.
Overloads	Navigates to overloaded members of the current member in the current type.
Overloads within Ancestors	Navigates to overloaded members of the current member in the current and ancestor types.
Overrides	Navigates to overridden members of the current member.
Parent	Navigates to the parent language element.
Parent Member	Navigates to the parent member.
Parent Type	Navigates to the parent type declaration.
Previous Enum Element	Navigates to the previous enum element of the current enumeration declaration.
Previous Member	Navigates to the previous member in the current type declaration.
Previous Reference	Navigates to the previous references of the current identifier.
Previous Statement	Navigates to the previous statement in the current scope.
Statements	Navigates to the statements of the current scope.
Virtual Member Ancestor	Navigates to the virtual member ancestor of the current member implementation.

Quick File Navigation

Quick File Nav navigation CodeRush feature allows you to switch between all files in a solution. The shortcut key for this tool window is the **Ctrl+Alt+F**. Once you press the shortcut, the following window appears at the editor caret position or at the center of the Visual Studio main window:



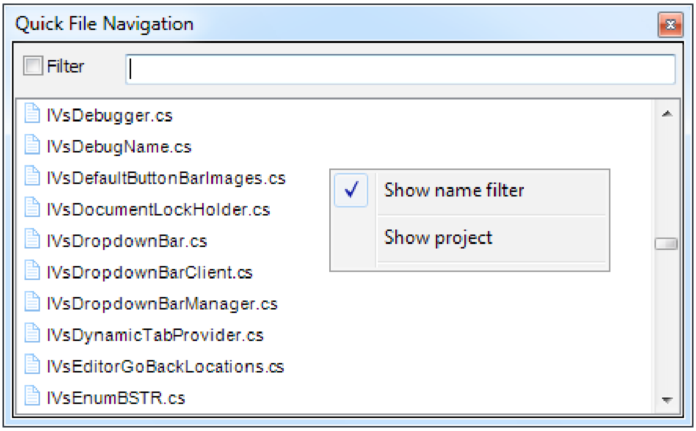
IVsTextLines.cs
IVsTextLinesEvents.cs
IVsTextMacroHelper.cs
IVsTextManager.cs
IVsTextManagerEvents.cs
IVsTextMarker.cs
IVsTextMarkerClient.cs
IVsTextMarkerClientAdvanced.cs
IVsTextMarkerClientEx.cs
IVsTextMarkerColorSet.cs
IVsTextMarkerContextProvider.cs
IVsTextMarkerGlyphDropHandler.cs
IVsTextMarkerType.cs

The window consists of two parts – the filter box, which can be hidden, and the files list sorted alphabetically. If you hover over a file name – its full path will be shown.

There are only two options you can tweak:

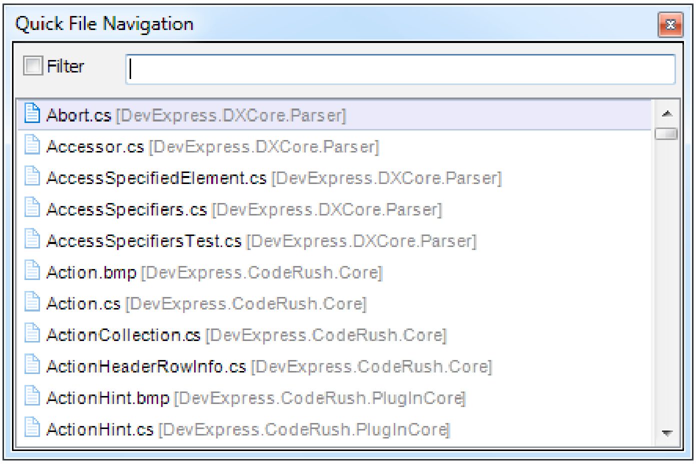
- toggle the visibility of the filter box (enabled, by default)
- toggle the visibility of a project name caption near a file name (disabled, by default)

These options are available via the right-click context menu:



IVsDebugger.cs
IVsDebugName.cs
IVsDefaultButtonBarImages.cs
IVsDocumentLockHolder.cs
IVsDropdownBar.cs
IVsDropdownBarClient.cs
IVsDropdownBarManager.cs
IVsDynamicTabProvider.cs
IVsEditorGoBackLocations.cs
IVsEnumBSTR.cs

With both options turned on, the window will look like this:



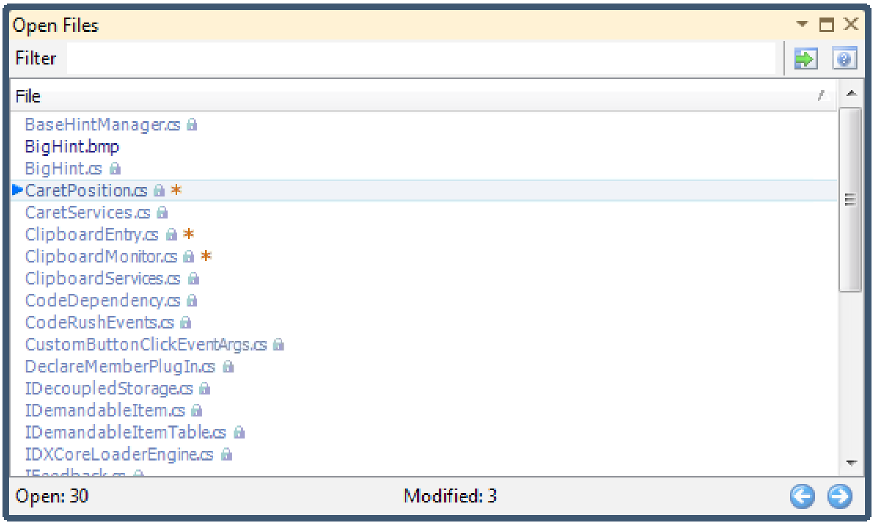
Abort.cs [DevExpress.DXCore.Parser]
Accessor.cs [DevExpress.DXCore.Parser]
AccessSpecifiedElement.cs [DevExpress.DXCore.Parser]
AccessSpecifiers.cs [DevExpress.DXCore.Parser]
AccessSpecifiersTest.cs [DevExpress.DXCore.Parser]
Action.bmp [DevExpress.CodeRush.Core]
Action.cs [DevExpress.CodeRush.Core]
ActionCollection.cs [DevExpress.CodeRush.Core]
ActionHeaderRowInfo.cs [DevExpress.CodeRush.Core]
ActionHint.bmp [DevExpress.CodeRush.PlugInCore]
ActionHint.cs [DevExpress.CodeRush.PlugInCore]

The filtering works the same way as in all other file-navigation windows, such as Browse Recent Files and Open Files. In the filter box, you have to type the file mask, for example "*.resx" (without quotes). Camel-case filtering is also available, so typing "AI" will find the "*AssemblyInfo.cs*" file, where "*AI*" will identify two words: "*Assembly*" and "*Info*".

Bear in mind, that this tool window is not modal, and it will be hidden once it loses focus. The *Esc* key will hide the window as well. *Up* and *Down* arrow keys are suitable for navigating between files in the list.

Open Files tool window

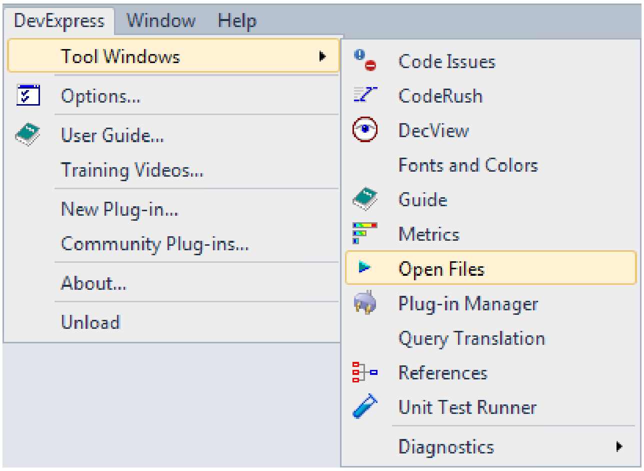
The **Open Files** CodeRush tool window lists files that are currently opened inside the Visual Studio IDE. The window allows you to quickly switch between files via a single mouse click. This window is similar to the Ctrl+Tab Visual Studio built-in window but has a few advanced options. This is what it looks like:



BaseHintManager.cs
BigHint.bmp
BigHint.cs
CaretPosition.cs *
CaretServices.cs
ClipboardEntry.cs *
ClipboardMonitor.cs *
ClipboardServices.cs
CodeDependency.cs
CodeRushEvents.cs
CustomButtonClickEventArgs.cs
DeclareMemberPlugIn.cs
IDecoupledStorage.cs
IDemandableItem.cs
IDemandableItemTable.cs
IDXCoreLoaderEngine.cs

Open: 30 Modified: 3

To open the window, click the **DevExpress** | **Tool Windows** | **Open files** menu item:



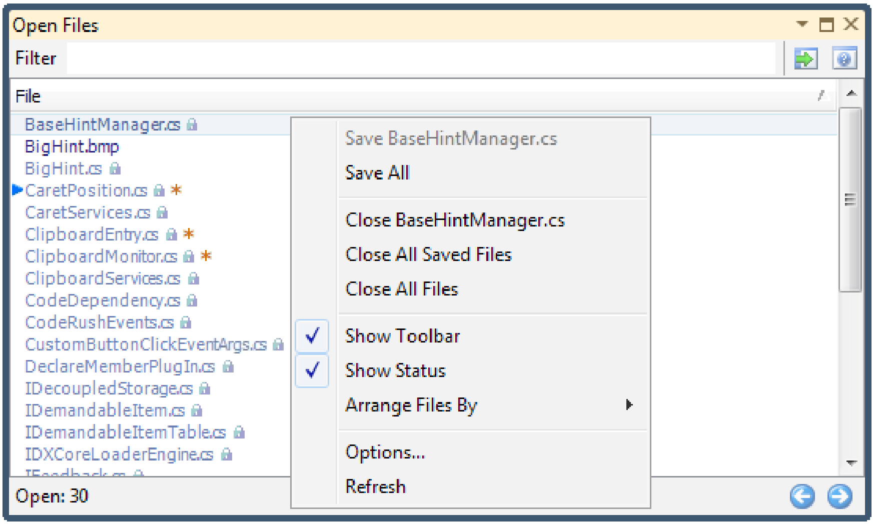
As always, you can create you own shortcut to toggle the visibility of this tool window. The name of the action is “OpenFilesWindowToggle”.

The window contains of three parts:

- The Filter toolbar with a couple of tool buttons
- The Files list
- The Status bar

The main part of the window contains the list of opened files. The list is being populated automatically once a file is opened inside the IDE. The currently active file is highlighted in a light blue color with a dark blue triangle on the left. If a file is modified, a red asterisk is shown over the file name. You can also see the ‘lock’ icon if the file has the read-only attribute set or locked by the source version control. Clicking the file name or pressing Enter will activate the file inside the IDE.

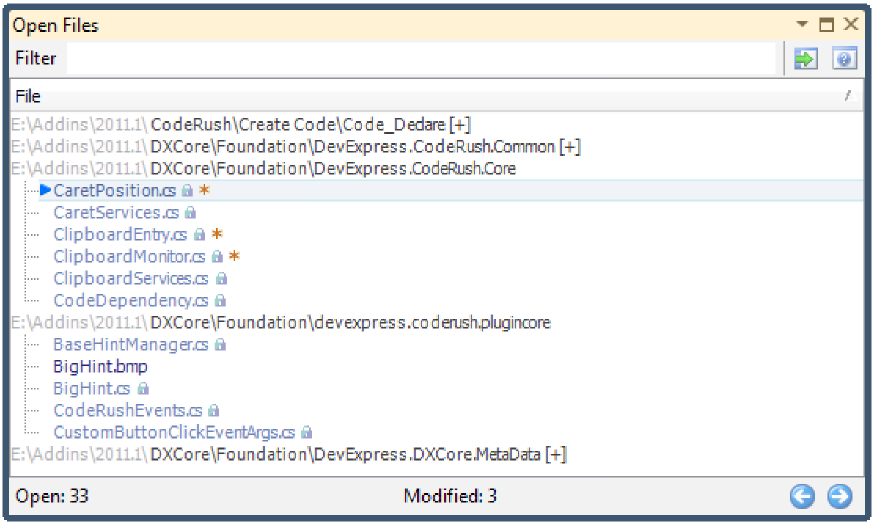
The list has a right-click context menu with additional options:



Note that the visibility of some options depends on the node you have selected in the list – a file node, a group of nodes (parent node with children) or no node.

Save “FileName”	Saves the highlighted file if one is modified. If the highlighted file is not modified, the option is disabled.
Save Group	Saves the selected group of files.
Save All	Saves all modified files.
Close “FileName”	Closes the highlighted file.
Close Group	Closes the selected group of files.
Close All Saved Files	Closes all files that are not modified.
Close All Files	Closes all files no matter whether modified or not.
Show Toolbar	Toggles the visibility of the Filter toolbar and its buttons.
Show Status	Toggles the visibility of the status bar.
Arrange Files By	Allows you to arrange files in the list by project, path or show a plain list of file names.
Options	Opens the options page inside the Options Dialog with advanced settings for the tool window.
Refresh	Refreshes the list of files. Click it if there are missing files in the list by some reason.

If you have arranged files by the project or the full path, you will see the appropriate parent nodes in the files list, for example:



```
E:\Addins\2011.1\CodeRush\Create Code\Code_Dedare [+]
E:\Addins\2011.1\DXCore\Foundation\DevExpress.CodeRush.Common [+]
E:\Addins\2011.1\DXCore\Foundation\DevExpress.CodeRush.Core
    CaretPosition.cs *
    CaretServices.cs
    ClipboardEntry.cs *
    ClipboardMonitor.cs *
    ClipboardServices.cs
    CodeDependency.cs
E:\Addins\2011.1\DXCore\Foundation\devexpress.coderush.plugincore
    BaseHintManager.cs
    BigHint.bmp
    BigHint.cs
    CodeRushEvents.cs
    CustomButtonClickEventArgs.cs
E:\Addins\2011.1\DXCore\Foundation\DevExpress.DXCore.MetaData [+]
```

Open: 33 Modified: 3

You can collapse and expand parent nodes by double clicking them. If a node is collapsed, you will see the [+] sign at the end of the node's caption.

If you click the header of the list you can toggle its sorting: ascending or descending.

The status bar shows the number of opened files and the number of modified files. There are also two navigation buttons to switch between files in the list. The first button activates the previous file in the list, and the second one activates the next file in the list. There are also two additional actions corresponding to these buttons:

- OpenFilesSelectNext – activates the next file from the list;

- OpenFilesSelectPrevious – activates the previous file from the list.

The Filter tool box allows you to enter the file name specific filter. For example, typing "**.resx*" in the text box will show only the files with the "*resx*" extension. You can type upper case letters in the filter box and this will find file names having those letters in the name, e.g. typing "*AI*" will find "*AssemblyInfo*" files.

To the right of the Filter text box there are customizable tool buttons. By default, there are two buttons:

- Show Start Page – opens the Visual Studio Start Page that allows you to create, open or choose a recent project for opening.

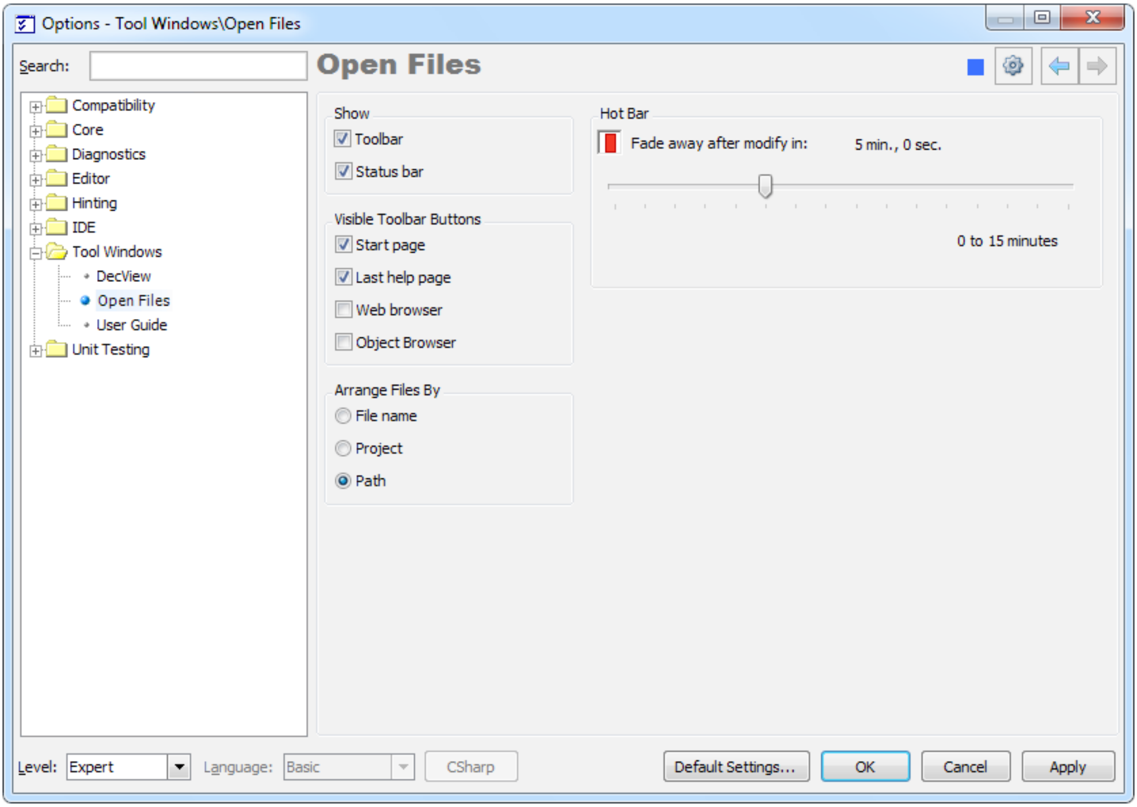
- Last Help Page – opens theVisual Studio integrated help system with the last visited page.

On the **Open Files** options page you can enable two more buttons:

- Web browser – opens Visual Studio integrated web browser

- Object browser – open Visual Studio Object browser tool window

Options of the tool window inside the Options Dialog are similar to options that you can tweak using the right-click context menu of the files list. Here is what the options page for the window looks like:



Show
☑ Toolbar
☑ Status bar

Visible Toolbar Buttons
☑ Start page
☑ Last help page
☐ Web browser
☐ Object Browser

Arrange Files By
○ File name
○ Project
● Path

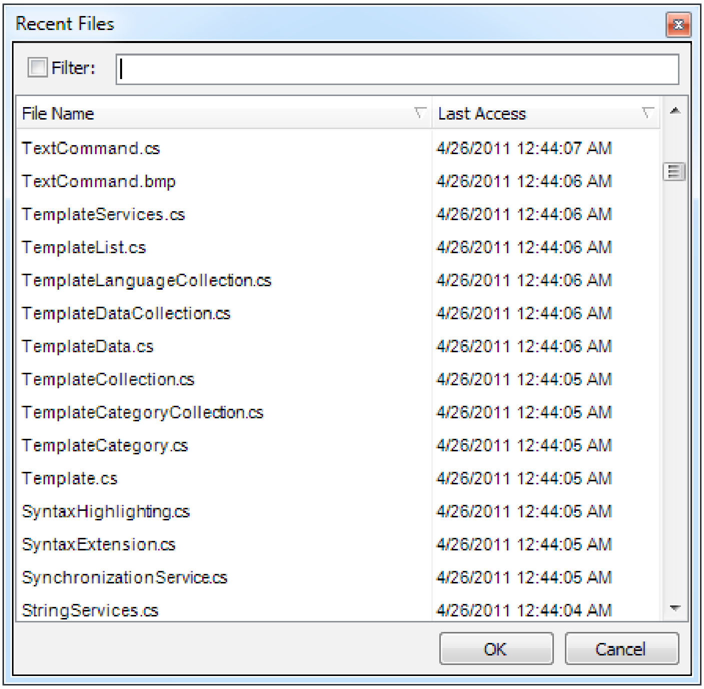
Hot Bar
Fade away after modify in: 5 min., 0 sec.
0 to 15 minutes

Level: Expert
Language: Basic
CSharp

The *Show* and *Arrange File By* options are the same as inside the context menu. The *Visible Toolbar Buttons* allows you to turn on or off buttons on the *Filter* toolbar. The *Hot Bar* option automatically refreshes the list of files within the specified amount of time.

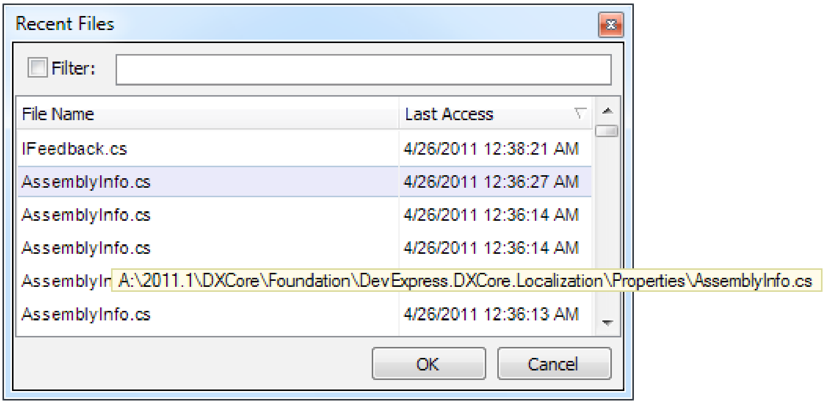
Browsing recently accessed files

CodeRush has a special tool window for browsing recently opened (accessed) files inside Visual Studio called **Browse Recent Files**. It is available via the corresponding **File** -> **Browse Recent Files**… menu item, or via the **Ctrl**+**Shift**+**.** (dot at the end) shortcut. This is what it looks like:



File Name	Last Access
TextCommand.cs	4/26/2011 12:44:07 AM
TextCommand.bmp	4/26/2011 12:44:06 AM
TemplateServices.cs	4/26/2011 12:44:06 AM
TemplateList.cs	4/26/2011 12:44:06 AM
TemplateLanguageCollection.cs	4/26/2011 12:44:06 AM
TemplateDataCollection.cs	4/26/2011 12:44:06 AM
TemplateData.cs	4/26/2011 12:44:06 AM
TemplateCollection.cs	4/26/2011 12:44:05 AM
TemplateCategoryCollection.cs	4/26/2011 12:44:05 AM
TemplateCategory.cs	4/26/2011 12:44:05 AM
Template.cs	4/26/2011 12:44:05 AM
SyntaxHighlighting.cs	4/26/2011 12:44:05 AM
SyntaxExtension.cs	4/26/2011 12:44:05 AM
SynchronizationService.cs	4/26/2011 12:44:05 AM
StringServices.cs	4/26/2011 12:44:04 AM

The majority of the window contains the list of files. By default, it stores the list of 500 files, which is configurable. To activate the file, double click it, or press the *Enter* key. If you select several files – you can open all of them at once with the *Enter* key. If you use the mouse, you can see the full path to the file when hovering over it. This allows you to choose the correct file, if there are different files with the same name:



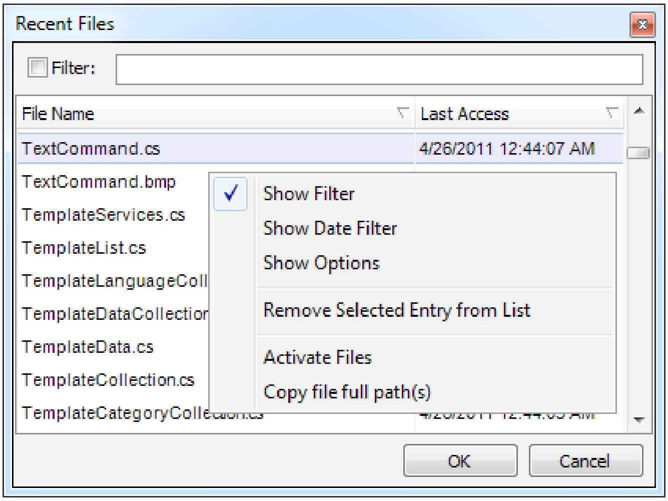
File Name	Last Access
IFeedback.cs	4/26/2011 12:38:21 AM
AssemblyInfo.cs	4/26/2011 12:36:27 AM
AssemblyInfo.cs	4/26/2011 12:36:14 AM
AssemblyInfo.cs	4/26/2011 12:36:14 AM
AssemblyIr A:\2011.1\DXCore\Foundation\DevExpress.DXCore.Localization\Properties\AssemblyInfo.cs	
AssemblyInfo.cs	4/26/2011 12:36:13 AM

The list of files contains two columns:

- File Name – shows a short name of a file.
- Last Access – shows the full date and time of the last access to a file.

You can sort any of these columns ascending or descending.

The list has a context menu available via right-clicking anywhere on the list:



File Name	Last Access
TextCommand.cs	4/26/2011 12:44:07 AM
TextCommand.bmp	
TemplateServices.cs	
TemplateList.cs	
TemplateLanguageColl	
TemplateDataCollectior	
TemplateData.cs	
TemplateCollection.cs	
TemplateCategoryCollection.cs	4/26/2011 12:44:05 AM

with the following items:

- Show Filter

Toggles the visibility of the filter text box at the top of the tool window.

- Show Date Filter

Toggles the visibility of the additional Date filter.

- Show Options

Opens the corresponding options page in the Options Dialog with additional settings for the tool window.

- Remove Selected Entry from List

Removes the selected entry or a number of selected entries from the list.

- Activate Files

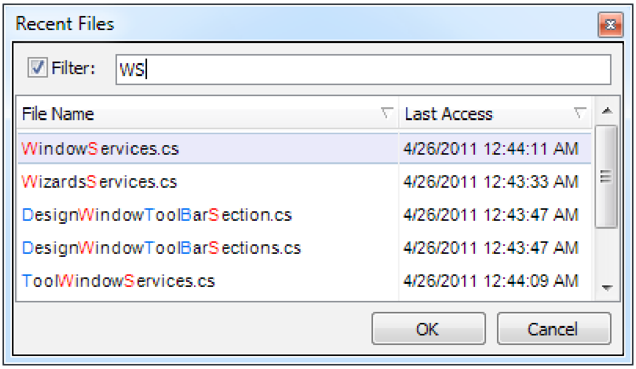
Activates the selected file or a number of selected files at once.

- Copy file full path(s)

Copies the fill path to the file into Clipboard.

At the top of the window, there's a filter text box, where you can type a filter for a file name. The filter contains two parts – the name or mask of a file and its extension. You can use the '*' wildcard, which is also used as a default file extension. For example, typing "A*.cs" (without quotes) will find all files with the ".cs" extension and the file name starting with the 'A' letter.

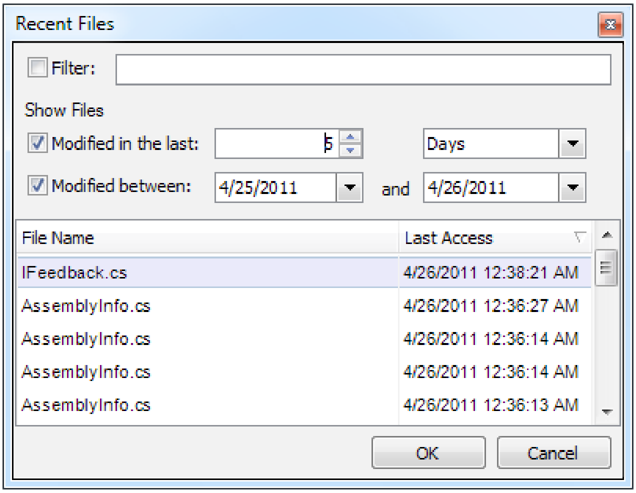
The filter text box also supports camel-case filtering. This kind of filtering works the following way: the upper-case letters specify the starting letters of words in a file name. For example, typing "AI" will find the "AssemblyInfo.cs", where "AI" identifies two words: "Assembly" and "Info". The starting letters will be highlighted in red and the remainder, not specified in camel-case filter search, will be highlighted in blue:



☑ Filter: WS

File Name	Last Access
WindowServices.cs	4/26/2011 12:44:11 AM
WizardsServices.cs	4/26/2011 12:43:33 AM
DesignWindowToolBarSection.cs	4/26/2011 12:43:47 AM
DesignWindowToolBarSections.cs	4/26/2011 12:43:47 AM
ToolWindowServices.cs	4/26/2011 12:44:09 AM

In addition to file names filter, there's the Date filter available, once you enable it via the context menu:



☐ Filter:

Show Files

☑ Modified in the last: 5 Days

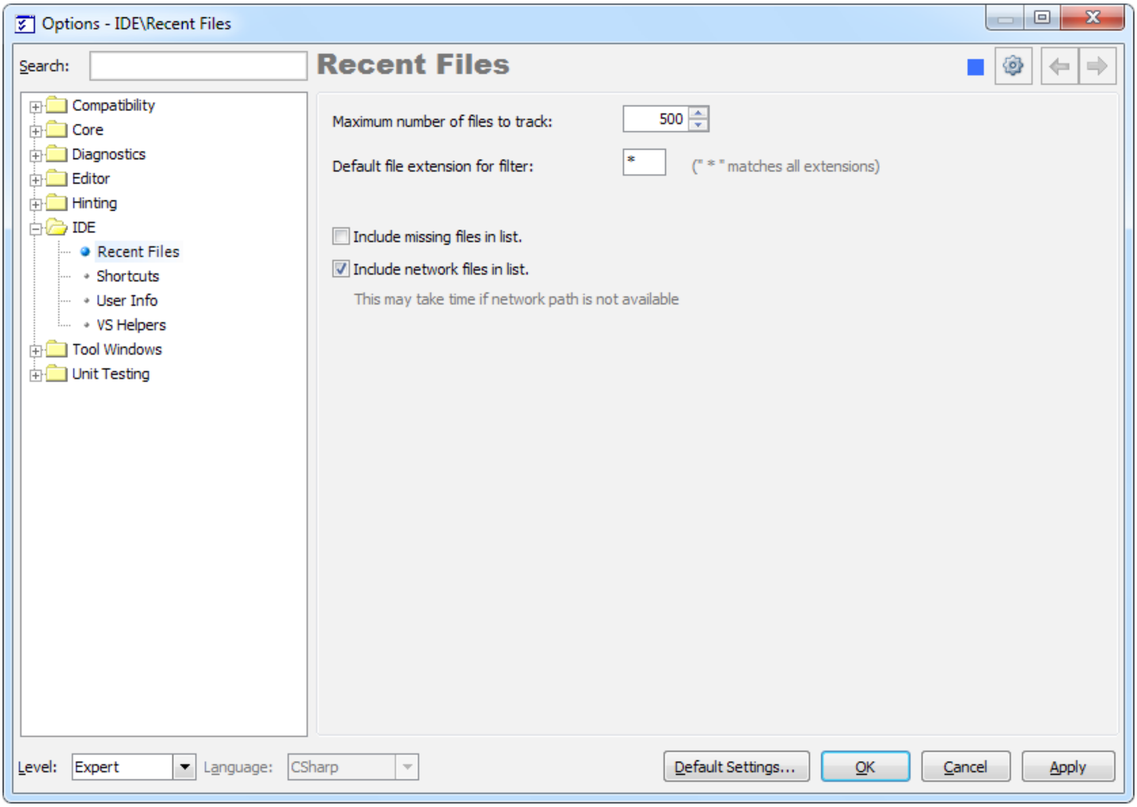
☑ Modified between: 4/25/2011 and 4/26/2011

File Name	Last Access
IFeedback.cs	4/26/2011 12:38:21 AM
AssemblyInfo.cs	4/26/2011 12:36:27 AM
AssemblyInfo.cs	4/26/2011 12:36:14 AM
AssemblyInfo.cs	4/26/2011 12:36:14 AM
AssemblyInfo.cs	4/26/2011 12:36:13 AM

This filter allows you to specify the modification date of files to be shown. There two ways of date filtering:

- Modified in the last X days (or month, or years) – enter the number (X) of days (or month or years) when the file was last modified.
- Modified between – enter the date range when the file was modified

Advanced settings for the **Recent Files** tool window are available on the corresponding options page. You can open it via the context menu choosing the "*Show Options*" menu item:



Maximum number of files to track: 500

Default file extension for filter: * (" * " matches all extensions)

☐ Include missing files in list.

☑ Include network files in list.

This may take time if network path is not available

Level: Expert Language: CSharp

Available options are:

- Maximum number of files to track (500-10,000)
- Default file extension for filter
- Include missing files in the list

This option specifies whether or not show the non-existing files in the list

- Include network files in the list (This may take time if network path is not available)

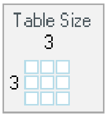
This option specifies whether or not to show the files located on the network

That covers “**Browse Recent Files**“. The similar features for navigation between files are:

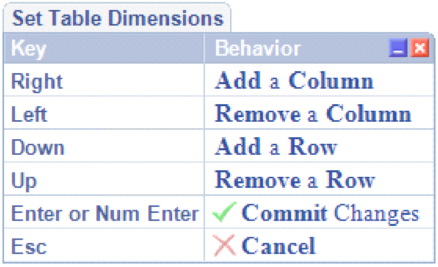
- Quick File Nav
- Open Files

Working with HTML tables

Generating any-size HTML tables and navigating between table cells is very easy when you have DevExpress CodeRush installed. To create a new table just type “.t” inside the HTML markup and press the Space bar. This code snippet template will execute the **Table Size UI** feature of **CodeRush**, and the following tiny window will appear:



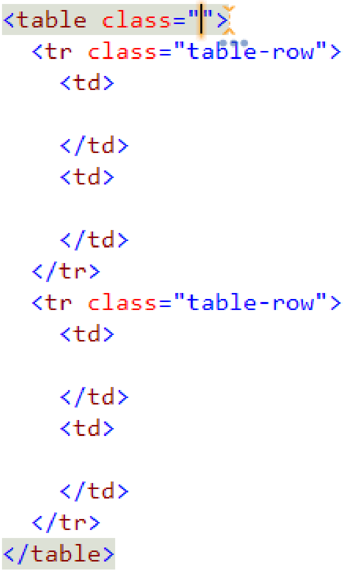
This small window enables you to choose the size of the generated table. Set the columns and rows count by using the arrow keys:



Set Table Dimensions

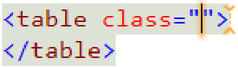
Key	Behavior
Right	Add a Column
Left	Remove a Column
Down	Add a Row
Up	Remove a Row
Enter or Num Enter	✓ Commit Changes
Esc	✗ Cancel

Once the size is set, press the Enter key to generate the table with the specified size, for example, this the 2×2 table created:



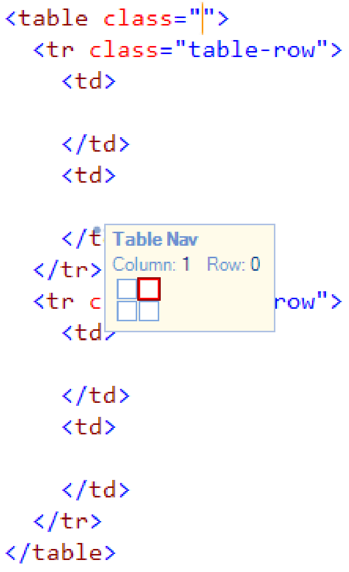
```
<table class="">
  <tr class="table-row">
    <td>

    </td>
    <td>

    </td>
  </tr>
  <tr class="table-row">
    <td>

    </td>
    <td>

    </td>
  </tr>
</table>
```

The Esc key cancels a specific-sized table creation and inserts a default table expansion:



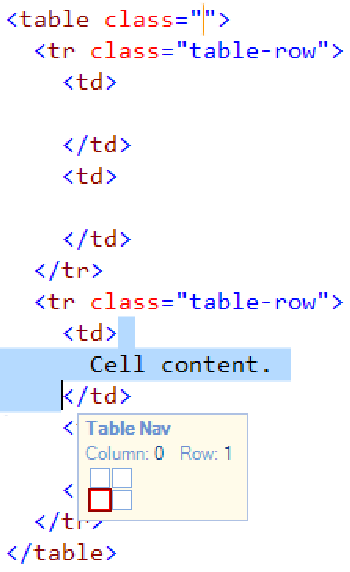
```
<table class="">
</table>
```

Another part of the **Table Size UI** feature is the ability to navigate between table cells using the Tab and Shift+Tab keys. Pressing Tab will move the editor caret to the next cell of the current table and show the ‘map’ of the table – the overall size of the table and the current cell that is highlighted in red:



```
<table class="">
  <tr class="table-row">
    <td>

    </td>
    <td>

    </t
  </tr>
  <tr c                  row">
    <td>

    </td>
    <td>

    </td>
  </tr>
</table>
```

If the cell you are navigating into has some content, it will be completely selected just in case you are going to replace it with new content:



```
<table class="">
  <tr class="table-row">
    <td>

    </td>
    <td>

    </td>
  </tr>
  <tr class="table-row">
    <td>
      Cell content.
    </td>
    <

    <
  </t
</table>
```

Don’t forget the expansion of the **Table Size UI** feature – “**.t**“.

Step Into Member

The **Step Into Member** navigation feature is used in ***Debug mode*** only. It allows you to skip unnecessary members while debugging and drill down into the member you choose.

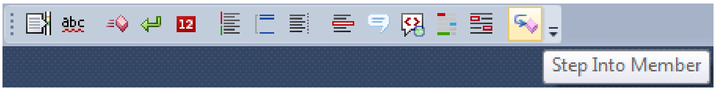
Consider that you are debugging some source code and the current statement has numerous method calls and property references:

IncreaseSalary(***FindPersonById***(***GetPersonId***(*LastName*, *FirstName*, *Age*)), *Salary*)

Here we have three method calls and four property references. The IDE debugger will step into each one. The **Step Into Member** feature allows you to step into the definite member under the editor caret position, skipping all others. So, instead of stepping through every member call, you can choose the member you would like to step into – just move the text caret on the member name you want to drill into, and click the **Step Into Member** button on the DXCore Visualize Toolbar, or press the **Ctrl+Shift+F11** keyboard shortcut.

For example, if you want to step into the *IncreaseSalary* method, skipping the *LastName*, *FirstName*, *Age* and *Salary* property getters, and without stepping into *FindPersonById*, *GetPersonId* methods – move the editor caret at the *IncreaseSalary* method and use **Step Into Member**. After that the program execution will jump to the first statement inside the *IncreaseSalary* method you specified.

This is what the **Step Into Member** icon looks like on the DXCore Visualize Toolbar:



Navigating in the abstract source code tree

When CodeRush/DXCore has parsed the source code, an abstract source code tree is built. You can see the structure of the code tree using the Expression Lab diagnostic tool window. When you understand the source tree, you can use several small tools to quickly navigate inside children and siblings of the tree.

Here are available actions that allow you to quickly jump to different logical blocks inside the source tree. All actions move the editor caret to the desired tree node:

Action	Short description	Shortcut
NavPreviousSibling	Move to previous sibling	Ctrl + Up Arrow
NavNextSibling	Move to next sibling	Ctrl + Down Arrow
NavParent	Move to parent	Ctrl + Alt + Up Arrow
NavFirstChild	Move to first child	Ctrl + Alt + Down Arrow
NavLastChild	Move to last child	{Underfined}

Note that some of the shortcuts are disabled by default (e.g., NavPreviousSibling and NavNextSibling).

Having these tools in your arsenal may boost your navigation performance inside the code editor a lot. For instance, you can press Ctrl+Up and Ctrl+Down to jump between previous and next members of a class.

Navigation between expressions of the parsed source tree

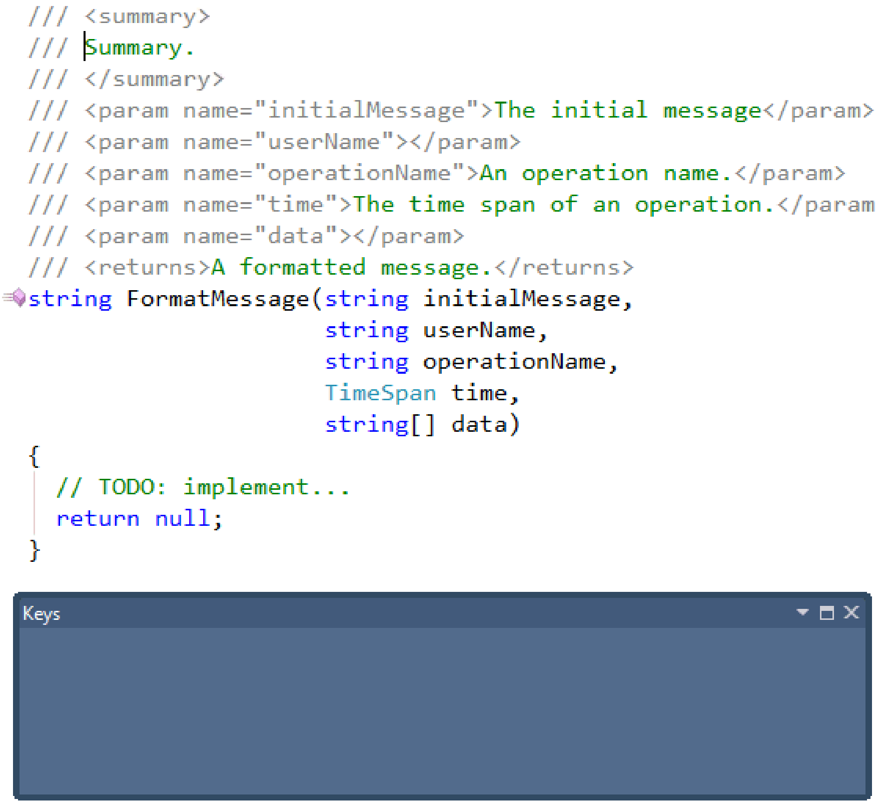
After DXCore has parsed the source code and built an abstract source tree for the active source file, you can navigate between its nodes and detail nodes by using the special navigation feature shipped with CodeRush. The feature is called the **Expression Focus**. It allows you to navigate between the smallest parts of the source tree – expressions.

Expressions are strings, numbers, identifier references, binary operations, method calls and others. You can take an overview of expressions for the current source tree, using the Expression Lab **DXCore** plug-in. The feature may be useful for plug-in developers as well as users, reviewing the code and editing some its parts, such as an XML Doc comments.

You can move in both directions between expressions: forward and backward. There are corresponding shortcuts to move in these directions:

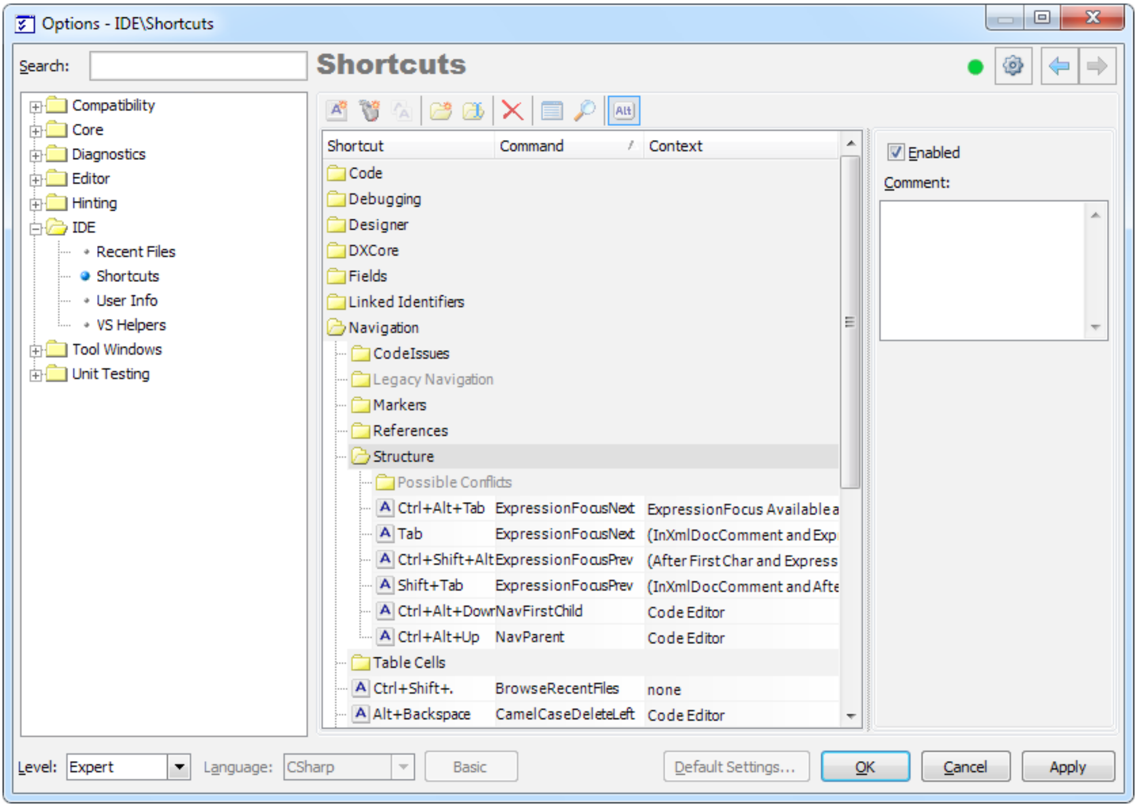
- Ctrl+Alt+Tab and Tab to move to the next expression
- Ctrl+Shift+Alt+Tab and Shift+Tab to move to the previous expression

The feature was widely available via the simple shortcuts like Tab (to move forward) and Shift+Tab (to move backward). However, when the Tab to Next Reference feature appeared, which is bound to the same shortcuts (Tab and Shift+Tab), then the **Expression Focus** become less important, and the availability of its shortcuts has been reduced by changing its context. Currently, the feature is mostly available inside XML Doc comments. So, it allows you to easily edit XML API documentation like this (*the special Key Watcher tool window plug-in shows my key presses*):



```
/// <summary>
/// Summary.
/// </summary>
/// <param name="initialMessage">The initial message</param>
/// <param name="userName"></param>
/// <param name="operationName">An operation name.</param>
/// <param name="time">The time span of an operation.</param>
/// <param name="data"></param>
/// <returns>A formatted message.</returns>
string FormatMessage(string initialMessage,
                     string userName,
                     string operationName,
                     TimeSpan time,
                     string[] data)
{
  // TODO: implement...
  return null;
}
```

To change the default key bindings, go to the Shortcuts options page and tweak them inside the **Navigation | Structure** folder:



Shortcut	Command	Context
Code		
Debugging		
Designer		
DXCore		
Fields		
Linked Identifiers		
Navigation		
CodeIssues		
Legacy Navigation		
Markers		
References		
Structure		
Possible Conflicts		
Ctrl+Alt+Tab	ExpressionFocusNext	ExpressionFocus Available a
Tab	ExpressionFocusNext	(InXmlDocComment and Exp
Ctrl+Shift+Alt	ExpressionFocusPrev	(After First Char and Express
Shift+Tab	ExpressionFocusPrev	(InXmlDocComment and Afte
Ctrl+Alt+Dowr	NavFirstChild	Code Editor
Ctrl+Alt+Up	NavParent	Code Editor
Table Cells		
Ctrl+Shift+.	BrowseRecentFiles	none
Alt+Backspace	CamelCaseDeleteLeft	Code Editor

☑ Enabled

Level: Expert

Language: CSharp

Or you can create your own shortcuts and assign them to the *ExpressionFocusNext* and *ExpressionFocusPrev* actions.

Chapter 9. Unit testing service

Overview

The **Unit Testing Technology** shipped in CodeRush Pro allows you to manage, navigate, run, and debug unit test cases of different unit testing frameworks. The technology consists of several parts:

- Different unit testing frameworks support
- Code editor UI and test runner tool window
- Shortcuts and code templates for creating, running and debugging tests
- Programmatic extensibility and support for managing testing frameworks

Frameworks

Full native support for all popular unit testing frameworks is included out of the box. The supported testing frameworks are:

- VSTest – a Visual Studio integrated unit testing framework
- NUnit – a unit-testing framework for all .Net languages
- MBUnit – a generative unit test framework
- xUnit – a unit testing tool for the .NET Framework
- MSpec – a Context/Specification framework geared towards removing language noise and simpliifying tests
- MS Silverlight – a simple, extensible unit testing framework for Silverlight developers

UI

If the solution contains test cases and the appropriate testing framework assemblies are referenced, the **Unit Testing Service** recognizes them and marks with a test icon inside the code editor that allows you to choose test actions by clicking one. The other way to run tests is to right-click an item inside the Solution Explorer. This will run all test cases inside the scope depending on the item clicked: inside a source file, inside a project, or inside an entire solution. The Visual Studio Output window will have information about operations with test cases. The Unit Test Runner tool window collects all test cases of an entire solution into a single window.

Shortcuts

Several keyboard shortcuts dedicated for unit testing, enable you to manage test cases with a single keystroke inside the code editor. This will run or debug test cases inside the scope depending on the current context, or repeat the last performed tests.

CodeRush code templates for different testing frameworks allow you to generate test fixtures, test methods, assertions, testing attributes with a few keystrokes. See the NUnit testing code templates, for example.

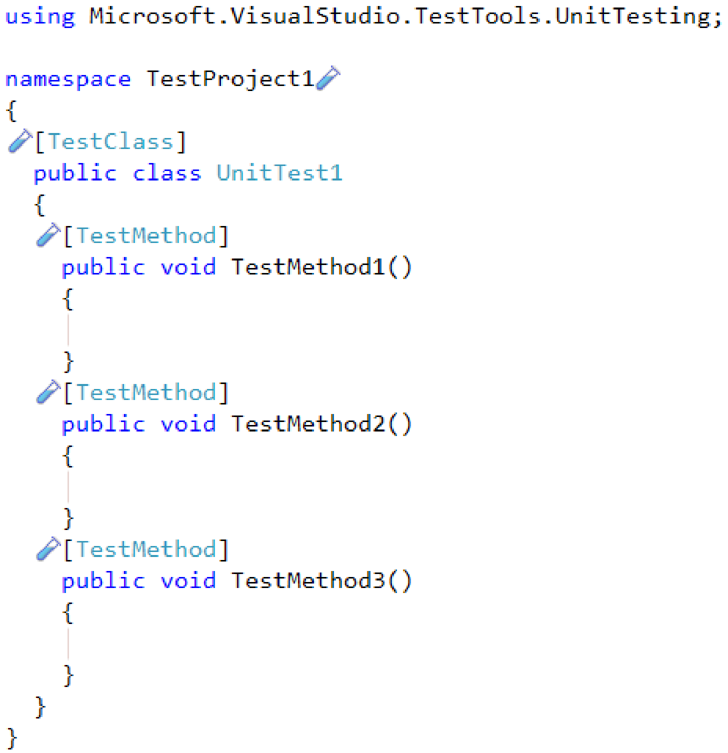
Extensibility

Programmatic extensibility allows you to add support for additional testing frameworks that do not exist in the current implementation. Also you can programmatically manipulate any test cases from the DXCore plug-in.

Here’s a link to the video made by one of the **CodeRush** inspired users (TheThoughtfulCoder.com) about the **Unit Test Runner**.

Unit testing inside the code editor and the Solution Explorer

Let’s compare the benefit of the CodeRush Unit Testing Service against the native Visual Studio unit testing support as an example. The first things that may catch your attention are test icons near test methods, test fixtures and namespaces containing test cases:

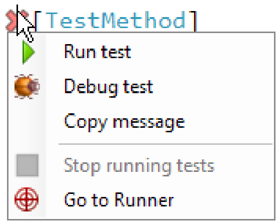


```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProject1
{
  [TestClass]
  public class UnitTest1
  {
    [TestMethod]
    public void TestMethod1()
    {

    }
    [TestMethod]
    public void TestMethod2()
    {

    }
    [TestMethod]
    public void TestMethod3()
    {

    }
  }
}
```

You can see different test icons depending on the status of a test:

🧪	Test(s) has not run yet
✔	Test(s) has run and passed
✓	Test(s) has run and passed, however there have been changes to the code since the last run
✖	Test(s) has run and failed
✗	Test(s) has run and failed, however there have been changes to the code since the last run
▲	Test(s) has been ignored

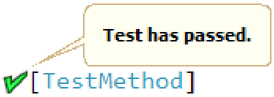
By clicking on this icon, a context menu appears giving you several options:



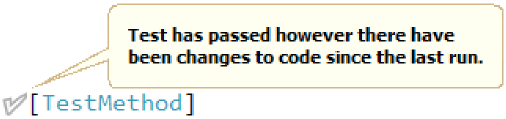
This lets you run or debug a single test in all supported frameworks, including Silverlight. Note that Visual Studio's built-in test runner can't do this.

Test fixtures and namespaces containing unit tests have icons summarizing the run results of their contents. For example, if at least one test has failed, the namespace and the test fixture will both have a red "**X**" icon, indicating they contain failed test cases.

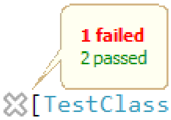
If you hover the mouse over one of the test icons, you can see the status of the test in a summary hint:


Test has passed.

```
[TestMethod]
```

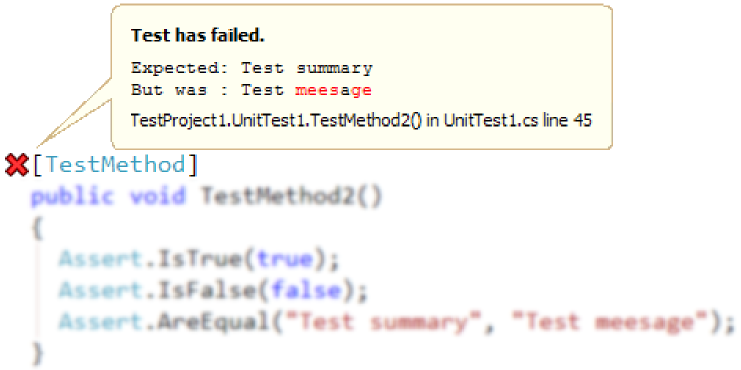

Test has passed however there have been changes to code since the last run.

```
[TestMethod]
```


1 failed
2 passed

```
[TestClass]
```

The failed state of a test will show additional testing results such as message, expected results and a call stack:


Test has failed.

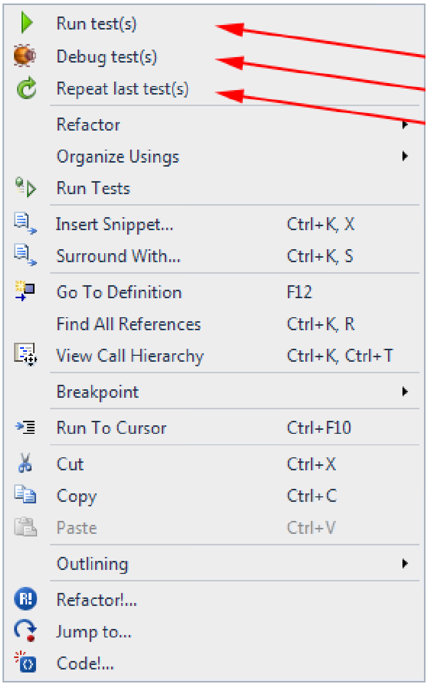
```
Expected: Test summary
But was : Test meesage
```

TestProject1.UnitTest1.TestMethod2() in UnitTest1.cs line 45

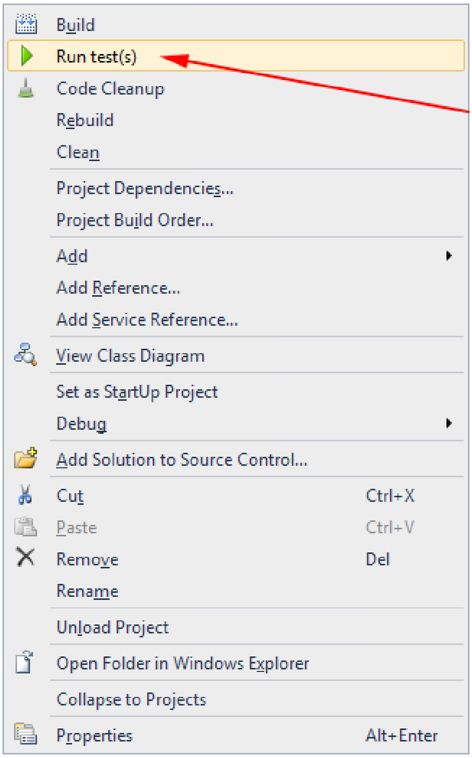
```
[TestMethod]
public void TestMethod2()
{
   Assert.IsTrue(true);
   Assert.IsFalse(false);
   Assert.AreEqual("Test summary", "Test meesage");
}
```

Instead of seeing results in the Visual Studio's Test Results window, you see them straight in the code. Such visualization provides a better overview of a test state, and you can apply changes to a test immediately if required.

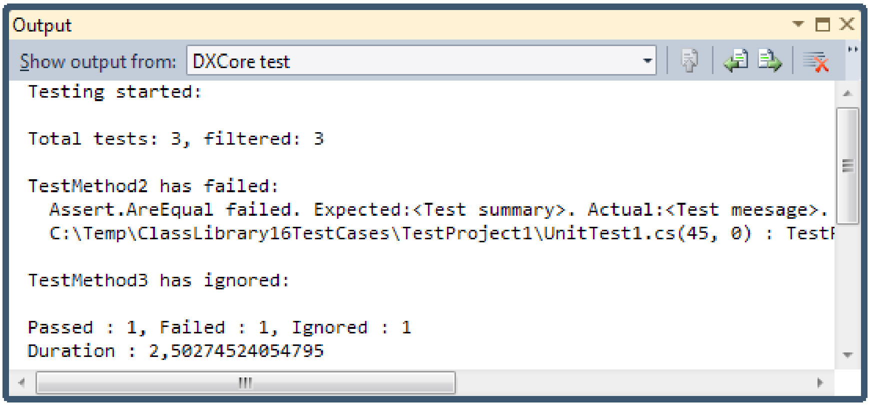
Tests can also be run or debug test cases from the code editor context menu:



And from the Solution Explorer:

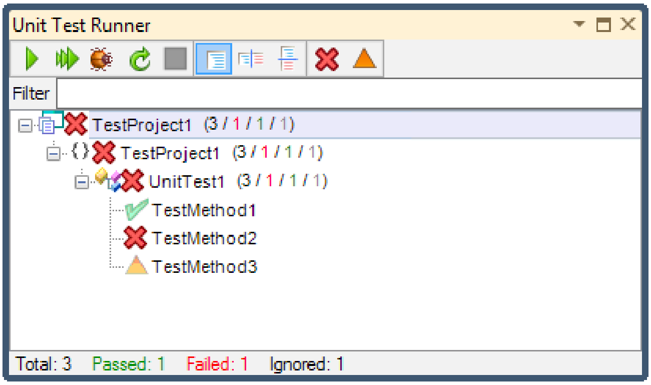


The Visual Studio Output window will have all details about test runs performed:



```
Testing started:

Total tests: 3, filtered: 3

TestMethod2 has failed:
  Assert.AreEqual failed. Expected:<Test summary>. Actual:<Test meesage>.
  C:\Temp\ClassLibrary16TestCases\TestProject1\UnitTest1.cs(45, 0) : TestF

TestMethod3 has ignored:

Passed : 1, Failed : 1, Ignored : 1
Duration : 2,50274524054795
```

The summary of all test cases inside entire solution can be seen in the special **Unit Test Runner** window:



```
TestProject1 (3 / 1 / 1 / 1)
  TestProject1 (3 / 1 / 1 / 1)
    UnitTest1 (3 / 1 / 1 / 1)
      TestMethod1
      TestMethod2
      TestMethod3
Total: 3   Passed: 1   Failed: 1   Ignored: 1
```

Don’t forget that you can use keyboard shortcuts without interacting with the code editor.

Machine.Specifications (mspec) testing framework support

The CodeRush Unit Test Service supports the running and debugging of testing scenarios of the Mashine.Specificat-tions (mspec) framework.
To run or debug mspec tests, do the following steps:

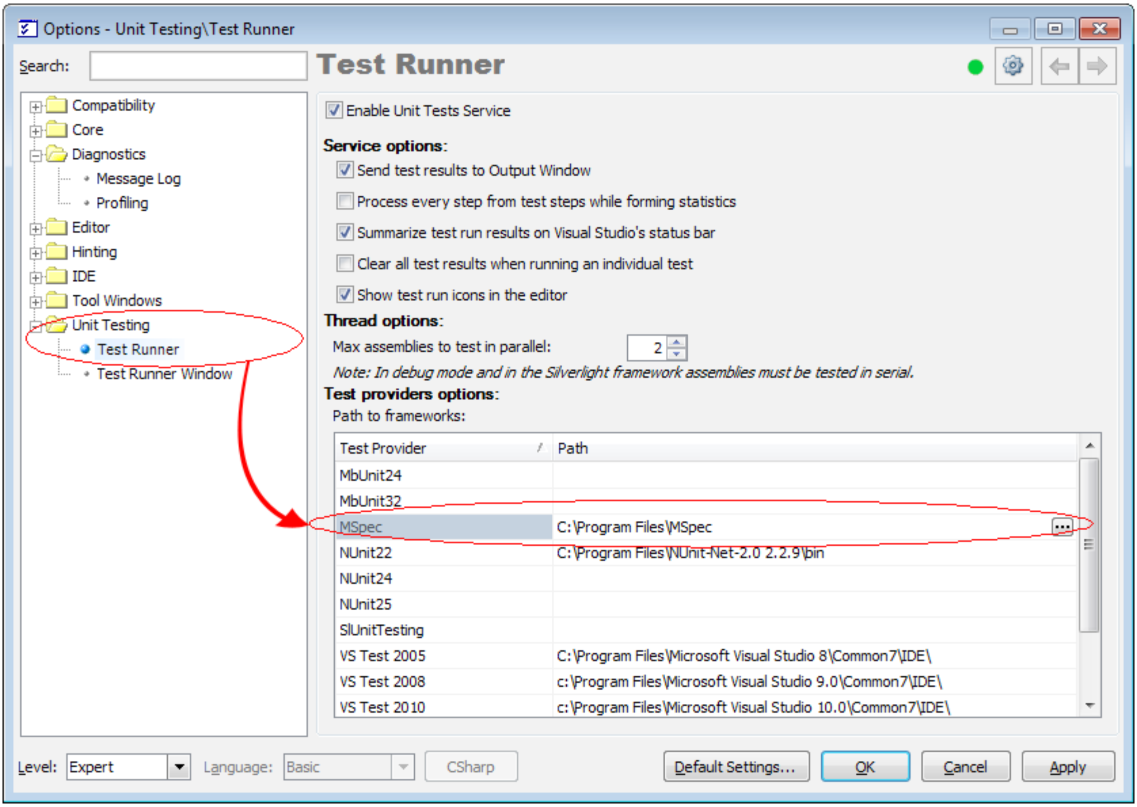
- install the latest version of the MSpec framework;

- specify the path to the framework inside **Test Runner** options page, if it wasn’t detected automatically;

- add necessary framework assemblies to the project, such as Machine.Specifications.dll.

That’s it.

The path to the installed framework should be detected automatically. If by some reason it wasn’t, please go to the Options Dialog and manually specify it:



Test Runner

☑ Enable Unit Tests Service

Service options:

☑ Send test results to Output Window

☐ Process every step from test steps while forming statistics

☑ Summarize test run results on Visual Studio's status bar

☐ Clear all test results when running an individual test

☑ Show test run icons in the editor

Thread options:

Max assemblies to test in parallel: 2

Note: In debug mode and in the Silverlight framework assemblies must be tested in serial.

Test providers options:

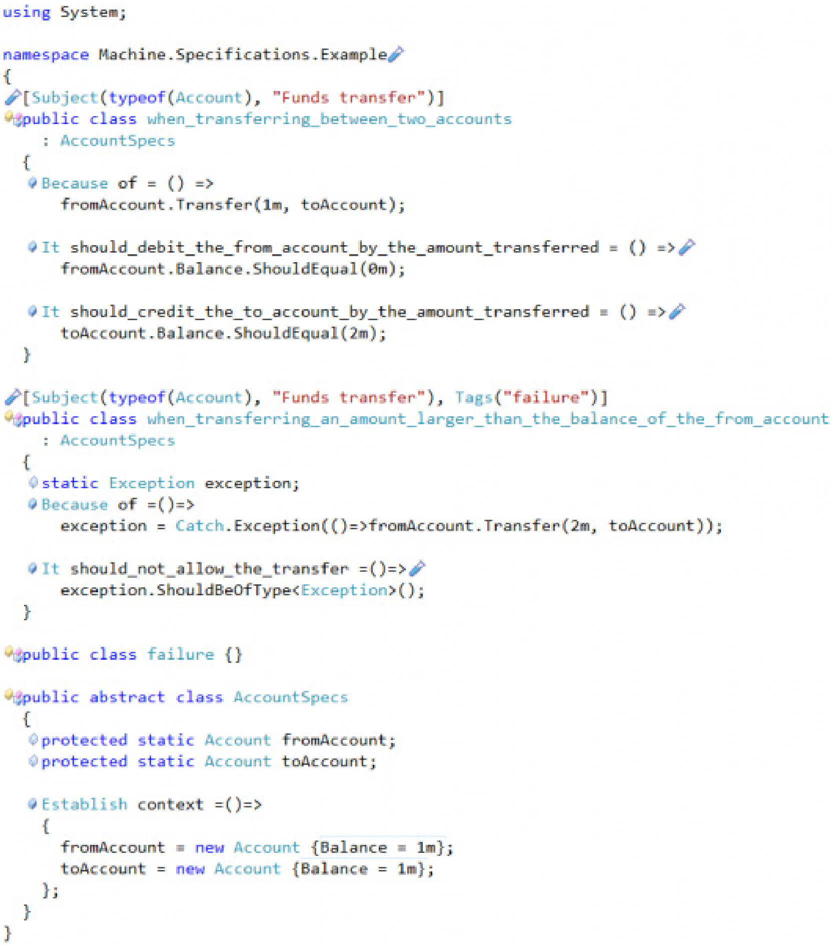
Path to frameworks:

Test Provider	Path
MbUnit24	
MbUnit32	
MSpec	C:\Program Files\MSpec
NUnit22	C:\Program Files\NUnit-Net-2.0 2.2.9\bin
NUnit24	
NUnit25	
SlUnitTesting	
VS Test 2005	C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\
VS Test 2008	c:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\
VS Test 2010	c:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\

As you may know, the most common keywords of the framework are *Subject*, *Establish*, *Because* and *It*: declare the *Subject* of your spec, *Establish* a context of the spec, *Because* something occurs (the action being tested), *It* should do something – an assertion, which is usually the extension methods/fluent interfaces based on *NUnit/xUnit* Assert classes.

There are additional keywords and attributes such as *Behaviours* and *Behaves_like* for more complex scenarios. If you need to perform some cleanup, use the *Cleanup* keyword, which provides teardown for the specification.

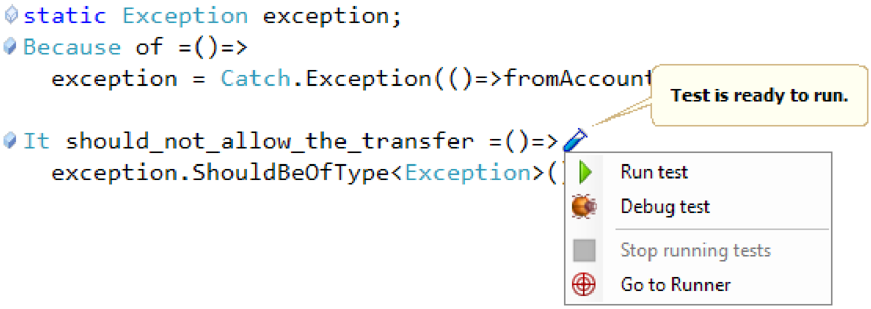
Here’s the classical example shipped with **MSpec**, you can see test icons inside the code editor near each assertion:



```
using System;

namespace Machine.Specifications.Example
{
[Subject(typeof(Account), "Funds transfer")]
public class when_transferring_between_two_accounts
  : AccountSpecs
 {
  Because of = () =>
    fromAccount.Transfer(1m, toAccount);

  It should_debit_the_from_account_by_the_amount_transferred = () =>
    fromAccount.Balance.ShouldEqual(0m);

  It should_credit_the_to_account_by_the_amount_transferred = () =>
    toAccount.Balance.ShouldEqual(2m);
 }

[Subject(typeof(Account), "Funds transfer"), Tags("failure")]
public class when_transferring_an_amount_larger_than_the_balance_of_the_from_account
  : AccountSpecs
 {
  static Exception exception;
  Because of =()=>
    exception = Catch.Exception(()=>fromAccount.Transfer(2m, toAccount));

  It should_not_allow_the_transfer =()=>
    exception.ShouldBeOfType<Exception>();
 }

public class failure {}

public abstract class AccountSpecs
 {
  protected static Account fromAccount;
  protected static Account toAccount;

  Establish context =()=>
   {
    fromAccount = new Account {Balance = 1m};
    toAccount = new Account {Balance = 1m};
   };
 }
}
```

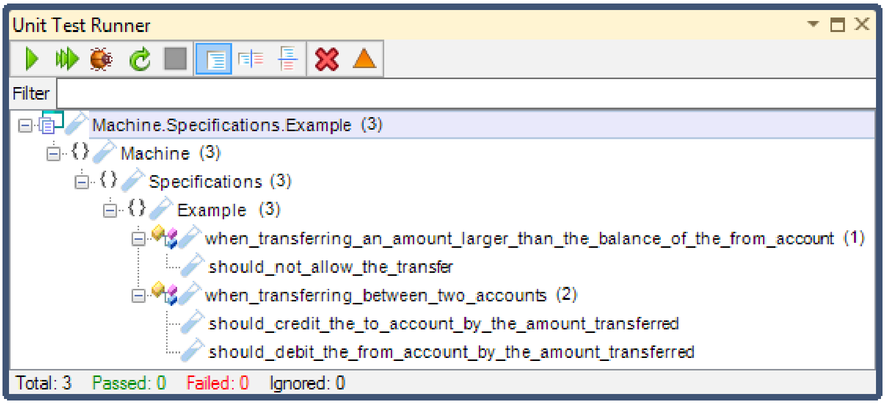
(click the image to enlarge)

Clicking an test icon allows you to run or debug tests:



```
static Exception exception;
Because of =()=>
  exception = Catch.Exception(()=>fromAccount
It should_not_allow_the_transfer =()=>
  exception.ShouldBeOfType<Exception>(
```

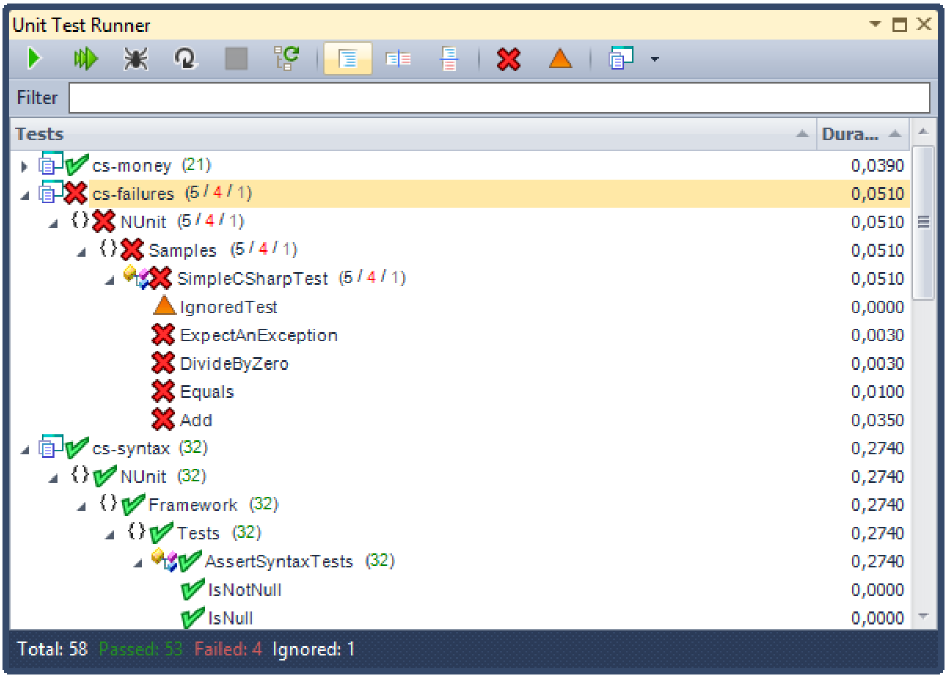
You can also see all tests inside the main **Unit Test Runner** window:



```
Unit Test Runner
Filter
Machine.Specifications.Example (3)
  Machine (3)
    Specifications (3)
      Example (3)
        when_transferring_an_amount_larger_than_the_balance_of_the_from_account (1)
          should_not_allow_the_transfer
        when_transferring_between_two_accounts (2)
          should_credit_the_to_account_by_the_amount_transferred
          should_debit_the_from_account_by_the_amount_transferred
Total: 3   Passed: 0   Failed: 0   Ignored: 0
```

Managing tests with the Unit Test Runner tool window

The **Unit Test Runner** tool window is designed to review, navigate, execute, and manage unit test cases of the entire solution. Here is what it looks like:



```
Unit Test Runner
Filter
Tests                                        Dura...
cs-money (21)                                0,0390
cs-failures (5 / 4 / 1)                      0,0510
  NUnit (5 / 4 / 1)                          0,0510
    Samples (5 / 4 / 1)                      0,0510
      SimpleCSharpTest (5 / 4 / 1)           0,0510
        IgnoredTest                          0,0000
        ExpectAnException                    0,0030
        DivideByZero                         0,0030
        Equals                               0,0100
        Add                                  0,0350
cs-syntax (32)                               0,2740
  NUnit (32)                                 0,2740
    Framework (32)                           0,2740
      Tests (32)                             0,2740
        AssertSyntaxTests (32)               0,2740
          IsNotNull                          0,0000
          IsNull                             0,0000
Total: 58  Passed: 53  Failed: 4  Ignored: 1
```

The window contains the following parts:

- A toolbar to execute test operations and change window properties;
- A filter box to find required test cases;
- A solution tests tree;
- Details of tests run;

- A status bar with a quick status info.

Toolbar

The toolbar has the following options:

Icon	Button	Description
▶	Run	Runs a single test or several tests from the selected node in the tests tree.
▶▶	Run all	Runs all test cases in the current solution.
🐞	Debug	Debugs a single test or several tests from the selected node in the tests tree.
↻	Repeat last test(s)	Runs test cases that were previously executed or debugged.
■	Stop	Stops the current test run process, if any.
↻	Refresh tests tree	Performs a tests search operation to find new tests or remove the deleted ones.
☰	Hide test run details	Hides the details tabs of the test runs.
☰	Show test run details on right	Sets the landscape layout for the tests tree and details of a test run.
☰	Show test run details below	Sets the portrait layout for the tests tree and details of a test run.
✖	Show failed only	Filters the tests tree to show only failed tests.
▲	Show ignored only	Filters the tests tree to show only ignored tests.
▣	Grouping	Groups the tests tree by Project, Category, Namespace, or without grouping (flat view).

Filter and Tests Tree

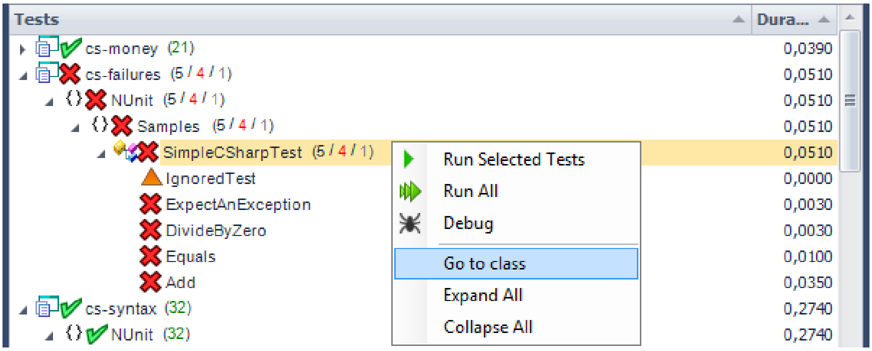
The filter text box allows you to find the tests by name.

The tests tree contains two columns:

- Tests. Shows test cases with a parent project, namespace and class nodes.

- Duration. Shows the duration of each run test run.

The tree has a right-click context menu with the following options:

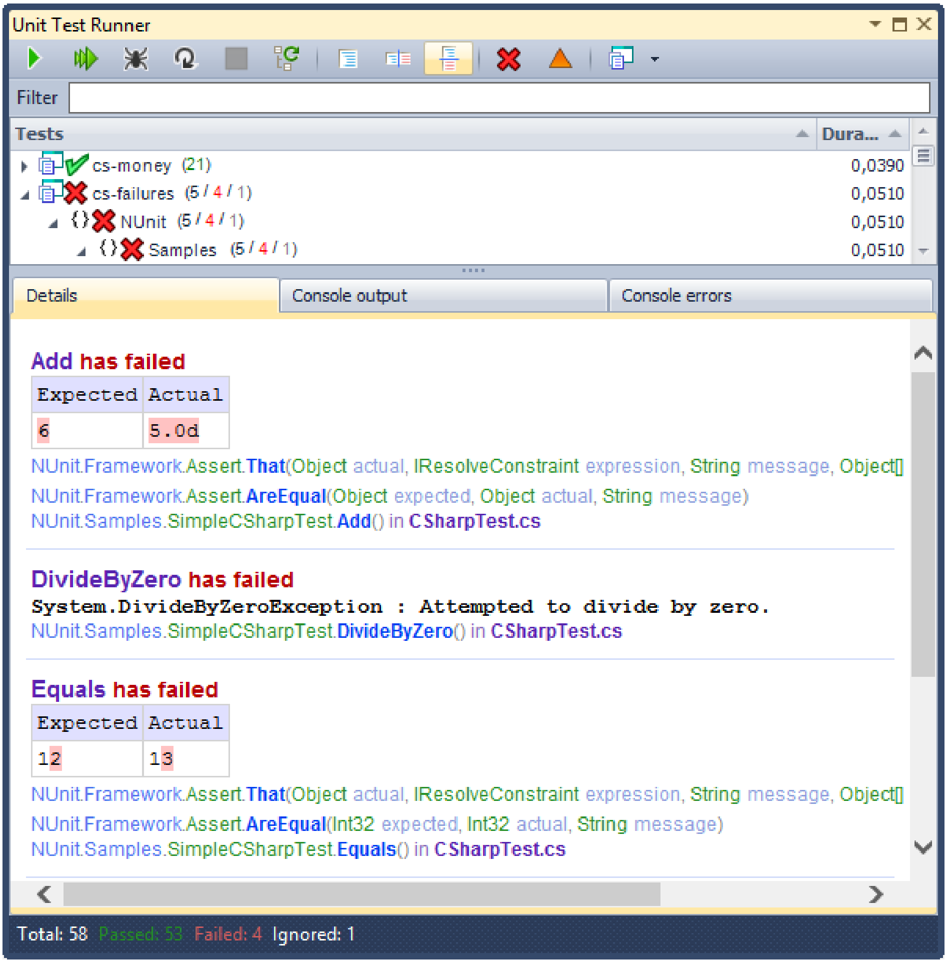


Tests	Dura...
✓ cs-money (21)	0,0390
✖ cs-failures (5 / 4 / 1)	0,0510
{} ✖ NUnit (5 / 4 / 1)	0,0510
{} ✖ Samples (5 / 4 / 1)	0,0510
✖ SimpleCSharpTest (5 / 4 / 1)	0,0510
▲ IgnoredTest	0,0000
✖ ExpectAnException	0,0030
✖ DivideByZero	0,0030
✖ Equals	0,0100
✖ Add	0,0350
✓ cs-syntax (32)	0,2740
{} ✓ NUnit (32)	0,2740

- Go to class (test): navigates to the corresponding item selected in the tests tree.
- Expand All: expands all nodes of the tests tree.
- Collapse All: collapses all nodes of the tests tree.

Test Run Details and Status Bar

The details tab of the test run displays a syntax-highlighted call stacks which makes it easier to see what led to a failure:



```
cs-money (21)                 0,0390
cs-failures (5 / 4 / 1)       0,0510
  NUnit (5 / 4 / 1)           0,0510
    Samples (5 / 4 / 1)       0,0510
```

Add has failed

Expected	Actual
6	5.0d

NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression, String message, Object[]
NUnit.Framework.Assert.AreEqual(Object expected, Object actual, String message)
NUnit.Samples.SimpleCSharpTest.Add() in CSharpTest.cs

DivideByZero has failed
System.DivideByZeroException : Attempted to divide by zero.
NUnit.Samples.SimpleCSharpTest.DivideByZero() in CSharpTest.cs

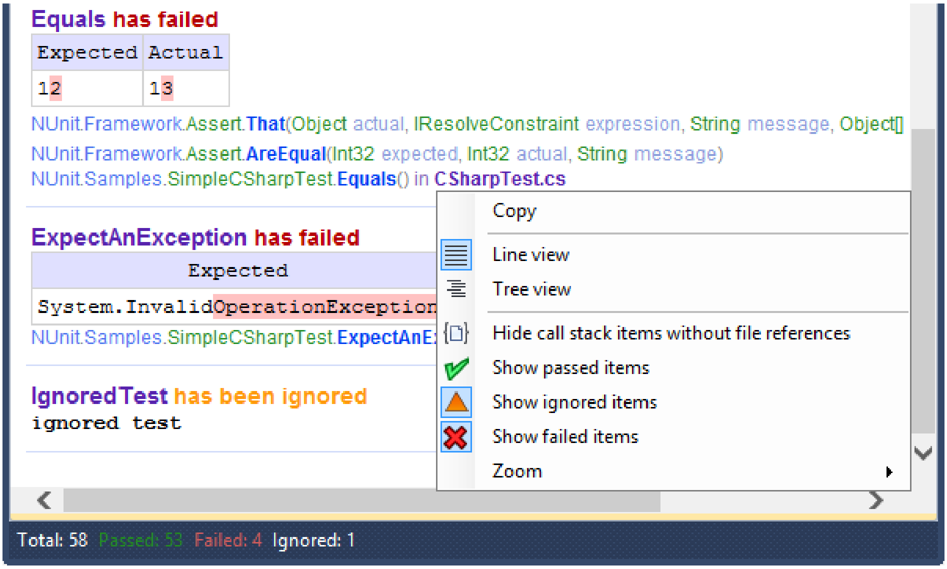
Equals has failed

Expected	Actual
12	13

NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression, String message, Object[]
NUnit.Framework.Assert.AreEqual(Int32 expected, Int32 actual, String message)
NUnit.Samples.SimpleCSharpTest.Equals() in CSharpTest.cs

Total: 58 Passed: 53 Failed: 4 Ignored: 1

You can click the file name to open and navigate to the file with a failed test. All colors in this tab are customizable on the Test Runner Window option page in the Options Dialog. Additional options for this tab are available in the right-click context menu:



Equals has failed

Expected	Actual
12	13

NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression, String message, Object[]
NUnit.Framework.Assert.AreEqual(Int32 expected, Int32 actual, String message)
NUnit.Samples.SimpleCSharpTest.Equals() in CSharpTest.cs

ExpectAnException has failed

Expected
System.InvalidOperationException

NUnit.Samples.SimpleCSharpTest.ExpectAnE

IgnoredTest has been ignored
ignored test

Total: 58 Passed: 53 Failed: 4 Ignored: 1

Available options are:

Icon	Menu item	Description
-	Copy	Copies the information from the Details tab into Clipboard.
≣	Line View	Switches the call stack view to the line view.
≣	Tree View	Switches the call stack view to the tree view.
{□}	Hide call stack items without file references	Toggles the visibility of call stack rows depending on the file names presence to filter the call stack and make it easier to read.
✓	Show passed items	Toggles the visibility of the passed tests.
▲	Show ignored items	Toggles the visibility of the ignored tests.
✖	Show failed items	Toggles the visibility of the failed tests.
-	Zoom ->	Allows you to select the zoom level of the details information.

The Console output and Console Errors tabs shows the standard output and errors of the Console correspondingly, if any.

The status bar shows a summary of total tests, passed, failed, ignored tests.

Don’t forget that you can execute the unit test operations directly from the code editor or via the dedicated unit testting shortcuts.

Shortcuts and actions

CodeRush Unit Test Runner has a set of predefined shortcuts useful for running and debugging unit test cases inside Visual Studio. All shortcuts are easy to remember: they start from a general *Ctrl*+***T*** keystroke (where ***T*** means ***T***esting) followed by a second key which specifies the operation, for example:

- **D** – **D**ebug
- **F** – **F**ile run test cases
- **P** – **P**roject run test cases
- **S** – **S**olution run test cases
- and so on…

Here's the table of **Unit Test Runner** shortcuts and their corresponding actions. Note that some actions don't have default shortcuts set.

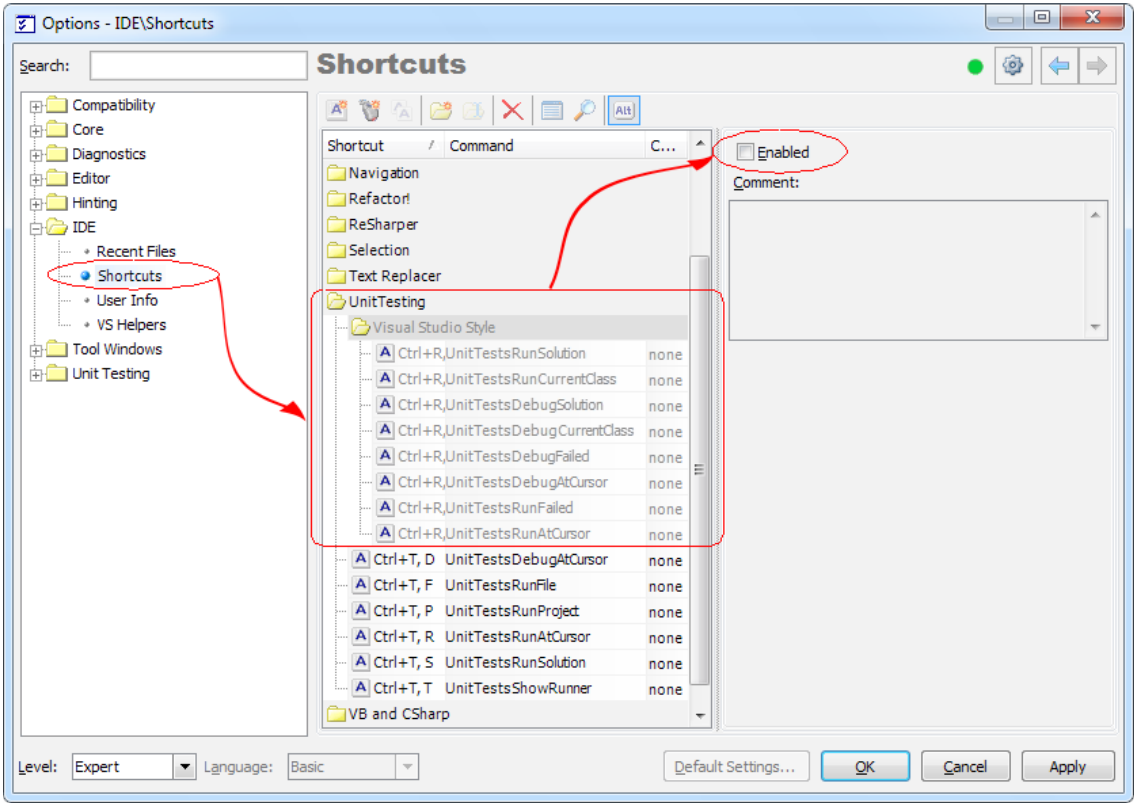
Shortcut	Action	Description
Ctrl+T, D	UnitTestsDebugAtCursor	Starts the debugging of a test case(s) at the editor caret (cursor) location. The scope is determined from the position of the editor caret; for example, inside a method, a fixture, or a namespace it will start debugging appropriate test cases inside of the mentioned scope.
Ctrl+T, F	UnitTestsRunFile	Runs test cases located in the current source file.
Ctrl+T, P	UnitTestsRunProject	Runs test cases located in the current project (selected inside Solution Explorer).
	UnitTestsRunCategory	Runs test cases located in the specified category (passed as a parameter, separated by a semicolon).
Ctrl+T, R	UnitTestsRunAtCursor	Runs test cases at the editor caret (cursor) location. The scope is determined from the position of the editor caret; for example, inside a method, a fixture, or a namespace it will start debugging appropriate test cases inside of the mentioned scope.
Ctrl+T, S	UnitTestsRunSolution	Runs test cases located in the current solution.
Ctrl+T, T	UnitTestsShowRunner	Shows the main Unit Test Runner tool window.
	RepeatLastTests	Runs test cases that were previously executed or debugged.
	TestRun	Runs all test cases at the editor caret (cursor) location. The scope is determined from the position of the editor caret, for example, inside a method, a fixture, or a namespace will start debugging appropriate test cases inside of the mentioned scope.
	TestDebug	Starts the debugging of a test case(s) at the editor caret (cursor) location. The scope is determined from the position of the editor caret; for example, inside a method, a fixture, or a namespace it will start debugging appropriate test cases inside of the mentioned scope.
	RunFolderTests	Runs test cases located in the current folder inside Solution Explorer.
	UnitTestsRunCurrentClass	Runs test cases located in the current source type declaration.
	UnitTestsDebugCurrentClass	Starts debugging test cases located in the current source type declaration.
	UnitTestsDebugCategory	Starts debugging test cases located in the specified category (passed as a parameter, separated by a semicolon).
	UnitTestsRunFailed	Runs test cases that were failing in the previous run.

UnitTestsDebugFailed	Starts debugging test cases that were failing in the previous run.
UnitTestsDebugSolution	Starts debugging test cases located in the current solution.

There is also a second set of shortcuts called “*Visual Studio Style*” that is disabled by default. These shortcuts start with the general *Ctrl*+***R*** keystroke:

Shortcut	Action	Description
Ctrl+R, A	UnitTestsRunSolution	Runs test cases located in the current solution.
Ctrl+R, C	UnitTestsRunCurrentClass	Runs test cases located in the current source type declaration.
Ctrl+R, Ctrl+A	UnitTestsDebugSolution	Starts debugging test cases located in the current solution.
Ctrl+R, Ctrl+C	UnitTestsDebugCurrentClass	Starts debugging test cases located in the current source type declaration.
Ctrl+R, Ctrl+F	UnitTestsDebugFailed	Starts debugging test cases that were failing in the previous run.
Ctrl+R, Ctrl+T	UnitTestsDebugAtCursor	Starts debugging of a test case(s) at the editor caret (cursor) location. The scope is determined from the position of the editor caret, for example, inside a method, a fixture, or a namespace will start debugging appropriate test cases inside of the mentioned scope.
Ctrl+R, F	UnitTestsRunFailed	Runs test cases that were failing in the previous run.
Ctrl+R, T	UnitTestsRunAtCursor	Runs test cases at the editor caret (cursor) location. The scope is determined from the position of the editor caret, for example, inside a method, a fixture, or a namespace will start debugging appropriate test cases inside of the mentioned scope.

To enable these shortcuts, open the Shortcuts options page in the CodeRush Options Dialog and click the *Enabled* check box on the “*Visual Studio Style*” folder inside the *UnitTesting* parent folder (*click on the image to enlarge*):



Shortcut	Command	C...
Navigation		
Refactor!		
ReSharper		
Selection		
Text Replacer		
UnitTesting		
Visual Studio Style		
Ctrl+R,UnitTestsRunSolution		none
Ctrl+R,UnitTestsRunCurrentClass		none
Ctrl+R,UnitTestsDebugSolution		none
Ctrl+R,UnitTestsDebugCurrentClass		none
Ctrl+R,UnitTestsDebugFailed		none
Ctrl+R,UnitTestsDebugAtCursor		none
Ctrl+R,UnitTestsRunFailed		none
Ctrl+R,UnitTestsRunAtCursor		none
Ctrl+T, D	UnitTestsDebugAtCursor	none
Ctrl+T, F	UnitTestsRunFile	none
Ctrl+T, P	UnitTestsRunProject	none
Ctrl+T, R	UnitTestsRunAtCursor	none
Ctrl+T, S	UnitTestsRunSolution	none
Ctrl+T, T	UnitTestsShowRunner	none
VB and CSharp		

☐ Enabled

Comment:

Level: Expert

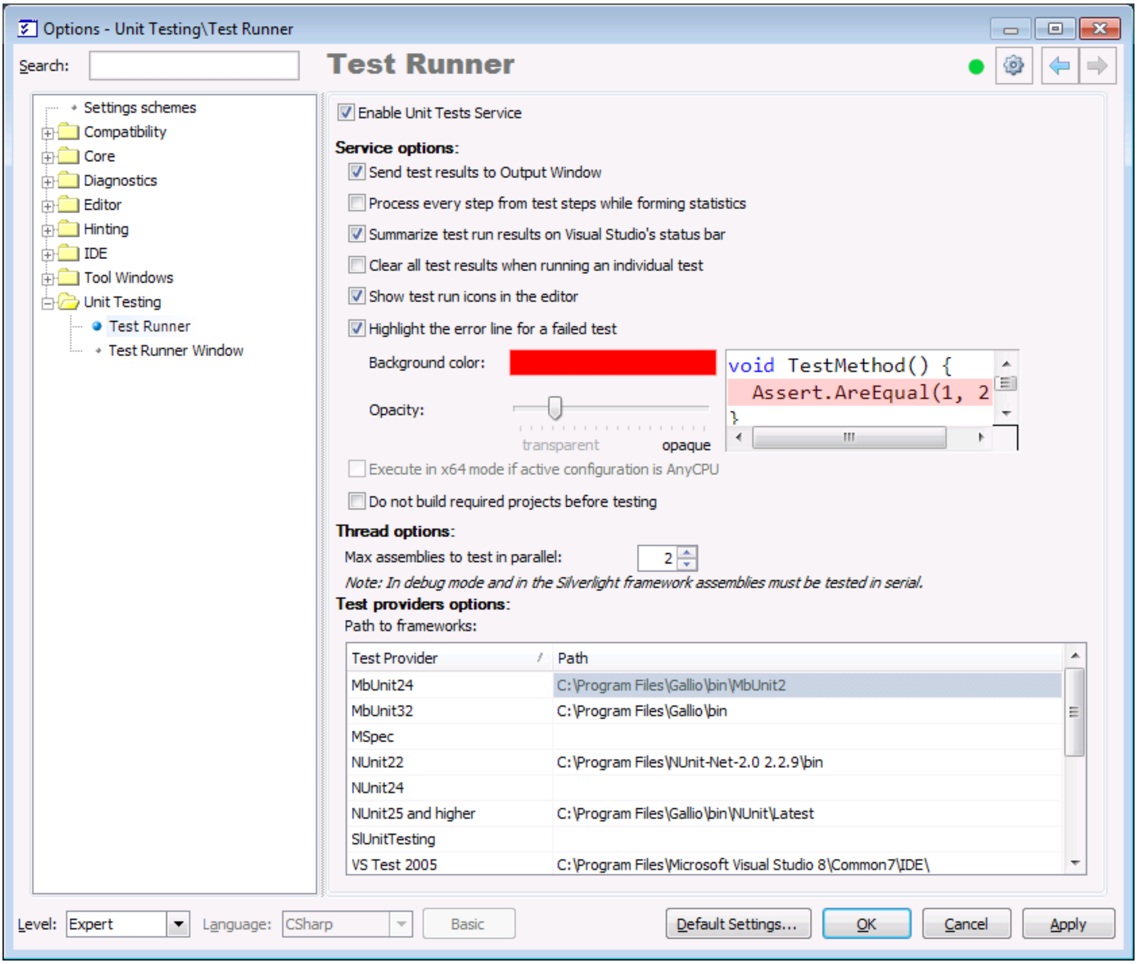
Language: Basic

If you would like to assign a shortcut key for those actions that are missing them, please read the "*How to assign a shortcut key to a particular action*" topic.

Configuration and options

The Unit Test Runner has two dedicated option pages in the Options Dialog. The first one is called **Test Runner** inside the Unit Testing category. It provides you with the capability to tweak the system settings of the Unit Testing Service as well as required paths to the testing frameworks. The second option page named **Test Runner Window** allows you to configure the Test Runner tool window, its behavior and visual appearance preferences. Let's take a closer look at both options pages.

Test Runner



Test Runner

☑ Enable Unit Tests Service

Service options:

☑ Send test results to Output Window

☐ Process every step from test steps while forming statistics

☑ Summarize test run results on Visual Studio's status bar

☐ Clear all test results when running an individual test

☑ Show test run icons in the editor

☑ Highlight the error line for a failed test

Background color: 🟥

Opacity: transparent — opaque

```
void TestMethod() {
  Assert.AreEqual(1, 2
}
```

☐ Execute in x64 mode if active configuration is AnyCPU

☐ Do not build required projects before testing

Thread options:

Max assemblies to test in parallel: 2

Note: In debug mode and in the Silverlight framework assemblies must be tested in serial.

Test providers options:

Path to frameworks:

Test Provider	Path
MbUnit24	C:\Program Files\Gallio\bin\MbUnit2
MbUnit32	C:\Program Files\Gallio\bin
MSpec	
NUnit22	C:\Program Files\NUnit-Net-2.0 2.2.9\bin
NUnit24	
NUnit25 and higher	C:\Program Files\Gallio\bin\NUnit\Latest
SlUnitTesting	
VS Test 2005	C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\

Level: Expert Language: CSharp

- **Enable Unit Test Service**

Specifies whether or not the CodeRush unit testing capability is available inside Visual Studio. If this option is turned off, CodeRush will not search for unit test cases, will not draw testing icons inside the code editor and the capability of running test cases will be unavailable.

Service options

- **Send test results to Output Window**. Specifies whether or not to send the Unit Test Service messages to the Visual Studio Output window.

- **Process every step from test steps while forming statistics**. This is the MBUnit v3.2 specific option. It specifies whether or not to treat the dynamically generated tests as a single test result (option disabled) or multiple test results (option enabled).

- **Summarize test run results on Visual Studio’s status bar**. Specifies whether or not to show the testing results statistics on the IDE status bar.

- **Clear all test results when running an individual test**. Specifies whether or not to clear results of the previous test runs inside the Unit Test Runner tool window before running a single test case.

- **Show test run icons in the editor**. Specifies whether or not to draw clickable icons near testing methods, test fixtures and namespaces containing test cases.

- **Highlight the error line for a failed test**. Specifies whether or not to visually highlight the line of code inside the code editor where the test case failed with the specified background color and opacity.

- **Execute in x64 mode if active configuration is AnyCPU**. If you have an x64 operating system, this option allows you to specify whether or not test cases are executed in x64 mode if the current solution platforms configuration is set to AnyCPU.

- **Do not build required projects before testing**. This option forbids the rebuilding of the project and its dependent projects before test execution. This may increase the startup time of the unit tests execution, but may produce incorrect testing results if you modified the source code before tests execution.

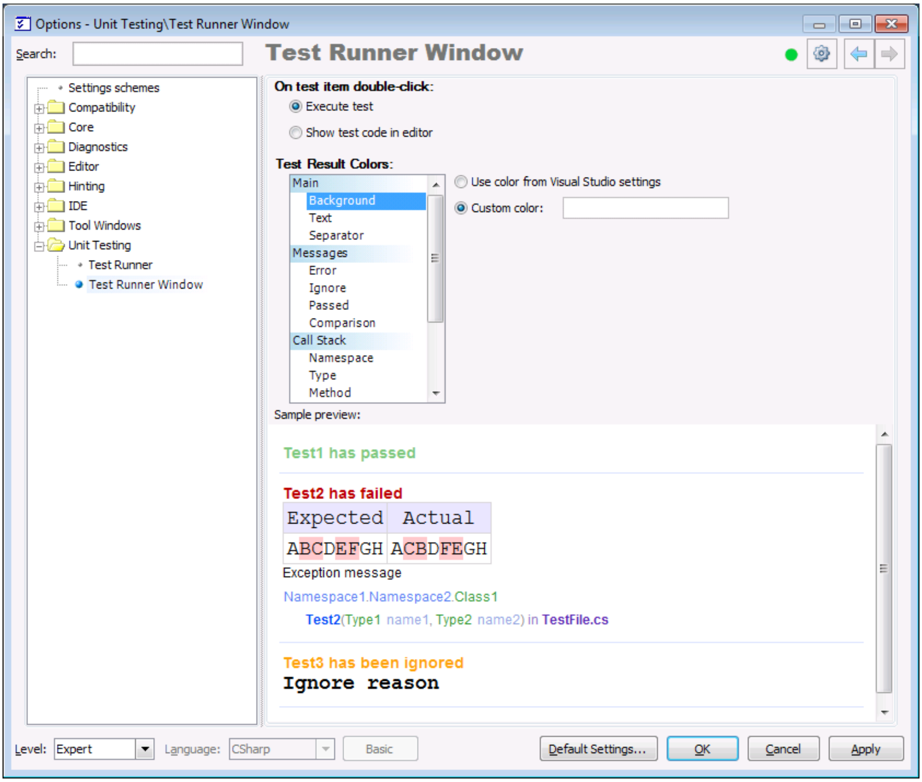
Thread options

- **Max assemblies to test in parallel**. Specifies the number of threads used for unit testing: 1 to 5. This may increase the speed of unit tests execution. Note that this option is not applied to the Silverlight unit testing framework.

Test providers options

Specifies the paths of each of the supported unit testing frameworks such as NUnit, MBUnit, MSpec, VSTest, xUnit, SlUnitTesting, etc. This is required for a correct execution of the test cases via the Unit Test Service.

Test Runner Window



On test item double-click:

● Execute test

○ Show test code in editor

Test Result Colors:

Main
Background
Text
Separator
Messages
Error
Ignore
Passed
Comparison
Call Stack
Namespace
Type
Method

○ Use color from Visual Studio settings

● Custom color:

Sample preview:

Test1 has passed

Test2 has failed

Expected	Actual
ABCDEFGH	ACBDFEGH

Exception message

Namespace1.Namespace2.Class1

Test2(Type1 name1, Type2 name2) in TestFile.cs

Test3 has been ignored

Ignore reason

Level: Expert Language: CSharp Basic

- **On test item double-click**. You can choose what default action should be performed when a test case item is double-clicked inside the tool window: execute a test or navigate to a test case inside the code editor.

Test Result Colors

Lists all available visual elements of the testing results of the tool window such as messages, call stack, delimiters and the main area. The option page allows you to apply the default Visual Studio color or a custom one for any of the listed items. Inside the Sample Preview area, you can see the current visual scheme and see your changes immediately to find the best visual presentation for unit testing results.

Code templates for generating unit testing code

There are many code templates available to quickly create tests, test fixtures and assertion calls for all supported unit testing frameworks. For each test framework, they similar , so if you switched to another framework, you can use the same code templates to generate code which will be valid in a new test framework.

Common code templates are:

Template	Description
tm	Creates a test method.
tc/tf	Creates a test fixture (class).
tsu	Creates a ‘set up’ method.
ttd	Creates a ‘tear down’ method.
tfsu	Creates a ‘class initialize’ method.

Asserts code templates:

Template	Description
ae	Creates an ‘are equal’ assertion call.
an	Creates an ‘is null’ assertion call.
at	Creates an ‘is true’ assertion call.
af	Creates an ‘is false’ assertion call.
ai	Creates an ‘ignore’ assertion call.
etc…	

There are also code templates for generating test attributes and using statements with the required namespace references.

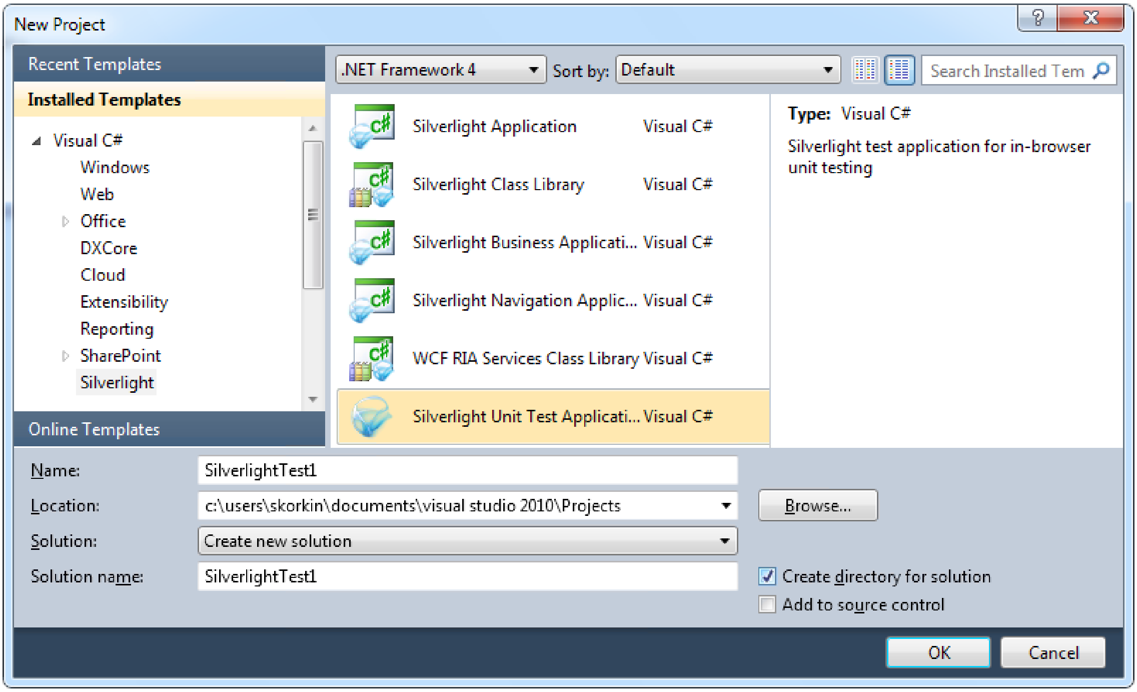
Here are the code templates for various test frameworks:

- Code templates for MbUnit
- Code templates for MS VS testing
- Code templates for NUnit
- Code templates for xUnit
- Code templates for Silverlight

Silverlight testing framework support

The CodeRush Unit Test Runner supports running tests from the **Silverlight Unit Test Framework** which is a part of **Silverlight Toolkit** now, so it's easy to test your Silverlight applications.

Once you install the **Silverlight Unit Test Framework**, you are able to create the *Silverlight Unit Test Application* project. Here is what the *Add Project* dialog looks with the *Silverlight* tab highlighted:



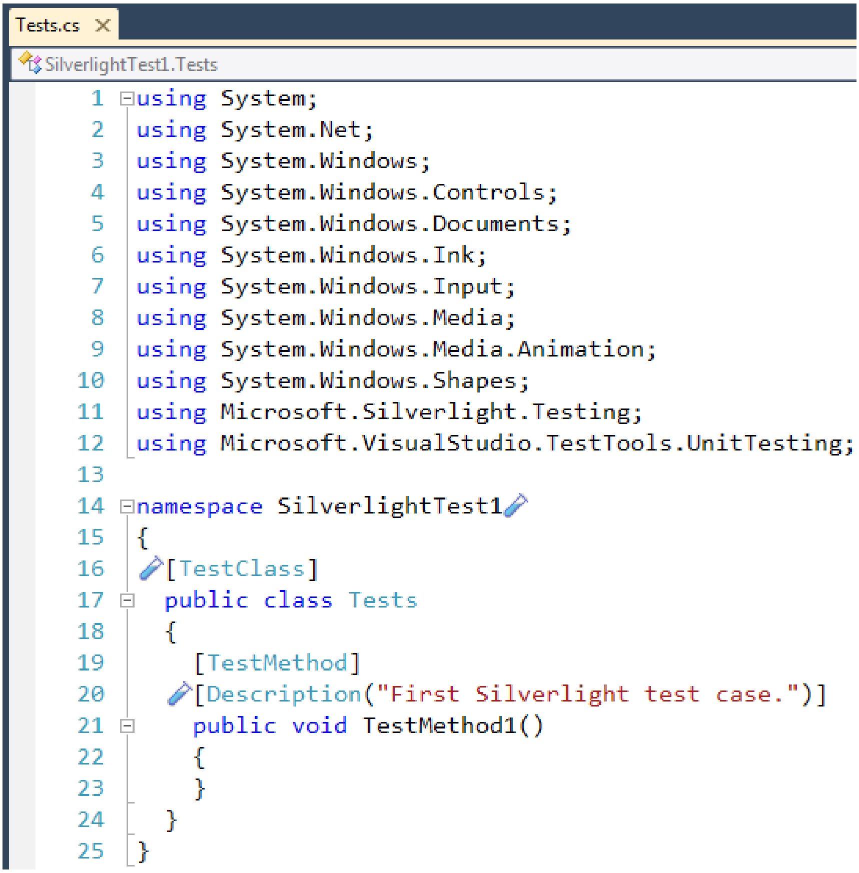
Silverlight Application Visual C#
Silverlight Class Library Visual C#
Silverlight Business Applicati... Visual C#
Silverlight Navigation Applic... Visual C#
WCF RIA Services Class Library Visual C#
Silverlight Unit Test Applicati... Visual C#

Type: Visual C#

Silverlight test application for in-browser unit testing

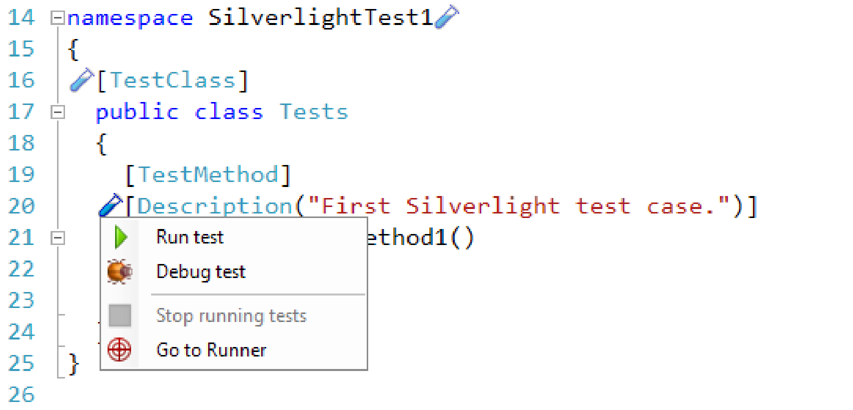
Name: SilverlightTest1
Location: c:\users\skorkin\documents\visual studio 2010\Projects
Solution: Create new solution
Solution name: SilverlightTest1
☑ Create directory for solution
□ Add to source control

If you create this type of a project, you will get the source file containing a single test case. If you ever worked with Microsoft unit testing in Visual Studio, you will find the API quite familiar, including all key classes, attributes and namespaces: the *TestClass* attribute identifies the class that contains unit tests. The*TestMethod* attribute identifies methods that represent a single unit test case. *Description* attribute data will be shown in a runner UI when unit tests are run. You can see the testing icons appeared immediately over the test case:



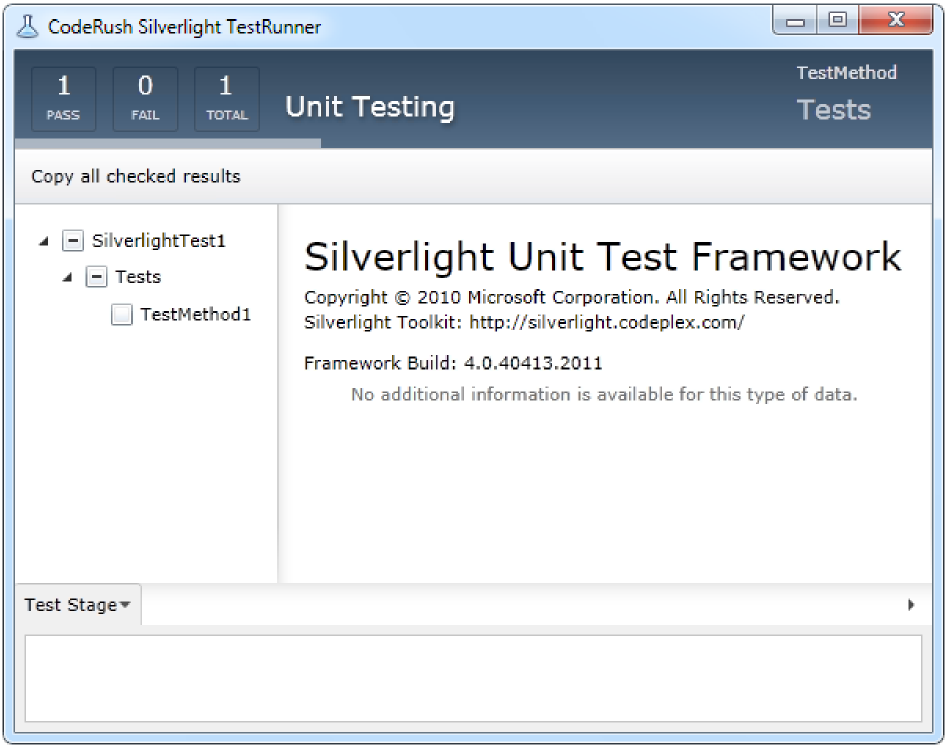
```
using System;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.Silverlight.Testing;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace SilverlightTest1
{
  [TestClass]
  public class Tests
  {
    [TestMethod]
    [Description("First Silverlight test case.")]
    public void TestMethod1()
    {
    }
  }
}
```

The test case is ready to run – you don't have to install any other assemblies or tweak any testing settings. All you need is justto click an icon and choose whether to run or debug the test case:



```
namespace SilverlightTest1
{
  [TestClass]
  public class Tests
  {
    [TestMethod]
    [Description("First Silverlight test case.")]
                                     ethod1()



}
```

When you run a test case, the **CodeRush Silverlight TestRunner** window appears. It uses native support for running Silverlight unit test cases:



Silverlight Unit Test Framework

Copyright © 2010 Microsoft Corporation. All Rights Reserved.
Silverlight Toolkit: http://silverlight.codeplex.com/

Framework Build: 4.0.40413.2011

No additional information is available for this type of data.

After tests are run, the **Silverlight Test Runner** will be automatically closed, and you can see the test run summary in the Visual Studio output window, inside the Unit Test Runner tool window or right in the code editor.

Support for XUnit, NUnit, MBUnit, and Visual Studio testing frameworks

In addition to the MSpec and Silverlight unit testing frameworks, the following test frameworks are supported by the CodeRush Unit Test Runner out of the box:

- MbUnit, versions 2.4 and 3.2;
- NUnit, versions 2.2, 2.4, 2.5 and higher;
- xUnit, versions 1.5, 1.6, 1.7, 1.8 and 1.9;
- Visual Studio built-in framework for Visual Studio 2008, 2010, 2012.

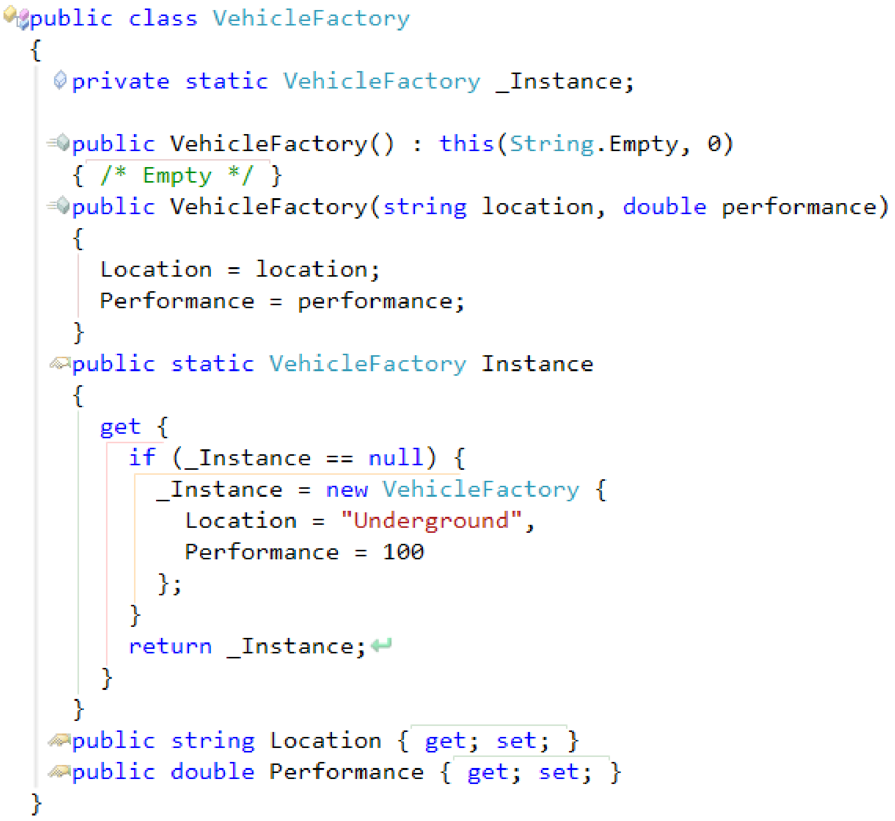
To let the **Unit Test Runner** find and execute test cases from the mentioned testing frameworks, you should specify the correct paths to them on the Test Runner option page if they were not resolved automatically.

Due to the extensible Test Runner architecture, you can easily add a new unit test framework support programmatically.

Chapter 10. Code editor visualization

Structural Highlighting

Structural Highlighting helps you visually navigate the structure of the code. Matching delimiters are connected with low-contrast lines (you can configure the color of the line for each type of code block individually) that are easy to read when this information is important, and easy to ignore when your mind is on the code. For example, see the vertical and horizontal lines in this code sample:



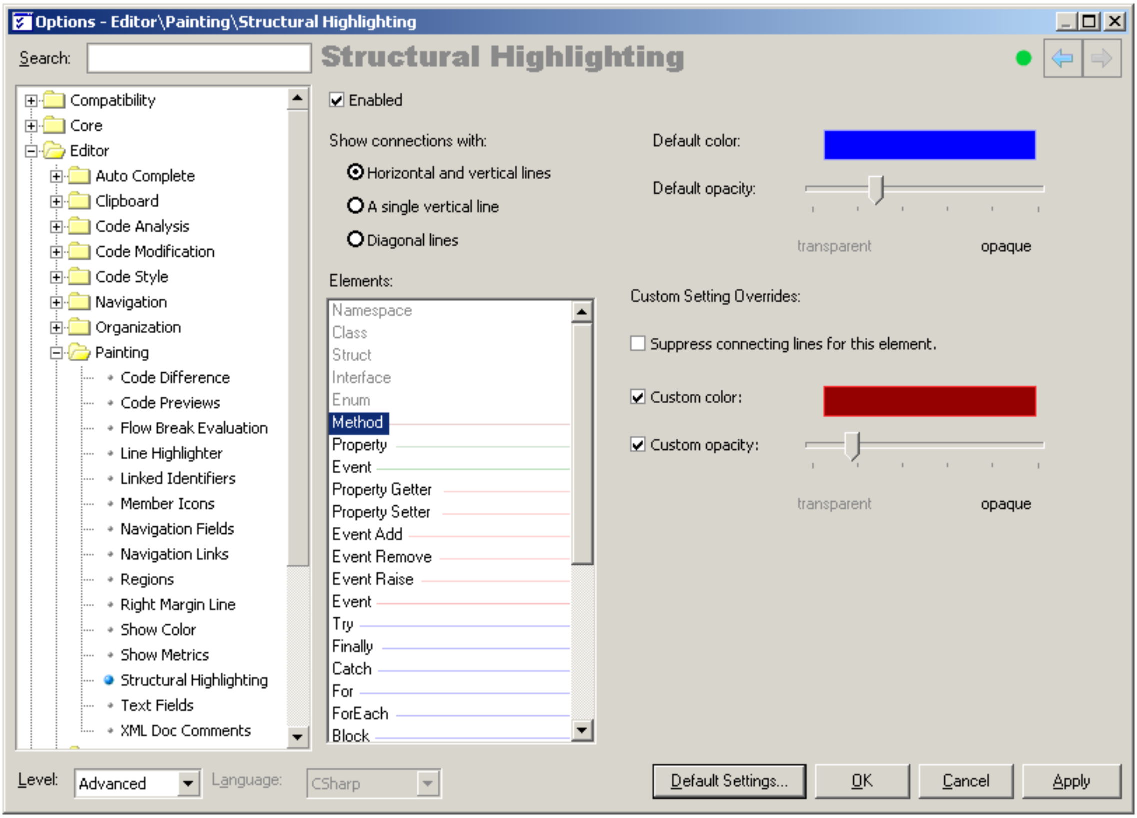
```
public class VehicleFactory
{
  private static VehicleFactory _Instance;

  public VehicleFactory() : this(String.Empty, 0)
  { /* Empty */ }
  public VehicleFactory(string location, double performance)
  {
    Location = location;
    Performance = performance;
  }
  public static VehicleFactory Instance
  {
    get {
      if (_Instance == null) {
        _Instance = new VehicleFactory {
          Location = "Underground",
          Performance = 100
        };
      }
      return _Instance;
    }
  }
  public string Location { get; set; }
  public double Performance { get; set; }
}
```

The feature is useful for understanding the flow of large methods or third-party C# code using a different leading brace position than you might otherwise work with. It is especially useful when a code block takes up more room than can be displayed on the screen at once.

You have complete control over how and when the line is drawn for different code elements, so you can change:

- Line color
- Line opacity
- Line connection style

Unfortunately, the options page (“**Editor | Painting | Structural Highlighting**“) in the Options Dialog is not lanpguage dependent at the moment, but it is planned to be improved in the future.



☑ Enabled

Show connections with:

● Horizontal and vertical lines

○ A single vertical line

○ Diagonal lines

Default color: 🟦

Default opacity:

Custom Setting Overrides:

□ Suppress connecting lines for this element.

☑ Custom color: 🟥

☑ Custom opacity:

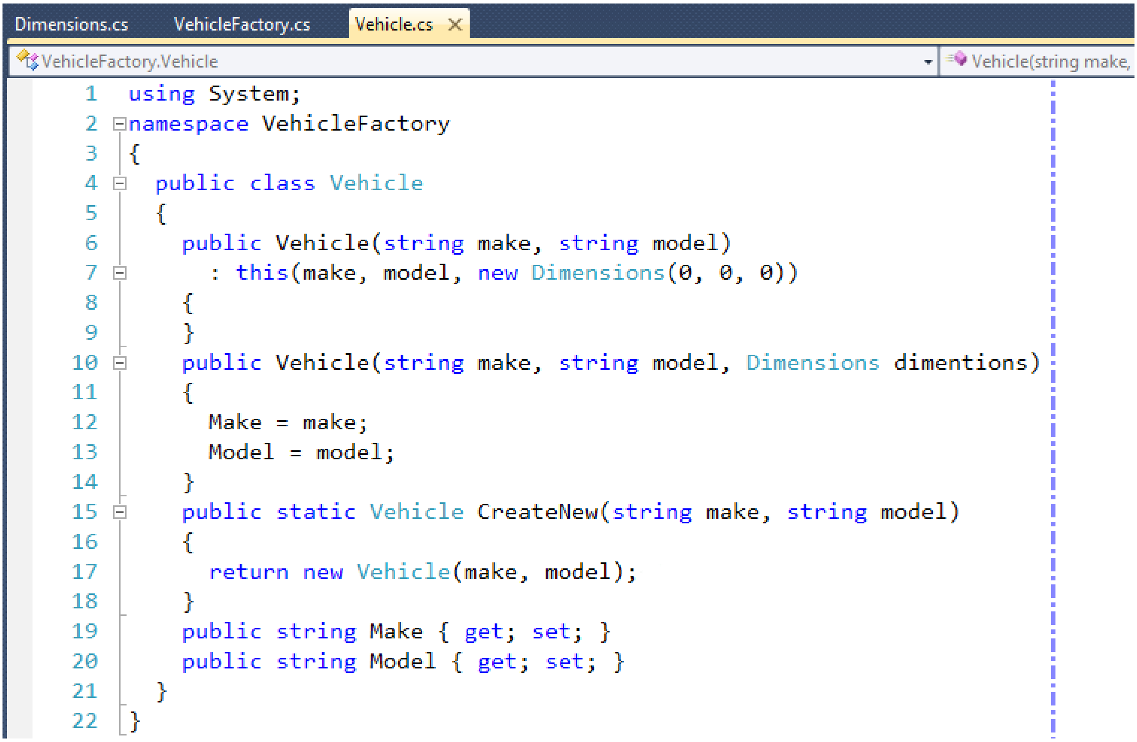
Level: Advanced

Language: CSharp

Even better, this plug-ins ships with the source code, so you might be able to add the functionality you desire.

Right Margin Line

Good code style suggests limiting the length of a code line to 80 characters, and only 70 characters for indented code examples to be used in documentation. The **Right Margin Line** helps you to visually indicate the specified line length. It doesn't prevent typing to the right of it, but lets you to identify and break long expressions over multiple lines.



```
using System;
namespace VehicleFactory
{
  public class Vehicle
  {
    public Vehicle(string make, string model)
      : this(make, model, new Dimensions(0, 0, 0))
    {
    }
    public Vehicle(string make, string model, Dimensions dimentions)
    {
      Make = make;
      Model = model;
    }
    public static Vehicle CreateNew(string make, string model)
    {
      return new Vehicle(make, model);
    }
    public string Make { get; set; }
    public string Model { get; set; }
  }
}
```

On the other side, Visual Studio has built-in margin line support. You can enable it via the registry editor in the following entry at:

[HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\%VS_VERSION%\Text Editor]

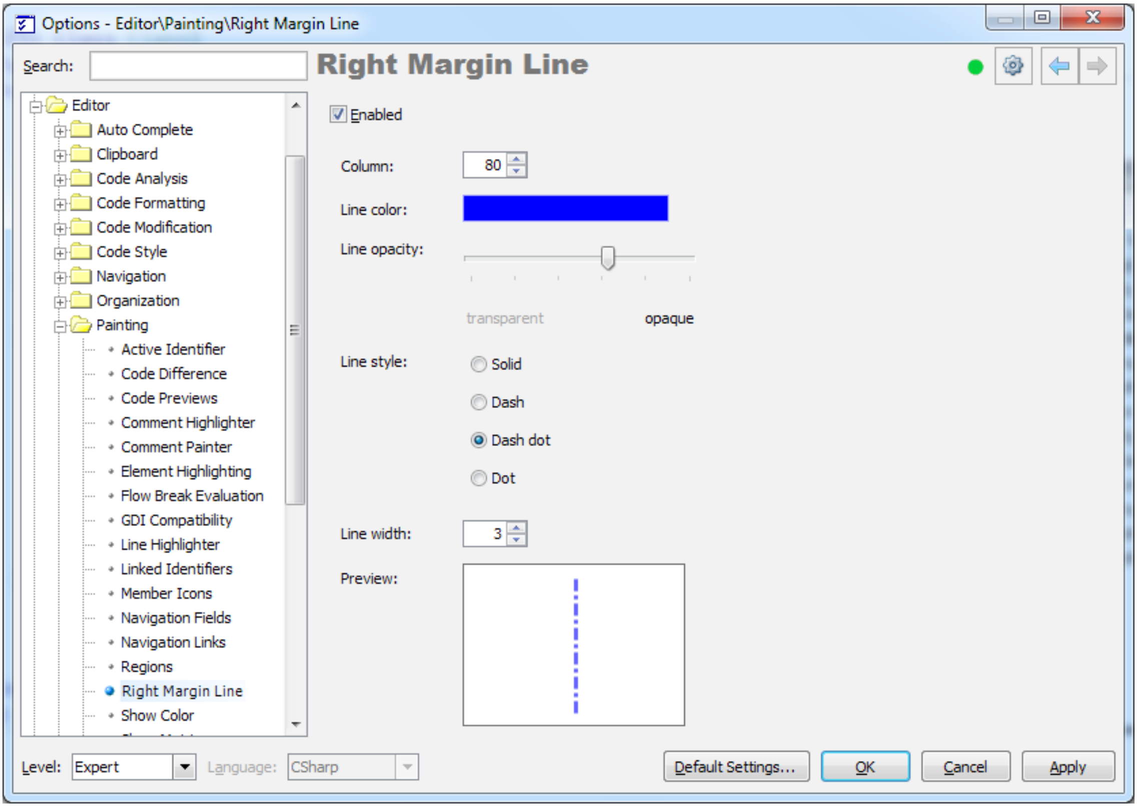
Where %**VS_VERSION**% is a version of the IDE you're using, e.g. **10.0** (VS۲۰۱۰), **9.0** (VS۲۰۰۸), **8.0** (VS۲۰۰۵), **7 .1** (VS۲۰۰۳).

The following string key should be added:

"Guides"="RGB(64,64,128), 80"

The first part specifies the color, while the other one (80) is the column the line will be displayed. This will produce a blue margin line right before the 80th column. It looks pretty well on white and black backgrounds, but you can of course change it easily to any color and position. Note: the registry entry is different for Express editions of Visual Studio.

However, the **Right Margin Line** shipping with CodeRush Pro has a few more options you can specify on the "**Editor** | **Painting** | **Right Margin Line**" options page in the Options Dialog. You can specify not only line color, its opacity and position, but also line style and line width:



Options - Editor\Painting\Right Margin Line

Right Margin Line

☑ Enabled

Column: 80

Line color:

Line opacity: transparent — opaque

Line style:
○ Solid
○ Dash
● Dash dot
○ Dot

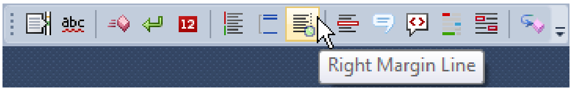
Line width: 3

Preview:

Level: Expert

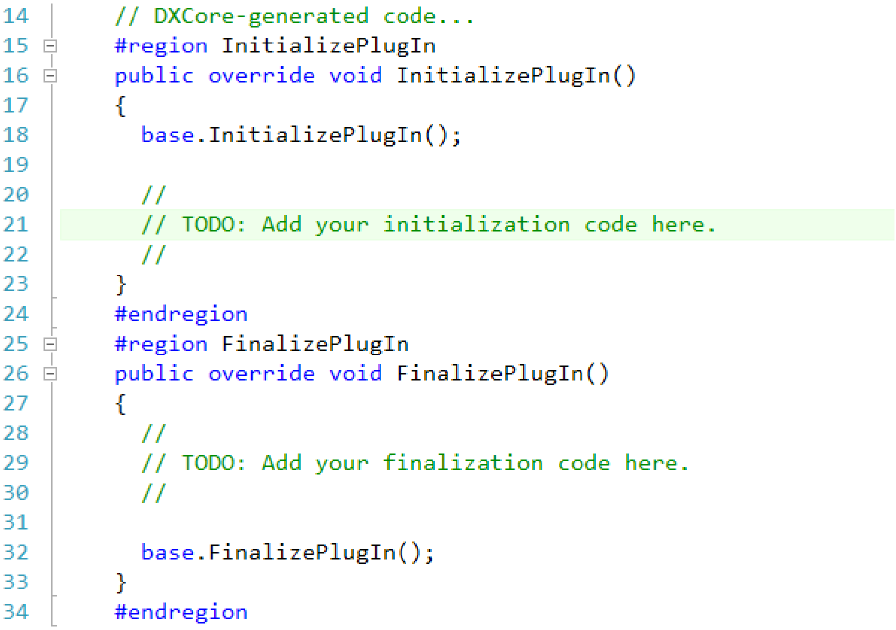
Language: CSharp

The **Right Margin Line** feature is located inside an open source "***CR_RightMarginLine***" sample plug-in in the Shared Source solution of **CodeRush**. The feature is visual, so it is available on the DXCore Visualize toolbar:



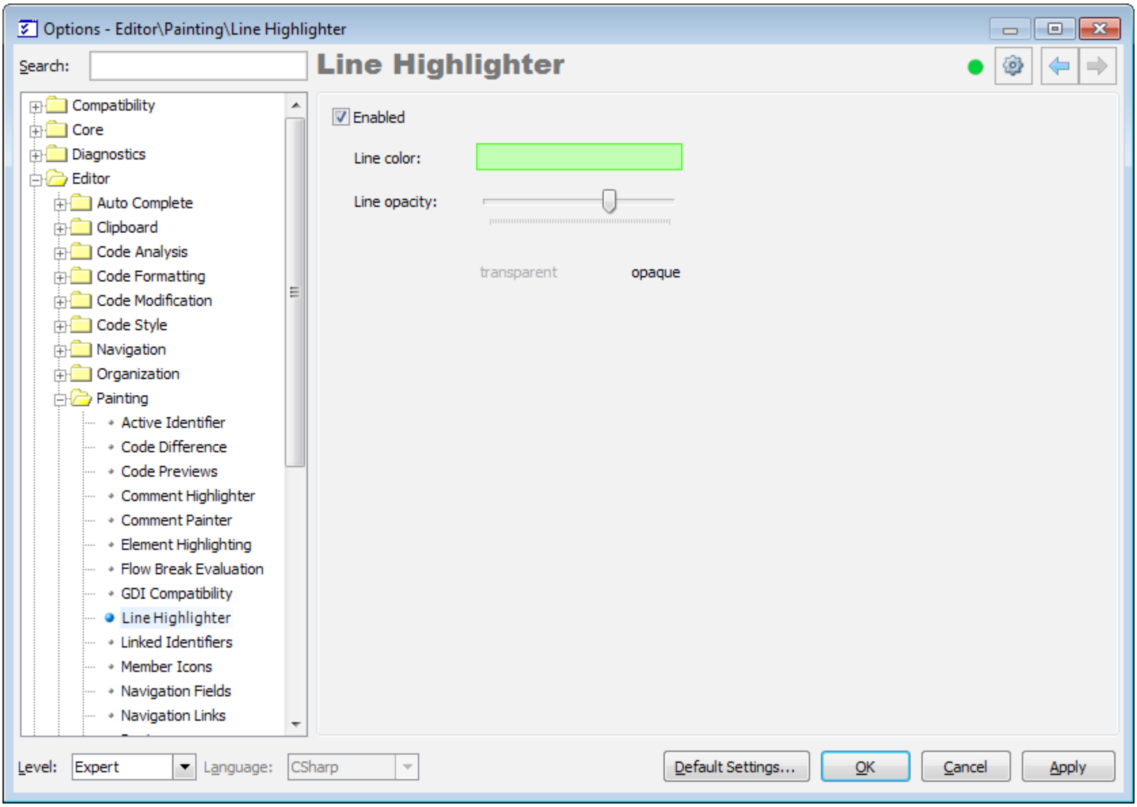
Line Highlighter

As the name says, the **Line Highlighter** feature visually indicates the current line in the Visual Studio code editor. Such a visual line indication allows you to quickly find where the tiny flashing text caret is. This is especially valuable for large screen resolutions. This is what it looks like:



```
// DXCore-generated code...
#region InitializePlugIn
public override void InitializePlugIn()
{
  base.InitializePlugIn();

  //
  // TODO: Add your initialization code here.
  //
}
#endregion
#region FinalizePlugIn
public override void FinalizePlugIn()
{
  //
  // TODO: Add your finalization code here.
  //

  base.FinalizePlugIn();
}
#endregion
```

You are able to tweak the color and opacity of the line on the **Editor** | **Painting** | **Line Highlighter** options page in the Options Dialog:



Options - Editor\Painting\Line Highlighter

Line Highlighter

☑ Enabled

Line color: 🟩

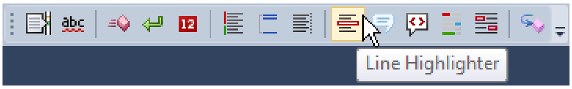
Line opacity:

transparent opaque

Level: Expert

Language: CSharp

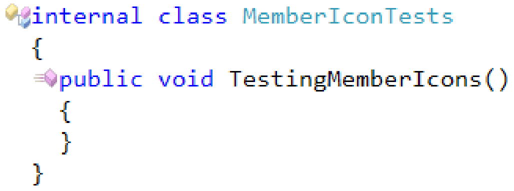
The feature is not enabled by default. To quickly enable it – toggle the appropriate button on the DXCore Visualize toolbar or check the **Enabled** checkbox on the **Line Highlighter** options page:



The plug-in containing this feature is included into the CodeRush Shared Source solution. This means you can modify its source, according to your preference.

Member Icons

The **Member Icons** visualization feature shows small icons over each member (*method*, *property*, *field*, *etc*) or type (*class*, *struct*, *interface*, *etc*). These icons appear to the left of a target member or type:



```
internal class MemberIconTests
{
  public void TestingMemberIcons()
  {
  }
}
```

There are two groups of icons that can be shown:

- **Visibility Modifier Icons**, which indicate the member visibility modifier. The following visibility modifier icons are available:

Icon	Visibility modifier
☞	Public
✉	Internal
🔑✉	Internal Protected
🔑	Protected
🔒	Private

Access modifier icons can also be displayed if the visibility keyword is omitted.

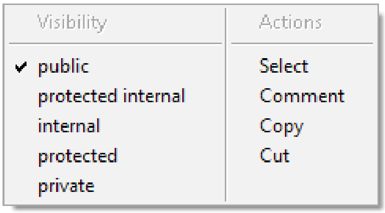
- **Member Type Icons**, which indicate the type of a member. The icons differ for static (Shared) and instance members. The following member type icons are available:

Member	Instance	Static ((Shared
Class		
Struct		
Interface		
Delegate		
Enumeration		
Method		

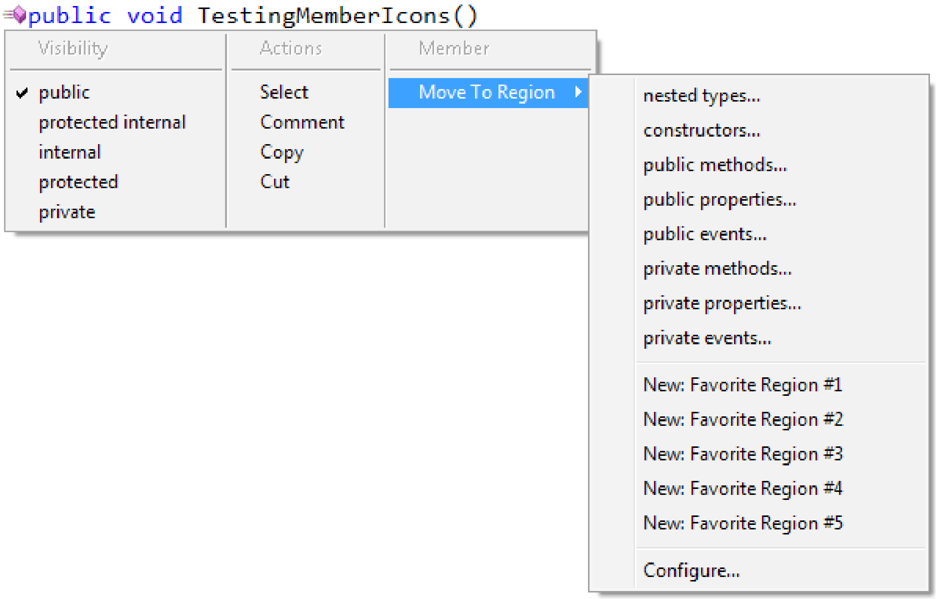
Constructor	⇛◆	⇛◇
Property	▰	▱
Indexed Property	▦	▦
Event	⚡	⚡
Constant	◆	◇
Field	◆	◇

You can click the member icon to present a drop-down menu of valid access modifiers for the member and several other actions which can be performed with a member, such as:

- Select
- Comment
- Copy
- Cut



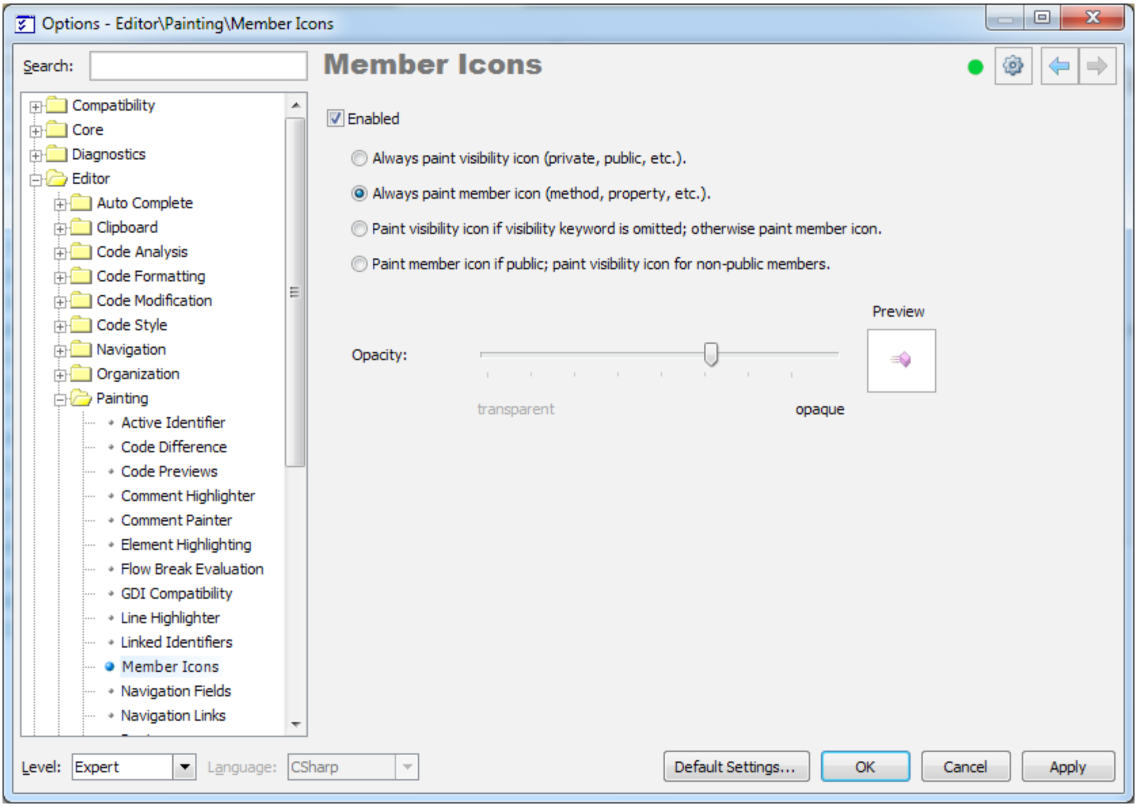
Other DXCore/CodeRush plug-ins can add additional menu items into the Member drop-down menu. An example is the "Member Mover" plug-in, which allows you to move a particular member into a region by your preference:



```
public void TestingMemberIcons()
```

You can also add any items into this menu programmatically.

The group of icons for members can be chosen on the **Editor** | **Painting** | **Member** Icons options page in the Options Dialog. There are additional settings available, such as the opacity of the icon and several icon appearance options:



Options - Editor\Painting\Member Icons

Search:

Member Icons

☑ Enabled

○ Always paint visibility icon (private, public, etc.).

● Always paint member icon (method, property, etc.).

○ Paint visibility icon if visibility keyword is omitted; otherwise paint member icon.

○ Paint member icon if public; paint visibility icon for non-public members.

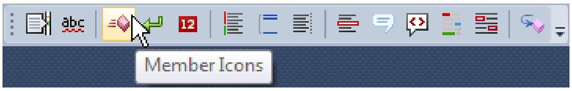
Opacity:

Preview

Level: Expert

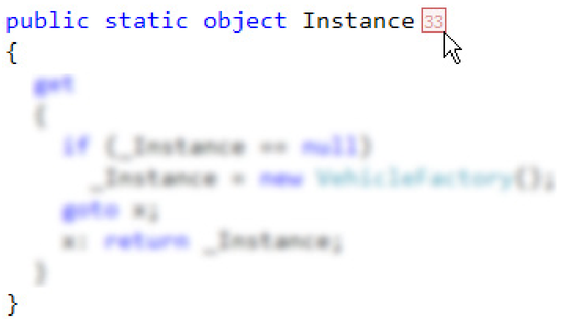
Language: CSharp

To quickly turn **Member Icons** on or off, use the appropriate icon on the DXCore Visualize toolbar:



Show Metrics

CodeRush Pro includes the **Show Metrics** (also known as **Code Metrics**) visualization feature, which is indented to evaluate code metrics right in the code editor while you are writing and/or reviewing the code. It shows a specific code metric for each method, property or event (with add and remove methods) near its declaration:



```
public static object Instance 33
{
  get
  {
    if (_Instance == null)
      _Instance = new VehicleFactory();
    goto x;
    x: return _Instance;
  }
}
```

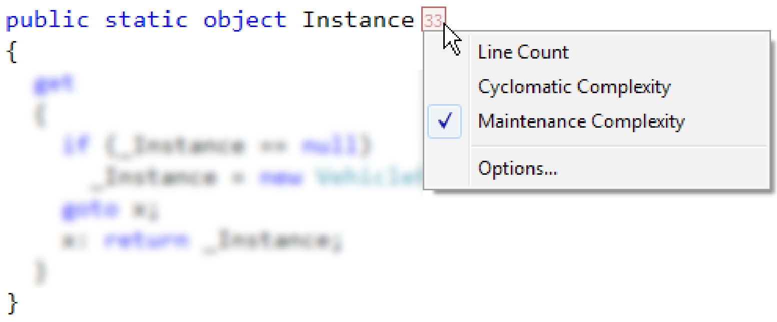
Code metrics itself is a set of software measures that provide developers with better insight into the code they are developing. Using code metrics allows program managers and developers to estimate complexity of a project already developed or even only being developed and to estimate the scope of work, stylistics of a program being developed and efforts of every developer participating in creation of a particular solution. By taking advantage of code metrics, developers can understand which types and/or methods should be reworked or tested more thoroughly.

However, metrics can only serve as recommendations, one cannot rely fully on them for developing software, programmers try to minimize or maximize this or that aspect of their program and can refer to various tricks up to reducing efficiency of the program's operation.

The following metrics are available in the **Show Metrics** feature:

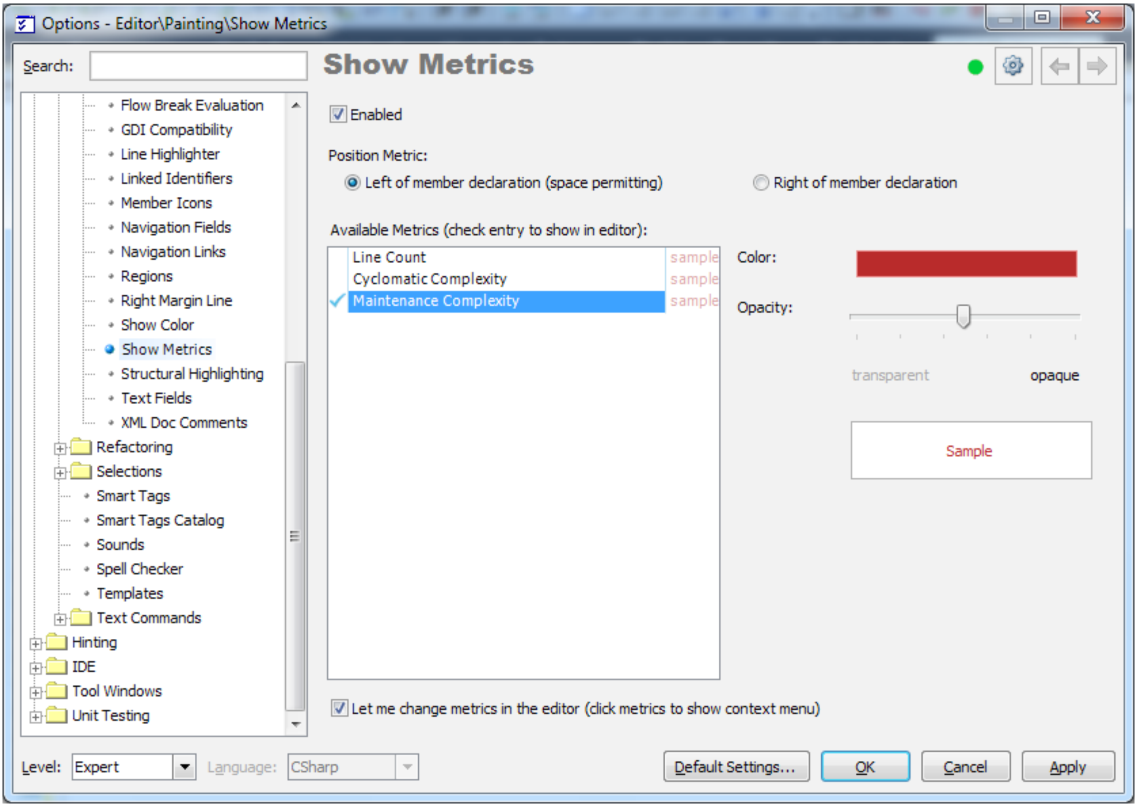
Metric Name	Description
Cyclomatic Complexity	Measures the structural complexity of the code. It is calculated using the number of different code paths in the flow of the program. A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable. To learn more about this metric see the "Feeling the need for a new software metric" topic.
Line Count	Indicates the number of lines of code. A very high count might indicate that a type or method is trying to do too much work and should be split up. It might also indicate that the type or method might be hard to maintain.
Maintenance Complexity	This code metric is developed by IDE tools chief architect – Mark Miller. To some degree it combines elements of **cyclomatic complexity** and number of function calls plus property references inside a particular member. Every element of the code has a weighted point value, all the way down to local variable declarations, assignment statements, expressions in for loops, unary operations, etc. See the "Here's your new code metric" topic for more details on this metric.

You can specify the shown metric by clicking on the **Metric icon** and choose it from the drop down menu:



```
public static object Instance 33
{
  get
  {
    if (_Instance == null)
      _Instance = new VehicleFa
    goto x;
    x: return _Instance;
  }
}
```

Or you can change it on the **Editor** | **Painting** | **Show Metrics** option page in the Options Dialog:



Show Metrics

☑ Enabled

Position Metric:

● Left of member declaration (space permitting) ○ Right of member declaration

Available Metrics (check entry to show in editor):

	Line Count	sample
	Cyclomatic Complexity	sample
✓	Maintenance Complexity	sample

Color: 🟥

Opacity: transparent — opaque

Sample

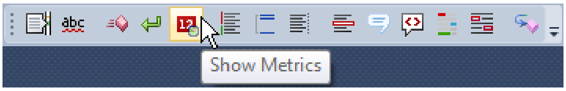
☑ Let me change metrics in the editor (click metrics to show context menu)

Level: Expert Language: CSharp

There are additional options you can specify for this feature on the options page:

- Enable or disable **Code Metrics** feature
- Position of the **Metric icon**: left of member declaration or right of member declaration
- Color and opacity for each code metric provider available
- Option that allows you to change metrics right in the code editor (by shown the drop down menu).

You are also able to quickly enable/disable this feature on the DXCore Visualize toolbar:



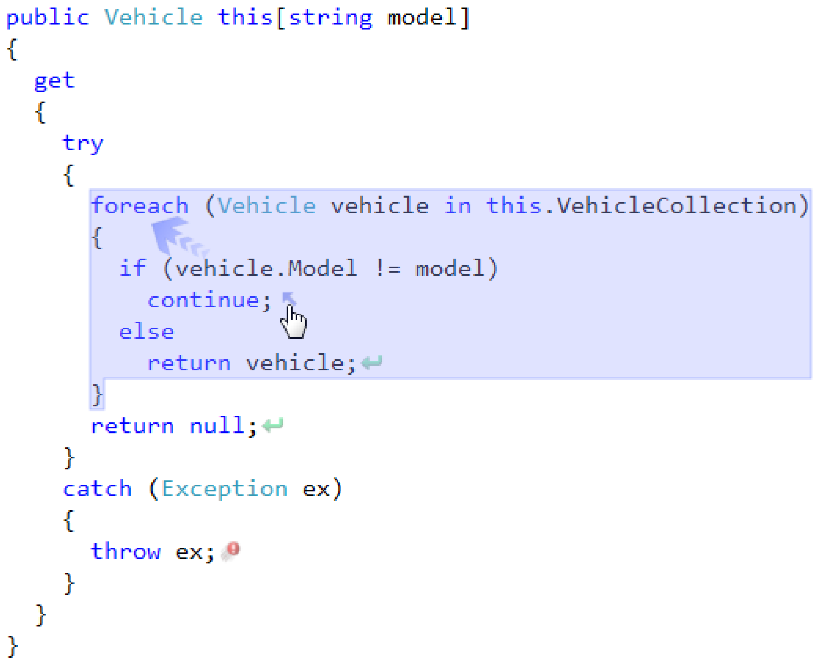
Refactor! Pro contains another feature for analyzing code metrics – it is the **Metrics tool window**, which shows you graphical presentation of code metrics for the entire source code tree.

You are welcome to add your own code metrics using the CodeMetricProvider DXCore component, which will be automatically available in both **CodeRush's Code Metrics** feature and **Refactor!'s Metrics tool window**. The feature resides in the open source "**CR_ShowMetrics**" plug-in sample from the Shared Source solution.

Flow Break Icons

Flow break icons (also known as **Flow Break Evaluation** feature) are visual indicators appearing in the code editor at the end of lines containing code that can alter the flow of program execution. **Flow break icons** are useful when reviewing complex code with multiple loops and/or breaks – these icons are drawn near the flow language keywords (see the table below) which help you understand the code flow evaluation.

Hovering over these icons using the mouse cursor will show you an animated arrow pointing to the next point of execution (where the code will continue after the break). In addition, clicking on one of those icons will move the editor caret to the target location in the source code, where the program path goes. A temporal system marker is dropped near the icon after you click an icon, to allow you to return back when you press **ESC** key (within ٦٠ seconds).



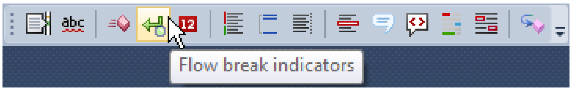
```
public Vehicle this[string model]
{
  get
  {
    try
    {
      foreach (Vehicle vehicle in this.VehicleCollection)
      {
        if (vehicle.Model != model)
          continue;
        else
          return vehicle;
      }
      return null;
    }
    catch (Exception ex)
    {
      throw ex;
    }
  }
}
```

Here's the table of **flow break icons** and language keywords that change the program flow:

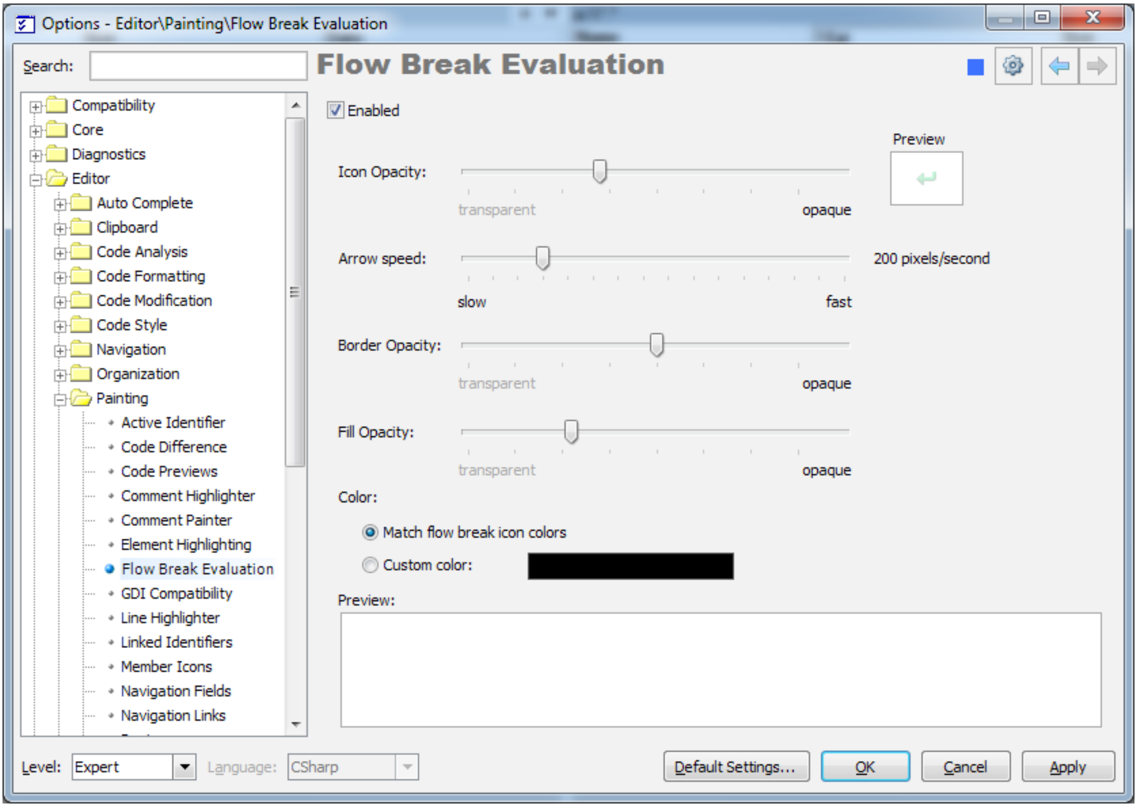
Icon	C# keyword	VB keyword	Target location
↩	return	Return	• The finally statement if located inside of a try or catch block. • The code line following the anonymous method or lambda expression if located inside of these. • The end of the current method or function or property accessor.
❗	throw	Throw	• The finally statement if located inside of a try or catch block. • The end of the current method or function or property accessor.
↖	continue	Continue	• The beginning of the current loop or finally statement if inside of a try or catch block.

↙	break	Exit Sub, Exit Function, Exit For, Exit Do, Exit While, Exit Select	• The finally statement if located inside of a try or catch block. • The statement following the current loop, if present.
→	goto	GoTo	• The code line with the appropriate label element.
	yield return	-	Same as "return" icon.
	yield break	-	Same as "return" icon.

By default, the **Flow Break Icons** feature is enabled. You can easily turn it off from the DXCore Visualize toolbar:



On the "**Editor** | **Painting** | **Flow Break Evaluation**" options page in the Options Dialog, you can specify the icon and border opacity, arrow speed and color, etc.



Flow Break Evaluation

☑ Enabled

Icon Opacity: transparent – opaque

Preview

Arrow speed: slow – fast 200 pixels/second

Border Opacity: transparent – opaque

Fill Opacity: transparent – opaque

Color:

● Match flow break icon colors

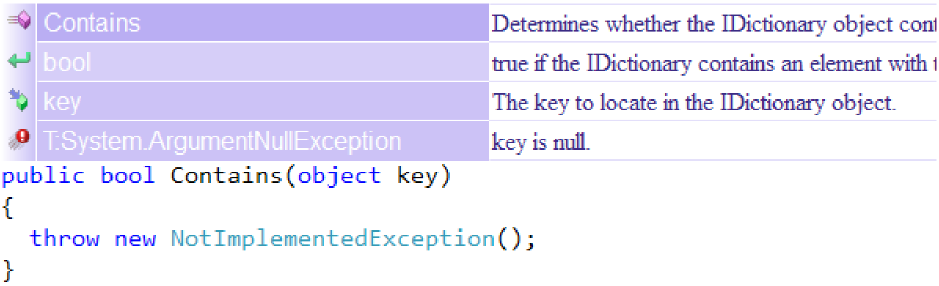
○ Custom color:

Preview:

Level: Expert Language: CSharp

XML Doc Comments Painter

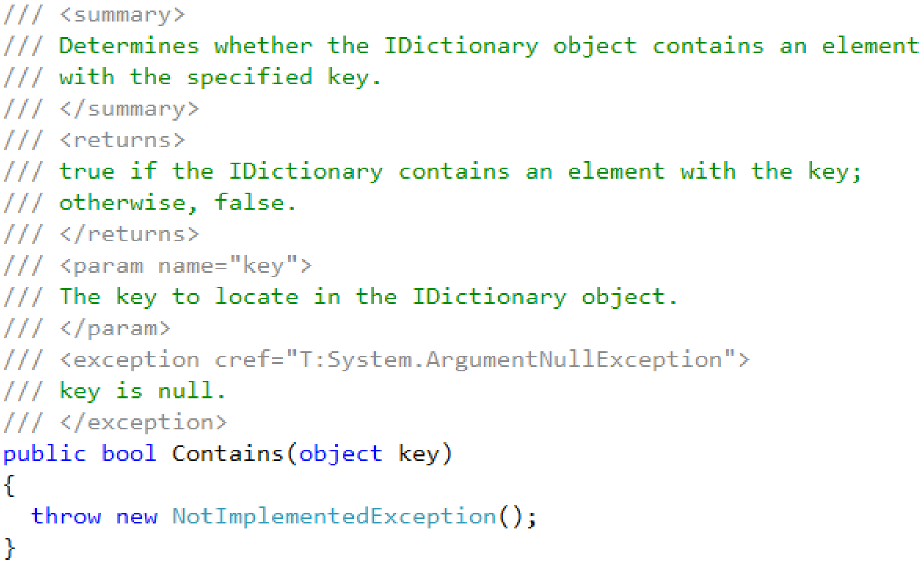
The **XML Doc Comments Painter** feature of CodeRush Pro paints over XML documentation comments in the code editor, so they are easier to read in a nice-looking form. It visually replaces all XML doc tags and make them hidden, and leave the important information only:



Contains	Determines whether the IDictionary object cont
bool	true if the IDictionary contains an element with t
key	The key to locate in the IDictionary object.
T:System.ArgumentNullException	key is null.

```
public bool Contains(object key)
{
  throw new NotImplementedException();
}
```

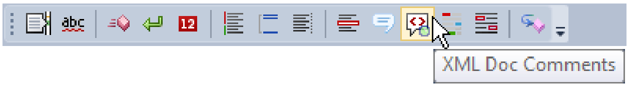
instead of this:



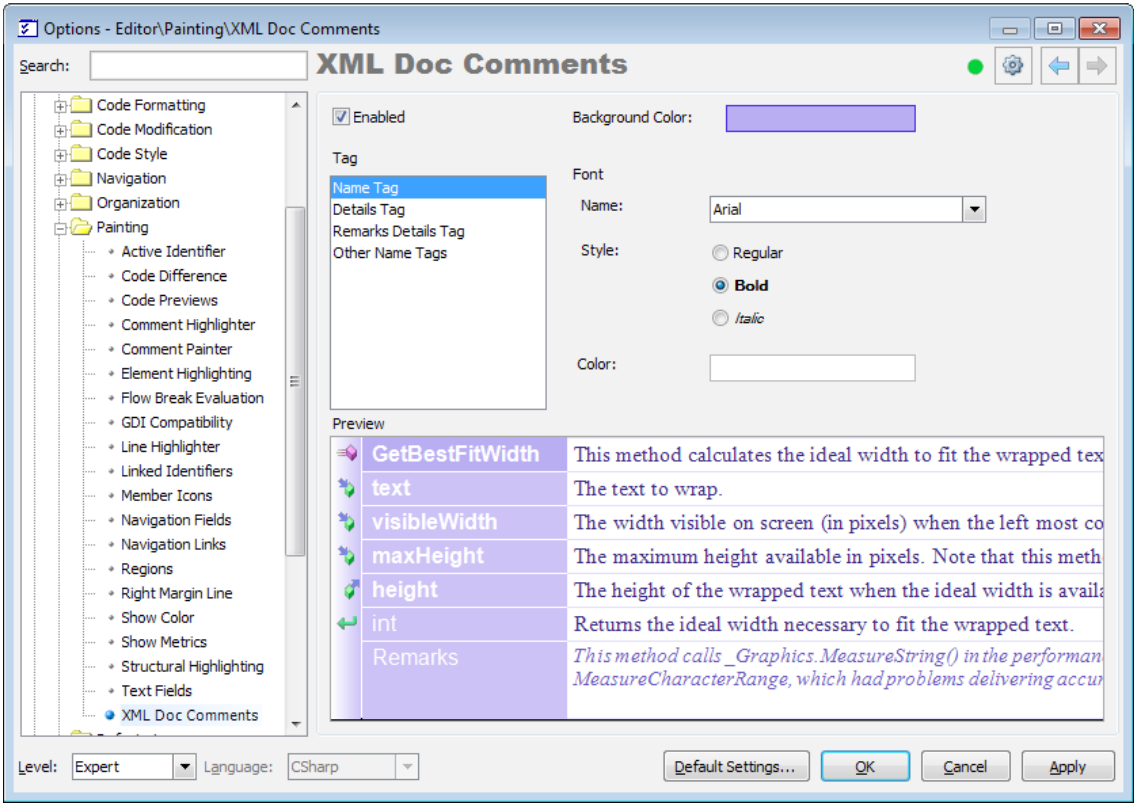
```
/// <summary>
/// Determines whether the IDictionary object contains an element
/// with the specified key.
/// </summary>
/// <returns>
/// true if the IDictionary contains an element with the key;
/// otherwise, false.
/// </returns>
/// <param name="key">
/// The key to locate in the IDictionary object.
/// </param>
/// <exception cref="T:System.ArgumentNullException">
/// key is null.
/// </exception>
public bool Contains(object key)
{
  throw new NotImplementedException();
}
```

If you want to edit an XML doc comment, just move the editor caret inside of it, so the original comment become visible.

You can quickly toggle the availability of the feature on the DXCore Visualize Toolbar:



The feature is open source from the CodeRush Shared Source Solution, so you can tweak it, according to your preference. There are numerous options available for you on the **Editor** | **Painting** | **XML Doc Comments** options page in the Options Dialog:



☑ Enabled

Background Color:

Tag
- Name Tag
- Details Tag
- Remarks Details Tag
- Other Name Tags

Font
Name: Arial
Style: ○ Regular ● **Bold** ○ *Italic*
Color:

Preview

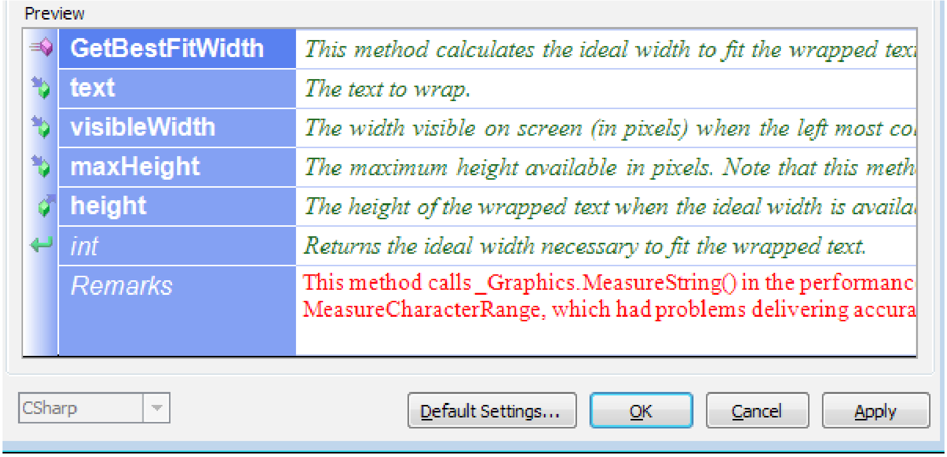
GetBestFitWidth	This method calculates the ideal width to fit the wrapped tex
text	The text to wrap.
visibleWidth	The width visible on screen (in pixels) when the left most co
maxHeight	The maximum height available in pixels. Note that this meth
height	The height of the wrapped text when the ideal width is availa
int	Returns the ideal width necessary to fit the wrapped text.
Remarks	*This method calls _Graphics.MeasureString() in the performan* *MeasureCharacterRange, which had problems delivering accur*

Level: Expert Language: CSharp

The *Enabled* check box toggles the feature on or off. The most important part of the Options page is the *Tag* list, where you can choose a cell (*Name, Details, Remarks, Other*) of a drawn table, and tweak the following options:

Option Name	Description
Background Color	The background color of the left column.
Font	The font name used for a text in a left or right column.
Style	The style of the font: Regular, Bold or Italic.
Color	The color of the font for the selected tag.

The bottom part of the Options page has a *Preview* box where you can see the current changes you've made to the feature, for example:



Preview

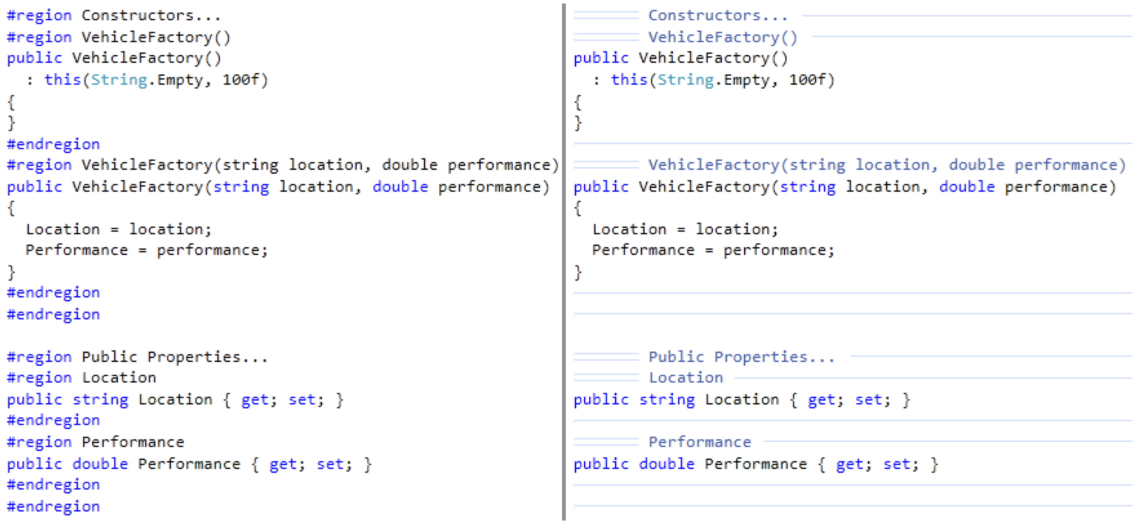
GetBestFitWidth	*This method calculates the ideal width to fit the wrapped text*
text	*The text to wrap.*
visibleWidth	*The width visible on screen (in pixels) when the left most co*
maxHeight	*The maximum height available in pixels. Note that this meth*
height	*The height of the wrapped text when the ideal width is availa*
int	*Returns the ideal width necessary to fit the wrapped text.*
Remarks	This method calls _Graphics.MeasureString() in the performanc MeasureCharacterRange, which had problems delivering accura

You can also bind a key to toggle functionality of the **XML Doc Comments Painter**, using the **ToggleXMLDocComments** action name.

Region Painting

CodeRush can paint region directives (#region, #endregion) in a different way than Visual Studio, to reduce the visual noise associated with these directives when the region is expanded. Collapsed regions can also be painted in a different color, according to your preference. If your code is full of regions, the**Region Painting** visual feature will definitely help you to concentrate and focus on the code blocks instead of insignificant stuff.

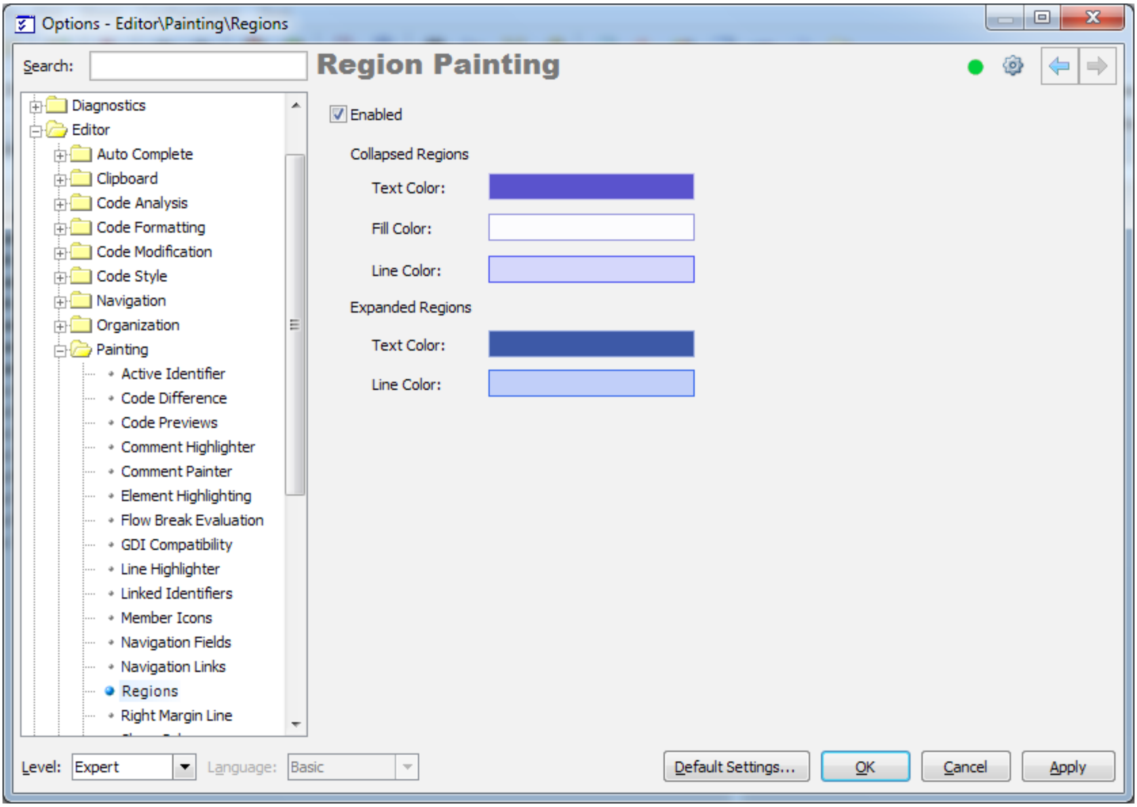
Compare these two views of a source code with the disabled feature on the left side, and enabled feature on the right:



```
#region Constructors...
#region VehicleFactory()
public VehicleFactory()
  : this(String.Empty, 100f)
{
}
#endregion
#region VehicleFactory(string location, double performance)
public VehicleFactory(string location, double performance)
{
  Location = location;
  Performance = performance;
}
#endregion
#endregion

#region Public Properties...
#region Location
public string Location { get; set; }
#endregion
#region Performance
public double Performance { get; set; }
#endregion
#endregion
```

```
        Constructors...
        VehicleFactory()
public VehicleFactory()
  : this(String.Empty, 100f)
{
}

        VehicleFactory(string location, double performance)
public VehicleFactory(string location, double performance)
{
  Location = location;
  Performance = performance;
}



        Public Properties...
        Location
public string Location { get; set; }

        Performance
public double Performance { get; set; }

```

Isn't the view on the right much better? No? OK, you're able to disable it. By default, the **Region Painting** feature is enabled. You can disable on the DXCore Visualize toolbar which contains a set of toggle buttons allowing you to turn of/off the Visualization features. You can also disable the feature within the "**Editor** | **Painting** | **Region Painting**" options page in the Options Dialog. Here, you can also specify the color settings of collapsed and expanded regions such as *line color*, *fill color* (*background color*) and *text color:*



Region Painting

☑ Enabled

Collapsed Regions

Text Color:

Fill Color:

Line Color:

Expanded Regions

Text Color:

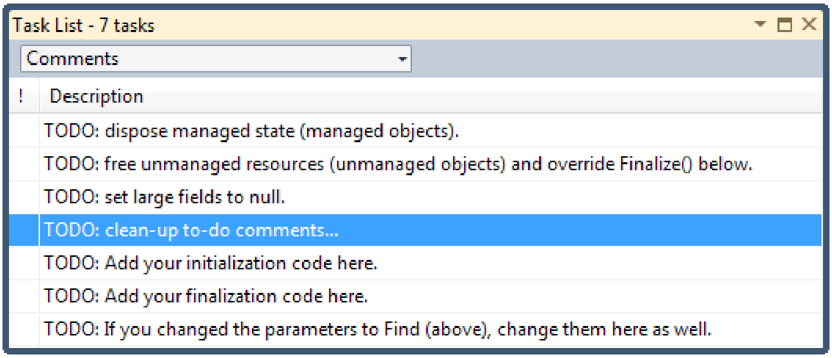
Line Color:

Level: Expert

Language: Basic

Comment Highlighter

Occasionally you may need to note a place in a source file that needs to be revisited later for correction or improvements. There are three standard codes used to designate such places: TODO, HACK and UNDONE. Visual Studio has the built-in Task List tool window, which shows the list of such comments. To open the list, go to **View** menu -> **Task List**:

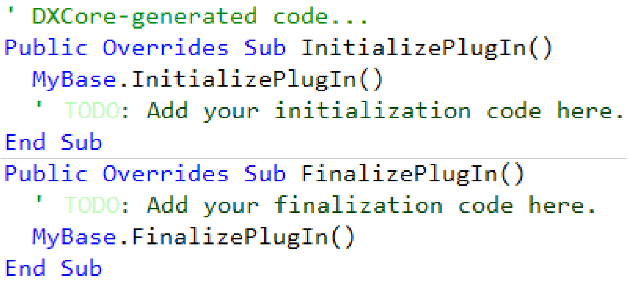

Task List - 7 tasks

Comments

!	Description
	TODO: dispose managed state (managed objects).
	TODO: free unmanaged resources (unmanaged objects) and override Finalize() below.
	TODO: set large fields to null.
	TODO: clean-up to-do comments...
	TODO: Add your initialization code here.
	TODO: Add your finalization code here.
	TODO: If you changed the parameters to Find (above), change them here as well.

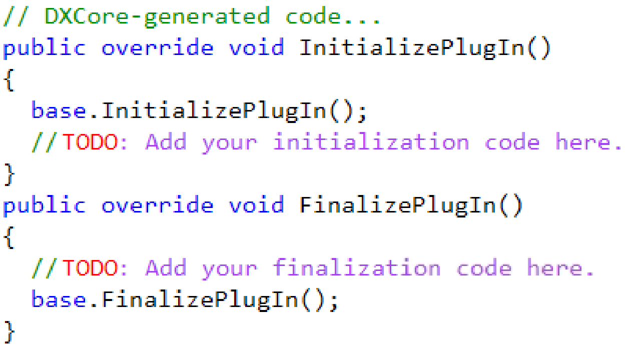
The list of the presented comments in the window can be tweaked on the “Environment -> Task List” page inside Visual Studio options dialog (Tools -> Options…).

CodeRush offers an additional visualization feature to highlight the most often used TODO comments, called **Comment Highlighter**. This feature paints the ” TODO:” comment part in a low-contrast color, and the rest of the to-do text in a different, higher-contrast color. The highlighting works in both *CSharp*and *Visual Basic*, and disappears once comment is activated, in other words, the editor caret is inside of a comment. This is what it may look like:



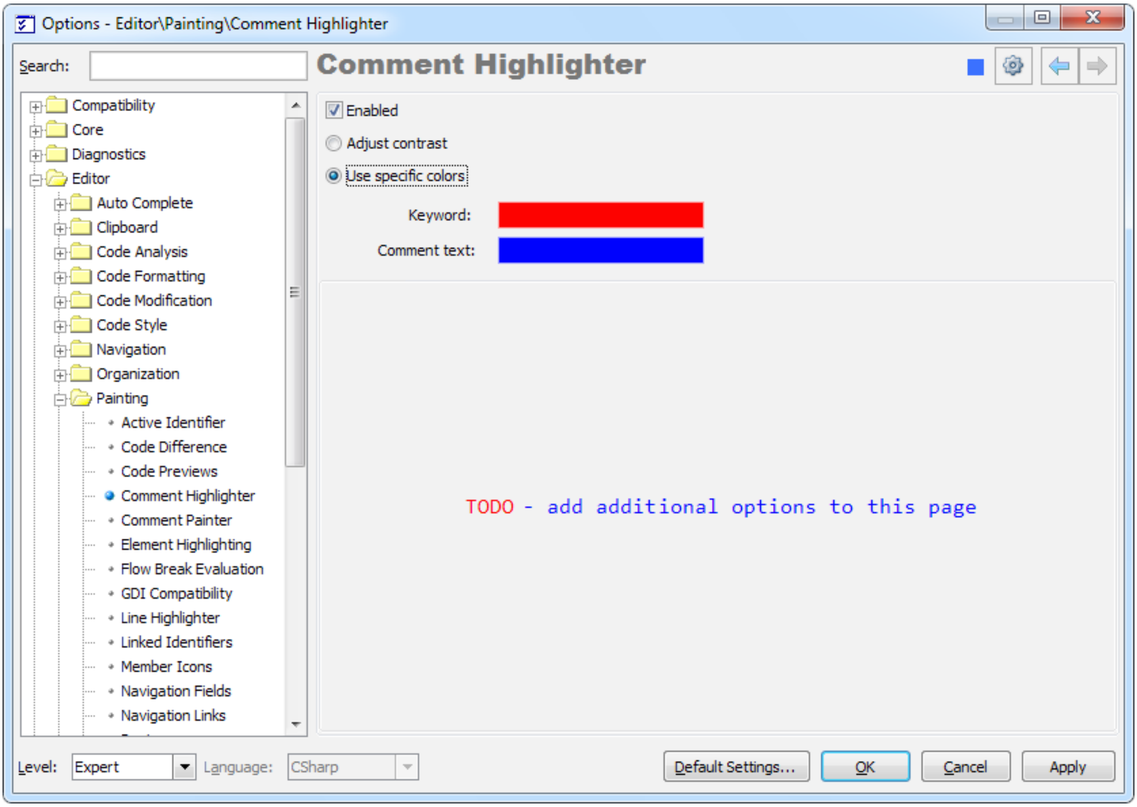
```
' DXCore-generated code...
Public Overrides Sub InitializePlugIn()
  MyBase.InitializePlugIn()
  ' TODO: Add your initialization code here.
End Sub
Public Overrides Sub FinalizePlugIn()
  ' TODO: Add your finalization code here.
  MyBase.FinalizePlugIn()
End Sub
```

On the other hand, you are able to change the colors used for highlighting of such comments instead of contrast adjustment, for example:



```
// DXCore-generated code...
public override void InitializePlugIn()
{
  base.InitializePlugIn();
  //TODO: Add your initialization code here.
}
public override void FinalizePlugIn()
{
  //TODO: Add your finalization code here.
  base.FinalizePlugIn();
}
```

These options can be changed on the **Editor** | **Painting** | **Comment Highlighter** options page in the IDETools Options Dialog (DevExpress -> Options…):



☑ Enabled

○ Adjust contrast

● Use specific colors

Keyword: 🟥

Comment text: 🟦

```
TODO - add additional options to this page
```

Level: Expert Language: CSharp

Don’t forget that there are code templates you may use for creating fast comment areas , like “/t” for “// TODO”, “/h” for “// HACK”, etc.

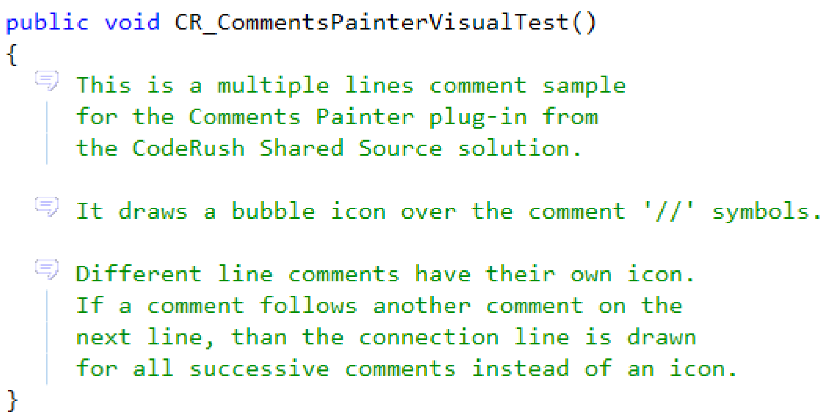
The feature resides in the open source “**CR_CommentHighlighter**” plug-in from the CodeRush Shared Source solution. This allows you to add other comments to be highlighted or improve the current implementation. The plug-in is not installed if you have version 10.2.5 or earlier of IDE Tools. You have to compile it manually from the **Shared Source** solution, located in your installation folder, e.g.

C:\Program Files\DevExpress 2010.2\IDETools\System\CodeRush\SOURCES\Shared Source.sln

In version10.2.6 and up of CodeRush Pro, this plug-in is installed, but disabled, by default. You can enable it on the options page mentioned above.

Comment Painter

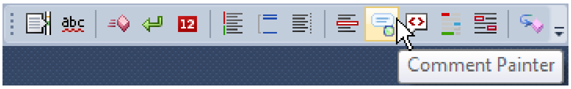
The **Comment Painter** feature draws a bubble icon over the comment “//” symbols in CSharp, or “ ‘ ” (single quote) in Visual Basic. The source code of this feature shows how to paint bitmaps on the code editor. If the comment is active (e.g. text caret is located inside the comment) then no icon is drawn, allowing you edit the comment.



```
public void CR_CommentsPainterVisualTest()
{
    This is a multiple lines comment sample
    for the Comments Painter plug-in from
    the CodeRush Shared Source solution.

    It draws a bubble icon over the comment '//' symbols.

    Different line comments have their own icon.
    If a comment follows another comment on the
    next line, than the connection line is drawn
    for all successive comments instead of an icon.
}
```

The feature resides in the open source "**CR_CommentPainter**" plug-in sample from the Shared Source solution.

On the "***Editor\Painting\Comment Painter***" options page in the Options Dialog, you can enable or disable this feature. By default, this feature is installed but inactive (disabled). You're also able to quickly enable/disable this feature on the DXCore Visualize toolbar:



Chapter 11. Code refactoring support

Introduction

CodeRush is the defacto standard refactoring toolset for Visual Studio. It fuses a language-independent state-of-the-art code shaping engine with a highly-optimized user experience. Unlike other refactoring solutions that target the system architect, CodeRush is designed to help all developers craft and sculpt their code with the speed and efficiency needed to meet their line of business demands. Refactor! combines extreme power with unprecedented ease of use.

Key advantages:

- Many time-consuming and potentially error causing code rearrangement tasks you deal with everyday are performed automatically. In most cases, you can forget about copying pieces of code to the clipboard, pasting them to new locations and correcting expressions to match the syntax. You can forget about having to find and replace each and every variable, field or constant entry to change its name to avoid breaking your code. All you have to do is press a shortcut and choose the desired refactoring – and you automatically get the new code you need, which will compile and do exactly the same thing as your old code did.

- When you write code, you focus on a single task. Assume that you are writing a method that uses the same number or string in several places, so you want to create a constant which will make it easier to manage the code. If you're not using CodeRush, you'll have to decide whether to create it immediately (moving the caret, introducing a constant, returning to the method you were writing) or introduce it after you've finished with the method (moving the caret, introducing the constant and then accurately replacing all the necessary entries). With CodeRush, you don't have to choose – just write the code using your numbers or string – CodeRush will introduce a constant when you ask, and it will replace all the entries, so that you don't have to jump around in your code. This means you can focus on the functionality and not the formatting and syntax. If you are writing a complex method and thinking of extracting a part of it into another method, you will also be happy with CodeRush. Just concentrate on the method without worrying about the formatting, CodeRush will handle this for you. After you've finished, CodeRush will extract the piece of code you select, and automatically create the correct syntax. And there are many more examples…

What is refactoring? Benefits of a code refactoring

*"**Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring."*

Martin Fowler, www.refactoring.com

Code refactoring is the process of changing a computer program's source code without modifying its external functional behavior, in order to improve some of the nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

Refactoring is usually motivated by noting a code's smell. For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by refactoring the source code, or transforming it into a new form that behaves the same as before, but that no longer "smells". For a long routine, extract one or more smaller subroutines. Or, for duplicate routines, remove the duplication and utilize one shared function in their place. Failure to perform refactoring can result in accumulating technical debt.
There are two general categories of benefits to the activity of refactoring**:**

1. **Maintainability**. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.

2. **Extensibility**. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none may have existed before.

From www.wikipedia.org – Code refactoring

If we take a look at the refactorings provides, we may see that the list is very long. This may make you think that performing a refactoring can be a difficult task. For example, to use built-in Visual Studio refactorings, you need to use a menu, or remember all shortcuts for every refactoring. However, on the other hand, in CodeRush there's only a single shortcut to perform a refactoring on a block of code. By default, the shortcut is bound to **CTRL+`** (back tick). This shortcut brings a popup menu with available refactorings at the caret position. There are also a few different ways to perform a code refactoring.

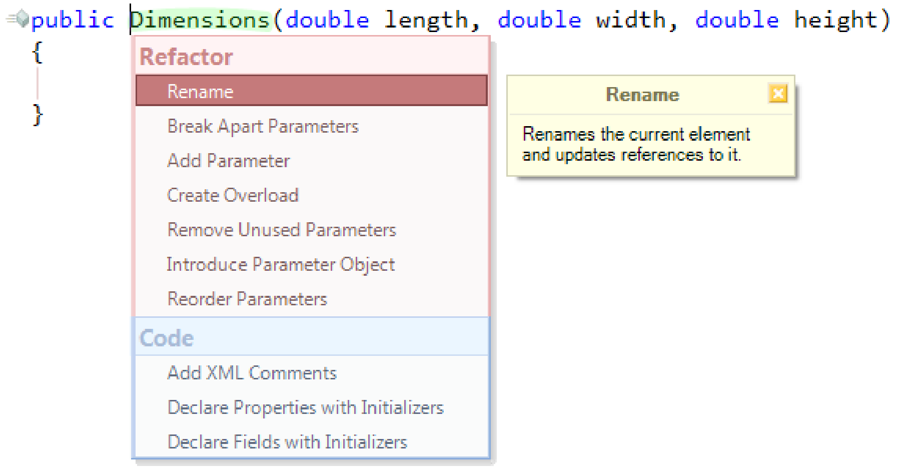
Almost all refactorings have a preview hint which allows you to see the resulting code before a particular refactoring is performed – to learn more, see the "Refactorings preview hinting" topic.

How to perform a code refactoring

There are several methods which allow you to perform a particular refactoring. Multiple methods are provided to ensure that you feel comfortable inside the Visual Studio IDE. These approaches are available:

- The **Refactor! popup menu** (shown via a single shortcut)

There is a built-in shortcut that invokes the **Refactor! popup menu** and lists all available refactoring/code operations for the current caret position and/or code selection. The default shortcut is the **CTRL+`** (back tick) key combination. Note, that you can configure the shortcut to immediately apply a refactoring if only one operation is available. See the "**IDE tools smart tag**" section below to learn more about possible options.



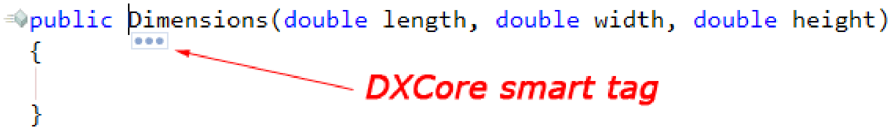
```
public Dimensions(double length, double width, double height)
{
}
```

Renames the current element and updates references to it.

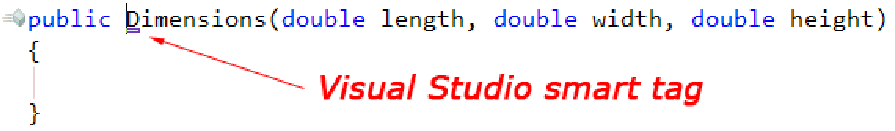
You can also create custom shortcuts to invoke the specific refactoring immediately, without having to go through the popup menu. To learn more, see the "How to assign a separate shortcut to a particular refactoring" topic.

- **DXCore smart tag** (and/or **Visual Studio**'s **smart tag**)

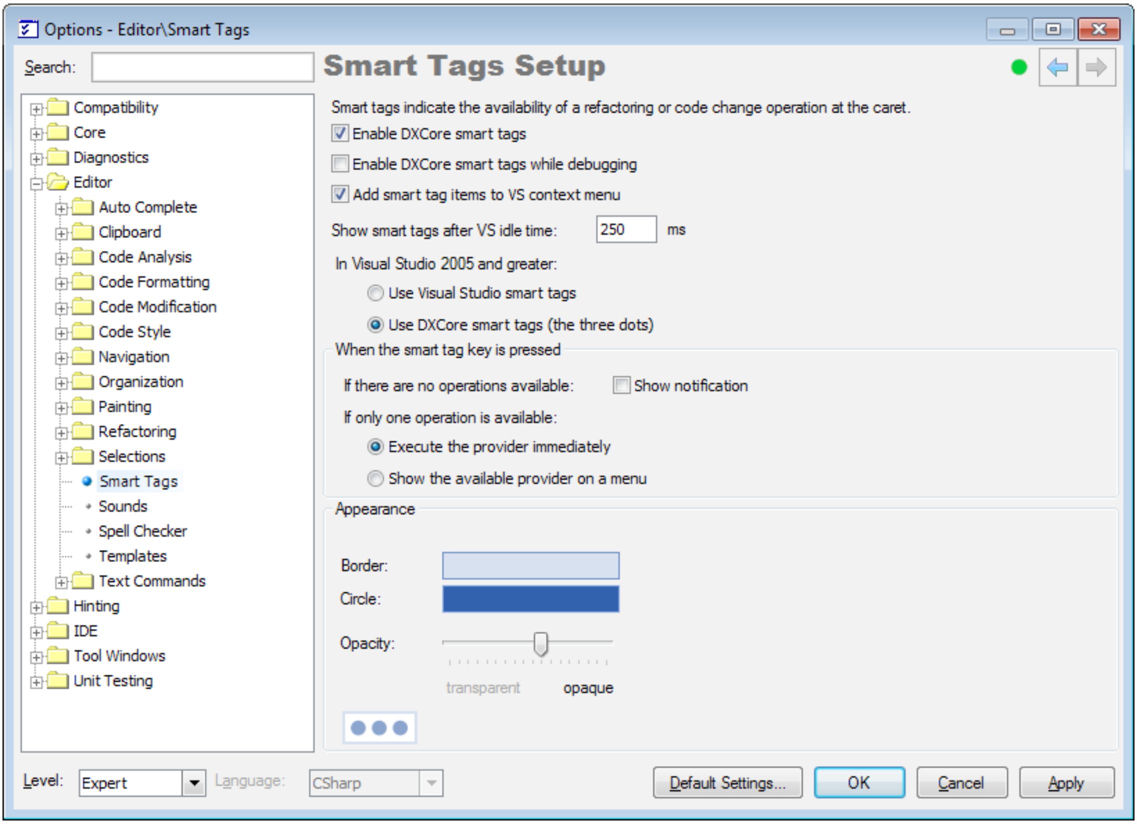
The other way of invoking the **Refactor! popup menu** is to click a **DXCore smart tag**. The type of a smart tag style is configurable, so you may use the built-in Visual Studio's smart tag instead. You can tweak lots of the **DXCore smart tag**'s options on the "**Smart Tags**" options page in the Options Dialog. The page allows you to choose whether to use the **DXCore** or **Visual Studio**'s **smart tag**, appearance of the **DXCore smart tag**, presence of smart tag items in the **Visual Studio**'s **context menu** and the default behavior of the smart tag shortcut, when only one operation is available.



```
public Dimensions(double length, double width, double height)
{
}
```



```
public Dimensions(double length, double width, double height)
{
}
```



Options - Editor\Smart Tags

Smart Tags Setup

Smart tags indicate the availability of a refactoring or code change operation at the caret.

☑ Enable DXCore smart tags

☐ Enable DXCore smart tags while debugging

☑ Add smart tag items to VS context menu

Show smart tags after VS idle time: 250 ms

In Visual Studio 2005 and greater:

○ Use Visual Studio smart tags

● Use DXCore smart tags (the three dots)

When the smart tag key is pressed

If there are no operations available: ☐ Show notification

If only one operation is available:

● Execute the provider immediately

○ Show the available provider on a menu

Appearance

Border:

Circle:

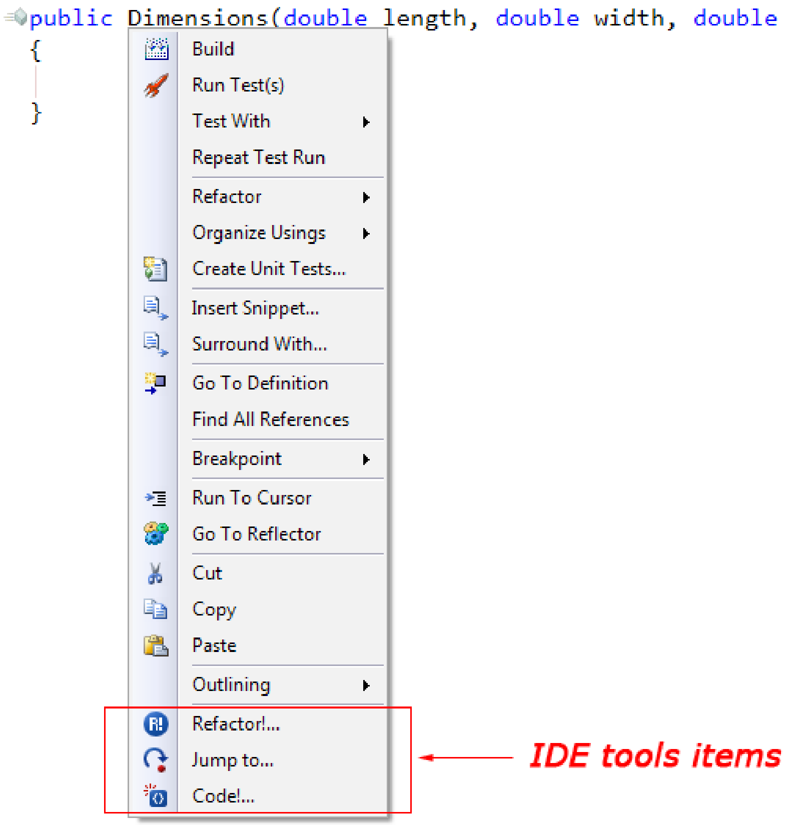
Opacity:

Level: Expert

Language: CSharp

- **Visual Studio**'s **context menu**

IDE tools integrate their items into the Visual Studio's code editor context menu. You can configure adding these items on the same "**Smart Tags**" options page above.



```
public Dimensions(double length, double width, double height)
{

}
```

- **Intelligent Paste**

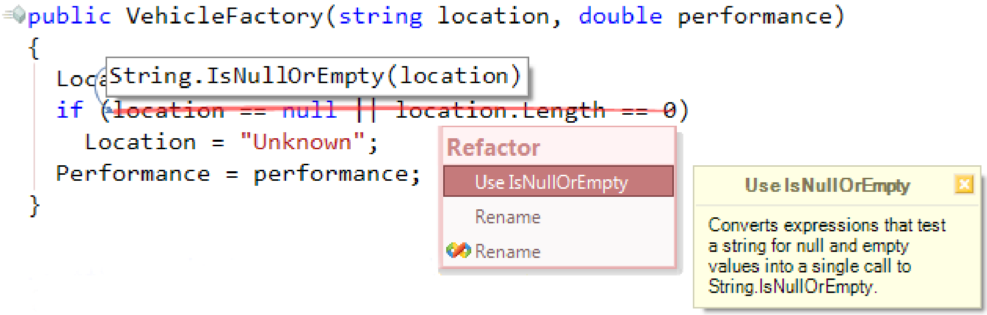
Some refactorings/code operations can be applied by cutting a particular code block and pasting it at a particular position. This feature is only available for a small subset of refactorings that involve moving code blocks. These refactorings are:

- Extract Method
- Introduce Local
- Declare Method

Note: you can control the availability of **Intelligent Paste** and its behavior via the "**Editor** | **Clipboard** | **Intelligent Paste Setup**" page in the Options dialog. The page allows you to change the maximum number of lines that you can cut and paste to apply the **Intelligent Paste** feature.

Refactorings preview hinting

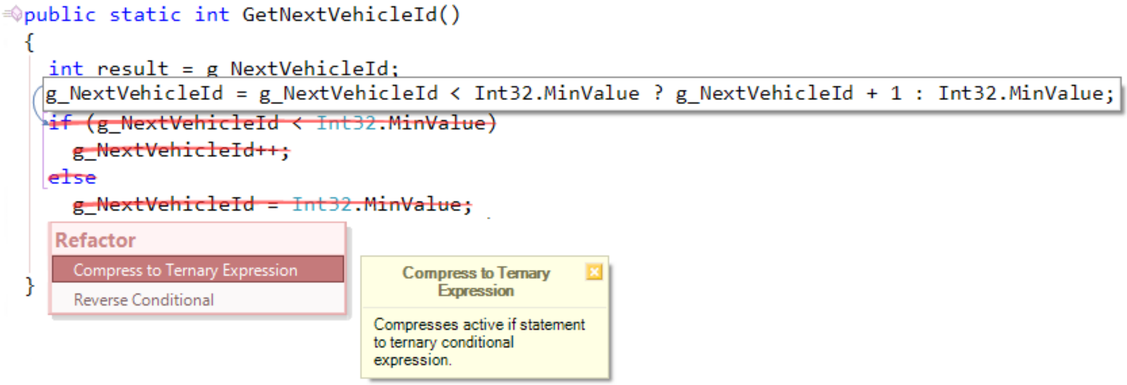
Almost all refactorings have a preview hint which allows you to see the resulting code before a particular refactoring is performed. This is very useful, because it is a great help to understand what will happen in the code once an action is taken. Moving among available refactorings in the **Refactor! popup menu** or a **SmartTag popup menu** will be reflected in the preview for each one. Here are previews of some possible refactorings you can perform using the **Refactor! popup menu**:



```
public VehicleFactory(string location, double performance)
{
  Loc String.IsNullOrEmpty(location)
  if (~~location == null || location.Length == 0~~)
    Location = "Unknown";
  Performance = performance;
}
```

Use IsNullOrEmpty

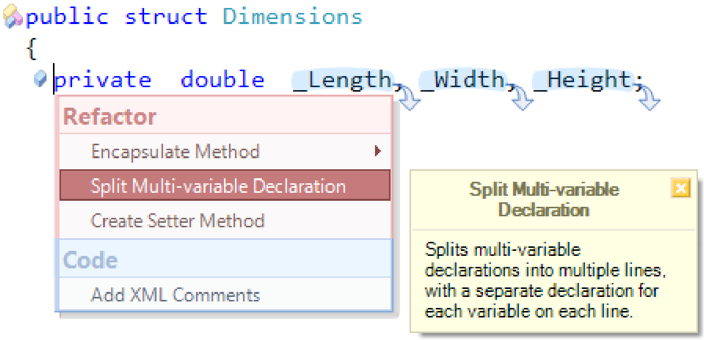
Converts expressions that test a string for null and empty values into a single call to String.IsNullOrEmpty.



```
public static int GetNextVehicleId()
{
  int result = g_NextVehicleId;
  g_NextVehicleId = g_NextVehicleId < Int32.MinValue ? g_NextVehicleId + 1 : Int32.MinValue;
  ~~if (g_NextVehicleId < Int32.MinValue)~~
    ~~g_NextVehicleId++;~~
  ~~else~~
    ~~g_NextVehicleId = Int32.MinValue;~~
}
```

Compress to Ternary Expression

Compresses active if statement to ternary conditional expression.



```
public struct Dimensions
{
  private double _Length, _Width, _Height;
```

Split Multi-variable Declaration

Splits multi-variable declarations into multiple lines, with a separate declaration for each variable on each line.

While the **Refactor! popup menu** is up, you are able to control preview hinting by holding the following keys:

- **CTRL** – will hide the menu and the big yellow hint with a description of a refactoring.
- **SHIFT** – will hide the resulting code preview window with a blue arrow pointing to the target code location

NOTE: these two keys aren’t available if you’re using a different way of listing available refactorings, e.g. **SmartTag popup menu**.

Refactorings that change signatures

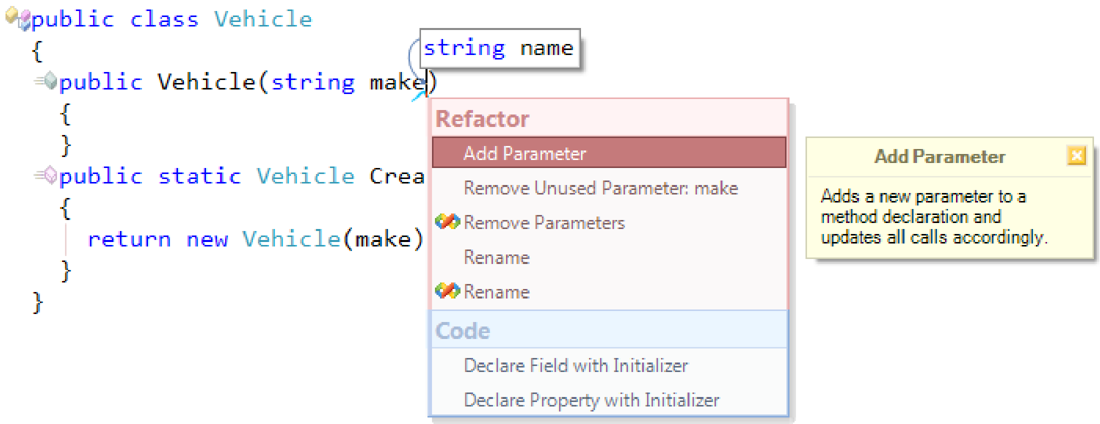
Add Parameter

The refactoring adds a new parameter to a method declaration and updates all calls accordingly. This refactoring is useful when you need to quickly add a new parameter to an existing method because it needs more information from its caller that wasn't passed in before. Bear in mind that if there are alternatives available against doing this refactoring, it is preferred to use those alternatives instead, because they don't lead to increasing the length of parameters lists. Long parameter lists are hard to remember and often involve data clumps.

Opposite:

The opposite of this refactoring is the Remove Parameter refactoring. As an alternative you may use the Introduce Parameter Object refactoring, which consolidates selected parameters of a method into a single class or structure, and simplifies the method's parameters list.

Sample:

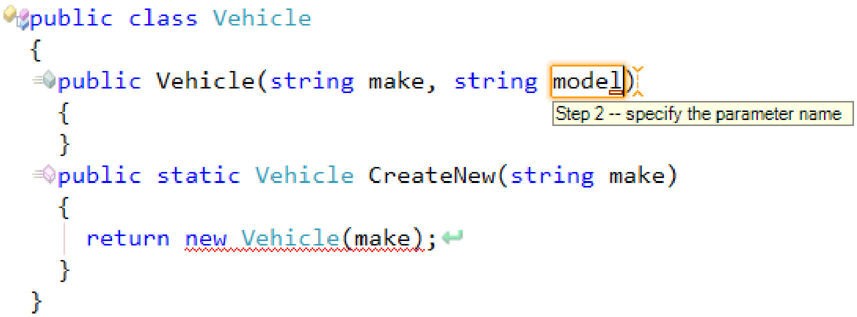


```
public class Vehicle
{
  public Vehicle(string make)
  {
  }
  public static Vehicle Crea
  {
    return new Vehicle(make)
  }
}
```

string name
Refactor
Add Parameter
Remove Unused Parameter: make
Remove Parameters
Rename
Rename
Code
Declare Field with Initializer
Declare Property with Initializer
Add Parameter

Adds a new parameter to a method declaration and updates all calls accordingly.

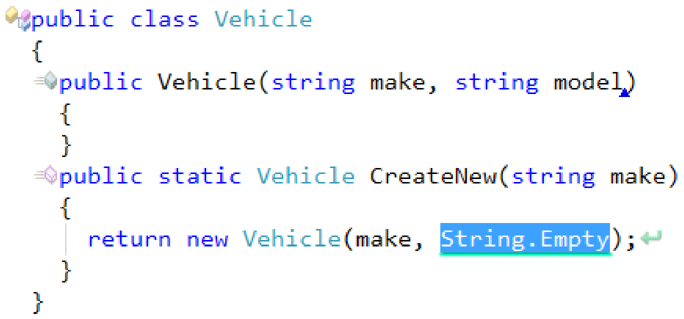
The interactive phase of refactoring:



```
public class Vehicle
{
  public Vehicle(string make, string model)
  {
  }
  public static Vehicle CreateNew(string make)
  {
    return new Vehicle(make);
  }
}
```

Step 2 -- specify the parameter name

Result:



```
public class Vehicle
{
  public Vehicle(string make, string model)
  {
  }
  public static Vehicle CreateNew(string make)
  {
    return new Vehicle(make, String.Empty);
  }
}
```

Notes:

- This refactoring enables **Text Fields** to make it easier for you to type the parameter type and name.

- Once you've finished entering the parameter type and name, each method call is updated with a new parameter. An empty value corresponding to the specified type is used by default (*0* for integer, *String. Empty* for string, etc).

- Automatically created **Navigation Links** allow you to easily navigate through all method calls and change default parameter values.

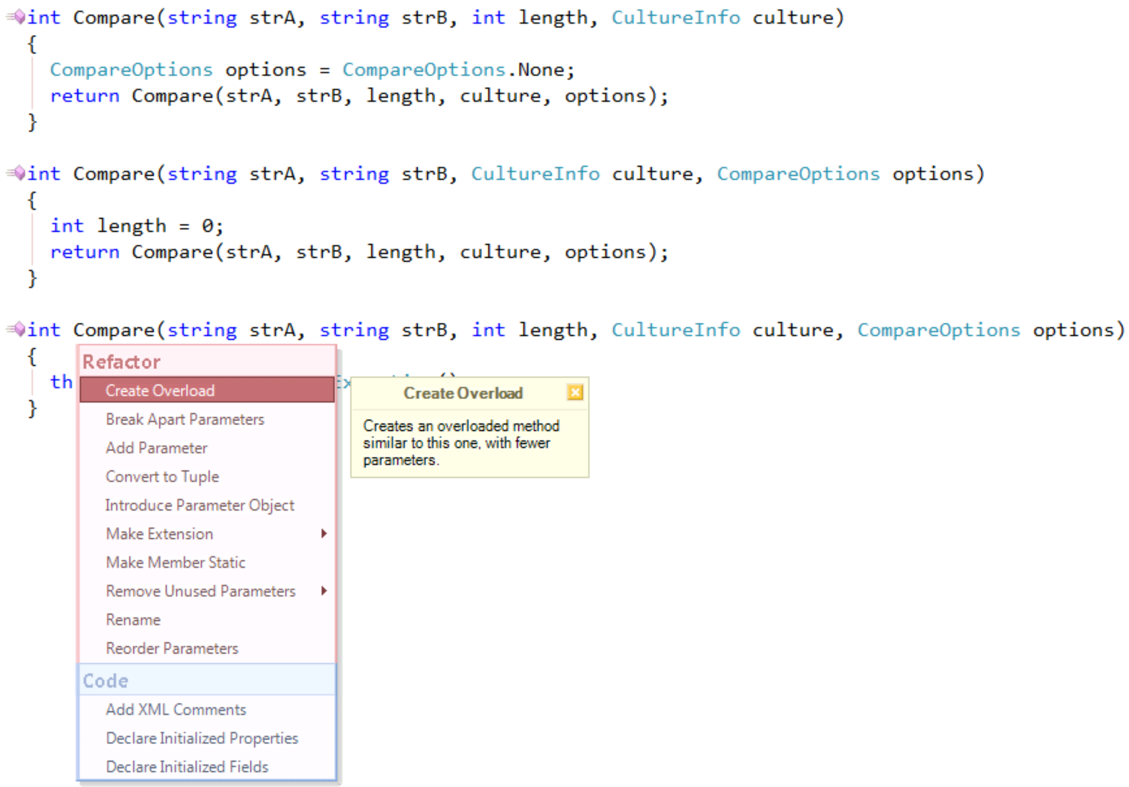
- A marker is dropped at parameter declaration statement. This allows you to easily get back to the method declaration when you are done with parameter values.

- The refactoring is pretty simple and doesn't have any options available.

Create Overload

The process of creating more than one method in a class with the same name is called as method overloading. Method overloading allows the developer to define several methods with the same name but with a different set of parameters. This way, one does not have to remember the names of multiple functions that serve a similar core purpose. When a call is made to one of these overloaded methods, the compiler automatically determines which of the methods should be used according to the arguments used in the call.

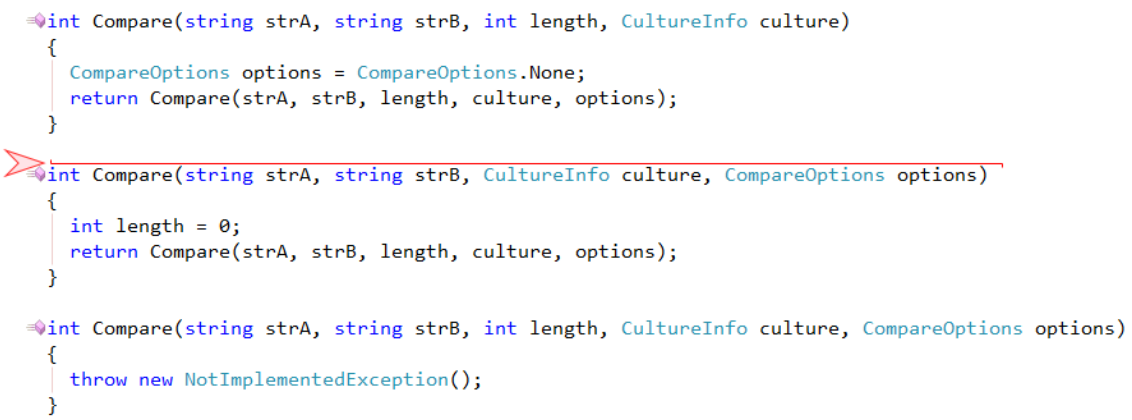
For method overloading purposes, you can utilize the **Create Overload** refactoring. Once you apply the refactoring on a method:



```
int Compare(string strA, string strB, int length, CultureInfo culture)
{
  CompareOptions options = CompareOptions.None;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, CultureInfo culture, CompareOptions options)
{
  int length = 0;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
{
  th
}
```

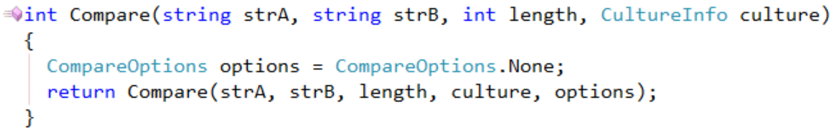
Creates an overloaded method similar to this one, with fewer parameters.

you can choose the destination of the new overload with the target picker:

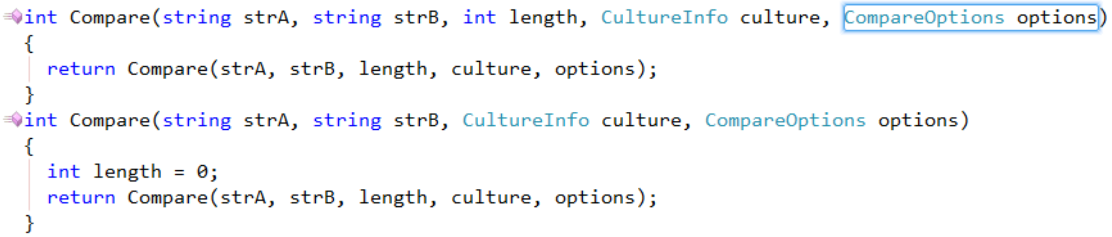


```
int Compare(string strA, string strB, int length, CultureInfo culture)
{
  CompareOptions options = CompareOptions.None;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, CultureInfo culture, CompareOptions options)
{
  int length = 0;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
{
  throw new NotImplementedException();
}
```

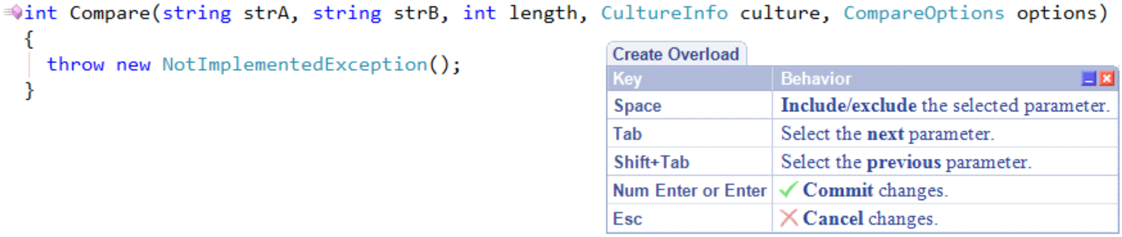
When you are done with target selection, press Enter to commit. Now, you are able to choose what parameters you want to remove from the initial method to create a new overload. Initially, the newly declared method has the same signature as the source method:



```
int Compare(string strA, string strB, int length, CultureInfo culture)
{
  CompareOptions options = CompareOptions.None;
  return Compare(strA, strB, length, culture, options);
}
```



```
int Compare(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
{
  return Compare(strA, strB, length, culture, options);
}
int Compare(string strA, string strB, CultureInfo culture, CompareOptions options)
{
  int length = 0;
  return Compare(strA, strB, length, culture, options);
}
```



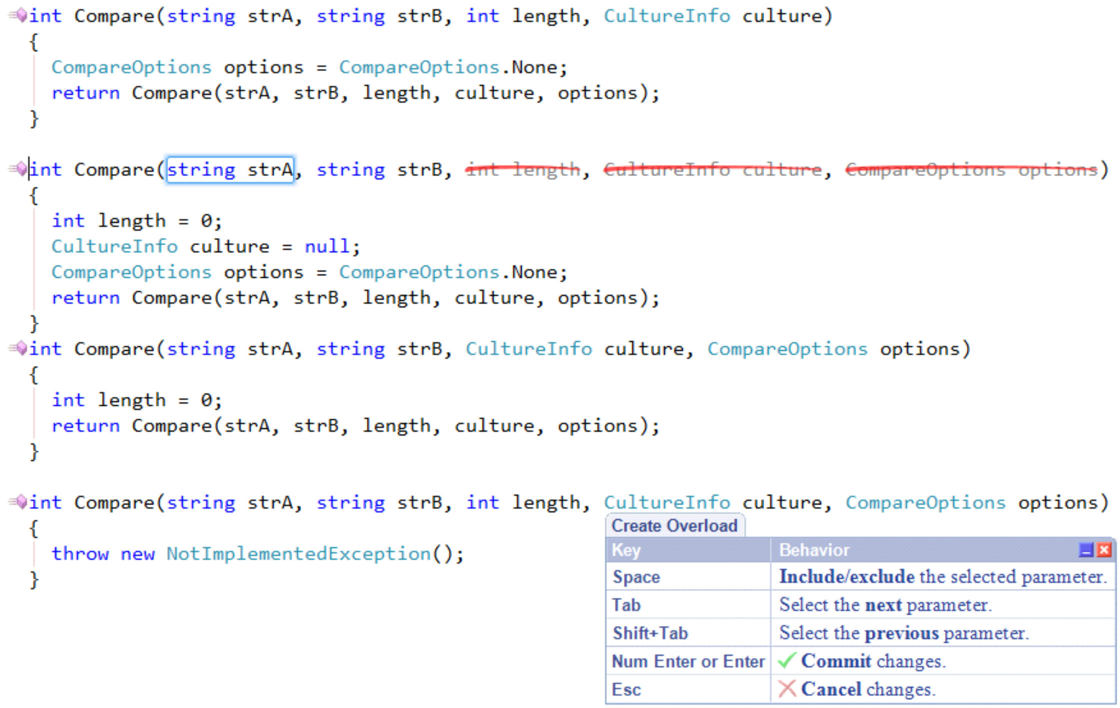
```
int Compare(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
{
  throw new NotImplementedException();
}
```

Create Overload

Key	Behavior
Space	**Include/exclude** the selected parameter.
Tab	Select the **next** parameter.
Shift+Tab	Select the **previous** parameter.
Num Enter or Enter	✓ **Commit** changes.
Esc	✗ **Cancel** changes.

While overloading methods, a rule to follow is that overloaded methods must differ, either in the number of arguments they take or the data type of at least one argument. The refactoring allows you to remove any number of parameters to create a new overload. If you want to add an additional parameter, use theAdd Parameter refactoring for this purpose.

The shortcuts hint will help you with the available keyboard keys used to remove parameters. You can use arrow keys to select a parameter, and the Space bar to remove the currently selected parameter. The preview hint will illustrate the resulting method overload declaration, which parameters will be removed, and how the parameters will be replaced with the local declarations initialized to default values:

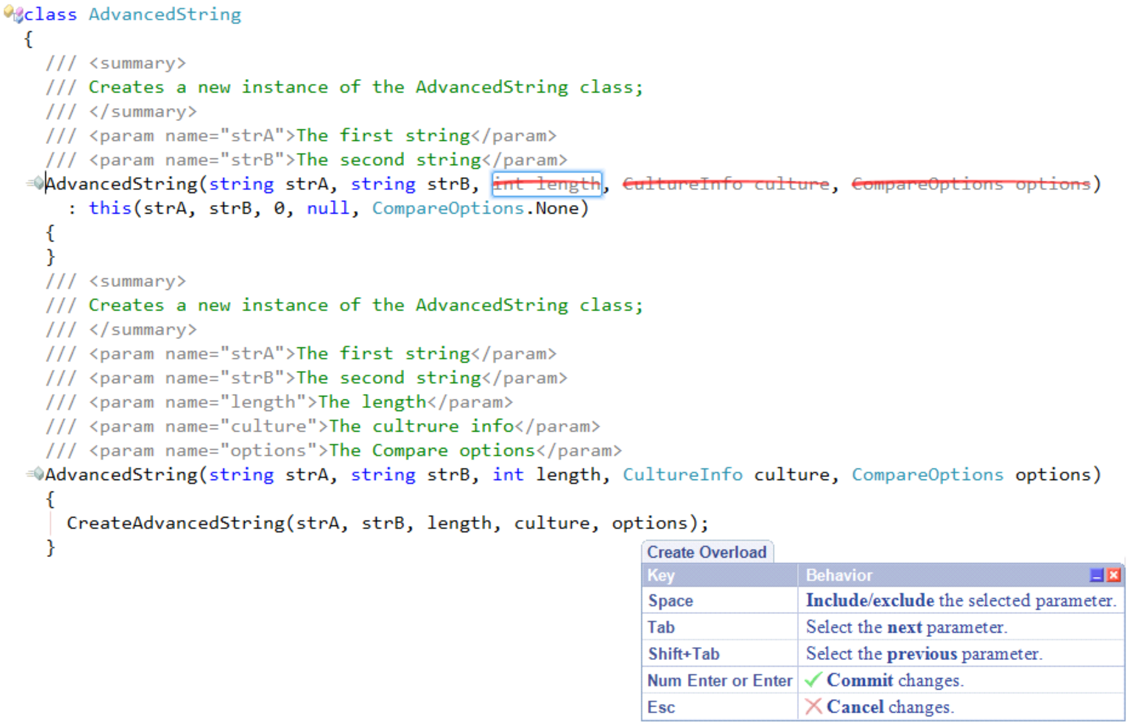


```
int Compare(string strA, string strB, int length, CultureInfo culture)
{
  CompareOptions options = CompareOptions.None;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, ~~int length~~, ~~CultureInfo culture~~, ~~CompareOptions options~~)
{
  int length = 0;
  CultureInfo culture = null;
  CompareOptions options = CompareOptions.None;
  return Compare(strA, strB, length, culture, options);
}
int Compare(string strA, string strB, CultureInfo culture, CompareOptions options)
{
  int length = 0;
  return Compare(strA, strB, length, culture, options);
}

int Compare(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
{
  throw new NotImplementedException();
}
```

Create Overload

Key	Behavior
Space	**Include/exclude** the selected parameter.
Tab	Select the **next** parameter.
Shift+Tab	Select the **previous** parameter.
Num Enter or Enter	✓ **Commit** changes.
Esc	✗ **Cancel** changes.

You can also create overloaded constructors as well as methods. In this case, the additional constructors are simply added to the code with a call to the source constructor:



```
class AdvancedString
{
  /// <summary>
  /// Creates a new instance of the AdvancedString class;
  /// </summary>
  /// <param name="strA">The first string</param>
  /// <param name="strB">The second string</param>
  AdvancedString(string strA, string strB, ~~int length~~, ~~CultureInfo culture~~, ~~CompareOptions options~~)
    : this(strA, strB, 0, null, CompareOptions.None)
  {
  }
  /// <summary>
  /// Creates a new instance of the AdvancedString class;
  /// </summary>
  /// <param name="strA">The first string</param>
  /// <param name="strB">The second string</param>
  /// <param name="length">The length</param>
  /// <param name="culture">The cultrure info</param>
  /// <param name="options">The Compare options</param>
  AdvancedString(string strA, string strB, int length, CultureInfo culture, CompareOptions options)
  {
    CreateAdvancedString(strA, strB, length, culture, options);
  }
```

Create Overload

Key	Behavior
Space	**Include/exclude** the selected parameter.
Tab	Select the **next** parameter.
Shift+Tab	Select the **previous** parameter.
Num Enter or Enter	✓ **Commit** changes.
Esc	✗ **Cancel** changes.

The XML documentation attached to a member is also updated correspondingly.

Decompose Parameter

The **Decompose Parameter** refactoring splits a single parameter into one or more parameters, depending on the function of the original parameter. The refactoring analyzes how the parameter is used and which of its properties are accessed, after which it can replace a single parameter into several others of the appropriate type for each property being accessed through the original parameter.

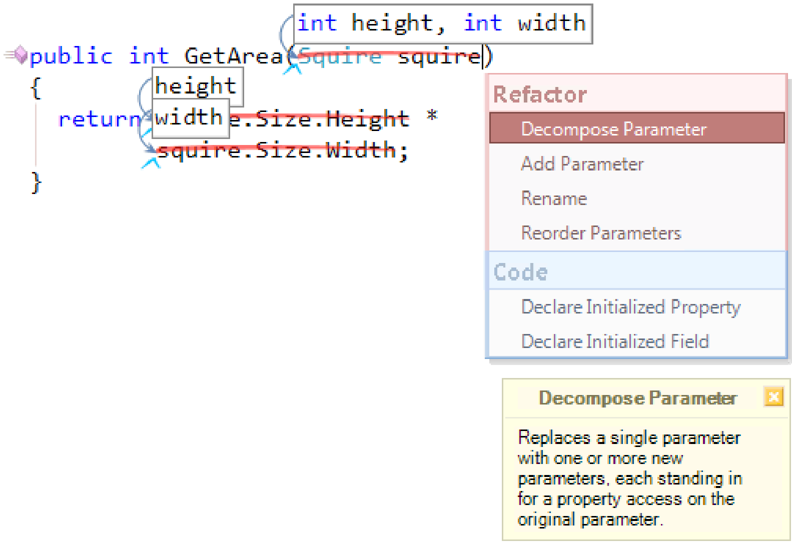
If the parameter serves as a container for other objects, it will be split into its internal objects that are used in the code. For example, this allows you to create different overloads of a method and may create the method with the widely used parameters. Consider the following class:

```
class Square
{
  public System.Drawing.Size Size { get; set; }
  public System.Drawing.Point Location { get; set; }
}
```

And a helper method that returns the area of the squire:

```
public int GetArea(Square square)
{
  return square.Size.Height * square.Size.Width;
}
```

You can split the squire parameter into two parameters – height and width:



```
public int GetArea(~~Squire squire~~)
{
    return ~~squire.Size.Height~~ *
        ~~squire.Size.Width~~;
}
```

Replaces a single parameter with one or more new parameters, each standing in for a property access on the original parameter.

and get the following method:

```
public int GetArea(int height,  int width)
{
  return height * width;
}
```

The new method allows you to pass the height and width without creating the *Square* object.

The opposite of the **Decompose Parameter** refactoring is the Introduce Parameter Object refactoring.

Introduce Parameter Object

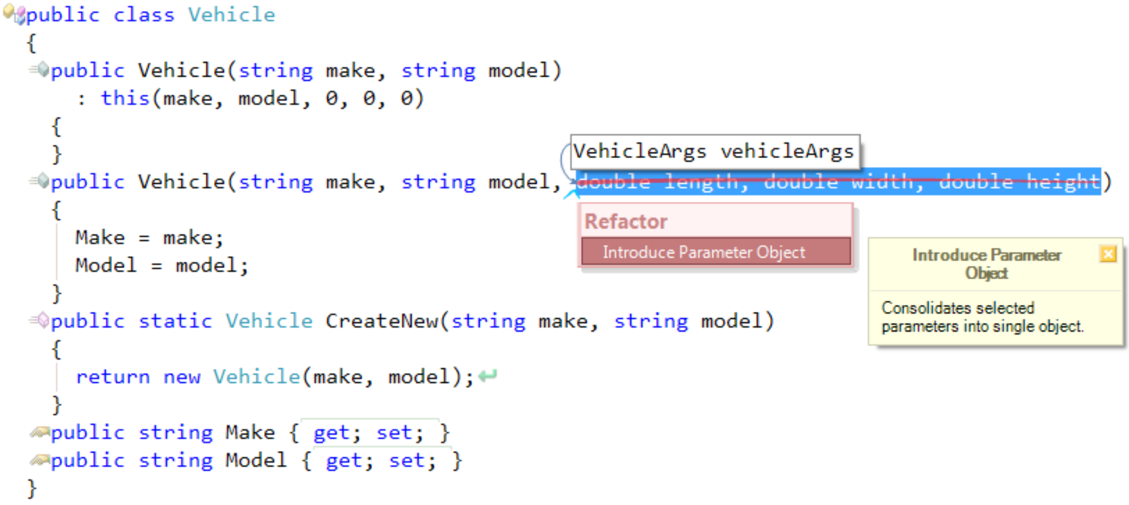
The **Introduce Parameter Object** refactoring consolidates selected parameters into single object. If you frequently need to pass similar sets of values to methods that tend to be passed together, it might be useful to encapsulate these values into an object that carries all of this data. It is worthwhile to turn these parameters into objects just to group the data together. As the result, calling statements will become more compact and you will be able to add data processing logic to the newly declared object. This refactoring is also useful because it reduces the size of the parameter lists, and long parameter lists are hard to read and understand.

The most useful benefit of this refactoring is that once you have clumped together the parameters, you'll soon see behavior that you can also move into the new object. Often the method bodies have common manipulations of the parameter values. By moving this behavior into the new object, you can remove a lot of duplicated code.

Opposite:

The opposite of this refactoring is the Decompose Parameters refactoring.

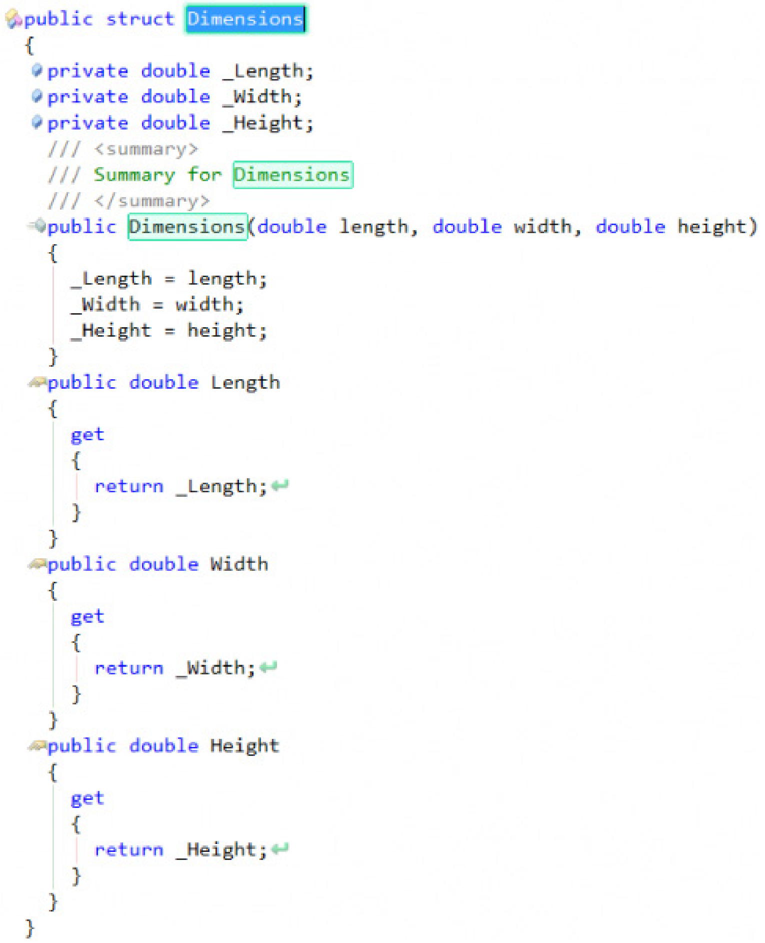
Sample:



```
public class Vehicle
{
  public Vehicle(string make, string model)
    : this(make, model, 0, 0, 0)
  {
  }
  public Vehicle(string make, string model, double length, double width, double height)
  {
    Make = make;
    Model = model;
  }
  public static Vehicle CreateNew(string make, string model)
  {
    return new Vehicle(make, model);
  }
  public string Make { get; set; }
  public string Model { get; set; }
}
```

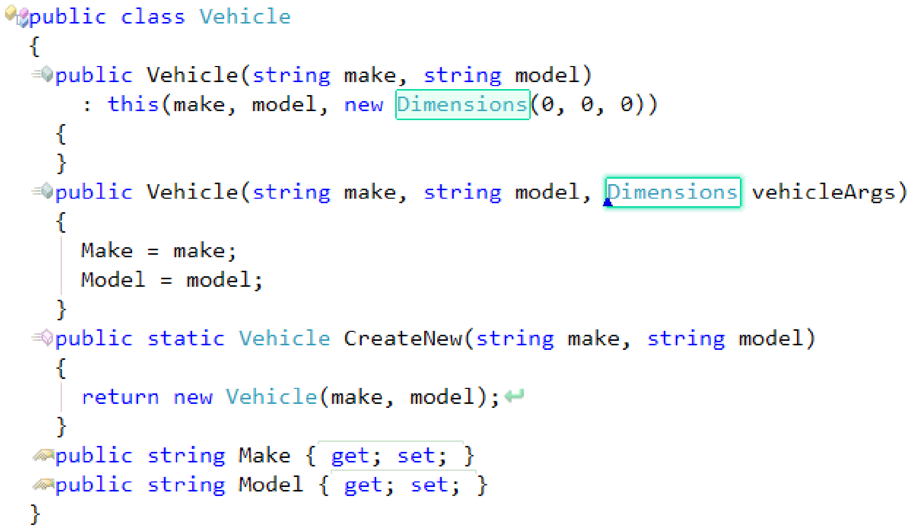
VehicleArgs vehicleArgs
Refactor
Introduce Parameter Object
Introduce Parameter Object
Consolidates selected parameters into single object.

Result object:



```
public struct Dimensions
{
  private double _Length;
  private double _Width;
  private double _Height;
  /// <summary>
  /// Summary for Dimensions
  /// </summary>
  public Dimensions(double length, double width, double height)
  {
    _Length = length;
    _Width = width;
    _Height = height;
  }
  public double Length
  {
    get
    {
      return _Length;
    }
  }
  public double Width
  {
    get
    {
      return _Width;
    }
  }
  public double Height
  {
    get
    {
      return _Height;
    }
  }
}
```

Calling side:



```
public class Vehicle
{
  public Vehicle(string make, string model)
    : this(make, model, new Dimensions(0, 0, 0))
  {
  }
  public Vehicle(string make, string model, Dimensions vehicleArgs)
  {
    Make = make;
    Model = model;
  }
  public static Vehicle CreateNew(string make, string model)
  {
    return new Vehicle(make, model);
  }
  public string Make { get; set; }
  public string Model { get; set; }
}
```

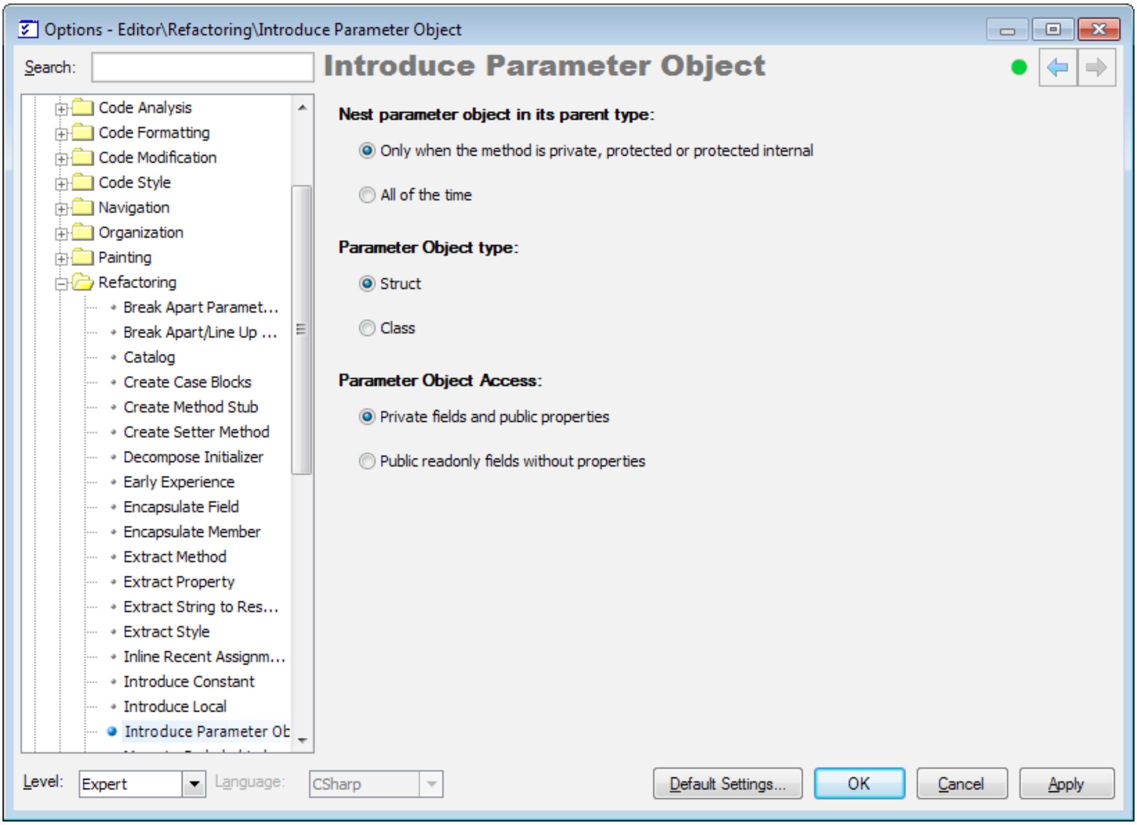
Notes:

- This refactoring automatically declares a new class or structure. The created object has a constructor that initializes all properties to values passed as parameters. All initialized values can be accessed via public read-only properties, which replace simple parameters in the original code. Note that Introduce Parameter Object uses public read-only fields instead of properties. See Options below.

- All method calls are automatically updated to pass the newly declared object instead of multiple parameters.

Options:

The **Editor** | **Refactoring** | **Introduce Parameter Object** page of the Options Dialog allows you to specify whether to declare a class or a struct, where to declare it, and whether it should have properties or fields in it.



Options - Editor\Refactoring\Introduce Parameter Object

Introduce Parameter Object

Nest parameter object in its parent type:

● Only when the method is private, protected or protected internal

○ All of the time

Parameter Object type:

● Struct

○ Class

Parameter Object Access:

● Private fields and public properties

○ Public readonly fields without properties

Level: Expert

Language: CSharp

Convert to Point

The **Convert to Point** refactoring is based on the Introduce Parameter Object refactoring with the difference that it doesn't create a new object for parameters. Instead, it uses a 'Point' structure when there is a pair of two numeric parameters of a method definition are selected.

Depending on the project type and numeric parameter types, the **Convert to Point** may use one of the following Point structures:

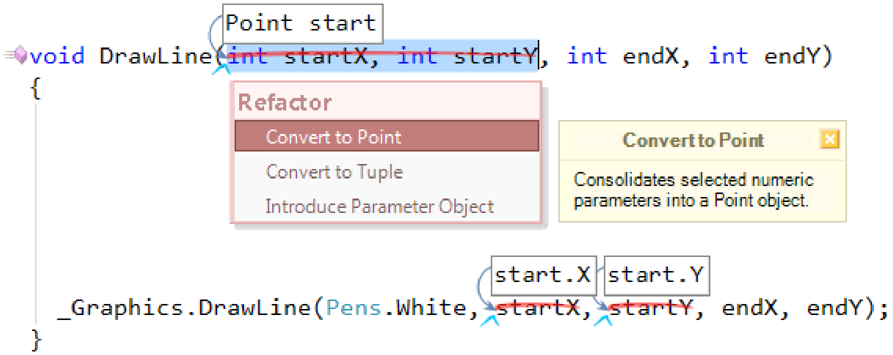
- System.Drawing.Point
- System.Drawing.PointF
- System.Windows.Point (in WPF)

Note however, if the project doesn't reference the assemblies for the Point structure (e.g. System.Drawing.dll) the refactoring won't be available.

The **Convert to Point** can be useful when you work with graphics and appropriate graphic objects, for example:

```
void DrawLine(int startX, int startY, int endX, int endY)
{
   _Graphics.DrawLine(Pens.White, startX, startY, endX, endY);
}
```

In the code preview hint you are able to see the resulting code:



```
void DrawLine(~~int startX, int startY~~, int endX, int endY)
{
   _Graphics.DrawLine(Pens.White, ~~startX~~, ~~startY~~, endX, endY);
}
```

Consolidates selected numeric parameters into a Point object.

Applying it twice will result in:

```
void DrawLine(Point start, Point end)
{
   _Graphics.DrawLine(Pens.White, start.X, start.Y, end.X, end.Y);
}
```

All calling sites will be updated automatically once the refactoring is performed.

Convert to Tuple

The **Convert to Tuple** refactoring shipped in Refactor! Pro is similar to the Introduce Parameter Object refactoring, but it doesn't create a new object – it uses the built-in .NET Framework ٤,٠ **Tuple** object.

The *Tuple* object should have at least one component, and can have a maximum of ٨ components. So, there are 8 different *Tuple* classes:

1. Tuple<T1>

2. Tuple<T1, T2>

3. Tuple<T1, T2, T3>

4. Tuple<T1, T2, T3, T4>

5. Tuple<T1, T2, T3, T4, T5>

6. Tuple<T1, T2, T3, T4, T5, T6>

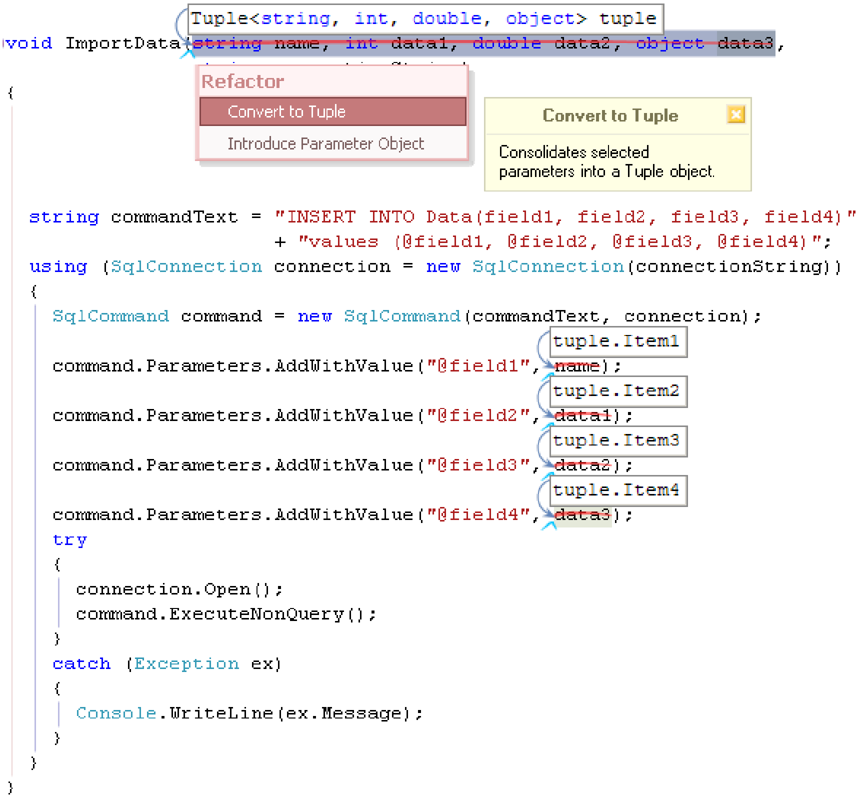
7. Tuple<T1, T2, T3, T4, T5, T6, T7>

8. Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>

You can instantiate a new *Tuple* object using any of these classes depending on the number of arguments you will pass. Also, there is the special static *Tuple.Create* method which has ٨ overloads, each of them taking … ١ ٨ arguments. The *Tuple* object will expose the *Item1 … Item8* properties, returning the values of the passed-in arguments of the appropriate type. Actually, *Tuple* supports more than ٨ arguments as it expects the ٨th argument as another *Tuple*, e.g.:

Tuple<Int32, Int32, Int32, Int32, Int32, Int32, Int32, Tuple<String, String, String>>

Here's a sample where the **Convert to Tuple** might be useful. The sample method imports some data into an SQL data base. The data consists of four objects – we'll store these objects as a single *Tuple* object.



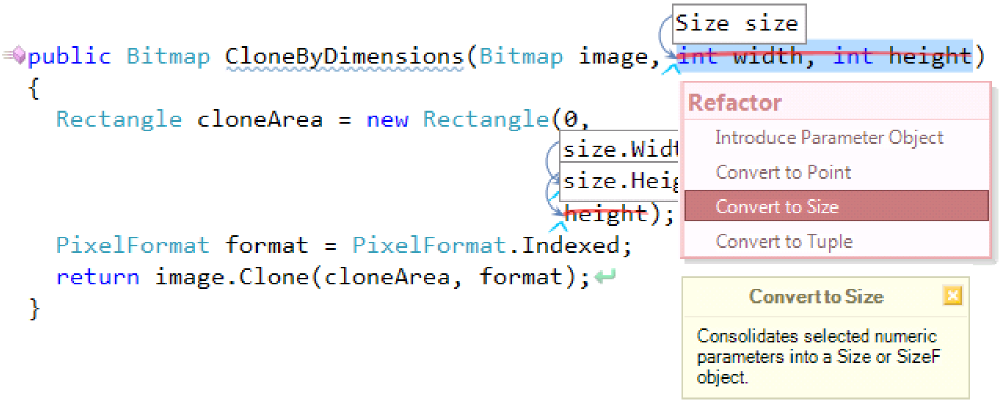
```
                  Tuple<string, int, double, object> tuple
void ImportData(string name, int data1, double data2, object data3,
{

  string commandText = "INSERT INTO Data(field1, field2, field3, field4)"
                     + "values (@field1, @field2, @field3, @field4)";
  using (SqlConnection connection = new SqlConnection(connectionString))
  {
    SqlCommand command = new SqlCommand(commandText, connection);
                                                 tuple.Item1
    command.Parameters.AddWithValue("@field1", name);
                                                 tuple.Item2
    command.Parameters.AddWithValue("@field2", data1);
                                                 tuple.Item3
    command.Parameters.AddWithValue("@field3", data2);
                                                 tuple.Item4
    command.Parameters.AddWithValue("@field4", data3);
    try
    {
      connection.Open();
      command.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Consolidates selected parameters into a Tuple object.

Convert to Size

The **Convert to Size** refactoring from DevExpress Refactor! Pro consolidates selected numeric parameters into a System.Drawing.Size (System.Windows.Size in WPF projects) or SizeF object. It might be useful to combine two parameters into a single parameter, reducing the overall number of method parameters and increasing code readability.

The refactoring is available when two parameters are called ‘width’ and ‘height’, or parameter names end with the ‘width’ and ‘height’, for example:

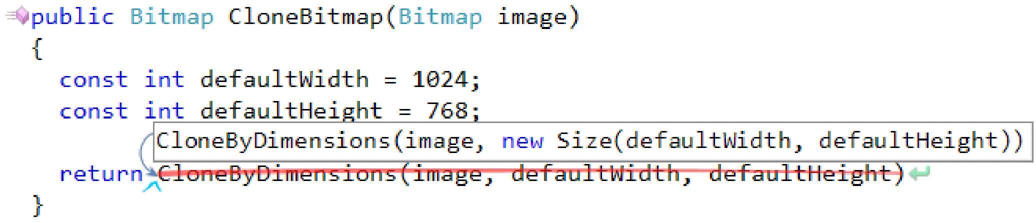


```
                                                 Size size
public Bitmap CloneByDimensions(Bitmap image, ~~int width, int height~~)
{
  Rectangle cloneArea = new Rectangle(0,
                                      size.Widt
                                      size.Heig
                                      ~~height~~);
  PixelFormat format = PixelFormat.Indexed;
  return image.Clone(cloneArea, format);
}
```

Convert to Size

Consolidates selected numeric parameters into a Size or SizeF object.

All calling sides are updated correspondingly when the refactoring is performed:



```
public Bitmap CloneBitmap(Bitmap image)
{
  const int defaultWidth = 1024;
  const int defaultHeight = 768;
         CloneByDimensions(image, new Size(defaultWidth, defaultHeight))
  return ~~CloneByDimensions(image, defaultWidth, defaultHeight)~~
}
```

The opposite of the **Convert to Size** refactoring is the Decompose Parameter refactoring.

Converting between static and instance members

Classes allow you to create instance members and static members. Instance members are available when an instance of the class is created and have access to the object’s data. Static members do not require an object created and can be called directly.

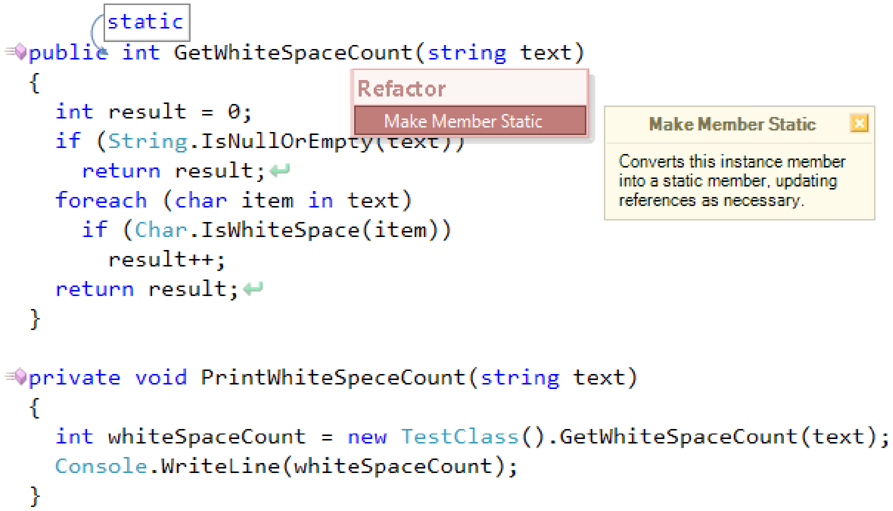
Static methods are useful when creating functions that are not reliant on any instance of a class. An example of the extensive use of static members is the *System.Math* class, which is a library of mathematical functions and constants provided by the .NET framework.

If you declare instance fields, they are created and initialized when an instance of the class is created. Static fields apply only to the class itself. No matter how many instances of a class there are, there is only one instance of that classes’ static fields. In contrast, instance fields may have different instances and values for each class instance created. Theoretically, static members perform faster than instance members, because there is no need to create an instance of a class. However, the penalty for using instance methods is minor and should only be noticeable when calling billions or trillions of members. If the software that you are developing is performance critical, it may be worthwhile to convert some instance members to static members. But bear in mind, that may decrease the readability and maintainability of the code.

To help you easily convert static members into instance members and vice versa, Refactor! Pro provides two refactorings:

- **Make Member Static**
- **Make Member Non-static**

The first one allows you to convert an instance member that can be made static into a static one. An instance member should not reference any other instance members to be static. The refactoring not only adds a static keyword ('Shared' in VB) but also updates all references accordingly:

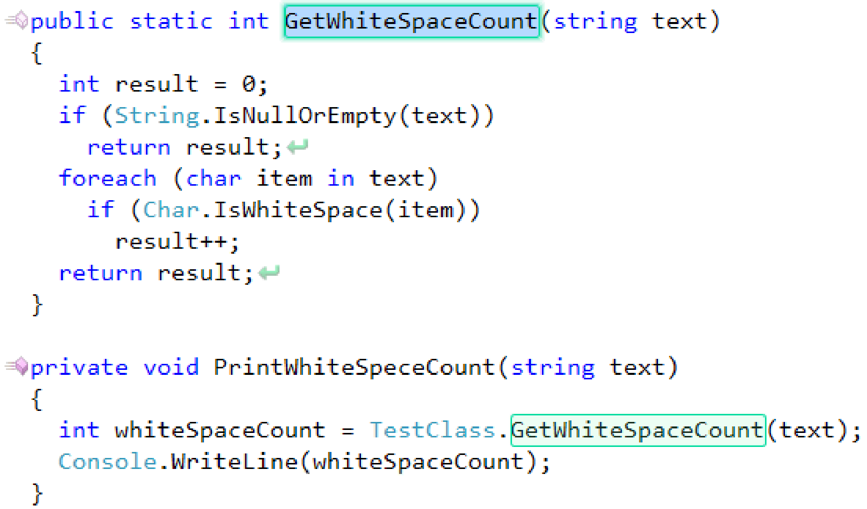


Make Member Static

Converts this instance member into a static member, updating references as necessary.

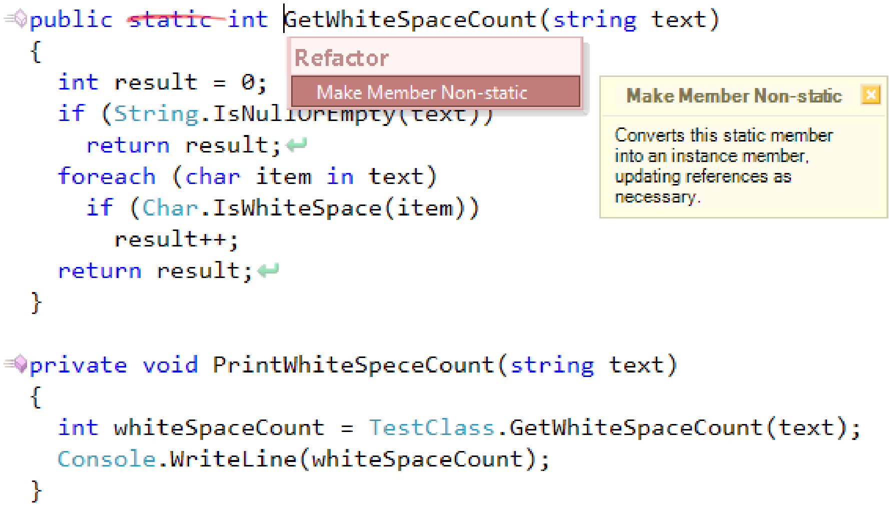
```
public int GetWhiteSpaceCount(string text)
{
  int result = 0;
  if (String.IsNullOrEmpty(text))
    return result;
  foreach (char item in text)
    if (Char.IsWhiteSpace(item))
      result++;
  return result;
}

private void PrintWhiteSpeceCount(string text)
{
  int whiteSpaceCount = new TestClass().GetWhiteSpaceCount(text);
  Console.WriteLine(whiteSpaceCount);
}
```

Applying the refactoring will result in the following code:



```
public static int GetWhiteSpaceCount(string text)
{
  int result = 0;
  if (String.IsNullOrEmpty(text))
    return result;
  foreach (char item in text)
    if (Char.IsWhiteSpace(item))
      result++;
  return result;
}

private void PrintWhiteSpeceCount(string text)
{
  int whiteSpaceCount = TestClass.GetWhiteSpaceCount(text);
  Console.WriteLine(whiteSpaceCount);
}
```

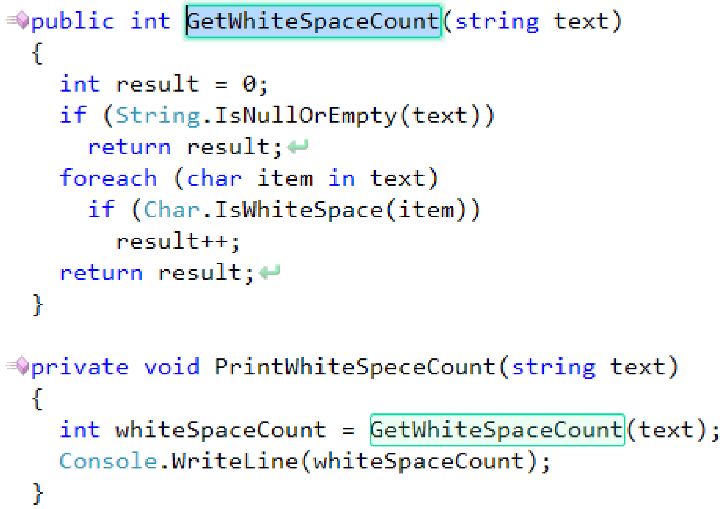
The second refactoring is the opposite of the first one. If you have a static method – it can change it to a regular instance member, and replace its references with a new class instance creation and with an appropriate call to the source member:



```
public ~~static~~ int GetWhiteSpaceCount(string text)
{
  int result = 0;
  if (String.IsNullOrEmpty(text))
    return result;
  foreach (char item in text)
    if (Char.IsWhiteSpace(item))
      result++;
  return result;
}

private void PrintWhiteSpeceCount(string text)
{
  int whiteSpaceCount = TestClass.GetWhiteSpaceCount(text);
  Console.WriteLine(whiteSpaceCount);
}
```

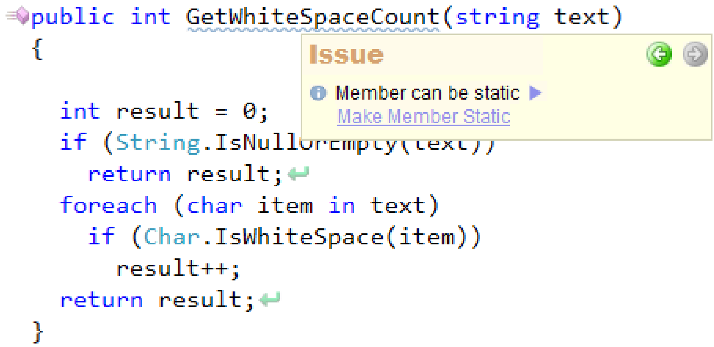
Converts this static member into an instance member, updating references as necessary.

Applying the refactoring will result in the following code:



```
public int GetWhiteSpaceCount(string text)
{
  int result = 0;
  if (String.IsNullOrEmpty(text))
    return result;
  foreach (char item in text)
    if (Char.IsWhiteSpace(item))
      result++;
  return result;
}

private void PrintWhiteSpeceCount(string text)
{
  int whiteSpaceCount = GetWhiteSpaceCount(text);
  Console.WriteLine(whiteSpaceCount);
}
```

Both refactorings are available for methods, properties and field members of a class. You can see the '**Member can be static**' (or '**Member can be shared**' in VB) code issue if you have CodeRush Pro installed and the Code Issues feature is turned on:



```
public int GetWhiteSpaceCount(string text)
{
  int result = 0;
  if (String.IsNullOrEmpty(text))
    return result;
  foreach (char item in text)
    if (Char.IsWhiteSpace(item))
      result++;
  return result;
}
```

Issue

Member can be static

Make Member Static

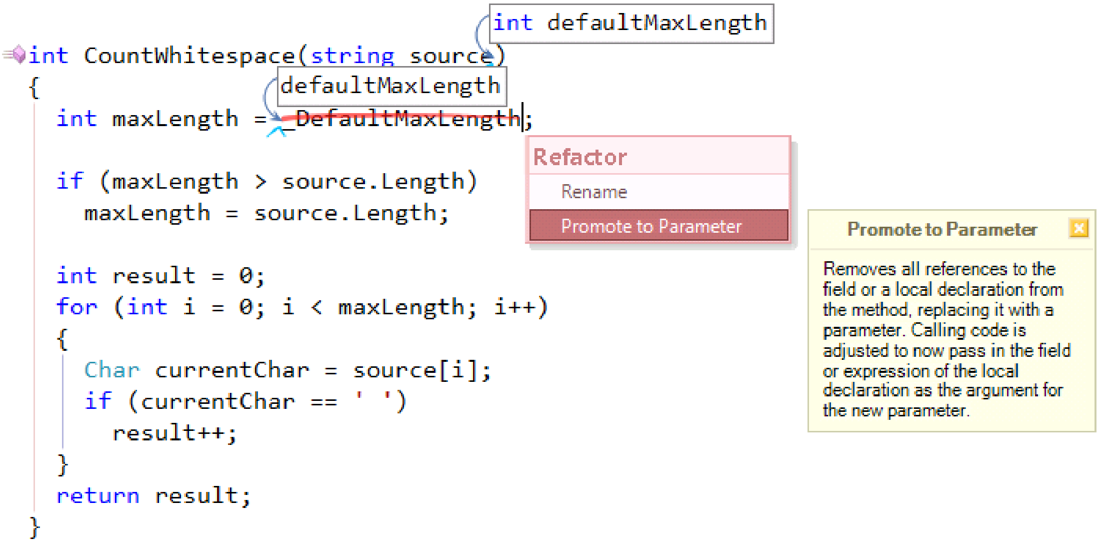
Promoting locals and constant value expressions into parameters

There are times when you realize that a local variable or a field reference within a method would be more useful if it was a parameter. Having a new parameter on a method will increase its flexibility for consumers. To convert the local variable, you should remove its declaration from the body of the method, add it as a parameter and replace all occurrences of a local to a new parameter. The same steps must be performed in the case of field references. Furthermore, all method references in the entire solution should be updated to pass a new value as an argument to a method with a new signature.

With the help of the **Promote to Parameter** refactoring, you can promote local variables, field references and any primitive values into parameters and make the required updates to the code automatically.

Choose a local variable, a field reference or a constant value and apply the refactoring to promote it into a parameter:

A field reference:

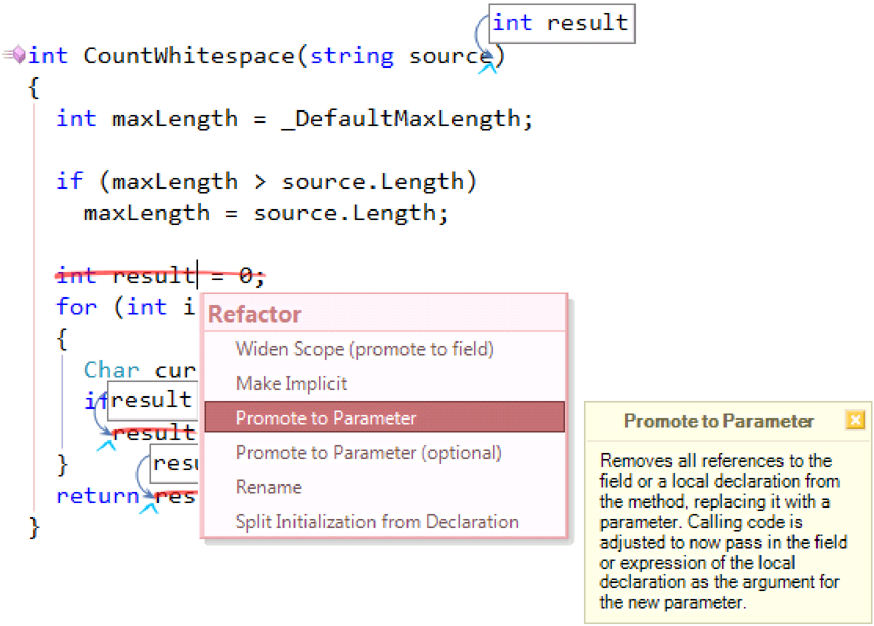


```
int CountWhitespace(string source)
{
  int maxLength = _DefaultMaxLength;

  if (maxLength > source.Length)
    maxLength = source.Length;

  int result = 0;
  for (int i = 0; i < maxLength; i++)
  {
    Char currentChar = source[i];
    if (currentChar == ' ')
      result++;
  }
  return result;
}
```

Promote to Parameter

Removes all references to the field or a local declaration from the method, replacing it with a parameter. Calling code is adjusted to now pass in the field or expression of the local declaration as the argument for the new parameter.

A local variable:



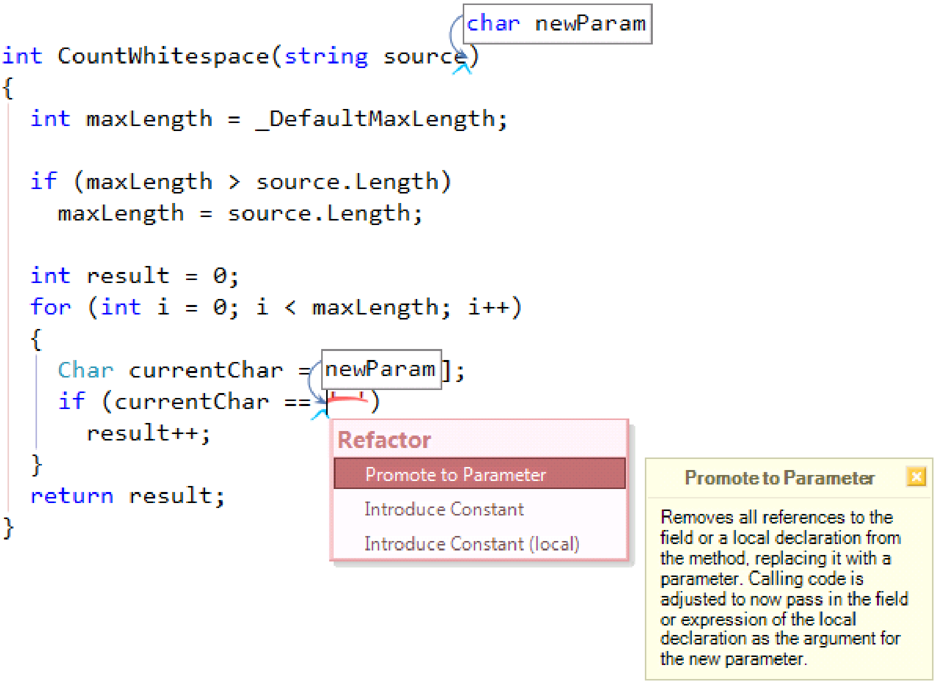
```
                                       int result
int CountWhitespace(string source)
{
  int maxLength = _DefaultMaxLength;

  if (maxLength > source.Length)
    maxLength = source.Length;

  ~~int result = 0;~~
  for (int i
  {
    Char cur
    ifresult
     result
  }  resu
  return res
}
```

Refactor
Widen Scope (promote to field)
Make Implicit
Promote to Parameter
Promote to Parameter (optional)
Rename
Split Initialization from Declaration

Promote to Parameter

Removes all references to the field or a local declaration from the method, replacing it with a parameter. Calling code is adjusted to now pass in the field or expression of the local declaration as the argument for the new parameter.

A constant value expression:



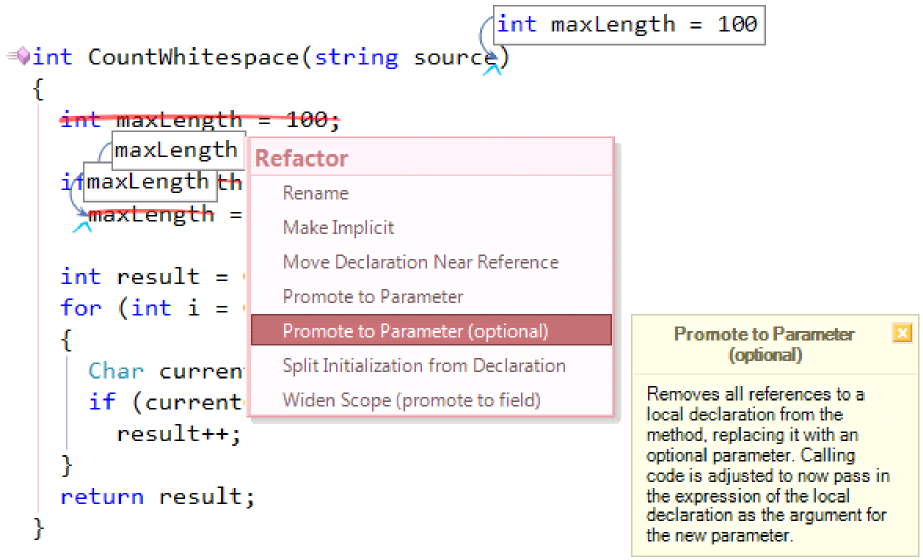
```
                                  char newParam
int CountWhitespace(string source)
{
  int maxLength = _DefaultMaxLength;

  if (maxLength > source.Length)
    maxLength = source.Length;

  int result = 0;
  for (int i = 0; i < maxLength; i++)
  {
    Char currentChar = newParam];
    if (currentChar == ' ')
      result++;
  }
  return result;
}
```

Promote to Parameter

Removes all references to the field or a local declaration from the method, replacing it with a parameter. Calling code is adjusted to now pass in the field or expression of the local declaration as the argument for the new parameter.

In every case, a new parameter is added and all calls to the method are updated as required. You can immediately rename a new parameter according to your preference.

A second version of the refactoring called **Promote to Parameter (optional)** behaves in absolutely the same manner as its original version. The difference between two refactorings is that the latter creates an optional parameter instead of the usual parameter:



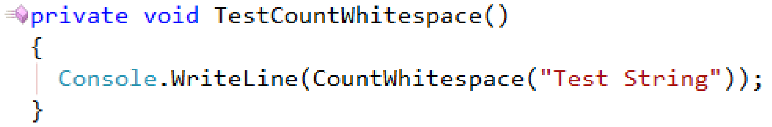
```
                                               int maxLength = 100
int CountWhitespace(string source)
{
  ~~int maxLength = 100;~~
     maxLength
  ifmaxLength th
    ~~maxLength~~ =

  int result =
  for (int i =
  {
    Char curren
    if (current
      result++;
  }
  return result;
}
```

Refactor
Rename
Make Implicit
Move Declaration Near Reference
Promote to Parameter
Promote to Parameter (optional)
Split Initialization from Declaration
Widen Scope (promote to field)

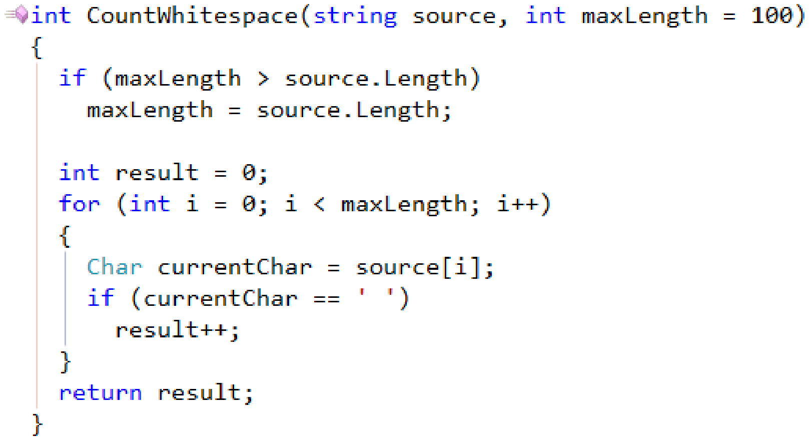
Promote to Parameter (optional)

Removes all references to a local declaration from the method, replacing it with an optional parameter. Calling code is adjusted to now pass in the expression of the local declaration as the argument for the new parameter.

Optional parameters provide default values within the member's signature. So, if an argument is omitted from a call to that method, the default value will be used instead:



```
private void TestCountWhitespace()
{
  Console.WriteLine(CountWhitespace("Test String"));
}
```



```
int CountWhitespace(string source, int maxLength = 100)
{
  if (maxLength > source.Length)
    maxLength = source.Length;

  int result = 0;
  for (int i = 0; i < maxLength; i++)
  {
    Char currentChar = source[i];
    if (currentChar == ' ')
      result++;
  }
  return result;
}
```

If we compare the **Promote to Parameter** refactoring to the **Promote Local Variable to Parameter** refactoring built-in to Visual Studio (VS refactoring), we see the following advantages of the refactoring shipped in CodeRush:

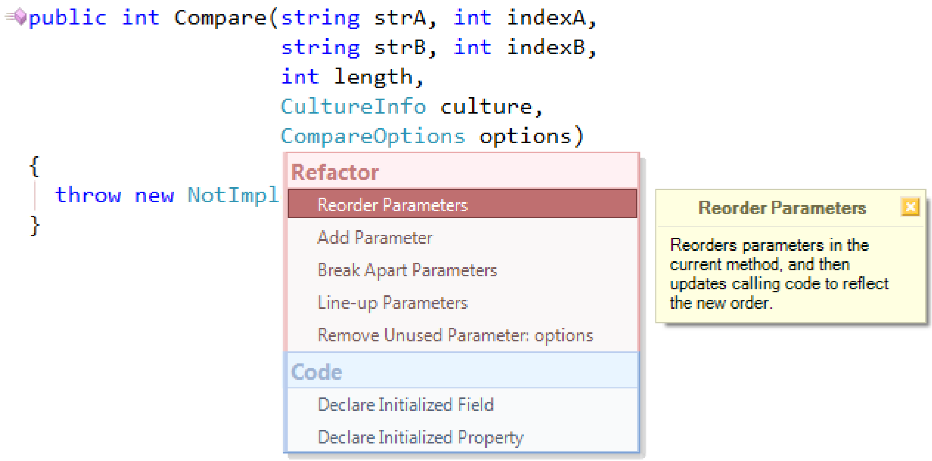
- The VS refactoring does not allow you to create optional parameters

- The VS refactoring can not be applied on the field reference within a method body
- The VS refactoring can not be applied on constant value expressions within a method body.

Reordering method parameters

Sometimes you may find that various method parameters are ordered illogically, or when method parameters appear in a different order relative to one another, similar methods. In this case, if you prefer the parameters appear in a consistent and logical order, you must not only change the order in the method signature but also update all calls to the method. Take into account complex cases when you can move return values of a function into our parameters and vice versa. Refactoring such complex cases is a rather complex and time-consuming task, especially if there are too many calls.

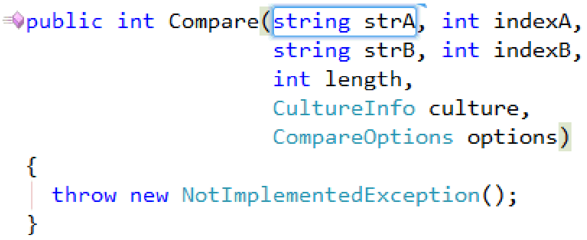
The **Reorder Parameters** refactoring from CodeRush can perform this task automatically and make reordering parameters fun:



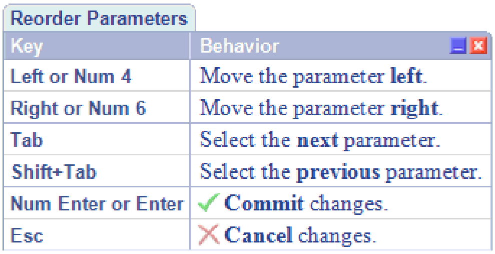
```
public int Compare(string strA, int indexA,
                   string strB, int indexB,
                   int length,
                   CultureInfo culture,
                   CompareOptions options)
{
    throw new NotImpl
}
```

Reorders parameters in the current method, and then updates calling code to reflect the new order.

It has a great animated interactive phase where you choose parameter positions that you've never seen before (unless you've used this CodeRush refactoring earlier). Here is it in action:



```
public int Compare(string strA, int indexA,
                   string strB, int indexB,
                   int length,
                   CultureInfo culture,
                   CompareOptions options)
{
    throw new NotImplementedException();
}
```

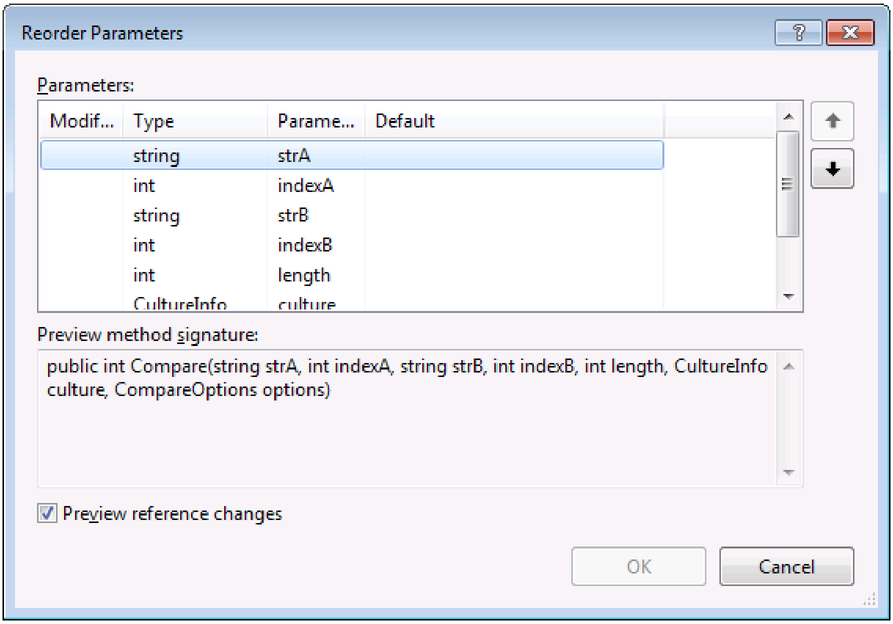


Reorder Parameters

Key	Behavior
Left or Num 4	Move the parameter **left**.
Right or Num 6	Move the parameter **right**.
Tab	Select the **next** parameter.
Shift+Tab	Select the **previous** parameter.
Num Enter or Enter	✓ **Commit** changes.
Esc	✗ **Cancel** changes.

Once the refactoring is applied, all method calls are modified automatically in the entire solution along with method overloads, if any.

Of course, you can use the built-in Visual Studio refactoring version. The built-in refactoring command provides a dialog window that allows you to change the order of the parameters for a method as follows:



Reorder Parameters

Parameters:

Modif...	Type	Parame...	Default
	string	strA	
	int	indexA	
	string	strB	
	int	indexB	
	int	length	
	CultureInfo	culture	

Preview method signature:

public int Compare(string strA, int indexA, string strB, int indexB, int length, CultureInfo culture, CompareOptions options)

☑ Preview reference changes

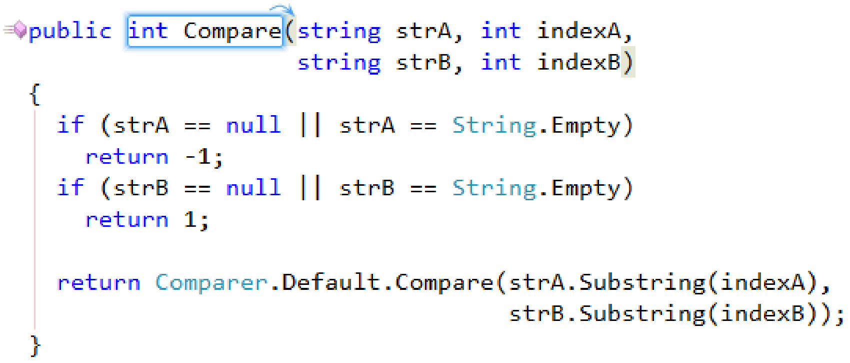
The dialog window shows the method's parameters. To change the order of parameters, click the item that you wish to move in the list. Then, click the up or down arrow buttons to move the selected item. A preview method signature is displayed as you make changes. Once you are done with the new signature, click the OK button to perform the updates.

All in all, it's up to you to decide which version of the refactoring – CodeRush or Visual Studio, is more convenient. However, if we compare the **Reorder Parameters** refactoring shipped in CodeRush and the **Reorder Parameters** refactoring built-in to Visual Studio (VS refactoring), we see the following advantages in the refactoring shipped in CodeRush:

- The VS refactoring does not allow you to move out parameters to a return value, but CodeRush refactoring does;

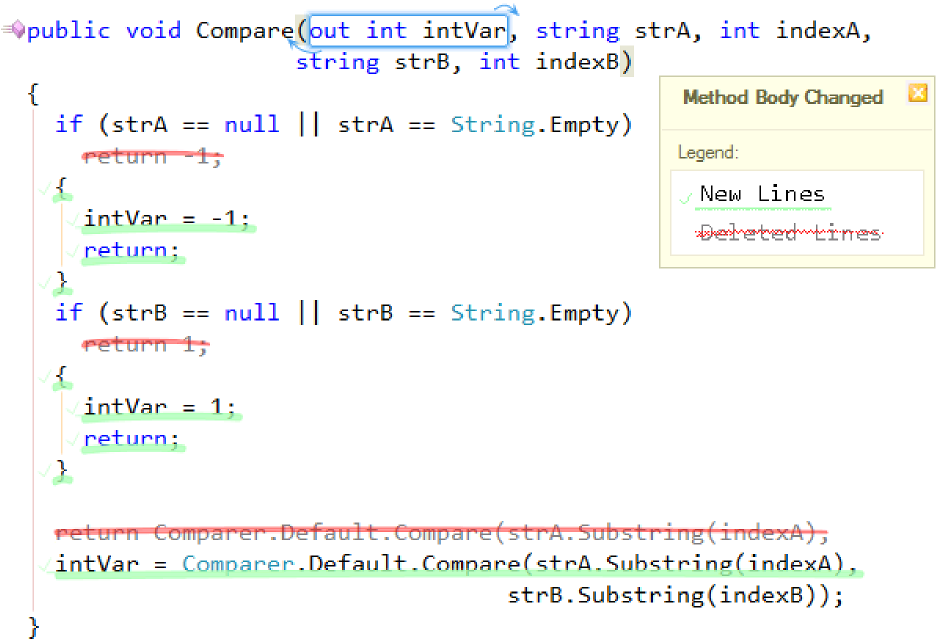
- The VS refactoring does not allow you to convert the return value to an out parameter, but CodeRush refactoring does;

- The CodeRush version of the refactoring provides an advanced preview hint of changes that are being applied to the resulting method declaration, for instance, if you are moving the return value to an out parameter for this method:



```
public int Compare(string strA, int indexA,
                   string strB, int indexB)
{
  if (strA == null || strA == String.Empty)
    return -1;
  if (strB == null || strB == String.Empty)
    return 1;

  return Comparer.Default.Compare(strA.Substring(indexA),
                                  strB.Substring(indexB));
}
```

The refactoring must change all return statements and assign the returned value to the new out parameter as follows:



```
public void Compare(out int intVar, string strA, int indexA,
                   string strB, int indexB)
{
  if (strA == null || strA == String.Empty)
    ~~return -1;~~
  {
    intVar = -1;
    return;
  }
  if (strB == null || strB == String.Empty)
    ~~return 1;~~
  {
    intVar = 1;
    return;
  }

  ~~return Comparer.Default.Compare(strA.Substring(indexA),~~
  intVar = Comparer.Default.Compare(strA.Substring(indexA),
                                  strB.Substring(indexB));
}
```

The CodeRush refactoring has several options available on the Editor | Refactoring | Reorder Parameters option page in the Options Dialog. You can tweak the following options for this refactoring:

- The 'Animate parameter reordering' option specifies whether or not an interactive animation is enabled.
- The 'Use hint for the code changes' option specifies whether or not additional preview hints are enabled.
- The 'Leave changed code as comments' option specifies whether or not modified code changes should be left in comments for a future code review.

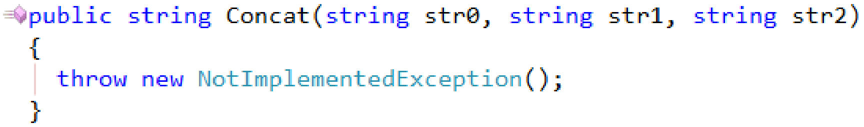
Safe rename a public API member and mark it obsolete

There are times when public API members must be deprecated or its signature must be changed. In this case, we can not remove or modify public members, because other application that depend on these APIs may become broken and/or uncompilable. If you'd like to supersede a public member, you can use a special attribute called 'Obsolete'. The Obsolete attribute marks an API member as one that is no longer recommended for use.

The Obsolete attribute is useful for refactoring and general management of your project's source code over the development life cycle. Each use of a member marked obsolete will subsequently generate a warning or an error, depending on how the attribute is configured.

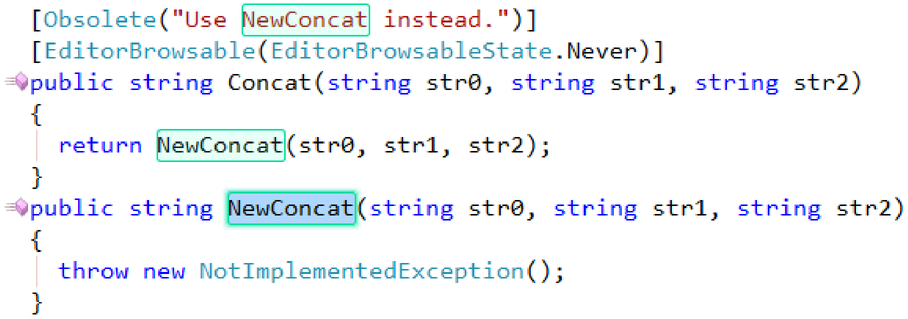
To create a new public member based on the old one, you can utilize the **Safe Rename** refactoring. The refactoring creates a copy of a non-private method or property in the class in a manner that will not break any dependent code, such as code in third-party applications that use that member.

Consider the following public member:



```
public string Concat(string str0, string str1, string str2)
{
  throw new NotImplementedException();
}
```

Since you know that you will want to deprecate this code, marking the method with the Obsolete attribute will be a constant reminder for you and code consumers that its use is not recommended. The displayed message can also explain the reason why the member is obsolete as well as provide an alternative. The refactoring adds the corresponding message for you once you apply it:



```
[Obsolete("Use NewConcat instead.")]
[EditorBrowsable(EditorBrowsableState.Never)]
public string Concat(string str0, string str1, string str2)
{
  return NewConcat(str0, str1, str2);
}
public string NewConcat(string str0, string str1, string str2)
{
  throw new NotImplementedException();
}
```

It also adds the EditorBrowsable attribute which specifies the browsable state of a property or method from within the code editor.

By providing a permanent reminder about unrecommended code, the refactoring helps in versioning software programs and coordinating the methods employed by different developers. This technique plays a vital role in the design of software that gets updated with newer versions and needs to be used by other consumers without breaking the existing applications. That is why it allows you to maintain the software's backward compatibility and code maintainability throughout the development life cycle.

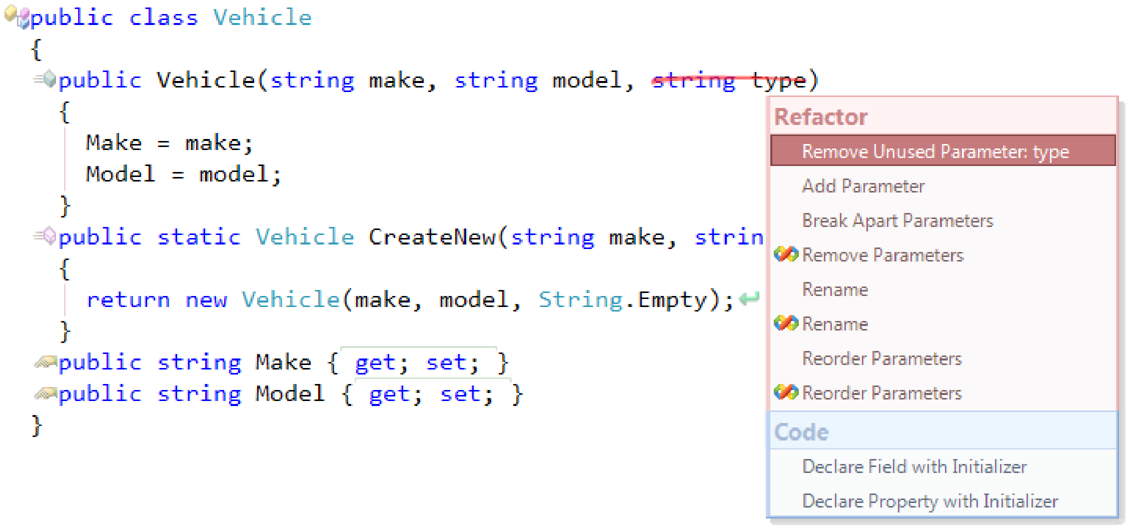
Remove Parameter

Also known as **Remove Unused Parameter**. This refactoring removes an unused parameter from a method declarration, and updates all calls accordingly. It is very useful when a parameter is no longer used by the method body. A spurious parameter doesn't cause any problems, and you probably might need it again later. But most of the time this is the wrong choice, because a parameter indicates information that is needed. In this case, a caller has to worry about what values to pass for a fictitious, unused parameter. By not removing the parameter you are making further work for everyone who uses the method.

Opposite:

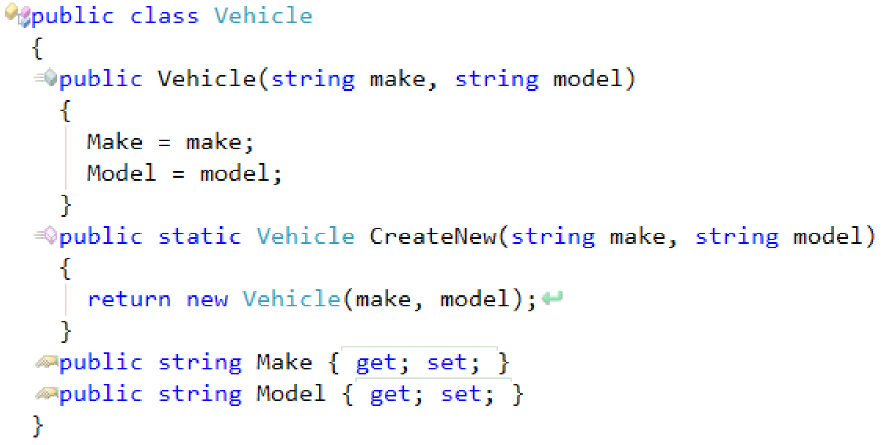
The opposite of this refactoring is the Add Parameter refactoring.

Sample:



```
public class Vehicle
{
  public Vehicle(string make, string model, ~~string~~ type)
  {
    Make = make;
    Model = model;
  }
  public static Vehicle CreateNew(string make, strin
  {
    return new Vehicle(make, model, String.Empty);
  }
  public string Make { get; set; }
  public string Model { get; set; }
}
```

Result:



```
public class Vehicle
{
  public Vehicle(string make, string model)
  {
    Make = make;
    Model = model;
  }
  public static Vehicle CreateNew(string make, string model)
  {
    return new Vehicle(make, model);
  }
  public string Make { get; set; }
  public string Model { get; set; }
}
```

Refactoring multi-conditional statements: the switch statement and the “if-else-if” ladder

There are two methods for multi-conditional processing: the “if-else-if” ladder and the switch statement. The If-Else-If ladder is a combination of ‘if’ and ‘else’ statements that is used to test a series of conditions. If the condition of the first ‘if’ statement is met, the code within the ‘if’ executes. Otherwise, the program flow is passed to the neighborhood ‘else’ statement, in which the next ‘if’ statement is located. This continues as a series of ‘if’ statements nested within the previous ‘else’ statements until all conditions have been checked. A default branch or a code block may be executed in a final ‘else’ statement if no condition is previously met.

The switch statement is similar to the ‘if-else-if’ ladder as it provides multiple branching. In the case of the switch statement, the value of a variable or expression is tested against a series of different values (cases), until a match is found. The code block within the matched case statement is executed. If none of the cases match, then the optional default case is executed.

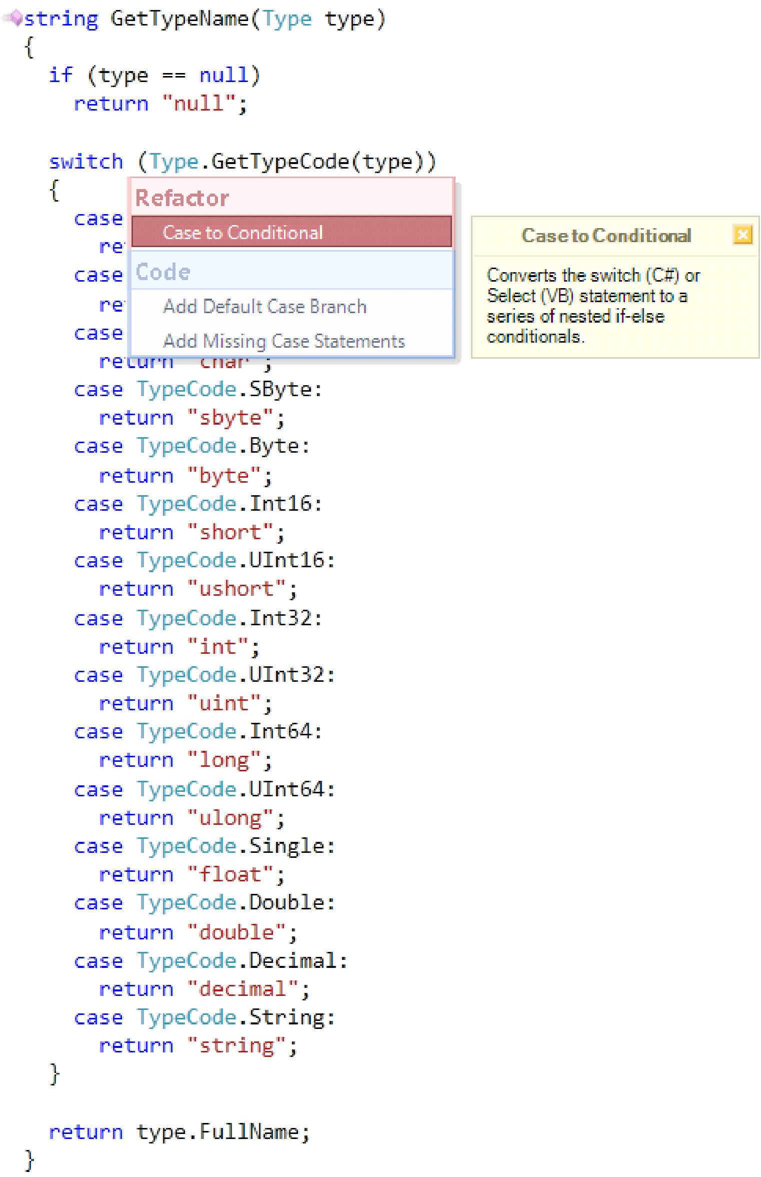
We can find the following differences between these two multi-condition evaluation approaches:

- the switch statement is generally considered easier to read;
- the switch statement is less flexible than the if-else-if ladder, because it only permits testing of a single expression against a list of discrete values;
- the switch statement is generally considered to be more efficient, because the compiler is able to optimize the switch statement. The compiler may re-order the testing to provide the fastest execution, because each case within a switch statement does not rely on previous cases. In the case of the if-else-if ladder, the code must process each if statement in the order determined by the developer.

All in all, it is up to you to choose the option that makes the code more readable and maintainable and which approach to use: the “if-else-if” ladder or the switch statement. To help you easily convert one approach to another and vice versa CodeRush bundled with Refactor! Pro contains two refactorings:

- **Conditional to Case**
- **Case to Conditional**

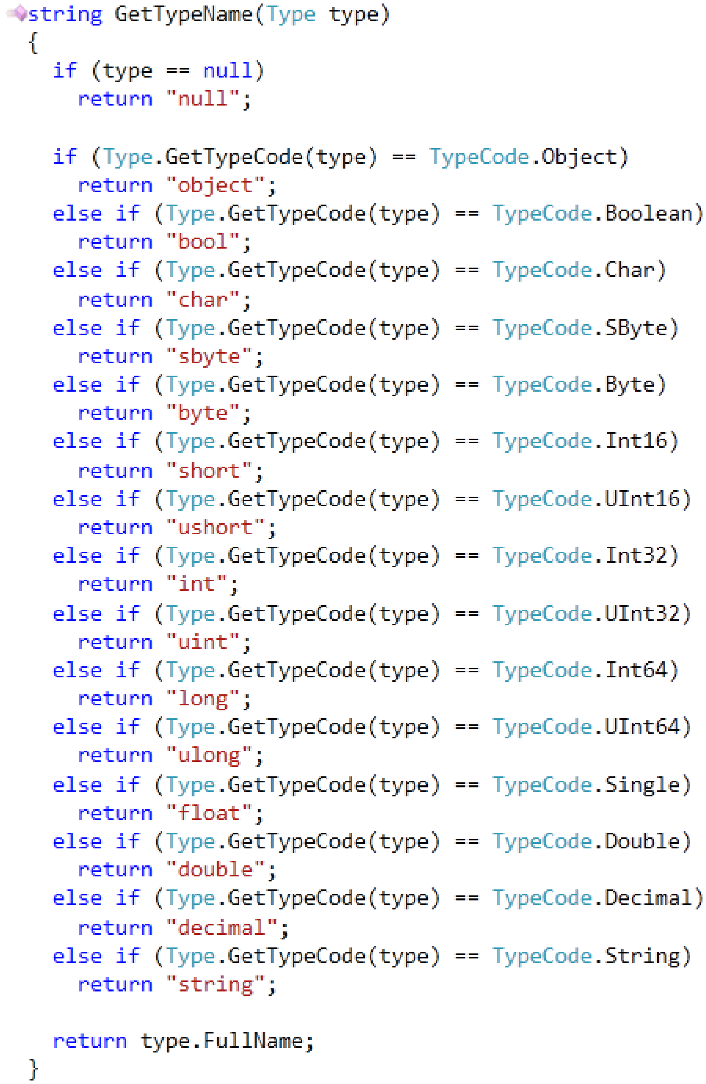
The **Case to Conditional** refactoring converts the switch (C#, C++) or Select (VB) statement into a combination of if/else statements, e.g.:



Converts the switch (C#) or Select (VB) statement to a series of nested if-else conditionals.

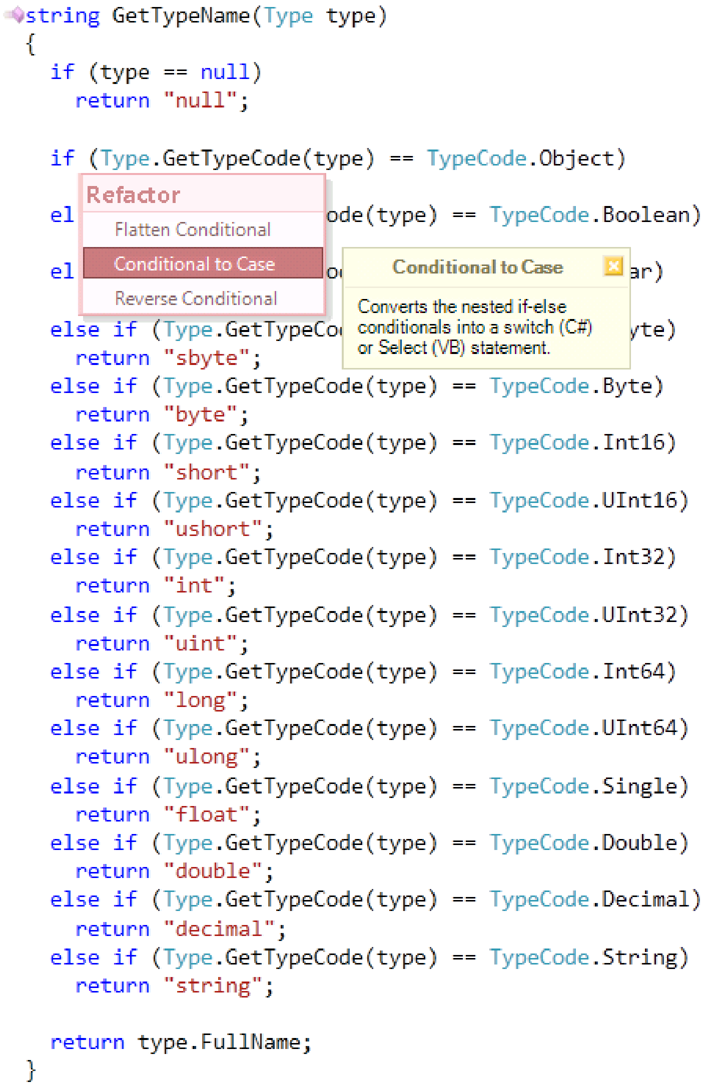
```
string GetTypeName(Type type)
{
  if (type == null)
    return "null";

  switch (Type.GetTypeCode(type))
  {
    case
      re
    case
      re
    case
      return "char";
    case TypeCode.SByte:
      return "sbyte";
    case TypeCode.Byte:
      return "byte";
    case TypeCode.Int16:
      return "short";
    case TypeCode.UInt16:
      return "ushort";
    case TypeCode.Int32:
      return "int";
    case TypeCode.UInt32:
      return "uint";
    case TypeCode.Int64:
      return "long";
    case TypeCode.UInt64:
      return "ulong";
    case TypeCode.Single:
      return "float";
    case TypeCode.Double:
      return "double";
    case TypeCode.Decimal:
      return "decimal";
    case TypeCode.String:
      return "string";
  }

  return type.FullName;
}
```

Result:



```
string GetTypeName(Type type)
{
  if (type == null)
    return "null";

  if (Type.GetTypeCode(type) == TypeCode.Object)
    return "object";
  else if (Type.GetTypeCode(type) == TypeCode.Boolean)
    return "bool";
  else if (Type.GetTypeCode(type) == TypeCode.Char)
    return "char";
  else if (Type.GetTypeCode(type) == TypeCode.SByte)
    return "sbyte";
  else if (Type.GetTypeCode(type) == TypeCode.Byte)
    return "byte";
  else if (Type.GetTypeCode(type) == TypeCode.Int16)
    return "short";
  else if (Type.GetTypeCode(type) == TypeCode.UInt16)
    return "ushort";
  else if (Type.GetTypeCode(type) == TypeCode.Int32)
    return "int";
  else if (Type.GetTypeCode(type) == TypeCode.UInt32)
    return "uint";
  else if (Type.GetTypeCode(type) == TypeCode.Int64)
    return "long";
  else if (Type.GetTypeCode(type) == TypeCode.UInt64)
    return "ulong";
  else if (Type.GetTypeCode(type) == TypeCode.Single)
    return "float";
  else if (Type.GetTypeCode(type) == TypeCode.Double)
    return "double";
  else if (Type.GetTypeCode(type) == TypeCode.Decimal)
    return "decimal";
  else if (Type.GetTypeCode(type) == TypeCode.String)
    return "string";

  return type.FullName;
}
```

The **Conditional to Case** refactoring is the opposite of the **Case to Conditional** refactorings which converts a set of if/else statements into a switch/Select statement:



```
string GetTypeName(Type type)
{
  if (type == null)
    return "null";

  if (Type.GetTypeCode(type) == TypeCode.Object)

  el                     ode(type) == TypeCode.Boolean)

  el                     o                        ar)

  else if (Type.GetTypeCo                         yte)
    return "sbyte";
  else if (Type.GetTypeCode(type) == TypeCode.Byte)
    return "byte";
  else if (Type.GetTypeCode(type) == TypeCode.Int16)
    return "short";
  else if (Type.GetTypeCode(type) == TypeCode.UInt16)
    return "ushort";
  else if (Type.GetTypeCode(type) == TypeCode.Int32)
    return "int";
  else if (Type.GetTypeCode(type) == TypeCode.UInt32)
    return "uint";
  else if (Type.GetTypeCode(type) == TypeCode.Int64)
    return "long";
  else if (Type.GetTypeCode(type) == TypeCode.UInt64)
    return "ulong";
  else if (Type.GetTypeCode(type) == TypeCode.Single)
    return "float";
  else if (Type.GetTypeCode(type) == TypeCode.Double)
    return "double";
  else if (Type.GetTypeCode(type) == TypeCode.Decimal)
    return "decimal";
  else if (Type.GetTypeCode(type) == TypeCode.String)
    return "string";

  return type.FullName;
}
```

Converts the nested if-else conditionals into a switch (C#) or Select (VB) statement.

Both refactorings are available in C#, Visual Basic and C++ languages.

Refactoring ternary expressions into null-coalescing operations and vice versa

There are two interesting refactorings shipped in Refactor! Pro:

- **Compress to Null Coalescing Operation**
- **Expand Null Coalescing Operation**

The first one converts a ternary expression into an equivalent null coalescing operation. The second one is the opposite of the first one – it converts a null coalescing operation to an equivalent ternary expression. Both refactorings are available in *CSharp* only, because *Visual Basic*, for example, doesn't have a null coalescing operator.

Here’s an excerpt from the C# language specification about the null coalescing operator for reference:

- *A null coalescing expression of the form a ?? b requires a to be of a nullable type or reference type. If a is non-null, the result of a ?? b is a; otherwise, the result is b. The operation evaluates b only if a is null.*

- *The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form a ?? b ?? c is evaluated as a ?? (b ?? c). In general terms, an expression of the form E1 ?? E2 ?? ... ?? EN returns the first of the operands that is non-null, or null if all operands are null.*

So, the null coalescing operator provides an effective, compact way to check whether a value is null, and if so, return an alternative value. For example:

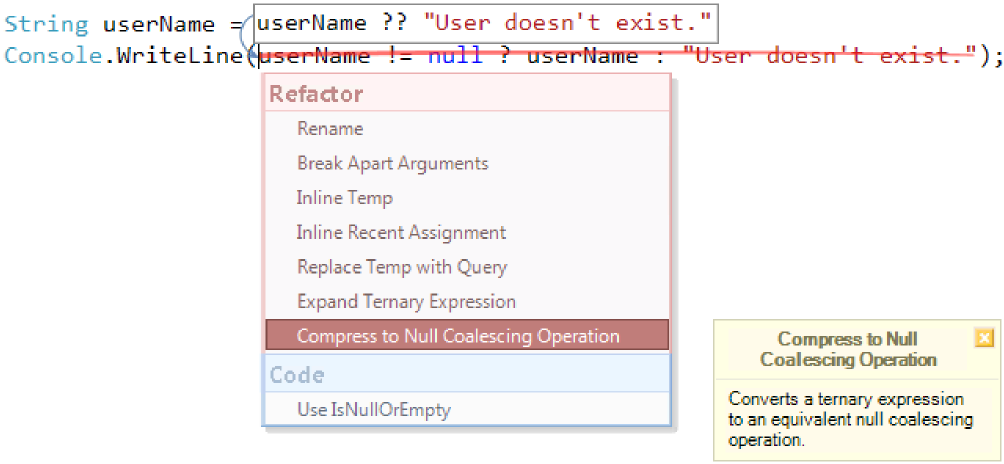
```
Brush myBrush = GetCachedBrush() ?? new SolidBrush(Color.Black);
```

In the code above, if the return value of *GetCachedBrush*() is not null, it is assigned to the myBrush variable. Otherwise, if it is null, a new *SolidBrush* is created and assigned.

The ?? operator is also useful for output. For example, instead of using the ternary operator,

```
String userName = GetUserName();

Console.WriteLine(userName != null ? userName : «User doesn’t exist.»);
```

You can apply the **Compress to Null Coalescing Operation** refactoring:



```
String userName = userName ?? "User doesn't exist."
Console.WriteLine(~~userName != null ? userName : "User doesn't exist."~~);
```

Converts a ternary expression to an equivalent null coalescing operation.

(*click on the image to see it in the original size*)

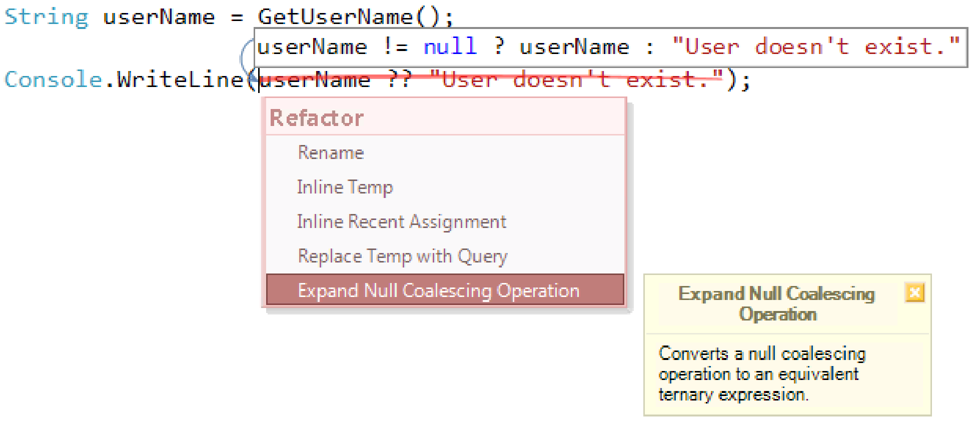
and use the null coalescing operator instead:

```
String userName = GetUserName();

Console.WriteLine(userName ?? «User doesn’t exist.»);
```

The **Expand Null Coalescing Operation** can be useful when you would like to change the condition of the check, for example. From a null check to something different, e.g.:

```
Console.WriteLine(userName != String.Empty ? userName : «User name is
undefined.»);
```

The preview hint of the refactoring will show you the resulting code before you apply it:



```
String userName = GetUserName();
                  userName != null ? userName : "User doesn't exist."
Console.WriteLine(userName ?? "User doesn't exist.");
```

Converts a null coalescing operation to an equivalent ternary expression.

(*click on the image to see it in the original size*)

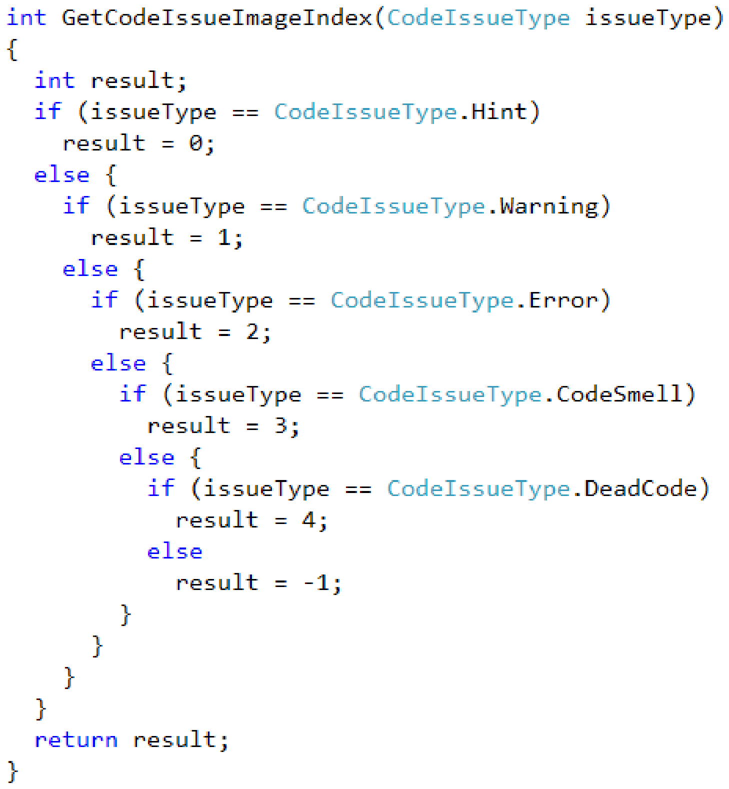
Flatten Conditional

This is one of the most powerful refactorings for conditionals – it helps you to simplify conditional statements by unindenting all or a portion of the conditional statement which improves the clarity of the code. This refactoring includes several different refactorings mentioned in *Martin Fowler*'s book, and a few others. These refactorings are applied, depending on the conditional blocks you have inside the source code. Here they are:

- **Replace Nested Conditional with Guard Clauses**

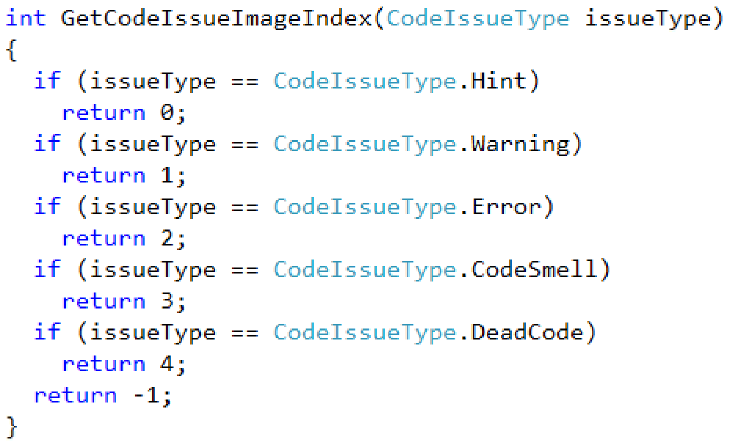
Replaces nested conditional with a guard clause, inverting the if-clause expression and exiting the code block if the new expression is satisfied. When the conditional behavior does not make the normal path of execution clear, you can remove nested if conditions with a series of separate if statements, for example:

Before refactoring:



```
int GetCodeIssueImageIndex(CodeIssueType issueType)
{
  int result;
  if (issueType == CodeIssueType.Hint)
    result = 0;
  else {
    if (issueType == CodeIssueType.Warning)
      result = 1;
    else {
      if (issueType == CodeIssueType.Error)
        result = 2;
      else {
        if (issueType == CodeIssueType.CodeSmell)
          result = 3;
        else {
          if (issueType == CodeIssueType.DeadCode)
            result = 4;
          else
            result = -1;
        }
      }
    }
  }
  return result;
}
```

Result:



```
int GetCodeIssueImageIndex(CodeIssueType issueType)
{
  if (issueType == CodeIssueType.Hint)
    return 0;
  if (issueType == CodeIssueType.Warning)
    return 1;
  if (issueType == CodeIssueType.Error)
    return 2;
  if (issueType == CodeIssueType.CodeSmell)
    return 3;
  if (issueType == CodeIssueType.DeadCode)
    return 4;
  return -1;
}
```

- **Remove Redundant Conditional**

Replaces conditional with a single statement, for example:

Before refactoring:

```
bool IsNotNull(object myObject)
{
  if (myObject == null)
    return false;
  else
    return true;
}
```

Result:

```
bool IsNotNull(object myObject)
{
  return myObject != null;
}
```

This part of the refactoring can also simplify ternary expressions:

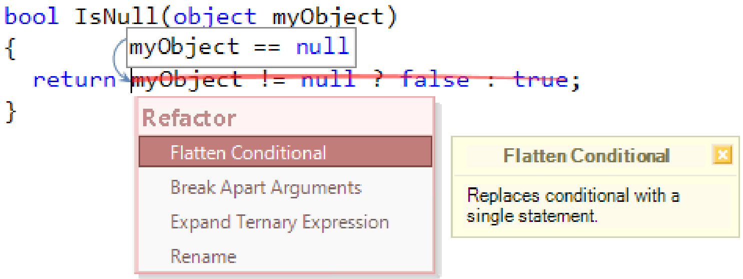
Before refactoring:

```
bool IsNull(object myObject)
{
  return myObject != null ? false : true;
}
```

Result:

```
bool IsNull(object myObject)
{
  return myObject == null;
}
```

As usual, you can see the preview hint, showing you the resulting code before applying the refactoring:



```
bool IsNull(object myObject)
{         myObject == null
  return myObject != null ? false : true;
}
```

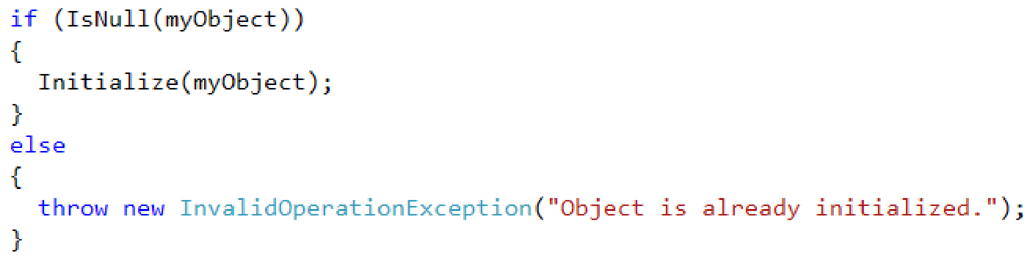
Flatten Conditional

Replaces conditional with a single statement.

- **Reverse Conditional followed by Remove Redundant Conditional**

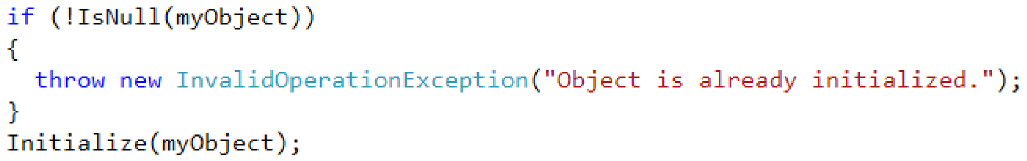
Inverts the logic of the conditional statement and unindents the redundant ‘else’ block, for example:

Before refactoring:



```
if (IsNull(myObject))
{
  Initialize(myObject);
}
else
{
  throw new InvalidOperationException("Object is already initialized.");
}
```

Result:

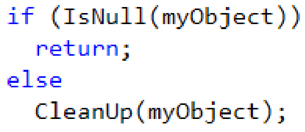


```
if (!IsNull(myObject))
{
  throw new InvalidOperationException("Object is already initialized.");
}
Initialize(myObject);
```

- **Remove Redundant Else**

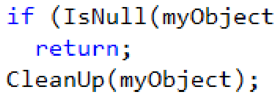
Removes the ‘else’ block delimiters and unindents it, for example:

Before refactoring:



```
if (IsNull(myObject))
  return;
else
  CleanUp(myObject);
```

Result:



```
if (IsNull(myObject))
  return;
CleanUp(myObject);
```

All of these refactorings are performed by **Flatten Conditional**. The refactoring fixes the following code issues:

- Nested conditional can be flattened
- Redundant Else Statement

Refactorings declarations and initializations

Refactorings for implicitly-typed local variables

Implicitly-typed local variables are variables declared without specifying the type explicitly. The type of such variables is being inferred from the expression that is used to initialize the variable at the time the code is compiled. Implicitly-typed variables are really useful for LINQ that creates anonymous types in queries, and for which you want to assign variables. However, implicitly-typed locals can be used with any variable declaration to enhance the readability, for example:

```
var CSharpCompilerVersionRequired = 3.0;
```

This implicitly-typed local variable will have the ‘double’ type. Or, better yet, this sample:

```
Dictionary<String, List<Int>> table = new Dictionary<String, List<Int>>();
```

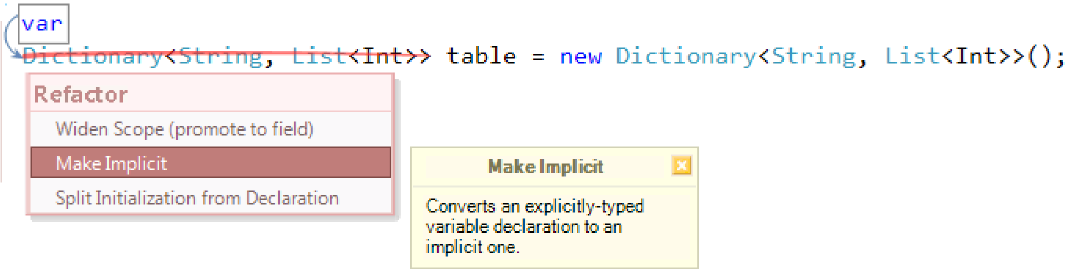
This code does not look impressive. If you don’t use implicitly-typed variables, you are forced to type the entire type: Dictionary<String, List<Int>>, instead of simply ‘*var*‘:

```
var table = new Dictionary<String, List<Int>>();
```

DevExpress Refactor! Pro and CodeRush Xpress has two refactorings, that allow you to convert from the implicitely-typed variables into explicitly-type variables, and vice versa. Here they are:

Make Implicit

Converts an explicitly-typed variable declaration to an implicit one. This is a simple refactoring which removes the type of a variable and makes it an implicitly-typed variable:



```
var
~~Dictionary<String, List<Int>>~~ table = new Dictionary<String, List<Int>>();
```

Refactor
Widen Scope (promote to field)
Make Implicit
Split Initialization from Declaration
Make Implicit

Converts an explicitly-typed variable declaration to an implicit one.

But sometimes, it does not only removes the type of a variable, but also converts an expression, so the compiler infers the type correctly without changing the logic of the code. Consider the following declaration:

```
ulong unsignedLong = 0;
```

or this:

```
int[] numbers = { 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 };
```

Applying the **Make Implicit** will result in producing the following code:

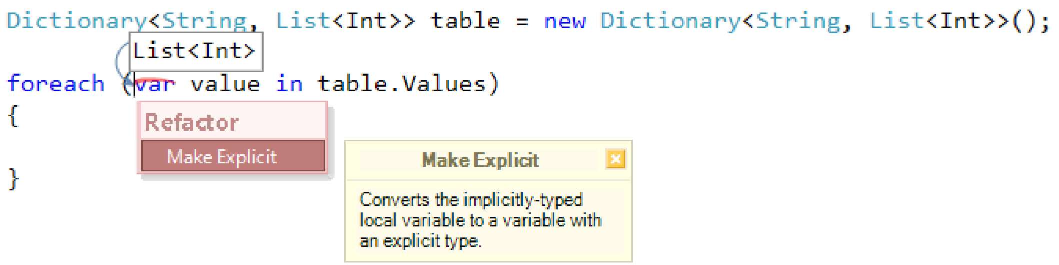
```
var unsignedLong = 0ul;
```

and

```
var numbers = new int[] { 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 };
```

Make Explicit

Converts the implicitly-typed local variable to a variable with an explicit type. This refactoring determines the type of the implicitly-typed variable before you compile the code and explicitly specifies it:



```
Dictionary<String, List<Int>> table = new Dictionary<String, List<Int>>();
foreach (var value in table.Values)
{

}
```

Make Explicit

Converts the implicitly-typed local variable to a variable with an explicit type.

There is also an advanced version of the **Make Explicit** refactoring called **Make Explicit** (**and Name Anonymous Type**). This refactoring converts the implicitly-typed local variable to a variable with an explicit type, then creates a named type to represent the expression of an anonymous type declaration, and replaces the anonymous type with a newly-declared type. To learn more about this refactoring, you may read the corresponding topic about refactorings for anonymous types.

Both **Make Implicit** and **Make Explicit** refactorings are available in *CSharp* and *Visual Basic* programming languages where implicitly-typed variable declarations are allowed starting from Visual Studio ٢٠٠٨ and the corresponding language versions.

Use Named Arguments

Named Arguments is a feature of the C# and Visual Basic languages introduced in the .NET Framework version 4.0. Named arguments allow you to explicitly specify a name for an argument for a particular parameter by associating the argument with the parameter’s name, rather than with the parameter’s position in the parameter list. Using named arguments frees you from the need to remember or to look up the order of parameters in the parameter lists of called methods.

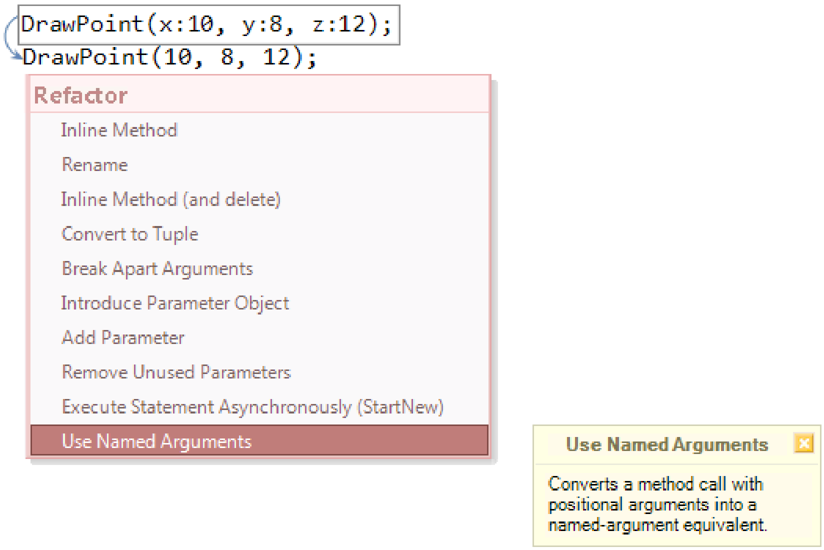
The **Use Named Arguments** refactoring shipped in Refactor! Pro converts position-based arguments inside a usual call to a method with a call with named arguments, for example:

```
DrawPoint(10, 8, 12);
```

Result:

```
DrawPoint(x: 10, y: 8, z: 12);
```

In the code preview hint you are able to see the resulting code:



```
DrawPoint(x:10, y:8, z:12);
DrawPoint(10, 8, 12);
```

Converts a method call with positional arguments into a named-argument equivalent.

Named arguments become very useful when a method definition also contains several optional parameters, and you want to specify which arguments you are passing. For example, we have the following method declaration:

```
void DrawPoint (int x = 0, int y = 0)
{
  // code goes here...
}
```

We could use named arguments to call the above method in any of the ways below:

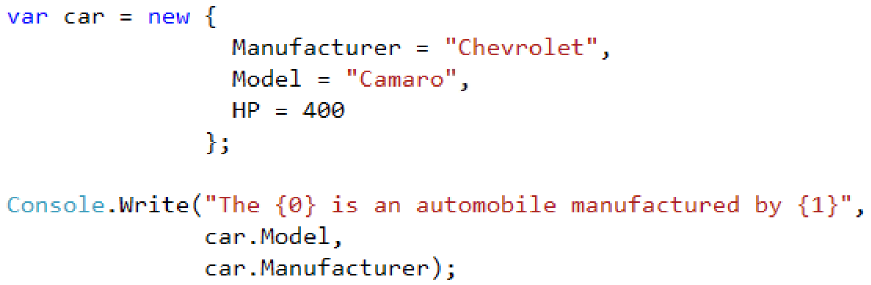
```
DrawPoint ();
DrawPoint (x: 10);
DrawPoint (y: 10);
DrawPoint (x: 10, y: 10);
DrawPoint (y: 10, x: 10);
```

Because both parameters of a method definition are optional, the first three cases will call a method with the default value passed for all non-specified arguments.

Name Anonymous Type

Anonymous types are type declarations that are generated automatically by the compiler without having to explicitly declare it. They provide a convenient way to encapsulate several read-only properties into a single object that is not declared in the code. Anonymous types are supported by C# and Visual Basic programming languages starting from Visual Studio 2008.

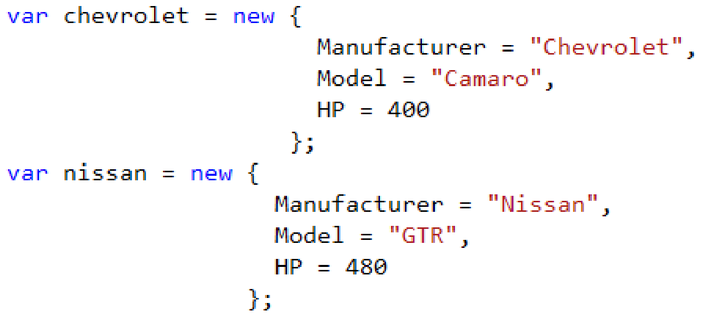
After you create an anonymous type object, you can assign it to an implicitly-typed local variable like this:



```
var car = new {
                Manufacturer = "Chevrolet",
                Model = "Camaro",
                HP = 400
              };

Console.Write("The {0} is an automobile manufactured by {1}",
              car.Model,
              car.Manufacturer);
```

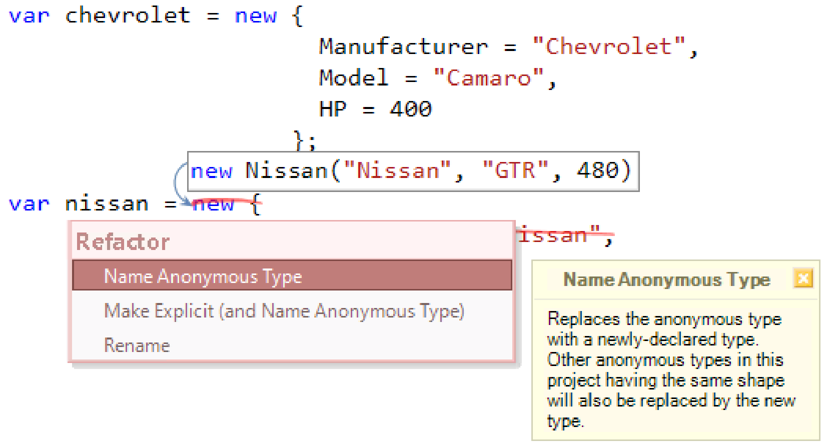
The compiler will automatically generate a new class for this anonymous type. If you want to persist it and have a source code of the class, you can automatically convert an anonymous type into a named type by using the **Name Anonymous Type** refactoring shipped in Refactor! Pro and CodeRush Xpress. This refactoring replaces an anonymous type with a one-to-one class declaration that is generated by the compiler. Additionally, all other anonymous types in this project having the same shape will also be replaced by the new type.

Consider the following code with two anonymous types:



```
var chevrolet = new {
                      Manufacturer = "Chevrolet",
                      Model = "Camaro",
                      HP = 400
                    };
var nissan = new {
                   Manufacturer = "Nissan",
                   Model = "GTR",
                   HP = 480
                 };
```

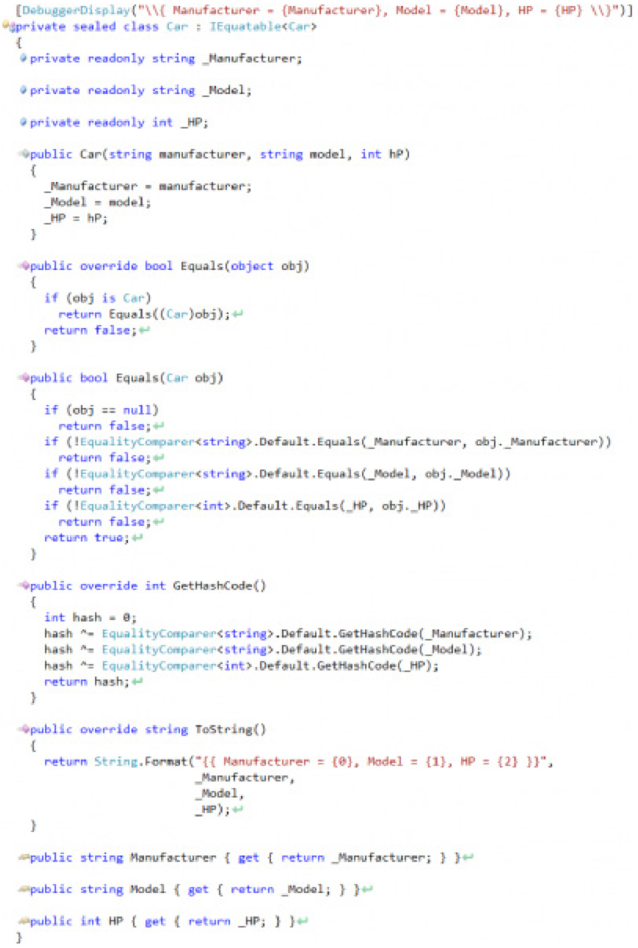
Applying the **Name Anonymous Type**:



```
var chevrolet = new {
                      Manufacturer = "Chevrolet",
                      Model = "Camaro",
                      HP = 400
                    };
                new Nissan("Nissan", "GTR", 480)
var nissan = new {
```

Replaces the anonymous type with a newly-declared type. Other anonymous types in this project having the same shape will also be replaced by the new type.

will result in the following class generated, that we immediately rename to “*Car*“ (*click the image to enlarge*):



```
[DebuggerDisplay("\\{ Manufacturer = {Manufacturer}, Model = {Model}, HP = {HP} \\}")]
private sealed class Car : IEquatable<Car>
{
  private readonly string _Manufacturer;

  private readonly string _Model;

  private readonly int _HP;

  public Car(string manufacturer, string model, int hP)
  {
    _Manufacturer = manufacturer;
    _Model = model;
    _HP = hP;
  }

  public override bool Equals(object obj)
  {
    if (obj is Car)
      return Equals((Car)obj);
    return false;
  }

  public bool Equals(Car obj)
  {
    if (obj == null)
      return false;
    if (!EqualityComparer<string>.Default.Equals(_Manufacturer, obj._Manufacturer))
      return false;
    if (!EqualityComparer<string>.Default.Equals(_Model, obj._Model))
      return false;
    if (!EqualityComparer<int>.Default.Equals(_HP, obj._HP))
      return false;
    return true;
  }

  public override int GetHashCode()
  {
    int hash = 0;
    hash ^= EqualityComparer<string>.Default.GetHashCode(_Manufacturer);
    hash ^= EqualityComparer<string>.Default.GetHashCode(_Model);
    hash ^= EqualityComparer<int>.Default.GetHashCode(_HP);
    return hash;
  }

  public override string ToString()
  {
    return String.Format("{{ Manufacturer = {0}, Model = {1}, HP = {2} }}",
                         _Manufacturer,
                         _Model,
                         _HP);
  }

  public string Manufacturer { get { return _Manufacturer; } }

  public string Model { get { return _Model; } }

  public int HP { get { return _HP; } }
}
```

And creation sides are changed to this:

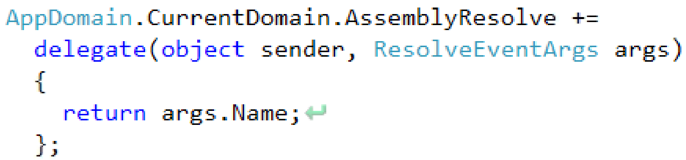
```
var chevrolet = new Car("Chevrolet", "Camaro", 400);
var nissan = new Car("Nissan", "GTR", 480);
```

Now you can declare instances of this class and extend the class as you desire.

There is one more similar refactoring for anonymous types – the **Make Explicit** (**and Name Anonymous Type**). In addition to creating a named type declaration that represents an anonymous type, it converts the implicitly-type local variable to a variable with an explicit type, replacing other anonymous types in this project having the same shape by the new type declaration:

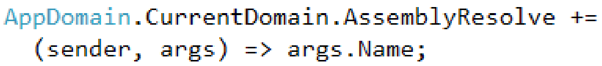
Refactorings anonymous methods and lambda expressions

An anonymous method is a method without a name – which is why it is called anonymous. You don't declare anonymous methods like regular methods. Instead, they get declared right inside a member as an inline code block. This is one of the greatest benefits of an anonymous method – it removes the requirement to create a named method elsewhere in the code, especially for simple operations. For example, you are able to pass an inline anonymous method as a delegate parameter, which means that you can use anonymous methods anywhere a delegate type is expected:



```
AppDomain.CurrentDomain.AssemblyResolve +=
  delegate(object sender, ResolveEventArgs args)
  {
    return args.Name;
  };
```

Lambda Expressions are simply an improvement to the syntax of anonymous methods. In other words, a lambda expression supersedes anonymous methods as a shorter way to write inline code. There is no longer a need to use the ‘delegate’ keyword, or provide the type of the method parameter. The type of a parameter can usually be inferred by the compiler from its use. The sample code above will look like this using lambda expressions:



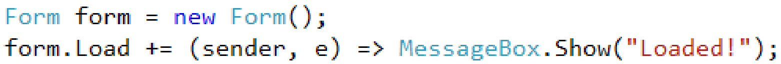
```
AppDomain.CurrentDomain.AssemblyResolve +=
  (sender, args) => args.Name;
```

The code looks much simpler and readable now. However, it takes some time to “acclimate” to this syntax. DevExpress Refactor! Pro and CodeRush Xpress has two pairs of refactorings, which are opposites of each other, and a couple of bonus refactorings for anonymous methods and lambda expressions. Before you apply a refactoring, you can see what your code will look like and learn the new syntax.

Pair #1 – conversion between anonymous methods and lambda expressions

- **Compress to Lambda Expression**

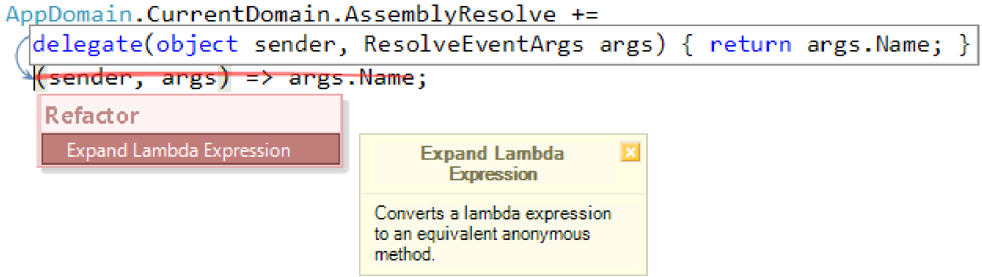
Converts an anonymous method into an equivalent lambda expression:



```
Form form = new Form();
form.Load += (sender, e) => MessageBox.Show("Loaded!");
```

- The opposite: **Expand Lambda Expression**

Converts a lambda expression to an equivalent anonymous method:



```
AppDomain.CurrentDomain.AssemblyResolve +=
  delegate(object sender, ResolveEventArgs args) { return args.Name; }
  ~~(sender, args) => args.Name;~~
```

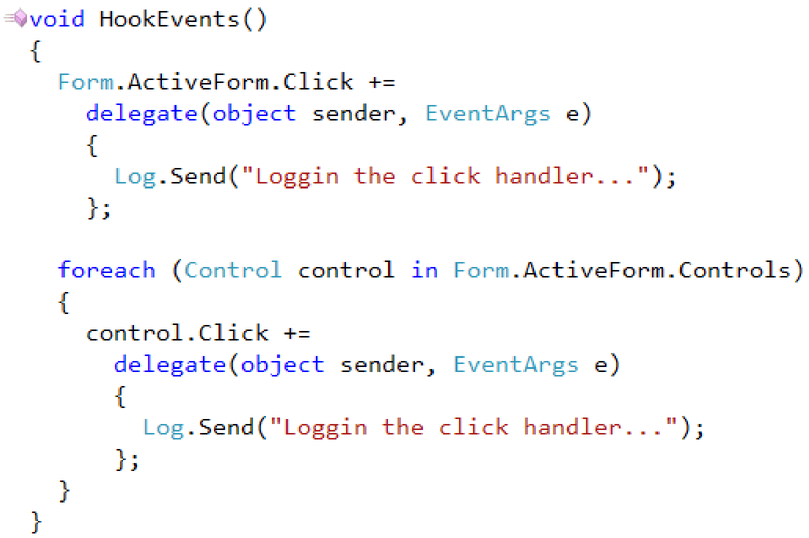
Expand Lambda Expression

Converts a lambda expression to an equivalent anonymous method.

Pair #2 – conversion between anonymous methods and regular methods

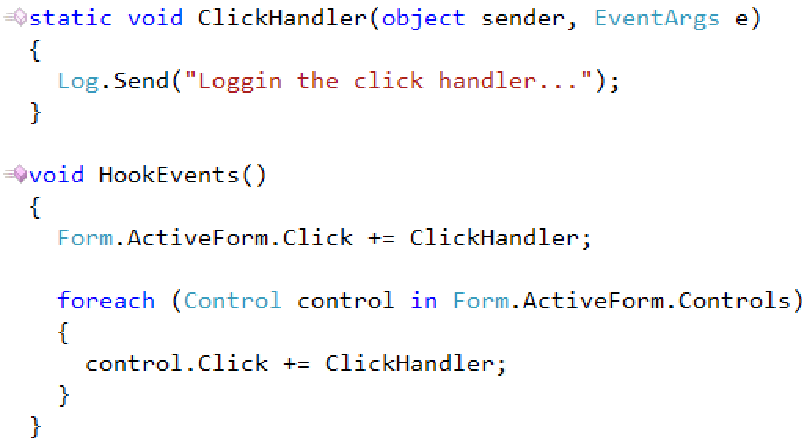
- **Name Anonymous Method**

Converts an anonymous method declaration that does not access local variables to a regular method declaration. The refactoring might be useful to make anonymous methods reusable in other locations. For example, when a single event handler is used for different events (especially when an events subscription code is located in different members):



```
void HookEvents()
{
  Form.ActiveForm.Click +=
    delegate(object sender, EventArgs e)
    {
      Log.Send("Loggin the click handler...");
    };

  foreach (Control control in Form.ActiveForm.Controls)
  {
    control.Click +=
      delegate(object sender, EventArgs e)
      {
        Log.Send("Loggin the click handler...");
      };
  }
}
```

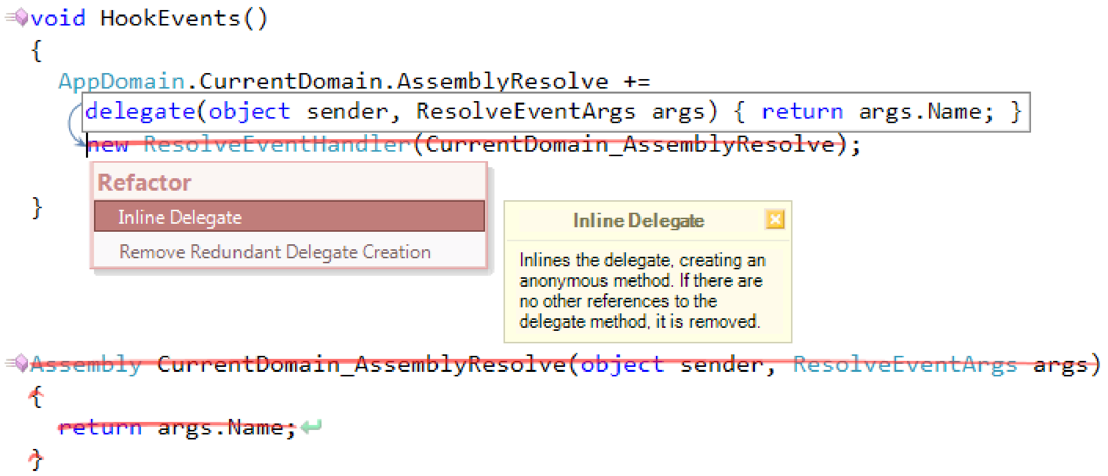
Naming anonymous methods will result in the following code:



```
static void ClickHandler(object sender, EventArgs e)
{
  Log.Send("Loggin the click handler...");
}

void HookEvents()
{
  Form.ActiveForm.Click += ClickHandler;

  foreach (Control control in Form.ActiveForm.Controls)
  {
    control.Click += ClickHandler;
  }
}
```

- The opposite: **Inline Delegate**

Converts a regular method into an anonymous method. If there are no other references to the delegate method, the method is removed.



```
void HookEvents()
{
  AppDomain.CurrentDomain.AssemblyResolve +=
    delegate(object sender, ResolveEventArgs args) { return args.Name; }
    ~~new ResolveEventHandler(CurrentDomain_AssemblyResolve);~~
}

~~Assembly CurrentDomain_AssemblyResolve(object sender, ResolveEventArgs args)~~
~~{~~
  ~~return args.Name;~~
~~}~~
```

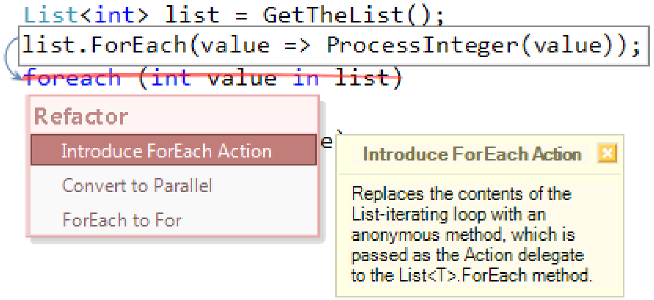
Inlines the delegate, creating an anonymous method. If there are no other references to the delegate method, it is removed.

- **Introduce ForEach Action**

Converts the generic List-iterating foreach loop into an anonymous method, which is passed as the Action delegate to the List.ForEach method. The ForEach method on a generic list performs the specified action on each element of the List. Consider the following code:

```
List<int> list = GetTheList();
foreach (int value in list)
{
  ProcessInteger(value);
}
```

Here’s the preview hint for this refactoring:



```
List<int> list = GetTheList();
list.ForEach(value => ProcessInteger(value));
~~foreach (int value in list)~~
```

Replaces the contents of the List-iterating loop with an anonymous method, which is passed as the Action delegate to the List<T>.ForEach method.

and the resulting code:

```
List<int> list = GetTheList();
list.ForEach(value => ProcessInteger(value));
```

- **Remove Redundant Type Specification**

Removes the redundant lambda parameter explicit type specification. The lambda expression may specify the type of its parameters explicitly, which is not required in most cases. The refactoring allows you to clean-up the lambda expression from the redundant type specification from this code:

```
Func<string, int, int> function =
  (string name, int param1, int param2) =>
  {
    ConvertValues(name, param1, param2)
  };
```

to this:

```
Func<string, int, int> function =
  (name, param1, param2) =>
  {
    ConvertValues(name, param1, param2);
  };
```

Refactoring loops and blocks

Adding and removing block delimiters in C#/C++ and JavaScript

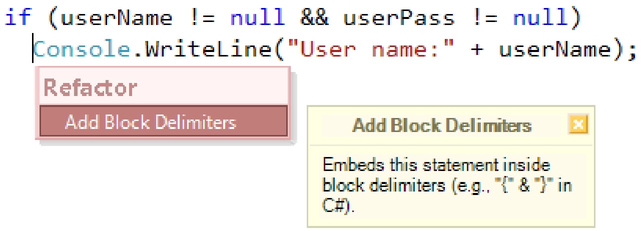
In C#, C++ and JavaScript languages, curly braces are used as block delimiters. A block allows multiple statements to be written inside. If a block contains a single statement, block delimiters, in most cases, are optional. For example, a single statement inside a loop does not require block delimiters:

```
foreach (Type item in Collection)
{
  Console.WriteLine(item);
}
```

There are two refactorings shipped in DevExpress Refactor! Pro and CodeRush Xpress that help you to add or res move optional block delimiters:

- **Add Block Delimiters**
- **Remove Block Delimiters**

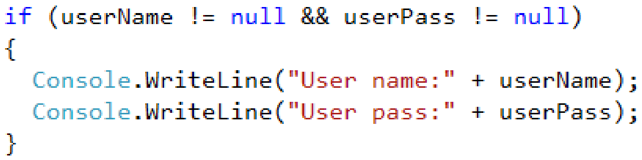
The **Add Block Delimiters** refactoring embeds a single statement into curly braces. This refactoring is available at the start of the statement:



```
if (userName != null && userPass != null)
  Console.WriteLine("User name:" + userName);
```

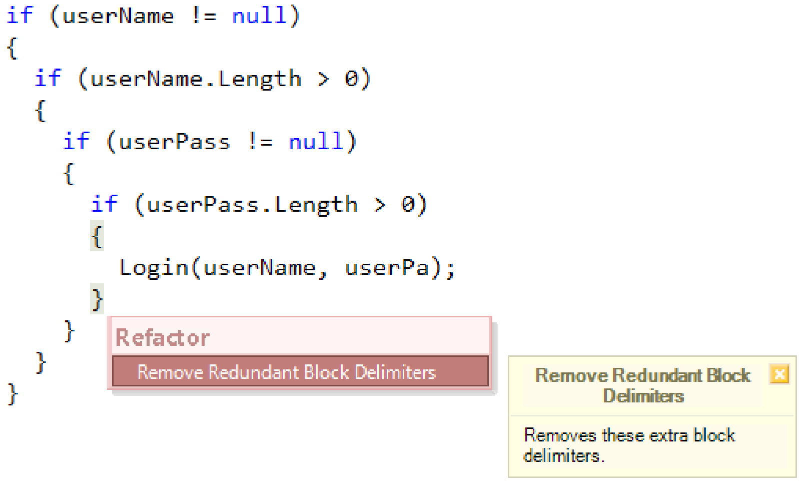
Embeds this statement inside block delimiters (e.g., "{" & "}" in C#).

Applying this refactoring may increase code readability and allow you to make the parent block a multi-statement block, so you can add additional statements inside, for example:



```
if (userName != null && userPass != null)
{
  Console.WriteLine("User name:" + userName);
  Console.WriteLine("User pass:" + userPass);
}
```

The **Remove Block Delimiters** refactoring, on the other hand, removes optional curly braces if appropriate. It is available on a curly brace:

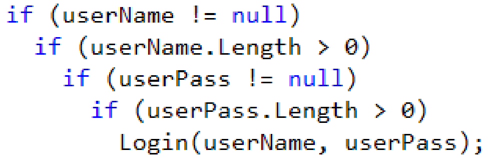


```
if (userName != null)
{
  if (userName.Length > 0)
  {
    if (userPass != null)
    {
      if (userPass.Length > 0)
      {
        Login(userName, userPa);
      }
    }
  }
}
```

Remove Redundant Block Delimiters

Removes these extra block delimiters.

This may also improve readability in certain cases:



```
if (userName != null)
  if (userName.Length > 0)
    if (userPass != null)
      if (userPass.Length > 0)
        Login(userName, userPass);
```

As always, refactorings are smart, and are not available where they are not appropriate.

Converting loops: ForEach to For and back

Let's take a look at the two CodeRush refactorings allowing us to convert a for-loop into foreach-loop, and a foreach-loop into for-loop, and compare these two loops to see what the difference is between them and which one is preferred.

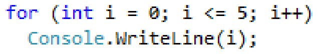
The for-loop is useful for iterating over elements and for sequential processing. It uses an iteration variable and is good when iterating over many items sequentially with a precise condition of the iteration termination. No collection of elements is required, but we can use the iteration variable to index a separate collection. In its basic form, the loop contains three sections:

1. An initialized iteration variable which specifies where to start;

2. A condition for the loop termination;

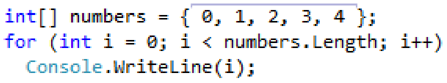
3. An iteration expression which modifies the iteration variable value.

The initialization, condition and iteration sections control how the loop will operate at run-time. Here is an example of a simple for-loop:



```
for (int i = 0; i <= 5; i++)
  Console.WriteLine(i);
```

And one more example:



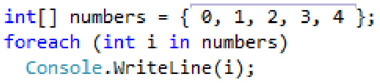
```
int[] numbers = { 0, 1, 2, 3, 4 };
for (int i = 0; i < numbers.Length; i++)
  Console.WriteLine(i);
```

Notice that the second loop iterates over array items starting from the very first item until all items are cycled. In such cases, when you require to cycle through each object in an array or a collection and when each item should be processed in turn, the foreach-loop might be more preferred. When the foreach-loop is used, each item in the collection is processed in series until every item has been referenced, or the loop has been terminated by a break command. It has two sections instead of three as the for-loop:

1. The type which determines the data type of the iterator during the loop's execution.

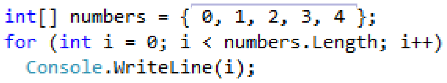
2. The collection section which names the collection or array that is to be cycled through.

Here is what it looks like if we cycle through the same numbers array as before:



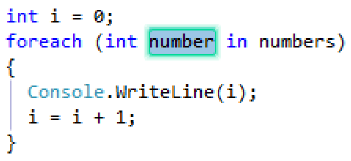
```
int[] numbers = { 0, 1, 2, 3, 4 };
foreach (int i in numbers)
  Console.WriteLine(i);
```

Which is much easier to read then the same for-loop, isn't it?



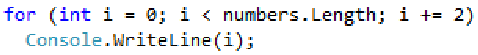
```
int[] numbers = { 0, 1, 2, 3, 4 };
for (int i = 0; i < numbers.Length; i++)
  Console.WriteLine(i);
```

The **For to ForEach** refactoring allows you to automatically convert the for-loop that iterates over items of a collection or array into the simpler foreach-loop. Just place the editor caret on the for-loop and press the CodeRush shortcut, you will see the preview hint of the resulting code. Once the refactoring is applied, the loop type is converted without modifying the logic of the code:



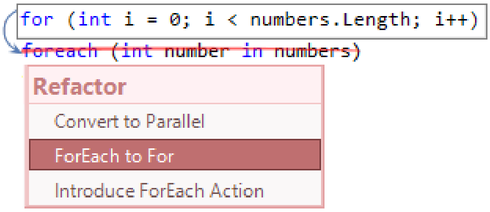
```
int i = 0;
foreach (int number in numbers)
{
  Console.WriteLine(i);
  i = i + 1;
}
```

However, sometimes when you need to modify the way the loop iterates over items, you might want to do the opposite operation – convert the foreach-loop into a for-loop, because the iteration variable in the for-loop provides more flexibility than a foreach-loop. Using the for-loop, you can iterate in a back order, or iterate over even indexes, or use other iterating algorithm, for example:



```
for (int i = 0; i < numbers.Length; i += 2)
  Console.WriteLine(i);
```

This loop increments the iteration variable by 2 each step of the loop. This means that the loop will iterate only over the even items of the array whereas the foreach-loop does not provide the capability to control which items and in which order are iterated. To help you to easily convert the foreach-loop into a for-loop, the opposite **ForEach to For** refactoring exists:



```
for (int i = 0; i < numbers.Length; i++)
```

~~foreach (int number in numbers)~~

Which results in the equal implementation of the loop but using a different loop type.

There are several differences between two loops that might be interesting and should be considered while choosing the right loop type:

- The items of the collection or array cannot be modified inside the foreach-loop. The reason you cannot remove or add elements inside a foreach-loop is that the looping mechanism requires that the state is preserved. The runtime cannot verify if you removed or added an item that is to be looped over in a subsequent iteration. The for-loop does not have such a restriction, because you have a full control over the iteration algorithm.

- The speed of the loops iteration, which is critical in many applications. There are performance factors related to the for-loop. A foreach-loop has at best equivalent performance in regular looping conditions as the for-loop. The for-loop is often the fastest way to loop over a series of numbers as it does not require allocations on the managed heap and is easy for the compiler to optimize. When it goes to loop through, foreach-loop copies the current array to a new one for the operation. While for-loop doesn't care of that part.

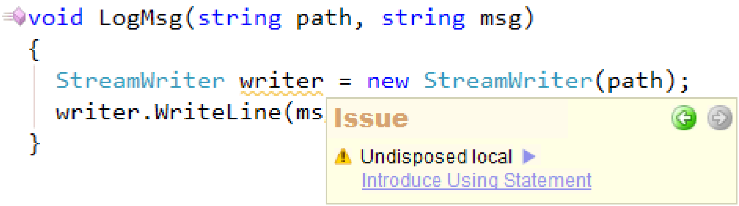
Conclusion

If you are planning to create high-performance code that is not iterating collections, it is better to use the for-loop. Even for collections, a foreach-loop is easier to read and understand, but it's not that efficient as the for-loop. A foreach-loop can be used when writing less important parts of a program, as it reduces the complexity of the notation, but choose the for-loop for 'hot' loops as a precaution, and use the specific loop converting automated refactorings for this task.

Refactoring using statements

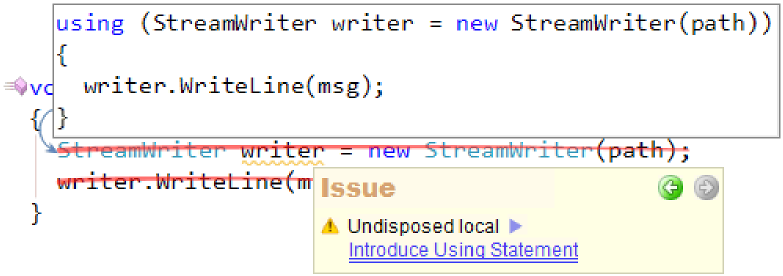
As many of you know, the using statement is a good tool for managing types which will be accessing unmanaged resources. The using statement provides a simple and convenient syntax that ensures that objects that implement the IDisposable interface are correctly disposed.

If the object is not disposed, CodeRush highlights the undisposed variables with the "Undisposed local" code issue:



```
void LogMsg(string path, string msg)
{
    StreamWriter writer = new StreamWriter(path);
    writer.WriteLine(ms
}
```

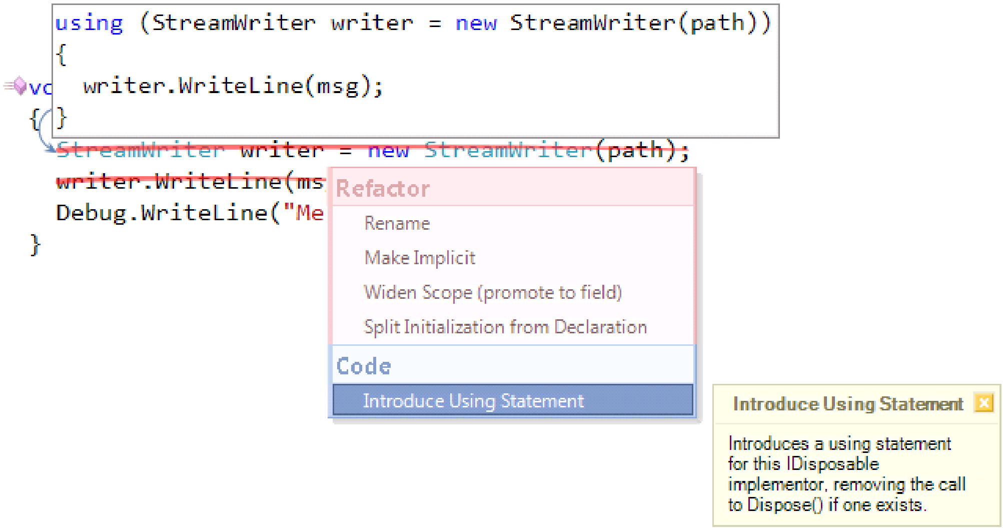
To fix this, you can apply the automatic fix – the **Introduce Using Statement** code provider:



```
using (StreamWriter writer = new StreamWriter(path))
{
  writer.WriteLine(msg);
}
```

```
vo
{
    StreamWriter writer = new StreamWriter(path);
    writer.WriteLine(m
}
```

The **Introduce Using Statement** code provider declares a using statement for the specified IDisposable implementer, removing the call to Dispose() if it exists. The code provider automatically detects and covers the required code into the using statement if applied on an undisposed variable, for example:



```
using (StreamWriter writer = new StreamWriter(path))
{
  writer.WriteLine(msg);
}
```

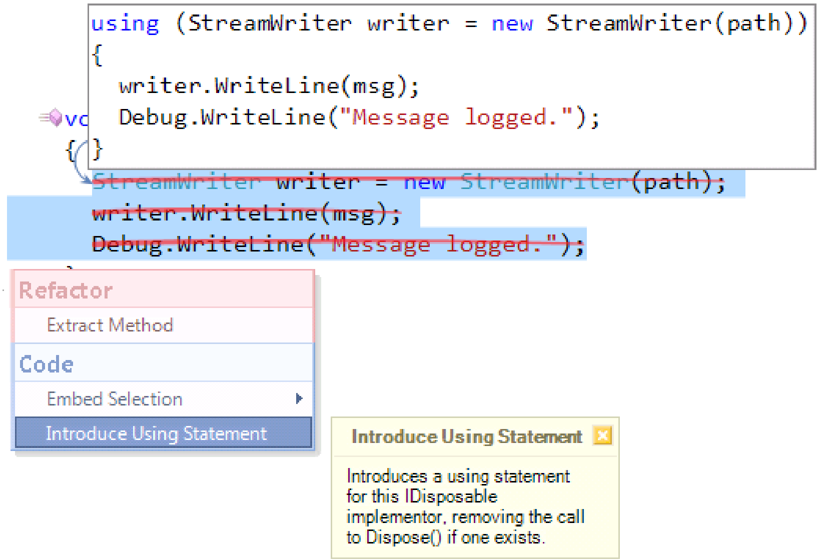
```
vo
{
    StreamWriter writer = new StreamWriter(path);
    writer.WriteLine(ms
    Debug.WriteLine("Me
}
```

Refactor
Rename
Make Implicit
Widen Scope (promote to field)
Split Initialization from Declaration
Code
Introduce Using Statement

Introduce Using Statement

Introduces a using statement for this IDisposable implementor, removing the call to Dispose() if one exists.

The refactoring preview hint helps you to see the refactored code, and the resulting code before applying it. If you wish to extend the code that must be placed inside the using statement, manually select it (including all required code):

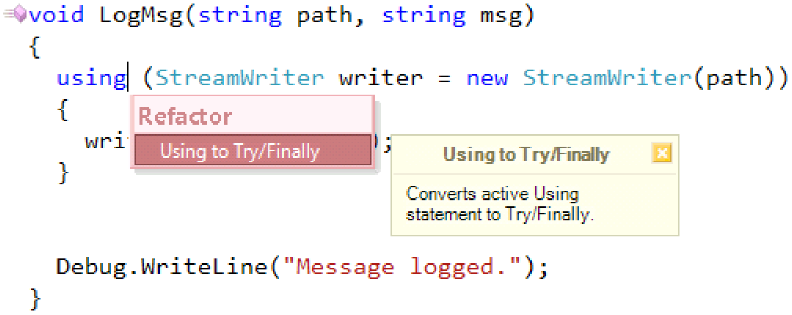


```
using (StreamWriter writer = new StreamWriter(path))
{
  writer.WriteLine(msg);
  Debug.WriteLine("Message logged.");
}
```

```
~~StreamWriter writer = new StreamWriter(path);~~
~~writer.WriteLine(msg);~~
~~Debug.WriteLine("Message logged.");~~
```

Introduces a using statement for this IDisposable implementor, removing the call to Dispose() if one exists.

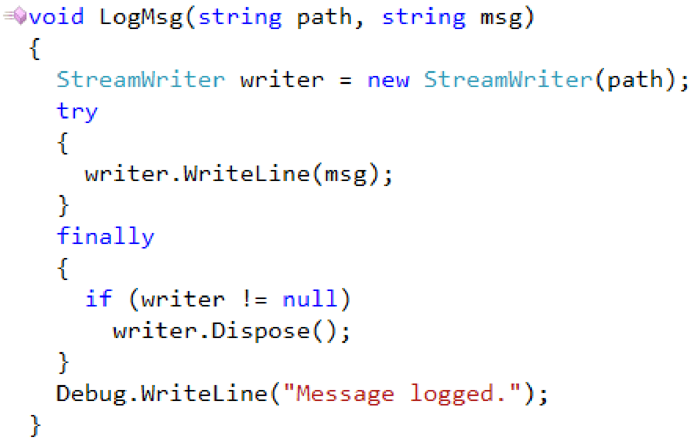
Instead of the using statement, you can call the object’s Dispose method to indicate that you want the object to clean up its resources. However, in this case, you must make sure that the Dispose() call will always be executed by wrapping the entire code into the try/finally block. To do that, you can apply anotherrefactoring that converts the selected using statement into a try/finally block – the **Using to Try/Finally** refactoring:



```
void LogMsg(string path, string msg)
{
  using (StreamWriter writer = new StreamWriter(path))
  {
    wri                                   );
  }

  Debug.WriteLine("Message logged.");
}
```

Converts active Using statement to Try/Finally.

The result of the refactoring:

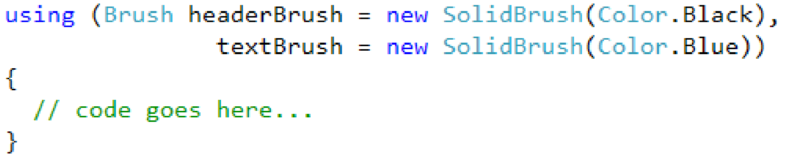


```
void LogMsg(string path, string msg)
{
  StreamWriter writer = new StreamWriter(path);
  try
  {
    writer.WriteLine(msg);
  }
  finally
  {
    if (writer != null)
      writer.Dispose();
  }
  Debug.WriteLine("Message logged.");
}
```

The Dispose() method will be called in the finally block even if an exception has been thrown during processing. But note that the scope of the variable is now changed. When the variable is declared inside of the using statement expression, the scope of the variable is the using statement and it is not visible outside of one. But after the statement is converted into a try/finally block, the scope of the variable will be increased, so it is visible to both try and finally blocks and the rest of the method (or parent block). This leads to the possibility that the object may be accidentally used again after control leaves the finally block even though the object probably no longer has access to its unmanaged resources. In other words, the object will no longer be fully initialized, and may cause an exception to be thrown if it is accessed. The Introduce Using Statement can fix this by converting try/finally block back to the using statement, so that the variable will remain in the scope of the using statement.

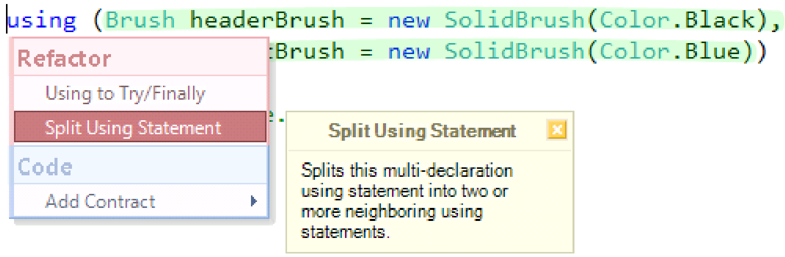
Both refactoring and code provider are the opposite of each other. You can refactor and improve the old code that uses the try/finally block or do the opposite – create a try/finally block for your advanced requirements.

The using statement may include multiple instances of the same type separated by a comma:



```
using (Brush headerBrush = new SolidBrush(Color.Black),
             textBrush = new SolidBrush(Color.Blue))
{
  // code goes here...
}
```

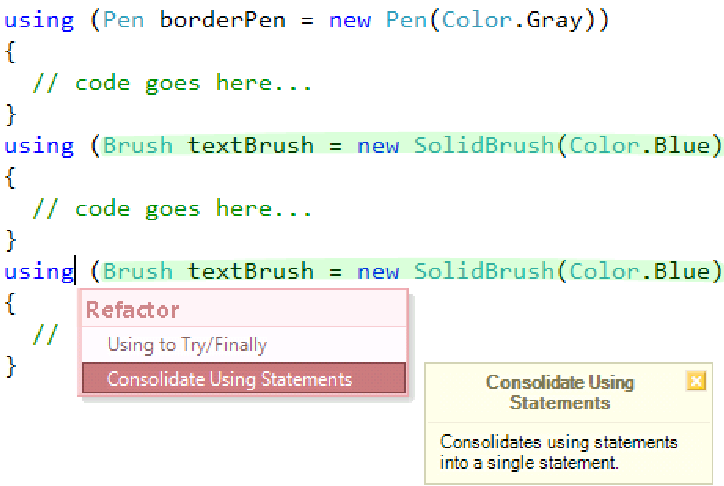
In this case these objects will both have the same scope. If you want to change the scope of one of the variables and write additional code, you can apply the **Split Using Statement** refactoring. The refactoring breaks the multi-declaration using statement into two or more neighboring using statements:



```
using (Brush headerBrush = new SolidBrush(Color.Black),
                  Brush = new SolidBrush(Color.Blue))
```

Splits this multi-declaration using statement into two or more neighboring using statements.

This refactoring has an opposite refactoring called the **Consolidate Using Statements** which combines several neighboring or nested using statements that cover variables of the same type into a single using statement:



```
using (Pen borderPen = new Pen(Color.Gray))
{
  // code goes here...
}
using (Brush textBrush = new SolidBrush(Color.Blue))
{
  // code goes here...
}
using (Brush textBrush = new SolidBrush(Color.Blue))
{
  //
}
```

Consolidate Using Statements

Consolidates using statements into a single statement.

All mentioned refactorings are available in both C# and Visual Basic languages if the specific language version supports the using statement.

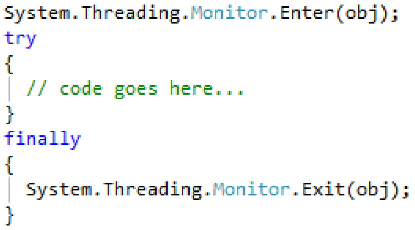
Lock to Try/Finally

Locking is essential in programs with multi-threading. It restricts code from being executed by more than one thread at the same time. The lock statement gets the exclusive monitor for an object and ensures that no other thread can access the object until the lock is released.

The lock statement automatically generates exception safe code, and in fact it is a syntactic shortcut for a call to the methods Monitor.Enter(obj) and Monitor.Exit(obj) with a try/finally block. However, if the thread is aborted after the lock is acquired but before entering the try block, the lock will not be released. Also, bear in mind that the Monitor. Enter() waits to acquire the lock forever which may introduce a possible deadlock condition.

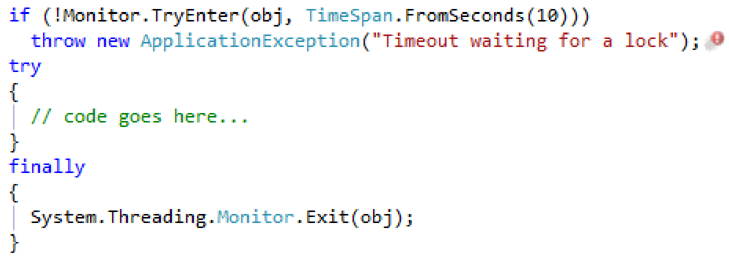
To avoid these issues with the lock statement, it is better either to use different overloads of the Monitor.Enter() call which take additional arguments, such as amount of time or a number of milliseconds to wait while trying to obtain a lock, or use the Monitor.TryEnter() call which lets you specify a timeout for an operation. But before you can use any of these workarounds, you need to convert the lock statement into its equivalent code. The **Lock to Try/Finally** refactoring can help you to automate this task.

The refactoring produces the following result once it is applied:



```
System.Threading.Monitor.Enter(obj);
try
{
  // code goes here...
}
finally
{
  System.Threading.Monitor.Exit(obj);
}
```

Now you can modify the code as required, for instance:

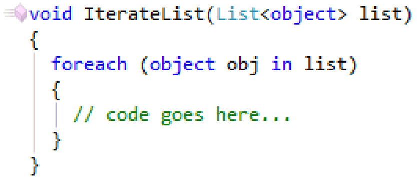


```
if (!Monitor.TryEnter(obj, TimeSpan.FromSeconds(10)))
  throw new ApplicationException("Timeout waiting for a lock");
try
{
  // code goes here...
}
finally
{
  System.Threading.Monitor.Exit(obj);
}
```

This code does not have issues as with the lock statement mentioned above.

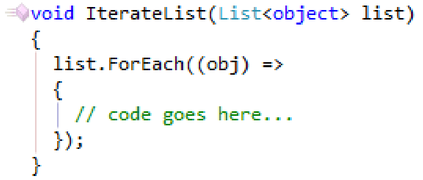
Introduce ForEach Action

It is a rather trivial task to iterate over each item of the generic List using a foreach-loop as follows:



```
void IterateList(List<object> list)
  {
    foreach (object obj in list)
    {
      // code goes here...
    }
  }
```

However, you might be aware that the generic list has its own ForEach method which performs the specified operation (Action) on each element of the list:

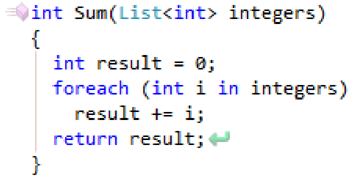


```
void IterateList(List<object> list)
  {
    list.ForEach((obj) =>
    {
      // code goes here...
    });
  }
```

So, what's the difference between these two methods of iterating list items? Let's find out.

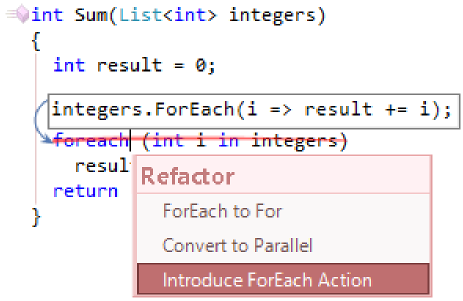
The first difference is that the foreach-loop does not allow you to modify the state by adding or removing items while iterating over the collection. But with the List.ForEach call you can modify the underlying collection without questions asked, where as with the foreach syntax, you'll get an exception if you do that.

The second difference is about performance. Let's compare the standard foreach-loop and the ForEach call on the generic list with the following block of code:



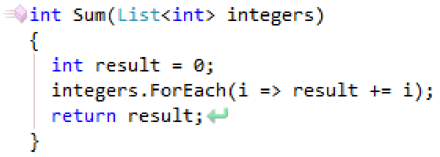
```
int Sum(List<int> integers)
  {
    int result = 0;
    foreach (int i in integers)
      result += i;
    return result;
  }
```

To convert this code into a ForEach call, the **Introduce ForEach Action** refactoring can be used:



```
int Sum(List<int> integers)
{
  int result = 0;
  integers.ForEach(i => result += i);
  foreach (int i in integers)
    resul
  return
}
```

that will produce the required code automatically once applied:



```
int Sum(List<int> integers)
{
  int result = 0;
  integers.ForEach(i => result += i);
  return result;
}
```

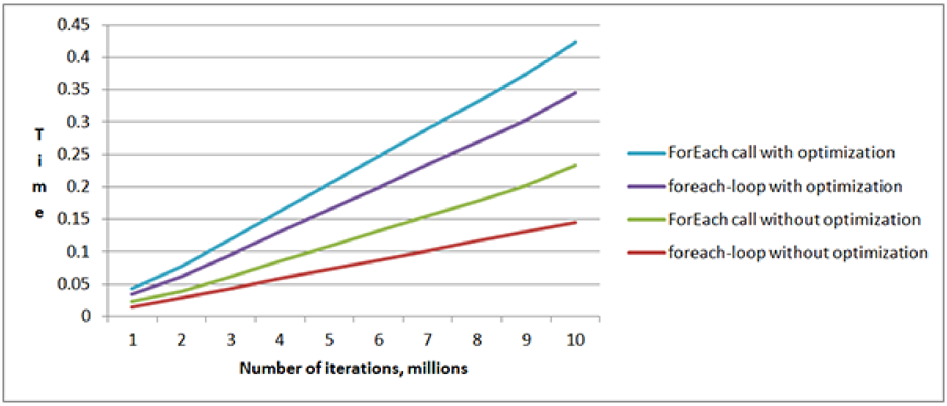
Now, we can test both list-iteration methods in a test application with a different number of iterations. I got the following results on my machine (Core2 Duo T7250 @2.00Ghz, 3.00Gb RAM):

Number of iterations	foreach-loop	List<int>.ForEach
1,000	0.000021	0.000015
5,000	0.000082	0.000049
10,000	0.000156	0.000094
50,000	0.000786	0.000510
100,000	0.001551	0.000889
500,000	0.007299	0.004454
1,000,000	0.014920	0.008751
5,000,000	0.072565	0.043666
10,000,000	0.145069	0.087591

According to this result, it turns out that the ForEach call is much faster than the default foreach-loop. However, this result is taken without the ‘Optimize code’ build option set. If we turn it on, we will get the following result:

# iterations	foreach-loop	List<int>.ForEach
1,000	0.000016	0.000013
5,000	0.000062	0.000045
10,000	0.000119	0.000084
50,000	0.000555	0.000322
100,000	0.001113	0.000790
500,000	0.005541	0.003921
1,000,000	0.011354	0.007964
5,000,000	0.055643	0.038843
10,000,000	0.112101	0.078943

If we compare two results, we see that the foreach-loop gets optimized in about 23%, but the ForEach does not receive as much optimization. However, the ForEach call is still faster than the foreach-loop, whether the code is optimized or not. Let’s take a look at the picture illustrating the results:



So, the ForEach might be preferred in performance critical code blocks. If you’d like to convert the existing foreach-loop iterating over the generic list, consider using the **Introduce ForEach Action** refactoring which will convert the foreach-loop quickly.

Refactoring Visual Basic With statements

The With statement (in Visual Basic) is used to execute a series of statements of the particular object without requalifying the name of the object. In other words, we can use the With statement to reduce the number of times we needed to manually type the full path to the target object. For example, this function inverts the background color of the active control of the active form in the current Windows application:

```
Sub InvertActiveControlColor()
  Dim R As Byte = ActiveForm.ActiveControl.BackColor.R
  Dim G As Byte = ActiveForm.ActiveControl.BackColor.G
  Dim B As Byte = ActiveForm.ActiveControl.BackColor.B

  Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
  ActiveForm.ActiveControl.BackColor = invertedColor
End Sub
```

Using the With statement in this code will look as follows:

```
Sub InvertActiveControlColor()
  With ActiveForm.ActiveControl.BackColor
    Dim R As Byte = .R
    Dim G As Byte = .G
    Dim B As Byte = .B

    Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
    ActiveForm.ActiveControl.BackColor = invertedColor
  End With
End Sub
```

Inlining the temporary variables will yield into the following code:

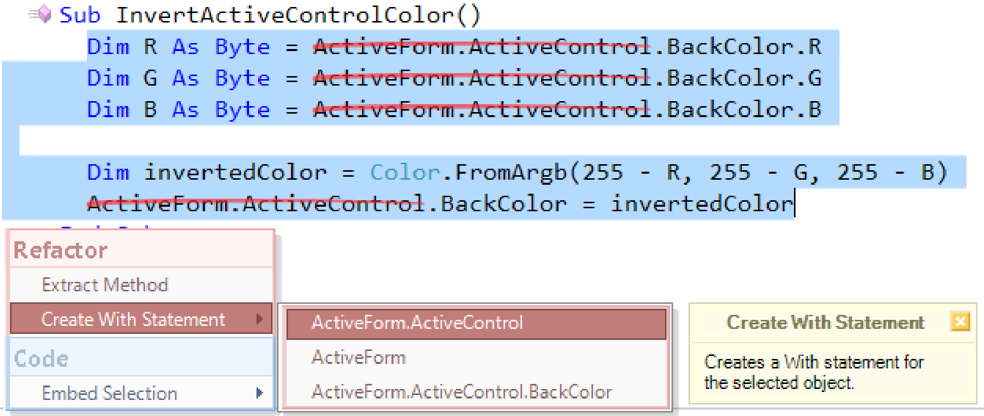
```
Sub InvertActiveControlColor()
  With ActiveForm.ActiveControl.BackColor
    Dim invertedColor = Color.FromArgb(255 - .R, 255 - .G, 255 - .B)
    ActiveForm.ActiveControl.BackColor = invertedColor
  End With
End Sub
```

Given the choice of creating a With statement for use with an object or a standard initialization, it is recommended to choose the With statement over the usual initialization because it can improve performance and your code will be more efficient. If your code has repeated invocations of a property or a method, the With statement will create a local variable and assign it to the object that you used as an expression of the With statement, so the method or property will only be called once instead of several times on each invocation.

DevExpress Refactor! Pro and CodeRush Xpress has several automatic refactoring operations that allow you to manipulate a With statement without manual code retyping:

Create With Statement

This refactoring creates a With statement for the selected object from the selected code. For example, you can apply the **Create With Statement** on the code above to automatically create a With statement:

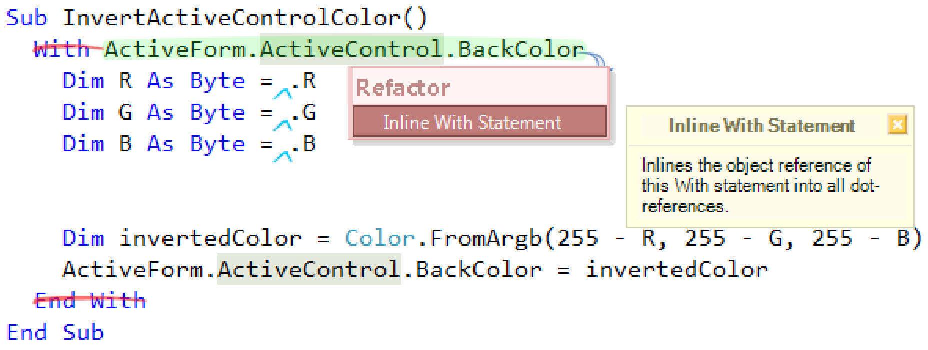


```
Sub InvertActiveControlColor()
    Dim R As Byte = ~~ActiveForm.ActiveControl~~.BackColor.R
    Dim G As Byte = ~~ActiveForm.ActiveControl~~.BackColor.G
    Dim B As Byte = ~~ActiveForm.ActiveControl~~.BackColor.B

    Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
    ~~ActiveForm.ActiveControl~~.BackColor = invertedColor
```

It allows you to choose one of the appropriate objects if there are several objects available. The chosen object will be used as an expression of the With statement.

Inline With Statement

This refactoring is the opposite of the **Create With Statement** refactoring. It allows you to inline the expression of the With statement back into the source code:



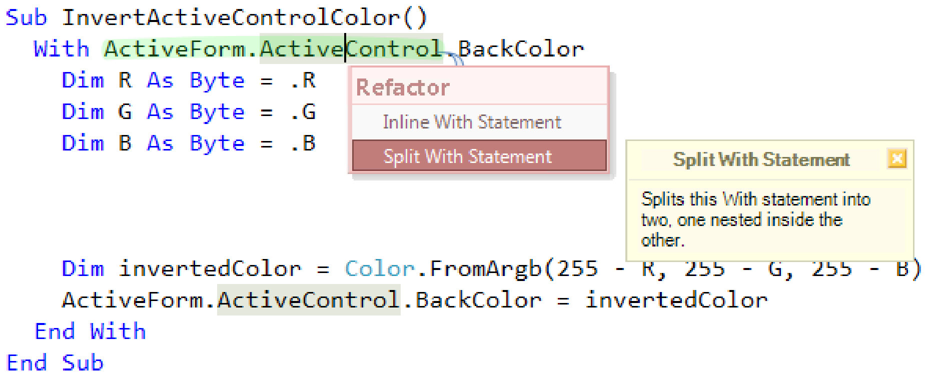
```
Sub InvertActiveControlColor()
  ~~With~~ ActiveForm.ActiveControl.BackColor
    Dim R As Byte = .R
    Dim G As Byte = .G
    Dim B As Byte = .B


    Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
    ActiveForm.ActiveControl.BackColor = invertedColor
  ~~End With~~
End Sub
```

Inlines the object reference of this With statement into all dot-references.

It is useful when you want to change the object that is used as the expression of the existing With statement and recreate a With statement that will use a different object.

Split With Statement

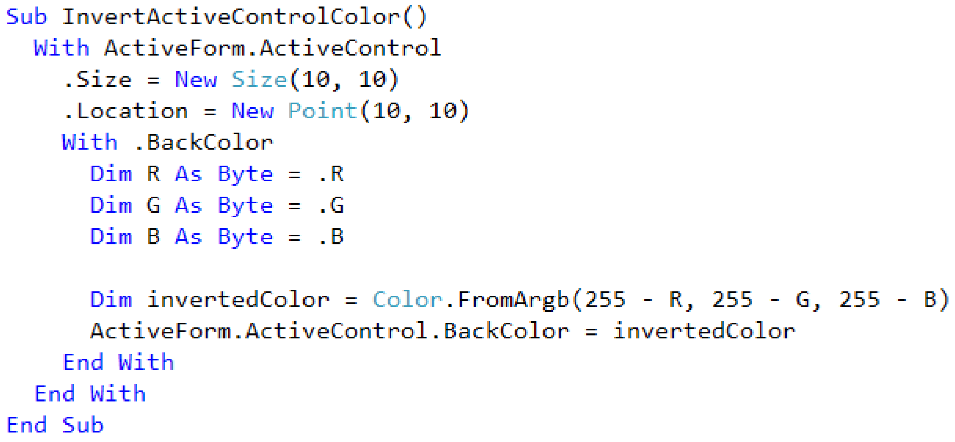
This refactoring allows you to automatically split the With statement into two With statements. This is useful when you need to access properties of a parent object that is used inside the expression of the With statement. One With statement will be nested in the other one, for example:



```
Sub InvertActiveControlColor()
  With ActiveForm.ActiveControl.BackColor
    Dim R As Byte = .R
    Dim G As Byte = .G
    Dim B As Byte = .B


    Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
    ActiveForm.ActiveControl.BackColor = invertedColor
  End With
End Sub
```

Splits this With statement into two, one nested inside the other.

Once this refactoring is executed, you can access properties of the parent object as well:



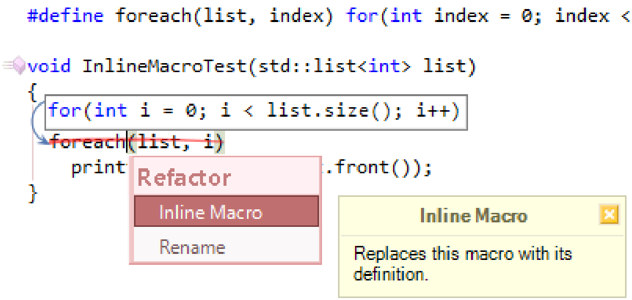
```
Sub InvertActiveControlColor()
  With ActiveForm.ActiveControl
    .Size = New Size(10, 10)
    .Location = New Point(10, 10)
    With .BackColor
      Dim R As Byte = .R
      Dim G As Byte = .G
      Dim B As Byte = .B

      Dim invertedColor = Color.FromArgb(255 - R, 255 - G, 255 - B)
      ActiveForm.ActiveControl.BackColor = invertedColor
    End With
  End With
End Sub
```

Refactoring C++ macros

CodeRush Pro includes three refactorings that work with macros, aliases and typedef declarations in C++. Here they are:

Inline Macro

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced with the contents of the macro by the preprocessor when an application is being compiled. To replace the current macro with its definition in the source code, the **Inline Macro** refactoring can be used. If the current macro contains other nested macros in its definition, all macros are inlined:



```
#define foreach(list, index) for(int index = 0; index < list.size(); index++)

void InlineMacroTest(std::list<int> list)
{
  for(int i = 0; i < list.size(); i++)
  ~~foreach(list, i)~~
    print          .front());
}
```

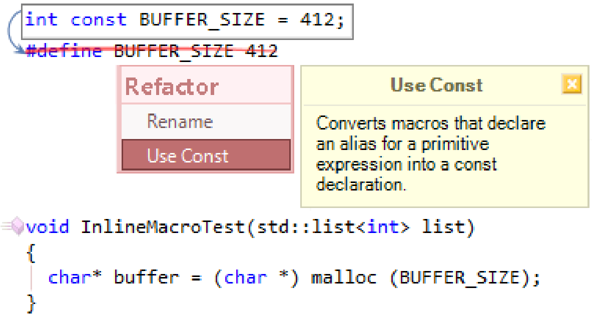
Inline Macro

Replaces this macro with its definition.

The refactoring available for all types of macros: object-like macros that resemble data objects when used, and function-like macros that resemble function calls.

Use Const

The **Use Const** refactoring converts an object-like macro that defines a simple identifier with a value into a const declaration:



```
int const BUFFER_SIZE = 412;
~~#define BUFFER_SIZE 412~~

void InlineMacroTest(std::list<int> list)
{
  char* buffer = (char *) malloc (BUFFER_SIZE);
}
```

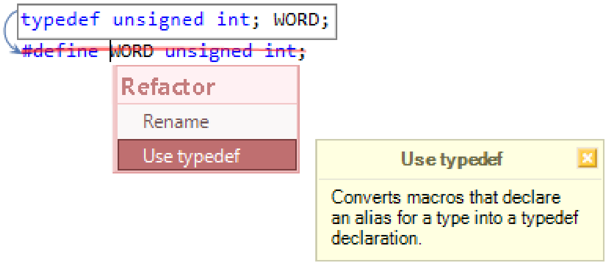
Use Const

Converts macros that declare an alias for a primitive expression into a const declaration.

A constant declaration is useful for parameters which are used in the application but are do not need to be modified after the application is compiled. It has an advantage over the macros in that it is understood and used by the compiler itself. In other words, it is not substituted into the source code by the preprocessor before reaching the compiler as a macro definition, so error messages might be more helpful when using const declarations.

Use typedef

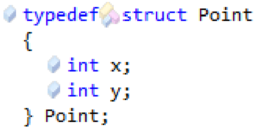
The **Use typedef** refactoring converts an object-like macro that defines a built-in data type or an existing class that is either provided by one f the libraries or defined in the code into a typedef declaration:



```
typedef unsigned int; WORD;
~~#define WORD unsigned int;~~
```

Converts macros that declare an alias for a type into a typedef declaration.

You can use typedef declarations to construct shorter or more meaningful names for types already defined by the language or for types that you have declared, e.g.:



```
typedef struct Point
{
    int x;
    int y;
} Point;
```

Typedef names allow you to encapsulate implementation details that may change and make your code easier to modify, and more understandable.

Refactorings for moving and extracting methods

Extract Method

The **Extract Method** refactoring creates a new method from the selected code block. The selection is replaced with appropriate calling code to invoke the newly-declared method. The **Extract Method** is great when you need to turn a big, complex method into smaller, simpler ones. Small methods are much easier to maintain, and encourage code reuse, and also have the following advantages:

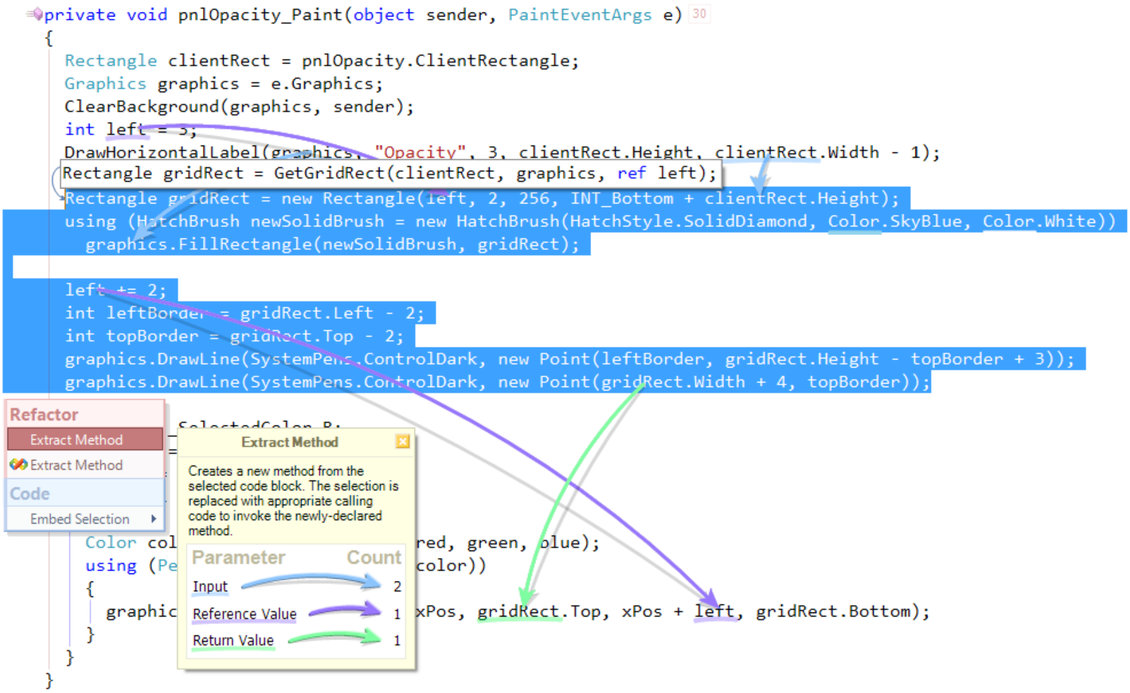
- It increases the chance that other methods can use a simple method when the method is finely organized and well-formed.

- It allows the higher-level methods to read more like a series of comments, which improves the code readability. Simple methods with good names comment themselves, and improve overall code clarity. Overriding also is easier when the methods are finely grained.

Opposite:

The opposite of this refactoring is the Inline Method refactoring.

Sample:



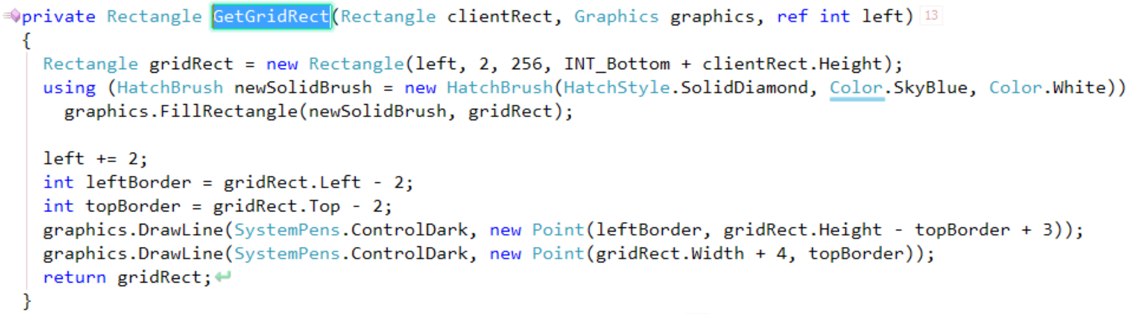
```
private void pnlOpacity_Paint(object sender, PaintEventArgs e) 30
{
  Rectangle clientRect = pnlOpacity.ClientRectangle;
  Graphics graphics = e.Graphics;
  ClearBackground(graphics, sender);
  int left = 3;
  DrawHorizontalLabel(graphics, "Opacity", 3, clientRect.Height, clientRect.Width - 1);
  Rectangle gridRect = GetGridRect(clientRect, graphics, ref left);
  Rectangle gridRect = new Rectangle(left, 2, 256, INT_Bottom + clientRect.Height);
  using (HatchBrush newSolidBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.SkyBlue, Color.White))
    graphics.FillRectangle(newSolidBrush, gridRect);

  left += 2;
  int leftBorder = gridRect.Left - 2;
  int topBorder = gridRect.Top - 2;
  graphics.DrawLine(SystemPens.ControlDark, new Point(leftBorder, gridRect.Height - topBorder + 3));
  graphics.DrawLine(SystemPens.ControlDark, new Point(gridRect.Width + 4, topBorder));

  SelectedColor.B;
    Color col                      red, green, blue);
    using (Pe                      color))
    {
      graphic                      xPos, gridRect.Top, xPos + left, gridRect.Bottom);
    }
  }
}
```

Extract Method

Creates a new method from the selected code block. The selection is replaced with appropriate calling code to invoke the newly-declared method.

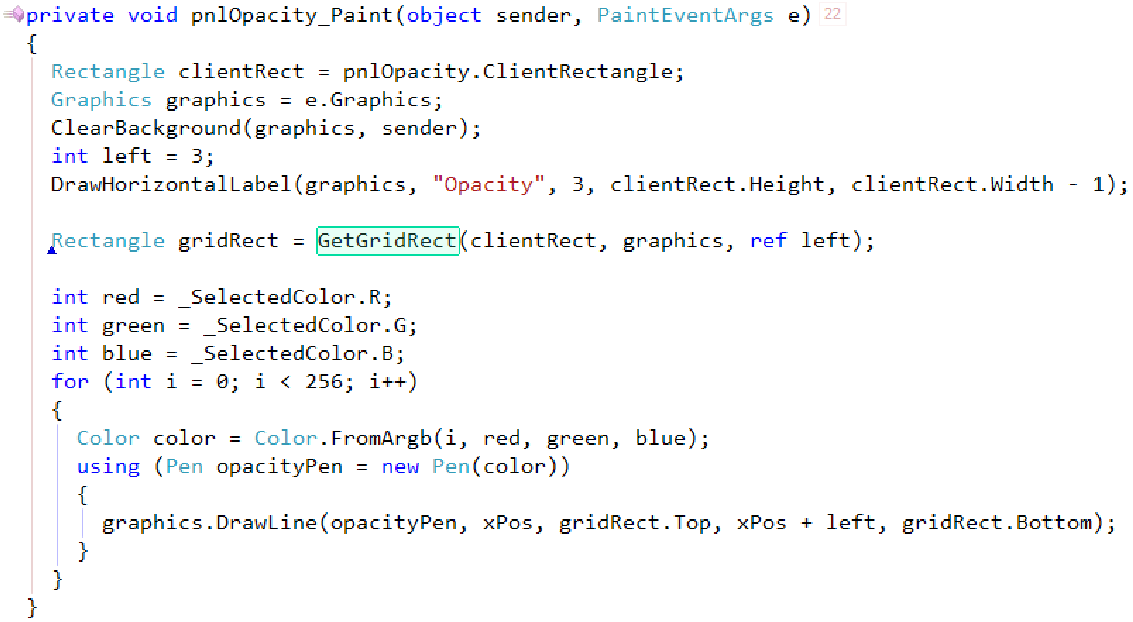
Parameter	Count
Input	2
Reference Value	1
Return Value	1

Result:



```
private Rectangle GetGridRect(Rectangle clientRect, Graphics graphics, ref int left) 13
{
  Rectangle gridRect = new Rectangle(left, 2, 256, INT_Bottom + clientRect.Height);
  using (HatchBrush newSolidBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.SkyBlue, Color.White))
    graphics.FillRectangle(newSolidBrush, gridRect);

  left += 2;
  int leftBorder = gridRect.Left - 2;
  int topBorder = gridRect.Top - 2;
  graphics.DrawLine(SystemPens.ControlDark, new Point(leftBorder, gridRect.Height - topBorder + 3));
  graphics.DrawLine(SystemPens.ControlDark, new Point(gridRect.Width + 4, topBorder));
  return gridRect;
}
```

Calling side:



```
private void pnlOpacity_Paint(object sender, PaintEventArgs e) 22
{
  Rectangle clientRect = pnlOpacity.ClientRectangle;
  Graphics graphics = e.Graphics;
  ClearBackground(graphics, sender);
  int left = 3;
  DrawHorizontalLabel(graphics, "Opacity", 3, clientRect.Height, clientRect.Width - 1);

  Rectangle gridRect = GetGridRect(clientRect, graphics, ref left);

  int red = _SelectedColor.R;
  int green = _SelectedColor.G;
  int blue = _SelectedColor.B;
  for (int i = 0; i < 256; i++)
  {
    Color color = Color.FromArgb(i, red, green, blue);
    using (Pen opacityPen = new Pen(color))
    {
      graphics.DrawLine(opacityPen, xPos, gridRect.Top, xPos + left, gridRect.Bottom);
    }
  }
}
```

Notes:

- You can choose the newly declared method’s location using the target picker (see **Options** below).

- Right after **Extract Method** has completed, **Rename** is automatically enabled for all the extracted method’s parameters so you can easily give them appropriate names.

- Local variables used in the extracted code block are automatically passed to the newly created method as parameters.

- If a variable hasn’t been assigned before the extracted block and is modified within this block and used after the block, it is passed to the extracted method as an out parameter.

- If a variable has been assigned before the extracted block, is modified within this block and is used after the block in the source method, it is passed as ref parameter.

- The extracted method can be organized as a *Function* instead of *Sub* in Visual Basic (or as a function rather than a *void* procedure in C#) if there are out parameters. See **Options** for more information.

- If the source method is a function and the extracted code block contains return statements that return this function’s value, the extracted method is also organized as a function having the same return value as the source function.

- If the extracted code block is an expression, the extracted method will be a function with the appropriate return value type.

- If the source method is a *Sub* in Visual Basic (or *void* function in C#) and the extracted code block contains return statements, then the extracted method includes an additional out parameter that indicates whether the source function should return after the extracted function call. The appropriate code is inserted into the source function automatically.

Tips:

- If the extracted method's return value is assigned to a variable that is referred but not changed later, you can use **Inline Temp** to replace all variable occurrences with the extracted method call. This will eliminate a redundant local variable.

- If the extracted method contains an out parameter that you want to make the method's return value, use the **Reorder Parameters** refactoring to move the out parameter left until it becomes the return value.

Options:

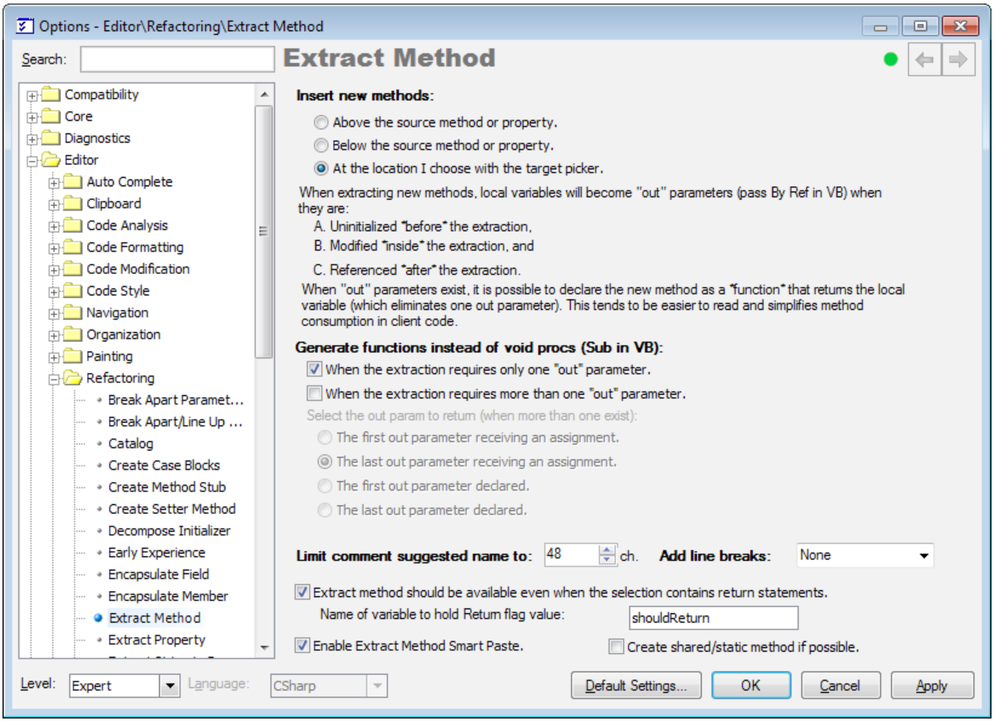
- You can choose whether the extracted method should be located before or after the source method or property or whether its position should be specified manually by using the target picker.

- You can choose whether the extracted method should be a *Function* instead of a *Sub* in Visual Basic (function instead of a *void* procedure in C#) to become available only when you have a single out parameter, or available when you have multiple out parameters.

- If you've chosen to generate functions when multiple out parameters are available, you can choose which parameter should be the function's default return value – the first parameter receiving an assignment, the last parameter receiving an assignment, the first out parameter declared or the last out parameter declared.

- You can choose whether the **Extract Method** should be available for code blocks that contain return statements. If so, you can set the name for the variable holding the return flag.

- You can enable or disable the Intelligent Paste option that allows you to cut code blocks and paste them inside a class or struct automatically enabling **Extract Method**.



Options - Editor\Refactoring\Extract Method

Extract Method

Insert new methods:

○ Above the source method or property.
○ Below the source method or property.
● At the location I choose with the target picker.

When extracting new methods, local variables will become "out" parameters (pass By Ref in VB) when they are:
A. Uninitialized "before" the extraction,
B. Modified "inside" the extraction, and
C. Referenced "after" the extraction.

When "out" parameters exist, it is possible to declare the new method as a "function" that returns the local variable (which eliminates one out parameter). This tends to be easier to read and simplifies method consumption in client code.

Generate functions instead of void procs (Sub in VB):

☑ When the extraction requires only one "out" parameter.
☐ When the extraction requires more than one "out" parameter.

Select the out param to return (when more than one exist):

○ The first out parameter receiving an assignment.
● The last out parameter receiving an assignment.
○ The first out parameter declared.
○ The last out parameter declared.

Limit comment suggested name to: 48 ch. Add line breaks: None

☑ Extract method should be available even when the selection contains return statements.
Name of variable to hold Return flag value: shouldReturn

☑ Enable Extract Method Smart Paste. ☐ Create shared/static method if possible.

Level: Expert Language: CSharp

Inline Method

The **Inline Method** refactoring lets you move the method's body into the body of its callers and remove the method declaration, if needed. **Inline Method** is helpful when you have a group of methods that seem badly factored. You can inline them all into one big method, and then re-extract the methods.

Another reason to use **Inline Method** is that sometimes you do come across a method in which the body is as clear as the name. Or, you refactor the body of the code into something that is just as clear as the name. When this happens, you should then get rid of the method.

There are two versions of this refactoring:

- **Inline Method**
- **Inline Method (and delete)**

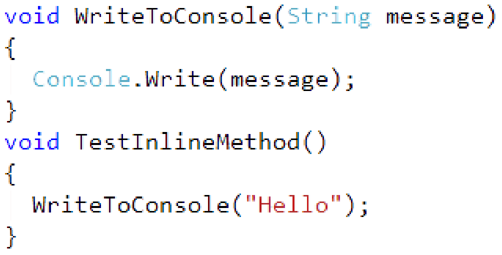
The difference between the two is that the second one removes the source method, if appropriate.

The refactoring is available either on the source method name or the calling side.

Opposite

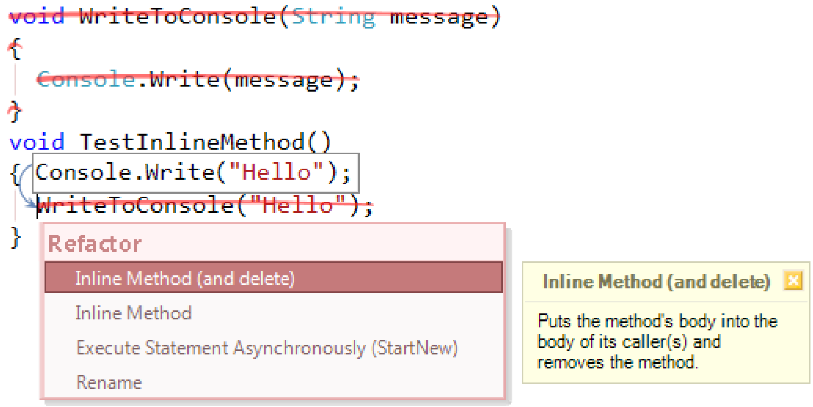
The opposite of this refactoring is the Extract Method refactoring.

Source sample code



```
void WriteToConsole(String message)
{
  Console.Write(message);
}
void TestInlineMethod()
{
  WriteToConsole("Hello");
}
```

Refactoring preview

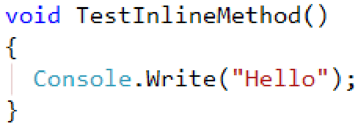


```
~~void WriteToConsole(String message)~~
~~{~~
  ~~Console.Write(message);~~
~~}~~
void TestInlineMethod()
{ Console.Write("Hello");
  ~~WriteToConsole("Hello");~~
}
```

Refactor
Inline Method (and delete)
Inline Method
Execute Statement Asynchronously (StartNew)
Rename
Inline Method (and delete)

Puts the method's body into the body of its caller(s) and removes the method.

Result



```
void TestInlineMethod()
{
  Console.Write("Hello");
}
```

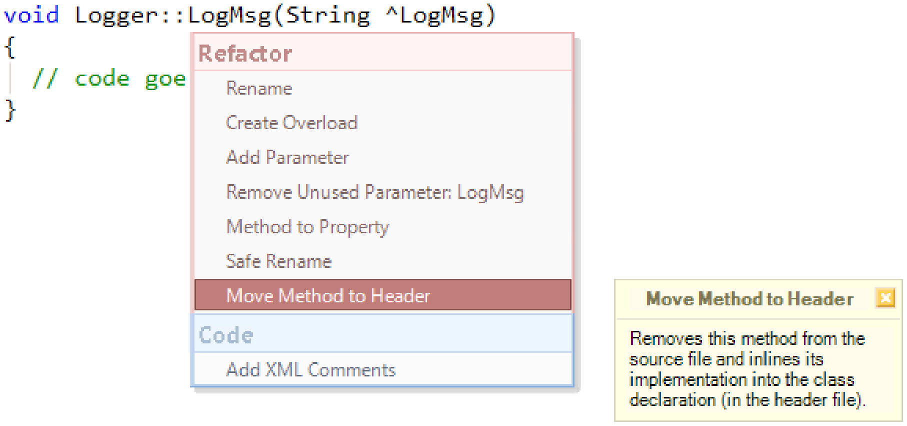
Moving methods to source file and back to the header

There are two refactorings specific to the C++ language:

- **Move Method to Source File**
- **Move Method to Header**

The first one moves a method from a source file into the class declaration that is inside the header file. The second one moves the method's implementation to a source file leaving the declaration in the header file. Having these refactorings makes it much easier to move method bodies between the header and source files.

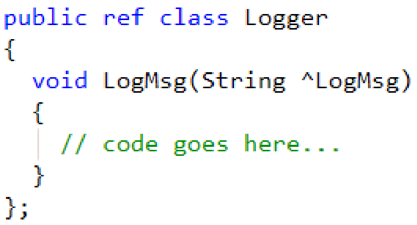
Here are some simple samples:



```
void Logger::LogMsg(String ^LogMsg)
{
  // code goe
}
```

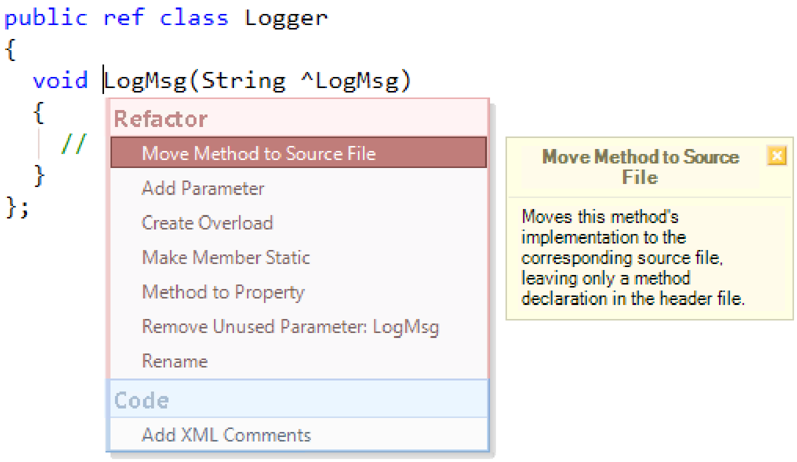
Removes this method from the source file and inlines its implementation into the class declaration (in the header file).

Result:



```
public ref class Logger
{
  void LogMsg(String ^LogMsg)
  {
    // code goes here...
  }
};
```

And vice versa:



```
public ref class Logger
{
  void LogMsg(String ^LogMsg)
  {
    //
  }
};
```

Moves this method's implementation to the corresponding source file, leaving only a method declaration in the header file.

Refactorings for simplifying of parallel computing development

Many personal computers and workstations have two or four cores that enable multiple threads to be executed simultaneously. .Net Framework ver. 4.0 has been introduced a standardized and simplified way for creating robust, scalable and reliable multi-threaded applications. The parallel programming extension of .NET 4.0 allows the developer to create applications that exploit the power of multi-core and multi-processor computers. The flexible thread APIs of the extension are much simpler to use and more powerful than standard .NET threads.

The extension implements the concept of automatic dynamic parallelization of applications. It provides both ease-of-use and scalability in development of parallel programs. The concept is naturally integrated into .NET Framework by means of templatized classes (introduced in C# with generics) that encapsulate all low-level details such as threading, synchronization, scheduling, load balancing, etc., which makes the extension a powerful tool for implementing high-performance parallel applications.

Refactor! Pro provides several parallel computing refactorings that can help you to parallelize your code to distribute work across multiple processors. Here they are:

- **Convert to Parallel**

Converts the code to run in parallel.

- **Execute Statements in Parallel**

Executes the selected independent statements in parallel.

- **Execute Statements in Serial**

Moves the child statements out from the *Parallel.Invoke* call and executes them serially.

- **Execute Statements Asynchronously (FromAsync)**

Passes the appropriate *BeginXXX* and *EndXXX* methods (corresponding to this statement) to the *FromAsync* method of *Task.Factory*, launching an asynchronous operation and returning a handle to the associated *Task*.

- **Execute Statements Asynchronously (StartNew)**

Passes the current statement to the *StartNew* method of *Task.Factory*, launching an asynchronous operation and returning a handle to the associated *Task*.

All refactorings work in both *CSharp* and *Visual Basic* languages. In this article we will review the first three refactorings, and see how they help to speed-up the code in a real sample. There are several parallel programming methods we are going to use:

- Parallel.For
- AsParallel
- Parallel.Invoke
- and a standard serial calculation method to compare with.

Consider that we have the function that determines whether the specified number is prime:

```
public class Prime
{
  /// <summary>
  /// Determines whether the specified number is prime.
  /// </summary>
  /// <param name=»number»>The number.</param>
  /// <returns>
  /// true if the specified number is prime; otherwise, false.
  /// </returns>
  public static bool IsPrime(int number)
  {
    for (int d = 2; d <= number / 2; d++)
    {
       if (number % d == 0)
         return false;
    }
    return number > 1;
  }
}
```

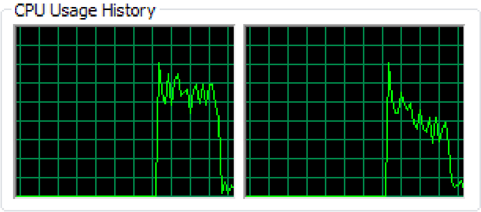
And the standard function that returns the list of primes in the specified limit:

```
static List Find(int limit)
{
  var result = new List();

  for (int i = 2; i < limit; i++)
  {
    if (Prime.IsPrime(i))
      result.Add(i);
  }

  return result;
}
```

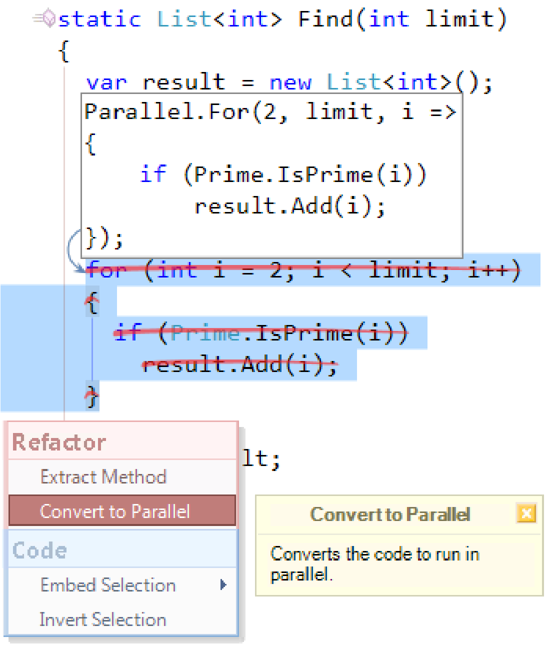
Using the System.Diagnostics.Stopwatch class, we will count how much time it takes for finding all primes within the limit. The limit will be 300000:

```
static void Main(string[] args)
{
  Stopwatch stopwatch = Stopwatch.StartNew();
  Find(٣٠٠٠٠٠);
  stopwatch.Stop();
  Console.WriteLine(«Time passed: « +
                            stopwatch.ElapsedMilliseconds + « ms.»);
}
```

The result of the standard code run is 18328 ms. Average CPU usage is 52%. Here's the CPU usage history of the Intel(R) Core(TM)2 DUO CPU:

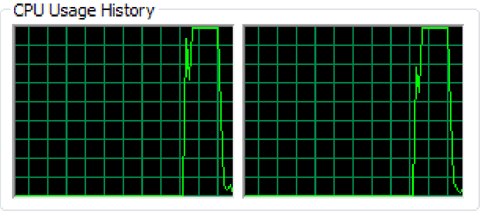


Now, let’s use the **Convert to Parallel refactoring** and improve the code:



```
static List<int> Find(int limit)
{
    var result = new List<int>();
    Parallel.For(2, limit, i =>
    {
        if (Prime.IsPrime(i))
            result.Add(i);
    });
    ~~for (int i = 2; i < limit; i++)~~
    ~~{~~
        ~~if (Prime.IsPrime(i))~~
            ~~result.Add(i);~~
    ~~}~~
                                lt;
```

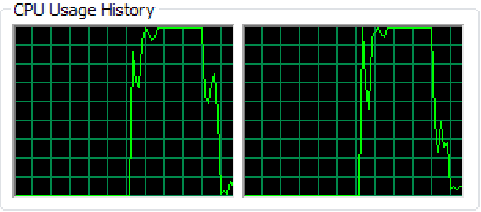
The result of the *Parallel.For* code run is ٩٧٢٧ ms. Average CPU usage is ٪١٠٠. CPU usage history:



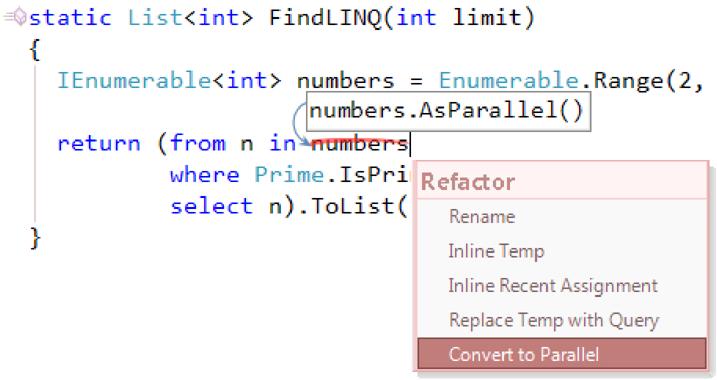
Let’s change the Find method to use LINQ instead:

```
static List FindLINQ(int limit)
{
  IEnumerable numbers = Enumerable.Range(٢, limit - ١);
  return (from n in numbers
          where Prime.IsPrime(n)
          select n).ToList();
}
```

Result time: 19555 ms. Average CPU usage: 100%. CPU usage history:

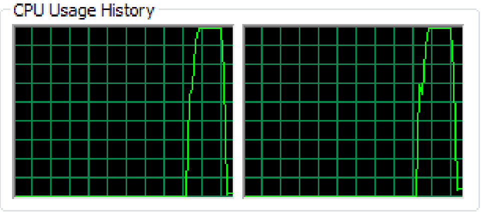


Once again, using the same **Convert to Parallel** refactoring change the code:



```
static List<int> FindLINQ(int limit)
{
    IEnumerable<int> numbers = Enumerable.Range(2, limit - 1);
                               numbers.AsParallel()
    return (from n in numbers
            where Prime.IsPri
            select n).ToList(
}
```

Result time: 10111 ms. Average CPU usage: 100%. CPU usage history:



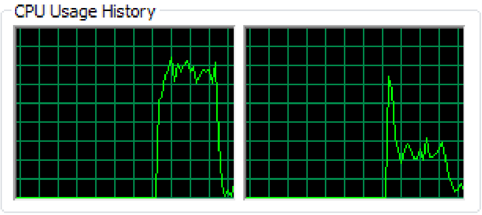
Now let's use the *Parallel.Invoke* method. To use this method, we'll split the limited number into ١٠ parts and make calculation in parallel. An additional helper method is needed in this case:

```
static void CheckRange(List result, int n, int limit, int factor)
{
  for (int i = n * (limit / factor); i < (n + 1) * (limit / factor); i++)
  {
    if (Prime.IsPrime(i))
      result.Add(i);
  }
}
```

The standard code will look like this:

```
static List  FindRange(int limit)
{
  var result = new List();
  CheckRange(result, ٠, limit, ١٠);
  CheckRange(result, ١, limit, ١٠);
  CheckRange(result, ٢, limit, ١٠);
  CheckRange(result, ٣, limit, ١٠);
  CheckRange(result, ٤, limit, ١٠);
  CheckRange(result, ٥, limit, ١٠);
  CheckRange(result, ٦, limit, ١٠);
  CheckRange(result, ٧, limit, ١٠);
  CheckRange(result, ٨, limit, ١٠);
  CheckRange(result, ٩, limit, ١٠);
  return result;
}
```

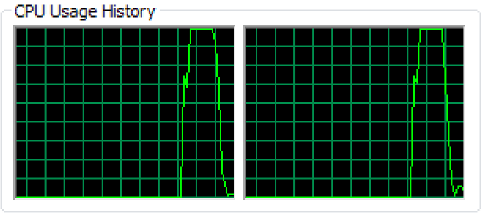
Result time without parallel optimization: 17816 ms. CPU usage: 52%. CPU usage history:



Now, use the **Execute Statements in Parallel** refactoring: The *Parallel.Invoke* method takes an array of delegates as an argument. The new code will look like this:

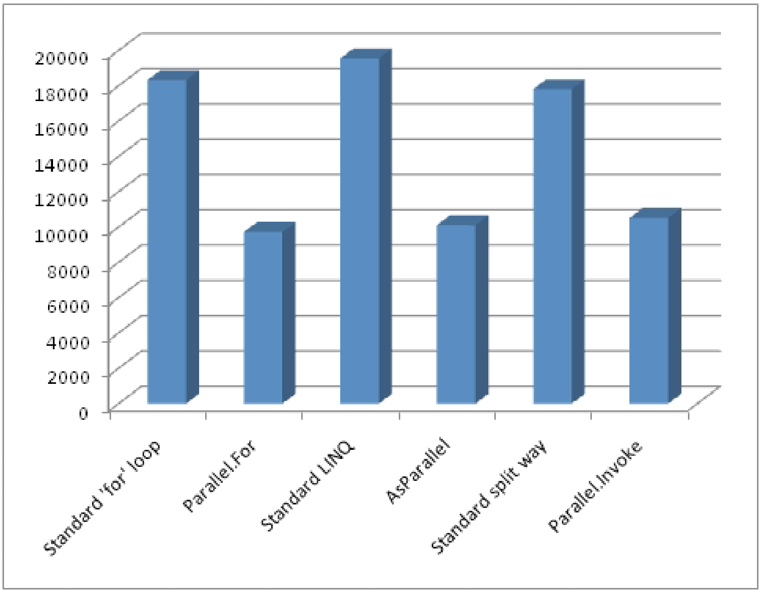
```
static List FindRangeParallel(int limit)
{
  var result = new List();
  Parallel.Invoke(
    () => CheckRange(result, ٠, limit, ١٠),
    () => CheckRange(result, ١, limit, ١٠),
    () => CheckRange(result, ٢, limit, ١٠),
    () => CheckRange(result, ٣, limit, ١٠),
    () => CheckRange(result, ٤, limit, ١٠),
    () => CheckRange(result, ٥, limit, ١٠),
    () => CheckRange(result, ٦, limit, ١٠),
    () => CheckRange(result, ٧, limit, ١٠),
    () => CheckRange(result, ٨, limit, ١٠),
    () => CheckRange(result, ٩, limit, ١٠));
  return result;
}
```

Result time: 10530 ms. CPU usage: 100%. CPU usage history:

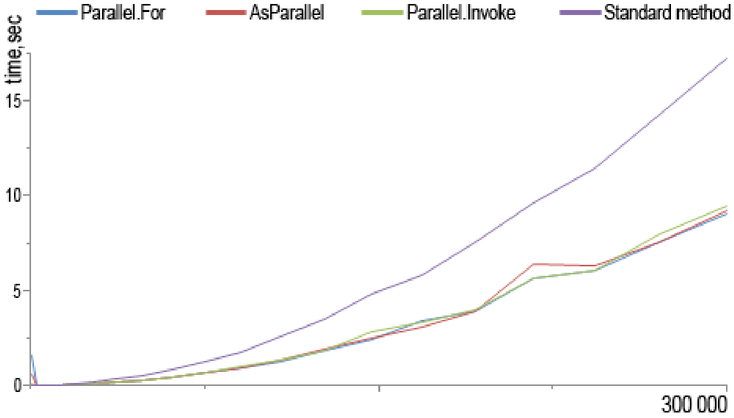


The **Execute Statements in Serial** refactoring is an the opposite of the **Execute Statements in Parallel**, which moves all code from the *Parallel.Invoke* call and executes it as usual (serially).

Here’s the comparison chart of the different parallel methods used in comparison to the standard serial methods:



Time chart:



As you can see, using parallel computing almost doubles the speed of the code, even when the number of calculations is small. Parallel refactorings from Refactor! Pro make it easy to improve the speed of your code.

Refactorings to execute statements asynchronously

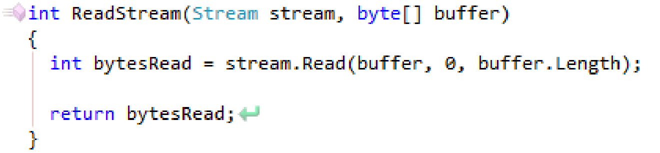
In continuing with the ‘Refactorings for simplifying of the .NET 4.0 parallel computing development‘ thread, let’s review the additional refactoring for executing statements asynchronously called in the same manner – **Execute Statement Asynchronously**. The refactoring is available in two versions:

- **Execute Statement Asynchronously (FromAsync)**
- **Execute Statement Asynchronously (StartNew)**

The FromAsync version of the refactoring launches an asynchronous operation passing a pair of the corresponding

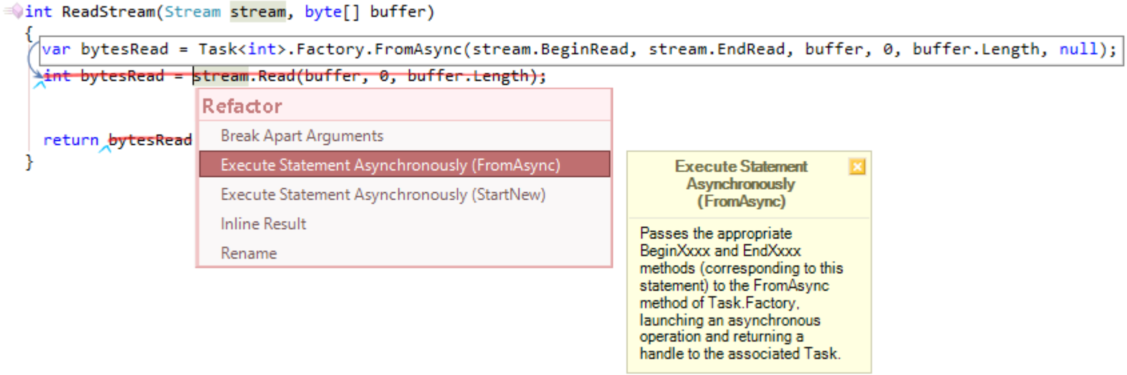
BeginXxx and EndXxx methods of the selected statement to the FromAsync call of the Task.Factory. The FromAsync call provides a convenient mechanism which uses the BeginXxx and EndXxx methods of the Asynchronous Programming Model (APM) to create a Task.

The TaskFactory simplifies creation and startup of simple background tasks dramatically. For example, consider that we have a Stream and a byte buffer, and we want to read from that stream into the buffer. Synchronously, we could do the following:



```
int ReadStream(Stream stream, byte[] buffer)
{
  int bytesRead = stream.Read(buffer, 0, buffer.Length);

  return bytesRead;
}
```

The **Execute Statement Asynchronously (FromAsync) refactoring** will allow us to convert this code into the FromAsync call:



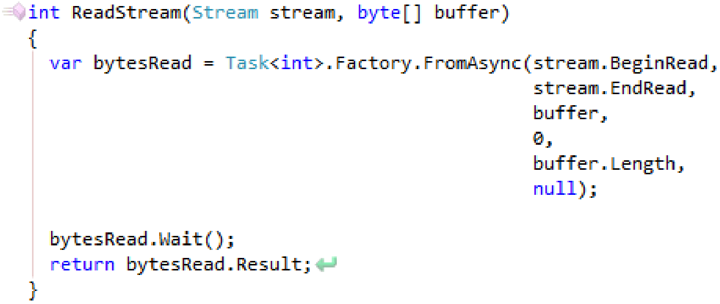
```
int ReadStream(Stream stream, byte[] buffer)
{
  var bytesRead = Task<int>.Factory.FromAsync(stream.BeginRead, stream.EndRead, buffer, 0, buffer.Length, null);
  ~~int bytesRead = stream.Read(buffer, 0, buffer.Length);~~


  return ~~bytesRead~~
}
```

Refactor
Break Apart Arguments
Execute Statement Asynchronously (FromAsync)
Execute Statement Asynchronously (StartNew)
Inline Result
Rename

Execute Statement Asynchronously (FromAsync)

Passes the appropriate BeginXxxx and EndXxxx methods (corresponding to this statement) to the FromAsync method of Task.Factory, launching an asynchronous operation and returning a handle to the associated Task.

resulting in the following one:

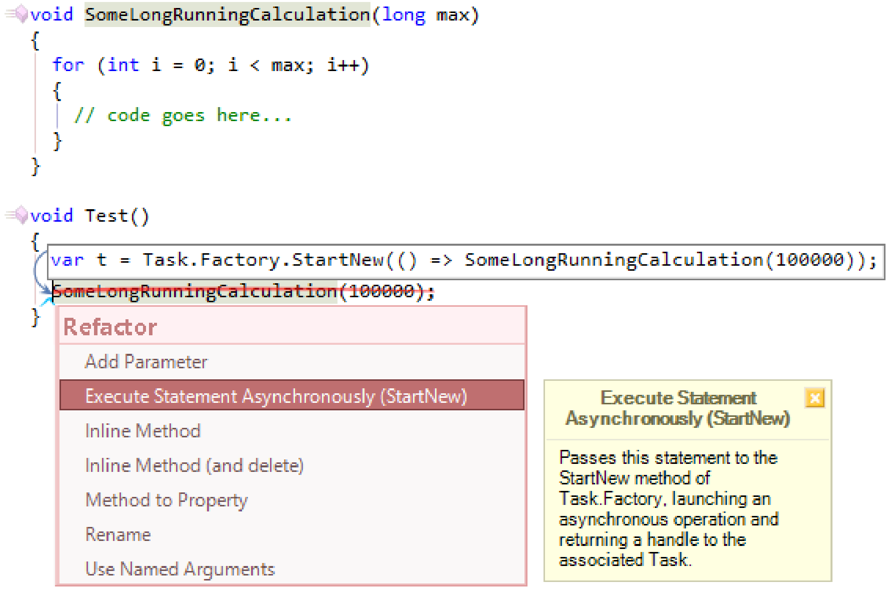


```
int ReadStream(Stream stream, byte[] buffer)
{
  var bytesRead = Task<int>.Factory.FromAsync(stream.BeginRead,
                                              stream.EndRead,
                                              buffer,
                                              0,
                                              buffer.Length,
                                              null);

  bytesRead.Wait();
  return bytesRead.Result;
}
```

The FromAsync call returns a handle to the associated Task. The resulting Task object will, by default, be executed on a thread pool thread. Combine this support with the ability to do Task.WaitAll, Task.WaitAny, Task.Factory.ContinueWhenAll, and Task.Factory.ContinueWhenAny, and we can achieve a very useful synchronization functionality having just a little code.

The second version of the refactoring (**StartNew**) launches an asynchronous operation by passing the current statement to the StartNew method of Task.Factory. For instance, using StartNew is more efficient than creating and starting tasks manually, because the StartNew method not only starts a new task but also applies the required synchronization to the process. If you construct a task using the Task's constructor, you then pay this synchronization cost when calling the Start method, because you need to have protection against the chance that another thread is concurrently calling Start. However, if we use the Task.Factory.StartNew, we know that the task will have already been scheduled and started by the time we get the Task reference returned, which means that it is no longer possible for threads to race to call Start, because every call to Start will fail. In the case of using StartNew, we can avoid the additional synchronization and take a faster and easier approach for scheduling a Task.

The **Execute Statement Asynchronously (StartNew)** refactoring can be applied on any statement:

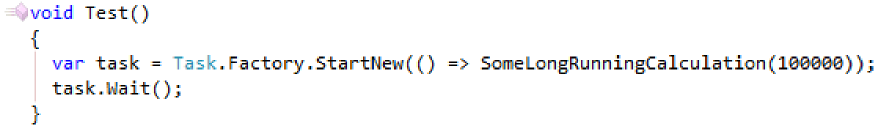


```
void SomeLongRunningCalculation(long max)
{
  for (int i = 0; i < max; i++)
  {
    // code goes here...
  }
}

void Test()
{
  var t = Task.Factory.StartNew(() => SomeLongRunningCalculation(100000));
  ~~SomeLongRunningCalculation(100000);~~
}
```

Execute Statement Asynchronously (StartNew)

Passes this statement to the StartNew method of Task.Factory, launching an asynchronous operation and returning a handle to the associated Task.

which will result in:



```
void Test()
{
  var task = Task.Factory.StartNew(() => SomeLongRunningCalculation(100000));
  task.Wait();
}
```

The difference between the StartNew and FromAsync methods is that using the latter for the things like Stream (i.e., when API offers the appropriate BeginXxx and EndXxx methods), you will actually end up using the async I/O underneath the covers, e.g., I/O Completion Ports in Windows which provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. The FromAsync method in this case provides the best way to achieve I/O scalability, and is more efficient than blocking multiple CPU threads doing a synchronous operation using the StartNew method.

Refactoring properties and fields

Refactorings specific to auto-implemented properties

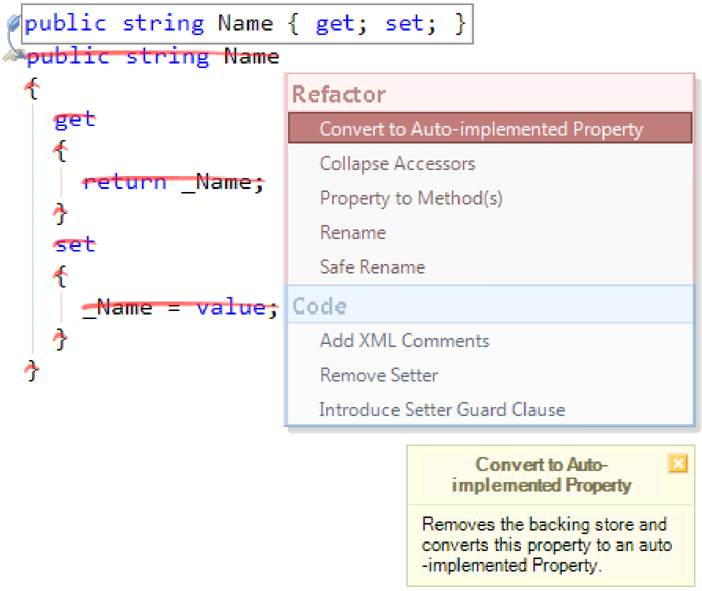
Auto-implemented properties enable you to quickly specify a property without having to write logic for the property accessors. The auto-property's logic and the field serving as a backing store are automatically generated by the compiler. Such properties appeared in C# version 3.0 and Visual Basic version 10.0.

Refactor! Pro has several refactorings that allow you to convert "full" properties with a backing store field into auto-implemented properties and vice versa. They are **Convert to Auto-implemented Property** and **Create Backing Store**.

The **Convert to Auto-implemented Property** refactoring is available for properties that encapsulate a private field without additional logic inside accessors, for example:

```
private int _Name;
public int Name
{
   get
   {
      return _Name;
   }
   set
   {
      _Name = value;
   }
}
```

The refactoring is available here showing you a preview hint of the resulting code:



```
public string Name { get; set; }
public string Name
{
   get
   {
      return _Name;
   }
   set
   {
      _Name = value;
   }
}
```

Refactor
Convert to Auto-implemented Property
Collapse Accessors
Property to Method(s)
Rename
Safe Rename
Code
Add XML Comments
Remove Setter
Introduce Setter Guard Clause
Convert to Auto-implemented Property

Removes the backing store and converts this property to an auto -implemented Property.

After the refactoring is executed, you get a simple auto-implemented property:

```
public string Name { get; set; }
```

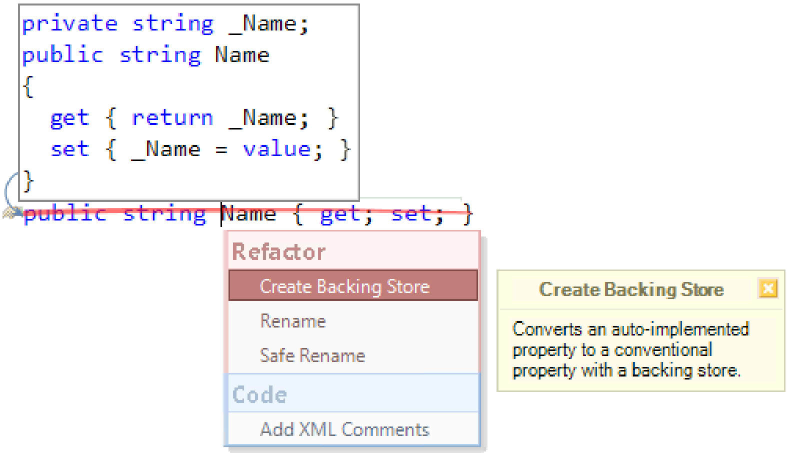
The refactoring performs the following actions automatically:

- removes the backing store field for the property;
- replaces the original property with an auto-implemented property;
- replaces all field references to references to the new property;

There's a second version of the refactoring called **Convert to Auto-implemented Property (convert all)** which does absolutely the same actions, but for all properties in the current type declaration that can to be converted into auto-properties.

The opposite refactoring, called **Create Backing Store**, converts an auto-implemented property into a full property with a field serving as a backing store. This can be useful if you'd like to add additional logic for one of the property's accessors. Or, if you're going to prevent an auto-implemented property from being serialized, because auto-properties don't support the [NonSerializable] attribute.

Here's a sample preview hint of the **Create Backing Store** refactoring:



```
private string _Name;
public string Name
{
  get { return _Name; }
  set { _Name = value; }
}
```

~~public string Name { get; set; }~~

Converts an auto-implemented property to a conventional property with a backing store.

Encapsulation refactorings

As we know, Encapsulation is an important object-oriented programming concept. It is also known as a data hiding mechanism. Encapsulation enables a group of properties, methods and other members to be considered a single unit or object. The idea of encapsulation is that an object's internal data should not be directly accessible from an object instance. With correct encapsulation, a developer does not need to understand how the class actually operates in order to communicate with it via its publicly available methods and properties.

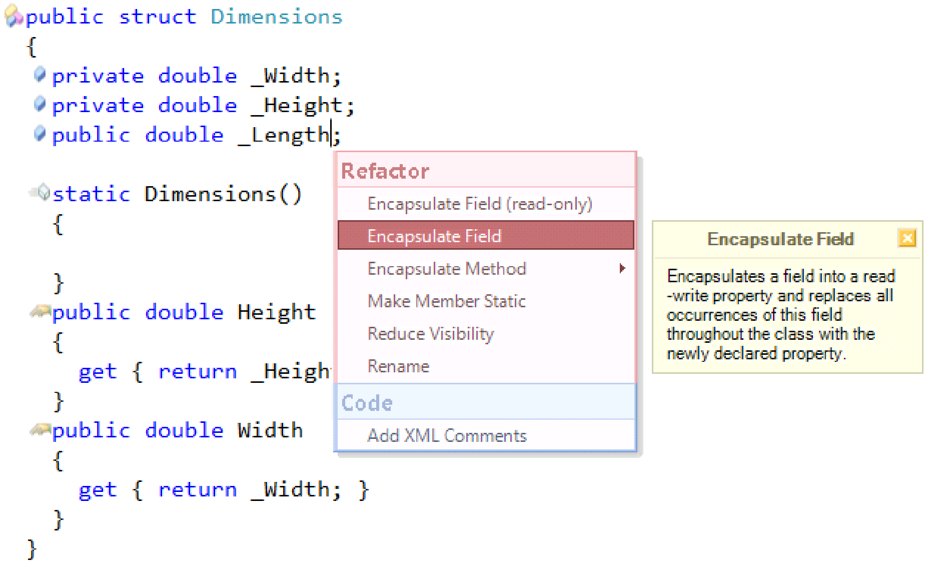
CodeRush has several refactorings that allow you to encapsulate fields and its members easily. They are:

- **Encapsulate Field**
- **Encapsulate Field (read-only)**
- **Encapsulate Method**

- **Encapsulate Property**
- **Encapsulate Event**

The **Encapsulate Field** refactoring encapsulates a field into a read-write property and replaces all occurrences of this field with the newly declared property. If a field is non-private, the refactoring will change it into a private field (encapsulation). The new property is given the same visibility as the original field (e.g., public, internal, etc.).

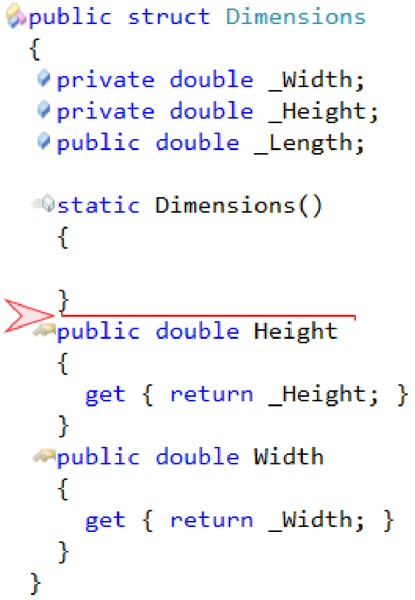
Apply the refactoring to an active field to encapsulate it:



```
public struct Dimensions
{
  private double _Width;
  private double _Height;
  public double _Length;

  static Dimensions()
  {

  }
  public double Height
  {
    get { return _Heigh
  }
  public double Width
  {
    get { return _Width; }
  }
}
```

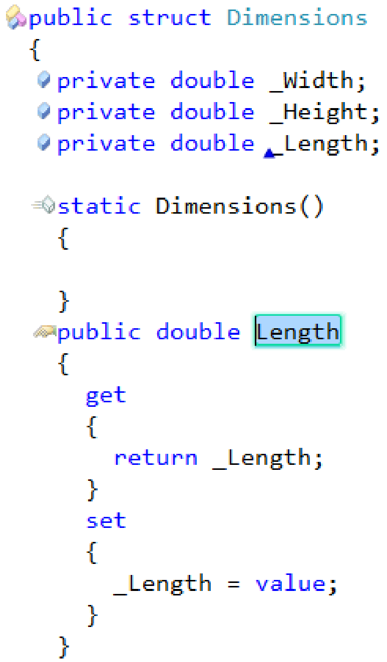
Encapsulates a field into a read -write property and replaces all occurrences of this field throughout the class with the newly declared property.

You can choose the target location of the property with the help of a target picker:



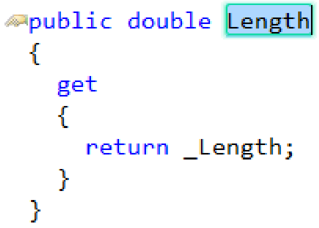
```
public struct Dimensions
{
  private double _Width;
  private double _Height;
  public double _Length;

  static Dimensions()
  {

  }
  public double Height
  {
    get { return _Height; }
  }
  public double Width
  {
    get { return _Width; }
  }
}
```

The resulting property looks like this:



```
public struct Dimensions
{
  private double _Width;
  private double _Height;
  private double _Length;

  static Dimensions()
  {

  }
  public double Length
  {
    get
    {
      return _Length;
    }
    set
    {
      _Length = value;
    }
  }
```

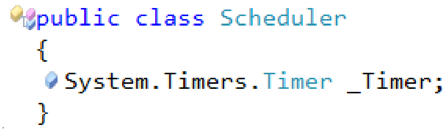
The second **Encapsulate Field** (**read-only**) refactoring is absolutely the same as Encapsulate Field, but creates a read-only property without writing access:



```
public double Length
{
  get
  {
    return _Length;
  }
}
```

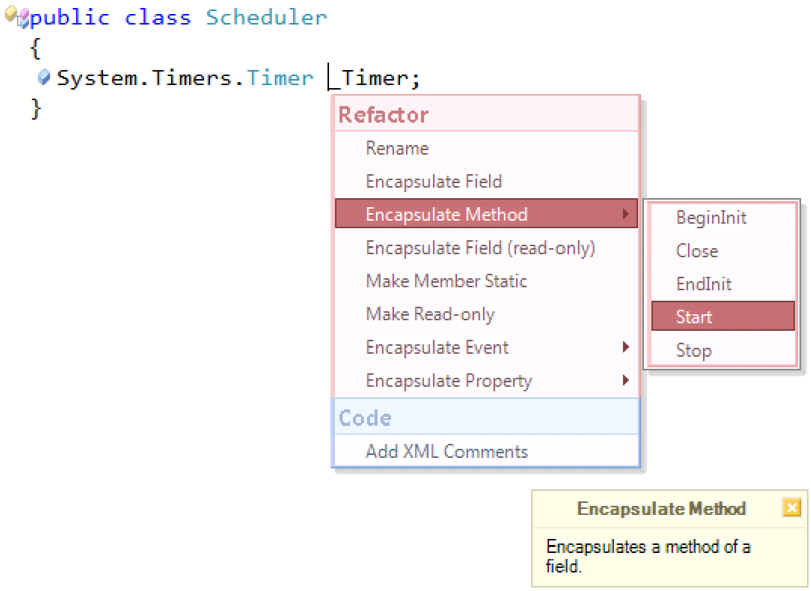
The other refactorings are also available on a field. They allow you to encapsulate a property, a method or an event of a field. These refactorings create new accessing members to members of an instance of a field.

Consider we have the following field declaration:



```
public class Scheduler
{
  System.Timers.Timer _Timer;
}
```

These refactorings allows you to choose which member you would like to encapsulate. For instance, the **Encapsulate Method** refactoring:

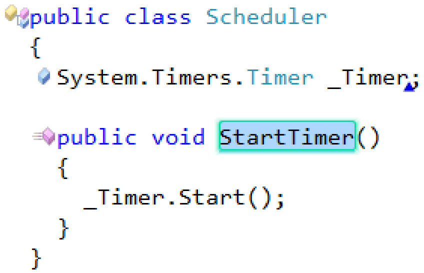


```
public class Scheduler
{
 System.Timers.Timer _Timer;
}
```

Encapsulate Method

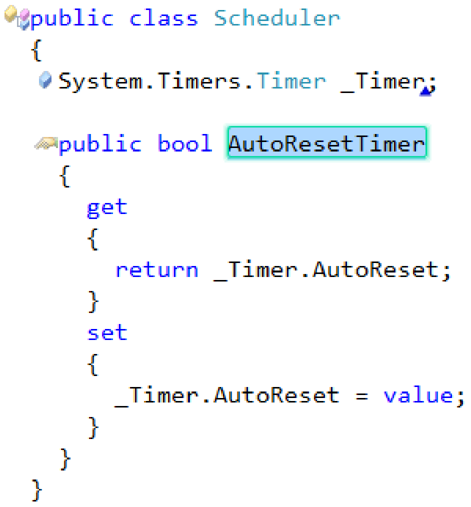
Encapsulates a method of a field.

The refactoring will produce the corresponding code once applied:



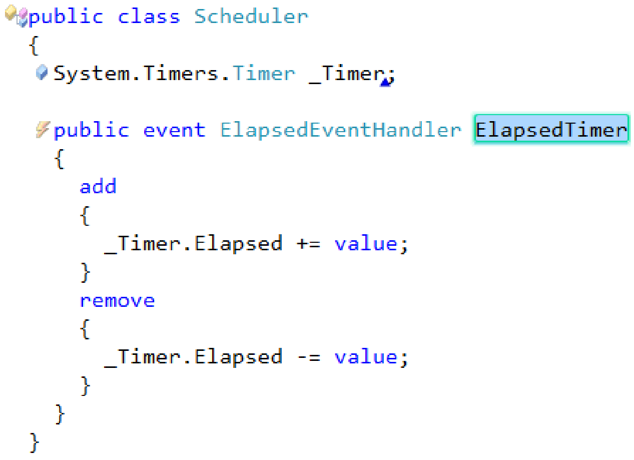
```
public class Scheduler
{
 System.Timers.Timer _Timer;

 public void StartTimer()
  {
    _Timer.Start();
  }
}
```

Here is the result of **Encapsulate Property** for the AutoReset property:



```
public class Scheduler
{
 System.Timers.Timer _Timer;

 public bool AutoResetTimer
  {
    get
    {
      return _Timer.AutoReset;
    }
    set
    {
      _Timer.AutoReset = value;
    }
  }
}
```

And finally, the result of the **Encapsulate Event** for the XXX event:



```
public class Scheduler
{
  System.Timers.Timer _Timer;

  public event ElapsedEventHandler ElapsedTimer
  {
    add
    {
      _Timer.Elapsed += value;
    }
    remove
    {
      _Timer.Elapsed -= value;
    }
  }
}
```

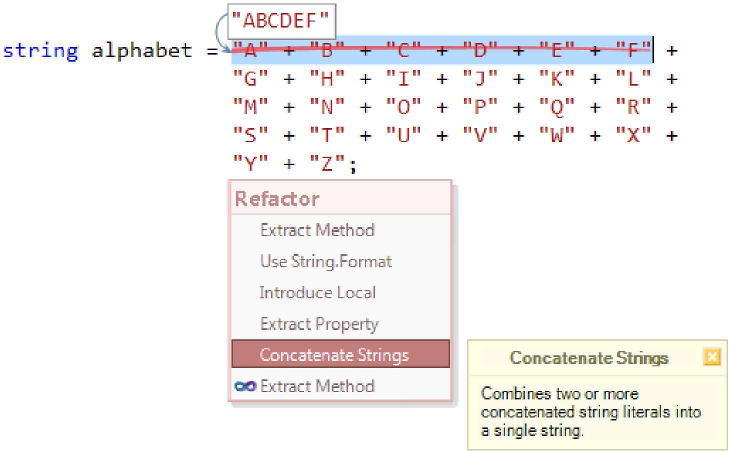
All refactorings allow you to choose the target location and drop a marker to easily move back at the original location where the refactoring was applied.

Refactoring strings

Concatenating and splitting strings

Concatenation is the process of appending one string to the end of another string. A string is basically a sequence of Unicode characters. An important property of strings is that they are read-only (immutable). Once a string has been created, it cannot be changed. So, when a string is updated, the .NET framework actually discards the original string and creates a new string. In other words, the concatenation of strings is creating an entirely new string, allocating enough memory for everything, copying all the data from the existing values of all concatenated strings and then copying the data from each string. As the string grows, the amount of data it has to copy each time grows too. Having already concatenated strings allows you to avoid such problems.

DevExpress Refactor! Pro has a simple refactoring to combine strings called **Concatenate Strings**. This refactoring concatenates all selected strings into a single string. Here’s its preview hint for the selected strings:

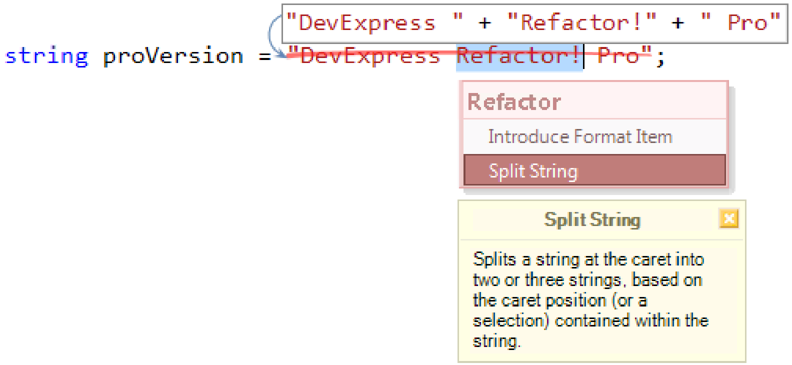


```
                     "ABCDEF"
string alphabet = ~~"A" + "B" + "C" + "D" + "E" + "F"~~ +
                  "G" + "H" + "I" + "J" + "K" + "L" +
                  "M" + "N" + "O" + "P" + "Q" + "R" +
                  "S" + "T" + "U" + "V" + "W" + "X" +
                  "Y" + "Z";
```

Combines two or more concatenated string literals into a single string.

When concatenating verbatim and non-verbatim strings, they are combined into a common format depending on the type of a first string. If the first string is verbatim, then the resulting string will be also verbatim, otherwise not.

The opposite of this refactoring is the **Split String** refactoring. This refactoring is also very simple. It can break the active string into two or three pieces depending on the editor caret position and the selection state. If you select a part of a string, this part will be extracted into a new string and combined with two other start and end parts:

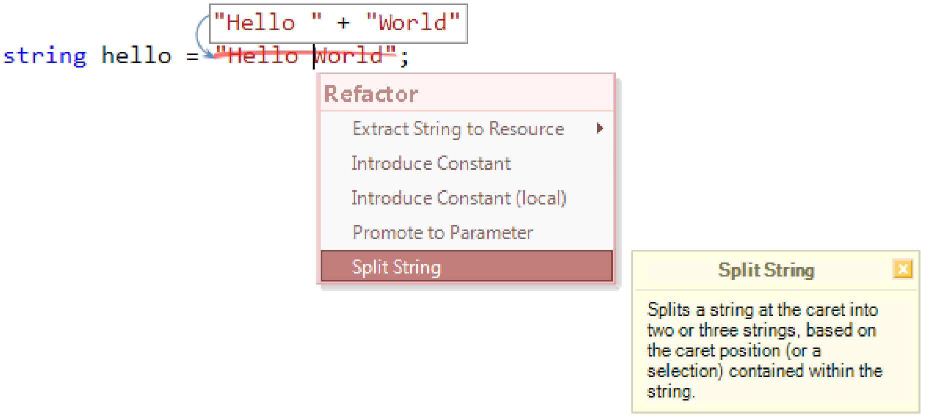


```
"DevExpress " + "Refactor!" + " Pro"
string proVersion = "DevExpress Refactor! Pro";
```

Split String

Splits a string at the caret into two or three strings, based on the caret position (or a selection) contained within the string.

If there’s no string selection, only two parts are produced by breaking the string into two pieces:

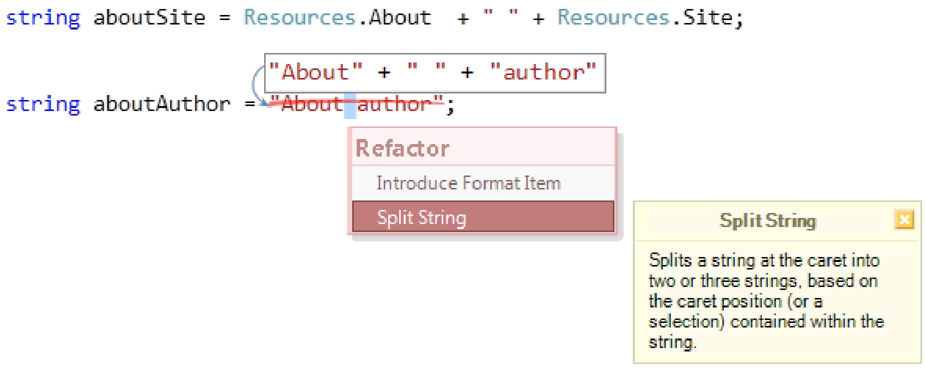


```
"Hello " + "World"
string hello = "Hello World";
```

Split String

Splits a string at the caret into two or three strings, based on the caret position (or a selection) contained within the string.

The refactoring may be useful for localization purposes, especially if some part of a string is already localized, but the other one is not yet localized. Here, you can split the string:

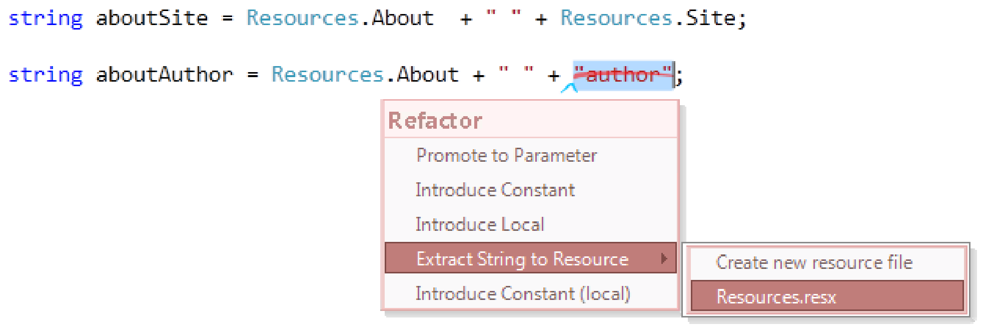


```
string aboutSite = Resources.About  + " " + Resources.Site;

"About" + " " + "author"
string aboutAuthor = "About author";
```

Split String

Splits a string at the caret into two or three strings, based on the caret position (or a selection) contained within the string.

and replace its parts to localized versions:



```
string aboutSite = Resources.About  + " " + Resources.Site;

string aboutAuthor = Resources.About + " " + "author";
```

Bear in mind that if the editor caret is positioned at an unbreakable sequence of characters, such as a string format item (e.g. “{0}”), an escape character (e.g. “\r”, “\n”), a unicode character code (e.g. “\uFFFF”), or the editor caret is at the very beginning or an end of a string, the refactoring will not be available in these cases.

Also, refer to other refactorings that work with strings.

String.Format-specific refactorings

Refactor! Pro provides several refactorings to create and organize the .NET String.Format call. The String.Format call is a static method that receives a string that specifies where the following arguments should be inserted, and how they should be formatted. You can specify the display options for each argument individually using the String.Format call.

These refactorings are:

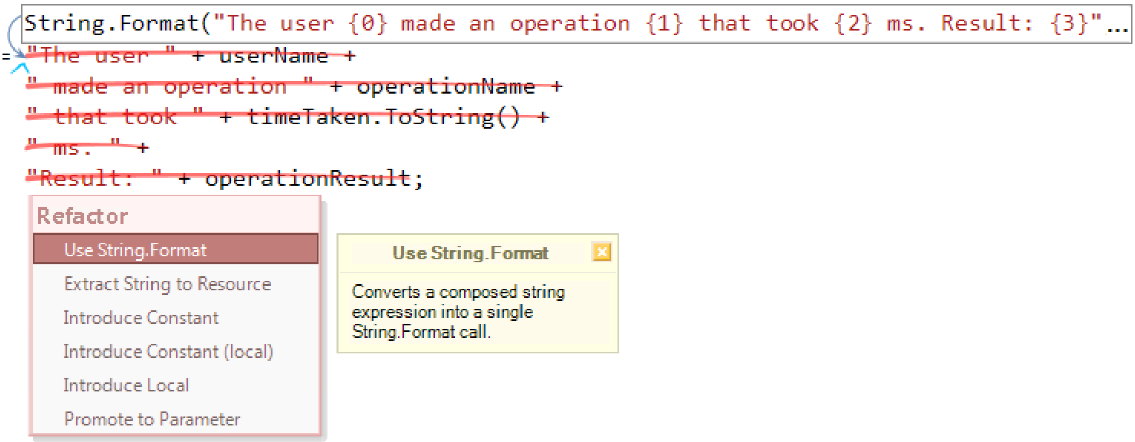
- **Use String.Format**
- **Introduce Format Item**
- **Inline Format Item**
- **Remove Redundant Call**

The **Use String.Format** refactoring converts a composed string expression into a single String.Format call. This refactoring makes string concatenations much more readable and less error prone. It also makes it much easier to update your string, prepare its arguments for localization and add more parameters to it. However, on the other hand, the String.Format call may decrease performance a bit.

Imagine the following code:

```
String auditMsg = "The user " + userName +
                  " made an operation " + operationName +
                  " that took " + timeTaken.ToString() +
                  " ms. " +
                  "Result: " + operationResult;
```

Before you apply the refactoring, you can see the resulting code (*click the image to enlarge*):



```
String.Format("The user {0} made an operation {1} that took {2} ms. Result: {3}"...
= ~~"The user " + userName +~~
  ~~" made an operation " + operationName +~~
  ~~" that took " + timeTaken.ToString() +~~
  ~~" ms. " +~~
  ~~"Result: " + operationResult~~;
```

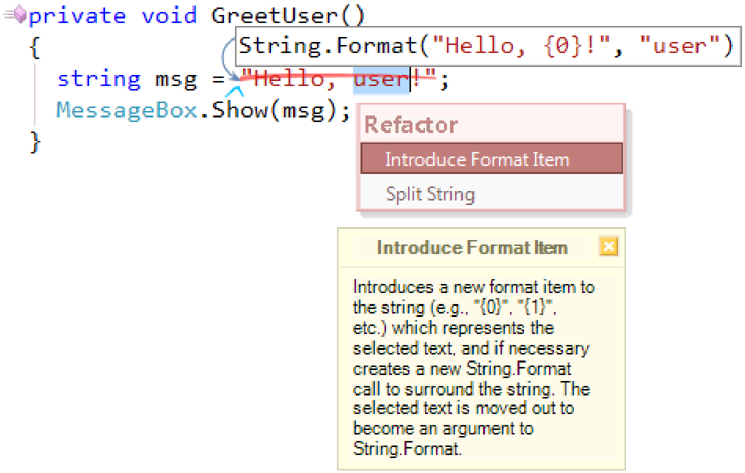
Converts a composed string expression into a single String.Format call.

After we apply the **Use String.Format** refactoring, we will get the following code:

```
String.Format("The user {0} made an operation {1} that took {2} ms. Result: {3}",
              userName,
              operationName,
              timeTaken,
              operationResult);
```

This seems to be much more readable…

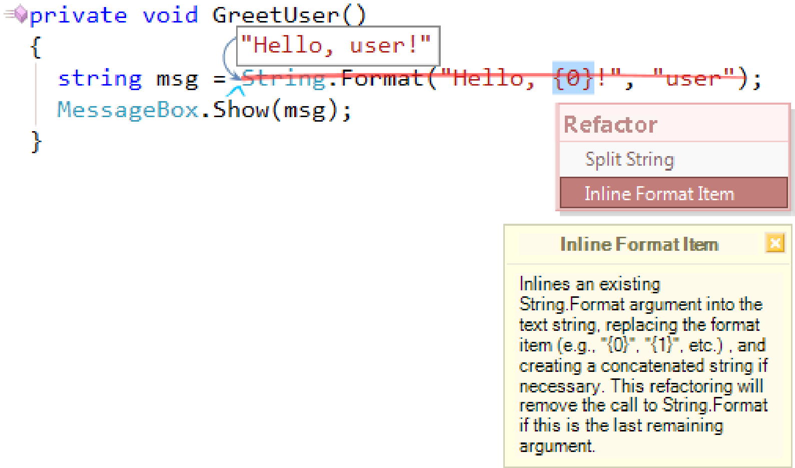
The **Introduce Format Item** refactoring adds a new format item to the string (e.g., “{١}” ,“{٠}”, etc.) which represents the selected text, and if necessary creates a new String.Format call to surround the string. The selected text is extracted to become an argument to the String.Format call and an argument placeholder is left in the format string. To apply the refactoring, select a part of a string that you would like to use as an argument to the String.Format call. Here is the sample:



```
private void GreetUser()
{                         String.Format("Hello, {0}!", "user")
   string msg = ~~"Hello, user!"~~;
   MessageBox.Show(msg);
}
```

Introduces a new format item to the string (e.g., "{0}", "{1}", etc.) which represents the selected text, and if necessary creates a new String.Format call to surround the string. The selected text is moved out to become an argument to String.Format.

The **Inline Format Item** refactoring is the opposite of the **Introduce Format Item**. It inlines an existing String. Format argument into the text string, replacing the format item (e.g., “{0}”, “{1}”, etc.) , and creating a concatenated string, if necessary:



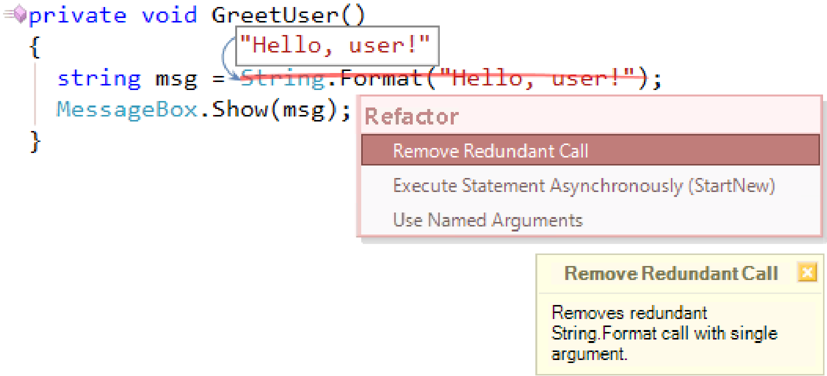
```
private void GreetUser()
{
                      "Hello, user!"
  string msg = String.Format("Hello, {0}!", "user");
  MessageBox.Show(msg);
}
```

Inline Format Item

Inlines an existing String.Format argument into the text string, replacing the format item (e.g., "{0}", "{1}", etc.) , and creating a concatenated string if necessary. This refactoring will remove the call to String.Format if this is the last remaining argument.

This refactoring will remove the call to String.Format if it is applied in the last remaining argument:

A part of the **Remove Redundant Call** refactoring functionality allows you to remove the redundant String.Format call with a single string parameter:



```
private void GreetUser()
{
                      "Hello, user!"
  string msg = String.Format("Hello, user!");
  MessageBox.Show(msg);
}
```

Remove Redundant Call

Removes redundant String.Format call with single argument.

As always, the preview hint in every refactoring will show you the resulting code before you apply it. This allows you to see what code will be produced after the refactoring is performed.

Also, refer to other refactorings that work with strings.

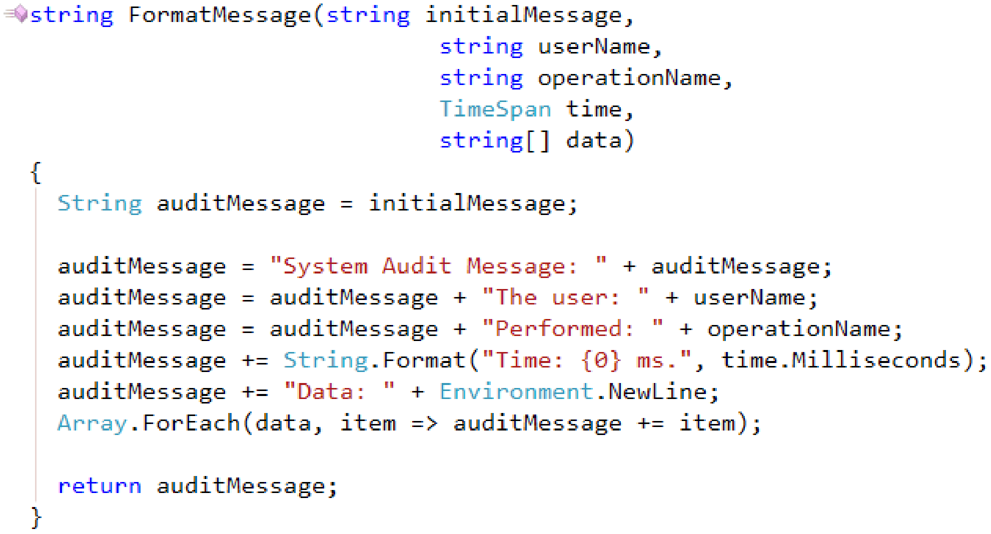
Use StringBuilder

In addition to the refactorings that work with the concatenated strings and the String.Format call, there is another useful refactoring called **Use StringBuilder**. This refactoring replaces the string concatenation operations with corresponding methods of the StringBuilder class.

Using the StringBuilder class may improve the performance of the string concatenation operations. The performance of these operations depends on how often a memory allocation occurs. A usual string concatenation operation always allocates memory, and a StringBuilder concatenation operation , on the other hand, only allocates memory if the size of the StringBuilder object's buffer is not enough to store the new data. This means that the String class is preferable for the concatenation operation of a fixed number of strings. In that case, the individual concatenation operations might even be combined into a single operation by the compiler. A StringBuilder object is preferable for a concatenation operation if an arbitrary number of strings are concatenated. For example, when you don't have a loop, generally you should avoid StringBuilder. There is a lot of logic in the StringBuilder class that will be slow on small string operations and it can make your code more cumbersome.

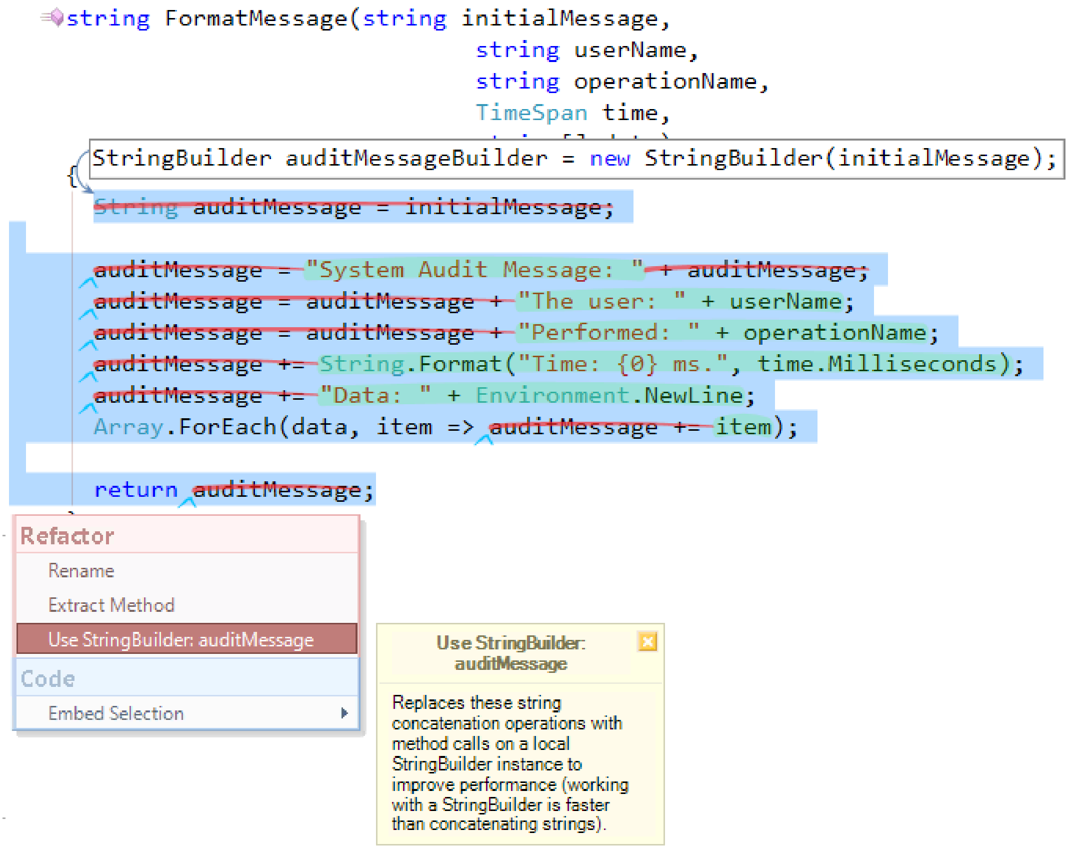
However, there is another benefit of using the StringBuilder class in regard to the memory consumption. The more temporary objects created, the more often the .NET Garbage Collector runs. The StringBuilder class creates fewer temporary objects than the usual string concatenation operations, and therefore adds less memory pressure.

The **Use StringBuilder** refactoring uses the appropriate *Append, Insert, AppendFormat* calls of the StringBuilder class when it is applied on a selected piece of strings concatenations code. It can also recreate an instance of the StringBuilder, if necessary. For example, consider the following code:



```
string FormatMessage(string initialMessage,
                            string userName,
                            string operationName,
                            TimeSpan time,
                            string[] data)
{
  String auditMessage = initialMessage;

  auditMessage = "System Audit Message: " + auditMessage;
  auditMessage = auditMessage + "The user: " + userName;
  auditMessage = auditMessage + "Performed: " + operationName;
  auditMessage += String.Format("Time: {0} ms.", time.Milliseconds);
  auditMessage += "Data: " + Environment.NewLine;
  Array.ForEach(data, item => auditMessage += item);

  return auditMessage;
}
```

This method formats a new system audit message initializing it to an initial message, adding a new caption and additional data, such as a user name, operation name and the time taken by an operation. The code can be optimized to use the StringBuilder class. To apply it, select the string operations and the string declaration you would like to convert (you will see the preview hint of the resulting code changes):



```
string FormatMessage(string initialMessage,
                     string userName,
                     string operationName,
                     TimeSpan time,
    StringBuilder auditMessageBuilder = new StringBuilder(initialMessage);
{
    ~~String auditMessage = initialMessage;~~

    ~~auditMessage = ~~"System Audit Message: "~~ + auditMessage;~~
    ~~auditMessage = auditMessage + ~~"The user: " + userName;
    ~~auditMessage = auditMessage + ~~"Performed: " + operationName;
    ~~auditMessage +=~~ String.Format("Time: {0} ms.", time.Milliseconds);
    ~~auditMessage +=~~ "Data: " + Environment.NewLine;
    Array.ForEach(data, item => ~~auditMessage +=~~ item);

    return ~~auditMessage~~;
```

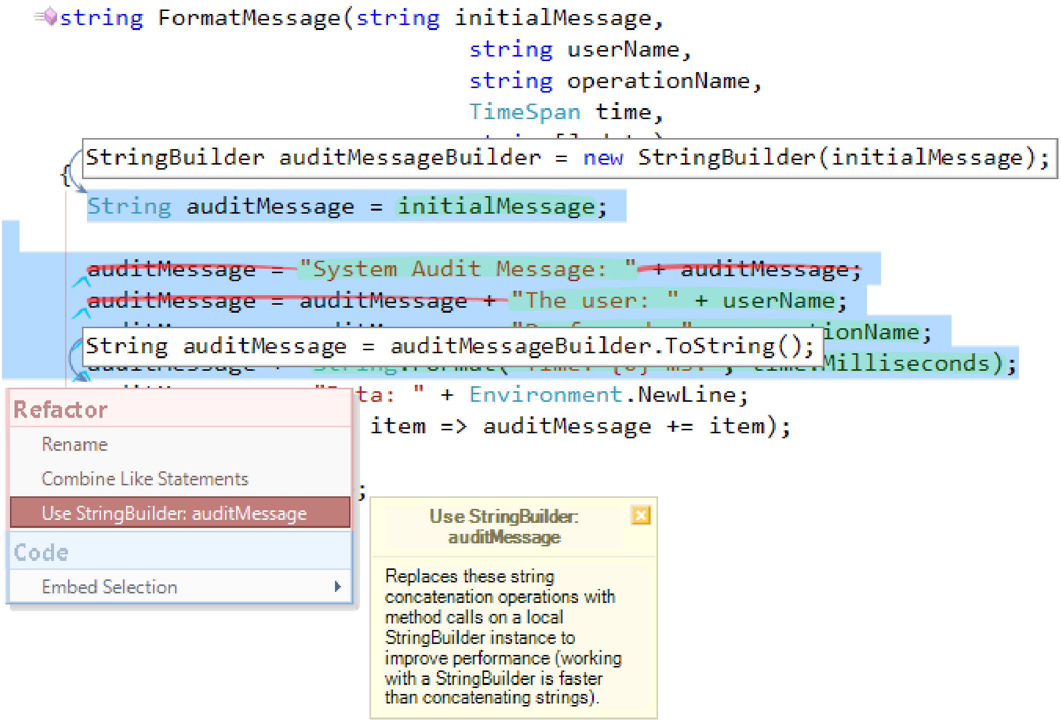
Use StringBuilder: auditMessage

Replaces these string concatenation operations with method calls on a local StringBuilder instance to improve performance (working with a StringBuilder is faster than concatenating strings).

The resulting code will look like this:

```
string FormatMessage(string initialMessage,
                     string userName,
                     string operationName,
                     TimeSpan time,
                     string[] data)
{
    StringBuilder auditMessageBuilder = new StringBuilder(initialMessage);

    auditMessageBuilder.Insert(0, "System Audit Message: ");
    auditMessageBuilder.Append("The user: " + userName);
    auditMessageBuilder.Append("Performed: " + operationName);
    auditMessageBuilder.AppendFormat("Time: {0} ms.", time.Milliseconds);
    auditMessageBuilder.Append("Data: " + Environment.NewLine);
    Array.ForEach(data, item => auditMessageBuilder.Append(item));

    return auditMessageBuilder.ToString();
}
```

If you select a part of the code like this:



```
string FormatMessage(string initialMessage,
                             string userName,
                             string operationName,
                             TimeSpan time,
StringBuilder auditMessageBuilder = new StringBuilder(initialMessage);
{
  String auditMessage = initialMessage;

  auditMessage = "System Audit Message: " + auditMessage;
  auditMessage = auditMessage + "The user: " + userName;
                                                  ionName;
String auditMessage = auditMessageBuilder.ToString();
                                                  Milliseconds);
                         ta: " + Environment.NewLine;
                          item => auditMessage += item);
```

Use StringBuilder: auditMessage

Replaces these string concatenation operations with method calls on a local StringBuilder instance to improve performance (working with a StringBuilder is faster than concatenating strings).

You will get the corresponding code for the selected block of code:

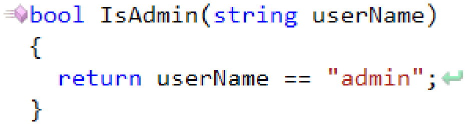
```
string FormatMessage(string initialMessage,
                             string userName,
                             string operationName,
                             TimeSpan time,
                             string[] data)
{
  StringBuilder auditMessageBuilder = new StringBuilder(initialMessage);

  auditMessageBuilder.Insert(0, "System Audit Message: ");
  auditMessageBuilder.Append("The user: " + userName);
  auditMessageBuilder.Append("Performed: " + operationName);
  auditMessageBuilder.AppendFormat("Time: {0} ms.", time.Milliseconds);
  String auditMessage = auditMessageBuilder.ToString();
  auditMessage += "Data: " + Environment.NewLine;
  Array.ForEach(data, item => auditMessage += item);

  return auditMessage;
}
```

This code is much easier to read and maintain. It will perform much faster, especially if the data array passed to the method has many items.

Also, refer to other refactorings that work with strings.

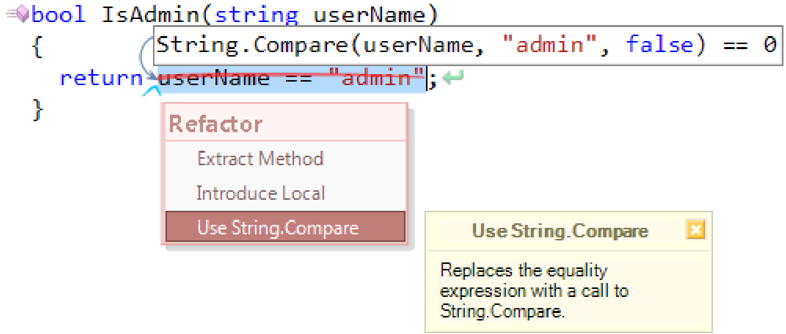
Use String.Compare

The **Use String.Compare** refactoring shipped in DevExpress Refactor! Pro allows you to convert a usual string equality comparison (==) into a more flexible *Compare* call on the *System.String* class. Consider the following code:



```
bool IsAdmin(string userName)
{
  return userName == "admin";
}
```

The refactoring will show you resulting code preview hint:



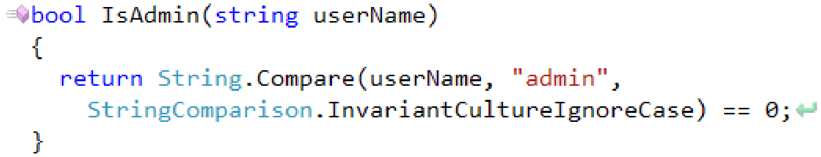
```
bool IsAdmin(string userName)
{          String.Compare(userName, "admin", false) == 0
  return userName == "admin";
}
```

Use String.Compare

Replaces the equality expression with a call to String.Compare.

The created *String.Compare* compares two specified string objects, taking into account their case by passing the boolean "false" as an argument (*IgnoreCase*), and returns an integer that indicates their relative position in the sort order, indicating whether one string is ordered before another in alphabetical order, or whether it is ordered after or is equivalent.

You can also modify the *String.Compare* call to take a *StringComparison* enumeration argument which lets you specify culture-insensitive or case-insensitive comparisons. Only *String.Compare* call allows you to specify a CultureInfo and perform comparisons using a culture other than the default (current) culture. This argument not only allows you to define the exact comparison behavior you intended, but also makes your code more readable for other developers:



```
bool IsAdmin(string userName)
{
  return String.Compare(userName, "admin",
    StringComparison.InvariantCultureIgnoreCase) == 0;
}
```

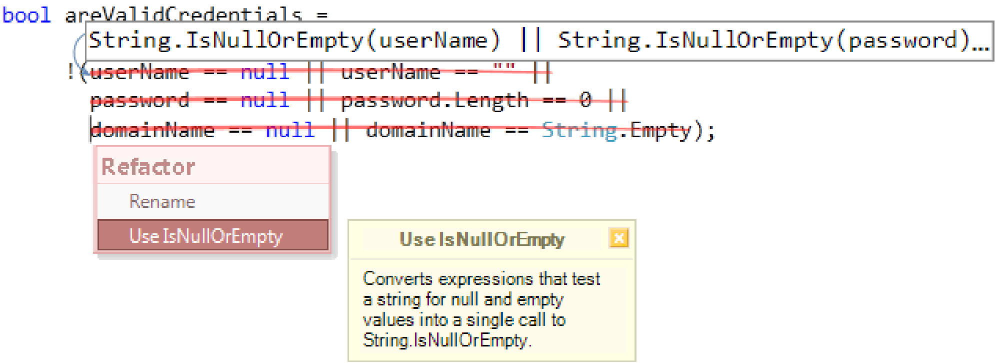
Also, refer to other refactorings that work with strings.

Use IsNullOrEmpty

When checking that a user has provided a valid input, you will often want to ensure that it is not null or empty. Strings are reference types and can be equal to a null value like any other reference type. Strings can also be empty, meaning their values equal "" and they have zero length. The *IsNullOrEmpty* call of the *System.String* class indicates whether the given string is a null or an empty ("") string. Before this call appeared in the .NET Framework, we were checking strings as follows (for example):

```
bool areValidCredentials =
     !(userName == null || userName == "" ||
       password == null || password.Length == 0 ||
       domainName == null || domainName == String.Empty);
```

This code checks the userName, password and domainName strings that are null and uses different checks for the string contents. All these checks can be replaced with the corresponding *IsNullOrEmpty* call in by applying the **Use IsNullOrEmpty** refactoring. The **Use IsNullOrEmpty** refactoring shipped inDevExpress Refactor! Pro allows you to convert the string checking code into the appropriate call in a single step:



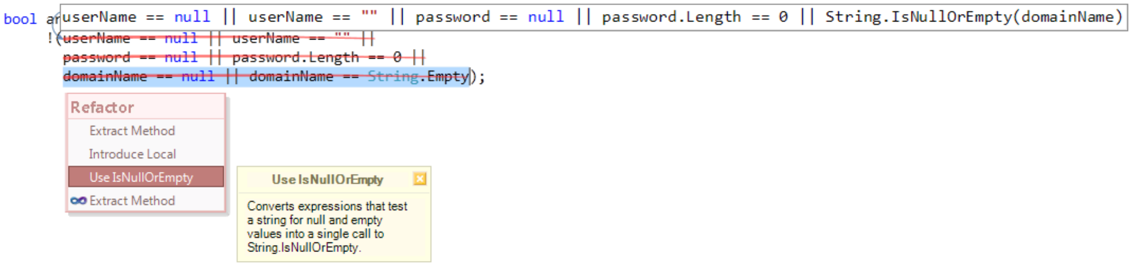
```
bool areValidCredentials =
      String.IsNullOrEmpty(userName) || String.IsNullOrEmpty(password)...
     !(~~userName == null || userName == "" ||~~
       ~~password == null || password.Length == 0 ||~~
       ~~domainName == null || domainName == String.Empty~~);
```

Converts expressions that test a string for null and empty values into a single call to String.IsNullOrEmpty.

The result produced is now much more readable:

```
bool areValidCredentials =
     !(String.IsNullOrEmpty(userName) ||
       String.IsNullOrEmpty(password) ||
       String.IsNullOrEmpty(domainName));
```

If you don’t want to replace all checks, you can select those to be converted:



```
bool ar userName == null || userName == "" || password == null || password.Length == 0 || String.IsNullOrEmpty(domainName)
     !(~~userName == null || userName == "" ||~~
       ~~password == null || password.Length == 0 ||~~
       ~~domainName == null || domainName == String.Empty~~);
```

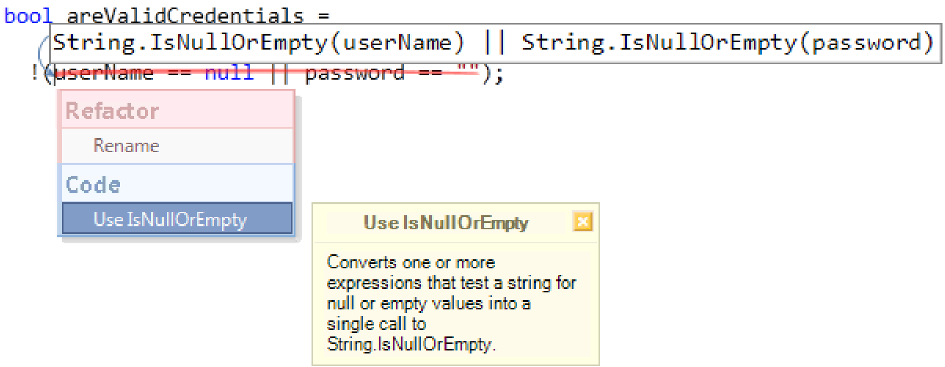
Converts expressions that test a string for null and empty values into a single call to String.IsNullOrEmpty.

Only selected checks will be replaced by the refactoring:

```
bool areValidCredentials =
     !(userName == null || userName == "" ||
       password == null || password.Length == 0 ||
       String.IsNullOrEmpty(domainName));
```

The refactoring requires both checks (for null and empty) to occur at once. Strings can be either equal to the null literal, meaning they don’t reference any data, or empty, meaning they reference a character buffer that is zero characters long. *IsNullOrEmpty* call detects both of these conditions at the same time.

However, this refactoring also has a second version that does not require thes both conditions simultaneously. This version is implemented as a code provider operation which may change the program behavior. The **Use IsNullOrEmpty** code provider allows you to replace one of the checks to the*String.IsNullOrEmpty* call:



```
bool areValidCredentials =
    String.IsNullOrEmpty(userName) || String.IsNullOrEmpty(password)
   !(~~userName == null || password == ""~~);
```

Use IsNullOrEmpty

Converts one or more expressions that test a string for null or empty values into a single call to String.IsNullOrEmpty.

Use this code provider when you are sure that you need both checks made for a string, otherwise it may change the logic of the code – e.g. when you cannot have a null string value, but you can have an empty string value.

Also, refer to other refactorings that work with strings.

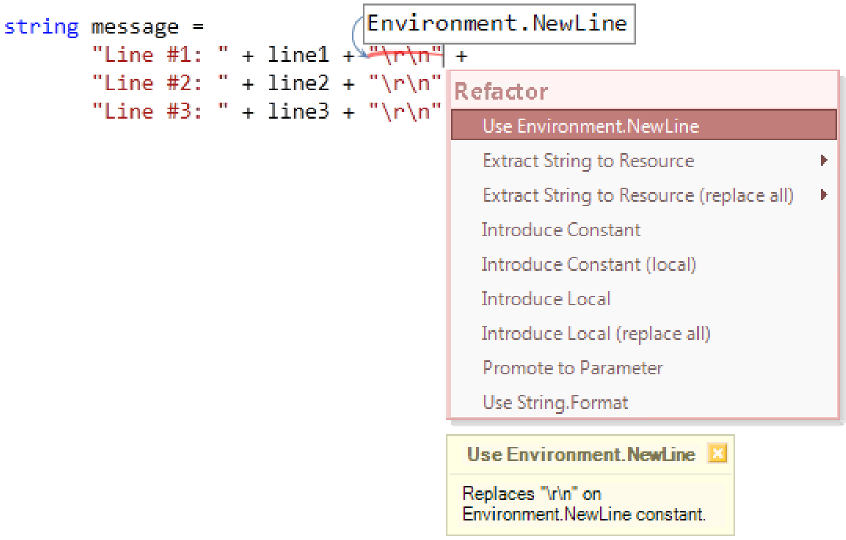
Use Environment.NewLine

The **Use Environment.NewLine** refactoring is one of the simplest refactoring shipped in DevExpress Refactor! Pro which improves code portability. This refactoring replaces the “\r\n” string with the value of the *Environment.NewLine* property reference. The *Environment.NewLine* is a static string property from the *System* namespace that is tied to the current executing environment (platform). It returns a valid “line feed/carriage return” string that corresponds to the current operating system, for example: “\r\n” for Windows platforms, or a string containing just a line feed (“\n“) for Unix platforms.

You may use this refactoring to improve the portability of the code and readability. Consider the following code:

```
string message =
       "Line #1: " + line1 + "\r\n" +
       "Line #2: " + line2 + "\r\n" +
       "Line #3: " + line3 + "\r\n";
```

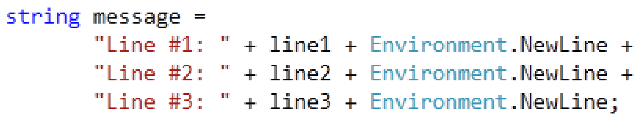
Applying this refactoring will help you replace all “\r\n” occurrences:



```
string message =
        "Line #1: " + line1 + "\r\n" +
        "Line #2: " + line2 + "\r\n"
        "Line #3: " + line3 + "\r\n"
```

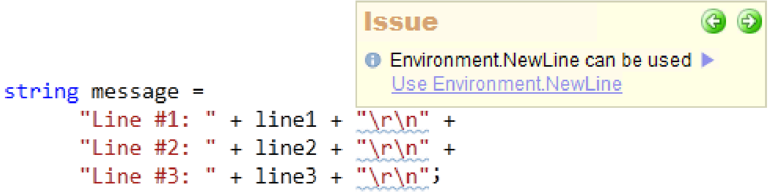
Replaces "\r\n" on Environment.NewLine constant.

Resulting code:



```
string message =
        "Line #1: " + line1 + Environment.NewLine +
        "Line #2: " + line2 + Environment.NewLine +
        "Line #3: " + line3 + Environment.NewLine;
```

Note if you have CodeRush Pro installed and the Code Issues feature turned on, you will see the special highlighting for the cases where the **Use Environment.NewLine** refactoring can be applied:



```
string message =
       "Line #1: " + line1 + "\r\n" +
       "Line #2: " + line2 + "\r\n" +
       "Line #3: " + line3 + "\r\n";
```

To learn more about the "**Environment.NewLine can be used**" code issue, please read the appropriate topic.

Also, refer to other refactorings that work with strings.

Refactorings for types and type casting

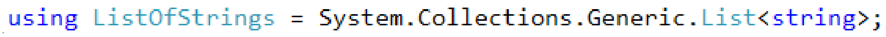
Introduce Alias

The **Introduce Alias** refactoring creates an identifier that serves as an alias for a namespace or type within the active source file. The alias directive can provide much neater and organized code. When several namespaces contain a matching class name, alias can also help to avoid a name conflict (ambiguous reference) without using the fully qualified type names, which improves overall code readability.

The refactoring allows you to choose whether to create a namespace or type alias via a sub menu. There are two variations of this refactoring – a simple **Introduce Alias** which creates an alias for the current type or namespace reference, and the **Introduce Alias (replace all)**, which replaces all occurrences of the type or namespace reference in the entire source file.

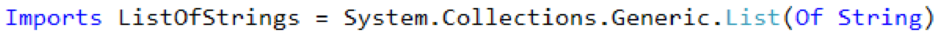
Once refactoring is performed, you can rename the newly created alias, because names of the alias references are linked together. When you are done with renaming, simply press the *Enter* key to commit your change:

CSharp:



```
using ListOfStrings = System.Collections.Generic.List<string>;
```

Visual Basic:



```
Imports ListOfStrings = System.Collections.Generic.List(Of String)
```

The opposite refactorings are **Inline Alias** and **Replace with Alias**.

Move Type to File

The **Move Type To File** refactoring allows users to move a given class, structure, interface, enumeration or delegate to a separate file. It is available when there are two or more types in the current source file.

Refactoring moves a type with a name that differs from the file name to a new source file. All comments, attributes, and XML doc comments relating to the type are moved with the type. The file will be located in the same folder as the current file and automatically added to the project. The name of the new file is based on the type's name on which refactoring is applied.

To perform the **Move Type to File** refactoring, place the editor caret at the type name, or a visibility keyword if specified. If visibility keyword is not specified than the first keyword range is used instead (e.g. 'class' (*CSharp*) or 'Class' (*Visual Basic*)).

After refactoring is executed, all namespace references in the new source file are automatically optimized, i.e. all unused namespace references are removed using the existing Optimize Namspace References refactoring.

Optimize Namespace References

The **Optimize Namespace References** refactoring removes namespace references (*usings* (in C#) or *Imports* (in VB)) that are not needed in the active source file, either because they were added automatically and not used, or because the code that did require them has been removed or relocated. Also, it can automatically sort used references after refactoring is performed: alphabetically, by length or not sort. An additional sorting option is to move all *System* references at the top of the list (file). The refactoring improves code readability.

On the other hand, Visual Studio already has a similar built-in feature called *Organize Using Statements*, available via right-clicking the context menu in the code editor. The "*Organize Usings*" feature provides a way to sort and remove using and alias declarations.

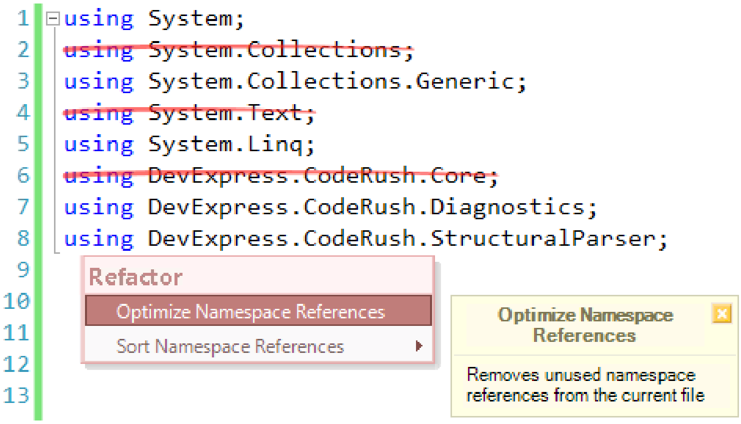
Let's compare these two features: Refactor! **Optimize Namespace References** and Visual Studio Organize Usings:

	VS Organize Usings	**Optimize Namespace References**
References sorting	✓	✓
Sorting methods	Alphabetically, Place 'System' directives first when sorting usings	Alphabetically, By Length, Move System References at the top

Aliases support	✓	✓ (Partially – unused aliases are removed, but not sorted)
Option to leave favorite references	✗	✓ (*)
Languages support	C# (For VB support you may install *Productivity Power Tools*)	C#, VB, C++
Visual Studio support	VS2008 and up	VS2005 and up
Works across solution	✗ (Macros can be written as a workaround)	✗ (DXCore plug-in can be written as a workaround)

(*) This option is available since release 11.1.

The **Optimize Namespace References** refactoring is shipped in Refactor! Pro. It has a preview hint as most of other refactorings, so you are able to see what references are going to be removed:

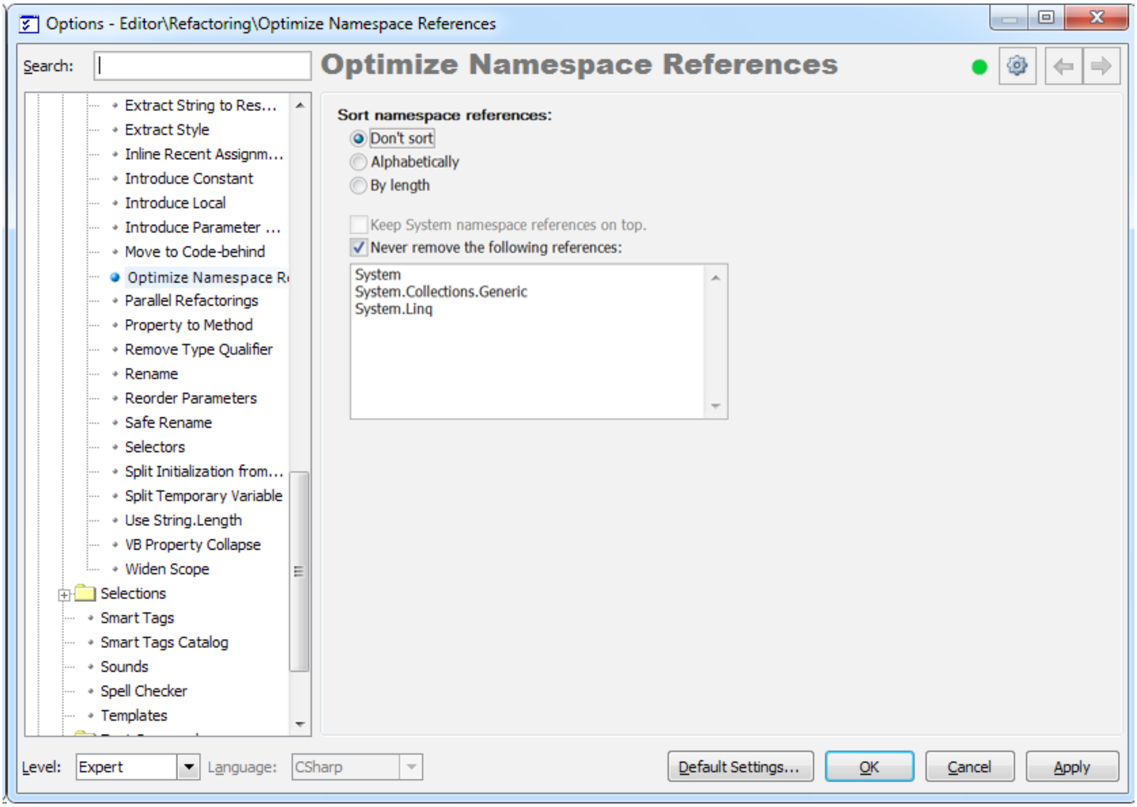


```
using System;
~~using System.Collections;~~
using System.Collections.Generic;
~~using System.Text;~~
using System.Linq;
~~using DevExpress.CodeRush.Core;~~
using DevExpress.CodeRush.Diagnostics;
using DevExpress.CodeRush.StructuralParser;
```

Optimize Namespace References

Removes unused namespace references from the current file

The sorting method and the list of references to keep is configurable on the **Optimize Namespace Reference** options page (in the “Editor | Refactoring” category) in the IDE Tools Options Dialog:



Options - Editor\Refactoring\Optimize Namespace References

Optimize Namespace References

Sort namespace references:

● Don't sort
○ Alphabetically
○ By length

□ Keep System namespace references on top.
☑ Never remove the following references:

```
System
System.Collections.Generic
System.Linq
```

Level: Expert   Language: CSharp

To sort namespace references without optimizing them, you are able to use the stand-alone Sort Namespace References refactoring, which allows you to choose the sorting algorithm right in the Popup menu instead of options page.

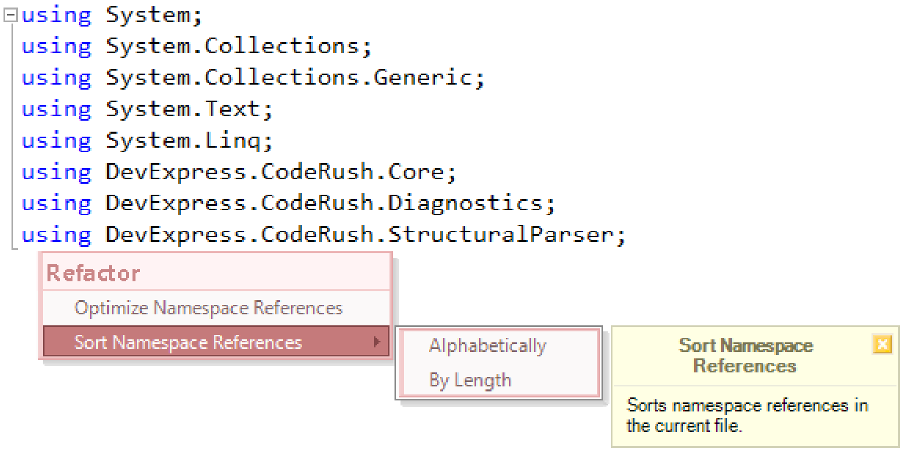
Sort Namespace References

The **Sort Namespace References** refactoring is a stand-alone part of the Optimize Namespace References refactoring. In contrast to that refactoring, the **Sort Namespace References** refactoring doesn't remove any references, and as the name says, it just sorts the 'using's (*C#*) or 'Imports' (*VB*) statements in the active source file.

The following sorting methods are available:

- Alphabetically
- By Length

These sorting methods are available only if the namespace references are not yet been sorted by any type of sorting:

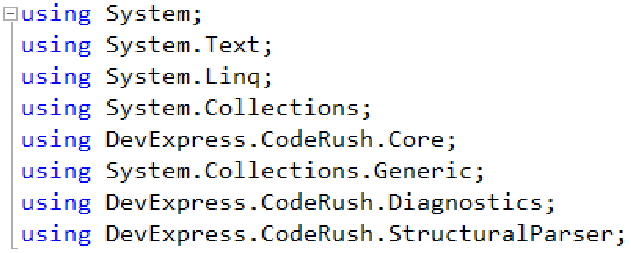


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using DevExpress.CodeRush.Core;
using DevExpress.CodeRush.Diagnostics;
using DevExpress.CodeRush.StructuralParser;
```

Refactor
Optimize Namespace References
Sort Namespace References
Alphabetically
By Length
Sort Namespace References

Sorts namespace references in the current file.

The result of ‘By Length’ sorting:



```
using System;
using System.Text;
using System.Linq;
using System.Collections;
using DevExpress.CodeRush.Core;
using System.Collections.Generic;
using DevExpress.CodeRush.Diagnostics;
using DevExpress.CodeRush.StructuralParser;
```

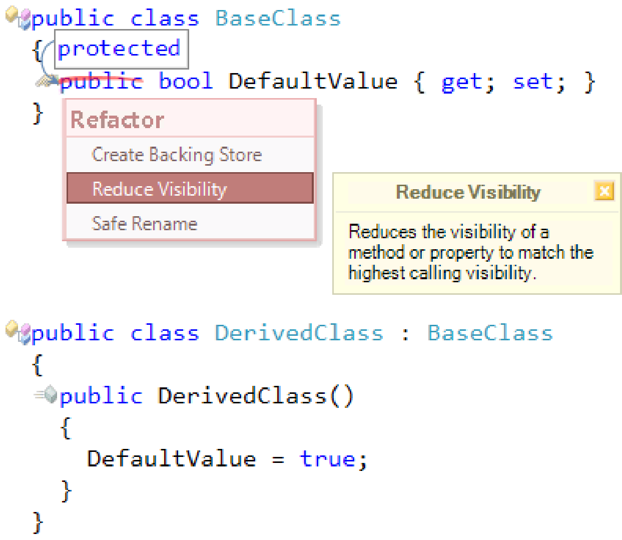
Refactoring visibilities

Reduce Visibility

Each member of a type declaration has an associated visibility (accessibility) modifier, which controls the access to this member for other members. The **Reduce Visibility** refactoring shipped in Refactor! Pro allows you to quickly reduce the visibility of a member to match the highest calling visibility, in other words, to restrict the visibility as much as possible.

For example, if a member is called only inside the current class and not outside of it, a member can have the ‘private’ visibility. If a member is called from only descendants of the current class, it may have the ‘protected’ visibility.

The refactoring is available on the visibility keyword:



```
public class BaseClass
{ protected
  public bool DefaultValue { get; set; }
}
```

Reduce Visibility

Reduces the visibility of a method or property to match the highest calling visibility.

```
public class DerivedClass : BaseClass
{
  public DerivedClass()
  {
    DefaultValue = true;
  }
}
```

Reducing the visibility may improve the clarity of your API.

Variable scope widening (Widen Scope)

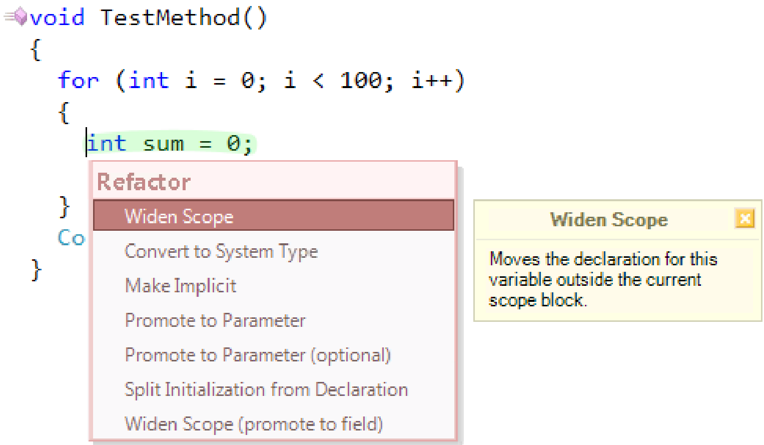
The scope of a variable declaration determines its visibility to the rest of a program. Scopes can be member-level, class-level and nested, where an inner scope may re-declare the meaning of a variable from an outer scope.

When declared inside a member, the scope of the variable is the entire member after the variable declaration, including all nested code blocks. This means that the variable is available to use within the member but when control passes to another member, the variable is unavailable.

A variable can also be declared within a code block. In this case, a variable will only be visible to the code within the block and not available outside of this code block. Code blocks can be nested. For example, a loop within a method within a class provides three levels of nested code blocks and therefore, three levels of nested scope. When a variable is declared within one of these nested scopes, it is visible only to the current scope and any scopes nested within it. This means that a variable declared within a loop is not visible outside of that loop, whereas a variable declared outside of the loop is also visible within the loop.

A wider scoped variable could be declared at the class-level, so that it can be used by any member within the class. In fact, a variable's scope is always the full extent of the code block it is declared within.

When you want to widen the scope of the variable, you can perform the **Widen Scope** refactoring to do this automatically. This refactoring moves the variable declaration outside the current scope block. If a variable is initialized with some value, the refactoring automatically determines whether to move the initialization value with the declaration or leave it at the source position:

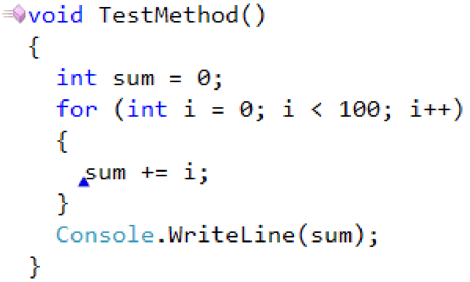


```
void TestMethod()
{
  for (int i = 0; i < 100; i++)
  {
    int sum = 0;

  }
  Co
}
```

Refactor
Widen Scope
Convert to System Type
Make Implicit
Promote to Parameter
Promote to Parameter (optional)
Split Initialization from Declaration
Widen Scope (promote to field)
Widen Scope

Moves the declaration for this variable outside the current scope block.

After it is executed, a navigation marker is dropped at the starting position, so you can navigate back by pressing Esc:

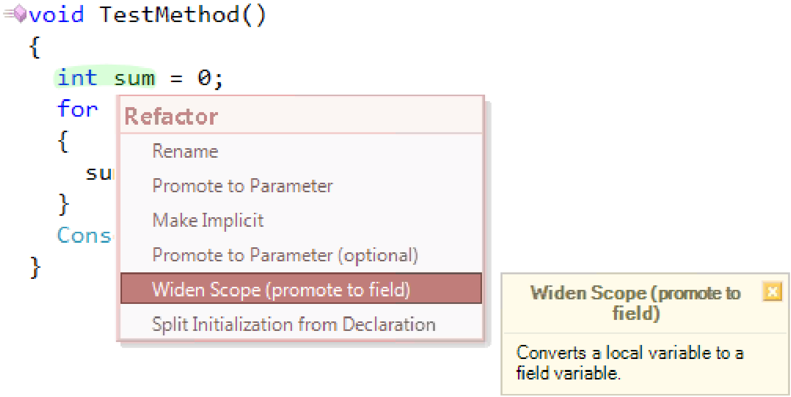


```
void TestMethod()
{
  int sum = 0;
  for (int i = 0; i < 100; i++)
  {
    sum += i;
  }
  Console.WriteLine(sum);
}
```

There is an option for this refactoring which specifies whether to increase the scope by one level, or always promote a variable to the maximum visibility (e.g. top of a method). The option is available in the **Editor** | **Refactoring** | **Widen Scope** options page in the CodeRush Options Dialog.

There are two additional modifications of the **Widen Scope** refactoring:

- **Widen Scope** (**promote to field**) – converts a local variable to a field variable. In other words, this refactoring moves a variable to the top class-level scope:

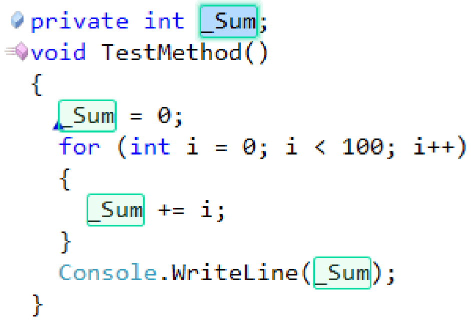


```
void TestMethod()
{
  int sum = 0;
  for
  {
    su
  }
  Cons
}
```

Refactor
Rename
Promote to Parameter
Make Implicit
Promote to Parameter (optional)
Widen Scope (promote to field)
Split Initialization from Declaration
Widen Scope (promote to field)

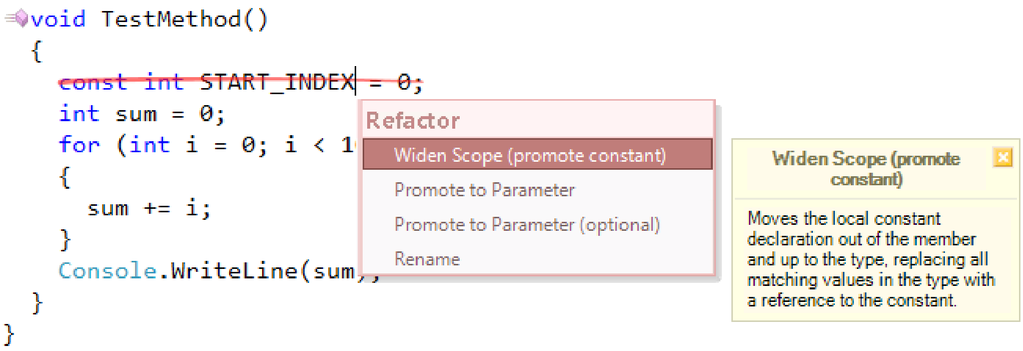
Converts a local variable to a field variable.

The field variable will be declared with the name corresponding to your Identifiers options and all references to it will be linked together:



```
private int _Sum;
void TestMethod()
  {
    _Sum = 0;
    for (int i = 0; i < 100; i++)
    {
      _Sum += i;
    }
    Console.WriteLine(_Sum);
  }
```

- **Widen Scope (promote constant)** – moves a local constant to a class-level scope and replaces all similar constant values with the reference to a constant:



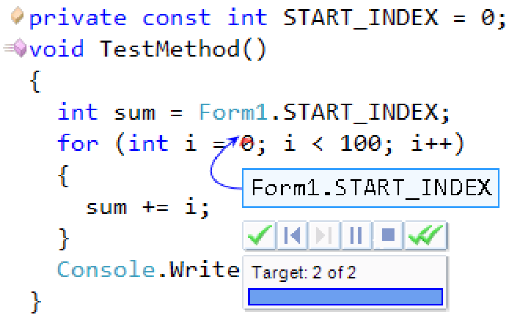
```
void TestMethod()
  {
    ~~const int START_INDEX = 0;~~
    int sum = 0;
    for (int i = 0; i < 1
    {
      sum += i;
    }
    Console.WriteLine(sum);
  }
}
```

Refactor
Widen Scope (promote constant)
Promote to Parameter
Promote to Parameter (optional)
Rename

Widen Scope (promote constant)

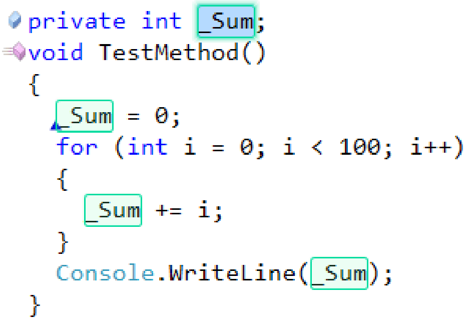
Moves the local constant declaration out of the member and up to the type, replacing all matching values in the type with a reference to the constant.

Once applied, you will be able to replace all similar expressions from the class to the reference on the newly created constant:



```
private const int START_INDEX = 0;
void TestMethod()
  {
    int sum = Form1.START_INDEX;
    for (int i = 0; i < 100; i++)
    {
      sum += i;
    }
    Console.Write
  }
```

When all expressions are replaced, they are linked together as well:



```
private int _Sum;
void TestMethod()
{
  _Sum = 0;
  for (int i = 0; i < 100; i++)
  {
    _Sum += i;
  }
  Console.WriteLine(_Sum);
}
```

This refactoring is useful when you want to introduce a class-level constant for all similar expressions of the current class.

Various refactorings

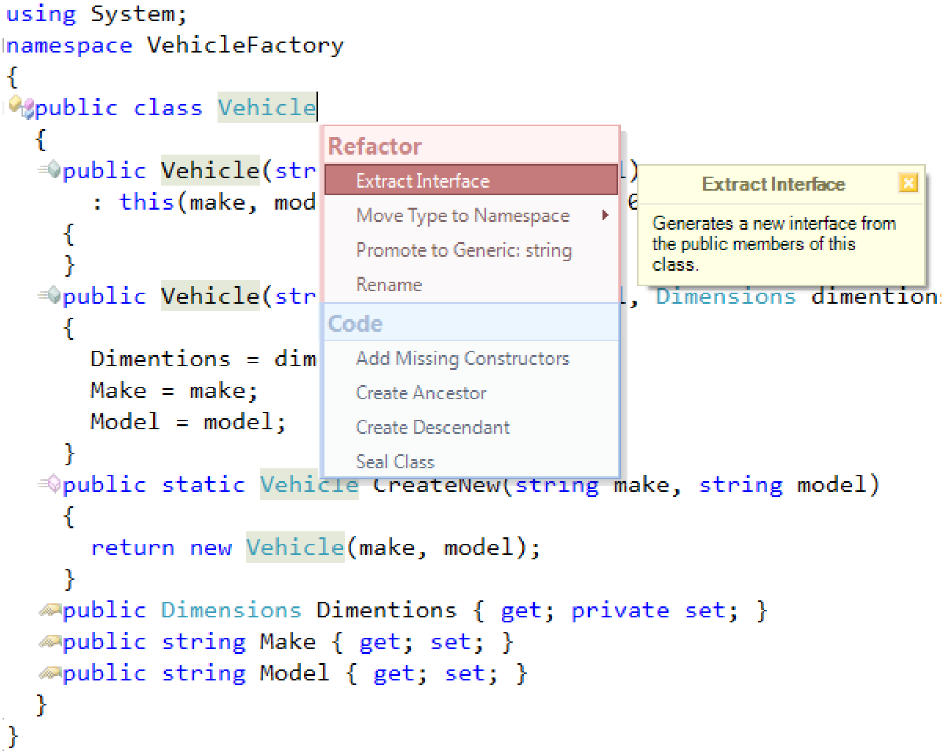
Extract Interface

Extract Interface is a refactoring that provides an easy way to create a new interface with members that originate from an existing class or struct.

When several clients use the same subset of members from a class or struct, or when multiple classes or structs have a subset of members in common, it can be useful to embody the subset of members in an interface. The **Extract Interface** will help to create the new interface for you. Just place the caret on the type name and perform a refactoring.

Extract Interface generates an interface above the type it is performed on, and makes the current type implement this interface. A new interface name is automatically linked with the current type ancestor, so you can easily rename the newly created interface. If you would like to move the new interface to its own file, use the Move Type to File refactoring.

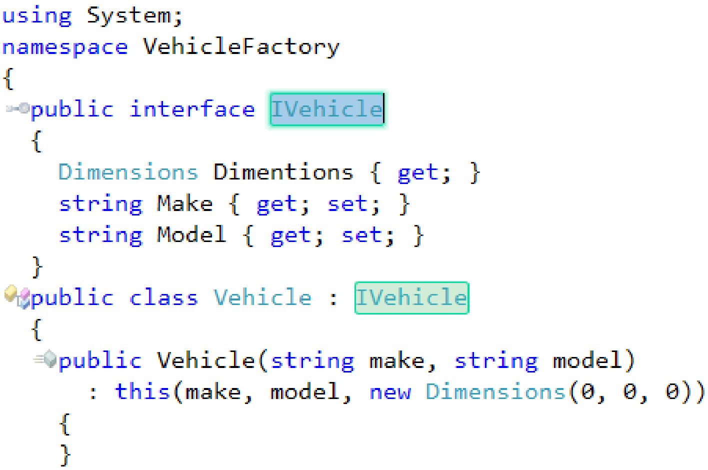
Sample:



```
using System;
namespace VehicleFactory
{
  public class Vehicle
  {
    public Vehicle(str                      l)
      : this(make, mod                      0
    {
    }
    public Vehicle(str                      l, Dimensions dimention
    {
      Dimentions = dim
      Make = make;
      Model = model;
    }
    public static Vehicle CreateNew(string make, string model)
    {
      return new Vehicle(make, model);
    }
    public Dimensions Dimentions { get; private set; }
    public string Make { get; set; }
    public string Model { get; set; }
  }
}
```

Generates a new interface from the public members of this class.

Result interface:



```
using System;
namespace VehicleFactory
{
  public interface IVehicle
  {
    Dimensions Dimentions { get; }
    string Make { get; set; }
    string Model { get; set; }
  }
  public class Vehicle : IVehicle
  {
    public Vehicle(string make, string model)
      : this(make, model, new Dimensions(0, 0, 0))
    {
    }
```

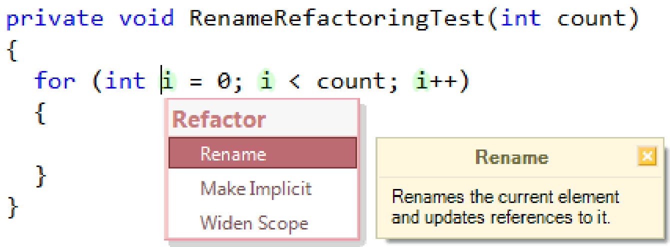
The refactoring doesn’t have additional options.

Rename

This refactoring provides an easy way to rename identifiers for code symbols such as locals, type members, types or namespaces and updating all their references.

Rename is one of the most useful refactorings of all. The developer can alter the name of the declaration and its occurrences (references) in the code in the whole solution.

Simply put the caret on the identifier and apply the refactoring. The refactoring will wrap all references and name of the declaration into linked identifiers, that help you easily change all occurrences simultaneously. The preview hint in **Rename** will show visible references in the code editor. Press **Enter** when you are done changing the name of the identifier to commit your changes.

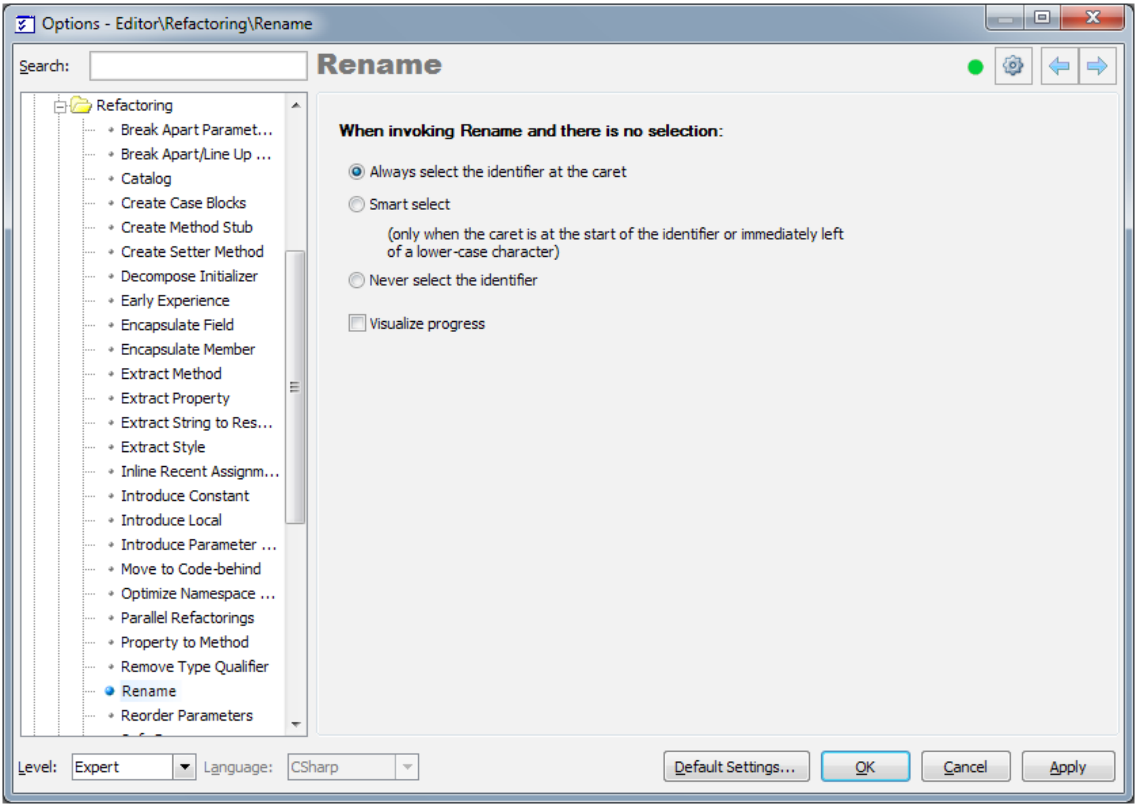


```
private void RenameRefactoringTest(int count)
{
  for (int i = 0; i < count; i++)
  {

  }
}
```

Rename

Renames the current element and updates references to it.

There are a few options available for this refactoring in the Options Dialog:



Options - Editor\Refactoring\Rename

Rename

When invoking Rename and there is no selection:

◉ Always select the identifier at the caret

○ Smart select

(only when the caret is at the start of the identifier or immediately left of a lower-case character)

○ Never select the identifier

□ Visualize progress

Level: Expert

Language: CSharp

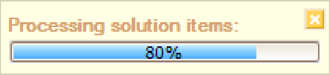
- How the text selection should be changed when the **Rename** is performed on the identifier, and no selection exist:

○ Identifier will be entirely selected

○ Nothing will ever be selected

○ Smart selection – for example, the identifier will be completely selected in the specific context (e.g. when the caret is at the start of the identifier or immediately left of a lower-case character)

- Option whether to show the progress visualization hint of the references search:



Use Implicit Line Continuation

The **Use Implicit Line Continuation** is a Visual Basic language specific refactoring available in Visual Studio ٢٠١٠ that removes redundant line-continuation underscore characters from an entire source file.

Visual Basic language version 10 has been improved in the area of line-continuation characters. Now, there are a lot of places in the code where an underscore is not necessary anymore, which means that Visual Basic is smarter about auto-detecting line continuation scenarios, and as a result, no longer expect you to explicitly indicate that the statement continues on the next code line. For example, the underscore is no longer necessary in the following cases:

- After the following punctuators: comma ‘,’, open parenthesis ‘(‘, open curly brace ‘{‘, begin embedded expression in XML ‘<%=’.

- Before the following punctuators: close parenthesis ‘)’, close curly brace ‘}’, end embedded expression in XML ‘%>’.

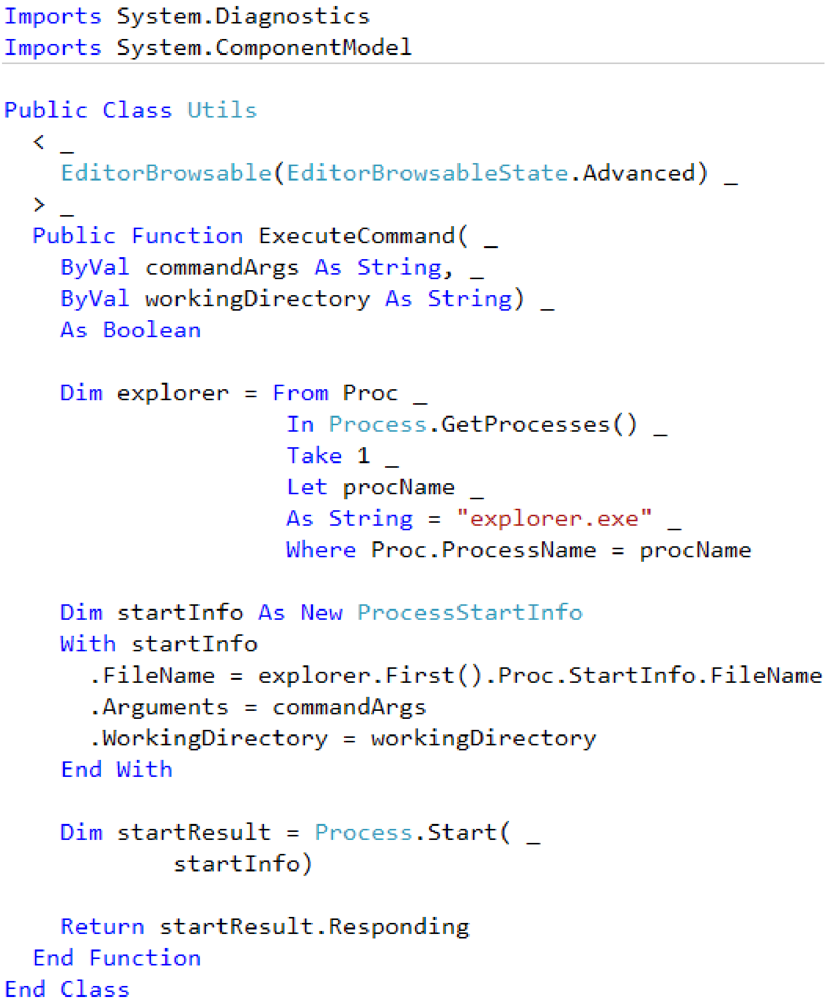
- After an open angle bracket ‘<’ in an attribute context, before a close angle bracket ‘>’ in an attribute context, and after a close angle bracket in a non-file-level attribute context.

- After binary operators in expression contexts.

- Before and after query expression operators.

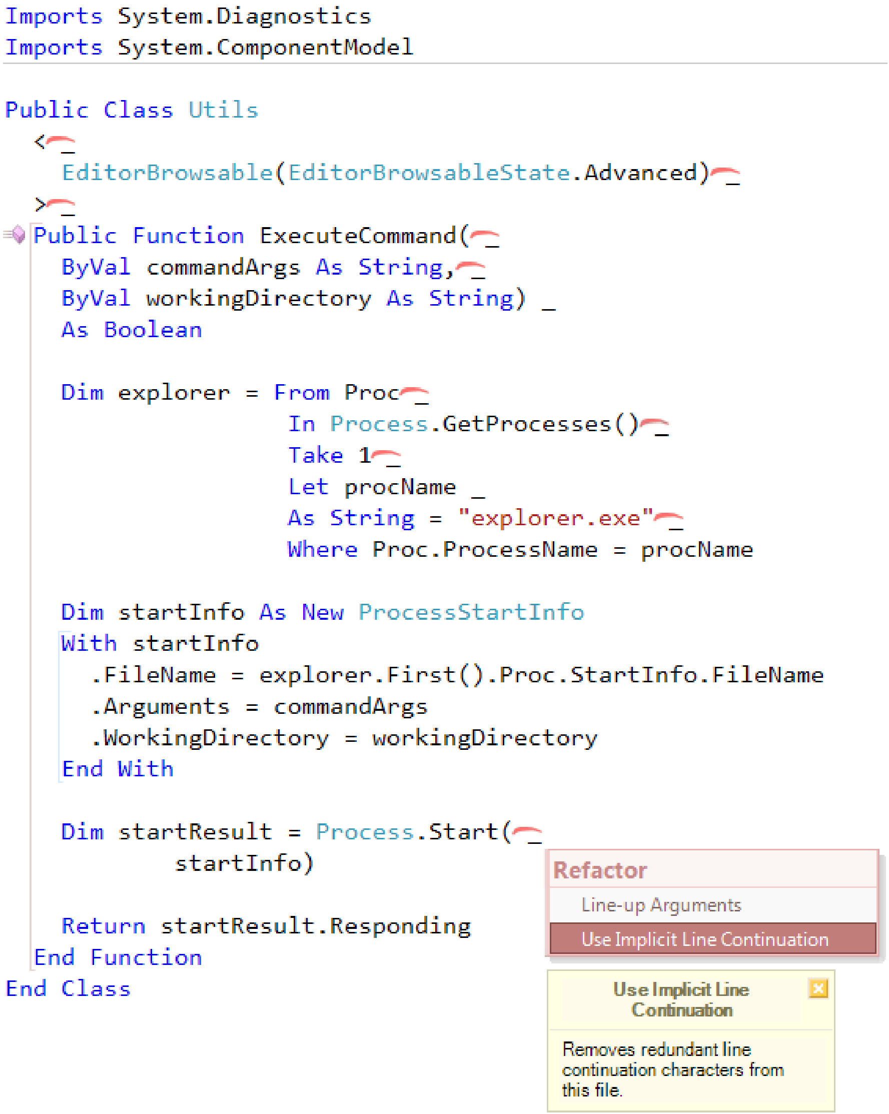
- and others.

Consider the following code:



```
Imports System.Diagnostics
Imports System.ComponentModel

Public Class Utils
  < _
    EditorBrowsable(EditorBrowsableState.Advanced) _
  > _
  Public Function ExecuteCommand( _
    ByVal commandArgs As String, _
    ByVal workingDirectory As String) _
    As Boolean

    Dim explorer = From Proc _
                   In Process.GetProcesses() _
                   Take 1 _
                   Let procName _
                   As String = "explorer.exe" _
                   Where Proc.ProcessName = procName

    Dim startInfo As New ProcessStartInfo
    With startInfo
      .FileName = explorer.First().Proc.StartInfo.FileName
      .Arguments = commandArgs
      .WorkingDirectory = workingDirectory
    End With

    Dim startResult = Process.Start( _
            startInfo)

    Return startResult.Responding
  End Function
End Class
```

To apply the refactoring move the editor caret to one of the underscore characters in the code. Before applying the **Use Implicit Continuation** refactoring, you can see the preview hint:

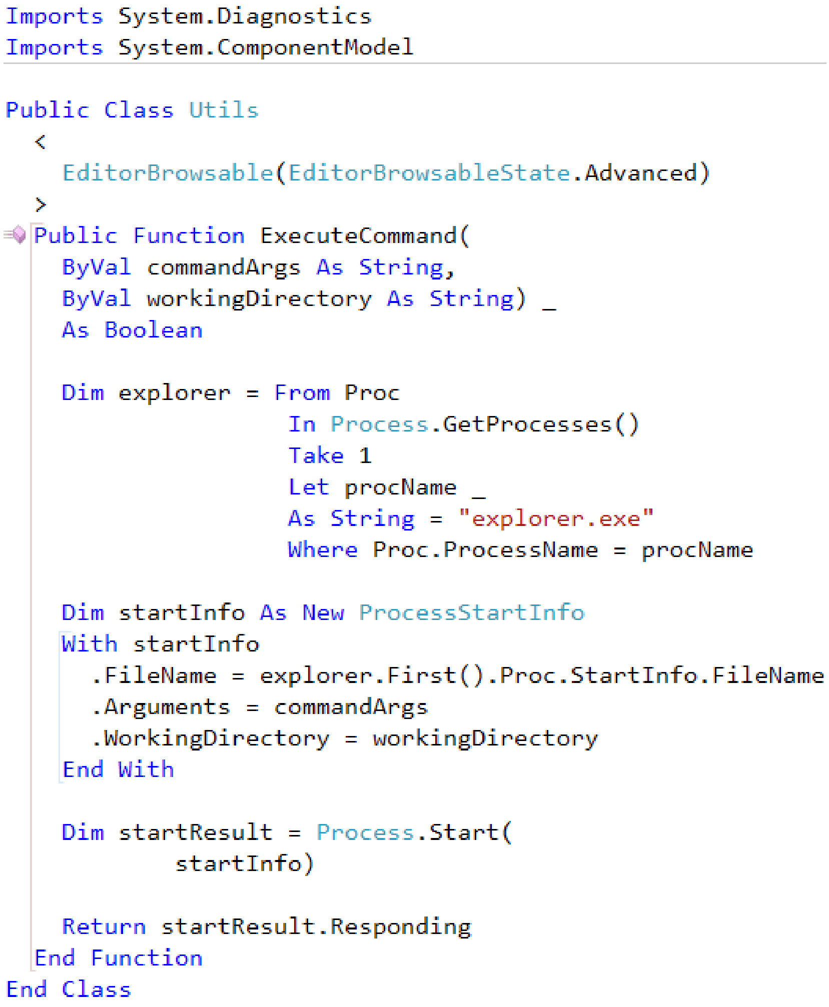


```
Imports System.Diagnostics
Imports System.ComponentModel

Public Class Utils
  <_
    EditorBrowsable(EditorBrowsableState.Advanced)_
  >_
  Public Function ExecuteCommand(_
    ByVal commandArgs As String,_
    ByVal workingDirectory As String) _
    As Boolean

    Dim explorer = From Proc_
                    In Process.GetProcesses()_
                    Take 1_
                    Let procName _
                    As String = "explorer.exe"_
                    Where Proc.ProcessName = procName

    Dim startInfo As New ProcessStartInfo
    With startInfo
      .FileName = explorer.First().Proc.StartInfo.FileName
      .Arguments = commandArgs
      .WorkingDirectory = workingDirectory
    End With

    Dim startResult = Process.Start(_
            startInfo)

    Return startResult.Responding
  End Function
End Class
```

Use Implicit Line Continuation

Removes redundant line continuation characters from this file.

Once the refactoring is applied you will get the following result:



```
Imports System.Diagnostics
Imports System.ComponentModel

Public Class Utils
  <
    EditorBrowsable(EditorBrowsableState.Advanced)
  >
  Public Function ExecuteCommand(
    ByVal commandArgs As String,
    ByVal workingDirectory As String) _
    As Boolean

    Dim explorer = From Proc
                    In Process.GetProcesses()
                    Take 1
                    Let procName _
                    As String = "explorer.exe"
                    Where Proc.ProcessName = procName

    Dim startInfo As New ProcessStartInfo
    With startInfo
      .FileName = explorer.First().Proc.StartInfo.FileName
      .Arguments = commandArgs
      .WorkingDirectory = workingDirectory
    End With

    Dim startResult = Process.Start(
            startInfo)

    Return startResult.Responding
  End Function
End Class
```

Note that a few underscores are left after the refactoring is performed, because they are actually needed in the code. If you remove them, the code is no longer compilable.

Chapter 12. CodeRush extensibility

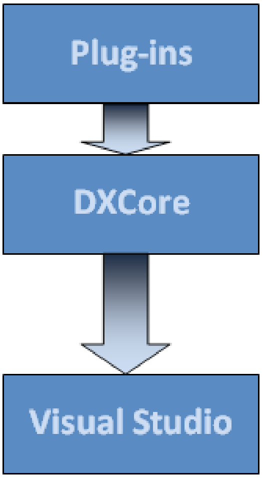
DXCore Framework plug-ins overview

The DXCore provides services, wizards, and a visual extensibility framework designed to make it easy to extend Visual Studio. All products like CodeRush and Refactor! Pro were designed upon **DXCore** for Visual Studio® .NET. In the interest of encouraging developers to be pro-active about productivity,DevExpress is making available for general download of **DXCore** framework so that developers can build their own productivity plug-ins to extend the Visual Studio themselves using a simple visual framework for IDE extension. The **DXCore** is absolutely free for personal use but it is not open source.

DXCore uses a plug-in based architecture and gives you the ability to create anything from tool windows (hosting whatever code you want) to "Actions" (items which can be tied to keyboard shortcuts and context menus) and refactorings, with an extremely rich context system, and a language agnostic code generation engine. It provides a very nice abstraction layer on top of the Visual Studio. It is also written to be version agnostic, so plug-ins written against **DxCore** will work for VS2008 2005 and up.

DXCore plug-ins architecture and design philosophy

A feature of **IDE tools** (CodeRush and/or Refactor! products) resides in DXCore plug-ins. A plug-in is special class that resides in an assembly that is loaded into the Visual Studio environment when **DXCore** starts-up. While plug-ins implement the high level solutions you see, they don't do all the work. Each plug-in references a namespace in the **DXCore**, which holds a powerful framework packed with low-level services and events.



The **DXCore** is loaded into Visual Studio as an Add-in for VS2005/VS2008 or as an Extension for VS2010.

Design Philosophy

The **DXCore** design philosophy is to keep the plug-in code simple. Complexity in a plug-in is a good indication that its code needs to be pushed as a new service and/or API down into the core. This philosophy results in the following benefits:

1. Client code (the plug-in's code) is simple, and easy to understand

2. The **DXCore** is powerful and robust framework

3. That power of the **DXCore** is completely available to third-party plug-in authors

Nearly all the code in a plug-in is there to support the high-level feature, with a noticeable absence of busywork. The code reflects the essence of the plug-in's purpose.

Design Goals

Ease of extensibility has been one of **DXCore** top design goals from the beginning. So it's no accident that the **DXCore** is the easiest way to extend Visual Studio IDE.

- Step-by-step Construction

Plug-in wizards create standard plug-ins, tool windows, and options pages. These wizards lay down the framework upon which your plug-ins will grow.

- Visual Extensibility

DXCore plug-ins are descendants of .NET controls, having the properties and a rich collection of useful events. Just drop these specialized controls onto your plug-in design surface, as needed. And because plug-ins are .NET controls, you can use Visual Studio's Property Inspector to change properties and create event handlers.

DXCore plug-in types and naming convention

DXCore has two types of plug-ins. They are "**System**" and "**Ordinary**" plug-ins. **System** plug-ins are located in the similar "System" folder, and all other plug-ins (usual or user plug-ins) reside in the "PlugIns" folder. These folders are located in the "**Bin**" folder of every product installed. The difference between "**System**" and ordinary plug-ins is that the former are always loaded before all other plug-ins.

There's a naming convention for plug-in assemblies that is recommended by **IDE tools** developers. This convention consists of a specific prefix in the name of an assembly. Here are possible name prefixes:

Prefix	Description
DX	A **DXCore** system plug-in, such as "*DX_CSharpLanguage.dll*".
CR	A **CodeRush** plug-in with some specific features, for example, "*CR_AdvancedSelections.dll*".
Refactor	A **Refactor!** plug-in that contains a single or few of refactorings, e.g. "*Refactor_ReorderParams.dll*".
Code	A **CodeRush** plug-in that contains code providers (similar to refactoring provider, but may change a program behavior). An example of such an assembly is "*Code_Declare.dll*".

That's it. If you're going to extend Visual Studio using the **DXCore**, it's extremely easy to do. Just bear in mind, that **DXCore** plug-ins are one of the most powerful extensions ever developed for Visual Studio IDE!

DXCore plug-ins overview

These are two primary types of **plug-ins** you can create:

1. Standard Plug-in

The standard plug-in is a workhorse of the extensibility landscape. It serves as a container for DXCore components, such as actions, providers, and your custom event handlers.

- **Tool Window**

Tool windows are modeless forms that can be docked inside the Visual Studio IDE. The *Solution Explorer*, *Toolbox*, and *Property Browser* are all examples of tool window plug-ins. Creating a tool window plug-in is easy. Just arrange your components in the tool window designer, set a few properties, and compile the project.

You can also add some additional project items into your plug-in project:

- Additional **Standard Plug-in** or a **Tool Window**
- Options page

Custom pages participate in the Options Dialog. If your plug-in has any kind of user-configuration, this is a way to go. Option pages offer a number of advantages over a custom roll-your-own approach.

A particular plug-in can be written in *CSharp* or *Visual Basic* languages only. Unfortunately, you can't use the *C++* language to write a plug-in, because this language has diverged from the common wizard model in Visual Studio (which other languages use), causing this support to be very non-trivial to add to**DXCore**.

How DXCore plug-ins are loaded

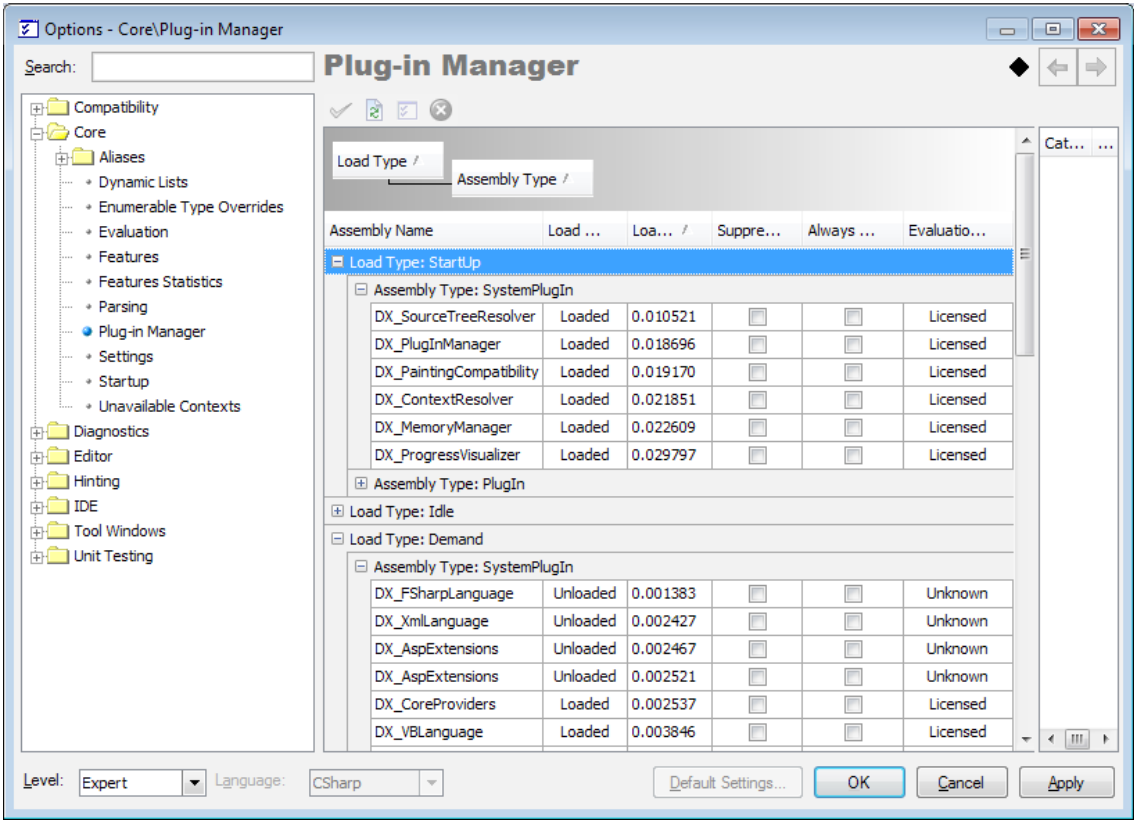
DXCore has a built-in **Loader Engine**, which is intended to improve the speed of its start-up process. However, the first **DXCore** start-up process is pretty long (it may take a minute or two). There's a reason that the first start-up takes much more time then the subsequent launches. When **DXCore** loads for the first time, it has to load all of the plug-ins found in both, "**System**" and "**PlugIns**" subfolders. At this time, **DXCore** profiles every loaded assembly: checks its load type, loading time, so in brief, all important and necessary information that **DXCore** should know about a particular assembly. All this information is saved to Loader profiles in an **XML** format. This information will be used on every subsequent **DXCore** launch, so that Visual Studio should start instantly, because, in the first place, plug-ins are loaded from profiles only after the splash screen goes away and, secondly, because Loader Engine has optimized**DXCore** loading process specially for your system.

Plug-in profiles are stored in the "**Loader"** subfolder of the Settings folder:

%AppData%\CodeRush for VS .NET\1.1\Settings.xml\Loader

If any of the plug-ins have changed on disk for some reason (recompiled and/or replaced) or a new plug-in is installed, the **Loader Engine** will notice that and correct its profiles for these specific plug-ins only.

You can see the profiles information on the "**Plug-in Manager**" options page in the Options Dialog:



Level: Expert

Language: CSharp

Assembly Name	Load ...	Loa...	Suppre...	Always ...	Evaluatio...
⊟ Load Type: StartUp					
⊟ Assembly Type: SystemPlugIn					
DX_SourceTreeResolver	Loaded	0.010521	☐	☐	Licensed
DX_PlugInManager	Loaded	0.018696	☐	☐	Licensed
DX_PaintingCompatibility	Loaded	0.019170	☐	☐	Licensed
DX_ContextResolver	Loaded	0.021851	☐	☐	Licensed
DX_MemoryManager	Loaded	0.022609	☐	☐	Licensed
DX_ProgressVisualizer	Loaded	0.029797	☐	☐	Licensed
⊞ Assembly Type: PlugIn					
⊞ Load Type: Idle					
⊟ Load Type: Demand					
⊟ Assembly Type: SystemPlugIn					
DX_FSharpLanguage	Unloaded	0.001383	☐	☐	Unknown
DX_XmlLanguage	Unloaded	0.002427	☐	☐	Unknown
DX_AspExtensions	Unloaded	0.002467	☐	☐	Unknown
DX_AspExtensions	Unloaded	0.002521	☐	☐	Unknown
DX_CoreProviders	Loaded	0.002537	☐	☐	Licensed
DX_VBLanguage	Loaded	0.003846	☐	☐	Licensed

Follow these steps to get to the “**Plug-in Manager**” options page:

1. From the DevExpress menu, select “Options…”.

2. In the tree view on the left, navigate to this folder:

Core

3. Select the “**Plug-in Manager**” options page.

NOTE: *This page level is Expert, and will only be visible if the Level combo on the lower-left of the Options dialog is set to Expert.*

If you have any trouble with loading of some plug-ins or **DXCore** itself, you may try to rebuild the **Loader** profiles. To do this, you need to remove the “**Loader”** folder with profiles from your system.

NOTE: if you are actively developing your own plug-in, it is recommended to change its “loading” setting to load it manually in the **Plug-In Manager**, to prevent it from being loaded automatically. This will allow you to compile and build your plug-in without errors due to file locks on the plug-in assembly. This option can be changed when you create a new DXCore plug-in. If you already have created a plug-in or modify the existing one, you may tweak the “***DXCoreAssembly***” attribute in the “***AssemblyInfo***” file of your plug-in solution to load it manually like this:

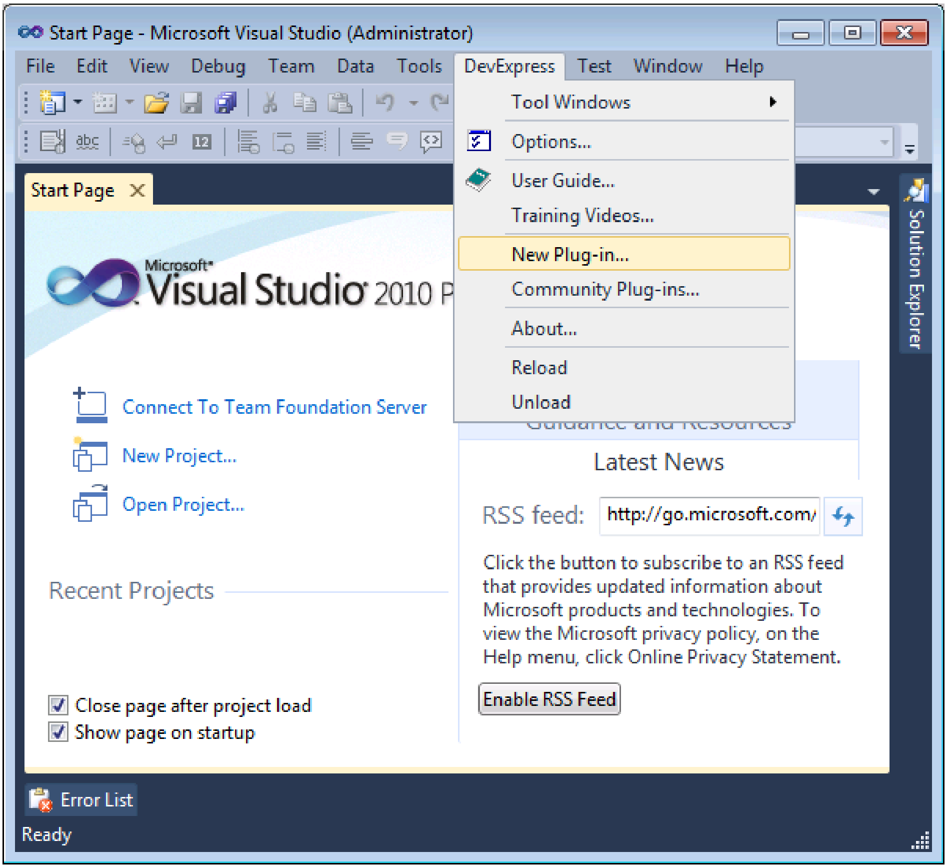
[assembly: DXCoreAssembly(DXCoreAssemblyType.PlugIn, “Assembly Title”, PlugInLoadType.Idle, LoadAbility-Type.LoadDisabled)]

Change the **LoadAbilityType** parameter to **LoadDisabled** to make a plug-in loaded manually.

How to create a new CodeRush (DXCore) plug-in

There are two ways to create a new CodeRush (DXCore) plug-in in Visual Studio:

1. Inside the IDE, from the DevExpress menu, access the “New Plug-in…” menu item:

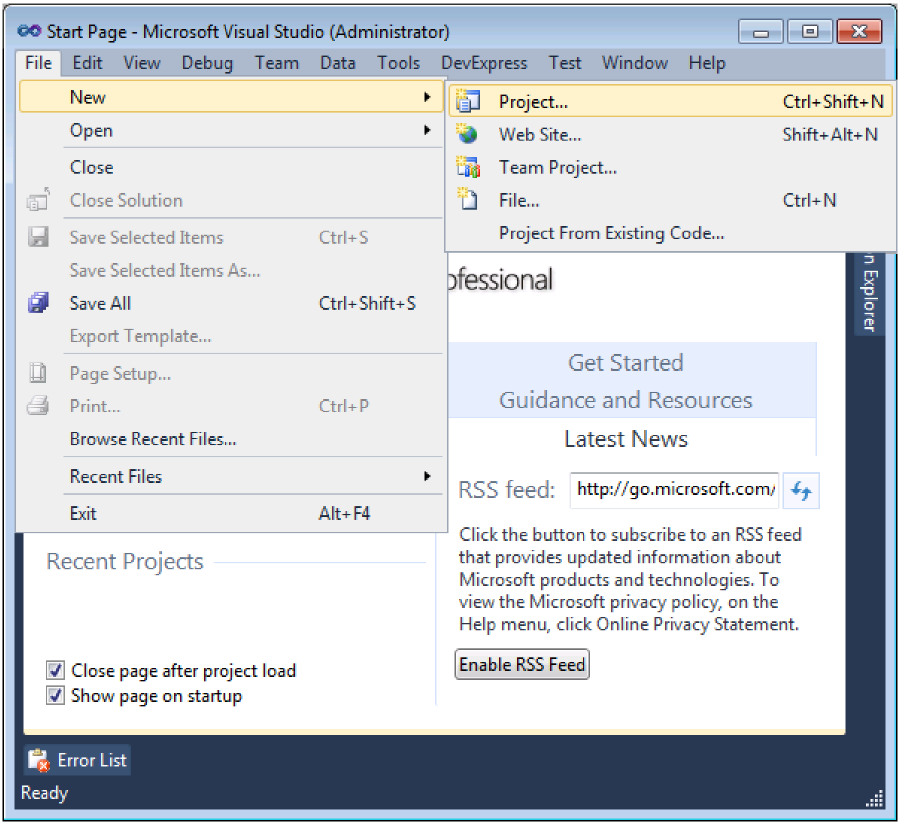


RSS feed: http://go.microsoft.com/

Click the button to subscribe to an RSS feed that provides updated information about Microsoft products and technologies. To view the Microsoft privacy policy, on the Help menu, click Online Privacy Statement.

☑ Close page after project load
☑ Show page on startup

2. Create a new project of type **DXCore** (using File –> New Project menu item):

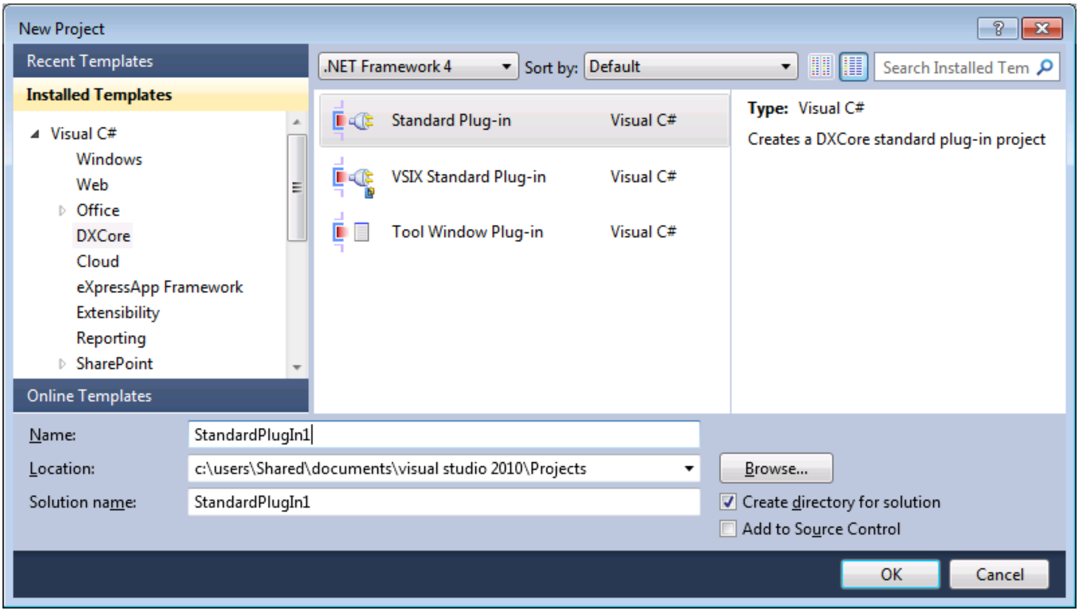


RSS feed: http://go.microsoft.com/

Click the button to subscribe to an RSS feed that provides updated information about Microsoft products and technologies. To view the Microsoft privacy policy, on the Help menu, click Online Privacy Statement.

☑ Close page after project load
☑ Show page on startup

In the next dialog, choose your preferred language (Visual C# or Visual Basic) and select the DXCore category to see available project templates for CodeRush plug-ins:



.NET Framework 4 Sort by: Default

Type: Visual C#

Creates a DXCore standard plug-in project

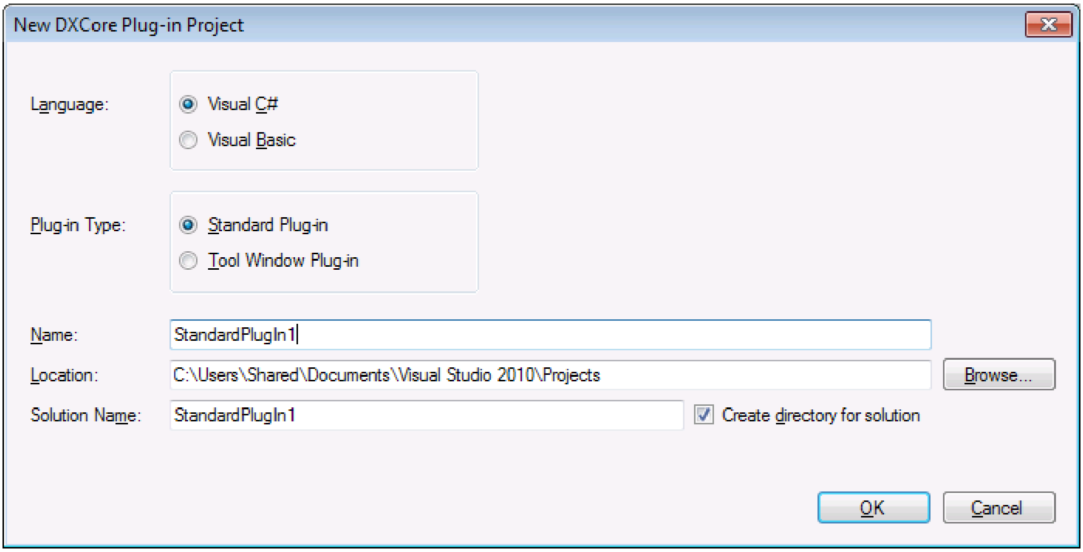
Name: StandardPlugIn1

Location: c:\users\Shared\documents\visual studio 2010\Projects

Solution name: StandardPlugIn1

☑ Create directory for solution
□ Add to Source Control

It is recommended to choose the "New Plug-in…" menu item, because in this case you can tweak additional settings for your new plug-in. The following dialog appears once you click it:



New DXCore Plug-in Project

Language: ● Visual C# ○ Visual Basic

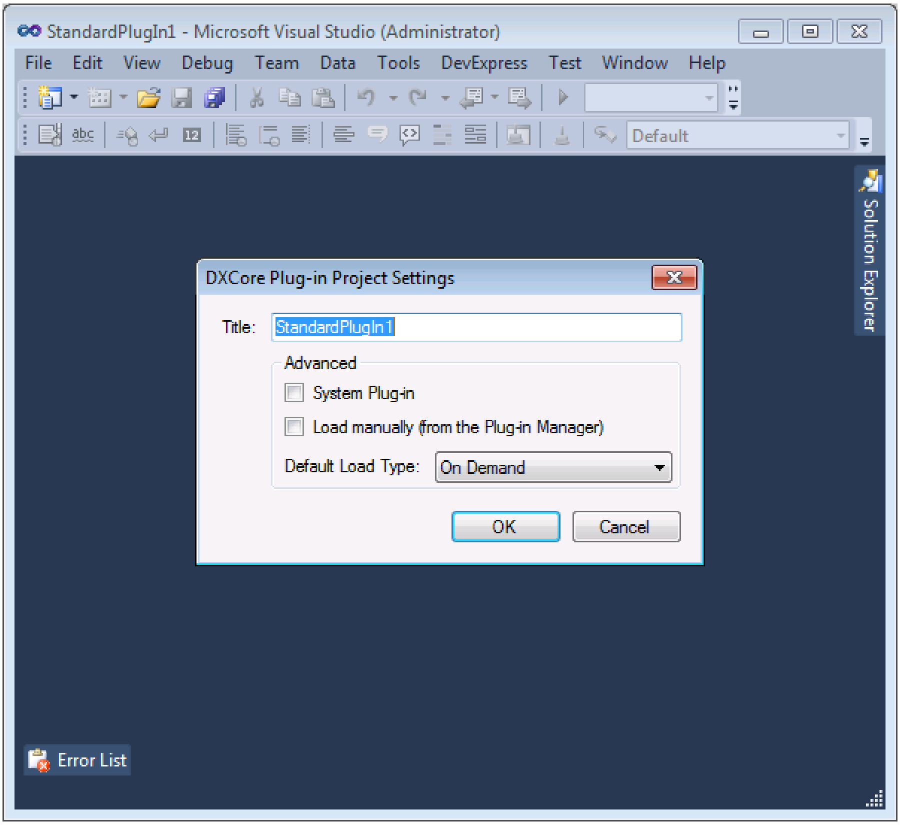
Plug-in Type: ● Standard Plug-in ○ Tool Window Plug-in

Name: StandardPlugIn1

Location: C:\Users\Shared\Documents\Visual Studio 2010\Projects

Solution Name: StandardPlugIn1 ☑ Create directory for solution

Choose your language of preference as well as the plug-in type, type in the plug-in name and its location if required. Click OK – the **DXCore Plug-in Project Settings** dialog will appear:



StandardPlugIn1 - Microsoft Visual Studio (Administrator)

DXCore Plug-in Project Settings

Title: StandardPlugIn1

Advanced

□ System Plug-in

□ Load manually (from the Plug-in Manager)

Default Load Type: On Demand

Here you can select a different title and change advanced plug-in settings, or just click OK to accept the default settings.

Available settings:

- **System Plug-In** – check it if you'd like your plug-in to be loaded before all other plug-ins do. Simple plug-in doesn't need this option in most of the time. System plug-ins are built to the "IDETools\System\DXCore\Bin\System\" folder and usual plug-in will be built to the "IDETools\System\DXCore\Bin\PlugIns\" folder.

- **Load Manually** – check it if you want to not load the plug-in automatically when **DXCore** loads in Visual Studio, this option is only relevant to the development phase. For example, suppose that whilst developing your plugin, you are suddenly required to do something else. But because DXCore loads your plug-in into memory on its startup automatically, the plugin's assembly is locked, so, you won't be able to recompile your plug-in whilst your Visual Studio is not closed. In order to avoid this scenario, change the "Load Manually" option to ensure that the DXCore never automatically loads your plugin. After that, you're able to load plugin using the **PlugIn Manager** (DevExpress -> Options -> Core -> PlugIn Manager). With the PlugIn Manager you have only load your plugin into the address space of a single instance of Visual Studio. This means that when you shut down this single copy of Visual Studio, you'll be able to continue development of your plugin.

- **Default Load Type** – different types of loading your plug-in:

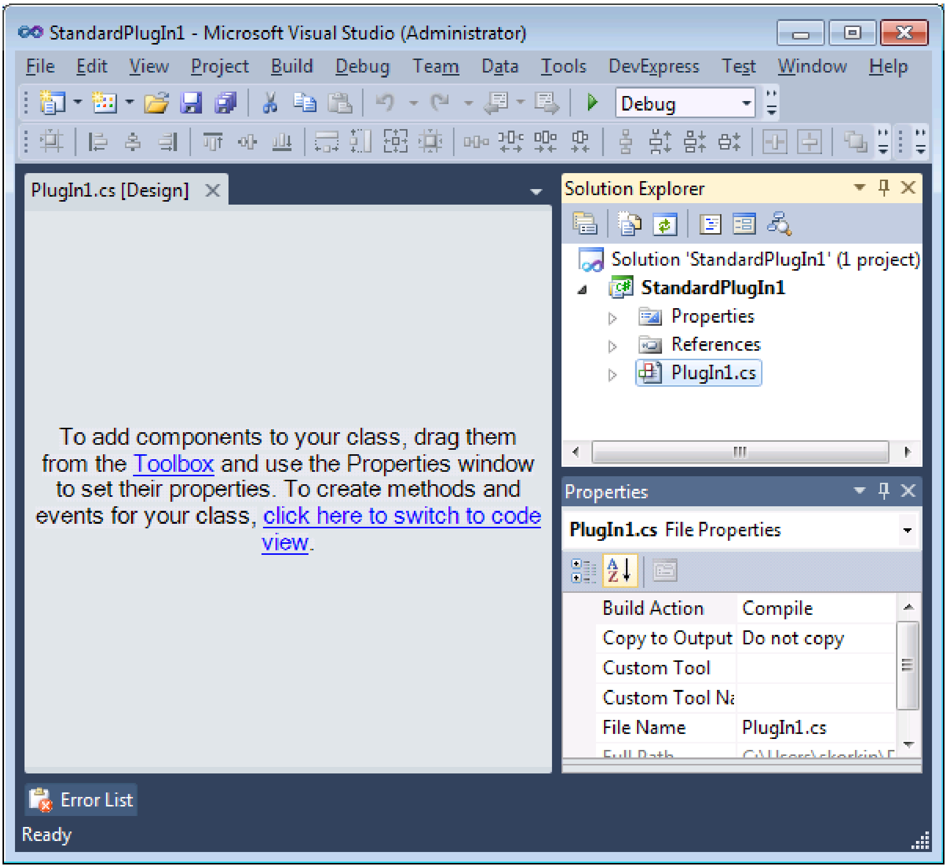
○ **On Demand** – plug-in is loaded when some process or action, for which the plugin should work, takes place, e.g. if your plugin listens to one of the events – then it will be loaded only when this event is being fired. Or, another example, if your plug-in contains an options page, the plugin will be loaded when the Options Dialog is going to display this page.

○ **On Idle** – plug-in will be loaded when Visual Studio editor is being idle (no user operations/interaction with IDE).

○ **At Start-up** – plugin will be loaded on DXCore start-up, which will add time to overall loading time.

The best option here is the default one – "On Demand". Most of the built-in DXCore plug-ins use this load type option.

Click OK to accept your settings; the plug-in designer surface will be opened in Visual Studio:


StandardPlugIn1 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Data Tools DevExpress Test Window Help
Debug
PlugIn1.cs [Design]
Solution Explorer
Solution 'StandardPlugIn1' (1 project)
StandardPlugIn1
Properties
References
PlugIn1.cs
Properties
PlugIn1.cs File Properties
Error List
Ready

To add components to your class, drag them from the Toolbox and use the Properties window to set their properties. To create methods and events for your class, click here to switch to code view.

Build Action	Compile
Copy to Output	Do not copy
Custom Tool	
Custom Tool Na	
File Name	PlugIn1.cs
Full Path	C:\Users\ckorkin\E

The new plug-in does not provide any functionality yet. You can drop any DXCore component on its surface and subscribe to required events including those coming from Visual Studio to extend the Visual Studio IDE.

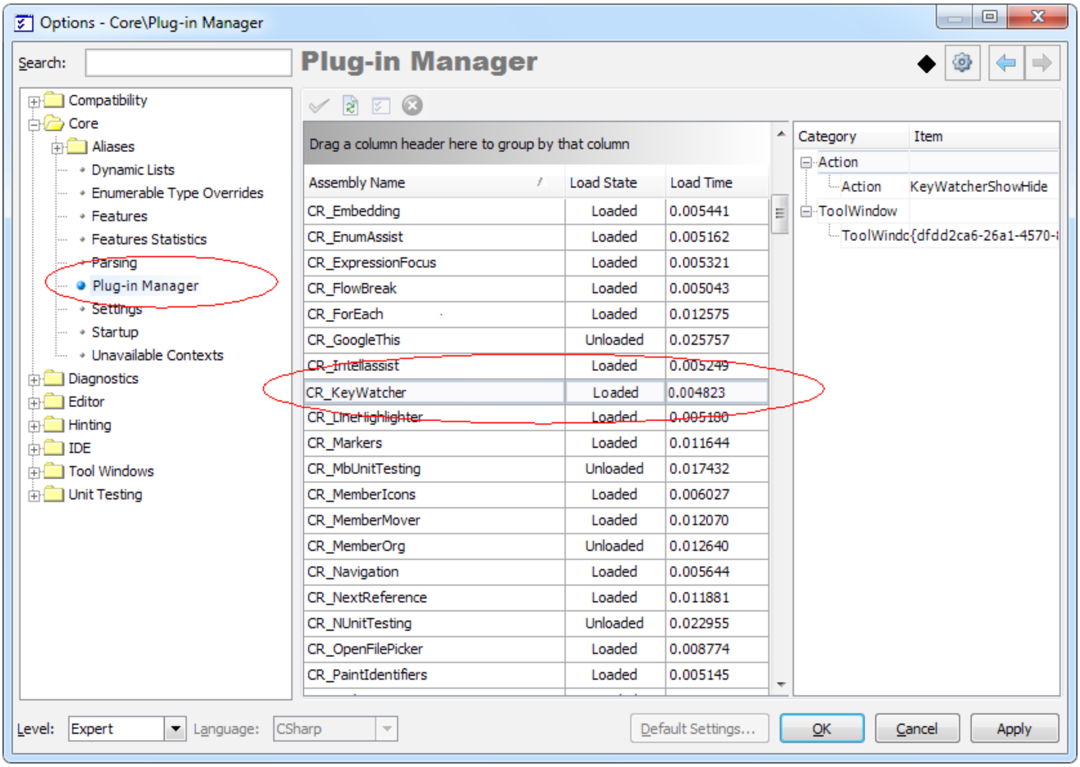
How to install a particular DXCore plug-in

To install a particular DXCore plug-in you need to copy the plug-in dll into your plug-ins directory. Usually it looks similar to this path:

%userprofile%\Documents\DevExpress\IDE Tools\Community\PlugIns\

where “%userprofile%\Documents” is a path to the user Documents folder.

Once you copied the plug-in assembly, start Visual Studio and the plug-in will be loaded automatically by the DX-Core. To see the overall list of plug-ins loaded you can use the **Plug-In Manager** options page inside the Options Dialog. Make sure that the plug-in you installed is listed in the **Plug-In Manager**and the state of the plug-in is “Loaded”. For example, if we install the KeyWatcher plug-in, it will appear in the list:



Options - Core\Plug-in Manager

Search:

Plug-in Manager

- Compatibility
- Core
  - Aliases
  - Dynamic Lists
  - Enumerable Type Overrides
  - Features
  - Features Statistics
  - Parsing
  - Plug-in Manager
  - Settings
  - Startup
  - Unavailable Contexts
- Diagnostics
- Editor
- Hinting
- IDE
- Tool Windows
- Unit Testing

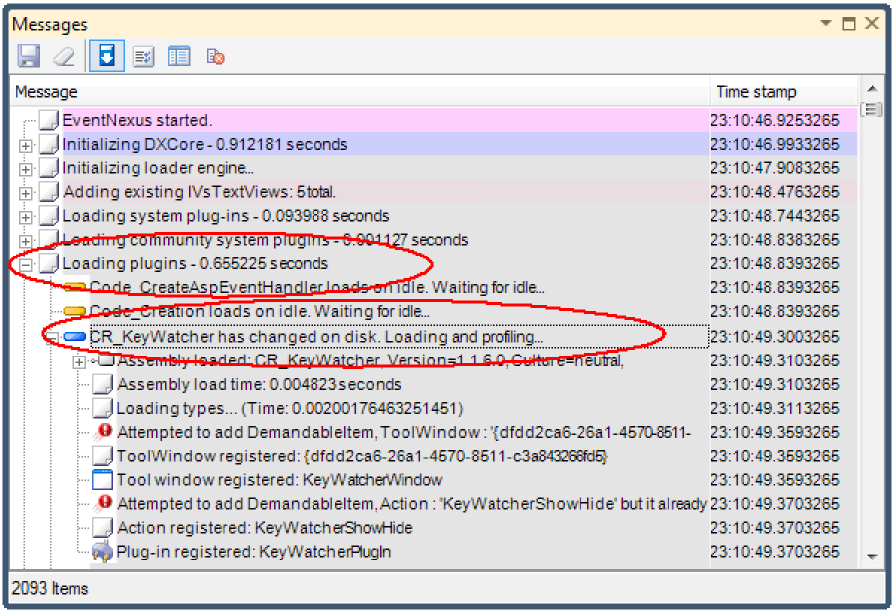
Drag a column header here to group by that column

Assembly Name	Load State	Load Time
CR_Embedding	Loaded	0.005441
CR_EnumAssist	Loaded	0.005162
CR_ExpressionFocus	Loaded	0.005321
CR_FlowBreak	Loaded	0.005043
CR_ForEach	Loaded	0.012575
CR_GoogleThis	Unloaded	0.025757
CR_Intellassist	Loaded	0.005249
CR_KeyWatcher	Loaded	0.004823
CR_LineHighlighter	Loaded	0.005180
CR_Markers	Loaded	0.011644
CR_MbUnitTesting	Unloaded	0.017432
CR_MemberIcons	Loaded	0.006027
CR_MemberMover	Loaded	0.012070
CR_MemberOrg	Unloaded	0.012640
CR_Navigation	Loaded	0.005644
CR_NextReference	Loaded	0.011881
CR_NUnitTesting	Unloaded	0.022955
CR_OpenFilePicker	Loaded	0.008774
CR_PaintIdentifiers	Loaded	0.005145

Category	Item
Action	
Action	KeyWatcherShowHide
ToolWindow	
ToolWindo	{dfdd2ca6-26a1-4570-8

Level: Expert   Language: CSharp

If you don't see the installed plug-in in the list, the Messages diagnostic tool window may have some clues. To open it, go to the **DevExpress** | **Tool Windows** | **Diagnostics** | **Messages** menu item. Navigate to the top of all diagnostic messages, expand the "*Loading plugins…*" item, find the "*CR_KeyWatcher*" plug-in and see the details of its load to find any clues:

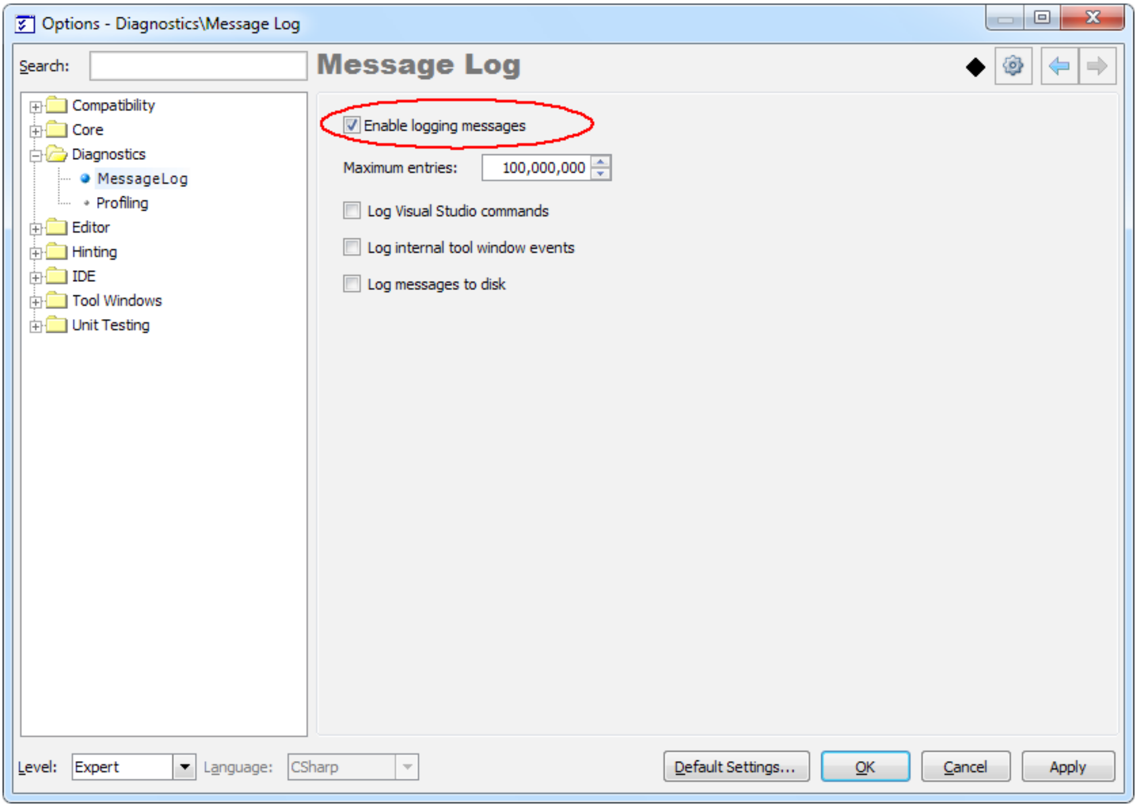


Messages

Message	Time stamp
EventNexus started.	23:10:46.9253265
Initializing DXCore - 0.912181 seconds	23:10:46.9933265
Initializing loader engine...	23:10:47.9083265
Adding existing IVsTextViews: 5 total.	23:10:48.4763265
Loading system plug-ins - 0.093988 seconds	23:10:48.7443265
Loading community system plugins - 0.001127 seconds	23:10:48.8383265
Loading plugins - 0.655225 seconds	23:10:48.8393265
Code_CreateAspEventHandler loads on idle. Waiting for idle...	23:10:48.8393265
Code_Creation loads on idle. Waiting for idle...	23:10:48.8393265
CR_KeyWatcher has changed on disk. Loading and profiling...	23:10:49.3003265
Assembly loaded: CR_KeyWatcher, Version=1.1.6.0, Culture=neutral,	23:10:49.3103265
Assembly load time: 0.004823 seconds	23:10:49.3103265
Loading types... (Time: 0.00200176463251451)	23:10:49.3113265
Attempted to add DemandableItem, ToolWindow : '{dfdd2ca6-26a1-4570-8511-	23:10:49.3593265
ToolWindow registered: {dfdd2ca6-26a1-4570-8511-c3a843266fd5}	23:10:49.3593265
Tool window registered: KeyWatcherWindow	23:10:49.3593265
Attempted to add DemandableItem, Action : 'KeyWatcherShowHide' but it already	23:10:49.3703265
Action registered: KeyWatcherShowHide	23:10:49.3703265
Plug-in registered: KeyWatcherPlugIn	23:10:49.3703265

2093 Items

Note that the diagnostic messages are not logged by default, you have to manually enable the logging on the **Diagnostics** | **Message Log** options page:



Options - Diagnostics\Message Log

Message Log

☑ Enable logging messages

Maximum entries: 100,000,000

□ Log Visual Studio commands

□ Log internal tool window events

□ Log messages to disk

Level: Expert

Language: CSharp

Troubleshooting the DXCore plug-in loading

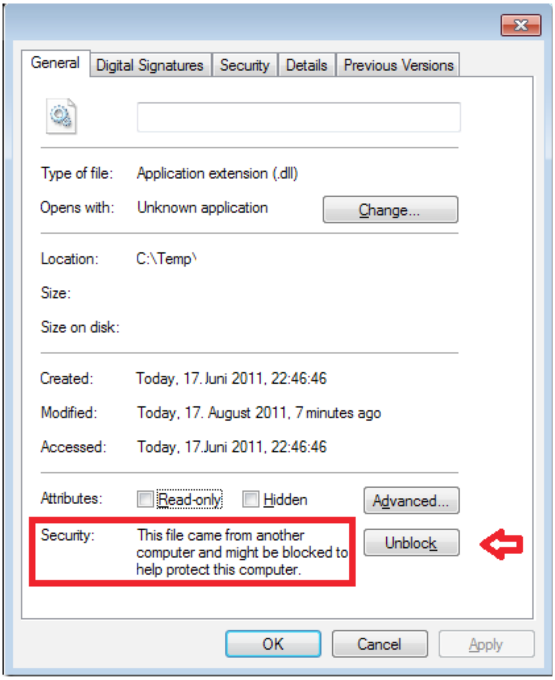
Sometimes CodeRush/DXCore plug-ins may not be loaded correctly. Here is a brief instruction if you'd like to investigate and fix the loading of the plug-in in question.

1) If you develop and compile the plug-in, make sure the plug-in is built into the correct folder. The output path should target to the Community Plug-ins folder. You can check the correct path by clicking the "Plug-ins…" button (or via the right-click context menu and choosing the corresponding menu item) in the "DevExpress -> About" dialog. On Windows 7 it may look like this:

C:\Users\Admin\Documents\DevExpress\IDE Tools\Community\PlugIns

If you got the plug-in from the DXCore Community or other (trusted!) place, simply copy the plug-in assembly in the folder above.

2) Make sure that the plug-in assembly is not blocked by Windows security. In the Windows Explorer, right-click the assembly and choose Properties window. On the General tab click the "Unblock" button if any:



Type of file: Application extension (.dll)

Opens with: Unknown application

Location: C:\Temp\

Size:

Size on disk:

Created: Today, 17. Juni 2011, 22:46:46

Modified: Today, 17. August 2011, 7 minutes ago

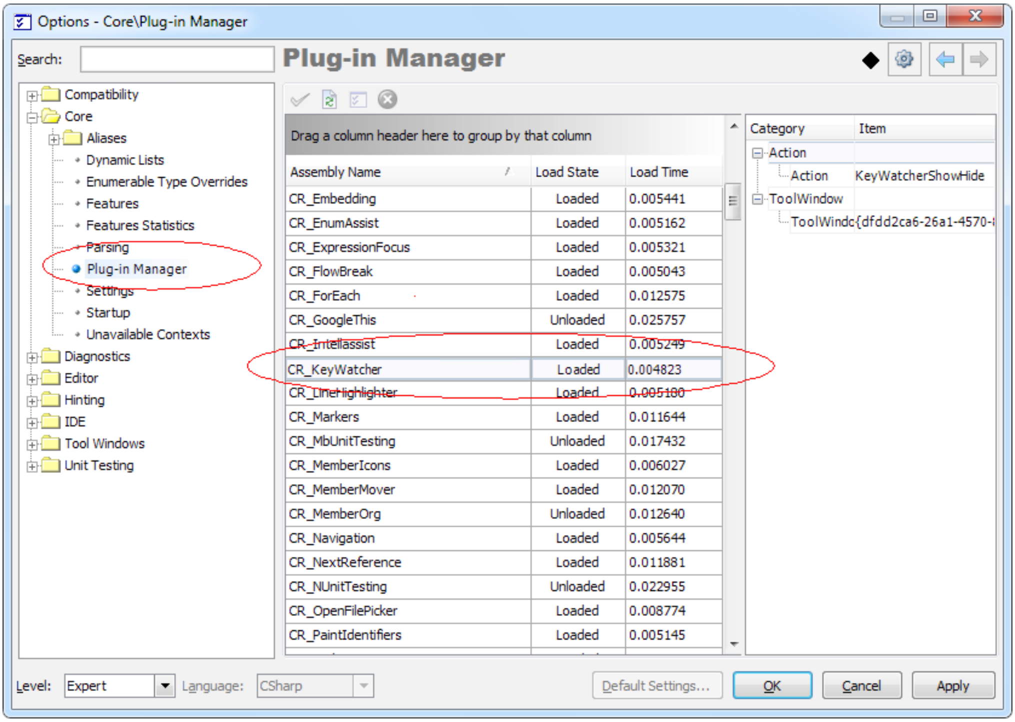
Accessed: Today, 17.Juni 2011, 22:46:46

Attributes: □ Read-only □ Hidden

Security: This file came from another computer and might be blocked to help protect this computer.

DXCore requires this to properly execute the code in the plugin.

3) Inside the Visual Studio IDE, make sure the plug-in is loaded via the **Plug-In Manager**. Follow these steps to get to the Plug-in Manager options page:



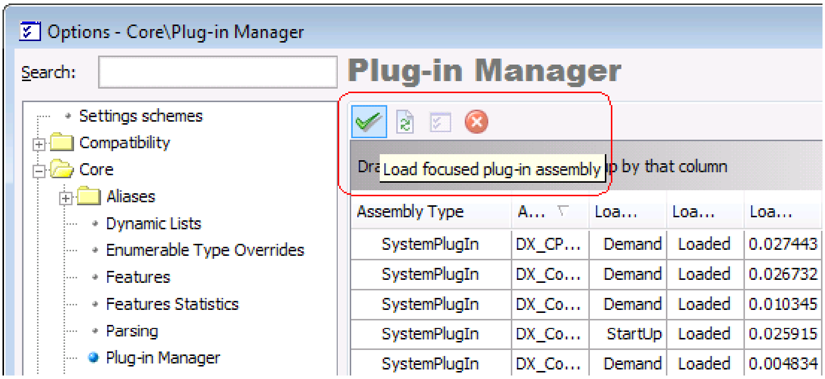
Assembly Name	Load State	Load Time
CR_Embedding	Loaded	0.005441
CR_EnumAssist	Loaded	0.005162
CR_ExpressionFocus	Loaded	0.005321
CR_FlowBreak	Loaded	0.005043
CR_ForEach	Loaded	0.012575
CR_GoogleThis	Unloaded	0.025757
CR_Intellassist	Loaded	0.005249
CR_KeyWatcher	Loaded	0.004823
CR_LineHighlighter	Loaded	0.005180
CR_Markers	Loaded	0.011644
CR_MbUnitTesting	Unloaded	0.017432
CR_MemberIcons	Loaded	0.006027
CR_MemberMover	Loaded	0.012070
CR_MemberOrg	Unloaded	0.012640
CR_Navigation	Loaded	0.005644
CR_NextReference	Loaded	0.011881
CR_NUnitTesting	Unloaded	0.022955
CR_OpenFilePicker	Loaded	0.008774
CR_PaintIdentifiers	Loaded	0.005145

Category	Item
Action	
Action	KeyWatcherShowHide
ToolWindow	
ToolWindc	{dfdd2ca6-26a1-4570-

Level: Expert Language: CSharp

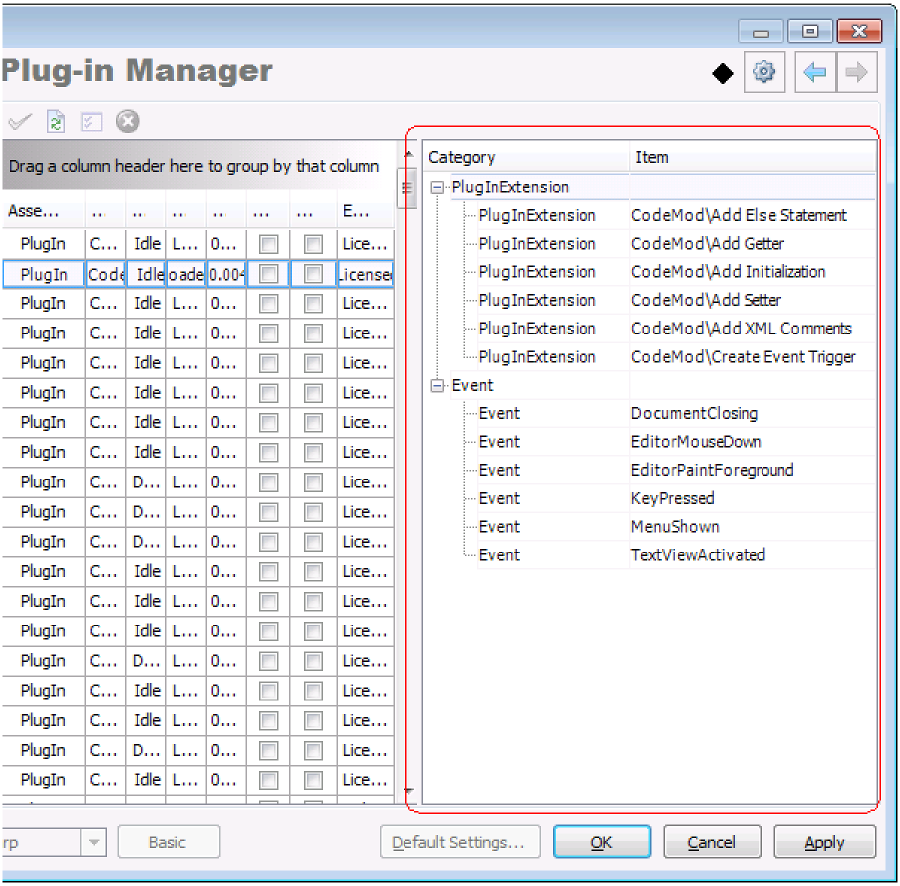
1. From the DevExpress menu, select "Options…";
2. In the tree view on the left, navigate to this folder: "Core";
3. Select the "Plug-in Manager" options page;
4. Find your plug-in and verify its state – Loaded/Unloaded:

If the plug-in is not loaded,click the Load Plug-in Assembly button:



Assembly Type	A…	Loa…	Loa…	Loa…
SystemPlugIn	DX_CP…	Demand	Loaded	0.027443
SystemPlugIn	DX_Co…	Demand	Loaded	0.026732
SystemPlugIn	DX_Co…	Demand	Loaded	0.010345
SystemPlugIn	DX_Co…	StartUp	Loaded	0.025915
SystemPlugIn	DX_Co…	Demand	Loaded	0.004834

If the plug-in is loaded, make sure that all required items were registered on the right side of the Plug-in Manager options page, for example:



Asse…	…	…	…	…	…	…	E…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	Code	Idle	oade	0.004	□	□	License
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	D…	L…	0…	□	□	Lice…
PlugIn	C…	D…	L…	0…	□	□	Lice…
PlugIn	C…	D…	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	D…	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…
PlugIn	C…	D…	L…	0…	□	□	Lice…
PlugIn	C…	Idle	L…	0…	□	□	Lice…

Category	Item
PlugInExtension	
PlugInExtension	CodeMod\Add Else Statement
PlugInExtension	CodeMod\Add Getter
PlugInExtension	CodeMod\Add Initialization
PlugInExtension	CodeMod\Add Setter
PlugInExtension	CodeMod\Add XML Comments
PlugInExtension	CodeMod\Create Event Trigger
Event	
Event	DocumentClosing
Event	EditorMouseDown
Event	EditorPaintForeground
Event	KeyPressed
Event	MenuShown
Event	TextViewActivated

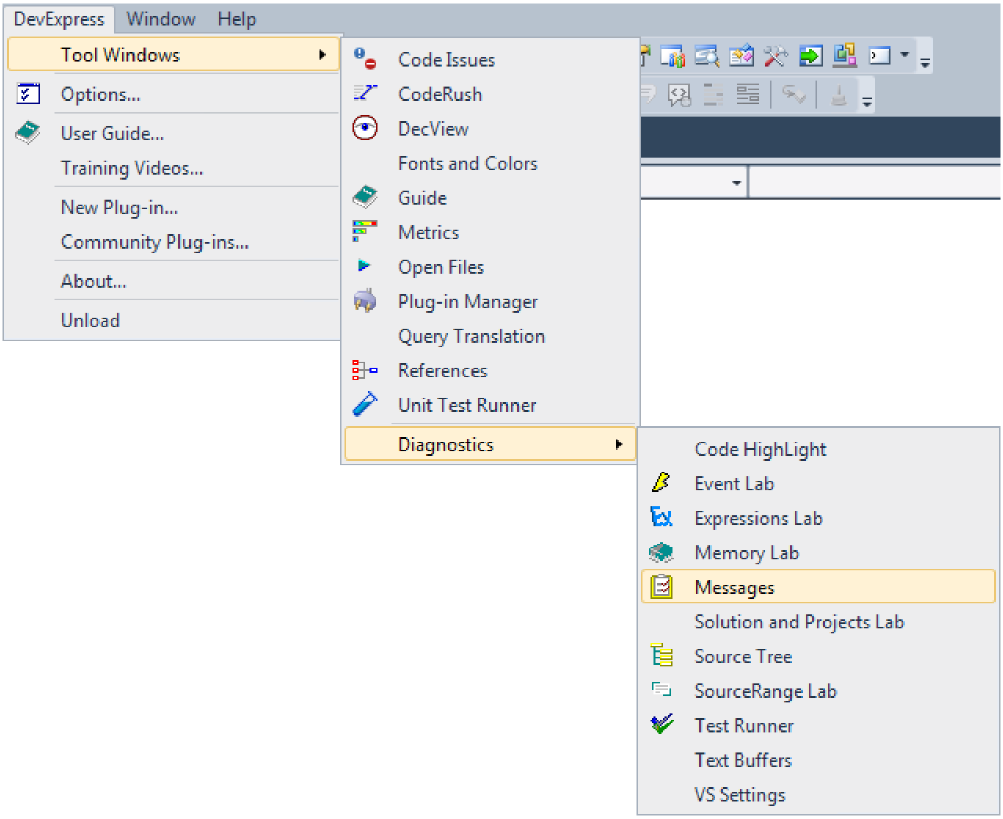
4) If the plug-in was not loaded or does not provide the desired feature, make sure that the feature is enabled. The plug-in may provide an options page which enables or disables the feature. If there is no options page, we can find other plug-in loading clues in the **DXCore** Message Log window.

By default, the message logging is disabled. You can enable it on the **Messages** options page in the **Diagnostics** category of the Options Dialog. After message logging is enabled, restart the Visual Studio IDE, open the Messages tool window from the DevExpress | Tool Windows | Diagnostics | Messages menu item and find out what's going on with this specific plug-in.

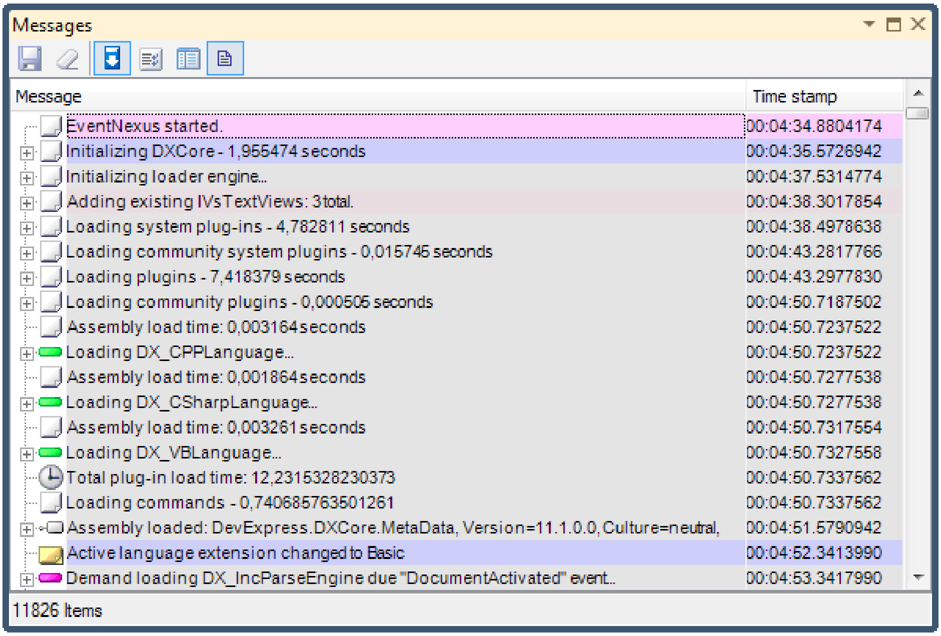
If nothing helps, the DevExpress Support Team is the best source to contact for a help.

Message Log

The **Message Log** is a CodeRush/DXCore diagnostics tool window that records a history of important IDE Tools and Visual Studio events live. The window is useful for diagnostic purposes and finding clues to unexpected **DevExpress IDE Tools** behavior. To open it, click the *DevExpress* | *Tool Windows* | *Diagnostics* | *Message Log* menu item:



Here is what it looks like:



Message	Time stamp
EventNexus started.	00:04:34.8804174
Initializing DXCore - 1,955474 seconds	00:04:35.5726942
Initializing loader engine...	00:04:37.5314774
Adding existing IVsTextViews: 3 total.	00:04:38.3017854
Loading system plug-ins - 4,782811 seconds	00:04:38.4978638
Loading community system plugins - 0,015745 seconds	00:04:43.2817766
Loading plugins - 7,418379 seconds	00:04:43.2977830
Loading community plugins - 0,000505 seconds	00:04:50.7187502
Assembly load time: 0,003164 seconds	00:04:50.7237522
Loading DX_CPPLanguage...	00:04:50.7237522
Assembly load time: 0,001864 seconds	00:04:50.7277538
Loading DX_CSharpLanguage...	00:04:50.7277538
Assembly load time: 0,003261 seconds	00:04:50.7317554
Loading DX_VBLanguage...	00:04:50.7327558
Total plug-in load time: 12,2315328230373	00:04:50.7337562
Loading commands - 0,740685763501261	00:04:50.7337562
Assembly loaded: DevExpress.DXCore.MetaData, Version=11.1.0.0, Culture=neutral,	00:04:51.5790942
Active language extension changed to Basic	00:04:52.3413990
Demand loading DX_IncParseEngine due "DocumentActivated" event...	00:04:53.3417990

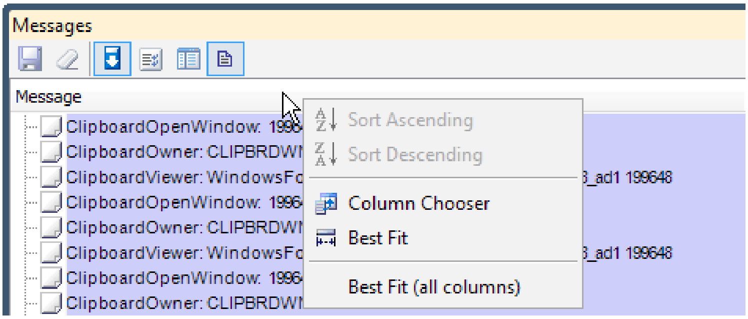
11826 Items

The **Message Log** is disabled, by default. To enable it, go to the *Diagnostics* | *Message Log* options page in the CodeRush Options Dialog and check the *Enabled* check box.

The window contains a toolbar with several buttons, a messages list with several columns, and a status bar. The main message list has two columns, by default:

- Message – the main column, where you can see the specific diagnostic message.
- Time stamp – the time stamp the message arrived on the Log window.

You can configure the columns of the main list by right clicking the top header and choose the appropriate items:



```
ClipboardOpenWindow: 1996
ClipboardOwner: CLIPBRDW
ClipboardViewer: WindowsF              3_ad1 199648
ClipboardOpenWindow: 1996
ClipboardOwner: CLIPBRDW
ClipboardViewer: WindowsF              3_ad1 199648
ClipboardOpenWindow: 1996
ClipboardOwner: CLIPBRDW
```

The items in the header popup menu are:

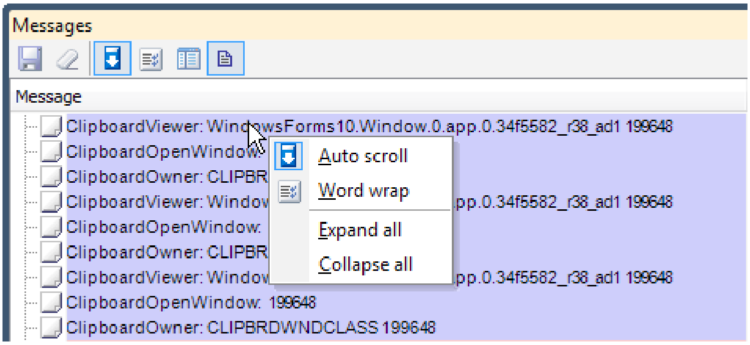
- Best Fit – automatically aligns the active column by width, so the content of the column is completely visible.
- Best Fit (all columns) – automatically aligns all columns by width.
- Columns Chooser – opens a small floating window where you can choose which columns should

be displayed in the main message list.

In addition to the *Message* and *Time Stamp* columns, you can make the following columns visible:

Category	The category of the message.
HasData	Boolean value, indicating whether a message has additional data.
Data	Additional data in a message.
Index	The index of a message.

If you right-click the messages list, and not its header, there's another popup context menu with a few items:



```
ClipboardViewer: WindowsForms10.Window.0.app.0.34f5582_r38_ad1 199648
ClipboardOpenWindow:
ClipboardOwner: CLIPBR
ClipboardViewer: Window                        pp.0.34f5582_r38_ad1 199648
ClipboardOpenWindow:
ClipboardOwner: CLIPBR
ClipboardViewer: Window                        pp.0.34f5582_r38_ad1 199648
ClipboardOpenWindow: 199648
ClipboardOwner: CLIPBRDWNDCLASS 199648
```

Items of this menu are:

- Auto scroll – toggles automatic scrolling of the window on or off for the latest message. If enabled, and the message is posted to the *Message Log*, the list will be automatically scrolled down, so the latest message becomes visible.

- Auto wrap – toggles word wrapping inside the *Messages* column on or off.

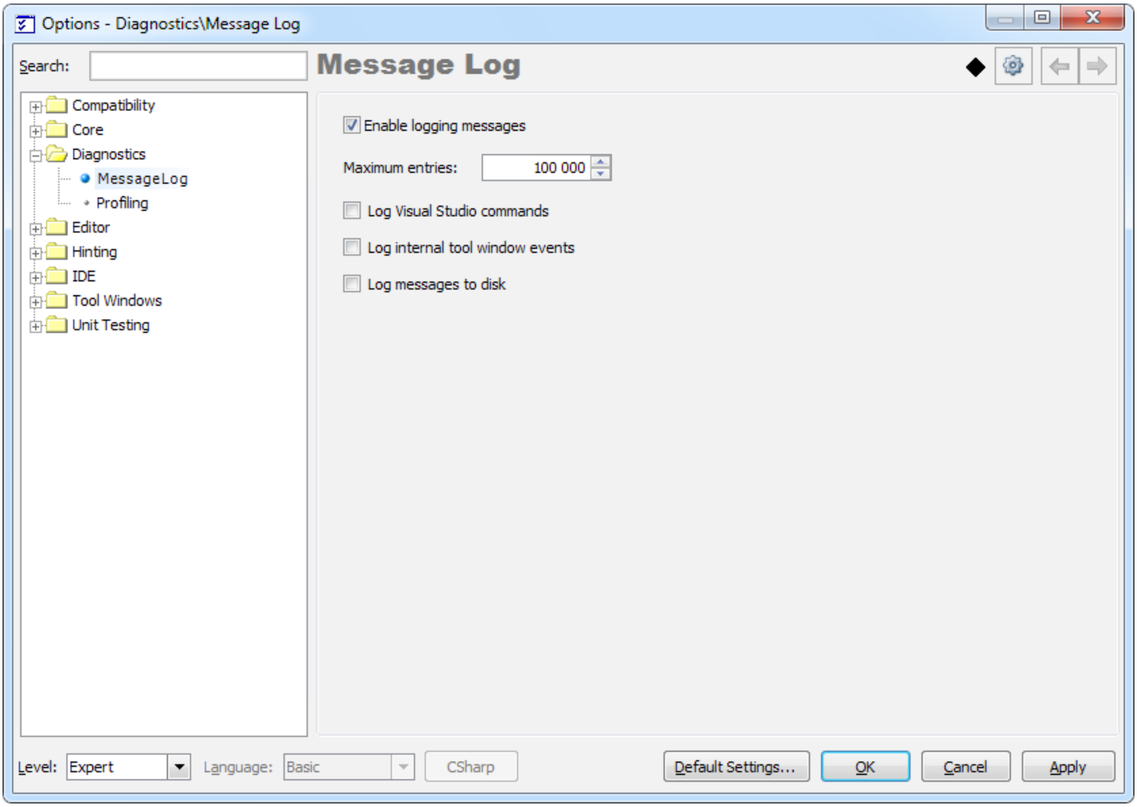
- Expand and Collapse columns allows you mange collapsible items of the *Messages* columns – make all of them visible or collapse messages with multiple items into a single row.

The toolbar has the following buttons:

Icon	Button name	Description
💾	Save log	Saves all messages into a *.dxlog file that you can send to DevExpress Support Services for investigation.
⌫	Clear log	Clears the list of messages.
⬇	Auto scroll	Enables or disables the auto scrolling of the messages list.
≡	Auto wrap	Enables or disables the auto word wrapping of the messages.
▤	Data view	Allows you to choose where you can see the details of a message, containing additional data, such as the exception being thrown. Available positions are: on the left, on the right and don't display.
🗎	Enables/Disables message logging	Toggles the message logging on or off for the current Visual Studio session.

The status bar of the *Messages* tool window shows the total number of messages in the main messages list.

There are additional options available on the *Diagnostics* | *Message Log* options page. Note that the page level is Expert, and will only be visible if the Level combo on the lower-left of the Options dialog is set to Expert:



Options - Diagnostics\Message Log

Message Log

☑ Enable logging messages

Maximum entries: 100 000

☐ Log Visual Studio commands

☐ Log internal tool window events

☐ Log messages to disk

Level: Expert

Language: Basic

Available options are:

- Enable logging messages – turns **DXCore/CodeRush** messages logging on or off.

- Maximum entries – the maximum number of messages in the list. Once the number is exceeded, the old messages are removed and the new ones are added to the end of the main messages list.

- Log Visual Studio commands – allows *Message Log* to intersect and log messages of the Visual Studio commands performed.

- Log internal tool window events – toggles the capability of the logging of events that fired internally inside Visual Studio tool windows

- Log messages to disk – enables or disables live message logging into a disk file. This might be useful when you experience an IDE crash that prevents you from saving messages from the *Message Log* tool window. You can find this file in the *Log* folder of the IDE Tools settings folder.

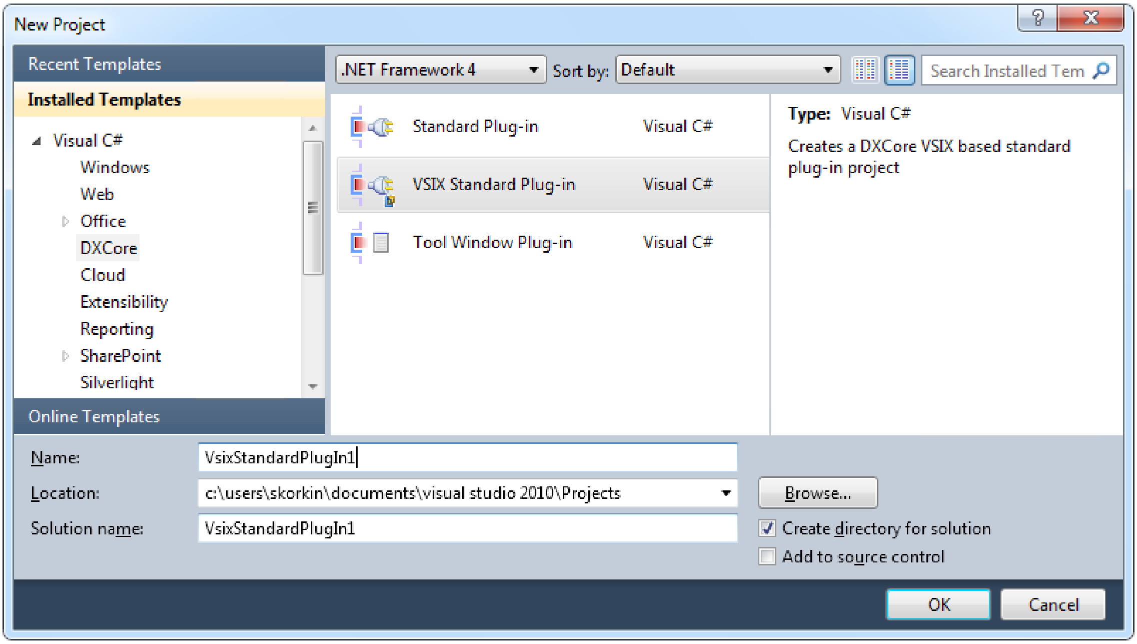
Deploying DXCore or community plug-ins using a VSIX extension

A DXCore plug-in is usually represented by a single assembly. It might also include some additional data, for exnample, language dictionaries for the Spell Checker **CodeRush** plug-in and, probably, setting files, such as shortcuts. To install the plug-in, simply copy an assembly to the Community Plug-ins folder that looks like this in most cases:

```
%Documents%\DevExpress\IDE Tools\Community
```

where *%Documents%* is your *Windows Documents* folder.

The community plug-in path is configurable and can be seen on the **IDE Tools** Settings options page. If a plug-in contains additional files, there should be some instructions on how to install it and where to copy those additional files. Starting with the ١١،٢ release, IDE Tools include a special **VSIX DXCore plug-in** project template that allows you to create a usual **DXCore plug-in** and deploy it through an Extension to the Visual Studio IDE:



.NET Framework 4

Sort by: Default

Type: Visual C#

Creates a DXCore VSIX based standard plug-in project

Name: VsixStandardPlugIn1

Location: c:\users\skorkin\documents\visual studio 2010\Projects

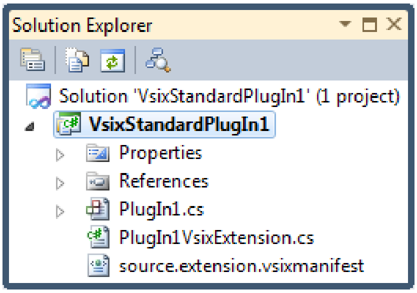
Solution name: VsixStandardPlugIn1

☑ Create directory for solution

□ Add to source control

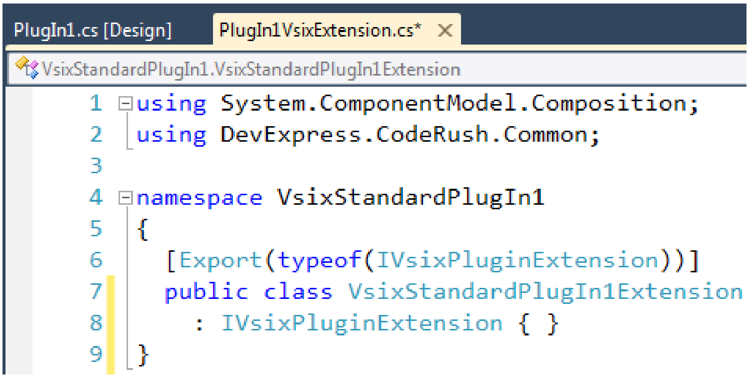
Note that it is necessary to install the Visual Studio SDK to be able to create projects of this type.

Here's a project structure inside the *Solution Explorer*:



Solution 'VsixStandardPlugIn1' (1 project)
VsixStandardPlugIn1
Properties
References
PlugIn1.cs
PlugIn1VsixExtension.cs
source.extension.vsixmanifest

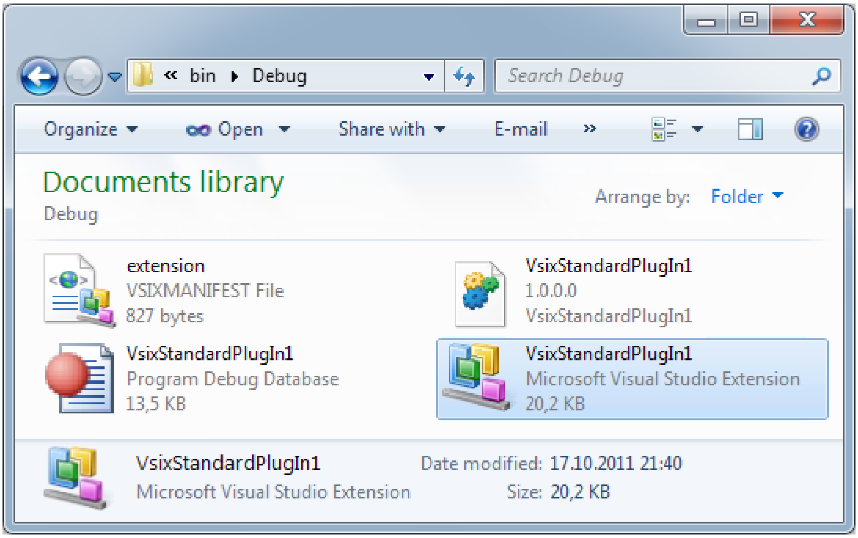
An extension references all assemblies required by DXCore and exports the *IVsixPluginExtension* implementer (from the *DevExpress.CodeRush.Common* assembly):



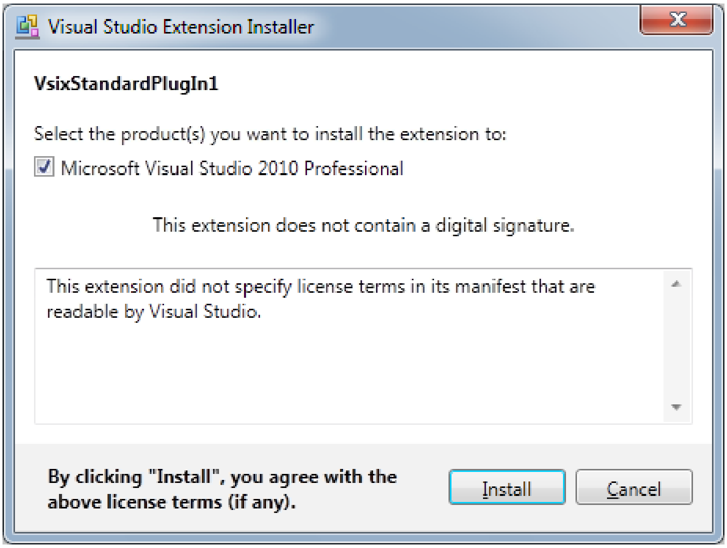
```
using System.ComponentModel.Composition;
using DevExpress.CodeRush.Common;

namespace VsixStandardPlugIn1
{
  [Export(typeof(IVsixPluginExtension))]
  public class VsixStandardPlugIn1Extension
    : IVsixPluginExtension { }
}
```

Then, **DXCore** imports all implementers and uses their assemblies as a location to load additional plug-ins. The VSIX extension may contain multiple plug-in assemblies that will be automatically loaded by DXCore when found.

Once the plug-in is successfully compiled, you will notice a regular **VSIX Extension** file created inside the output folder of the plug-in solution:



If you execute it, you will see the standard Visual Studio Extension Installer dialog:



VsixStandardPlugIn1

Select the product(s) you want to install the extension to:

☑ Microsoft Visual Studio 2010 Professional

This extension does not contain a digital signature.

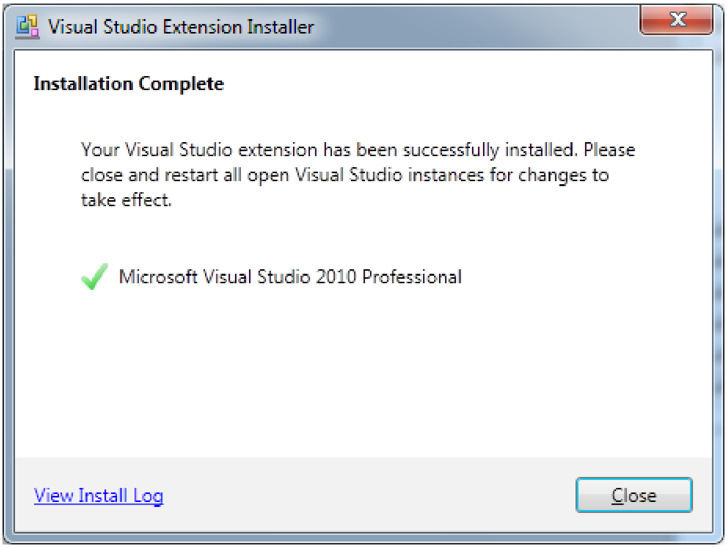
This extension did not specify license terms in its manifest that are readable by Visual Studio.

By clicking "Install", you agree with the above license terms (if any).

If you click Install, the installer will unpack the content of the plug-in to the following folder:

```
%AppData%\Microsoft\VisualStudio\10.0\Extensions
```

and show the result of an installation:

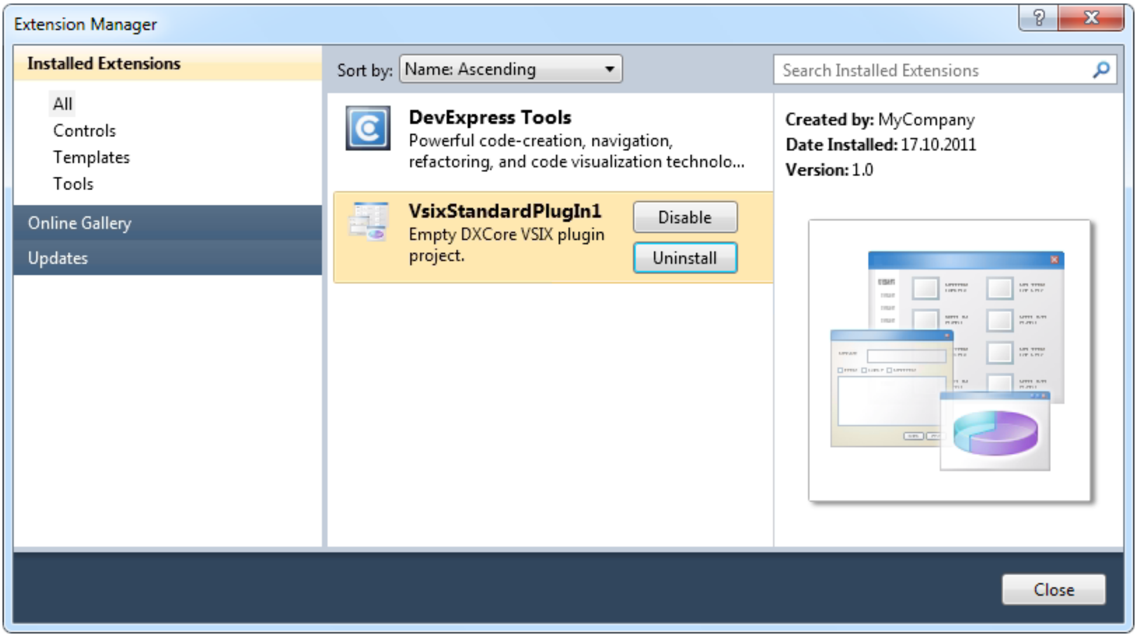


Installation Complete

Your Visual Studio extension has been successfully installed. Please close and restart all open Visual Studio instances for changes to take effect.

✓ Microsoft Visual Studio 2010 Professional

Once the plug-in is installed, you will see it inside the Visual Studio Extension Manager:



DevExpress Tools
Powerful code-creation, navigation, refactoring, and code visualization technolo...

VsixStandardPlugIn1
Empty DXCore VSIX plugin project.

Created by: MyCompany
Date Installed: 17.10.2011
Version: 1.0

You can disable any plug-in extension or completely uninstall it inside the Extension Manager by clicking the corresponding buttons.

Shared Source Samples Solution

CodeRush Pro is shipping with a plug-ins sample solution called “**Shared Source**”. It has several projects with open source plug-ins that are intended for learning plug-ins development using the DXCore framework. This solution is located inside your installation folder, e.g.:

C:\Program Files\DevExpress 2010.1\IDETools\System\CodeRush\SOURCES

You’re welcome to modify and compile the source code as you like.

Here’s the table of plug-ins available with the brief descriptions. Click on the plug-in name for more details, if appropriate.

Plug-in name	Description	Installed by default
CR_ClipboardHistory	One of the biggest source code plug-ins that shows the recent history of clipboard operations in a configurable window. The history can optionally persist across Visual Studio sessions.	✓
CR_CommentHighlighter	This feature paints the ” TODO:” comment part in a low-contrast color, and the rest of the to-do text in a different, higher-contrast color.	✓
CR_CommentPainter	The plug-in with a single “Comment Painter” feature draws a bubble icon over the comment “//” symbols in C#, or “’” (single quote) in Visual Basic. The source code shows how to paint bitmaps on the code editor.	✓

CR_ConvertToInteger	Contains a sample code for the “Convert to Integer” code provider, which converts an expression to an integer, using a call to Math.Ceiling() (or Math.Floor() or Math.Round()).	✓
CR_DropMarkerBeforeJump	The plug-in listens for the “Edit.GoToDefinition” Visual Studio command, and drops a marker before it is executed, so you can easily to go back once you finish navigation to definitions.	✓
CR_EnumAssist	An Intellassist extension plug-in that suggests enumeration vale ues in the appropriate places (e.g. when you’re going to type one of the enumeration elements).	✓
CR_ForEach	Implements the “ForEach” TextCommand, which allows templates to iterate through different kinds of language elements.	✓
CR_GoogleThis	This plug-in allows you to google the word/ identifier at the text caret.	✗
CR_Intellassist	The main plug-in for the Intellassist support of CodeRush.	✓
CR_LineHighlighter	Contains the “Line Highlighter” feature, which paints the current line in the code editor with a different background color.	✓
CR_MemberMover	Allows you to move members into regions by clicking on the Member Icon.	✓
CR_MemberOrg	Organizes members into logical groups that you define. Member Mover can wrap members into regions and specify a comment starting before each members group.	✓
CR_PathAssist	This is another Intellassist extension plug-in that suggests physical paths for files on your hard drive as you type them inside of a string.	✓
CR_RightMarginLine	Paints a vertical line at the column you specify. The line is completely configurable on the appropriate options page, e.g. line color, line style, line width, line opacity, etc.	✓
CR_ShowMetrics	Displays a different kind of metrics information to the left or right of a member declaration inside the code editor.	✓
CR_Sounds	Plays a specific sound file to notify a user when a feature is being performed.	✗
CR_StructuralHighlighting	Draws the light low-contrast vertical lines connecting matching block delimiters (e.g., “{” and “}” in C#, for example).	✓
CR_TemplateGoto	Adds the GoTo text command, which takes a single parameter, used to indicate the type of code to locate and there, insert a text expansion.	✓
CR_VSHelpers	Adds an option to make the most-recently-added project the start-up project. Contains the “Collapse to Projects” feature as well.	✓
CR_XmlDocCommentPainter	Paints over **XML Doc Comments** in the editor, representing them in a nice-looking visual style, improving its readability.	✓

CodeRush object for accessing to DXCore services

The **CodeRush** object provides access to **DXCore**/**CodeRush** services. Historically it was called **CodeRush**, because DXCore framework was not decoupled from the CodeRush product at the beginning; in other words, **DXCore** did not exist at that time. Today, it should technically be called **DXCore**.

The full list of services accessible through CodeRush object can be seen in the corresponding topic.

To programmatically use this object you have to reference the ***DevExpress.CodeRush.Core*** assembly in your project. The object is located in the same named ***DevExpress.CoreRush.Core*** namespace. Note that when you create a DXCore based plug-in this assembly is automatically referenced and the namespace is added for you into the source code of a plug-in.

For example, the **CodeRush** object has a property “**Breakpoint**” of type BreakpointServices, which in turn has properties and methods you can use. This service (**Breakpoint**) provides methods for retrieving and toggling breakpoints in the IDE code editor. If you want to call **BreakpointServices**‘ **Toggle**method, use code like this:

```
CodeRush.Breakpoint.Toggle(lineNumber)
```

In general, every property and method you will need in **DXCore** (**CodeRush**) is accessed similarly to this example

(e.g., **CodeRush** . ***ServiceName*** . ***PropertyOrMethod***).

There is a single method of the CodeRush object:

Name	Description
ShowURL	Opens a web browser page inside Visual Studio IDE with the passed in URL address.

And two events:

Name	Description
Loaded	Fired when CodeRush engine is loaded.
Unloading	Fired when CodeRush engine is being unloaded.

Option Pages and Decoupled Storage overview

You can add custom user-friendly **option pages** that will automatically be integrated into the IDETools Options Dialog. If your DXCore plug-in has any kind of user-configuration, the **options page** will allow you and other developers easily customize the features of your plug-in.

Option pages offer a number of advantages:

- They appear in one place, and are searchable by the user. This provides a consistent look and feel to the application.

- The storage mechanism is decoupled from the options page itself. Settings are stored in a consise tent location on disk, and easily transfer between machines.

- They have custom events that make it easy to read and write settings to disk.

Options pages have a surface of a minimum size of ٤٨٠×٥٣٠ pixels. At runtime, **options pages** can be resized to the bigger size, but they cannot be smaller than the minimum size, so everything needs to fit in this space. If you need more room for your options controls, you might consider a second **options page**, or more traditional strategies for dealing with space constraints, such as scroll bars or perhaps a tab control.

You can use any third-party controls, but it’s really only safe to do this if you’re the only developer who will use your plug-in. If you expect to share plug-ins with other developers who might not have a license to your third-party control, it is recommended to use only the standard .NET controls, or controls that ship with the **DXCore** (the controls located in the **“DXCore**…” groups of the Toolbox). Bear in mind that the appearance of the **Options Dialog** can be changed and **DXCore** controls will change their “look and feel” when another dialog skin is applied.

How to create a new plug-in Options page

This topic is about adding a DXCore Options page. It has three parts to make it easier to read and follow:

1. Adding and designing an options page (**this post**)
2. Implementing the options page settings storing logic
3. Using settings from an options page inside a plug-in

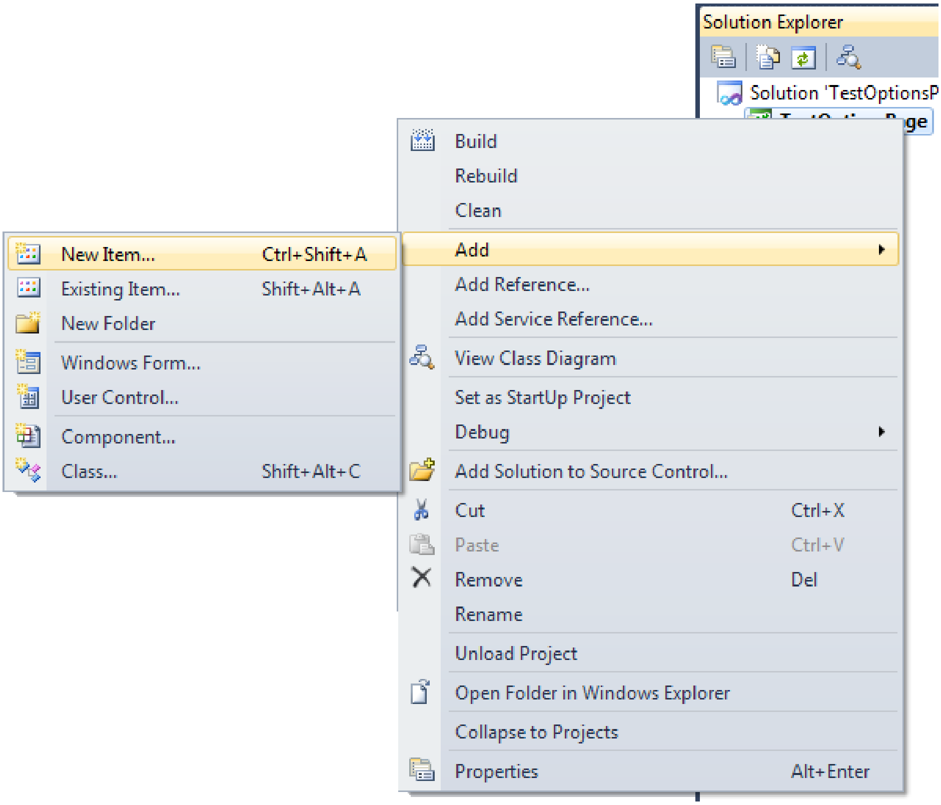
Here's the first part of the topic, and the steps we need to accomplish, to add an options page to your plug-in, and make it work:

1. Run the options page wizard
2. Layout controls on the page
3. Add runtime behavior code for controls, if necessary

Let's take a look at these steps in detail.

The Options Page Wizard

In the Visual Studio's Solution Explorer tool window, right-click your DXCore plug-in project and choose **Add** | **New Item**…



The **Add New Item** dialog appears. Follow these steps to add an **Options page**:

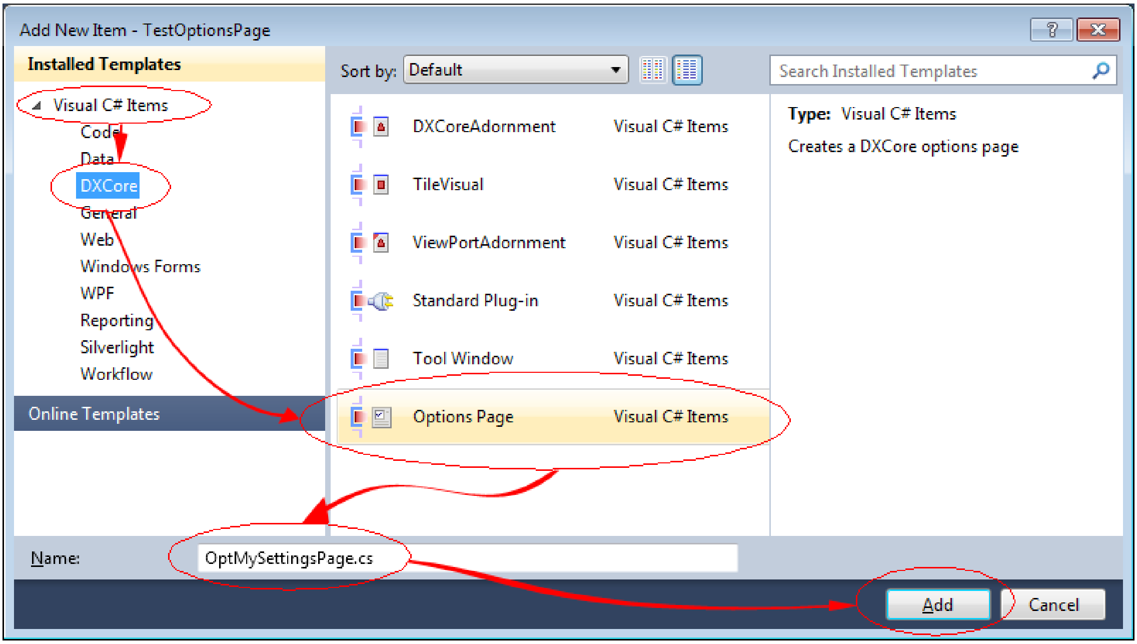
1. Choose the **Language** section of you plug-in (**CSharp** or **Visual Basic**)

2. Select the **DXCore** category inside of the **Language** section

3. Select the **Options Page** template

4. Specify a meaningful name. It is convention to prefix options pages with "**Opt**".

5. Click the **Add** button:



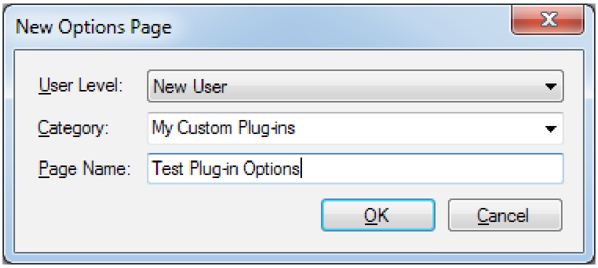
DXCoreAdornment	Visual C# Items
TileVisual	Visual C# Items
ViewPortAdornment	Visual C# Items
Standard Plug-in	Visual C# Items
Tool Window	Visual C# Items
Options Page	Visual C# Items

Type: Visual C# Items

Creates a DXCore options page

Name: OptMySettingsPage.cs

The **New Options Page** dialog will appear:



User Level: New User

Category: My Custom Plug-ins

Page Name: Test Plug-in Options

There are three options to set here:

- **User Level**

The User Level option determines when the page will be seen: for new users, for advanced users, or expert users. This mode is changed on the IDE tools Options Dialog, for example, at the **Expert** level, all options pages are visible.

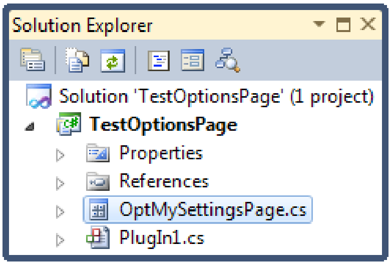
- **Category**

The Category list is populated with all the existing option categories automatically. You can select one of them, or specify your own. Sub-categories are separated with the backslash character (e.g., “**Editor\Painting**“).

- **Page Name**

Enter the page name (it is recommended to use only characters that are legal for file names). You can change all these options later, inside the options page source code, but you’ll save some time if you set them correctly from the beginning.

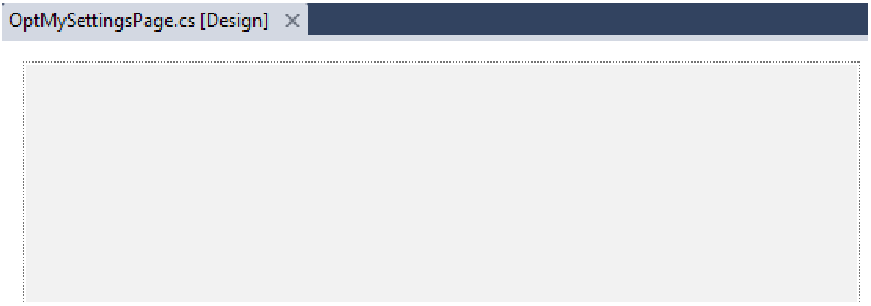
Click **OK**. The options page wizard will generate the new options page item and add it to the active project:



Solution Explorer

Solution 'TestOptionsPage' (1 project)
TestOptionsPage
Properties
References
OptMySettingsPage.cs
PlugIn1.cs

The **Options Page** design surface is activated in the editor after the item is added:



Layout controls on the page

Populate your options page with controls suitable for manipulating your options. If you plan on sharing this options page with other developers outside of your team, it is recommended using only the .NET controls that ship with Visual Studio, or controls that ship with the DXCore (the controls located in the “**DXCore**…” groups of the Toolbox).

Adding runtime behavior code for controls

Here, you might want to fill the control content with appropriate items, group radio buttons, implement drag-n-drop behavior for controls, etc.

The next step is to implement the logic of the options page to correctly read and write plug-in settings.

How to implement the Options page logic for storing settings

Here’s the second part of the post about adding an options page into your DXCore plug-in. See the other parts, to learn more:

1. Adding and designing an options page

2. Implementing the options page settings storing logic (**this post**)

3. Using settings from an options page inside a plug-in

Here are the steps we need to accomplish, to implement a logic for your options page, and make it load and save its settings correctly:

1. **Prepare** the options page (set values for the controls based on the settings in storage)

2. Optionally specify default values (enables the "**Defaults Settings…**" button)

3. **Commit** changes (transfer values on the controls to storage)

Preparing the options page

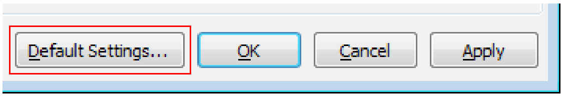
Handle the **PreparePage** event of your new options page. This event is triggered just before your options page is first shown to the user. Use the **ea.Storage** parameter to read the settings from the decoupled storage and assign them to your controls, using code like this:

```
private void OptMySettingsPage_PreparePage(object sender, DevExpress.CodeRush.Core.OptionsPageStorageEventArgs ea)
{
  chkEnabled.Checked = ea.Storage.ReadBoolean(«Preferences», «Enabled», true);
  // ...
}
```

Replace the "// …" in the sample code above with similar calls to transfer the loaded data to other controls on the form as needed.

Specifying default values

It is recommended to provide default settings for your options page. To do this, handle the **RestoreDefaults** event. If you handle this event, the "**Default Settings**…" button will be enabled on the Options Dialog when your page is displayed:



Use code like this, to initialize the controls to their default values:

```
private void OptMySettingsPage_RestoreDefaults(object sender, DevExpress.CodeRush.Core.OptionsPageEventArgs ea)
{
  chkEnabled.Checked = true;
  // ...
}
```

Clicking on the “**Default Settings**…” button will fire this event. Here, you need to adjust properties of the controls on the options page. This event is not supposed to store settings – just change the controls.

Commiting changes

Handle the **CommitChanges** event of your new options page. This event is triggered when the user clicks either the **OK** or the **Apply** button on the IDE tools Options Dialog. Use the **ea.Storage** parameter to write the settings to the decoupled storage, like this:

```
private void OptMySettingsPage_CommitChanges(object sender, DevExpress.CodeRush.Core.OptionsPageStorageEventArgs ea)
{
  ea.Storage.WriteBoolean(«Preferences», «Enabled», chkEnabled.Checked);
  // ...
}
```

These three events take care of saving and loading settings for the options page. The next step is to ensure that your plug-in is notified whenever users make a change on your new options page.

How to load stored settings from an Options page inside of a plug-in

This is the third and the latest part of a topic about adding an options page into your DXCore plug-in. See the other parts, to learn more:

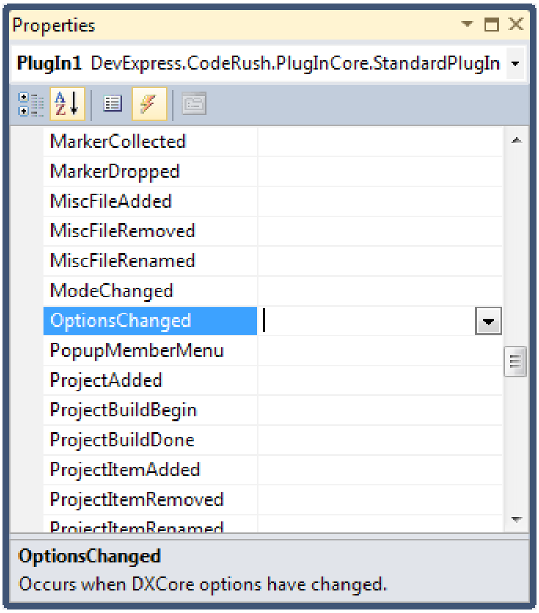
1. Adding and designing an options page

2. Implementing the options page settings storing logic

3. Using settings from an options page inside a plug-in (**this post**)

In this post, we are going to read plug-in settings from the decoupled storage and update them when they are changed on the options page.

In the Visual Studio form designer, activate your standard plug-in design surface. In the *Properties* tool window, click the **Events** button to see the available events. Double-click the **OptionsChanged** event to create an event handler for it.


Properties

PlugIn1 DevExpress.CodeRush.PlugInCore.StandardPlugIn

MarkerCollected
MarkerDropped
MiscFileAdded
MiscFileRemoved
MiscFileRenamed
ModeChanged
OptionsChanged
PopupMemberMenu
ProjectAdded
ProjectBuildBegin
ProjectBuildDone
ProjectItemAdded
ProjectItemRemoved
ProjectItemRenamed

OptionsChanged
Occurs when DXCore options have changed.

The **OptionsChanged** event is fired when the user clicks either the **OK** or the **Apply** button on the IDETools Options dialog (this event is triggered after your custom options page handles its *CommitChanges* event).

Inside the **OptionsChanged** event handler, add the following code to see if your page was changed:

```
private void PlugIn1_OptionsChanged(OptionsChangedEventArgs ea)
{
   if (ea.OptionsPages.Contains(typeof(OptMySettingsPage)))
      UpdateSettings();
}
```

Change "*OptMySettingsPage*" in this sample code to the class name of your options page. Now, it's time to implement the **UpdateSettings** method. It may look like this:

```
void UpdateSettings()
{
   using (DecoupledStorage storage = OptMySettingsPage.Storage)
   {
      _IsEnabled = storage.ReadBoolean(«Preferences», «Enabled», true);
      // ...
   }
}
```

_IsEnabled is a private boolean field that can be declared in this plug-in to hold the state of the option. Replace the "…" comment with similar assignments to other private fields declared in your plug-in.

Make ensure that the default values passed to *ReadBoolean* and other *ReadXxx* methods of the DecoupledStorage object match the default values specified in your *PreparePage* event handler, and in your *RestoreDefaults* event handler. Since these default values are showing up in three places, you might want to make them public static or constant fields. This is especially useful if you expect your default values to be changed often during development.

The final step is to initialize settings according to the stored state (or set them to defaults). The easiest way is to do this on a plug-in startup – just add a call to "**UpdateSettings**" at the end of the "*InitializePlugIn*" method of the standard plug-in instance.

```
// DXCore-generated code...
#region InitializePlugIn
public override void InitializePlugIn()
{
  base.InitializePlugIn();

  UpdateSettings();
}
#endregion
```

That's it – now you have a custom options page for you plug-in!

DXCore DecoupledStorage object for storing data

As the name says, the object handles decoupled storage for DXCore plug-ins. It is recommended for plug-in developers to use **DecoupledStorage** for persisting data (e.g. plug-in settings). The storage has its own structure and saved in a file of a specific format in the appropriate location.

Here are its **methods**:

Name	Description
Clear	Clears the contents of the active language-specific storage object (set through the*LanguageID* property). If *LanguageID* is empty (""), then the primary (language-neutral) storage object is cleared. To clear all storage objects, use*ClearAll*().
ClearAll	Clears the contents of the storage object and any language-specific storage object children. To clear only the active storage object (set through the*LanguageID* property), use the *Clear*() method.
Create	Returns an instance of the new **DecoupledStorage** with the specifed *BasePath*,*Category* and *PageName* properties.
Delete	Physically removes the storage from the system.
DeleteAll	Physically removes the storage and its folders from the system.
DeleteFolder	Physically removes the specified folder from the system.
DeleteKey	Deletes the specified key in the given section.
EraseSection	Deletes the specified section.

GetKeys	Returns a string array of keys for the given section.
GetRootFolders	Returns a collection of the root folders.
GetSections	Returns a string array of sections.
GetSubFolders	Returns a collection of subfolders inside the specified folder. Note that the value can be null if the folder contains subfolders without storage.
LoadSerialized	Loads a serializable object from the specified file name (which should be without a path). If this method is called when this storage object is in a language-specific state (e.g., *LanguageID* = “*CSharp*“), the actual file name will have the LanguageID inserted before the file extension (e.g., “*HelloWorld.bin*” becomes “*HelloWorld.CSharp.bin*“). If this storage object is in a language-neutral state (e.g., *LanguageID* = “”), then the file name is unchanged.
LoadSubFolders	Loads and initializes folders of this storage.
ReadBoolean	Reads a boolean value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadChar	Reads a char value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadColor	Reads a Color value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadDateTime	Reads a DateTime value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadDouble	Reads a Double value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadEnum	Reads a enumeration element value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadInt32	Reads a System.Int32 value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadSingle	Reads a Single value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadString	Reads a string value from the storage object. If the section or key was not found, this method returns the value passed in defaultValue.
ReadStrings	Reads a string array value from the storage object. If the section or key is not found, this method returns the value passed in defaultValue.
ReadXmlNode	Reads a System.Xml.XmlNode value from the storage object. If the name was not found, this method returns null.
SaveSerialized	Saves a serializable object to the specified file name (which should be without a path). If this method is called when this storage object is in a language-specific state (e.g., *LanguageID* = “*CSharp*“), the actual file name will have the LanguageID inserted before the file extension (e.g., “*HelloWorld.bin*” becomes “*HelloWorld.CSharp.bin*“). If this storage object is in a language-neutral state (e.g., *LanguageID* = “”), then the file name is unchanged.
SetStorageObject	Sets the *IStorageObject* implementer to be used for storing data in the particular format. Note that it is hidden from Intellisense.
UpdateStorage	Commits changes to disk. If any language-specific storage object children have been created, that data is also committed to disk.
ValueExists	Returns true if a value exists at the given section and key.
WriteBoolean	Writes a boolean value to the storage object.
WriteChar	Writes a char value to the storage object.
WriteColor	Writes a Color value to the storage object.

WriteDateTime	Writes a DateTime value to the storage object.
WriteDouble	Writes a Double value to the storage object.
WriteEnum	Writes a enumeration value to the storage object.
WriteInt32	Writes a Int32 value to the storage object.
WriteSingle	Writes a Single value to the storage object.
WriteStorageData	Writes all pending data to the storage.
WriteSting	Writes a string value to the storage object.
WriteStrings	Writes a string array value to the storage object.
WriteXmlNode	Writes a System.Xml.XmlNode value to the storage object.

See the "How to use DecoupledStorage to persist your settings" topic, to learn more on storage usage.

DXCore DecoupledStorage structure and settings file format

The DecoupledStorage object is used to handle storage for your DXCore plug-ins data. Usually, **DecoupledStorage** represents a single DXCore Options Page. If you are not going to create an Options Page, the **DecoupledStorage** will represent just named data storage.

The storage has three main identifiers (properties):

- *Category* (i.e. the path to the storage)
- *Name* (e.g. "MyPlugIn Settings")
- *Language* (e.g. C#, C++, VisualBasic)

In case of Options Pages, the *Name* property is used to identify the Page, and the *Category* is intended to identify the path to your page. The path represents the part of the tree on the left of the Options Dialog. IDE tools have their own categories inside of the Dialog. You are able to put your Options Page into any of the existing categories, or create a new one, dedicated specially to your plug-ins.

If you are not going to create an Options Page, the *Name* property is used to identify the storage, and the *Category* property will identify an optional physical path to your storage relative to the global DXCore settings storage path.

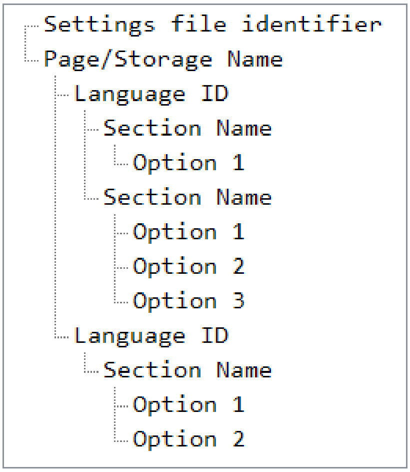
Currently, **DecoupledStorage** is represented by a single file in the XML format. In case you don't like XML format, you can implement your own decoupled storage object via the **IStorageObject** interface implementer. There is another storage implementer exists, which can store your settings in the INI files format.

Structurally, **DecoupledStorage** consists of the following elements:

- Folders
- Sections
- Options

Data (i.e. *Options*) is stored inside of *Sections*. These elements are accessible by name. Each *Option* element stores a single value of a specific type. The *Language* property of the **DecoupledStorage** determines which programming language this particular storage represents. Data for different languages is stored inside of a single storage. *Folders* are intended to represent complex, tree-like objects (e.g. see the Shortcuts options page). Each folder of the storage is stored in a separate file.

The settings file has hierarchical format to correspond to the structure of the **DecoupledStorage**:



```
Settings file identifier
Page/Storage Name
    Language ID
        Section Name
            Option 1
        Section Name
            Option 1
            Option 2
            Option 3
    Language ID
        Section Name
            Option 1
            Option 2
```

Here is an example of such a file:

```
<!--This is a DXCore settings file-->
<Page Name="OptionsDialog">
 <Language Name="*Neutral*">
  <Section Name="Position">
   <Option Name="ScreenHeight" Value="900" />
   <Option Name="ScreenWidth" Value="1440" />
   <Option Name="WindowState" Value="0" />
   <Option Name="Top" Value="157" />
   <Option Name="Left" Value="1453" />
   <Option Name="Width" Value="818" />
   <Option Name="Height" Value="552" />
   <Option Name="SplitOpen" Value="True" />
  </Section>
  <Section Name="Preferences">
   <Option Name="ShowHints" Value="False" />
   <Option Name="LastLanguage" Value="Basic" />
   <Option Name="UserLevel" Value="Expert" />
  </Section>
  <Section Name="Global">
   <Option Name="IsModal" Value="True" />
   <Option Name="EnableSkins" Value="False" />
  </Section>
 </Language>
</Page>
```

Settings file identifier is intended to distinguish **DXCore** storage files from other XML files and written as an XML comment. After the file identifier, the single *Page* element goes, which names the storage (or an Options Page). The name of the page is identical to the name of the file. The *Page* element may contain several or single *Language* elements, which contain *Sections* and *Options* by-turn.

That’s how the **DecoupledStorage** object is organized. It is not really necessary to understand, but it can be useful if you will use this object to persist your plug-in data.

How to use DXCore DecoupledStorage to persist plug-in data

To persist your data using the DecoupedStorage object, use the appropriate *ReadXXX* and *WriteXXX* methods. In case your settings are language dependant, **DecoupledStorage** includes a property “*LanguageID*” that can create and select child storage objects for the purposes of language-specific storage (e.g., to support independent settings for C#, C++, Basic, etc.). Just set the *LanguageID* to the appropriate Language identifier (e.g., “CSharp”, “C/C++”, “Basic”, etc.), and then read and write normally. To activate the language-neutral storage object, set the *LanguageID* to an empty string. If your code is inside a CodeRush Options Page, you can get the *LanguageID* through the Options page’s *LanguageID* property (also, you can handle the *LanguageChanged* event in your Options page to find out when the user wants to create settings for another programming language).

Here is a sample of writing and reading data via the **DecoupledStorage** object:

CSharp code:

- Writing data:

```
using (DecoupledStorage storage = new DecoupledStorage(“MyPlugInData”))
{
  storage.WriteInt٣٢(«Dementions», «Length», 100);
  storage.WriteInt32(«Dementions», «Height», 200);
  storage.WriteInt32(«Dementions», «Width», 300);
  storage.WriteString(«General», «ObjectName», «MyPlugInData»);
}
```

Reading data:

```
int length;
int height;
int width;
string objectName;
using (DecoupledStorage storage = new DecoupledStorage(“MyPlugInData”))
{
  length = storage.ReadInt32(«Dementions», «Length»);
  height = storage.ReadInt32(«Dementions», «Height»);
  width = storage.ReadInt32(«Dementions», «Width»);
  objectName = storage.ReadString(«General», «ObjectName»);
}
```

VisualBasic code:

- Writing data:

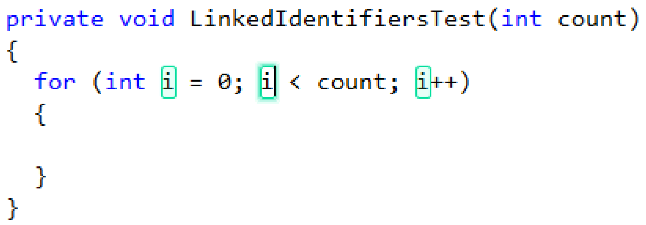
```
Using storage As DecoupledStorage = New DecoupledStorage(“MyPlugInData”)
  storage.WriteInt۳۲(«Dementions», «Length», 100)
  storage.WriteInt32(«Dementions», «Height», 200)
  storage.WriteInt32(«Dementions», «Width», 300)
  storage.WriteString(«General», «ObjectName», «MyPlugInData»)
End Using
```

Reading data:

```
Dim length As Integer
Dim height As Integer
Dim width As Integer
Dim objectName As String
Using storage As DecoupledStorage = New DecoupledStorage(“MyPlugInData”)
  length = storage.ReadInt32(«Dementions», «Length»)
  height = storage.ReadInt32(«Dementions», «Height»)
  width = storage.ReadInt32(«Dementions», «Width»)
  objectName = storage.ReadString(«General», «ObjectName»)
End Using
```

DXCore Linked Identifiers feature

Linked Identifiers are a built-in feature of DXCore, which allows you to simultaneously change similar pieces of the text (code) located in different places. If you change one **linked identifier**, the others that are associated with it will automatically be changed as well. For instance, linked identifiers are enabled when you apply the Rename refactoring.



```
private void LinkedIdentifiersTest(int count)
{
  for (int i = 0; i < count; i++)
  {

  }
}
```

There are two types of **linked identifiers**:

- **Simple**, single-document linked identifiers

These linked identifiers are connected to a single text document only. Link information of such identifiers is not persisted, so if you close and reopen a file, all the links located in that file will be lost.

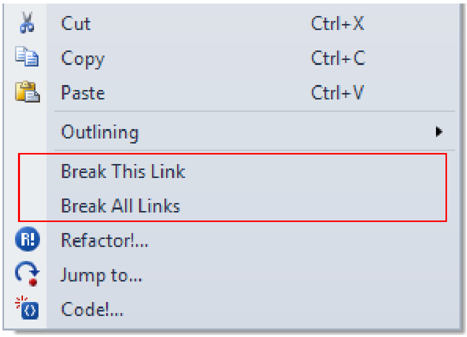
- **Complex**, multi-document linked identifiers

These linked identifiers can exist in several source files, no matter if they are opened or not. If you close all files where associated link identifiers reside, they will stay alive when you reopen one of the files.

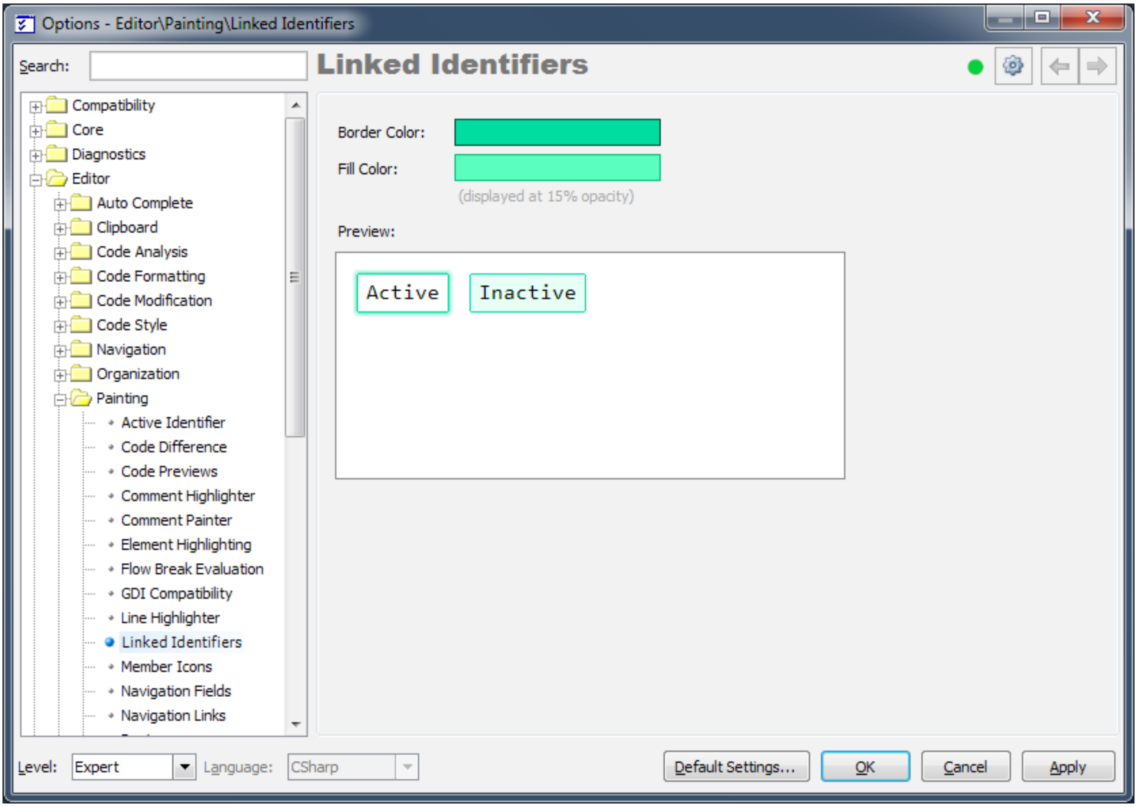
To change the text of a linked identifier, simply place the editor caret inside a link and enter the new text. When the linked identifier is active, the following actions are available:

Shortcut	Action	Description
Enter orNum Enter	Break All Linked Identifiers	Accepts changes in all linked identifiers in the currently active list.
Ctrl+Enter	Break Linked Identifier	Removes the currently active linked identifier from the active list.
Tab	Select Next Linked Identifier	Selects the next linked identifier.
Shift+Tab	Select Previous Linked Identifier	Selects the previous linked identifier.

Breaking identifiers actions are available via the code editor context menu as well:



You can customize the key bindings on the Shortcuts options page in the Options Dialog. Appearance options for the linked identifiers are available on the **Editor** | **Painting** | **Linked Identifiers** options page:



Border Color: 🟩

Fill Color: 🟩

(displayed at 15% opacity)

Preview:

Active Inactive

Level: Expert

Language: CSharp

DXCore adornments architecture

The Visual Studio 2010 IDE shell has been rewritten using the Windows Presentation Foundation (**WPF**), in other words, it has a completely different code editor based on the new **WPF** technology. Earlier versions of DXCore used **GDI** and Win٣٢ API calls to paint the inside code editor before Visual Studio ٢٠١٠ release, and would not work inside the new code editor. To bring painting support to Visual Studio ٢٠١٠ and leave the support of previous Visual Studio versions, **DXCore** has been also rewritten using the new painting abstraction layer, which has a split code base for different platforms (**WPF**, **GDI**). This abstraction layer is called the "**Adornments**" architecture. This architecture allows having a single code base for all versions of Visual Studio which helps to maintain both graphic platforms at once and have independent painting, non-dependant on the version of IDE used.

There are two main objects of the **DXCore adornments** architecture used:

1. **TextViewAdornment** (for **TextView** object) – is a base type of a code editor adornment object. An **adornment** is a graphical object drawn inside code editor. It can be a line, pixel, arrow, string, etc.

2. **TextDocumentAdornment** (for **TextDocument** object). Once it is added to a particular text document, it will create adornments (descendants of the **TextViewAdornment**) for each currently available (and opened/closed as well) text views of this text document automatically.

So, the **TextDocumentAdornment** object creates **TextViewAdornments** for all text views, and the **TextViewAdornment** object draws the graphical object itself. Having said that, first, you need to create a **TextDocumentAdornment** descendant object. Once you create it, you have to override the public**NewAdornment** method, which will create **TextViewAdornment** objects that has the following signature:

```
protected abstract TextViewAdornment NewAdornment(string feature, IElementFrame frame);
```

In this method, you need to create a specific **VisualObjectAdornment**. This is only one descendant of the **TextViewAdornment** object. The **NewAdornment** method must return the newly created **VisualObjectAdornment**. The **VisualObjectAdornment** object allows you to paint inside VisualStudio code editor and represent a graphical object (i.e. adornment). To make this happen, you need to override the **Render** virtual method of this object with the following signature:

```
public virtual void Render(IDrawingSurface context, ElementFrameGeometry geometry)
{
{
```

All painting logic must be located in this overridden method. There are two parameters passed to this method which are used to paint on the code editor:

1. The "**context**" of type "**IDrawingSurface**" – the code editor context on which painting happens. It has the following useful painting methods:

- DrawArrow
- DrawBezier
- DrawBezierLine
- DrawCircumference
- DrawCorner
- DrawEllipse
- DrawImage
- DrawLine
- DrawObject
- DrawPixel
- DrawPolygon
- DrawRectangle
- DrawSelectionBar
- DrawString
- etc

All these methods draw the corresponding object of the same method name.

2. The "**geometry**" of type "**ElementFrameGeometry**" – provides geometry information of an adornment (such as its *size*, *bounds*, *location*, *start point*, *end point*, *etc*). This information is calculated inside the **TextDocumentAdornment** constructors. All constructors accept one or several parameters which specify the coordinates of the text related

adornment. So, it means that an adornment is bound to a text (source code) inside a code editor which coordinates are specified by the *SourcePoint*, *SourceRange* or *DocPoint*, *DocRange* objects. This **ElementFrameGeometry** is initialized from the "**binding**" parameter of the "**IElementFrame**" type of the **NewAdornment** method, which stores all adornment geometry information passed into **TextDocumentAdornment** constructors.

There are several CodeRush open source samples which use adornment architecture to paint on the code editor. You might want to take a look at one of the following plug-ins to learn more on how to paint on the code editor: CR_CommentHighlighter, CR_CommentPainter, CR_StructuralHighlighting, CR_XmlDocCommentPainter.

There are also several project item wizards made for simplifying the visual adornments creation process.

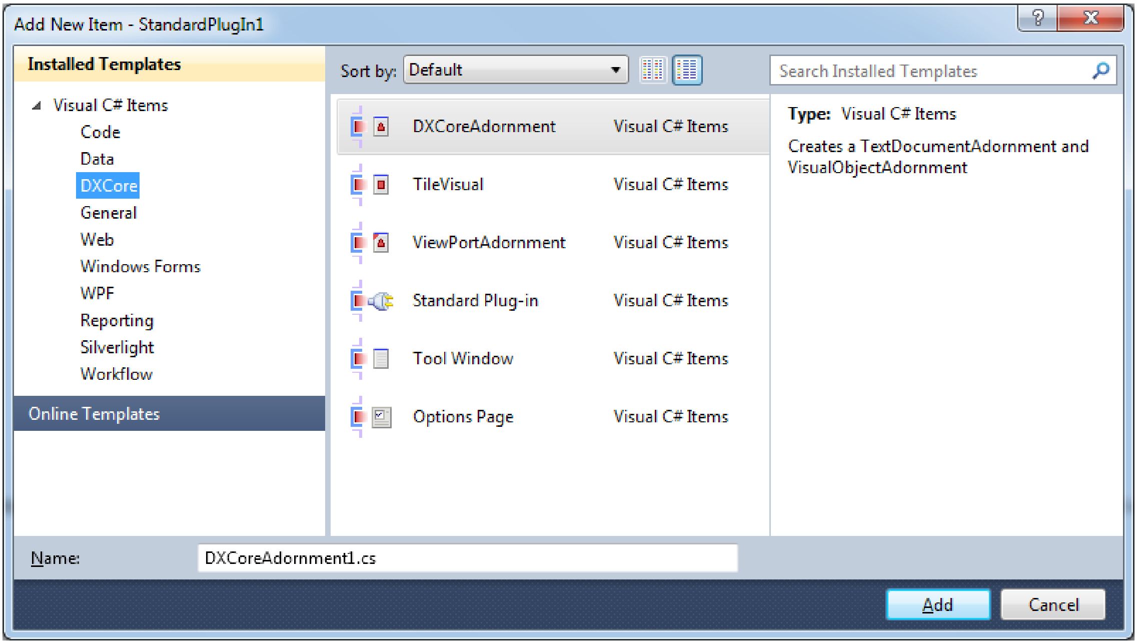
DXCore adornments creation and wizards

There are several project item wizards made for simplifying the DXCore visual adornments creation process:

- **DXCoreAdornment** – widespread adornments project item which consists of two main adornment objects bound to text coordinates: **TextDocumentAdornment** and **VisualObjectAdornments**.

- **ViewPortAdornment** – contains of **EditorAdornment** and **ViewPortAdornment** which can be added to a *TextDocument* or *TextView* and bound to screen or text view coordinates.

- **TileVisual** – contains of two DXCore tile adornment objects: **TextDocumentTile** and **TileVisual** to create tiles in the code editor. Tile adornments can react on mouse events.


Add New Item - StandardPlugIn1

Sort by: Default

Type: Visual C# Items

Creates a TextDocumentAdornment and VisualObjectAdornment

Name: DXCoreAdornment1.cs
Installed Templates
Visual C# Items
Code
Data
DXCore
General
Web
Windows Forms
WPF
Reporting
Silverlight
Workflow
Online Templates
Search Installed Templates
DXCoreAdornment Visual C# Items
TileVisual Visual C# Items
ViewPortAdornment Visual C# Items
Standard Plug-in Visual C# Items
Tool Window Visual C# Items
Options Page Visual C# Items
Add
Cancel

Here's the code of these project items:

DXCoreAdornment:

```
using System;

using System.Collections.Generic;

using DevExpress.DXCore.Adornments;
using DevExpress.DXCore.Platform.Drawing;
using DevExpress.CodeRush.Core;
using DevExpress.CodeRush.StructuralParser;

namespace StandardPlugIn1
{
  class DXCoreAdornment1DocumentAdornment : TextDocumentAdornment
  {
    public DXCoreAdornment1DocumentAdornment(SourceRange range)
      : base(range) { }
    protected override TextViewAdornment NewAdornment(string feature, IElement-
Frame frame)
    {
      return new DXCoreAdornment1ViewAdornment(feature, frame);
    }
  }
  class DXCoreAdornment1ViewAdornment : VisualObjectAdornment
  {
    public DXCoreAdornment1ViewAdornment(string feature, IElementFrame frame)
      : base(feature, frame) { }
    public override void Render(IDrawingSurface context, ElementFrameGeometry
geometry)
    {
      // TODO: Add adornment painting logic
      context.DrawRectangle(null, Colors.Red, geometry.StartRect);
      context.DrawRectangle(null, Colors.Green, geometry.EndRect);
      context.DrawRectangle(null, Colors.Blue, geometry.Bounds);
    }
  }
}
```

ViewPortAdornment:

```
using System;
using System.Collections.Generic;
using DevExpress.DXCore.Adornments;
using DevExpress.DXCore.Platform.Drawing;
using DevExpress.CodeRush.Core;
using DevExpress.CodeRush.StructuralParser;
```

```
namespace StandardPlugIn1
{
  class ViewPortAdornment1DocumentAdornment : EditorAdornment
  {
    public ViewPortAdornment1DocumentAdornment() { }
    protected override TextViewAdornment GetTextViewAdornment(TextView textView)
    {
      return new ViewPortAdornment1Adornment(textView);
    }
  }
  class ViewPortAdornment1Adornment : ViewPortAdornment
  {
    public ViewPortAdornment1Adornment(TextView textView)
      : base(textView, Point.Zero, false) { }
    public override void Render(IDrawingSurface context, ElementFrameGeometry geometry)
    {
      // TODO: Add adornment painting logic
    }
  }
}
```

TileVisual:

```
using System;
using System.Collections.Generic;
using DevExpress.DXCore.Adornments;
using DevExpress.DXCore.Platform.Drawing;
using DevExpress.CodeRush.Core;
using DevExpress.CodeRush.StructuralParser;


namespace StandardPlugIn1
{
  class TileVisual1DocumentAdornment : TextDocumentTile
  {
    public TileVisual1DocumentAdornment(DocPoint start, DocPoint end, CoreEventHub master, object obj)
      : base(start, end, master, obj) { }
    protected override TextViewAdornment NewAdornment(string feature, IElementFrame binding)
    {
      TileVisual1Adornment newAdornment = new TileVisual1Adornment(binding);
      newAdornment.Cursor = Cursor.Arrow;
      return newAdornment;
    }
  }
  class TileVisual1Adornment : TileVisual
  {
    public TileVisual1Adornment(IElementFrame binding)
      : base(binding) { }
    public override void Render(IDrawingSurface context, ElementFrameGeometry 
geometry)
    {
      // TODO: Add adornment painting logic
    }
  }
}
```

After a file with adornment objects is created, we need to add it to a **TextDocument** to make the adornments work. Here are a few ways to achieve this:

1) Simply add it to an active document anytime you want like this:

```
TextDocument document = CodeRush.Documents.ActiveTextDocument;

SourcePoint carePosition = CodeRush.Caret.SourcePoint;

DXCoreAdornment1DocumentAdornment docAdornment = new DXCoreAdornment1DocumentAdornment(carePosition);

document.AddAdornment(docAdornment);
```

This code gets the active text document, the current text caret position and adds the newly created adornment to the text document. To remove an adornment, call the **RemoveAdornments** on the *TextDocument* object:

```
document.RemoveAdornment(docAdornment);
```

2) The second and the preferred way is to add an adornment at the time the code editor is going to paint language elements. For this purpose, the **EventNexus.DecorateLanguageElement** event should be used. Inside the **InitializePlugIn** overridden method of your **StandardPlugIn** instance, subscribe to the**DecorateLanguageElement** event:

```
public override void InitializePlugIn()

{

  base.InitializePlugIn();

  EventNexus.DecorateLanguageElement += ehDecorateLanguageElement;
}
```

Don’t forget to unsubscribe from the event inside the **FinalizePlugIn**:

```
public override void FinalizePlugIn()

{

  EventNexus.DecorateLanguageElement -= ehDecorateLanguageElement;

  base.FinalizePlugIn();


}
```

Inside the **ehDecorateLanguageElement** event handler, create an instance of an adornment and add it to the **DecorateLanguageElementEventArgs**:

```
void ehDecorateLanguageElement(object sender, DecorateLanguageElementEventArgs
args)

{

  LanguageElement activeElement = args.LanguageElement;

  if (CanDrawLanguageElement(activeElement))

  {

    DXCoreAdornment\DocumentAdornment docAdornment = new DXCoreAdornment1Docu-
mentAdornment(activeElement);

    args.AddAdornment(docAdornment);

  }


}
```

In the **CanDrawLanguageElement** method, you may perform some requirements, e.g. filter language elements you want to be painted. There’s no need for manual removal of adornments in this case, because if the requirements are not met (e.g. no language elements to paint) then adornments won’t be added at all.

DXCore abstract source tree structure

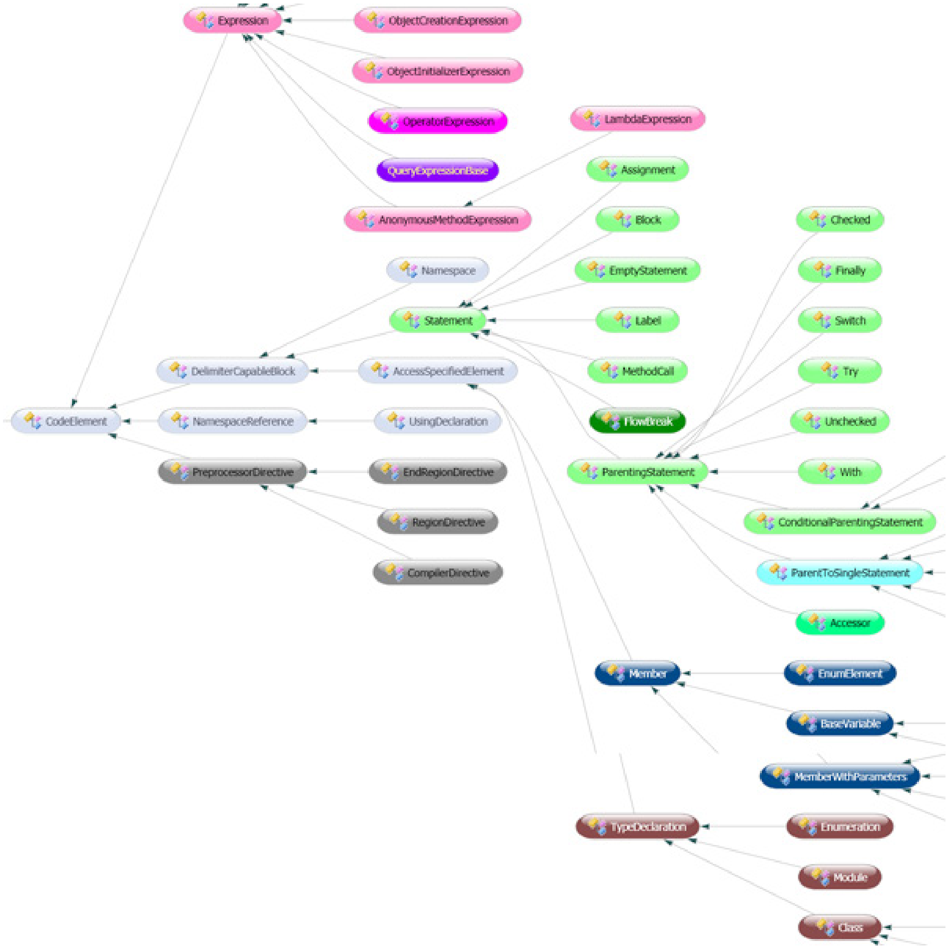
DXCore supports lots of programming languages provided by the language services in Visual Studio. These services provide language-specific support (such as *CSharp*, *Visual Basic*, *C++*) for editing source code in the integrated development environment (IDE). DXCore includes appropriate source code parsers for these programming languages. When the source code is parsed, **DXCore** builds an **abstract syntax tree** (**AST**) that is a representation of the syntactic structure of the source code of a particular programming language. This **abstract syntax tree** is unified for all languages supported by IDETools. That’s why it is easy to develop a language-independent feature as a DXCore plug-in – in most cases your feature will be available in all corresponding languages automatically.

The **abstract syntax tree** is represented by the **DXCore language elements**. **LanguageElement** is the main base class for a rich variety of classes which denote a construct occurring in the source code and present a corresponding node in the unified tree. You can see the result parsed source tree using theExpression Lab tool window available via the “*DevExpress* | *Tool Windows* | *Diagnostics* | *Expression Lab*” menu item in Visual Studio.

All **DXCore** source code parsers are maintained by a single assembly named “*DevExpress.DXCore.Parser*”. This assembly can be used for code parsing and code generation (to build language-specific source code). You can use it according to the DevExpress license agreement.

Here is a complete map of **LanguageElement** class inheritance, excluding some very rarely used language elements. Click on one of the images to open it in full size. Bear in mind that the size of the images is HUGE, and you have to scroll it to see the content. Click on the picture once again to close it.

Linear view:

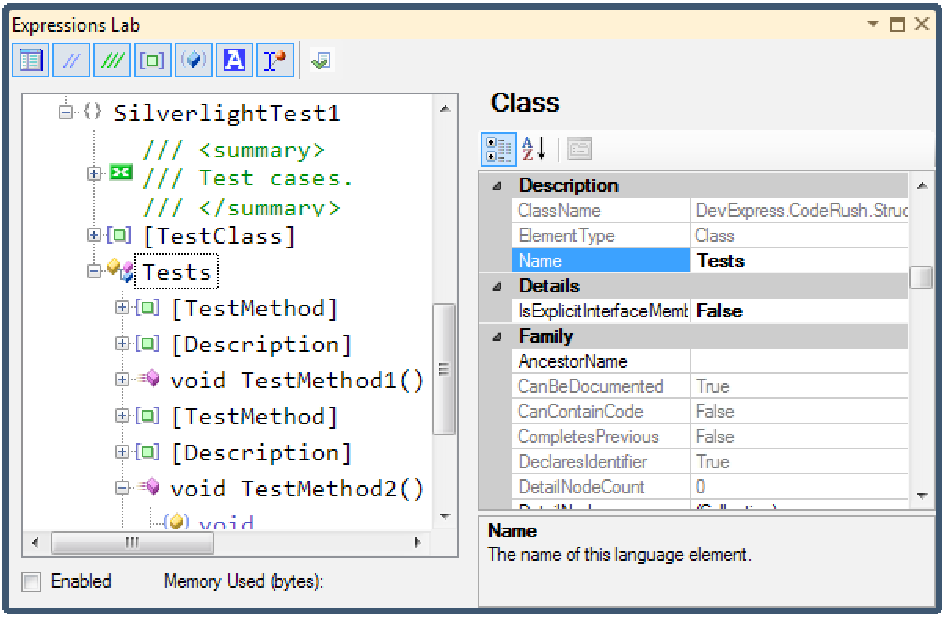


Clustered view:

DXCore Expression Lab tool window

The **Expression Lab** is a DXCore diagnostic plug-in containing the tool window that displays the hierarchical abstract source tree built by the **DXCore**. This is useful to learn the structure of the parsed source code and created a source tree to build your own DXCore plug-ins. When you know the structure of the tree, you can build your own trees or its parts and generate the appropriate code for all programming languages supported by the DXCore. Also, you can see the set of properties each language element has, to learn more on how to construct any specific elements.

This is what the tool window look like:



```
SilverlightTest1
    /// <summary>
    /// Test cases.
    /// </summary>
    [TestClass]
    Tests
        [TestMethod]
        [Description]
        void TestMethod1()
        [TestMethod]
        [Description]
        void TestMethod2()
            void
```

Class

Description	
ClassName	DevExpress.CodeRush.Struc
ElementType	Class
Name	**Tests**
Details	
IsExplicitInterfaceMemt	**False**
Family	
AncestorName	
CanBeDocumented	True
CanContainCode	False
CompletesPrevious	False
DeclaresIdentifier	True
DetailNodeCount	0

Name
The name of this language element.

On the left you can see the resulting hierarchical source tree of the active source file opened inside the IDE. The tree contains all language elements parsed by the **DXCore** and constructed into a single hierarchical tree. Each language element has its own icon to easier distinguish between them. You can collapse or expand nodes of the tree if necessary. By default, all nodes are collapsed, because too many nodes may distract from learning the specific parts of the source tree. If you would like to see a part of the tree for the specific source code, click inside the code editor and the **Expression Lab** will expand the appropriate nodes automatically, then select the part of the tree that represents the selected piece of code.

On the right side of the window, you can see the grid with the properties of the selected language element. The grid, containing the properties, can be categorized, either where properties are grouped by its categories, or listed alphabetically as a plain list. To toggle the view, click the corresponding button above the grid. On top of the properties grid, you can see the element type of the selected language element. Under the grid, you can see the name of the property and its description.

The toolbar has several buttons that toggle different options of the **Expression Lab** window. Here they are:

Icon	Button	Description
▤	Show/hide property grid	Toggles the visibility of the properties grid on the right part of the tool window.
//	Show Comments	Toggles the visibility of the parsed code comments in the hierarchical source tree.
///	Show XML Doc Comments	Toggles the visibility of the parsed XML Doc comments in the hierarchical source tree.
[□]	Show Attributes	Toggles the visibility of the parsed code attributes in the hierarchical source tree.

◈	Show detail nodes	Toggles the visibility of the detail nodes. Details nodes are language elements that do not represent the body of an element but its important characteristics. For example, if you have a conditional statement, its expression is a detail node, and its content is represented by language element nodes. Detail nodes are rendered with blue text in the tree.
A	Select code on click	Toggles the ability to select the source code inside the code editor when you click on the nodes of the hierarchical source tree.
I	Track Caret	Toggles the ability to expand the hierarchical source tree nodes automatically once the editor caret position is changed.
✓	Validate parse tree	Validates the hierarchical source tree for corrupted nodes. Shows a message box with the results of the validation.

And, the last capability for this window is that you can check how much memory is allocated to any part of the hierarchical source tree. To check that, click the *Enabled* checkbox below the source tree near the *Memory Used* label, and you will see the amount of memory used to store language elements for the selected part of the source tree.

Also, there is the *RunExpressionLab* action, to show the tool window using the keyboard shortcut. By default, this action is not bound to any shortcut. You can do this yourself if you plan to use the **Expression Lab** window while developing **DXCore plug-ins**.

How to enumerate solution and source code items

One of the trivial tasks when developing a DXCore plug-in is the enumeration of the active solution items, such as projects, source files, then interfaces, classes, methods, properties, statements, etc. A similar task is to get an active element (in other words, the element where the editor caret is located) inside the active source file to start working with one.

All of the items of the solution are represented by the DXCore classes, located in the "*DevExpress.DXCore.Parser*" assembly inside the "*DevExpress.CodeRush.StructuralParser*" namespace. Consider, we have a standard Visual Studio solution, **DXCore** uses the following classes to represent its hierarchy:

Class Name	Description
SolutionElement	Represents a Visual Studio solution.
ProjectElement	Represents a Visual Studio project. Holds a list of all files belonging to the project.
SourceFile	Represents a file (source code, resource file, bitmap file, XML page, HTML page, etc).

All elements of the file, containing source code, are parsed as DXCore abstract source tree structure, maintained by the language elements. These are the most often used elements inside a source code file:

Class Name	Description
Namespace	Represents a namespace declaration, e.g. "namespace N { }" (C#), "Namespace N … End Namespace" (VB).
NamespaceReference	e.g. "using System;" (C#), "Imports System" (VB)
Class, Struct, Interface, Enumeration	These are different kinds of type declarations.
Method, Property, Event, Variable	These are different kinds of type member declarations.

Hundreds of the elements consisting of statements and expressions, such as: *For, ForEach, If, Return, Throw, ElementReferenceExpression, TypeReferenceExpression, PrimitiveExpression*	These are different kinds of code blocks inside members.

To get an element of the solution, use the SourceModel DXCore service; it has numerous properties to work with the solution and code structure, such as: *ActiveSolution*, *ActiveProject*, *ActiveClass*, *ActiveMethod*, *ActiveProperty*, etc. For example, let's count the number of properties and constants declared inside the entire solution. Here's a sample code:

CSharp code:

```
void DoCount ()
{
  int propertiesCount = 0;
  int constantsCount = 0;
  SolutionElement activeSolution = CodeRush.Source.ActiveSolution;
  foreach (ProjectElement project in activeSolution.AllProjects)
    foreach (SourceFile sourceFile in project.AllFiles)
      foreach (TypeDeclaration typeDeclaration in sourceFile.AllTypes)
      {
        foreach (Property property in typeDeclaration.AllProperties)
          propertiesCount++;
        foreach (Const constant in typeDeclaration.AllConstants)
          constantsCount++;
      }
  Console.WriteLine(«Number of properties inside solution: « + propertiesCount);
  Console.WriteLine(«Number of constants inside solution: « + constantsCount);
}
```

As you can see, we get an instance of the active solution, then enumerating over all of its projects, then all files of all projects, then all types of all files and, finally, the members we need – properties and constants. Bear in mind, that if there's no solution opened, the *ActiveSolution* will return *null* (C#) or *Nothing*(VB).

For enumerating inner code blocks of members, there are properties like *AllStatements, AllExpressions, AllVariable* on the instance of a member language element. But these properties might be not enough if you would like to enumerate every piece of the code inside member in detail. To enumerate everything, you can use the **Nodes** and **DetailNodes** properties of the particular language element. **Nodes** property contains high-level root elements like statements and the **DetailNodes** property contains granular details specific to this particular element. For example, let take a look at this ‘for’ loop in **CSharp**:

```
for (int i = 0; i < 100; i++)
{
  Console.WriteLine(i);
  if (i > 50)
   break;
}
```

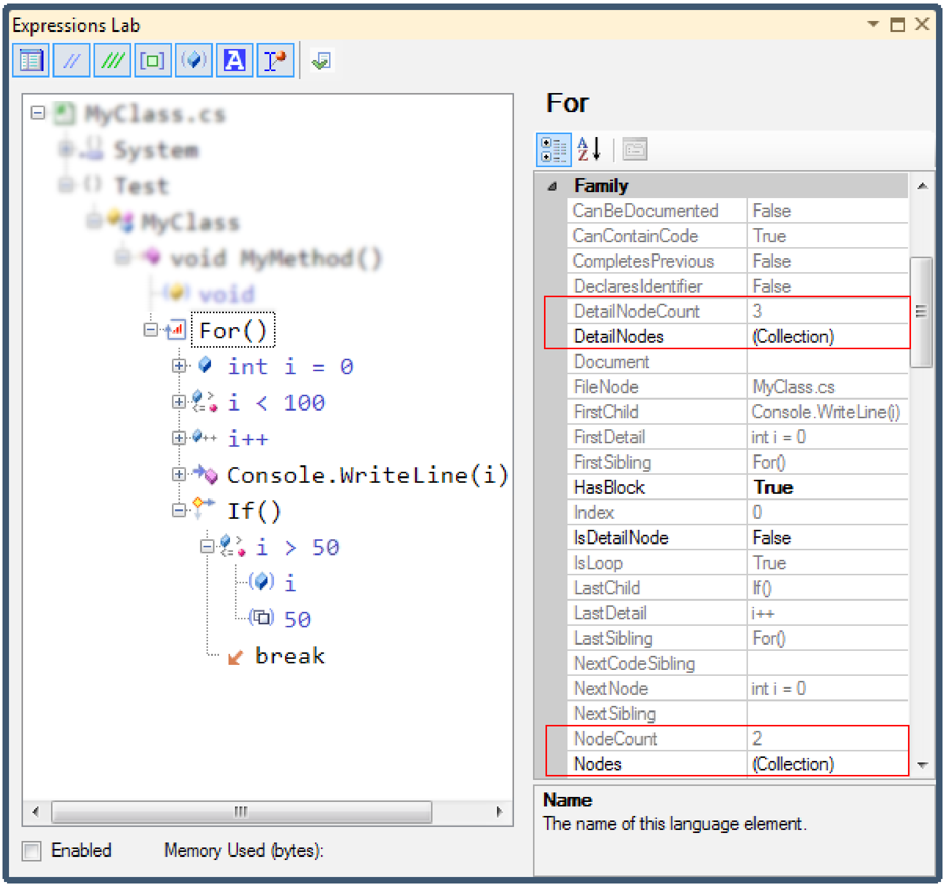
The **Nodes** property will contain two items:

- Language element named “*MethodCall*” – “Console.Write(i);”.
- Language element named “*If*” – “if (i < 50) break;”

The **DetailNodes** property, in turn, will contain three items:

- Language element named “*InitializedVariable*” – “int i = 0;”
- Language element named “*RelationalOperation*” – “i < 100;”
- Language element named “*UnaryIncrement*” – “i++”.

To better understand the difference between **Nodes** and **DetailNodes** properties, use the Expression Lab tool window available via **DevExpress | Tool Windows | Diagnostics | Expression Lab** menu item. Here is what the ‘for’ loop language element looks like inside this tool window:



Expressions Lab

```
MyClass.cs
  System
  Test
    MyClass
      void MyMethod()
        void
        For()
          int i = 0
          i < 100
          i++
          Console.WriteLine(i)
          If()
            i > 50
              i
              50
            break
```

For

Family	
CanBeDocumented	False
CanContainCode	True
CompletesPrevious	False
DeclaresIdentifier	False
DetailNodeCount	3
DetailNodes	(Collection)
Document	
FileNode	MyClass.cs
FirstChild	Console.WriteLine(i)
FirstDetail	int i = 0
FirstSibling	For()
HasBlock	**True**
Index	0
IsDetailNode	False
IsLoop	True
LastChild	If()
LastDetail	i++
LastSibling	For()
NextCodeSibling	
NextNode	int i = 0
NextSibling	
NodeCount	2
Nodes	(Collection)

Name
The name of this language element.

☐ Enabled Memory Used (bytes):

The other way to enumerate elements inside of a scope is to use the *ElementEnumerable* class, declared in the "*DevExpress.DXCore.Parser*" assembly. It may take one, two or three parameters, such as:

- The **Scope** – the *LanguageElement* instance inside of which you are going to enumerate inner elements, for example, a *SourceFile* or a *Method* instance.

- The optional **Type** of the inner elements parameter or a specific **IElementFilter** interface descendant. The "*Type*" parameter can be specified by the *LanguageElementType* enumeration (e.g. *LanguageElementType.Comment*) or a *System.Type* instance. For example:

CSharp code:

```
Class activeClass = CodeRush.Source.ActiveClass;
ElementEnumerable xmlCommentsEnumerable = new ElementEnumerable(activeClass, LanguageElementType.XmlDocComment);
ElementEnumerable throwExceptionsEnumerable = new ElementEnumerable(activeClass, typeof(Throw));
```

You are also able to pass an array of the *LanguageElementType* or *System.Type* instances.

The *IElementFilter* descendant allows you to specify more conditions for the element than just its type which specifies whether a particular element will be returned during an enumeration process. There are two methods on the *IElementFilter* interface to implement: *Apply* and *SkipChildren*.

The *Apply* method returns the value, which specifies whether the element will be returned when enumerating the scope. The *SkipChildren* specifies whether the enumerator should look inside the element that is passed the *Apply* procedure, or move on to the next element skipping its nodes. For example, here's the code of the filter that looks for the field members with the name starting with an underscore "_":

CSharp code:

```
public class FildsWithUnderscoreFilter : IElementFilter
{
  public bool Apply(IElement element)
  {
    return element is IFieldElement && element.Name.StartsWith(“_”);
  }
  public bool SkipChildren(IElement element)
  {
    return true;
  }
}
```

There is also the *DefaultElementFilters* class, defined with a few default element filters, such as *TypeOrNamespace* filter, *NonPrivateMember*, etc.

- The third optional parameter is the "UseRecursion" boolean parameter, which specifies whether the enumeration search should be performed recursively.

Conclusion

Enumeration of the solution and code blocks elements is one of the basic tasks for DXCore plug-in, which allows you to select elements to begin work with. Any consequent task, such as modifying elements, navigating to elements almost always needs to get list of an active or specific elements to start working with. Getting an active element allows you to define the condition whether a particular feature should be available, for example. E.g. the IsNullOrWhiteSpace contract provider needs to check if the active element is a *Parameter* element instance, and if it is not a *Parameter*, then it is not available.

How to parse source code using the DXCore integrated code parsers

There are times when you need to parse specific source files or blocks of code. Obviously, the DXCore Framework has many built-in parsers for various programming languages. They can be used inside the Visual Studio environment, or outside an IDE in any other application type, such as a *Console App*, for example. Later, this kind of app (a *Console App*) can be used in the project building process for code validation, code clean-up, automatic refactoring and any other task.

Let's see how can we parse the code inside a DXCore plug-in when it is loaded into the Visual Studio IDE. Here, we can use the Language and the Source Model **DXCore** services with several APIs specific for the parsing of the code:

- Parsing of the files outside Visual Studio:

CodeRush.Language.Parse("FileNameGoesHere") – parses the specified source file on disk and returns the LanguageElement that specifies it (most likely, a *SourceFile* instance).

- Parsing of the opened text documents (opened source files):

ParseActiveDocument, *Parse(TextDocument)* or *ParseDocument(TextDocument)* (from the Language **DXCore** service) – parses the specified text document (an opened source file) and returns the LanguageElement that specifies it (a *SourceFile* instance).

ParseIfTextChanged (from the SourceModel **DXCore** service) – parses the active or the specified document if the text has been changed (but not necessarily committed) since the last parse. Plug-in authors can call this method to ensure that the **DXCore** structural image is in sync with the file. For example, you might call this immediately after some text edits.

- Specific parsing methods from the Language service:

ParseExpression – parses an expression from the given string.

ParseString – parses the specified string and returns the LanguageElement that specifies it.

ParseTypeReferenceExpression – parses a type reference expression from the given string.

- Other *Parse* method overloads:

These take additional parameters, such as *ParserContext*, *RegionDirective*, *CompilerDirective*, *SourceRange*, *TextStringCollection*, etc. – these methods parse the specified source range of the given text document with the given context, and returns the *LanguageElement* that specifies it. Nodes parsed in the specified range will be appended to the end of the context's nodes. Does not trigger the *BeforeParse* or *AfterParse* events, nor does this method call *BindToCode* – the calling client code must do that (this allows the calling code to bind only the nodes within the parse range, and also append any trailing nodes to the end of the newly-parsed nodes).

For example, let's imagine we're going to release an open source project, but it is required to clean-up the source files from the comments. Here's a sample on how to achieve this for a specific file:

```
string fileName = “FileNameGoesHere”;
SourceFile parsedFile = CodeRush.Language.Parse(fileName) as SourceFile;
if (parsedFile != null)
{
  CleanUpFromComments(parsedFile);
}


// ...


/// <summary>
/// Removes all comments from the given file. Uses DXCore element enumerations methods and the DXCore File service to change the file.
/// </summary>
/// <param name=”parsedFile”>The file to clean-up.</param>
private void CleanUpFromComments(SourceFile parsedFile)
{
  LanguageElementType[] commentTypes = new LanguageElementType[] { LanguageElementType.Comment, LanguageElementType.XmlDocComment };
  ElementEnumerable elementEnumerable = new ElementEnumerable(parsedFile, commentTypes, true);
  FileChangeCollection fileChanges = new FileChangeCollection();
  foreach (LanguageElement comment in elementEnumerable)
    fileChanges.Add(new FileChange(parsedFile.FilePath, comment.Range, String.Empty));
  CodeRush.File.ApplyChanges(fileChanges);
}
```

Don’t forget, that you can also use the **DXCore standalone parser** assembly (*DevExpress.DXCore.Parser.dll*), to parse the source code outside Visual Studio. Here’s a sample above, corrected for using the **DXCore** parsers in any type of application:

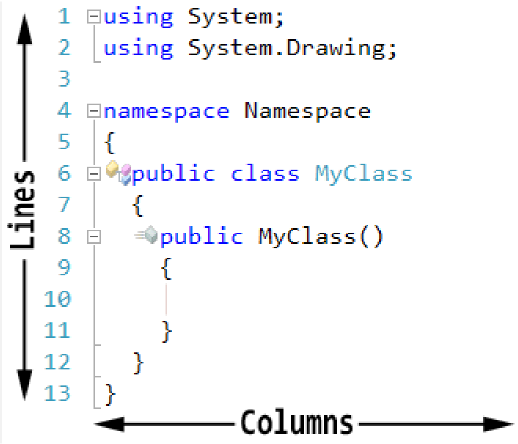
```
string fileName = “FileNameGoesHere”;
string extension = Path.GetExtension(fileName);
ParserBase parser = ParserFactory.CreateParserForFileExtension(extension);
if (parser != null)
{
  SourceFile parsedFile = parser.ParseFile(fileName) as SourceFile;
  if (parsedFile != null)
  {
    CleanUpFromComments(parsedFile);
  }
}
```

However, we can’t use DXCore services outside Visual Studio. So, in the sample above, the File service won’t be available for modification of files. We will use another technique in this case – for example, directly change the text of the file.

There are two **DXCore** plug-ins for the *CSharp* and *Visual Basic* languages, and two *Windows Forms Applications* for the same languages are attached as a sample. The *Windows Forms Application* projects uses a different mechanism for changing files, you are welcome to use any algorithm by your preference.

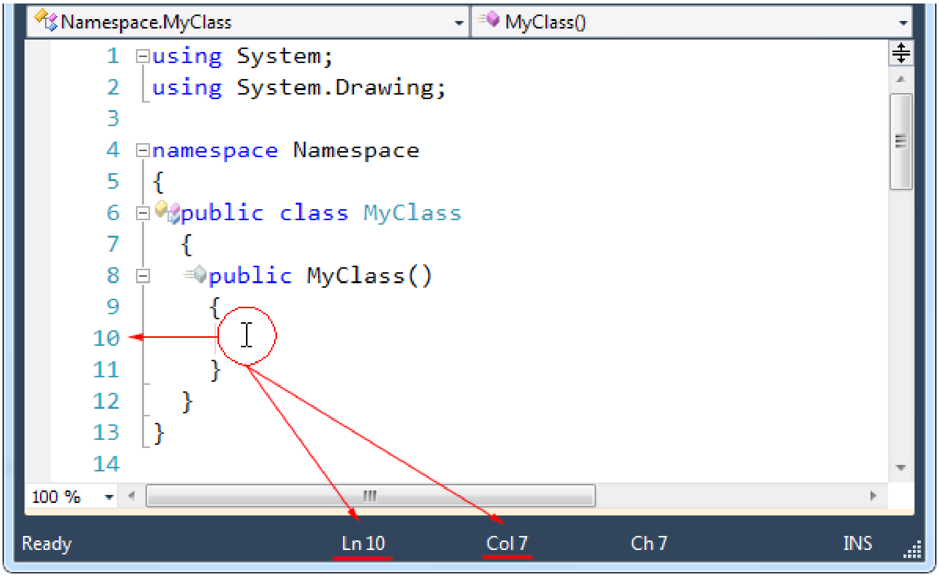
Source code and source tree elements coordinate inside source files

Inside Visual Studio IDE, the source code text has its coordinates: the line number and the column:



```
 1  using System;
 2  using System.Drawing;
 3
 4  namespace Namespace
 5  {
 6    public class MyClass
 7    {
 8      public MyClass()
 9      {
10
11      }
12    }
13  }
```

These coordinates are 1-based, and the first line has an index equal to 1. These values can be seen in the right bottom corner of the IDE:



```
 1  using System;
 2  using System.Drawing;
 3
 4  namespace Namespace
 5  {
 6      public class MyClass
 7      {
 8          public MyClass()
 9          {
10
11          }
12      }
13  }
14
```

- “Ln” is the line number
- “Col” is the column number (offset position)
- “Ch” is the character position (for example, a single ‘Tab’ character can take several columns)

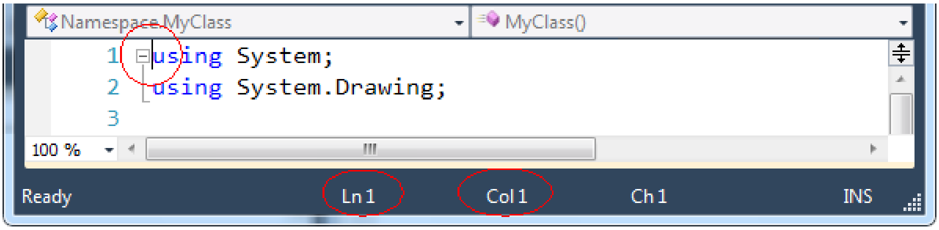
After DXCore automatically parses a source file, the abstract source tree is built for the entire file. Language elements have the same coordinates that correspond to the text coordinates inside of the source file. The coordinates of the language element are accessible through its properties, e.g.:

- *StartLine* - the starting line of a language element
- *StartOffset* – the starting offset (column) of a language element
- *EndLine* – the end line of a language element
- *EndOffset* – the end offset (column) of a language element

For example, this line of CSharp code:

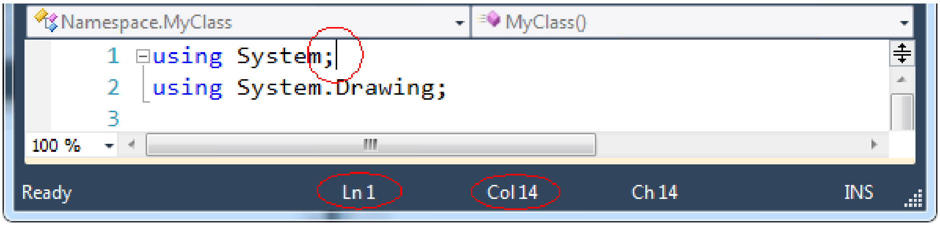
1	`using System;`

Has the following coordinates in Visual Studio:



```
1  using System;
2  using System.Drawing;
3
```

and



```
using System;
using System.Drawing;
```

The language element can be located on several lines, e.g. a *Class* element most likely will have different starting and ending lines. These four numbers are encapsulated into a single property of the *LanguageElement* called *Range*, which has a type of the *SourceRange* structure. The *SourceRange* may consist of four coordinates: starting line, starting offset, ending line, ending offset, or may consist of two others of the *SourcePoint* type, which represent the starting point (١,١) and the end point (١,١٤). So, the *SourcePoint* contains a single pair of a line/offset values:

```
SourceRange(int startLine, int startOffset, int endLine, int endOffset)
```

is the equivalent of:

```
SourceRange(new SourcePoint (startLine, startOffset), new SourcePoint(end-
Line, endOffset))
```

The range and the stating/ending point properties are useful to manipulate the text of the source file. You can replace the piece of code using the *SourceRange* value of the element, or insert some new code before or after an element using the *Start* or *End* SourcePoint values available thought the *Range* property:

```
SourceRange elementRange = LanguageElement.Range;
SourcePoint elementStartPoint = LanguageElement.Range.Start;
SourcePoint elementEndPoint = LanguageElement.Range.End;
```

For example, let's use these properties to remove, and then insert some text:

```
;TextDocument activeTextDocument = CodeRush.Documents.ActiveTextDocument
(if (activeTextDocument != null
}
;(activeTextDocument.DeleteText(elementRange
activeTextDocument.InsertText(elementStartPoint, «/* LanguageElement
;(«/* .was here
{
```

Now, let's observe useful APIs dealing with the source ranges and source point properties. On the *LanguageElement* instance there are the following properties and methods:

Name	Description
Range	The source range of this language element.

NameRange	The source range of the name of this language element.
GetCutRange()	Gets a SourceRange that includes leading and trailing white spaces. This method is useful for cutting, deleting or moving this LanguageElement, as this expanded SourceRange tends to produce a clean break in the code when removed.
GetFullBlockCoordinates()	Gets the full block coordinates for this language element. "Full block" is defined as this element and any partnering elements that complete this block (or that this block completes), including attributes, XML Doc Comments, and regions that tightly enclose this block. The coordinates may extend beyond the bounds of this particular language element.
GetFullBlockCutRange()	Gets a SourceRange that includes partnering elements, leading and trailing white spaces. Partnering elements are neighboring elements that complete this block (or that this element completes), and include attributes, XML Doc Comments, and regions that tightly enclose this block. This method is useful for cutting, deleting or moving this LanguageElement, as this expanded SourceRange tends to produce a clean break in the code when removed.
GetFullBlockRange()	Gets the full block range for this language element. "Full block" is defined as this element and any partnering elements that complete this block (or that this block completes), including attributes, XML Doc Comments, and regions that tightly enclose this block. The coordinates may extend beyond the bounds of this particular language element.

The three similar methods: *GetFullBlockRange*, *GetFullBlockCutRange* and *GetFullBlockCoordinates* can take a parameter of type *BlockElements*, which specify what elements should be included into the resulting range. The *BlockElement* is a flagged enumeration, containing these elements:

```
[Flags]
public enum BlockElements
{
  All,
  AllLeadingWhiteSpaces,
  AllSupportElements,
  AllTrailingWhiteSpaces,
  AllWhiteSpaces,
  Attributes,
  LeadingEmptyLines,
  LeadingWhiteSpace,
  None,
  Region,
  SupportComments,
  TrailingEmptyLines,
  TrailingWhiteSpace,
  WithoutUnsuitableRegions,
  XmlDocComments
}
```

Here's a brief overview of this enumeration:

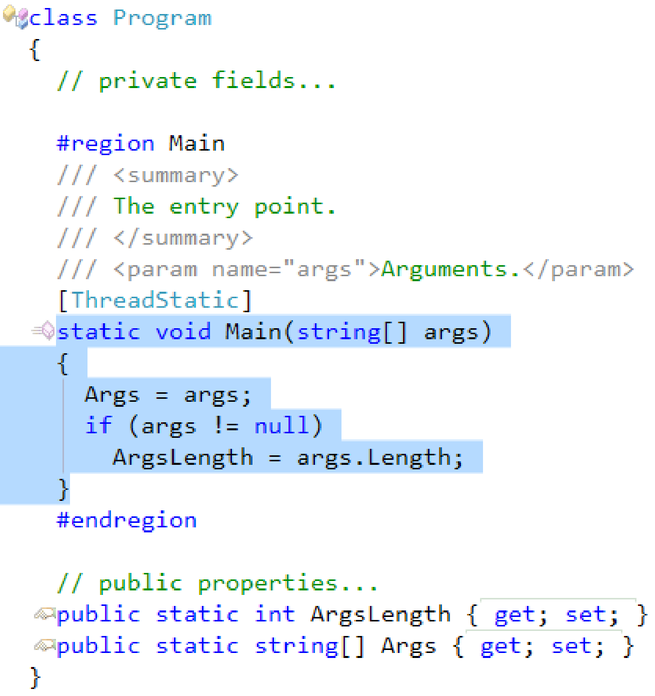
Name	Description
All	.Inlude everything: regions, comments, white space, etc
AllLeadingWhiteSpaces	Leading white space and leading empty lines
AllSupportElements	Support elements are: attributes, comments, xml doc comments
AllTrailingWhiteSpaces	Trailing white space and trailing empty lines
AllWhiteSpaces	Leading and trailing white spaces
Attributes	(Attributes attached to an element (e.g. to a method or property
LeadingEmptyLines	Empty leading lines
LeadingWhiteSpace	Leading white space
None	Nothing except the language element itself
Region	Region directives, e.g. #region … #end region
SupportComments	Comments attached to a language element
TrailingEmptyLines	Empty trailing lines
TrailingWhiteSpace	Trailing white space
WithoutUnsuitableRegions	Unsuitable regions are regions with a caption that is not equal to the element's name
XmlDocComments	Xml doc comments which describe a member or type declaration

Here are a few sample results of the above APIs for the following code:

```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name=»args»>Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

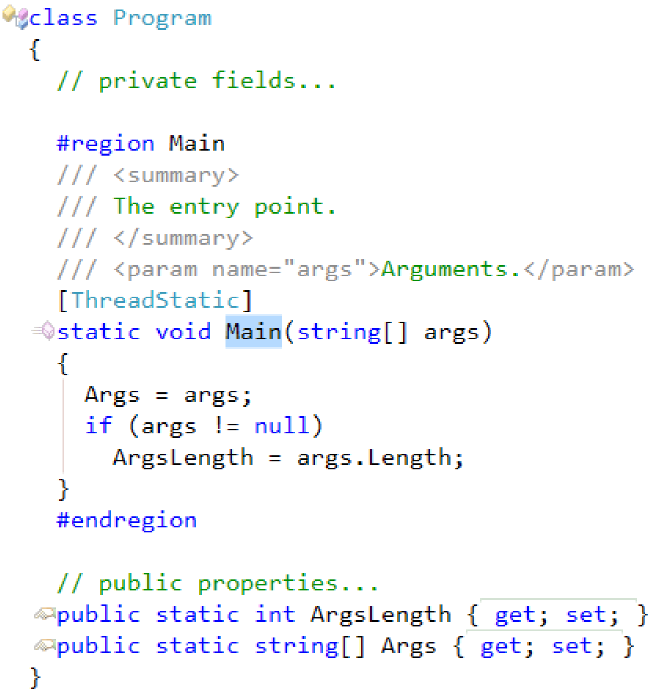
These are the results for the **Main** function:

- **Range**:



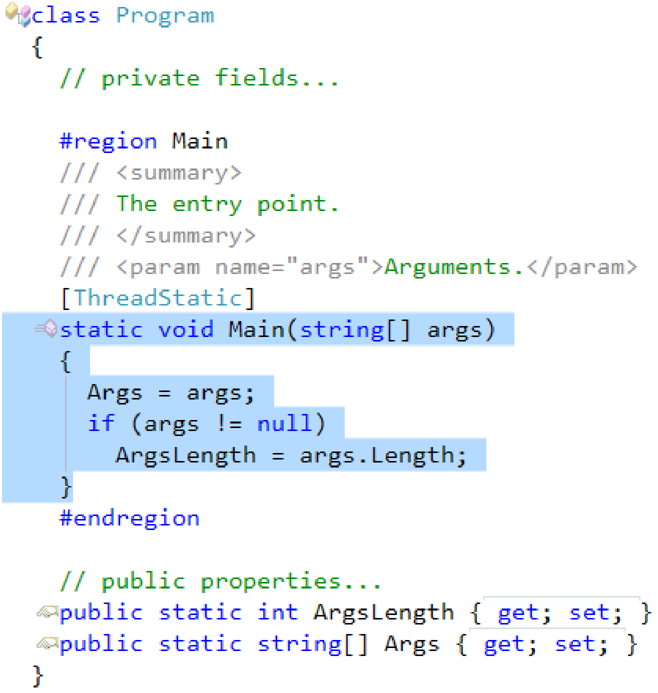
```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

- **NameRange**:



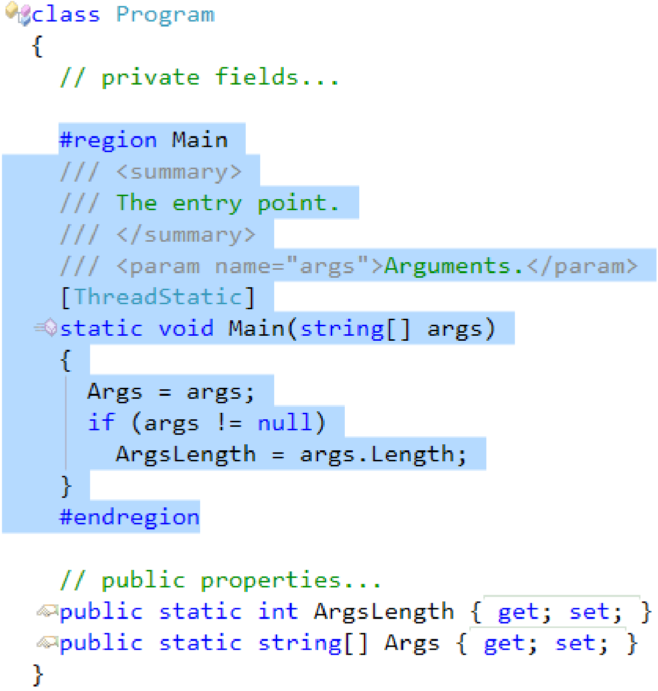
```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

- **GetCutRange():**



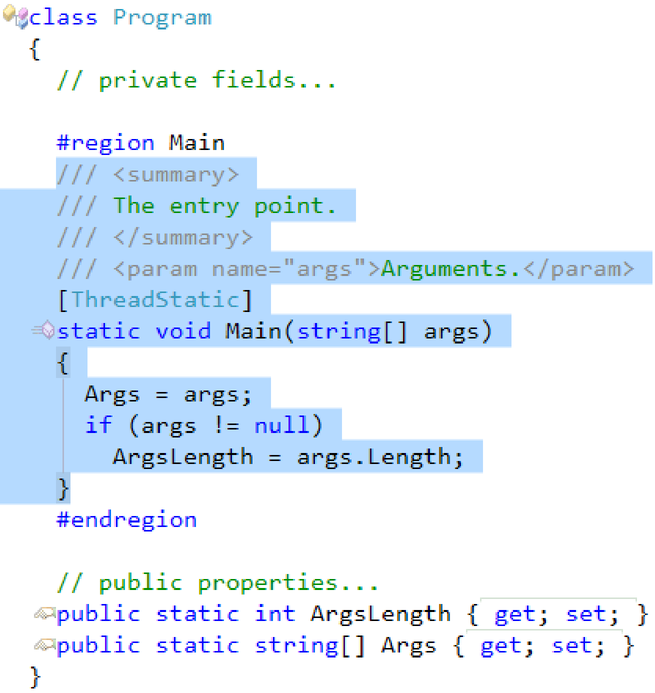
```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

- **GetFullBlockRange():**



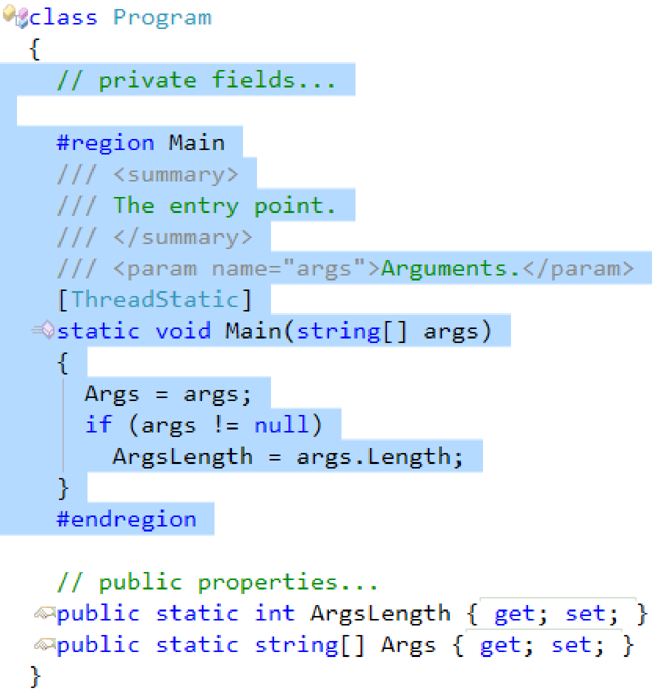
```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

- **GetFullBlockRange**(**BlockElements**.Attributes | **BlockElements**.XmlDocComments):



```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

- **GetFullBlockCutRange**():



```
class Program
{
  // private fields...

  #region Main
  /// <summary>
  /// The entry point.
  /// </summary>
  /// <param name="args">Arguments.</param>
  [ThreadStatic]
  static void Main(string[] args)
  {
    Args = args;
    if (args != null)
      ArgsLength = args.Length;
  }
  #endregion

  // public properties...
  public static int ArgsLength { get; set; }
  public static string[] Args { get; set; }
}
```

Your feedback is much appreciated. Let me know if I can improve this article for better understanding of the source code coordinates.

How to edit source files of an entire Visual Studio solution

Usually, simple edits of text files are accomplished using the *TextDocument* object, which represents an open source file inside the IDE. The text document object is easy to access through the **Documents** DXCore service. It has lots of useful methods for editing a text like *InsertText*, *DeleteText* and *SetText*, which take a source code coordinates and a new text for replacement as a parameters. However, to use a text document object, it is required for the file to be opened inside the Visual Studio environment. If a file is closed, there's no *TextDocument* object assigned to the file and you simply can't use its methods. In case you are going to edit closed and/or multiple files, there's a better way – the **FileChange** object (in the *DevExpress.CodeRush.Core.Replacement* namespace).

As the name says, the **FileChange** object represents a single change to a file. A file can be closed inside IDE and not even included into a project, so you can change any text file in your system. The object takes a few parameters:

- a path to a file
- a source range or a source point
- a text for insertion or replacement

If you pass a source point – the specified text will be inserted at the specified position. But if you pass a source range – the text inside a file will be replaced with the new one. The *FileChange* class has the corresponding properties:

Property	Description
Data	Data, associated with a change. Used internally.
Order	The order of this change in a particular file. The value is set automatically. However, if several changes are made at the same position, you can set this property to manage the order the changes are applied.
Path	A full path to a file.
Range	A source range of the text inside a file for replacement. If a source range has two equal source points, the text will be inserted without deletion.
Text	A new text to insert or replace inside a file.

The file change object should be added into a special collection to be applied – the **FileChangeCollection**. Then, you can apply a collection of file changes by using the File DXCore service:

CodeRush.File.ApplyChanges(collection);

Note, if you have only a single file change, you can call the *CodeRush.File.ChangeFile* method instead of creating a new *FileChange* object. The *ChangeFile* method inserts, deletes or replaces the text inside the specified source file.

Usage sample

As an example, illustrating how to edit source files, let's create a new DXCore plug-in, and add the Action DXCore control, that will modify all files of the current project by inserting a file header at the top (as a comment), if it doesn't exist.

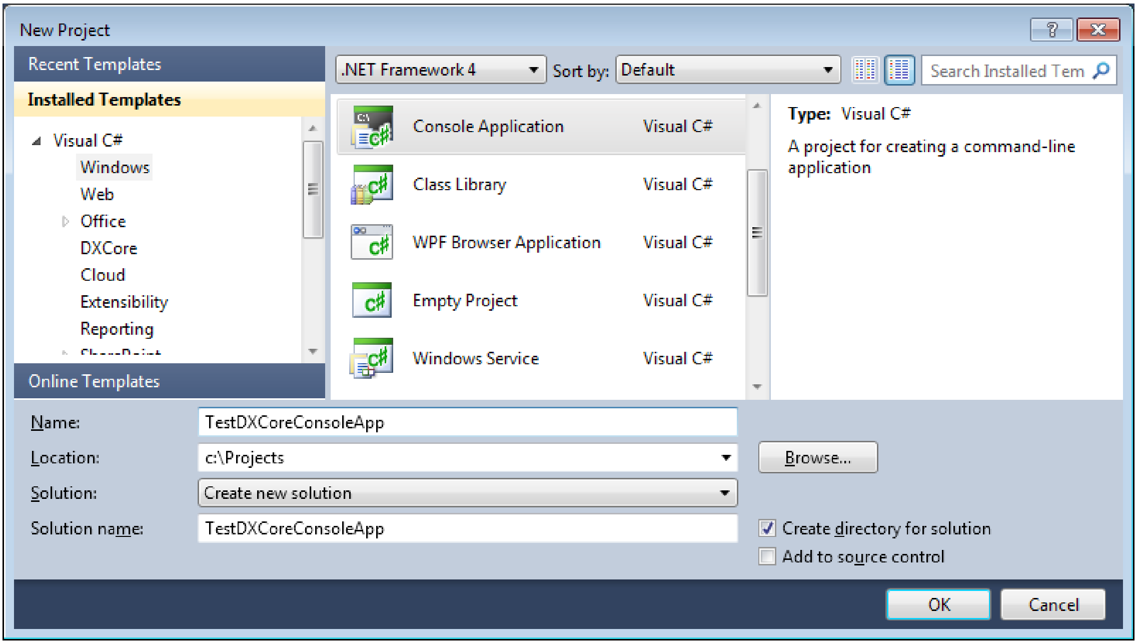
When the action is executed, we get the current project instance and enumerate all of its files. We check the name of every file, so it has a valid extension and it is not a designer file, nor should it already contain a header. If the file is OK, then we create a new *FileChange* object, specify the path to a file, a new insertion point (1,1), a file header text) and add it to a collection of file changes. When all files are enumerated, we apply the collection, so all files of the project now have a file header. Here’s the code of the plug-in:

```
public partial class PlugIn1 : StandardPlugIn
{
  private const string HEADER =
@”//---------------------------------------------------------------------------
// <copyright file=””{0}”” company=””My Company””>
//     Copyright (c) My Company, Inc. All rights reserved.
// </copyright>
//-------------------------------------------------------------------------
“;

  private void action1_Execute(ExecuteEventArgs ea)
  {
    ProjectElement activeProject = CodeRush.Source.ActiveProject;
    if (activeProject == null)
      return;

    FileChangeCollection fileChangeCollection = new FileChangeCollection();
    foreach (SourceFile sourceFile in activeProject.AllFiles)
    {
      if (!IsValidFile(sourceFile))
        continue;

      if (ContainsFileHeader(sourceFile))
        continue;

      string filePathAndName = sourceFile.Name;
      SourcePoint insertionPoint = new SourcePoint(1, 1);
      string newHeader = String.Format(HEADER, Path.GetFileName(filePathAnd-
Name));
      FileChange fileChange = new FileChange(filePathAndName, insertionPoint, ne-
wHeader);
      fileChangeCollection.Add(fileChange);
    }
    CodeRush.File.ApplyChanges(fileChangeCollection);
  }

  private bool IsValidFile(SourceFile sourceFile)
  {
    const string EXT = “.CS”;
    const string DESIGNER = “.DESIGNER”;

    string fileNameWithoutExtension = Path.GetFileNameWithoutExtension(source-
File.Name);
    if (Path.GetExtension(fileNameWithoutExtension).ToUpper() == DESIGNER)
      return false;

    string fileExtension = Path.GetExtension(sourceFile.Name);
    return fileExtension.ToUpper() == EXT;
  }

  private bool ContainsFileHeader(SourceFile sourceFile)
  {
    LanguageElement firstChild = sourceFile.FirstChild;
    return firstChild is Comment;
  }

}
```

How to use DXCore in a Console App outside of Visual Studio

Actually, DXCore is not designed to be used outside of Visual Studio, but there are always workarounds… In this article I'm going to show you how to use the **DXCore Framework** inside the regular *C# Console Application* to parse an entire solution and work with the abstract parsed tree. The solution should be passed-in as an argument to the program as a full complete path to the *.sln file. If there's no argument used, the hard-coded path to the test program is used, so the program will parse itself and print information about the solution, such as a list of all types used and the number of members inside of each class.

Let's create a new C# Console Application, call it *TestDXCoreConsoleApp* and save it inside the "*C:\Project*" folder:



Name: TestDXCoreConsoleApp

Location: c:\Projects

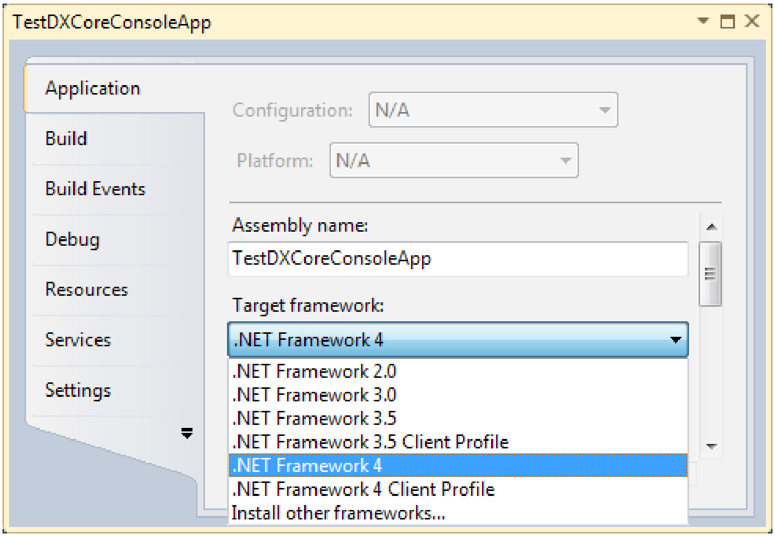
Solution: Create new solution

Solution name: TestDXCoreConsoleApp

☑ Create directory for solution

□ Add to source control

Then, we should change the Target Framework version of the new project to Framework 4.0, so it's not a "Target Framework 4.0 Client Profile", because some required assembly references don't support this version of the Target Framework:



Configuration: N/A

Platform: N/A

Assembly name: TestDXCoreConsoleApp

Target framework: .NET Framework 4

Now, let add required assembly references. Here's the list of what we need:

1) **DXCore** assemblies:

- DevExpress.CodeRush.Common
- DevExpress.CodeRush.Core
- DevExpress.CodeRush.StructuralParser
- DevExpress.CodeRush.VSCore
- DevExpress.DXCore.AssemblyResolver
- DevExpress.DXCore.Parser

These assemblies canbe found inside your DevExpress IDE Tools installation folder. For example, the path may look like this:

C:\Program Files\DevExpress 2011.1\IDETools\System\DXCore\BIN

2) Now, three additional assemblies for different program language support:

- DX_CPPLanguage
- DX_CSharpLanguage
- DX_VBLanguage

With these assemblies we are able to parse CSharp, Visual Basic and C++ projects. They can be found here:

C:\Program Files (x86)\DevExpress 2011.1\IDETools\System\DXCore\BIN\SYSTEM

3) .NET Framework assemblies:

- Microsoft.Build.BuildEngine.dll

4) And, finally, a couple of Visual Studio assemblies:

- EnvDTE

- VsLangProj

These two can be found in the “*PublicAssemblies*” folder:

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\PublicAssemblies\

Now, the **DXCore** support code. This code is required to load a solution, its projects and initialize **DXCore** parsers. I’ve added two folders:

1) The *Helpers* folder contains the following classes:

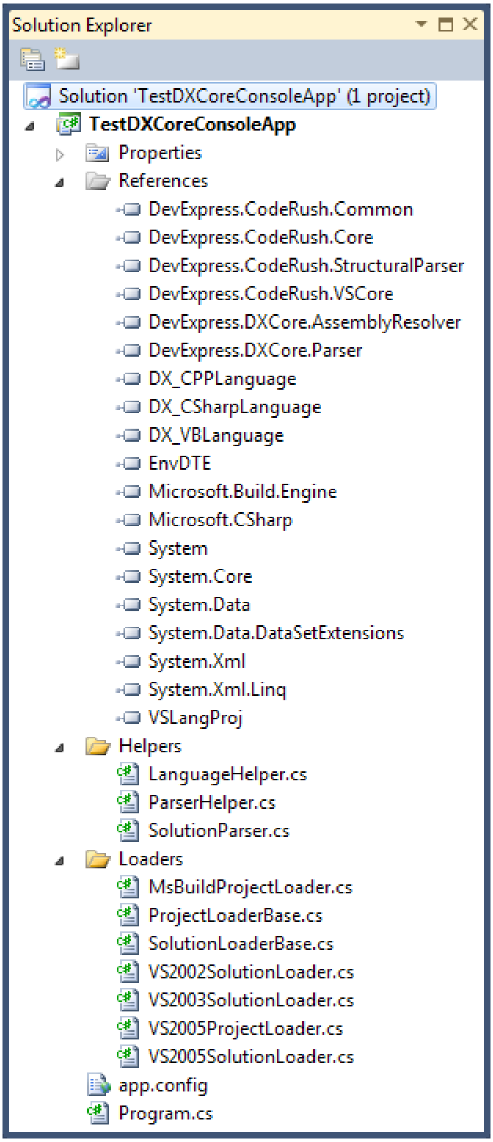
- *LanguageHelper.cs* – detects the language of projects (e.g. CSharp, Visual Basic or C++).

- *ParserHelper.cs* – initializes **DXCore** parsers, and a few important **DXCore** services – the Source Model service and the Language service which are used to parse source code.

- SolutionParser.cs – a helper class, which takes the path to the solution that you are going to parse. Calling the *GetParsedSolution* method will return the *SolutionElement*, which holds the abstract source tree of the entire solution.

2) The *Loaders* folder contains the Visual Studio project and solution loaders for different Visual Studio versions. They are used to parse *.XXproj and *.sln files. There are versions for VS2002, VS2003 and VS2005. There are no dedicated loaders for VS2008 and VS2010, because those loaders for the old VS versions are perfectly fine to reading and loading newer Visual Studio project and solution format files (e.g. 2008, 2010).

Here’s the final structure of the *TestDXCoreConsoleApp*:

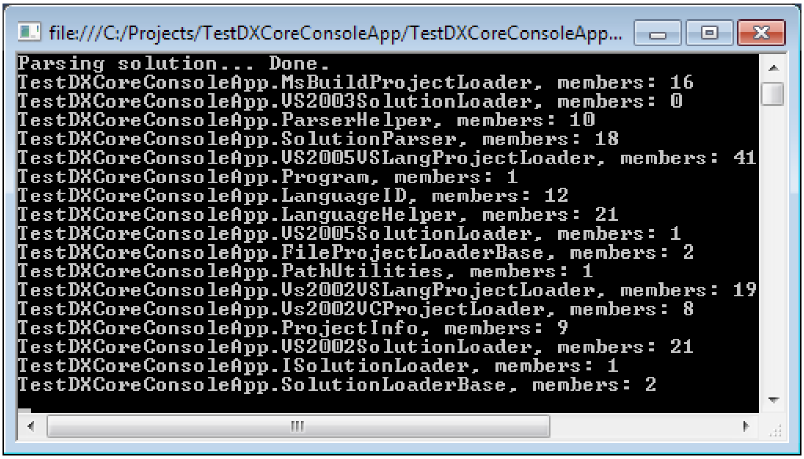


```
Solution Explorer
Solution 'TestDXCoreConsoleApp' (1 project)
  TestDXCoreConsoleApp
    Properties
    References
      DevExpress.CodeRush.Common
      DevExpress.CodeRush.Core
      DevExpress.CodeRush.StructuralParser
      DevExpress.CodeRush.VSCore
      DevExpress.DXCore.AssemblyResolver
      DevExpress.DXCore.Parser
      DX_CPPLanguage
      DX_CSharpLanguage
      DX_VBLanguage
      EnvDTE
      Microsoft.Build.Engine
      Microsoft.CSharp
      System
      System.Core
      System.Data
      System.Data.DataSetExtensions
      System.Xml
      System.Xml.Linq
      VSLangProj
    Helpers
      LanguageHelper.cs
      ParserHelper.cs
      SolutionParser.cs
    Loaders
      MsBuildProjectLoader.cs
      ProjectLoaderBase.cs
      SolutionLoaderBase.cs
      VS2002SolutionLoader.cs
      VS2003SolutionLoader.cs
      VS2005ProjectLoader.cs
      VS2005SolutionLoader.cs
    app.config
    Program.cs
```

The *TestDXCoreConsoleApp* with the full source is attached (267,457 bytes, C#, VS2010), so you may review the code and use it as you'd like. Here's the *Main* function of the Program class:

```
static void Main(string[] args)
{
  string SolutionPath;
  if (args != null && args.Length > 0)
    SolutionPath = args[0];
  else
    SolutionPath = @»c:\Projects\TestDXCoreConsoleApp\TestDXCoreConsoleApp.
sln»;
  try
  {
    ParserHelper.RegisterParserServices();

    Console.Write(«Parsing solution... «);

    SolutionParser solutionParser = new SolutionParser(SolutionPath);
    SolutionElement solution = solutionParser.GetParsedSolution();
    if (solution == null)
      return;

    Console.WriteLine(«Done.»);
    foreach (ProjectElement project in solution.AllProjects)
      foreach (SourceFile file in project.AllFiles)
        foreach (TypeDeclaration type in file.AllTypes)
        {
          Console.Write(type.FullName);
Count);   Console.WriteLine(«, members: « + ((ITypeElement)type).Members.
        }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
    ParserHelper.UnRegisterParserServices();
  }
  Console.ReadLine();

}
```

If you put the sources into the “*C:\Projects*” folder and run the program without any arguments specified, you should see the following result:


file:///C:/Projects/TestDXCoreConsoleApp/TestDXCoreConsoleApp...

```
Parsing solution... Done.
TestDXCoreConsoleApp.MsBuildProjectLoader, members: 16
TestDXCoreConsoleApp.VS2003SolutionLoader, members: 0
TestDXCoreConsoleApp.ParserHelper, members: 10
TestDXCoreConsoleApp.SolutionParser, members: 18
TestDXCoreConsoleApp.VS2005VSLangProjectLoader, members: 41
TestDXCoreConsoleApp.Program, members: 1
TestDXCoreConsoleApp.LanguageID, members: 12
TestDXCoreConsoleApp.LanguageHelper, members: 21
TestDXCoreConsoleApp.VS2005SolutionLoader, members: 1
TestDXCoreConsoleApp.FileProjectLoaderBase, members: 2
TestDXCoreConsoleApp.PathUtilities, members: 1
TestDXCoreConsoleApp.Vs2002VSLangProjectLoader, members: 19
TestDXCoreConsoleApp.Vs2002VCProjectLoader, members: 8
TestDXCoreConsoleApp.ProjectInfo, members: 9
TestDXCoreConsoleApp.VS2002SolutionLoader, members: 21
TestDXCoreConsoleApp.ISolutionLoader, members: 1
TestDXCoreConsoleApp.SolutionLoaderBase, members: 2
```

Press the *Enter* key to close the window. Bear in mind, that the parsing process may take some time, so you might need to wait a few seconds, until the entire solution is parsed. Let me know if you have any difficulties with compiling the sources.

DevExpress DXCore controls overview

DXCore contains uniquely-named duplicates (to avoid naming collision and other design-time issues) of most of DevExpress controls. These controls are located in the "***DevExpress.DXCore.Controls.****" assemblies. They can be used when writing DXCore plug-ins only. It is impossible to use these controls in an arbitrary application. You need the DXCore design-time assemblies to be able to use these components at design time. However, these assemblies are only available for customers who have a DXperience Subscription (customers who have already paid for the design-time portion for the corresponding controls in **DXCore**).

Also, there are several other **DXCore**-specific components added into your Visual Studio toolbox, which you can use inside your **DXCore** based plug-ins.

DXCore components and controls list

DXCore has several components added into your Visual Studio toolbox, which you can use inside your DXCore based plug-ins. These components are divided into the following categories:

- Commands
- Events
- Extensions
- Providers
- User Assistance/Interaction
- Visual Controls

Here they are in detail (the icon (a green tick or a red cross) on the last column in all tables specifies whether this control appears on the Visual Studio toolbox):

- **Commands**

Name	Description	
Action	Registers an action with **DXCore**/**CodeRush** and the Visual Studio IDE. Actions can be bound to keystrokes.	✓
Text Command	**TextCommands** move the editor caret, insert, remove, or change code, and bring up a UI for powerful, intelligent and potentially interactive code creation.	✓

- **Events**

Name	Description	
DXCore Events	Provides access to a host of Visual Studio and**DXCore**/**CodeRush** events. StandardPlugIn descends from this class and inherits its events.	✗

- **Extensions**

Name	Description	
Duplicate Line	Participates in the CodeRush Duplicate Line feature, providing intelligent custom suggestions based on the context.	✓
Intellassist	Participates in the CodeRush Intellassist feature, providing intelligent custom suggestions based on the context.	✓
Language	Registers support for a programming language, implementing a parser that generates a source tree of **DXCore** language elements.	✓
Selection Inversion	Adds custom intelligent selection inversions that can be applied to the selected code.	✓
Intelligent Paste	Adds custom Intelligent Paste expansion based on the context.	✓

- **Providers**

Name	Description	
Code	Provides a particular code operation which can change your code in any way, depending on the context.	✓
Code Metric	Calculates and returns a code metric used in the **Metrics** tool window of **Refactor! Pro**.	✓
Context	Provides a context entry which users can bind to shortcuts, code templates and other CodeRush solutions. Context determines if a feature is appropriate for use.	✓
Contract	Provides a code contract used in the **Add Contract** provider, which checks conditions for method parameters.	✗
Dynamic List	Returns a collection of **Dynamic Lists** used in the **CodeRush Templates Engine**.	✗
Issue	Provides a particular code issue check, which verifies the source code.	✓
Navigation	Provides a particular navigation action, which performs navigation inside of the source code.	✗
Refactoring	Provides a particular code refactoring operation, which can improve your source code without changing its logic in the definite context.	✓
Search	Returns a custom searcher used in the **Rename** refactoring or **Tab to Next Reference** navigation feature.	✓
Smart Tag	Provides a custom smart tag item used in the IDE context menu or in the Refactor! popup menu.	✗
String	Returns a string based on state, context, and/or parameters passed to this provider. String providers can be used in templates, text expansion, intelligent paste, and a number of other **CodeRush**features.	✓
Tutorial Content	Returns a section of HTML for displaying in the DXCore User Guide.	✓
Tutorial Page	Returns an entire HTML page, and registers it, so it appears in the topic tree of the DXCore User Guide.	✓
Unit Test	Provides support for custom testing engines used in the CodeRush Unit Test Runner.	✗

- **User Assistance/Interaction**

Name	Description

Action Hint	Points to a location on screen with a large colored arrow, displaying a short text message. Action hints inform new users about IDE tools features, as they occur.	✓
Animated Arrow	A descendant of the **AnimationFrame**, this control will animate an arrow that moves from one location in the source code to another. This animation is useful for illustrating relationships between bits of code.	✓
Animation Frame	Animates on the code editor, the form designer, or just about any part of Visual Studio.	✓
Big Feedback	Shows a custom feedback message (also known as **Billboard Message**) on the entire code editor surface.	✓
Big Hint	Shows a customizable hint with a title and specific information.	✗
Code View	Shows a syntax highlighted code preview window, useful for showing result code of any code changing operation.	✓
Reorder Selector	An interactive selector specified for the reordering of elements in the code editor.	✗
Locator Beacon	A descendant of the **AnimationFrame**, this control will animate a locator beacon anywhere you need one in Visual Studio.	✓
Shortcuts Hint	Shows a help hint with a table of available shortcut keys in the current context for the particular interactive feature.	✓
SourceRange Highlight	A descendant of the **AnimationFrame**, this control will highlight a section of code.	✗
Target Picker	A code location picker used for specifying the target location.	✗
Text Change Selector	An interactive dialog used for several code changing operations. Allows you to apply a single change, apply all changes, skip, or suppress and then resume code changing operations.	✗

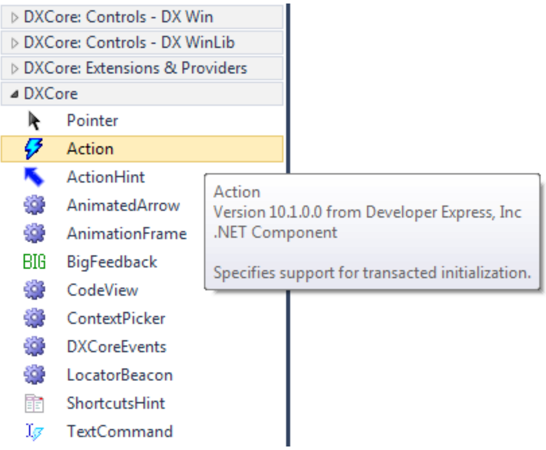
- **Visual Controls**

Name	Description	
Color Swatch	A color picker used for choosing a color.	✗
Context Picker	Allows users to specify sophisticated context expressions.	✓
LedLight	A small icon used for indicating a particular state using different colors.	✗
LedLightLabel	A small icon with a text used for indicating a particular state using different colors.	✗
Source Tree	Shows a full source tree of **DXCore** language elements of a specific parsed source file.	✗

DXCore Components – Action

The **Action** component is one of the primary, simplest and useful components of the DXCore. It registers an “action” within **DXCore**/CodeRush and the Visual Studio IDE, which can be triggered using either a keyboard shortcut or a mouse button (with the shift key if necessary), and/or placed within a Visual Studio menu (e.g. in the IDE main menu, Code Editor context menu, etc).

You can drop it onto your plug-in design surface from the “**DXCore**” category of Visual Studio Toolbox:

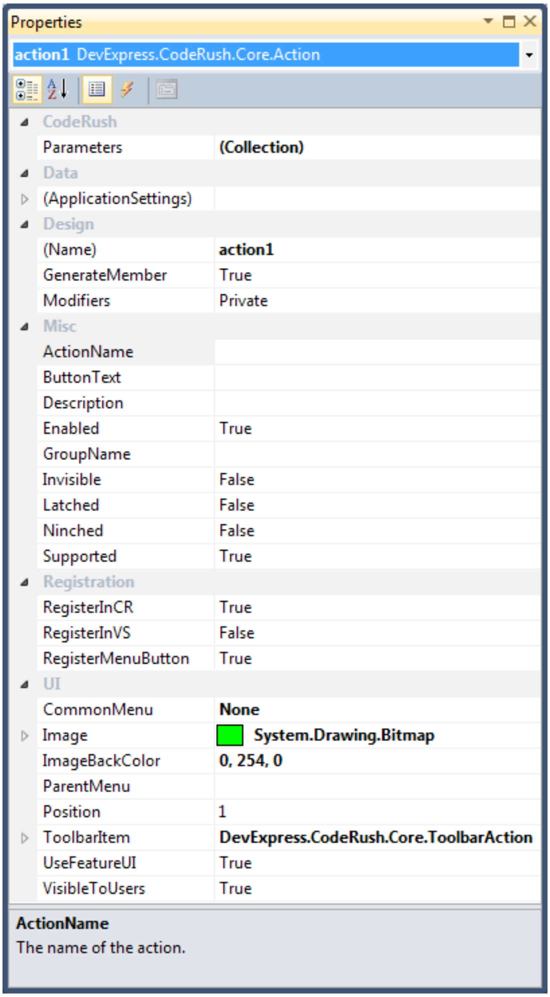


DXCore: Controls - DX Win
DXCore: Controls - DX WinLib
DXCore: Extensions & Providers
DXCore
- Pointer
- Action
- ActionHint
- AnimatedArrow
- AnimationFrame
- BigFeedback
- CodeView
- ContextPicker
- DXCoreEvents
- LocatorBeacon
- ShortcutsHint
- TextCommand

Action
Version 10.1.0.0 from Developer Express, Inc
.NET Component

Specifies support for transacted initialization.

It’s not a visual control, all you need is to tweak its properties and subscribe to its events. Here’s the **Properties** window, containing available properties, their categories and default values:



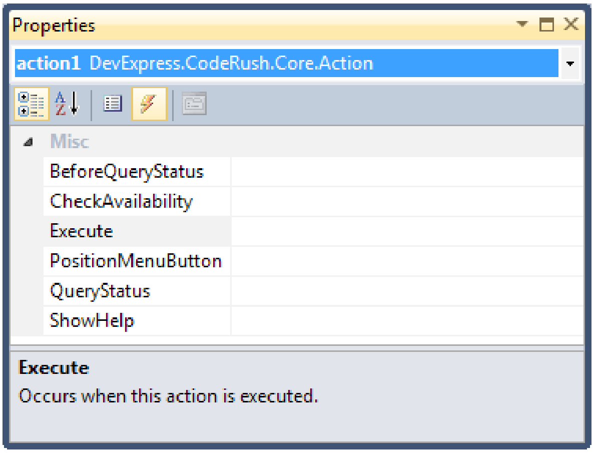
Properties

action1 DevExpress.CodeRush.Core.Action

CodeRush	
Parameters	(Collection)
Data	
(ApplicationSettings)	
Design	
(Name)	action1
GenerateMember	True
Modifiers	Private
Misc	
ActionName	
ButtonText	
Description	
Enabled	True
GroupName	
Invisible	False
Latched	False
Ninched	False
Supported	True
Registration	
RegisterInCR	True
RegisterInVS	False
RegisterMenuButton	True
UI	
CommonMenu	None
Image	🟩 System.Drawing.Bitmap
ImageBackColor	0, 254, 0
ParentMenu	
Position	1
ToolbarItem	DevExpress.CodeRush.Core.ToolbarAction
UseFeatureUI	True
VisibleToUsers	True

ActionName
The name of the action.

and the Events list:



Properties

action1 DevExpress.CodeRush.Core.Action

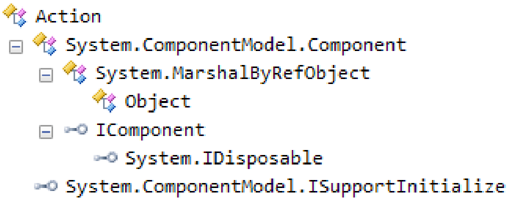
Misc

- BeforeQueryStatus
- CheckAvailability
- Execute
- PositionMenuButton
- QueryStatus
- ShowHelp

Execute

Occurs when this action is executed.

The **Action** component is located in the “***DevExpress.CodeRush.Core***” assembly. It is derived from the “***System.ComponentModel.Component***” class:



```
Action
⊟ System.ComponentModel.Component
  ⊟ System.MarshalByRefObject
      Object
  ⊟ IComponent
      System.IDisposable
  System.ComponentModel.ISupportInitialize
```

Here’s a list of properties in alphabetical order, without base class (**Component**) standard properties:

Property name	Description
ActionName	The name of the action displayed in the IDE Tools UI.
ButtonText	Text for buttons and menu items associated with this action.
CommonMenu	Visual Studio common menu enumeration, e.g. “File”, “Edit”, “View”, etc.
Description	A hint associated with this action, describing this action.
Enabled	Sets the enabled state for this action in the current context. You can change this property in a *BeforeQueryStatus* event handler.
GroupName	The name of the action group.
Image	Sets the bitmap image for this action. Action images are optional, but can be useful for menu items and toolbar buttons inside the IDE.
ImageBackColor	Sets the image background color for this action. It is used to determine which color is to be treated as transparent in the bitmap assigned to the Image property.
Invisible	Sets the visibility for this action.
Latched	Sets the latched (e.g., selected or checked) state for this action in the current context. You can change this property in a BeforeQueryStatus event handler.
Ninched	Sets the indeterminate state (neither checked nor cleared) for this action in the current context. You can change this property in a BeforeQueryStatus event handler.
Parameters	A collection of expected parameters for this action. Optional.

ParentMenu	Sets the name of the parenting command bar. If you want to create a menu item for this action, specify the name of the parenting command bar here (e.g., "Tools"). *ParentMenu* is an alternative to CommonMenu. Use the name of any menu you know of, to place the action on it, rather than one of "common" ones.
Position	Sets the position for this action in the specified ParentMenu. The first position is one. If you specify zero for this value, the item will be appended to the end of the menu. If you specify a negative number, the menu item position will be calculated from the end of the menu (e.g., if Position equals -1, then this menu item will be inserted before the last entry).
RegisterInCR	Registers this action with **DXCore**. If false, this action may still be available in the Visual Studio Options dialog on the Keyboard page. If you want to place this action in a menu or on a command bar, then you must set this property to true. If true, you can bind a keystroke to this command in the IDE Tools Options dialog on theShortcuts options page.
RegisterInVS	Registers this action as a Visual Studio command. If false, you can still bind a keystroke to this action in the IDE tools Options dialogon the Shortcuts page. If true, this action will be available in the Visual Studio Options dialog on the Keyboard page. If you want to place this action in a menu or on a command bar, then you must set this property to true.
RegisterMenuButton	Registers this menu button for a Visual Studio command. It works only if option RegisterInVS is set to true.
Supported	Indicates if this action is supported in the current context. You can change this property in a BeforeQueryStatus event handler.
ToolbarItem	An instance of the "DevExpress.CodeRush.Core.ToolbarAction" object that allows you to add this action to the DXCore Visualize toolbar. The ToolbarAction object has the following properties: "BeginGroup", "ButtonIsPressed", "Caption", "Image", "Index", "PlaceIntoToolbar".
UserFeatureUI	Show Feature UI before executing this action.
VisibleToUsers	If true, this Action will be visible to users, and will appear in the command list on the Shortcuts options page, and will also be documented in the User Guide. Setting this property to false will prevent this Action from appearing in the command list. However, bindings to this Action can still be established by typing in the Action name.

Events:

Event name	Description
BeforeQueryStatus	Occurs when the IDE requests the status of the action.
CheckAvailability	Checks the availability of this action in the current context.
Execute	Occurs when this action is executed. Fired whenever the assigned shortcut is triggered or the representative menu item is chosen.
PositionMenuButton	Occurs when the menu button for this action is being positioned within its parent menu or toolbar. Handling this, allows precise positioning of the menu button.
QueryStatus	Occurs before an action is accessed (displayed). This enables properties of this action to be set in the event handler before it reaches the user.
ShowHelp	Occurs before this action is being executed to notify the user about the action.

How to use Action DXCore control

There are only four important things to make the Action component available and working:

1. Create a new DXCore plug-in if you haven't done it yet

2. Drop the **Action** control on the plug-in designer surface

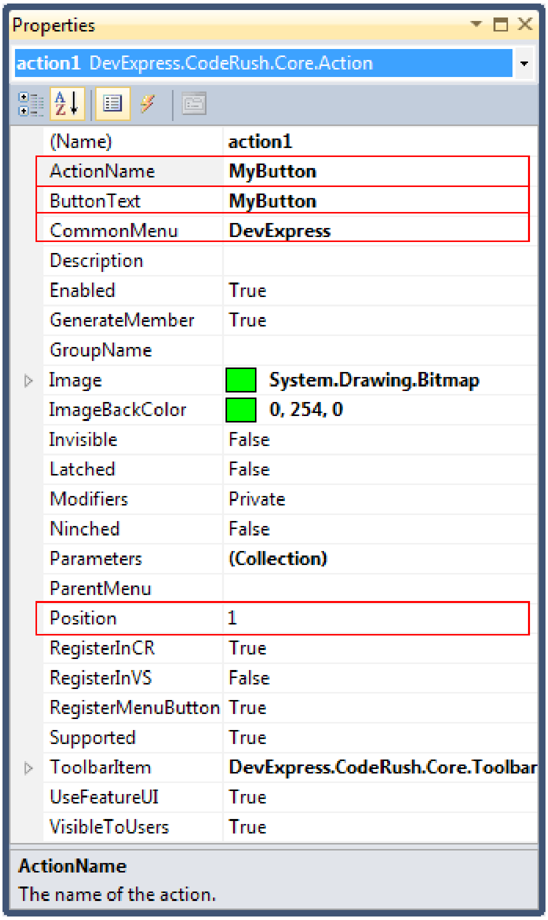
3. Fill the **ActionName** property

4. Handle the **Execute** event

That’s it. Using the **ActionName** you are able to assign a shortcut key, and once it’s pressed, the **Execute** event will do its job – perform the code you added into the event handler.

To make **Action** available inside the Visual Studio’s menu fill the following properties:

- **ButtonText** – the caption of the new button
- **CommonMenu** – the parent menu where the new button will reside
- **Position** – the position inside of the parent menu



Properties

action1 DevExpress.CodeRush.Core.Action

(Name)	action1
ActionName	MyButton
ButtonText	MyButton
CommonMenu	DevExpress
Description	
Enabled	True
GenerateMember	True
GroupName	
Image	🟩 System.Drawing.Bitmap
ImageBackColor	🟩 0, 254, 0
Invisible	False
Latched	False
Modifiers	Private
Ninched	False
Parameters	(Collection)
ParentMenu	
Position	1
RegisterInCR	True
RegisterInVS	False
RegisterMenuButton	True
Supported	True
ToolbarItem	DevExpress.CodeRush.Core.Toolbar
UseFeatureUI	True
VisibleToUsers	True

ActionName
The name of the action.

Here is the code of the **Execute** event handler:

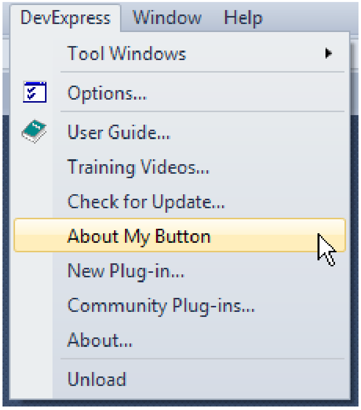
CSharp code:

```
private void action1_Execute(ExecuteEventArgs ea)
```

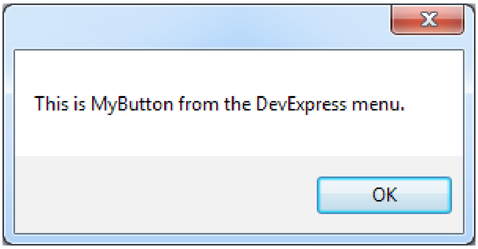
```
{

  MessageBox.Show(«This is MyButton from the DevExpress menu.»);

}
```

Next, compile the plug-in

- Open a new instance of the Visual Studio IDE
- Load **IDE tools** if they're not loaded automatically
- Verify that the new button appeared inside the DevExpress menu:



Once you click the button inside of a menu or press the specified shortcut, your code inside of the **Execute** event handler will be executed:

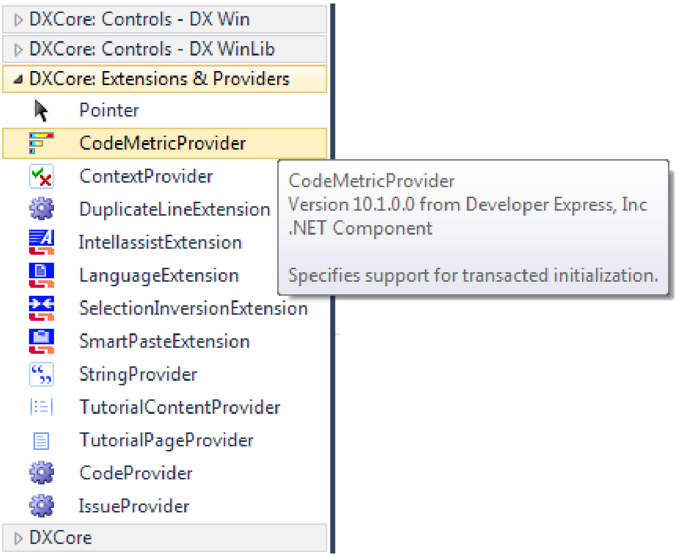


This is MyButton from the DevExpress menu.

DXCore Components – CodeMetricProvider

The **CodeMetricProvider** component calculates and returns a code metric used in the **Metrics tool window** of Refactor! Pro and the Show Metrics CodeRush feature. See the "Show Metrics" post to learn more about software metrics.

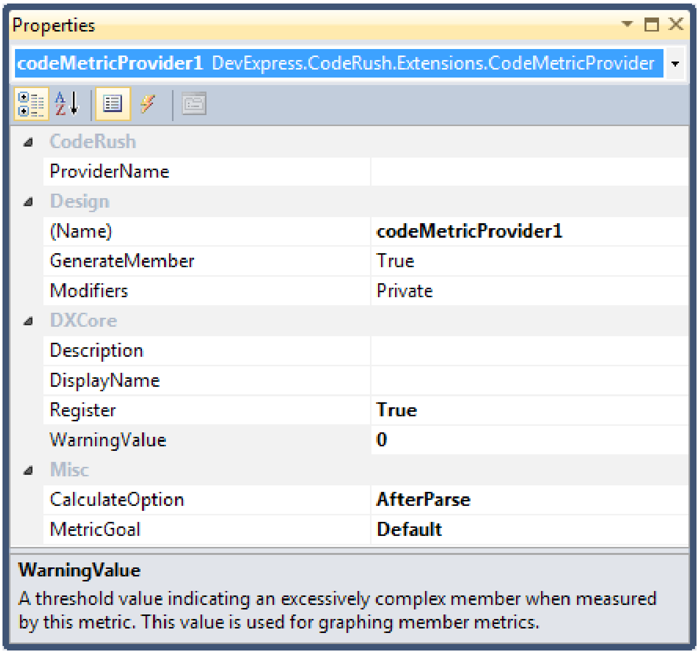
You can drop the **CodeMetricProvider** onto your plug-in design surface from the "***DXCore: Extensions & Providers***" category of Visual Studio Toolbox:



CodeMetricProvider
Version 10.1.0.0 from Developer Express, Inc
.NET Component

Specifies support for transacted initialization.

It's not a visual control, all you need is to tweak its properties and subscribe to its events. Here's the **Properties** window, containing available properties, their categories and default values:

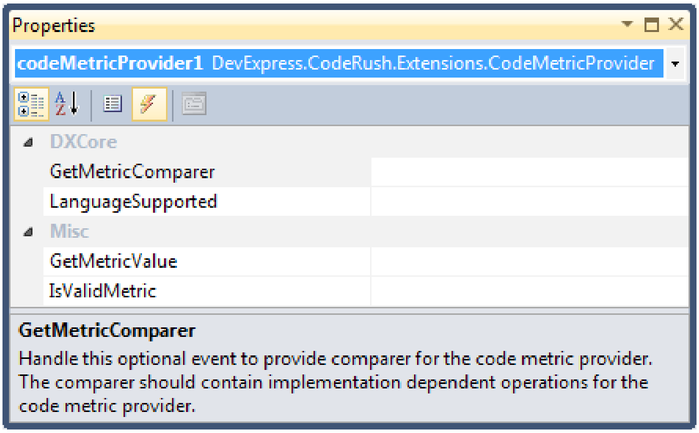


codeMetricProvider1 DevExpress.CodeRush.Extensions.CodeMetricProvider

CodeRush	
ProviderName	
Design	
(Name)	**codeMetricProvider1**
GenerateMember	True
Modifiers	Private
DXCore	
Description	
DisplayName	
Register	**True**
WarningValue	**0**
Misc	
CalculateOption	**AfterParse**
MetricGoal	**Default**

WarningValue
A threshold value indicating an excessively complex member when measured by this metric. This value is used for graphing member metrics.

and the **Events** list:



Properties

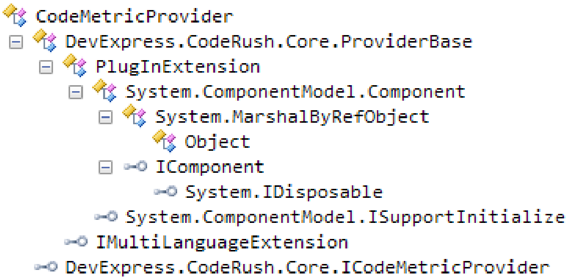
codeMetricProvider1 DevExpress.CodeRush.Extensions.CodeMetricProvider

DXCore
- GetMetricComparer
- LanguageSupported

Misc
- GetMetricValue
- IsValidMetric

GetMetricComparer
Handle this optional event to provide comparer for the code metric provider. The comparer should contain implementation dependent operations for the code metric provider.

The **CodeMetricProvider** component is located in the “***DevExpress.CodeRush.Extensions***” assembly. It is derived from the “***DevExpress.CodeRush.Core.ProviderBase***” class:



```
CodeMetricProvider
⊟ DevExpress.CodeRush.Core.ProviderBase
    ⊟ PlugInExtension
        ⊟ System.ComponentModel.Component
            ⊟ System.MarshalByRefObject
                  Object
            ⊟ IComponent
                  System.IDisposable
            System.ComponentModel.ISupportInitialize
        IMultiLanguageExtension
    DevExpress.CodeRush.Core.ICodeMetricProvider
```

Here’s a list of **properties** in alphabetical order:

Property name	Description
CalculateOption	Specifies one of the two options: *AfterParse* or *UserInvoked*, which indicates the time when the metric is being calculated – after solution parsing is done or manually by user request.
Description	Text, describing the purpose or behavior of this plug-in extension. This text may appear inside the User Guide, configuration UI, or the**About** box.
DisplayName	The text name that identifies this extension. This text may appear inside the **User Guide**, configuration UI, or the **About** box.
MetricGoal	Specifies the value of three states: *Default*, *Members* or *Types*, indicating the objects for which the metric is being calculated.
ProviderName	The name of this provider.
Register	If true, this extension will be available to the **DXCore**. Otherwise, it will be hidden.
WarningValue	A threshold value indicating an excessively complex member when measured by this metric. This value is used for graphing member metrics.

Events:

Event name	Description
GetMetricComparer	Handle this optional event to provide a comparer for the code metric provider. The comparer should contain implementation dependent operations for the code metric provider.

GetMetricValue	Handle this event to calculate and return a code metric value for the given language element.
IsValidMetric	Handle this optional event to specify if it is appropriate to calculate a code metric for the given language element.
LanguageSupported	Handle this optional event if you want to restrict your code metric provider to one or more programming languages. If you do not handle this event, the DXCoremakes your code metric provider available in all programming languages.

How to use CodeMetricProvider control to build you own code metric

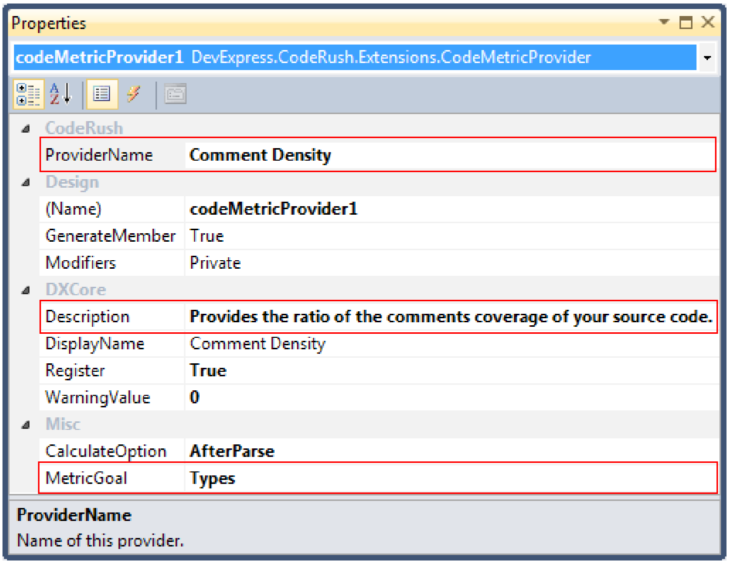
Let's implement the **Comment Density** code metric using the CodeMetricProvider DXCore component. This code metric provides the ratio of comment lines to all lines and indicates the comments coverage of your source code. We are going to calculate it in percents for all type declarations like this:

Comment Density = (comment lines / code lines) * 100

The density of comments metric value will be between 0 and 100 percents, and can be used as a quality indicator to see how much of the code is commented. So, if the comment density value for instance is below a certain value defined by the project manager, it indicates that the code is under commented. Let's assume that what's recommended is a commenting of at least 15% of the code and a maximum of 30%.

Follow these steps to get the **CodeMetricProvider** component working:

- Create a new DXCore plug-in if you haven't done it yet
- Drop the **CodeMetricProvider** control on the plug-in designer surface
- Fill the **ProviderName** property of the **CodeMetricProvider**
- Fill the **Description** property of the **CodeMetricProvider** if needed
- Set the **MetricGoal** property to **Types**



codeMetricProvider1 DevExpress.CodeRush.Extensions.CodeMetricProvider

CodeRush	
ProviderName	Comment Density
Design	
(Name)	codeMetricProvider1
GenerateMember	True
Modifiers	Private
DXCore	
Description	Provides the ratio of the comments coverage of your source code.
DisplayName	Comment Density
Register	True
WarningValue	0
Misc	
CalculateOption	AfterParse
MetricGoal	Types

ProviderName
Name of this provider.

- Then, subscribe to the **GetMetricValue** event:

CSharp code:

```
private void codeMetricProvider1_GetMetricValue(object sender, GetMetricValueEventArgs e)

{


}
```

Next, add a code for metric calculation into the **GetMetricValue** event handler:

CSharp code:

```
private void codeMetricProvider1_GetMetricValue(object sender, GetMetricValueEventArgs e)

{

  // determine if we are working with types...

  TypeDeclaration type = e.LanguageElement as TypeDeclaration;

  if (type == null)

    return;



  // counters...

  float typeCodeLines = type.Range.LineCount;

  float typeCommentLines = 0;



  // enumerating all comments in the given type...

  ElementEnumerable enumerable = new ElementEnumerable(type, LanguageElementType.Comment, true);

  foreach (Comment comment in enumerable)

    typeCommentLines += comment.Range.LineCount;



  // return result...

  e.Value = (int)(typeCommentLines / typeCodeLines * 100);

}
```

That’s it.

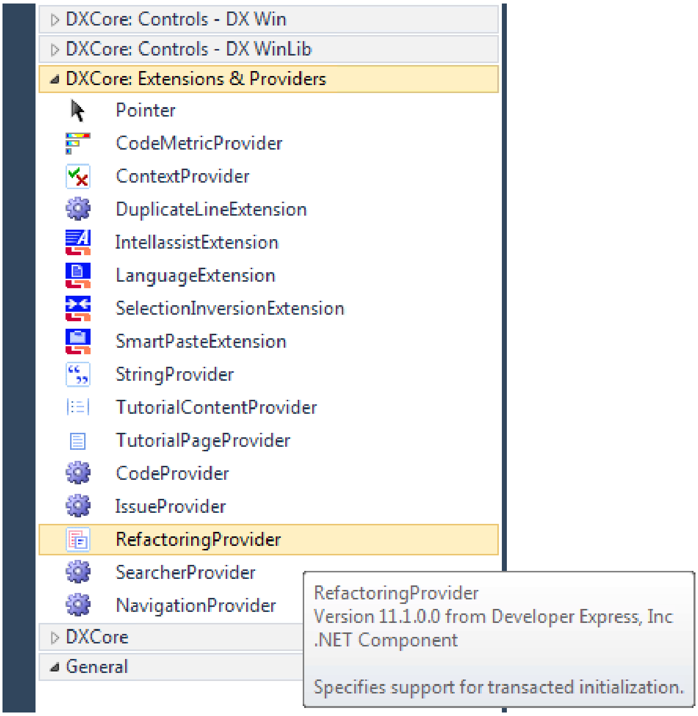
- Compile the plug-in
- Open a new instance of the Visual Studio IDE
- Load **IDE tools** if they’re not loaded automatically
- Open any project and check how the new code metric works inside the **Refactor! Metrics** tool window (**DevExpress** | **Tool Windows** | **Metrics**):

If you have any issues or need any further assistance, please post a comment here or contact DevExpress Support Services.

DXCore components – Refactoring provider

The **RefactoringProvider** DXCore component provides a specific code refactoring operation, which can improve your source code without changing its logic in a definite context.

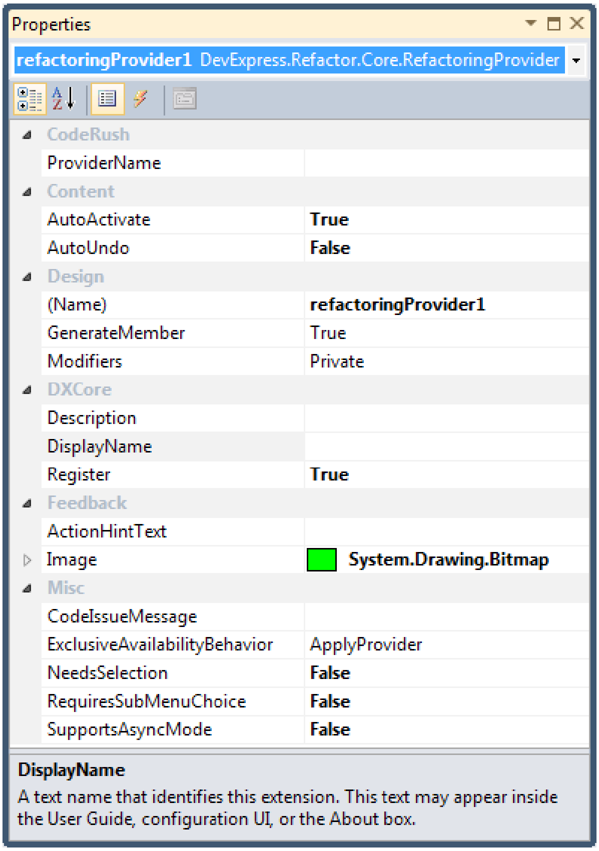
You can drop the **RefactoringProvider** control onto your plug-in design surface from the “**DXCore: Extensions & Providers**” category of Visual Studio Toolbox:



RefactoringProvider
Version 11.1.0.0 from Developer Express, Inc
.NET Component

Specifies support for transacted initialization.

Here’s the *Properties* window, containing available properties, their categories and default values:

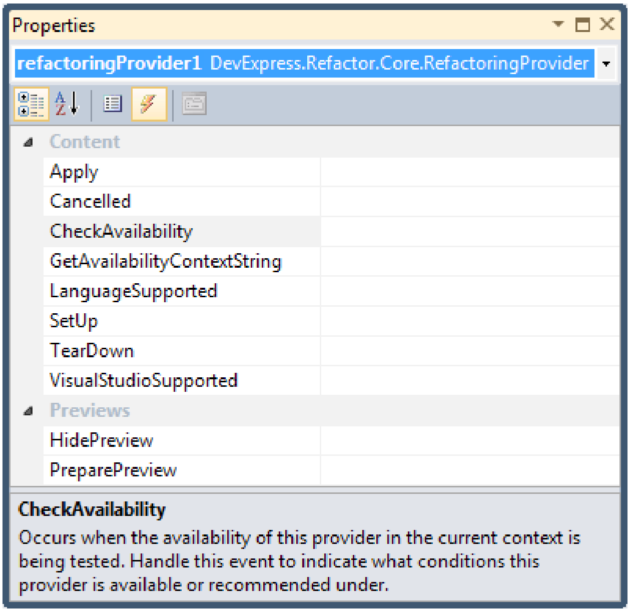


refactoringProvider1 DevExpress.Refactor.Core.RefactoringProvider

CodeRush	
ProviderName	
Content	
AutoActivate	**True**
AutoUndo	**False**
Design	
(Name)	**refactoringProvider1**
GenerateMember	True
Modifiers	Private
DXCore	
Description	
DisplayName	
Register	**True**
Feedback	
ActionHintText	
Image	🟩 **System.Drawing.Bitmap**
Misc	
CodeIssueMessage	
ExclusiveAvailabilityBehavior	ApplyProvider
NeedsSelection	**False**
RequiresSubMenuChoice	**False**
SupportsAsyncMode	**False**

DisplayName
A text name that identifies this extension. This text may appear inside the User Guide, configuration UI, or the About box.

and the *Events* list:

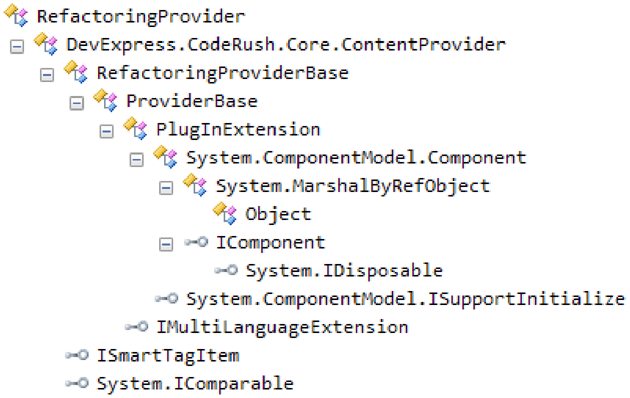


refactoringProvider1 DevExpress.Refactor.Core.RefactoringProvider

Content	
Apply	
Cancelled	
CheckAvailability	
GetAvailabilityContextString	
LanguageSupported	
SetUp	
TearDown	
VisualStudioSupported	
Previews	
HidePreview	
PreparePreview	

CheckAvailability
Occurs when the availability of this provider in the current context is being tested. Handle this event to indicate what conditions this provider is available or recommended under.

The **RefactoringProvider** component is located in the “***DevExpress.Refactor.Core***” assembly. It is derived from the “*ContentProvider*” base type from the “***DevExpress.CodeRush.Core***” assembly:



```
RefactoringProvider
 ⊟ DevExpress.CodeRush.Core.ContentProvider
    ⊟ RefactoringProviderBase
       ⊟ ProviderBase
          ⊟ PlugInExtension
             ⊟ System.ComponentModel.Component
                ⊟ System.MarshalByRefObject
                     Object
                ⊟ IComponent
                     System.IDisposable
               System.ComponentModel.ISupportInitialize
            IMultiLanguageExtension
      ISmartTagItem
      System.IComparable
```

Here’s a list of properties in alphabetical order with descriptions:

Property name	Description
ActionHintText	Gets or sets the text that will be displayed in an action hint after this provider is applied.
AutoActivate	If true, this provider will be automatically activated while it is executing and deactivated afterwards. If false, the provider needs to call the*Activate* and *Deactivate* methods of this provder at the appropriate times.
AutoUndo	If true, an undo unit will automatically be created to collect all text changes made by this provider during the *Apply* event. If false, the provider is responsible for handling the undo stack itself.
CodeIssueMessage	Set a unique string that connects this content provider to a particular code issue. This string should also be passed to the *ea.AddIssue*() method, called from the IssueProvider’s *CheckCodeIssues* event.
Description	Text, describing the purpose or behavior of this plug-in extension. This text may appear inside the User Guide, configuration UI, or the **About** box.
DisplayName	The text name that identifies this extension. This text may appear inside the **User Guide**, configuration UI, or the **About** box.
ExclusiveAvailabilityBehaviour	Determines what will happen when the **Refactor action** is invoked without parameters and this is the only provider available. Default behavior is to immediately apply the provider. However for certain providers having unexpected or potentially surprising side-effects, it may make more sense to show the popup menu, which provides an opportunity to review the impact (through the preview hint) before committing.
Image	Sets the bitmap image for this smart tag item. Images are optional, but can be useful for displaying in the context menu or the popup menu.
NeedsSelection	If true, this content provider requires a selection to determine availability. This is used by the Code Issues engine to select the range of the issue before a check is made (through the *CheckAvailability* event) to see if this content provider can provide a fix for the issue in question.
ProviderName	The name of this provider.

Register	If true, this extension will be available to the DXCore. Otherwise, it will be hidden.
SupportsAsyncMode	If true, the *CheckAvailability* event handler is performed on a separate thread, instead of the main UI thread.

Events list:

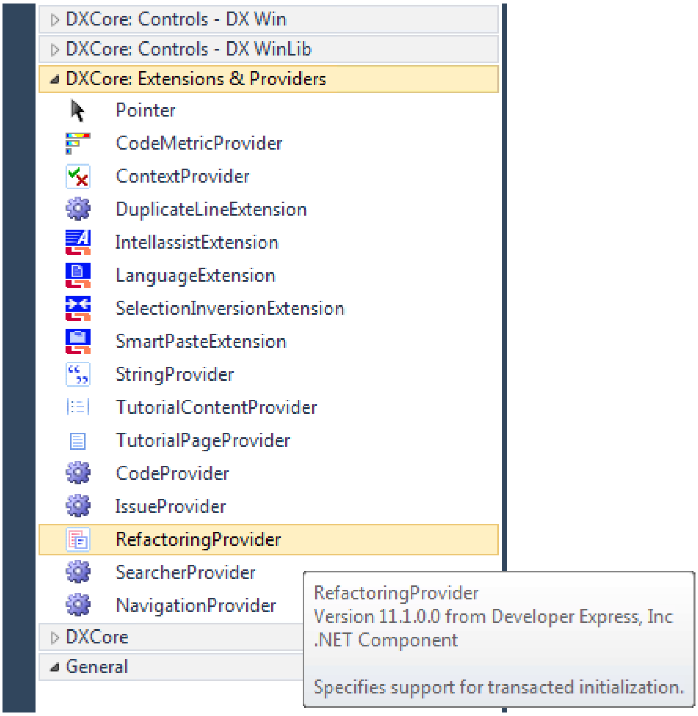
Event name	Description
Apply	Occurs when the provider needs to be applied. Handle this event to apply this provider to the currently existing code in the active text document. Note, that this event will never fire if the provider is not available so code that checks for provider availability is not necessary in a handler for this event.
Cancelled	Occurs when this provider has been cancelled. Implement code that will cancel this provider inside a handler for this event.
CheckAvailability	Occurs when the availability of this provider in the current context is being tested. Handle this event to indicate under what conditions this provider is available or recommended.
GetAvailabilityContextString	Occurs before the smart tag manager determines the command order based on historic usage. If your provider has distinct states that you would like separate entries for, handle this event and assign a unique string to the *Context* property of the event arguments that represents the distinct state. For example, the **Inline Temp** provider handles this event and sets the Context property to “″⁾ when only one instance of the temporary variable exists.”
HidePreview	Occurs when a menu item inside the popup menu is no longer highlighted. This can happen when the user selects a different provider, when the user executes the selected provider, or when the popup menu closes. If you display visual information in the *PreparePreview* event, you should also handle this event to clean up the preview.
LanguageSupported	Handle this optional event if you want to restrict your context to one or more programming languages. If you do not handle this event, theDXCore makes your context available in all programming languages.
PreparePreview	Occurs when the user highlights this provider inside the popup menu. Handle this optional event to display any additional visual information that can preview the impact of this provider. If you handle this event, you should clean up your preview in a *HidePreview* event handler.
SetUp	Occurs immediately before the *Apply* event is fired. Handle this optional event to perform any necessary setup before the provider is applied.
TearDown	Occurs immediately after the *Apply* event is fired. Handle this optional event to perform any necessary clean up after the provider is applied.
VisualStudioSupported	Handle this optional event if you want to restrict your provider to one or more Visual Studio versions. If you do not handle this event, your provider will be available in any Visual Studio version.

Creating a simplest refactoring using the RefactoringProvider control

In this article we will create the simplest refactoring which will remove the active comment. Sometimes, there are comments left that are not utilized any longer, such as implemented TO-DO comments, UNDONE comment that is now complete, etc. We will use the RefactoringProvider control shipped inDevExpress Refactor! Pro for this purpose.

Here are the steps we need to perform:

1. Create a new DXCore plug-in
2. Drop the RefactoringProvider component from the Toolbox:

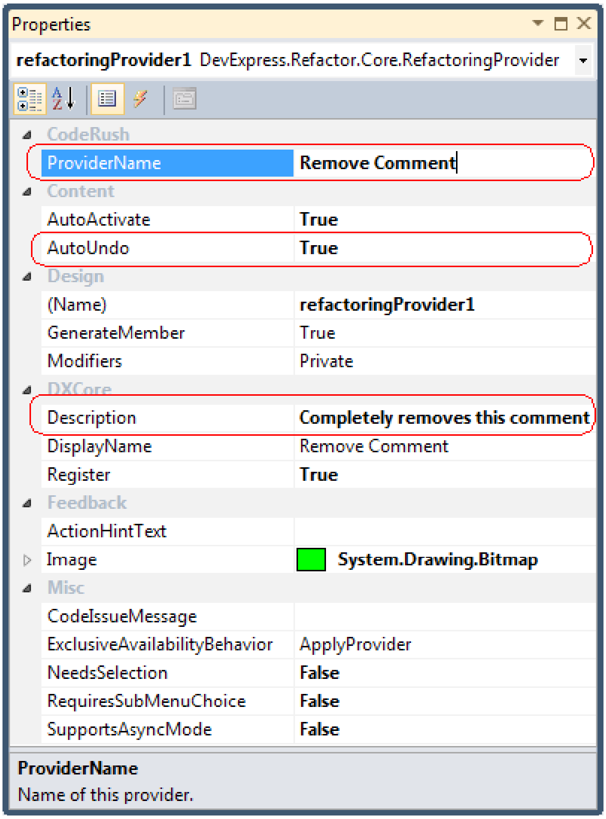


RefactoringProvider
Version 11.1.0.0 from Developer Express, Inc
.NET Component

Specifies support for transacted initialization.

3. Fill its properties:

- ProviderName (*e.g. "Remove Comment"*)
- Description (*e.g. "Completely removes this comment and its neighboring comments"*)
- AutoUndo (*true*)



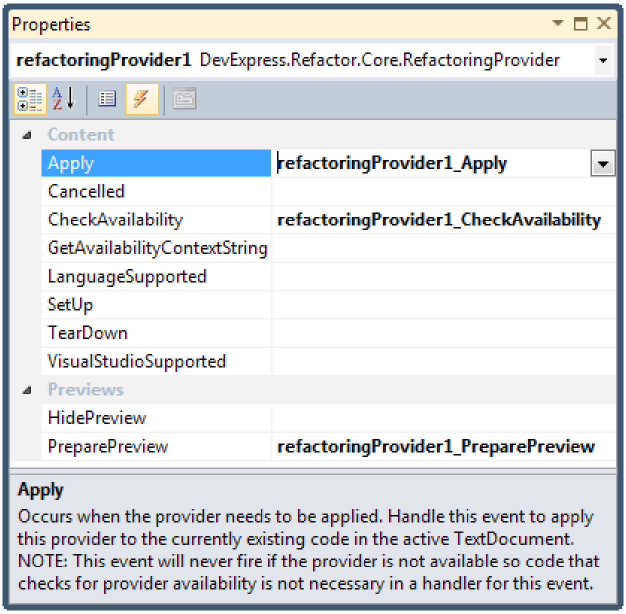
Properties

refactoringProvider1 DevExpress.Refactor.Core.RefactoringProvider

CodeRush	
ProviderName	Remove Comment
Content	
AutoActivate	True
AutoUndo	True
Design	
(Name)	refactoringProvider1
GenerateMember	True
Modifiers	Private
DXCore	
Description	Completely removes this comment
DisplayName	Remove Comment
Register	True
Feedback	
ActionHintText	
Image	🟩 System.Drawing.Bitmap
Misc	
CodeIssueMessage	
ExclusiveAvailabilityBehavior	ApplyProvider
NeedsSelection	False
RequiresSubMenuChoice	False
SupportsAsyncMode	False

ProviderName
Name of this provider.

4. Subscribe to the *CheckAvailability*, *Apply* and *PreparePreview* events:



Properties

refactoringProvider1 DevExpress.Refactor.Core.RefactoringProvider

Content	
Apply	refactoringProvider1_Apply
Cancelled	
CheckAvailability	refactoringProvider1_CheckAvailability
GetAvailabilityContextString	
LanguageSupported	
SetUp	
TearDown	
VisualStudioSupported	
Previews	
HidePreview	
PreparePreview	refactoringProvider1_PreparePreview

Apply
Occurs when the provider needs to be applied. Handle this event to apply this provider to the currently existing code in the active TextDocument. NOTE: This event will never fire if the provider is not available so code that checks for provider availability is not necessary in a handler for this event.

5. Add the following code into event handlers:

CheckAvailability event handler

To indicate that the refactoring provider is available in the current context, we need to set the corresponding *Available* property. This refactoring should be available when the active code element is a comment, so we simply check if the element is of the Comment type:

```
private void refactoringProvider1_CheckAvailability(object sender, CheckConten-
tAvailabilityEventArgs ea)

{

  ea.Available = ea.Element is Comment;


}
```

Apply event handler

In the *Apply* event handler we are going to remove the comment with its neighboring comments and surrounding white space characters including empty lines. To do this, we get the active comment, then calculate its cut range for removal using the special SourceRange methods, and, finally, deleting the calculated range:

```
private void refactoringProvider1_Apply(object sender, ApplyContentEventArgs ea)

{

  SourceRange cutRange = GetCutRange(ea.Element as Comment);

  ea.TextDocument.DeleteText(cutRange);


}
```

GetCutRange method:

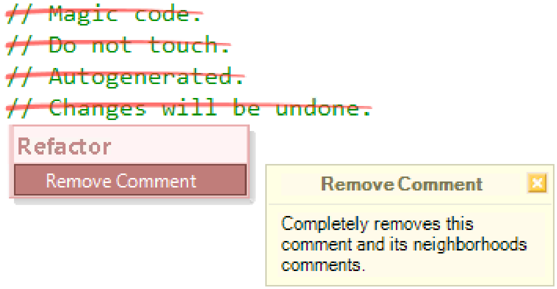
```
private SourceRange GetCutRange(Comment comment)

{

  Comment startComment;

  Comment endComment;

  comment.GetFirstAndLastConnectedComments(out startComment, out endComment);



  BlockElements blockElements = BlockElements.AllWhiteSpaces | BlockElements.
SupportComments;

  return new SourceRange(startComment.GetFullBlockCutRange(blockElements).
Start,

 endComment.GetFullBlockCutRange(blockElements).End);


}
```

PreparePreview event handler

This event handler is required if you would like to show a preview hint for the refactoring. We will strike-through the cut range of a comment:

```
private void refactoringProvider1_PreparePreview(object sender, PrepareContent-
PreviewEventArgs ea)

{

  SourceRange cutRange = GetCutRange(ea.Element as Comment);

  ea.AddStrikethrough(cutRange);


}
```

Now, compile the plug-in and see it in action:



```
~~// Magic code.~~
~~// Do not touch.~~
~~// Autogenerated.~~
~~// Changes will be undone.~~
```

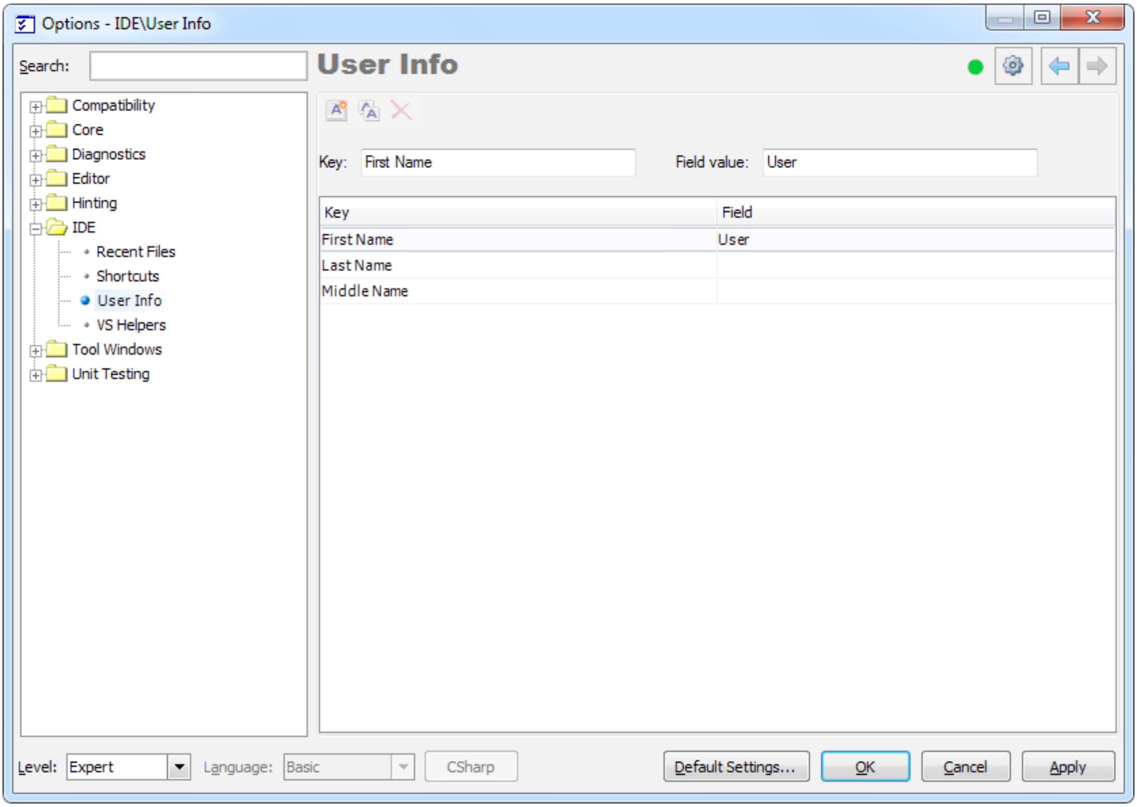
Completely removes this comment and its neighborhoods comments.

The sample code is attached (*14,098 bytes, Visual Studio 2010, C#*).

How to specify the CodeRush user information and use it inside text expansions

If you use the CodeRush code templates heavily, and they specify the author of the source code, you may find the **IDE | User Info** options page interesting. On this page, you can specify your first, last and middle name, and then use them inside the text expansions, for example, to create a file header and specify its author.

The information on the **User Info** page in the IDE Tools Options Dialog is personal for every user. If you create code templates that use the information from that page, other users that share the same templates do not have to edit templates. Instead, they only need to specify their names on the page, which is much easier. Here is what it looks like:



Key: First Name Field value: User

Key	Field
First Name	User
Last Name	
Middle Name	

Level: Expert Language: Basic

By default, there are only three key/values available:

- *First Name* – if not specified, automatically filled-in with the currently logged in user name.
- *Middle Name* – your middle name
- *Last Name* – your last name

You can add an additional key/value pair by clicking the *New Record* button. For example, you can specify your position or date of birth. Two other buttons allow you to edit or remove existing key/value pairs. Note that you cannot remove the three default key/values, only the items you have added manually.

The values provided on this page can be used by using the corresponding string providers. Here is the list of the user info specific string providers that will return the information that you request:

String Provider Name	Description
GetUserFirstName	Gets the user first name
GetUserInfo	Gets the user information specified by the key name passed as a parameter.
GetUserInitials	Gets the user initials based on the user first and last name.
GetUserLastName	Gets the user last name.
GetUserMiddleName	Gets the user middle name.

Now let’s create a simple template that will generate the file header based on the information from the User Info page. Before we start, specify your first and last name, and add the company field on the options page. Here is what the template may look like:

```
//-----------------------------------------------------------------------

// <copyright file=””«?FileName»”” company=””«?GetUserInfo(Company)»””>

//     Author: «?GetUserFirstName» «?GetUserLastName»

//     Copyright (c) «?GetUserInfo(Company)», Inc. All rights reserved.

// </copyright>

//-----------------------------------------------------------------------
```

Once this template is expanded, you will see, inserted, a header similar to this:

```
//-----------------------------------------------------------------------

// <copyright file=””C:\Projects\TestClass.cs”” company=””DevExpress””>

//     Author: Alex Skorkin

//     Copyright (c) DevExpress, Inc. All rights reserved.

// </copyright>

//-----------------------------------------------------------------------
```

Bear in mind that you can put this header into all source files of your solution automatically with a single keystroke. To learn more, please read the appropriate topic. I hope you find this interesting.

Appendix A. Terminology

Here is a list of terms often used when talking about **IDE tools** (including CodeRush, DXCore and Refactor!) for a better understanding of these products. The list is not complete and it is going to be updated from time to time.

Action

Actions associate a name with functionality, which can be triggered using either a keyboard shortcut or a mouse button, and/or placed within a Visual Studio menu. Some actions accept parameters that can change their behavior.

Action Hint

Action hints inform new users about **IDE tools** features, as they occur by pointing to a location on-screen with a large colored arrow, displaying a short text message.

Adornment

Adornment represents a graphical object inside VisualStudio code editor.

Alias (RegEx Alias)

A number of solutions in **IDE tools** rely upon regular expressions to match a text. **DXCore** extends the regular expression engine by adding aliases – the ability to replace a complex regular expression with a human-readable alias. **DXCore** includes a number of predefined aliases (e.g. *Identifier*, *Argument*,*Param*, *Initialization*, *StringConstant*, *Typecast*, *etc*).

Big Hint

Big hints are large tool tips which describe an active feature or provide other helpful information.

Billboard Hint

Billboard hinting briefly displays a large message that clearly communicates a significant state change. While the billboard hint is on screen, you can continue to work.

Code Declare

Code modification providers that intended to declare new source code language structures, e.g. new types, members, variables.

Code Issues

Also known as **Code Analysis** feature, that performs a static code analysis to find issues in your source code. It marks found issues with colored wavy lines and allows you to perform a fix immediately. The following issues can be found: errors, warnings, hints (suggestions), dead code and code smells.

Code Preview

Code Preview Window shows the syntax-highlighted source code. It is used in refactoring previews and some tool windows.

Code Provider

Code Provider represents a code modification operation provider (similar to a *Refactoring Provider*), which may change program behavior.

CodeRush

CodeRush is a powerful Visual Studio .NET add-in that enhances the developer experience by accelerating developer and team productivity via an integrated suite of technologies.

CodeRush Xpress

CodeRush Xpress is a completely free version of **CodeRush Pro** available to all Visual Studio 2008 and 2010 CSharp and Visual Basic developers, and offers a comprehensive suite of tools that enable you and your team to simplify and shape complex code – making it easier to read and less costly to maintain.

Context

The context determines if a particular feature is appropriate for use when working inside Visual Studio IDE by specifying a collection of boolean states (e.g. "*Caret is inside method*" or "*Menu is active*"). It is useful for distinguishing conditions where a feature should be available.

Decoupled Storage

The DecoupledStorage object is used to handle storage for your **DXCore** plug-ins data.

Dynamic List

Dynamic List represents a named list of mnemonics and its values that are used in the *CodeRush Templates Engine* for increasing the amount of possible template variations.

DXCore

DXCore provides services, wizards, and a visual extensibility framework designed to make it easy to extend Visual Studio. All products like **CodeRush** and **Refactor!** were designed on the **DXCore** Framework.

Editor Caret

Editor caret is a text cursor representing a current position in the Visual Studio code editor.

Embedding

The **Code Embedding** feature allows you to surround the selected block of code or text into an appropriate block.

Feature UI

Feature UI (also known as the "What Happened?" window) appears in the bottom right corner each time a particular **IDE tools'** feature is activated to inform you about what's going on inside Visual Studio IDE. In the majority of cases, the window is used to simplify learning of the **CodeRush** specific features.

Highlighting

Highlighting usually represents one of the CodeRush visualization features intended to visually concentrate and indicate particular source code elements.

Hinting

The discoverability elements to help you learn more about features in **IDE tools**, such as *Action Hints*, *Big Hints*, *Shortcut Hints*, *Billboard Hints*.

IDE tools

IDE tools bundle includes the following Visual Studio productivity add-ins: **DXCore**, **CodeRush Pro**, **CodeRush Xpress**, **Refactor! Pro**, **Refactor! for C++, Refactor! for ASP.NET**.

Intellassist

Intellassist feature auto-completes text at the editor caret with appropriate suggestions like an in-scope identifier, enumeration elements and others.

Inversion

Selection inversions take an existing selection and replace it with code that is essentially the opposite of the original. You can use it to swap assignment statements, invert the logic in boolean assignments to true or false, change the iteration direction on for-loops and more.

Language Element

LanguageElement is the main base class for a rich variety of classes that denote a construct occurring in the source code and present a corresponding node in the abstract source tree built by **DXCore** Framework.

Linked Identifiers

Linked Identifiers are a built-in feature of **DXCore**, which allows you to simultaneously change similar pieces of the text (code) located in different places. If you change one linked identifier, the others that are associated with it will automatically be changed as well.

Markers

Markers are navigation placeholders that remember important locations inside your source code you'll need to move to in the future.

Message Log

Message Log is a diagnostic tool window with valuable information, which can be helpful to troubleshoot issues appearing in **IDE tools**. It is available via the **DevExpress** | **Tool Windows** | **Diagnostics** | **Messages** menu item.

Nav Field

Navigation Fields allow you to quickly navigate between related code fragments. They highlight related fragments and allow you to easily tab through them in both directions.

Nav Link

Navigation Links allow you to quickly navigate between related pieces of code. They highlight particular code fragments and allow you to easily tab through them in both directions.

Nav Provider

Provides a navigation operation intended for quick moving inside your source code.

Options Dialog

Most of the aspects of the **IDE tools** (**CodeRush** and/or **Refactor!**) functionality can be customized using the Options dialog.

Options Page

Options page allows users to easily customize the features of **IDE tools** and other custom plug-ins. Options pages are automatically integrated into the Option Dialog.

Plug-In

A plug-in is a special class that resides in an assembly that is loaded into the Visual Studio environment when **DXCore** starts-up. **DXCore** plug-in enhances the functionality of **IDE tools**.

Plug-In Wizard

Plug-in wizards are simplifying the process of creating **DXCore** plug-ins, such as tool windows, standard plug-ins and options pages. These wizards lay down the framework upon which your plug-ins will grow.

Refactor! Popup Menu

Refactor! Popup menu is a drop-down menu that lists all currently available refactoring/code operations for the current editor caret position and/or code selection.

Refactoring provider

Represents a particular refactoring operation for the restructuring of an existing code block, altering its internal structure without changing its external behavior in order to improve readability and the maintainability of the source code.

Service

Service is a set of a **DXCore** related functionality that you can use when developing plug-ins.

Shortcut Binding

A shortcut binding assigns a key to an action or to a Visual Studio command. Both actions and commands can be assigned keystrokes in both Visual Studio's options dialog and in the **DevExpress** options dialog.

Shortcuts Hint

Shortcut hints list available shortcuts for the active state. Shortcut hints usually appear in response to a newly activated in-source user interface.

Smart Tag

Smart tag is small glyph that indicates the availability of refactorings or code provider operations.

String Providers

String providers are functions that return strings, and can be used to provide calculated information (e.g., date/time stamps, user names, active file name, etc.).

Target Fields

The target indicator shows where the caret will be positioned when you press **Num Enter** or **Enter** to accept last active text field.

Templates

Code templates are blocks of code that expand from short abbreviations typed into the code editor. Using code templates decreases code creation time, because it avoids having to type the entire code manually and allows creating regular code sections with only a few keystrokes.

Text Commands

Text commands are participating in the **CodeRush** code modification engine to perform a specific operation, e.g. move the editor caret, add an assembly reference, paste clipboard contents, invoke the Intellisense, etc.

Text Document

Text document represents an opened source code file in the Visual Studio code editor.

Text Expansion

Text expansion is a dynamic text containing encoded text commands and/or string providers, which is expanded into a block of code (text) with various elements of user/code interaction.

Text Fields

Text fields allow you to enter data and then press **Num Enter** or **Enter** to quickly move to the next field. Text fields are highlighted in the editor and usually have a tooltip below them describing the nature of the data expected.

Text View

The text view represents a view into a text document. Visual Studio may offer several views by splitting the main text document window horizontally or vertically. Text views are useful for painting on the editor and also for working with selections and the editor caret.

Tile

Tiles are rectangular regions on a text view designed to interact using the mouse. For example, CodeRush's Flow Break Icons feature is implemented with this technology.

Tool Window

Tools windows are child windows of the Visual Studio integrated development environment that are used to display information or provide a different kind of IDE functionality. For example, there are: CodeRush tool window and User Guide tool window.

User Guide

The User Guide tool window documents **DXCore** and dependent products, such as **CodeRush** and/or **Refactor!** if they're installed. It also includes documentation covering Visual Studio extensibility using **DXCore** Framework.

Visualize Toolbar

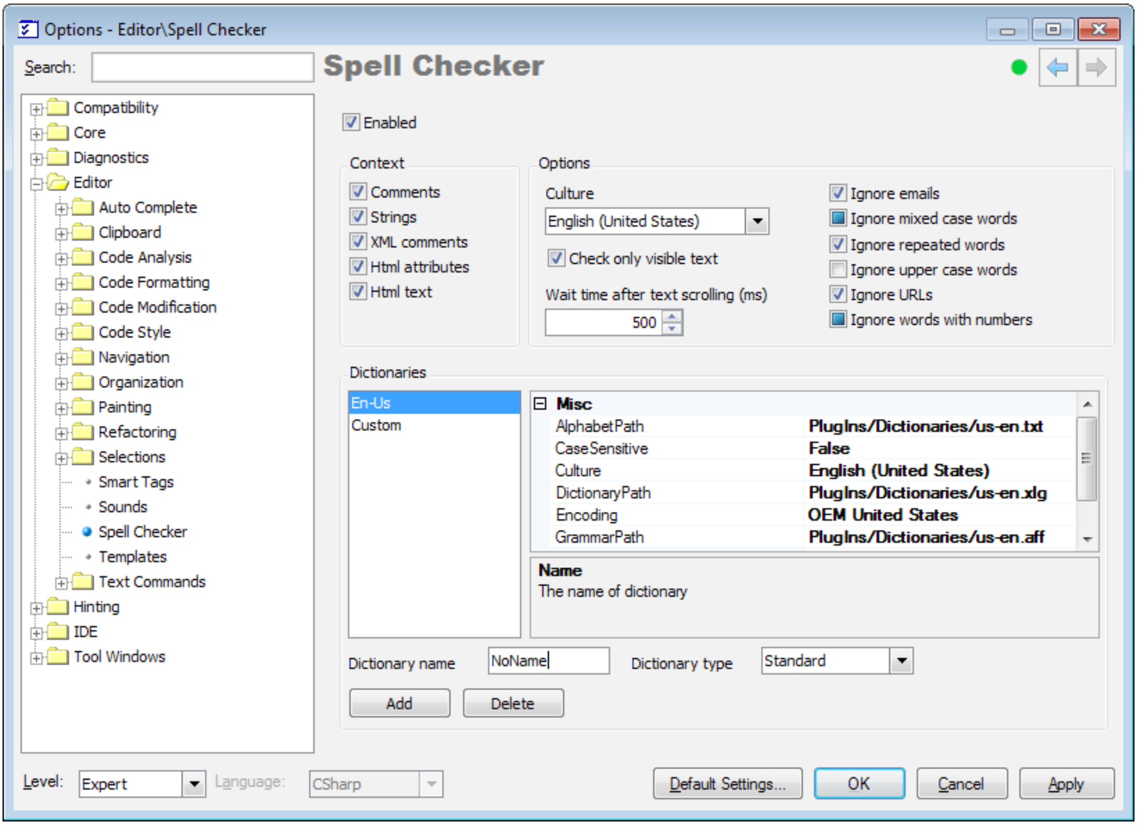
The Visualize Toolbar is similar to all other standard Visual Studio toolbars and contains a group of toggle buttons, allowing you to quickly turn off commands or controls that are typically related in their function.

Appendix B. CodeRush Options

IDE Tools Options Dialog

Most of the aspects of the **IDE tools** (CodeRush and/or Refactor!) functionality can be customized using the **Options dialog**, which can be accessed via the **DevExpress** | **Options**… menu item in your IDE. You can also use the **CTRL+SHIFT+ALT+O** shortcut.

Here's how it looks like:



☑ Enabled

Context

☑ Comments
☑ Strings
☑ XML comments
☑ Html attributes
☑ Html text

Options

Culture: English (United States)

☑ Check only visible text

Wait time after text scrolling (ms): 500

☑ Ignore emails
■ Ignore mixed case words
☑ Ignore repeated words
□ Ignore upper case words
☑ Ignore URLs
■ Ignore words with numbers

Dictionaries

En-Us
Custom

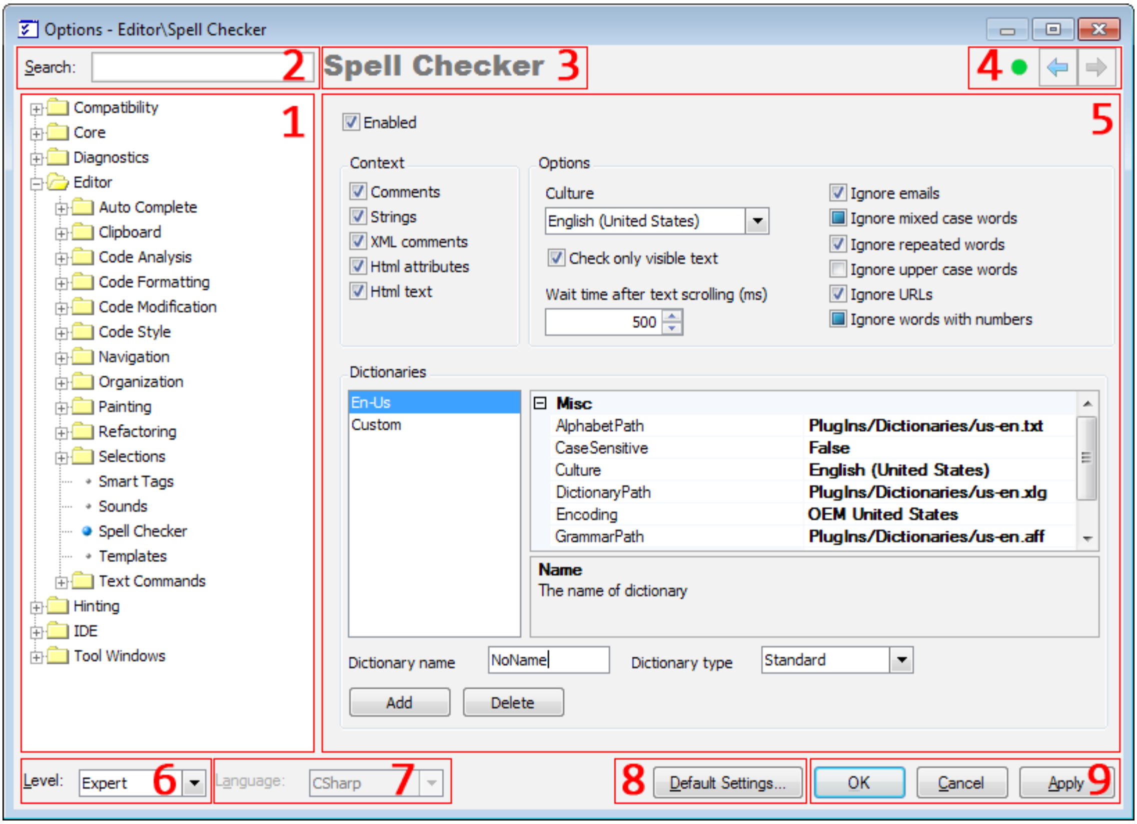
⊟ Misc	
AlphabetPath	PlugIns/Dictionaries/us-en.txt
CaseSensitive	False
Culture	English (United States)
DictionaryPath	PlugIns/Dictionaries/us-en.xlg
Encoding	OEM United States
GrammarPath	PlugIns/Dictionaries/us-en.aff

Name
The name of dictionary

Dictionary name: NoName
Dictionary type: Standard

Level: Expert
Language: CSharp

Now let's take a closer look at all the parts of this dialog:



Options - Editor\Spell Checker

Search:

Spell Checker

☑ Enabled

Context

☑ Comments
☑ Strings
☑ XML comments
☑ Html attributes
☑ Html text

Options

Culture: English (United States)

☑ Check only visible text

Wait time after text scrolling (ms): 500

☑ Ignore emails
■ Ignore mixed case words
☑ Ignore repeated words
□ Ignore upper case words
☑ Ignore URLs
■ Ignore words with numbers

Dictionaries

En-Us
Custom

Misc	
AlphabetPath	PlugIns/Dictionaries/us-en.txt
CaseSensitive	False
Culture	English (United States)
DictionaryPath	PlugIns/Dictionaries/us-en.xlg
Encoding	OEM United States
GrammarPath	PlugIns/Dictionaries/us-en.aff

Name
The name of dictionary

Dictionary name: NoName

Dictionary type: Standard

Level: Expert

Language: CSharp

1) **Sections/Categories**. All standard **DXCore** pages are subdivided into the following sections:

Section	Description
Compatibility	The Compatibility section contains **IDE tools** and Visual Studio compatibility options.
Core	Contains **DXCore** options.
Diagnostics	Contains **IDE tools** diagnostics options.
Editor	This section contains the primary option set. Options of this section are related to the code editor.
Hinting	Contains **IDE tools** hints options.
IDE	Contains options that affect Visual Studio behavior.
Tool Windows	The built-in tool windows options, e.g. User Guide tool window.

2) **Search** text box

The search text box in the upper left-most position of the options dialog is used to incrementally search for available pages by their title for quick navigation to a desired page.

3) **Page** name

The page name you’re currently viewing.

4) **Navigation buttons** and user level icon

In the upper right corner of the dialog, there are two **Navigation** buttons. These function in the same way as the "back" and "forward" buttons, allowing you to navigate back to a page previously visited, and then forward again to where you started. Near the navigation buttons there's a user level icon, so you can see a colored icon which represents the level of the current page. See **User Level** combo box (#6) for details.

5) **Options page contents**

This area shows all available settings for the current options page.

6) **User Level** combo box

There are three experience levels to choose from – **New User**, **Advanced** and **Expert**. Choosing the **New User** option will result in only the most commonly used settings being displayed, and the rest, which are less frequently used, being filtered out, making it easier to customize the settings.

7) **Language** selector

Some options are dependent upon the programming language used in a project. Use the language selector to specify the programming language that you are customizing the options for. Note that if the currently active project's language differs from the selected one, a warning hint will be displayed. See the "How the Language combobox on the Options Dialog is being populated" topic to learn more.

8) **Default Settings**… button

You can restore the default settings (for pages that support this) at any time by clicking the **Default Settings**… button. If you'd like to restore all factory default settings see the "Where IDE tools settings are stored" topic.

9) **OK**, **Cancel**, **Apply** buttons

These are self explanatory.

Notes:

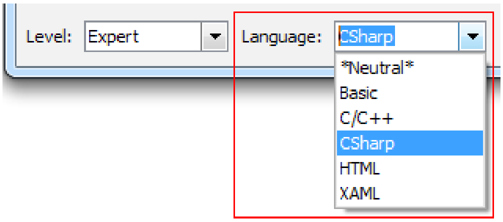
- Unfortunately, the options dialog is modal at the moment, which doesn't allow you to test settings when it is up.

- The **Options Dialog** remembers its size, location, and the last page you visited when closed. So, after the dialog is reopened, you'll see the same page you viewed before it was closed.

- You can easily add your own options pages to this dialog if you're writing **DXCore** plug-ins. They will appear in the standard sections or in a section that you choose.

How the Language combobox on the Options Dialog is being populated

Some people are asking how the **Language** combobox is populated in the Options Dialog. Once you have installed IDE tools and open the **Options Dialog,** you will see that the **Language** combobox is empty. But after some work in Visual Studio IDE this combobox is being populated with items. So, when and how is it populated?



The answer is that this combobox is populated with items when you open up some source files in Visual Studio IDE and have a corresponding language service for that file. So, if you open up an F# file (.fs) and you have the FSharp language service installed, you will see a new entry.

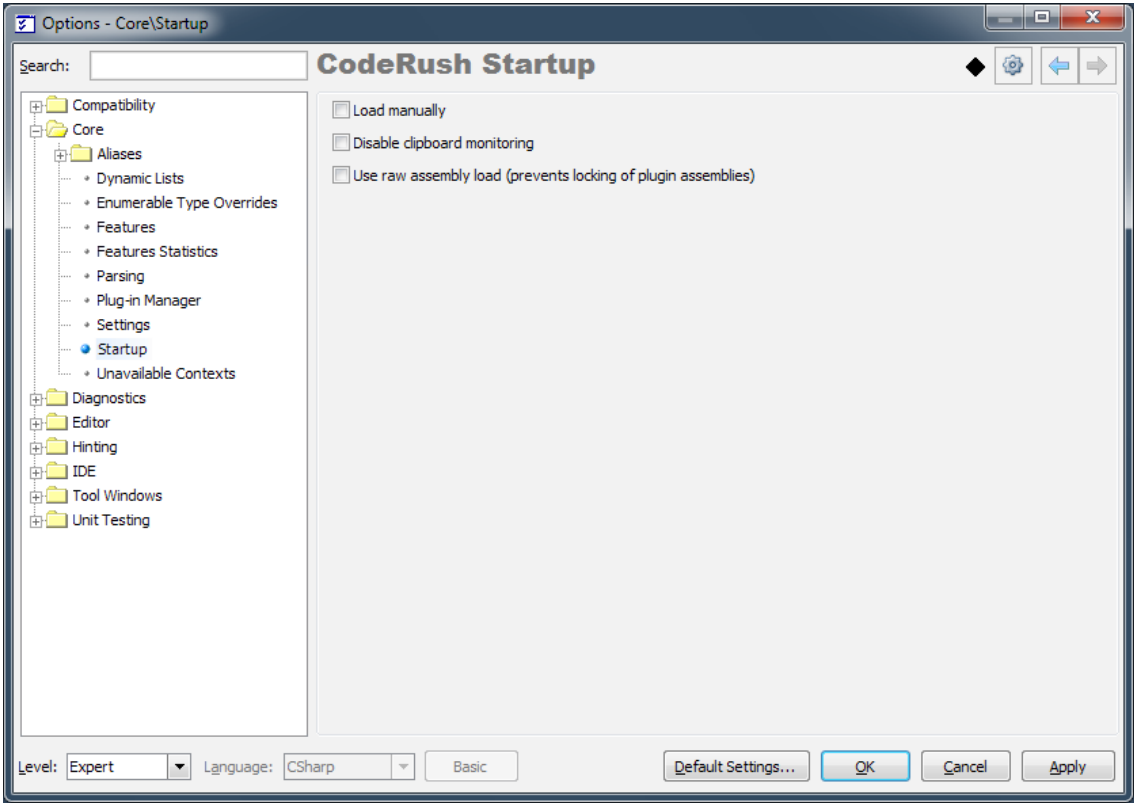
You may have a list that contains some of the following items:

- CSharp
- Basic
- C/C++
- FSharp
- JavaScript
- HTML
- XML
- CSS
- XAML
- T-SQL
- Plain Text

and others.

Startup options page

The **Startup** options page from the *Core* category allows you to tweak the IDE Tools start-up settings. The page level is expert, so do not change these options if you do not actually need this. Here is what the options page looks like (*click the image to enlarge*):



Options - Core\Startup

CodeRush Startup

☐ Load manually

☐ Disable clipboard monitoring

☐ Use raw assembly load (prevents locking of plugin assemblies)

Level: Expert

Language: CSharp

Available settings are:

- **Load manually**

Allows you to toggle automatic loading of **IDE Tools**. If checked, IDE Tools will not be automatically loaded. In this case, you have to select DevExpress -> Load or Tools -> Load DevExpress menu item (depending on the IDE Tools version you use). To learn more about the manual loading process, please refer to the “How to temporally disable IDE tools (load manually)” topic.

- **Disable clipboard monitoring**

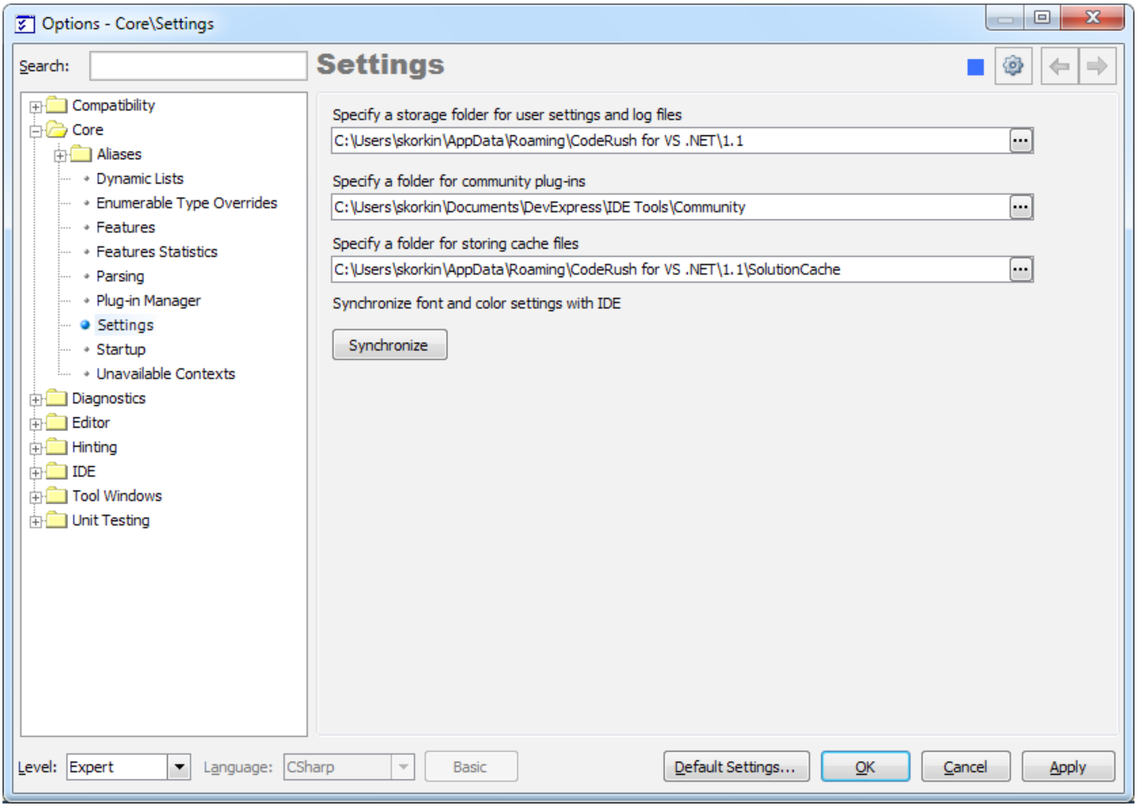
DXCore framework sets up a Windows Clipboard hook to monitor its content and clipboard-specific events. This option toggles the hook operability. If unchecked, then clipboard features will become disabled. The option is used when you have issues with the Windows Clipboard.

- **Use raw assembly load (prevents locking of plug-in assemblies)***

Specifies how the **DXCore Plug-in Loader** loads the plug-ins assemblies. If you are a DXCore plug-ins developer, use this option, so your plug-in assemblies are not being locked-up when they are loaded into another instance of the Visual Studio IDE. However, you cannot restart an already loaded plug-in – you have to restart your Visual Studio IDE to reload the recompiled plug-in. Note that this option is available starting from the next 2011.2 IDE Tools release.

Settings paths option page

IDE Tools (CodeRush, Refactor! and DXCore) use several physical paths to store its settings, log and cache files, and community plug-ins. All these paths are configurable on the **Core** | **Settings** options page in the Options Dialog. This is what the page looks like:



Settings

Specify a storage folder for user settings and log files: C:\Users\skorkin\AppData\Roaming\CodeRush for VS .NET\1.1

Specify a folder for community plug-ins: C:\Users\skorkin\Documents\DevExpress\IDE Tools\Community

Specify a folder for storing cache files: C:\Users\skorkin\AppData\Roaming\CodeRush for VS .NET\1.1\SolutionCache

Synchronize font and color settings with IDE

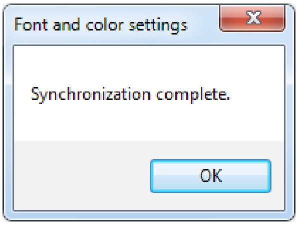
Level: Expert
Language: CSharp

You can specify the following paths:

- A storage folder for the user settings and the log files. This folder, in turn, contains two folders:
  - the “Settings.xml” folder where all settings of IDE Tools are stored
  - the “Log” folder with various log files, which can be used to troubleshoot the IDE Tools
- A folder for the community plug-ins. You may install a community plug-in into this folder and they will be automatically loaded by IDE Tools when found.
- A folder for storing the cache files. There are solution and assembly caches generated by DXCore. They are used to improve the overall performance when working with big projects. You can safely remove the cache folder, because it will be recreated by **DXCore** next time you open and use a particular project.

There is also an additional option to synchronize the font and color settings with IDE. These are Visual Studio IDE settings for fonts and colors, accessible via the “Tools -> Options -> Environment -> **Fonts and Colors**” menu item. Retrieving these settings from Visual Studio takes significant time, so IDE Tools simply retrieve them once on the first startup and stores them in its settings files for later reference to improve performance.

These settings are used by the **CodeRush** and **DXCore** features which paint over the Visual Studio code editor. For example, a BigHint component may use the background color specified in the Visual Studio Color options. Or, the Refactor!/CodeRush Popup menu uses the IDE code editor font and its size to correspond to the IDE. If you notice any color or font mismatch for different IDE Tools hints and visual features, or manually change the Visual Studio Font and Color settings, you might need to synchronize them with IDE Tools by clicking the *Synchronize* button. If synchronization is successful, you will see the corresponding message:



Font and color settings

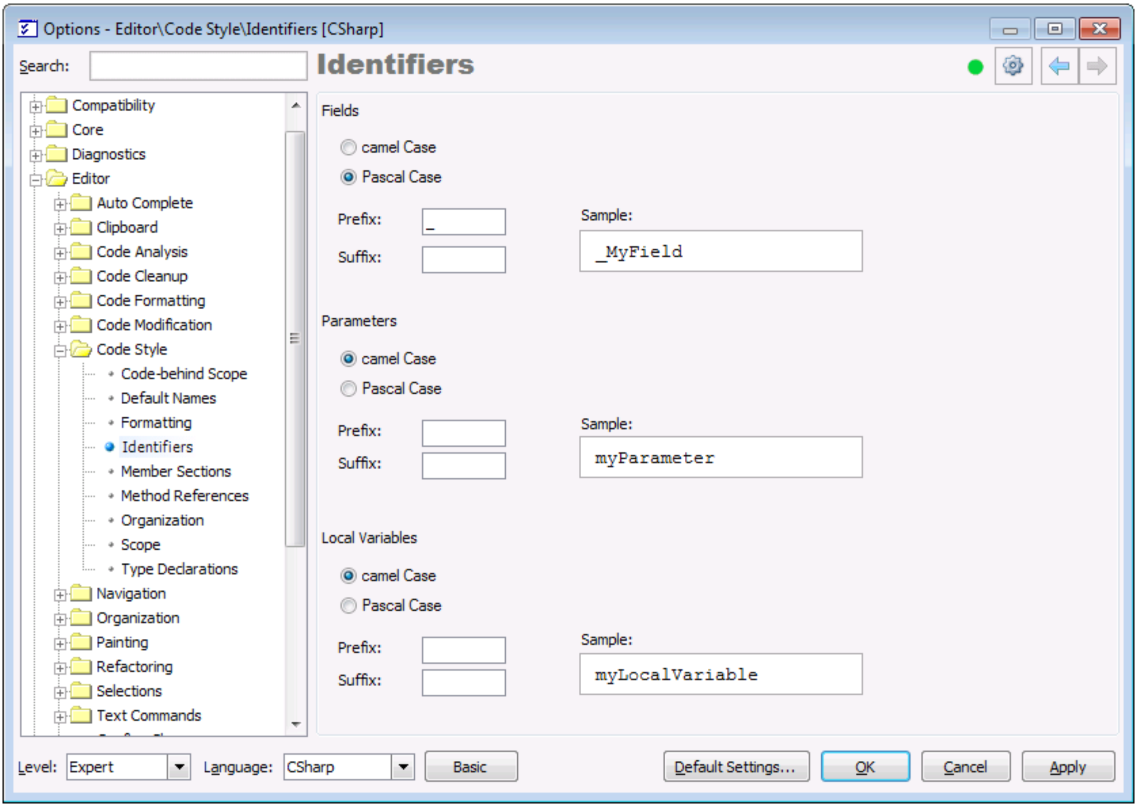
Synchronization complete.

Changing global identifiers style in CodeRush

Identifiers are names of various program elements in the code that uniquely identify a code element like namespace, class, interface, method, variable and others.

There are numerous identifiers style conventions which include usge of the Pascal or camel casing, use of underscore as a prefix, etc. That is why CodeRush allows you to configure the identifier style that will be globally applied for all features that create or generate new code: refactorings, code providers, code templates and others.

The options page is named **Identifier** and is located in the Editor | Code Style category in the Options Dialog:



Options - Editor\Code Style\Identifiers [CSharp]

Identifiers

Fields

○ camel Case
● Pascal Case

Prefix: _
Suffix:
Sample: _MyField

Parameters

● camel Case
○ Pascal Case

Prefix:
Suffix:
Sample: myParameter

Local Variables

● camel Case
○ Pascal Case

Prefix:
Suffix:
Sample: myLocalVariable

Level: Expert
Language: CSharp

The page allows you to tweak options for the following elements:

- Fields
- Parameters
- Local Variables

The following options can be changed:

- Casing: Pascal or camel
- Prefix of an identifier name
- Suffix of an identifier name

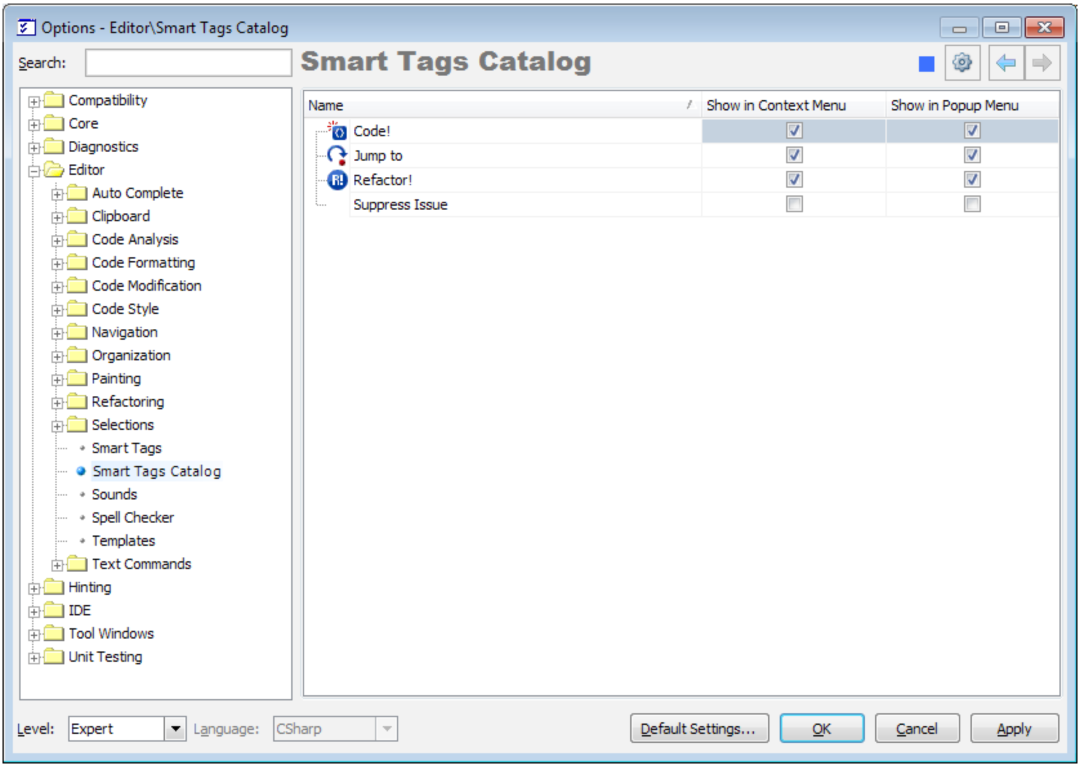
Once something is modified, you can see the preview of your change in the corresponding Sample box.

This options page is language dependent. You can change the identifier style for every language supported by CodeRush.

Smart Tags Catalog options page

The **Smart Tags Catalog** options page shows registered Smart Tags providers and allows you to control their appearance in the code editor context menu and CodeRush Popup menu.

The options page is simple. This is what it looks like:



Options - Editor\Smart Tags Catalog

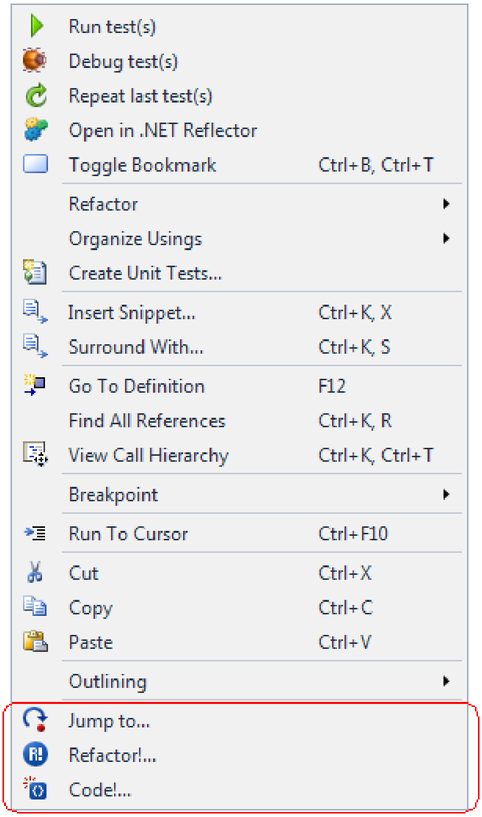
Smart Tags Catalog

Name	Show in Context Menu	Show in Popup Menu
Code!	☑	☑
Jump to	☑	☑
Refactor!	☑	☑
Suppress Issue	☐	☐

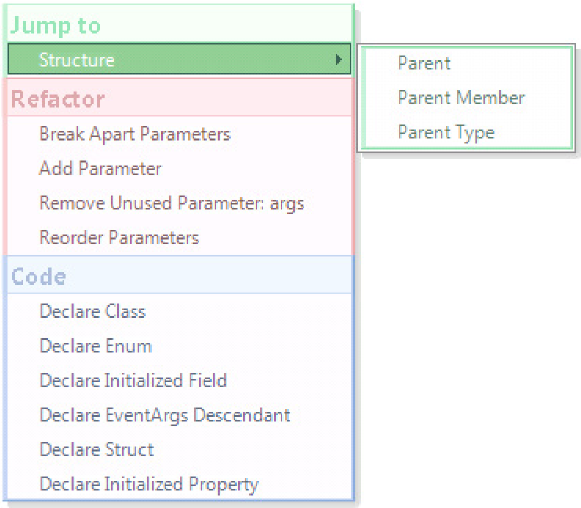
Level: Expert

Language: CSharp

Toggle the "Show in Context Menu" option to add or remove specific smart tag providers in the context menu of the code editor:



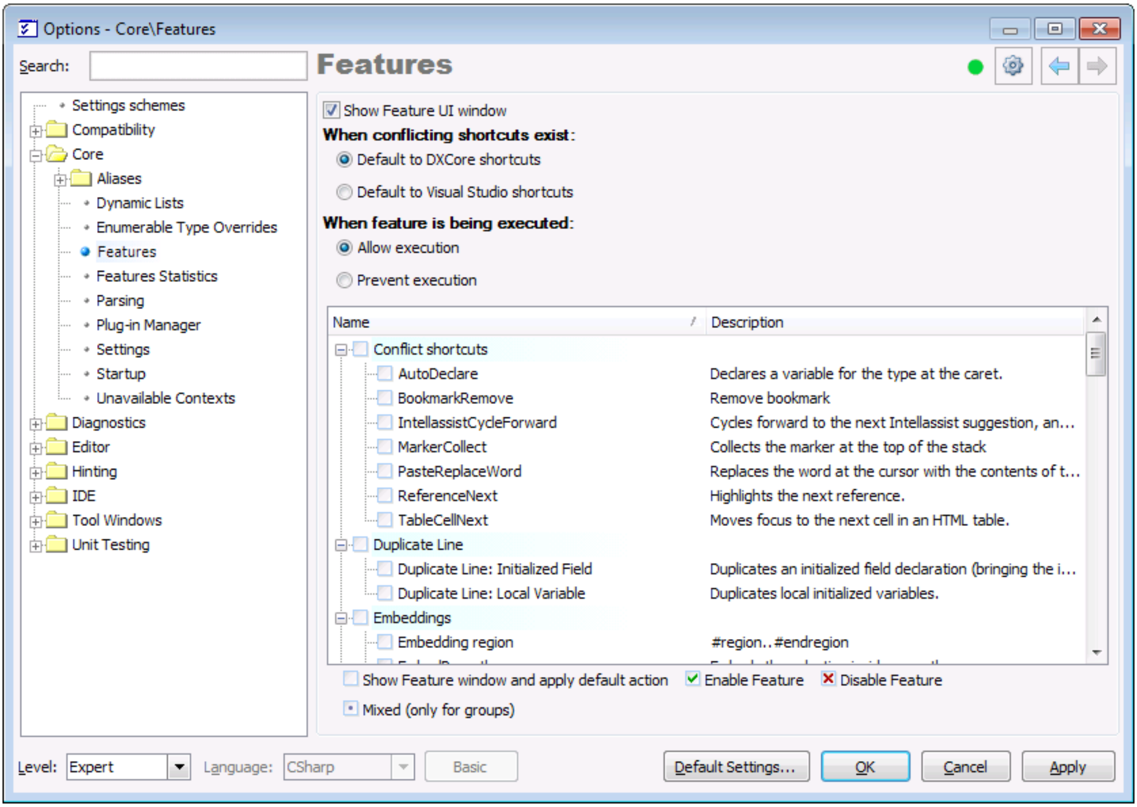
Toggle the "Show in Popup Menu" option to add or remove specific smart tag providers in the CodeRush/Refactor! popup menu:



Features options page

The **Features** options page in the Core category of the Options Dialog provides access to the **Feature UI** engine. The engine controls the feature execution, shows the What Happened hint when a particular CodeRush feature is executed, and allows you to turn it off if you don't need it. The What Happened hint is also shown when a conflict shortcut exists with Visual Studio IDE. In this case you can choose which feature should be performed by default whether it is Visual Studio one or CodeRush one.

This is what the Features options page looks like:



☑ Show Feature UI window

When conflicting shortcuts exist:

● Default to DXCore shortcuts

○ Default to Visual Studio shortcuts

When feature is being executed:

● Allow execution

○ Prevent execution

Name	Description
□ Conflict shortcuts	
□ AutoDeclare	Declares a variable for the type at the caret.
□ BookmarkRemove	Remove bookmark
□ IntellassistCycleForward	Cycles forward to the next Intellassist suggestion, an...
□ MarkerCollect	Collects the marker at the top of the stack
□ PasteReplaceWord	Replaces the word at the cursor with the contents of t...
□ ReferenceNext	Highlights the next reference.
□ TableCellNext	Moves focus to the next cell in an HTML table.
□ Duplicate Line	
□ Duplicate Line: Initialized Field	Duplicates an initialized field declaration (bringing the i...
□ Duplicate Line: Local Variable	Duplicates local initialized variables.
□ Embeddings	
□ Embedding region	#region..#endregion

□ Show Feature window and apply default action ✓ Enable Feature ☒ Disable Feature

▪ Mixed (only for groups)

Level: Expert Language: CSharp

The **Show Feature UI window** option controls the availability of the What Happened hint. If it is unchecked, you will never see the hint; otherwise, the hint is shown every time the CodeRush feature is executed.

The **When conflicting shortcuts exist** option allows you to choose the default preferred shortcut when CodeRush shortcut conflicts with Visual Studio. The chosen conflicting shortcut (CodeRush one or Visual Studio one) will be executed by default without further notification.

The **When feature is being executed** option allows you to specify the default action for the CodeRush feature: whether it should be allowed or suppressed. If you choose "allow", the feature will be executed as expected; otherwise, the feature will be suppressed and the What Happened hint will appear (if enabled).

After all options there is a tree list of features executed previously. It is populated every time a new CodeRush feature is executed. The list shows the feature name, its description and state. All features are grouped by categories.

The state of a feature can be of four values:

- Enable feature. The feature is enabled without restriction. No What Happened window appears for this feature in this state.

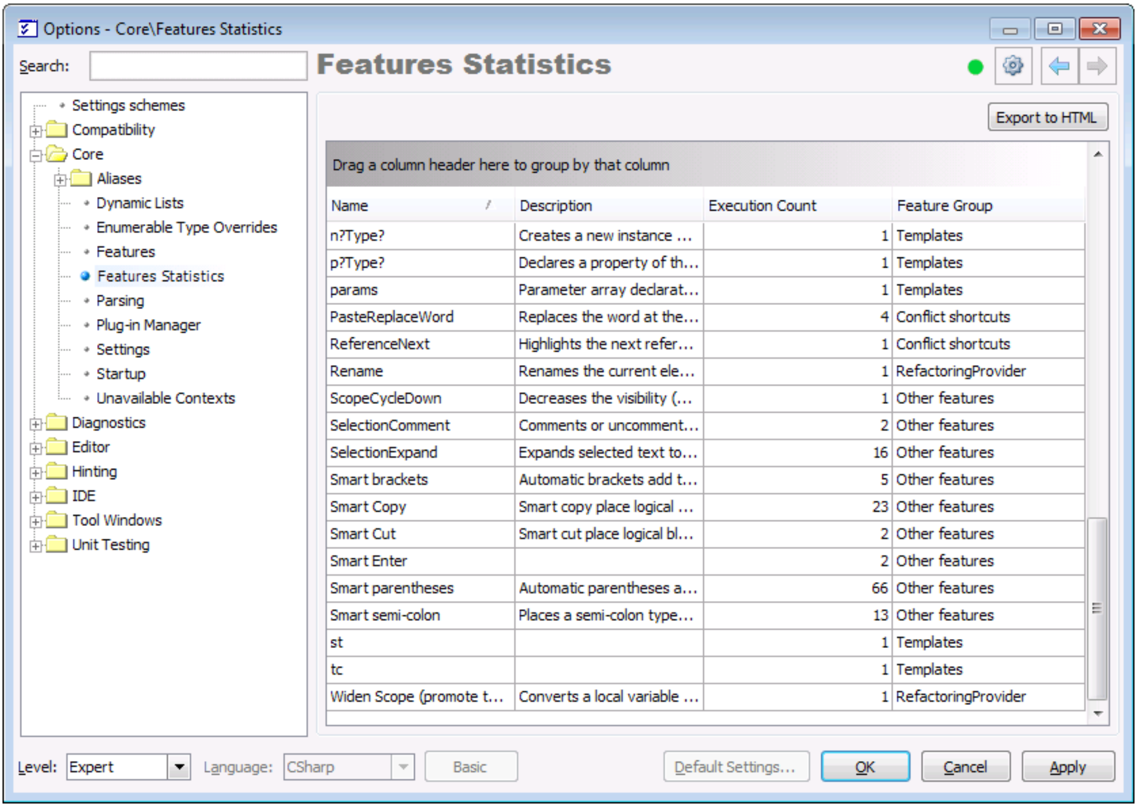
- Disable feature. The feature is disabled (suppressed), no execution allowed. No What Happened window appears for this feature in this state.

- Mixed (only for groups). This state is shown for groups when they have features of both states: enabled and disabled.

- Show Feature window and apply default action. The feature is registered and not suppressed. The What Happened hint will appear for this feature (if enabled).

Features Statistics options page

The **Features Statistics** options page is the part of the Features UI engine. It shows the execution count of a particular feature. The options page is located in the Core category of the Options Dialog:



Name	Description	Execution Count	Feature Group
n?Type?	Creates a new instance ...	1	Templates
p?Type?	Declares a property of th...	1	Templates
params	Parameter array declarat...	1	Templates
PasteReplaceWord	Replaces the word at the...	4	Conflict shortcuts
ReferenceNext	Highlights the next refer...	1	Conflict shortcuts
Rename	Renames the current ele...	1	RefactoringProvider
ScopeCycleDown	Decreases the visibility (...	1	Other features
SelectionComment	Comments or uncomment...	2	Other features
SelectionExpand	Expands selected text to...	16	Other features
Smart brackets	Automatic brackets add t...	5	Other features
Smart Copy	Smart copy place logical ...	23	Other features
Smart Cut	Smart cut place logical bl...	2	Other features
Smart Enter		2	Other features
Smart parentheses	Automatic parentheses a...	66	Other features
Smart semi-colon	Places a semi-colon type...	13	Other features
st		1	Templates
tc		1	Templates
Widen Scope (promote t...	Converts a local variable ...	1	RefactoringProvider

Level: Expert
Language: CSharp

The table of features contains four columns:

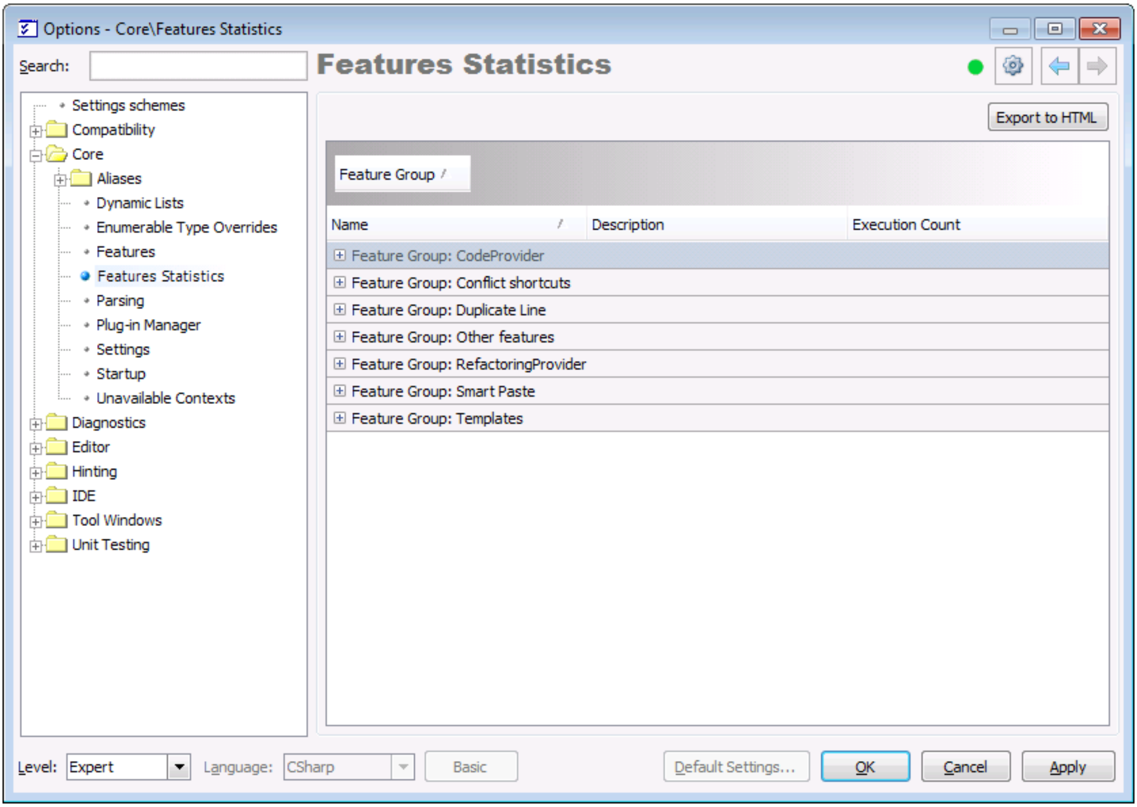
1. Name. The name of the feature.

2. Description. The description of the feature.

3. Execution Count. The number of times the feature has been executed.

4. Feature Group. The category name of the feature.

The list of features can be grouped by any column. It is recommended that you group the list by the Feature Group column:

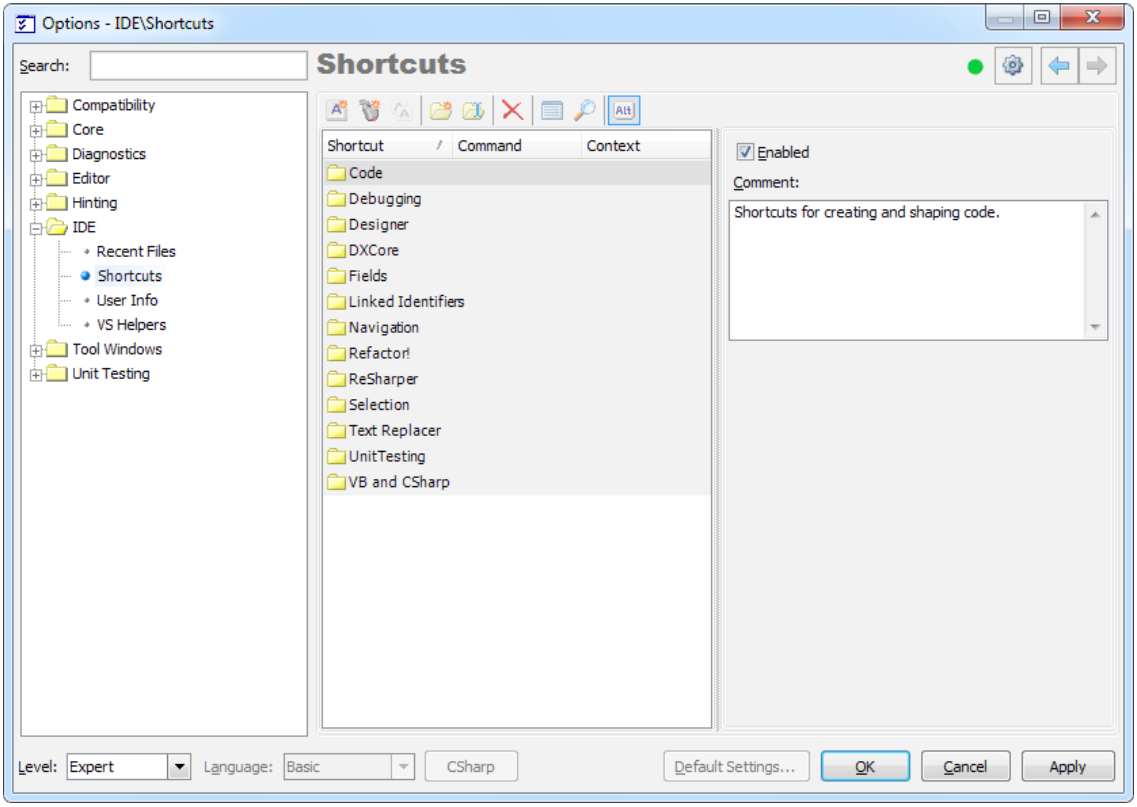


Name	Description	Execution Count
⊞ Feature Group: CodeProvider		
⊞ Feature Group: Conflict shortcuts		
⊞ Feature Group: Duplicate Line		
⊞ Feature Group: Other features		
⊞ Feature Group: RefactoringProvider		
⊞ Feature Group: Smart Paste		
⊞ Feature Group: Templates		

The statistics list can be exported into the HTML file format by clicking the Export to HTML button.

Shortcuts options page

The **Shortcuts** options page is an important and one of the most often used options pages. It allows you to create, modify and remove shortcut bindings for all IDE Tools products (CodeRush, DXCore, Refactor!). The page is located at the IDE | Shortcuts path in the CodeRush Options Dialog. Here is what it looks like:



Options - IDE\Shortcuts

Search:

Shortcuts

- Compatibility
- Core
- Diagnostics
- Editor
- Hinting
- IDE
  - Recent Files
  - Shortcuts
  - User Info
  - VS Helpers
- Tool Windows
- Unit Testing

Shortcut	Command	Context
Code		
Debugging		
Designer		
DXCore		
Fields		
Linked Identifiers		
Navigation		
Refactor!		
ReSharper		
Selection		
Text Replacer		
UnitTesting		
VB and CSharp		

☑ Enabled

Comment: Shortcuts for creating and shaping code.

Level: Expert

Language: Basic

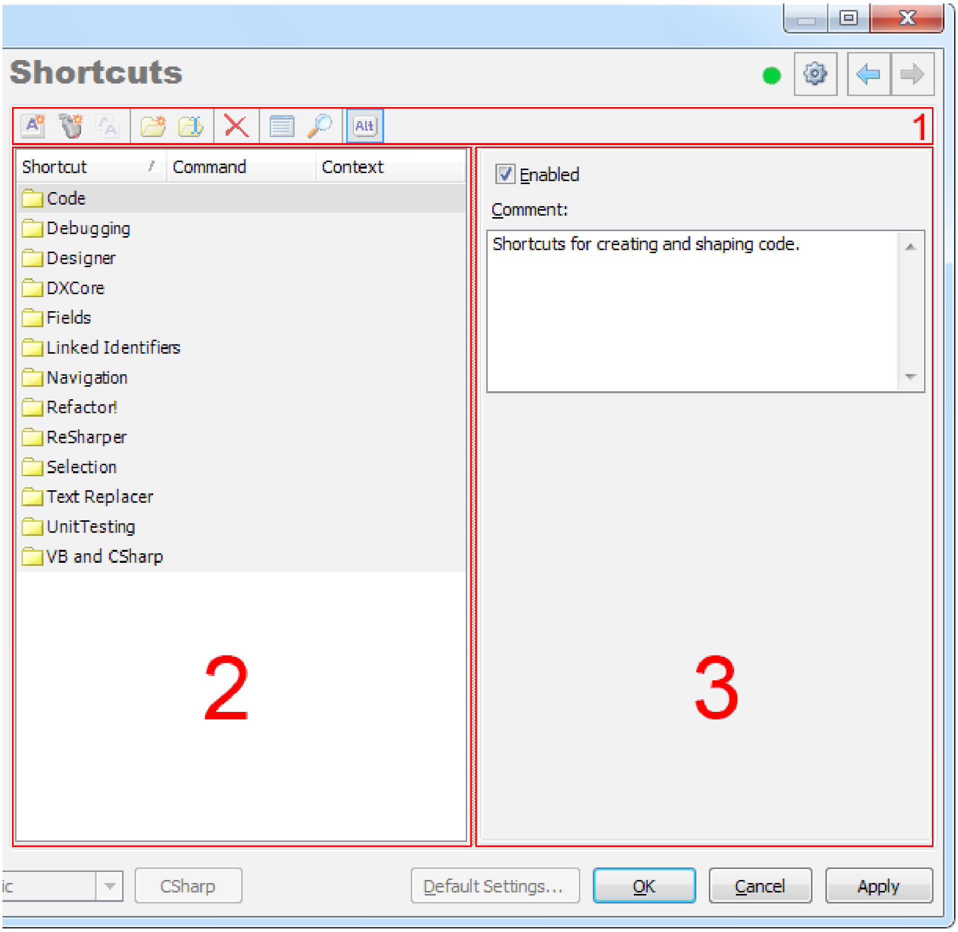
Shortcuts can be bound to keyboard strokes or mouse buttons. When creating a mouse shortcut you can use the left, right, middle, wheel and browse buttons in combination with shift key (*Ctrl, Alt, Shift*). Mouse shortcuts are available only when the code editor has focus. Keyboard shortcuts, on the other hand, are available anywhere in the Visual Studio IDE.

There are three main parts of the options page:

1. A toolbar with a several useful buttons at the top

2. The list of existing shortcuts with a couple of information columns on the left

3. A panel with available options on the right when an item in the list is selected



Shortcuts for creating and shaping code.

Let’s take a closer look at each part.

Shortcuts List (#2)

All shortcuts are grouped into the folders in the shortcuts list. Every folder may specify an area for which shortcuts are defined, e.g. *Navigation, Selection, Unit Testing*, and others. Some folders specify the product with dedicated shortcuts, e.g. *DXCore, Refactor!*. You can create your own folders, and modify (rename) or remove existing ones.

A folder is expanded by a single mouse click on its icon or a double clicking its name. Once expanded, you can see the sub-folders and shortcuts that are defined in this folder. The sub-folders may contain other sub-folders and additional shortcuts. Such structure is created to better organize numerous IDE Tools shortcuts.

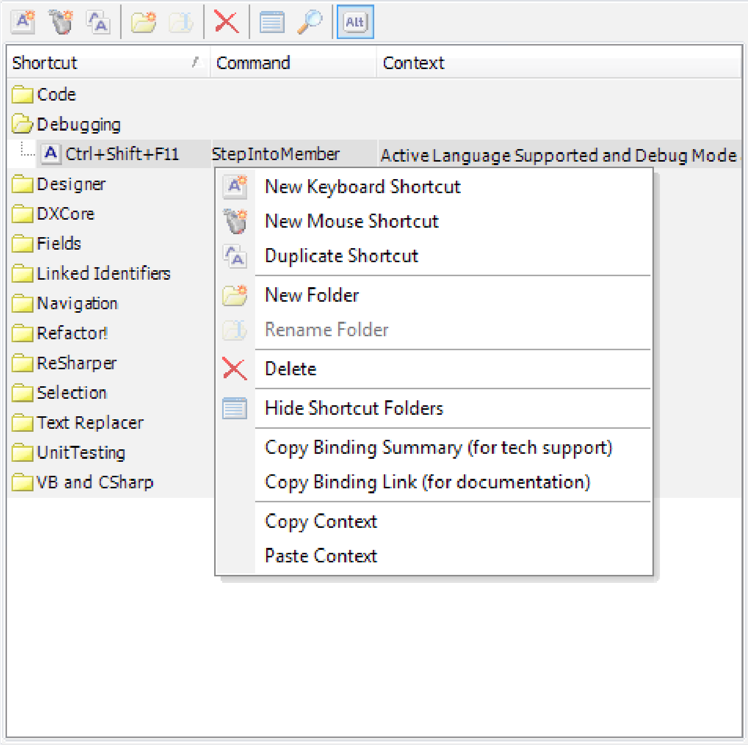
A shortcut item in the shortcuts list on the left has three columns:

1. *Shortcut* – a key or mouse binding that will execute the specified command once pressed or clicked. Note that you can have several shortcuts with the same key binding that will perform different or identical commands.

2. *Command* - the name of a command (an action name) that is being performed once the key binding is executed. Note that you can have several shortcuts bound to the same command name with different or identical key bindings.

3. *Context* – the context in which the key binding is valid and available. To learn more about contexts, please read an appropriate topic – Contexts Overview. Note that you can have several shortcuts bound to the same key binding or command name but perform differently, depending on the context set.

The shortcuts list has a context menu that replicates most of the toolbar buttons and adds a few others:



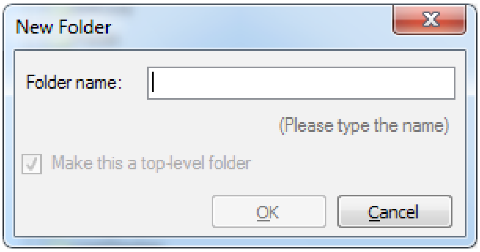
Shortcut	Command	Context
Code		
Debugging		
Ctrl+Shift+F11	StepIntoMember	Active Language Supported and Debug Mode
Designer		
DXCore		
Fields		
Linked Identifiers		
Navigation		
Refactor!		
ReSharper		
Selection		
Text Replacer		
UnitTesting		
VB and CSharp		

Here is an overview of the context menu items:

Icon	Has toolbar button	Item name	Description
	✓	New Keyboard Shortcut	Adds a new keyboard shortcut to the current shortcuts folder.
	✓	New Mouse Shortcut	Adds a new mouse shortcut to the current shortcuts folder.
	✓	Duplicate shortcut	Creates a copy of the selected shortcut. Note that this menu item is disabled if there is no shortcut selected.
	✓	New Folder	Opens the *New Folder* dialog where you can specify the name of a new folder and its location or cancel folder creation. The dialog will not let you create a folder when the folder with the same name already exists.

📁	✓	Rename Folder	Opens the *Rename Folder* dialog where you can specify a new name for the current folder. If no folder is selected, the corresponding menu item is disabled. The dialog will not let you specify the name of a folder when the folder with the same name already exists.
✗	✓	Delete	Removes the current shortcut or an entire folder with all shortcuts inside, depending on the selected item (whether it is a folder or a shortcut).
▤	✓	Hide Shortcut Folders	Hides all folders and shows all shortcuts in a plain list.
	✗	Show Shortcut Folders	Restores the folder structure once the folders were hidden. This menu item is hidden if folders were not hidden.
	✗	Collapse Folders	Collapses all folders to their definition, so its shortcuts are not visible. Note that this menu items is only available when you click a folder and not a shortcut.
	✗	Copy Binding Summary (for tech support)	Generates the most important information about the selected shortcut and copies it to the Clipboard. The information includes the following data: binding type (keyboard or mouse), the binding itself, command name, command parameters, comments, context, etc.
	✗	Copy Binding Link (for documentation)	Generates a command name tag for documenting it in the User Guideand copies it into the Clipboard.
	✗	Copy Context	Copies the context of the selected shortcut, so it is easier to initialize the context for new or similar shortcuts.
	✗	Paste Context	Initializes the context of the selected shortcut to the previously copied one. This is a handy function, allowing you to specify the context for a new shortcut based on the context of others.

Here are what the *New Folder* and the *Rename Folder* dialogs mentioned in the table look like:



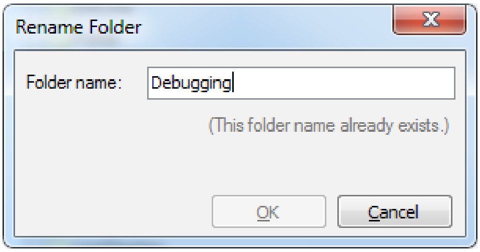
Folder name:

(Please type the name)

☑ Make this a top-level folder

(Note: If the "Make this a top-level folder" checkbox is unchecked, the folder will be created as a sub-folder of the currently selected folder; otherwise, it will be a root folder)

and



Rename Folder

Folder name: Debugging

(This folder name already exists.)

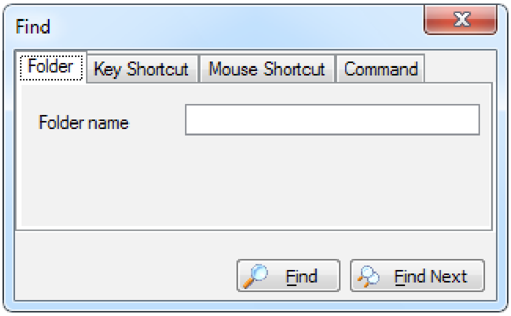
Toolbar (#1)

As you can see, almost all of these functions are also available on the toolbar located at the top of the page. However, the toolbar has two additional buttons that do not exist in the context menu:

- 🔎 - Find

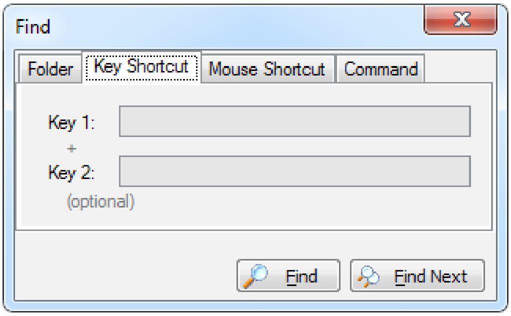
Opens the *Find* dialog which allows you to find an existing shortcut or a folder. The dialog has four tabs where you can specify different parameters for the dialog:

1) Folder:



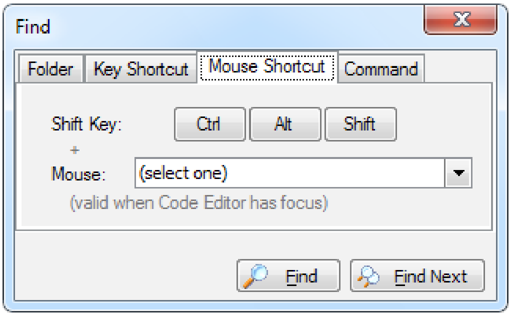
Here you can specify a folder name.

2) Key Shortcut:



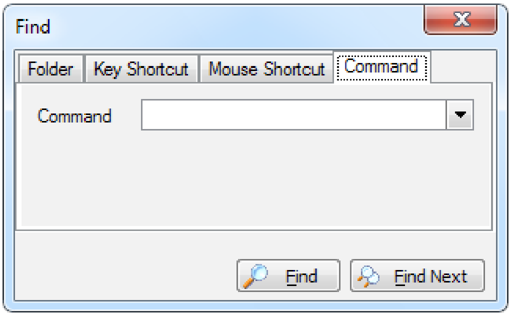
Here you can specify a key binding for a shortcut.

3) Mouse Shortcut:



Here you can specify a mouse button and a shift key for the shortcut.

4) Command:



Specify a command name or choose it from the list, to find the corresponding shortcut if one exists.

The dialog has two buttons at the bottom: *Find* and *Find Next*. The *Find* button searches for the first occurrence of a shortcut, and the *Find Next* button allows you to search for other shortcuts with the same parameters.

- Alt - Distinguish between left and right "Alt" (only for keyboard shortcuts)

Toggles an option to differentiate between the Alt (left Alt) and Alt Gr (right Alt) buttons. If enabled, both Alt keys will be treated as an Alt key, otherwise, the two Alt keys will be treated as different buttons. According to the caption of this option, it works only for keyboard shortcuts.

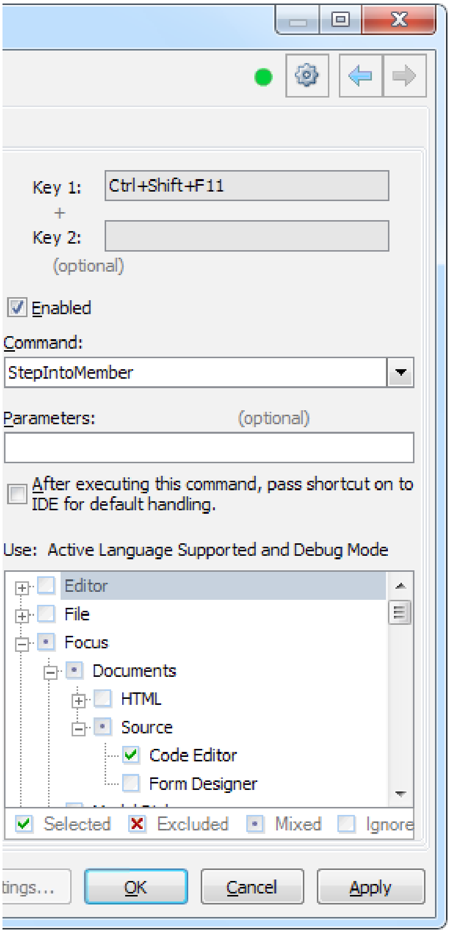
Shortcuts Settings (#3)

Now let's see the options panel on the right side of the **Shortcuts** options page. If you have a folder selected, there are two options available (*see the screenshot at the top of this article)*:

1. *Enabled*, which specifies the availability of all shortcuts in the folder

2. *Comment*, where you can specify a comment for a folder

If you have a keyboard shortcut selected in the shortcuts list, the options panel will look similar to this:



Key 1: Ctrl+Shift+F11
+
Key 2:
(optional)

☑ Enabled

Command: StepIntoMember

Parameters: (optional)

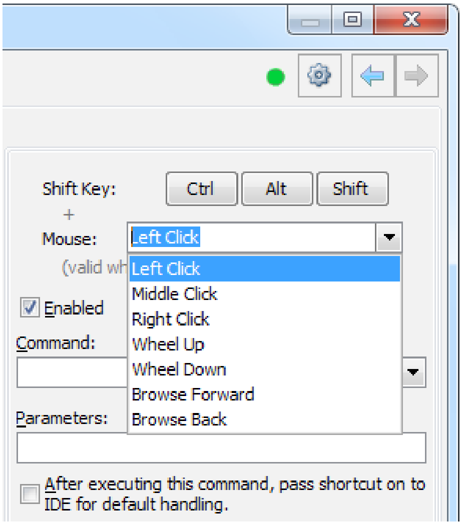
□ After executing this command, pass shortcut on to IDE for default handling.

Use: Active Language Supported and Debug Mode

- □ Editor
- □ File
- ▣ Focus
  - ▣ Documents
    - □ HTML
    - ▣ Source
      - ☑ Code Editor
      - □ Form Designer

On the top you can assign a key binding for a shortcut in the Key1 and/or Key2 fields. Press the keyboard sequence in the Key1 field to assign a key binding. The Key2 field allows you to specify a second part of a key binding. For example, you can specify the “Ctrl+T” keystroke as a Key1 and the “R” key as a Key2 – this will create a “Ctrl+T,R” shortcut when you will have to press the Ctrl+T first and then the R key immediately to execute a command.

For the Mouse shortcuts the panel fields will look a bit different. For such shortcuts, you can select a Shift Key and the mouse button from the list:



Shift Key:

Mouse: Left Click

☑ Enabled

Command:

Parameters:

□ After executing this command, pass shortcut on to IDE for default handling.

The *Enabled* option specifies whether the shortcut is available in the Visual Studio IDE. If it is not enabled, then the key binding will not do anything.

The *Command* combobox allows you to specify or choose a command name for a shortcut. Choose a command (an action) name and specify its parameters in the next field if required. Parameters are useful for commands that require them. For example, you can specify a refactoring name as a parameter to the Refactor command. This will apply the specified refactoring once the key binding is performed.

Next, the "*After executing this command, pass shortcut on to IDE for default handling*" option. This option specifies whether or not to send the performed key binding to the Visual Studio IDE. If unchecked, the key binding will perform a command and will not send a key to the IDE, so IDE will not intercept the key press and will not perform any actions bound to the same key binding.

Note, that if you assign both keys for a shortcut (in Key1 and Key2 areas) the "*After executing this command, pass shortcut on to IDE for default handling*" option will be checked and you will not be able to change it (it becomes disabled). This is, probably, a limitation of the current product version: you can not create a double-key shortcut and not have it passed on the command to the Visual Studio IDE. This is how the *Shortcuts Engine* is designed – if the Engine does not pass a second key to the IDE and perform an action bound to a shortcut, then Visual Studio IDE will not get a second keystroke and it will stay in the "*Waiting for second key of chord…*" mode until you press something. That is why the check box "*After executing this command, pass shortcut on to IDE for default handling*" is disabled and ticked if you specify a second key.

After all options for a shortcut, there is a context picker that allows you to specify a context for a particular shortcut.

When a shortcut is created and executed, and it has a conflict with the Visual Studio IDE shortcuts, you will see the Feature UI dialog that allows you to choose the default action to be performed in the future: whether the **CodeRush** shortcut or a Visual Studio one will be performed by default.

You can read the following shortcuts-related articles, to learn more on how to create shortcuts:

How to assign a shortcut key to a particular action

How to assign a separate shortcut to a particular refactoring

How to assign a shortcut key to a particular action

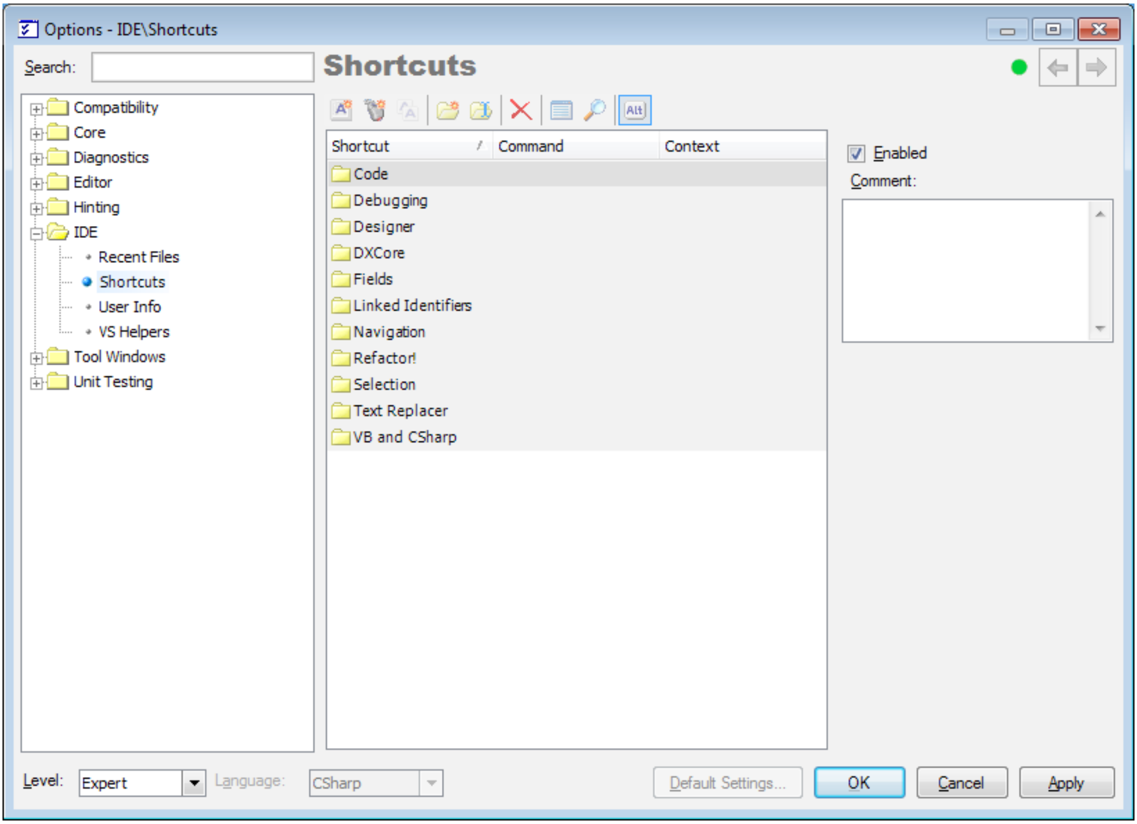
It can be easily accomplished from the "Shortcuts" options page in the Options Dialog. Follow these steps to get to the Shortcuts options page:

1. From the DevExpress menu, select Options…

2. In the tree view on the left, navigate to this folder: IDE.

3. Select the Shortcuts options page.

Here it is:



Options - IDE\Shortcuts

Search:

Shortcuts

☑ Enabled

Comment:

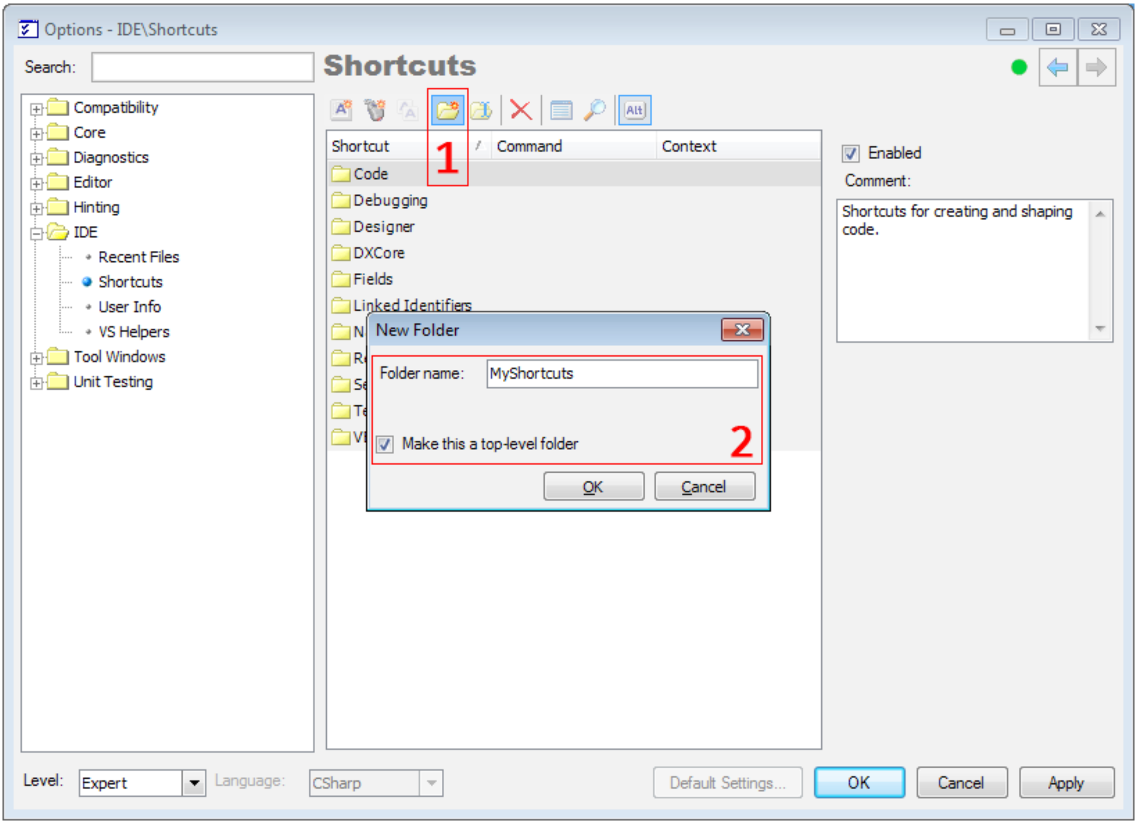
Level: Expert

Language: CSharp

To create a shortcut:

1. Create a new folder by clicking on the New Folder button.

2. Enter a folder name, e.g. "MyShortcuts" and check the "Make this a top-level folder" check box:



Search:

New Folder

Folder name: MyShortcuts

☑ Make this a top-level folder

☑ Enabled

Comment: Shortcuts for creating and shaping code.

Level: Expert

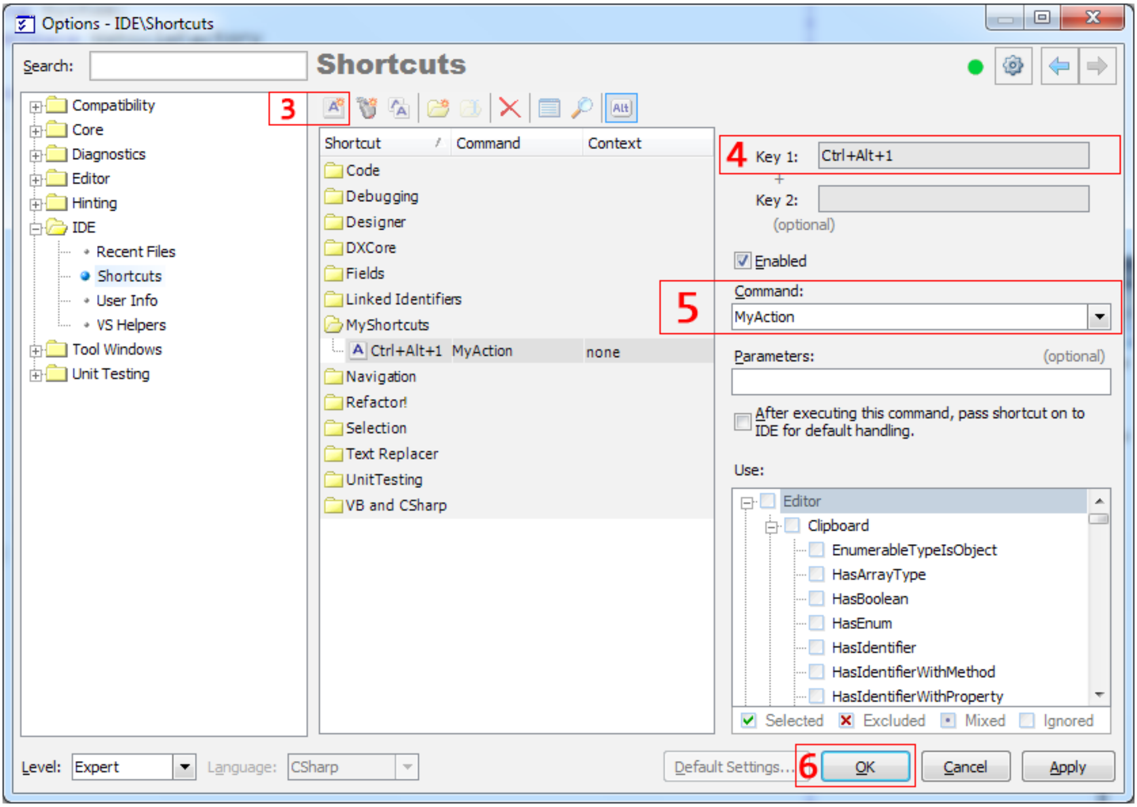
Language: CSharp

3. Click the “New Keyboard Shortcut” button.

4. Enter a key for this shortcut into the “Key 1” text box (and into the “Key 2” text box if necessary).

5. Select an action from the Command combo box, e.g. “MyAction”.

6. Click OK to save your settings.



Shortcut	Command	Context
Code		
Debugging		
Designer		
DXCore		
Fields		
Linked Identifiers		
MyShortcuts		
Ctrl+Alt+1	MyAction	none
Navigation		
Refactor!		
Selection		
Text Replacer		
UnitTesting		
VB and CSharp		

Key 1: Ctrl+Alt+1

Key 2: (optional)

☑ Enabled

Command: MyAction

Parameters: (optional)

□ After executing this command, pass shortcut on to IDE for default handling.

Level: Expert Language: CSharp

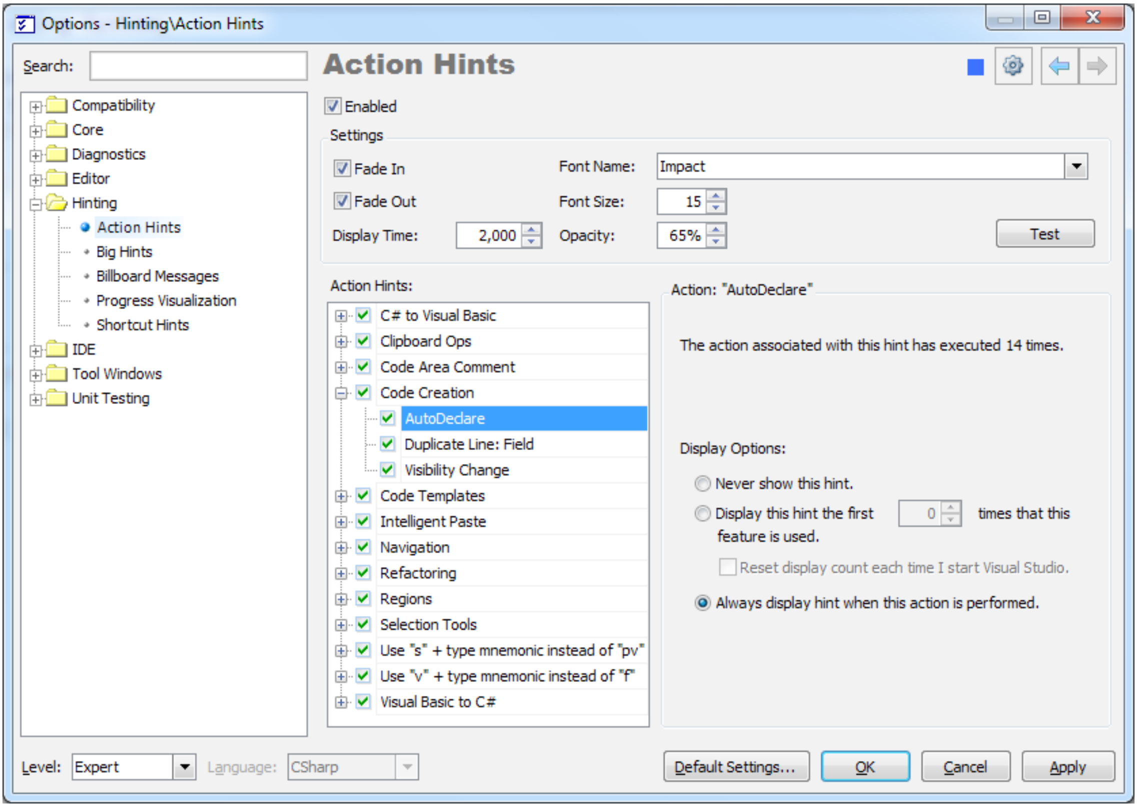
The shortcut has been created. Now the MyAction will be executed after pressing the shortcut key (CTRL+ALT+1 in this example).

Hinting\Action Hints option page

The **Action Hints** options page manages settings for DXCore action hints. The following **hint options** are available:

Option name	Description
Enabled	Globally enables or disables action hints availability.
Fade In	Determines whether or not an action hint will fade in (from transparent to the final Opacity setting) when it first appears. If unchecked, action hints will appear immediately.
Fade Out	Determines whether or not the action hint will fade out (from its Opacity setting to completely transparent) when it closes. If unchecked, action hints will disappear immediately.
Display Time	Determines how long (in milliseconds) an action hint will appear before closing.
Font Name	The font name for the text of action hints.
Font Size	The font size for the text of action hints.
Opacity	The opacity of action hints.

The “**Test**” button is used for action hints appearance testing.



Options - Hinting\Action Hints

Search:

- Compatibility
- Core
- Diagnostics
- Editor
- Hinting
  - Action Hints
  - Big Hints
  - Billboard Messages
  - Progress Visualization
  - Shortcut Hints
- IDE
- Tool Windows
- Unit Testing

Action Hints

☑ Enabled

Settings

☑ Fade In

☑ Fade Out

Display Time: 2,000

Font Name: Impact

Font Size: 15

Opacity: 65%

Action Hints:

- ☑ C# to Visual Basic
- ☑ Clipboard Ops
- ☑ Code Area Comment
- ☑ Code Creation
  - ☑ AutoDeclare
  - ☑ Duplicate Line: Field
  - ☑ Visibility Change
- ☑ Code Templates
- ☑ Intelligent Paste
- ☑ Navigation
- ☑ Refactoring
- ☑ Regions
- ☑ Selection Tools
- ☑ Use "s" + type mnemonic instead of "pv"
- ☑ Use "v" + type mnemonic instead of "f"
- ☑ Visual Basic to C#

Action: "AutoDeclare"

The action associated with this hint has executed 14 times.

Display Options:

○ Never show this hint.

○ Display this hint the first 0 times that this feature is used.

□ Reset display count each time I start Visual Studio.

● Always display hint when this action is performed.

Level: Expert

Language: CSharp

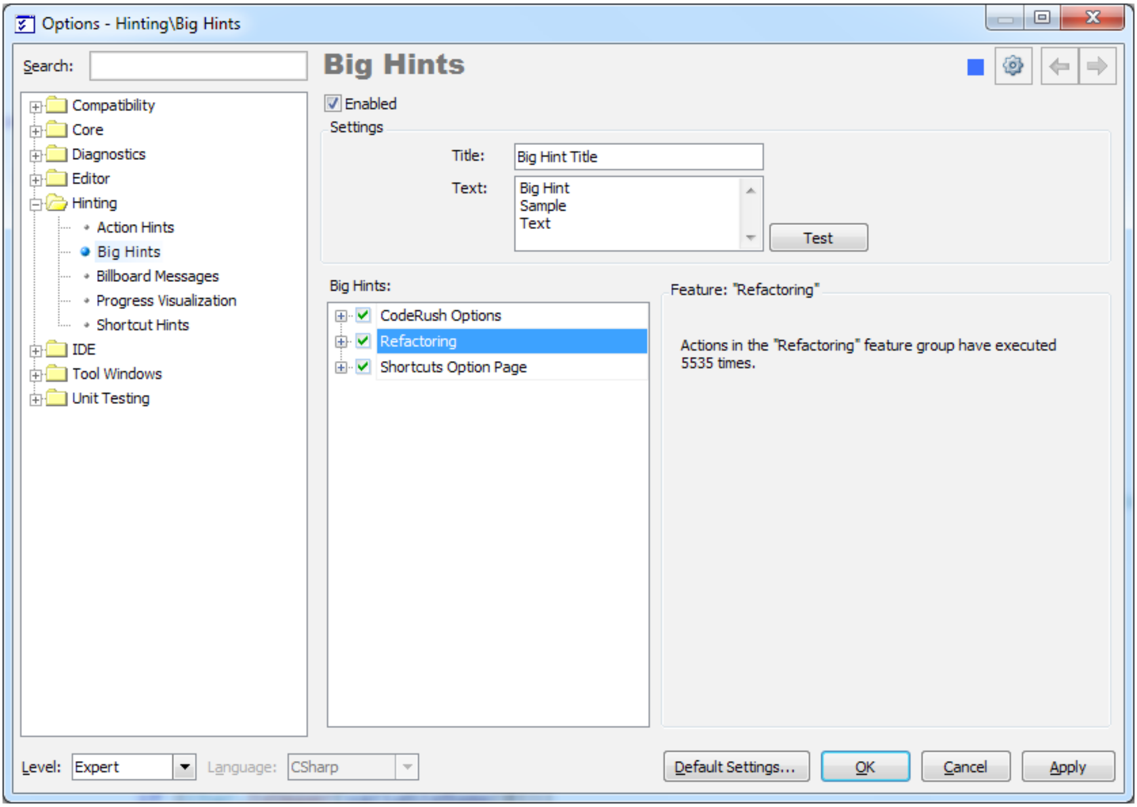
Under the global settings section there is a list of features which have been performed. You can specify **additional display options** for the action hints associated with these features:

Option name	Description
Never show this hint	The action hint will never be displayed for the selected feature.
Display this hint the first ### times that this feature is used	The number of times the action hint for the selected feature should be displayed before it is suppressed. If you want your action hint to be displayed this many times every IDE session, then check the *"Reset display count each time I start Visual Studio"* option.
Reset display count each time I start Visual Studio	Resets the display count for the selected feature each time DX-Core/CodeRush loads.
Always display hint when this action is performed	The action hint will be displayed each time the selected feature is executed.

In addition, there is **statistical info** shown for the selected feature – how many times this feature has been performed.

Hinting\Big Hints option page

This options page is similar to the Action Hints options page, but a bit simplified, however. The page manages settings for the DXCore big hints appearance, and allows you to test them. Also, here you can entirely disable the **big hints** from appearing.



☑ Enabled

Settings

Title: Big Hint Title

Text: Big Hint
Sample
Text

Big Hints:

- ☑ CodeRush Options
- ☑ Refactoring
- ☑ Shortcuts Option Page

Feature: "Refactoring"

Actions in the "Refactoring" feature group have executed 5535 times.

Level: Expert

Language: CSharp

Use the “**Test**” button to see what the **big hint** will look like with the specified title and text in the appropriate text boxes.

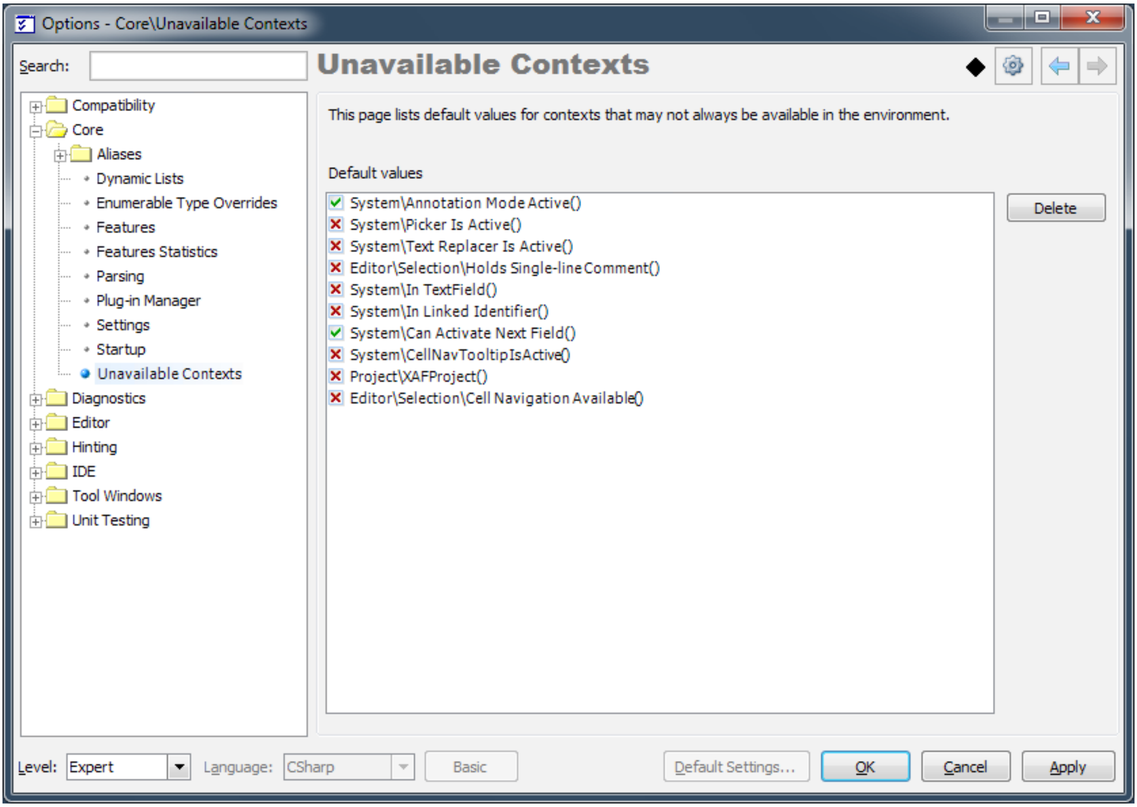
Under the global settings section (which is used for testing actually) there is a list of features which have been performed. You can specify **additional display options** for the action hints associated with these features:

Option name	**Description**
Never show this hint	The **big hint** will never be displayed for the selected feature.
Display this hint the first ### times that this feature is used	The number of times the **big hint** for the selected feature should be displayed before it is suppressed. If you want your **big hint** to be displayed this many times every IDE session, then check the “Reset display count each time I start Visual Studio” option.
Reset display count each time I start Visual Studio	Resets the display count for the selected feature each timeDXCore/CodeRush loads.
Always display hint when this action is performed	The big hint will be displayed each time the selected feature is executed.

In addition, there is **statistical info** shown for the selected feature – how many times this feature has been performed.

Unavailable Contexts option page

The **Unavailable Contexts** options page shows the list of DXCore contexts and its default values that may not always be available in IDE Tools:



Options - Core\Unavailable Contexts

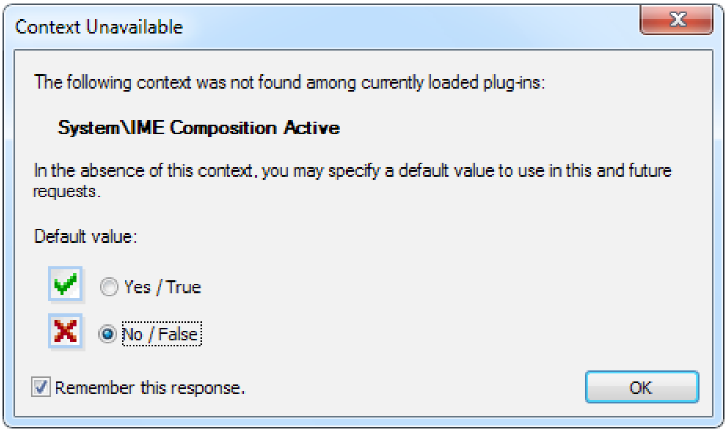
Unavailable Contexts

This page lists default values for contexts that may not always be available in the environment.

Default values

- ✓ System\Annotation Mode Active()
- ✗ System\Picker Is Active()
- ✗ System\Text Replacer Is Active()
- ✗ Editor\Selection\Holds Single-line Comment()
- ✗ System\In TextField()
- ✗ System\In Linked Identifier()
- ✓ System\Can Activate Next Field()
- ✗ System\CellNavTooltipIsActive()
- ✗ Project\XAFProject()
- ✗ Editor\Selection\Cell Navigation Available()

This could be the case when you have removed the plug-in containing a context, but a context is stored and used somewhere in the setting files (e.g. a keyboard shortcut or template). Once a context that does not actually exist is being checked, you will see the following dialog:



Context Unavailable

The following context was not found among currently loaded plug-ins:

System\IME Composition Active

In the absence of this context, you may specify a default value to use in this and future requests.

Default value:

✓ ○ Yes / True

✗ ● No / False

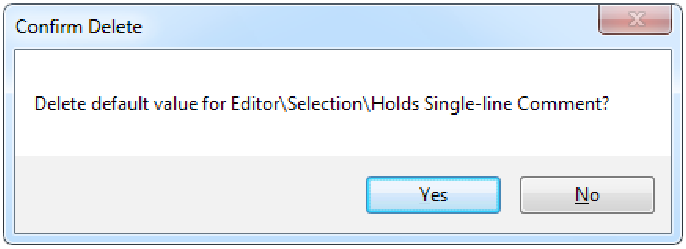
☑ Remember this response.

Choose a default value for the context whether it is satisfied or not satisfied by default. Your choice is saved on the **Unavailable Context** page inside the CodeRush Options Dialog, if you tick the "Remember this response" check-box.

An entry in the list on the options page has a small icon indicating your choice:

- (✓) satisfied when absent or
- (☒) not satisfied when absent

Click an icon to change the default value of a context. You can remove an entry from the list by clicking the *Delete* button and confirm the deletion:



Delete default value for Editor\Selection\Holds Single-line Comment?

If the removed context is being checked again and it still does not exist, the **Context Unavailable** dialog will appear again, asking you to specify a default value.