

Table of Contents

Introduction

Getting Started

Code Editor

Overview

Win Forms

Basic Features

Extended Features

WPF

Basic Features

Extended Features

Syntax Parsing

Advanced Topics

Scripter

Overview

C#/Visual Basic

Python/IronPython

TypeScript/JavaScript

Script Debugging

Debugger UI

Form Designer

Overview

Win Forms

WPF

Getting Started

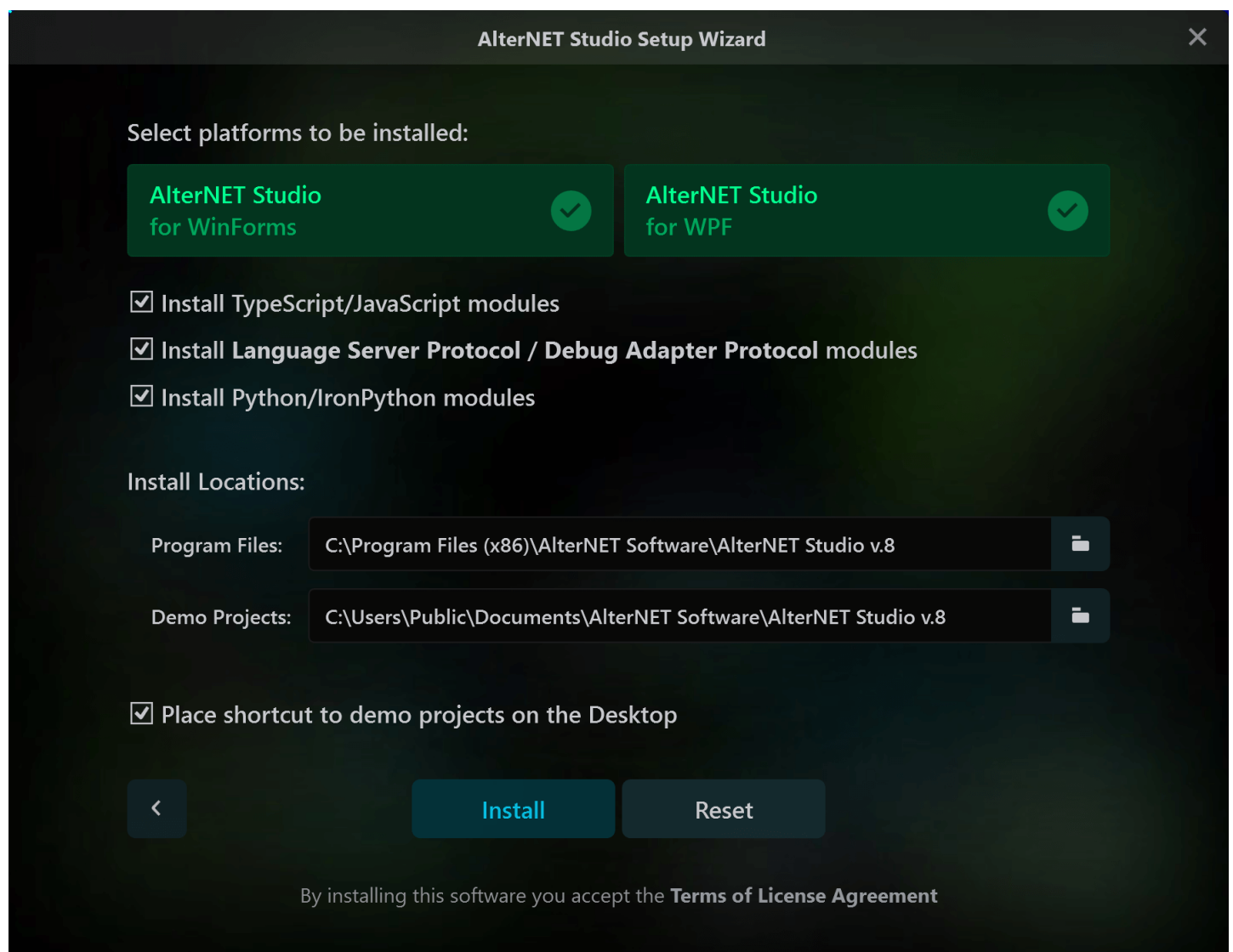
AlterNET Studio allows you to extend WinForms and WPF .NET applications with code editing, scripting and user interface designing capabilities.

Installation

AlterNET Studio requires .NET Framework 4.6.1+ and Visual Studio 2017, 2019 or 2022 to be installed on the target machine.

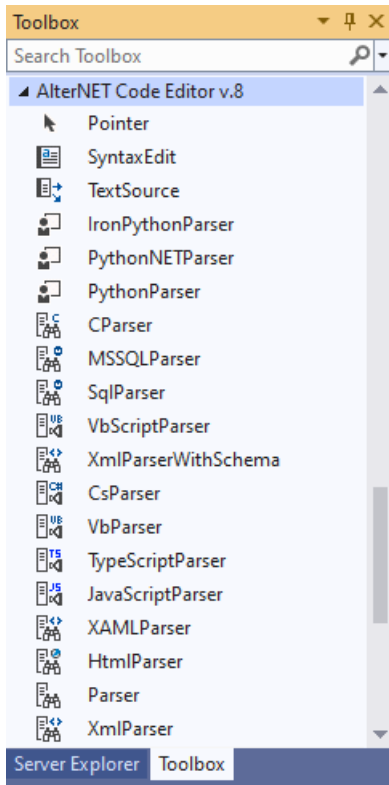
By default AlterNET Studio installation program installs AlterNET Studio binary files to Program Files (x86)\AlterNET Software\AlterNET Studio\Bin\ folder and example projects with source code in \Users\Public\Documents\AlterNET Software\AlterNET Studio folder. These settings can be changed if you select Customize in the installation wizard.

Advanced installation options include platform selection (WinForms, WPF or both), and inclusion of Python/IronPython, TypeScript/JavaScript and LangServer/DAP features.



Installation program registers Visual Studio extensions and places controls and components on the AlterNET Code Editor, AlterNET Scripter and AlterNET Form Designer tabs in the Visual Studio toolbox.

- [Code Editor](#)
- [Scripter](#)
- [Form Designer](#)



Other versions of .NET Framework 4.5.2, .NET Core 3.0, + Net 5.0 and .NET 6.0 are supported via NuGet packages. Complete list of NuGet packages can be found here:

<https://alternetsoft.com/download#nuget>

If you have a previous major version of AlterNET Studio, and decide to install the new one side-by-side, you will have two sets of Visual Studio Extensions and two sets of tabs, each one clearly displaying version number.

Demo and QuickStart projects

Once the product is installed, you can explore demos and quick start projects, either by compiling `Alternet.Studio.AllDemos` solutions or accessing these demos through the Demo Explorer tool which is added to Windows Start menu.

AlterNET Software Demo Explorer

Studio for C#/Visual Basic

WinForms

AlterNET Studio

Code Editor

Syntax Editor

Form Designer

QuickStart Projects

Editor

Scripter

Debugger

Designer

WPF

AlterNET Studio

Code Editor

Syntax Editor

Form Designer

QuickStart Projects

Studio for Python

WinForms

QuickStart Projects

Editor

Scripter

Debugger

WPF

Studio for IronPython

Studio for TypeScript

Studio for LSP/DAP

AlterNET Studio

This demo shows how all products can be integrated in one solution - from designing visual interface using Form Designer, to writing application logic in powerful Code Editor and executing and debugging this code with Scripter.

More Information:

AlterNET Studio allows you to extend your WinForms and WPF .NET application with code editing, scripting and user interface designing capabilities. It provides a set of component libraries that enables users of your applications to write scripts in C# and Visual Basic, programming languages to extend your application with custom functionality; run and debug these scripts with a fully-featured debugging engine and design visual user interfaces with graphical Form Designer.

Run

Open Project

AlterNET Studio 8.0

Copyright © 2016-2022 ALTERNET SOFTWARE PTY LTD

All rights reserved <http://www.alternetsoft.com>

Core Components

AlterNET Studio includes the following core components:

Code Editor

AlterNET Code Editor is a component library that brings efficient code editing functionality to the .NET applications. It provides code editing capabilities such as syntax highlighting, intellisense (code completion), code outlining, visual indicators for bookmarks, line styles, syntax errors and more.

AlterNET Code Editor matches most of the features of Visual Studio code editor and is specifically tailored for C#, Visual Basic, TypeScript, JavaScript, Python and XML code editing.

Scripter

AlterNET Scripter is a component library designed to integrate C#/Visual Basic, TypeScript/JavaScript and IronPython scripts into the .NET applications. It allows extending functionality of the application logic without recompiling and redeploying the application.

AlterNET Scripter provides a framework to compile and execute user-defined scripts along with the set of debugging tools enabling application developers to make application objects available to the scripts so they can write user-defined scenarios for these applications.

Form Designer

AlterNET Form Designer is a component library providing a quick and convenient way for creating visual user interfaces. It allows placing controls to the design surfaces, setting their initial properties and writing event handlers for their events.

AlterNET Form Designer includes WinForms and WPF designers, both supporting designing visual interfaces, serializing design content and running forms being designed.

Integrating AlterNET Studio components.

AlterNET Code Editor, Scripter and Form Designer can work together in the applications that require text editing, scripting or ui designing functionality. Code Editor is tailored for C#, Visual Basic, TypeScript, JavaScript, Python and XML code editing and can be used in conjunction with Form Designer and Scripter packages to provide code editing functionality for code-behind files, writing event handlers and script editing.

Refer to our AlterNET Studio demo project to see how [Code Editor](#), [Scripter](#) and [Form Designer](#) work together.

Licensing

AlterNET Studio requires a valid license to be installed for developing .NET applications that use its components. Evaluation-license is supplied upon AlterNET Studio installation and when consuming NuGet packages; these licenses are based on licx files technology provided by Microsoft and are valid for 30 days since first use.

Upon ordering a paid version of our product, a customer is sent a License key and will be able to activate it on the target computer. This key will support a number of activations but it is not transferable between development machines.

Oce you drag AlterNET Studio controls or components from the toolbox, such as [SyntaxEdit](#), [TextEditor](#), [FormDesignerControl](#), or [ScriptRun](#) on your form, the licx file will be added to your project under Properties folder with the content like this:

```
Alternet.Editor.SyntaxEdit, Alternet.Editor.v8, Version=8.0.0.0, Culture=neutral, PublicKeyToken=8032721e70924a63
```

```
Alternet.Scripter.ScriptRun, Alternet.Scripter.v8, Version=8.0.0.0, Culture=neutral, PublicKeyToken=8032721e70924a63
```

```
Alternet.FormDesigner.WinForms, Alternet.FormDesigner.v8, Version=8.0.0.0, Culture=neutral,  
PublicKeyToken=8032721e70924a63
```

In case you these components are created from the code, such a licx file with above content should be added to your project. It also can be copied from our demo projects if needed.

The design-time license is being checked when you work with this component at design-time, or when the project is compiled; and the nag screen reminding about the evaluation mode will appear once in a while. If the project compiled with an evaluation version of licensed components is run outside Microsoft Visual Studio debugger, a screen suggesting that the application was created with an evaluation version of AlterNET Studio will be displayed. Once a paid license is activated using the LicenseActivation tool, this nag screen will no longer appear.

When the evaluation period expires, you will still be able to compile and run your application from within Visual Studio, however applications created with expired license will not be run in standalone mode.

Below is more information about Microsoft license compiler and some discussions related to intent and purpose of licx files, using them with source code control systems, etc.

[https://msdn.microsoft.com/en-us/library/ha0k3c9f\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ha0k3c9f(v=vs.110).aspx)

<http://stackoverflow.com/questions/5628969/how-licenses-licx-file-is-used>

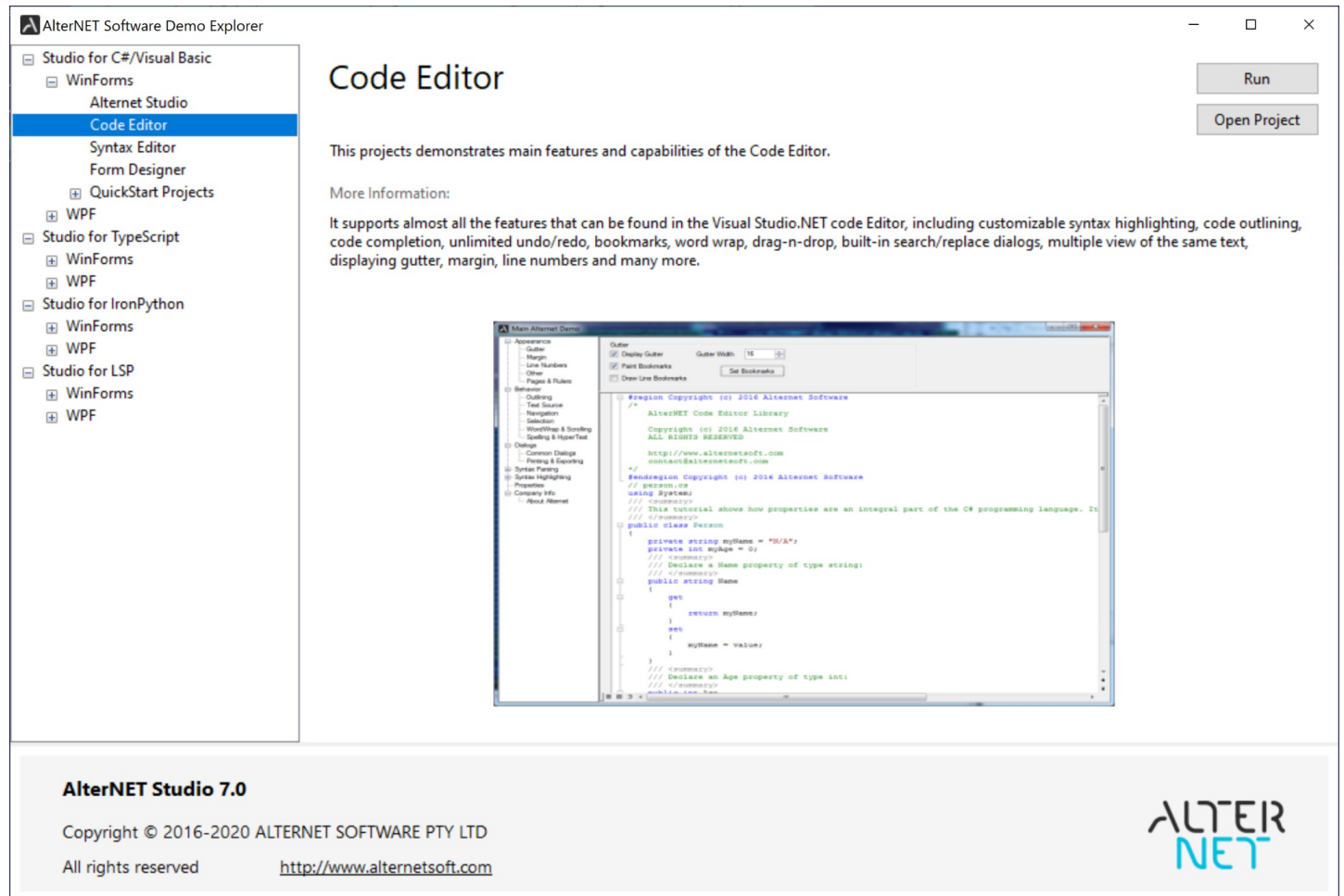
<http://stackoverflow.com/questions/51363/how-does-the-licenses-licx-based-net-component-licensing-model-work>

Code Editor Overview

AlterNET Code Editor is a .NET component library that brings efficient code editing functionality into your WinForms and WPF .NET applications. It provides code editing capabilities such as syntax highlighting, code completion and code outlining, visual indicators for bookmarks, line styles, syntax errors and much more.

The main components in the package are [SyntaxEdit](#) for WinForms and [TextEditor](#) for WPF. These controls provide text editing functionality and support almost all the features that can be found in the Visual Studio.NET code Editor, including customizable syntax highlighting, code outlining, code completion, unlimited undo/redo, bookmarks, word wrap, drag-n-drop, built-in search/replace dialogs, multiple view of the same text, displaying gutter, margin, line numbers and many more.

Code Editor includes a set of quick start projects, each one designed to highlight specific features of the component.



Below is brief overview of these projects:

Scroll Bar Annotations - Shows how text edit control can display markers about current line, syntax errors, bookmarks, modified lines and search results on the vertical scrollbar area.

Syntax Highlighting - Shows how text edit control can highlight syntax when working with different programming languages.

Code Completion - Shows how to display code completion while you type; either by getting code completion information from parser, or programmatically.

Code Outlining - Shows how to use outlining; either provided by parser, or programmatically.

Selection - Shows how to use different options to control text selection appearance and behavior in the text editor.

Undo/Redo - Shows how to use various options to control undo/redo behavior.

Search and Replace - Shows how to use built-in Search and Replace dialogs, and how to implement search across multiple

documents.

Gutter - Shows how to control the appearance of the gutter area and how to display various indicators on it.

Bookmarks - Shows how to set and navigate through numbered and though-loop bookmarks and how to set bookmarks navigation across multiple documents.

Word Warp - Shows how to configure text edit control to wrap words at the right-edge of the visible area or at a given position.

Line Styles - Shows how to display line indicators on the text editor control area and associated images on the gutter.

Print and Preview - Shows how to print and preview text editor control content and how to set different printing options.

Code Snippets - Shows how to display and use predefined code templates to speed-up entering frequently used fragments of code.

Multiple Views and Split View - Shows how to configure text editor windows to display and edit the same text content.

Margin - Shows how to use various options controlling the appearance and behavior of Margin line and UserMargin area next to gutter.

HyperText - Shows how to highlight hyperlinks in the text.

Page Layout - Shows how to configure text edit control to display its content as if it was positioned on the printed page.

Miscellaneous - Shows how to display white-space symbols and background images, as well as highlight matching brackets.

Customize - Shows example of options dialog that allows changing display settings of text edit control.

Roslyn-Based Parsing - Shows how to link text edit control to Microsoft Roslyn-based parsers that perform full syntax and semantic analysis of the C# or Visual Basic code and provide features like code completion, code outlining and syntax/semantic error highlighting.

TypeScript Parsing - Shows how to link text edit control to Microsoft TypeScript-based parsers that perform full syntax and semantic analysis of the TypeScript or JavaScript code and provide features like code completion, code outlining and syntax/semantic error highlighting.

Advanced Syntax Parsing - Shows how to link text edit control to parsers that perform syntax analysis for a set of programming languages and provide features like code completion, code outlining and syntax error highlighting.

Snippet Parsers - Shows how to implement C# or Visual Basic syntax and semantic analysis for sub-set of the code, like class or method body.

SQL DOM Parser - Shows how to implement syntax analysis for Microsoft SQL.

XAML Parser - Shows how to implement syntax analysis for XAML.

Lsp-based parsers (C/C++, Java ,Python, Lua XML, and PowerShell) - Shows how to implement syntax analysis for these languages using native servers.

Lsp Multiple Files - Shows how to combine multiple LSP documents into a single workspace.

Python Parsing - Shows how to implement syntax analysis for Python and IronPython.

Creating your first project

The first thing to do after creating a new WinForms or WPF application is to place the [SyntaxEdit](#) or [TextEditor](#) controls. These controls are the central components in the package, and in many cases they are the only ones that need to be placed on the form. These controls look similar to the standard multi-line text box, with the exception of having a gray band on the left of its client area, used to display line numbers, bookmarks and other visual indicators.

The following example demonstrates how to load text into the editor, and then to save it. Use the `LoadFile` method to load text from the file. The first parameter specifies the name of the file to be loaded into the control. The optional second parameter specifies encoding.

```
edit.LoadFile(openFileDialog1.FileName);
```

Saving of the text is performed in a similar way:

```
edit.SaveFile(saveFileDialog1.FileName);
```

It is possible to load text from streams instead of files, by substituting the previous two functions by [LoadStream](#) and [SaveStream](#).

Working with text

The Code Editor package includes a non-visual components: [TextSource](#) (or [TextSource](#) for WPF applications). For WindowsForms applications `TextSource` is accessible through Microsoft Visual Studio toolbox. The [SyntaxEdit](#) and [TextEditor](#) controls do not store the text being edited. This task is offloaded to the `TextSource` components, which provide a number of methods to manipulate the text content. Text edit controls can have `TextSource` explicitly assigned, and use internally created one in case it's not set. It gives clear separation between visualization and data layers, and also makes it possible to implement features like multiple views of the same text, by assigning a single [TextSource](#) to the multiple text editors. Visually, these editors can be either placed in a single window separated by a splitter control or in multiple windows. Most of the methods of `TextSource` components are also available via the edit controls.

Code Editor Basic Features (WinForms)

Code Editor matches most of the features of Visual Studio code editor, such as Selection, Code Completion, Code Outlining, Search/Replace, Navigation and Undo, and many more.

Editing features

Code Editor provides an extended set of methods and properties for the text modifications and navigating within the text content. Most of these methods are called implicitly when the user edits the text in the edit control or presses the arrow and `PageUp`, `PageDown`, `End` or `Home` keys. The most commonly used methods and properties are listed below:

- `Position` - gets or sets current position (Column, Row) within the text.
- `NewLine` - inserts new line at the current position and number of spaces or tabs according to the indentation options.
- `Insert` - inserts string at the current position.
- `DeleteRight` - deletes a given number of characters to the right of the current position.
- `DeleteLeft` - deletes a given number of characters to the right of the current position.
- `BreakLine` - inserts a line break at the given position.
- `UnBreakLine` - joins two lines at the end of a current line.
- `MoveCharLeft` - moves the current position to one character left.
- `MoveCharRight` - moves the current position to one character right.
- `MoveLineDown` - moves the current position to one line below.
- `MoveLineUp` - moves the current position to one line above.

. * `MoveFileBegin` - moves the current position to the beginning of the text.

. * `MoveFileEnd` - moves the current position to the end of the text.

All such modifications are translated to the underlying `TextSource` component, which also maintains the list of all edits so they can be undone.

Selection

Just like almost any text editor, the `SyntaxEdit` supports a concept of text selection and a wide range of operations on it. All the selection related aspects are controlled via the `Selection` property. Selections can be of two types: traditional stream-type selection, and block-type selection. The latter can be created by navigating the text with navigation keys holding *Shift* and *Alt* keys held together.

`BackColor` and `ForeColor` define the background and the foreground colors used to mark the currently selected text. `Selection.InActiveBackColor` and `Selection.InActiveForeColor` are used when the editor is out of focus.

The `Options` controls different aspects of behavior of selections.

- `DisableSelection` completely disables selection support in the editor.
- `DisableDragging` disables drag-n-drop operations on selection.
- `SelectBeyondEol` allows selection in the virtual space (if the `NavigateOptions.BeyondEol` is enabled)
- `UseColors` instructs the editor to use the same foreground colors for selected text, as the ones used for unselected text (i.e.

any syntax highlighting will be visible). Note for this to be useful, the section background color must be in contrast with all possible foreground colors.

- [HideSelection](#) causes the selection to become invisible when the editor loses focus.
- [SelectLineOnDbClick](#) allows the user to select the entire line by double-clicking on it.
- [DeselectOnCopy](#) causes selection to be removed after the user performs copy selection to clipboard operation.
- [PersistentBlocks](#) causes selection to be retained after the user has finished making it and has started other navigation.
- [OverwriteBlocks](#) causes the new input to overwrite the currently selected text.
- [SmartFormat](#) allows formatting blocks when pasting according to the rules defined by the syntax parser.
- [WordSelect](#) causes whole words to be selected rather than individual characters when using mouse selection.
- [DrawBorder](#) causes Edit control to draw border around selection
- [SelectLineOnTripleClick](#) allows to select whole line rather than single word by triple clicking the mouse
- [DeselectOnDbClick](#) causes selection to be cleared by dblclick.
- [ConvertToSpacesOnPaste](#) specifies that selection should convert all tabs to spaces in the text being pasted when `Lines.UseSpaces` is on.
- [RtfClipboard](#) causes selection to copy its content in text and rtf formats.
- [ClearOnDrag](#) causes selected text to be cleared after dragging from external source
- [CopyLineWhenEmpty](#) allows to copy whole line when selection is empty
- [DisableCodeSnippetOnTab](#) - disables code snippets insertion when pressing Tab key.
- [SelectWordOnCtrlClick](#) causes word under cursor to be selected when user holds Ctrl key
- [ExtendedBlockMode](#) causes text being typed to be inserted into the all selected lines within the rectangular block.

It is possible to programmatically select text by setting [SelectionStart](#) and [SelectionLength](#) properties, or with the help of [SetSelection](#) method.

The selected text can be retrieved or set via the [SelectedText](#) property.

Various operations can be programmatically performed on the current selection. Some of them are:

- [IsEmpty](#) checks whether there is any text selected.
- [SetSelection](#) selects the specified rectangular area.
- [SelectAll\(\)](#) selects the whole document.
- [Cut\(\)/Copy\(\)/Paste\(\)](#) performs standard operations like copying text to the clipboard, cutting text to the clipboard and pasting text from the clipboard.
- [IsPosInSelection](#) checks if the specified position lies within selection.
- [Clear\(\)](#) clears selection (this does not affect the text itself).
- [Move](#) moves or copies the currently selected text to a new location.
- [SmartFormat\(\)](#) formats the selected text according to the rules defined by the syntax parser.

- [LowerCase\(\)/UpperCase\(\)/Capitalize\(\)](#) change the case of the currently selected text.
- [UnIndent\(\)/UnIndent\(\)](#) change the indent of the currently selected text.

In fact, if some action can be performed by the user, it can also be performed programmatically:

```
if(!edit.Selection.IsEmpty)
edit.Selection.SelectedText =
"(" + edit.Selection.SelectedText + ")";
```

This code encloses the currently selected text in brackets.

Searching and Replacing

Among the operations that can be performed upon the text, there are operations of searching and replacing text strings. Unlike the standard multi-line text editor, which does not implement such a functionality, the [SyntaxEdit](#) control comes with the built-in support for them. It's ready to use out-of-the-box: when the user presses Ctrl+F key combination, the search dialog box appears:



The text-replace dialog can be activated by pressing Ctrl+H. Besides using the UI to control the process, all the operations can be executed programmatically by calling the corresponding methods of the [SyntaxEdit](#).

For example, to find some string, you could use the following code:

```
edit.Find("some string");
```

Or, with regular expressions:

```
edit.Find(" ", SearchOptions.RegularExpressions, new
System.Text.RegularExpressions.Regex("a.?z"));
```

To activate the Search Dialog:

```
edit.DisplaySearchDialog();
```

Moreover, the Search and Replace dialog box functionality is not hardwired: you can replace the dialog box by your own, by implementing the [ISearchDialog](#) interface, and assigning it to the editor by setting its SearchDialog property. The built-in dialog can serve as a good example and a starting point.

If you need to perform the Search and Replace operation without any user interaction, you can use the ReplaceAll method.

i.e.:

```
edit.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |
SearchOptions.EntireScope, out count);
```

After this, every occurrence of "bad" word in the entire text will be replaced by the "good".

Note, that this would move the cursor position to the place where the last replacement has been made, so if you need it to be

truly unnoticeable for the user, you need to enclose this call in the code which saves and restores the current cursor position:

```
System.Drawing.Point pos;
pos = edit.Position;
edit.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |
SearchOptions.EntireScope, out count);
edit.Position = pos;
```

Search/Replace function can work across multiple documents. In order to allow search to find text in multiple editors, you will need to set `SearchManager.Shared` to true and provide list of editors to perform search in its `InitSearch` event handlers and return/navigate to the appropriate editor in `GetSearch` event handler:

```
SearchManager.SharedSearch.Shared = true;
SearchManager.SharedSearch.InitSearch +=
    new InitSearchEvent(DoInitSearch);
SearchManager.SharedSearch.GetSearch +=
    new GetSearchEvent(DoGetSearch);

private void DoInitSearch(object sender, InitSearchEventArgs e)
{
    e.Search = GetActiveSyntaxEdit() as ISearch;
    foreach (var edit in editors.Values)
    {
        edit.SearchGlobal = true;
        e.SearchList.Add(edit.Source.FileName);
    }
}

private void DoGetSearch(object sender, GetSearchEventArgs e)
{
    foreach (var edit in editors.Values)
    {
        if (edit.Source.FileName == e.FileName)
        {
            e.Search = edit as ISearch;
            break;
        }
    }
}
```

Scroll Bars and Split View

The appearance and behavior of scrollbars is controlled by the [Scrolling](#) property.

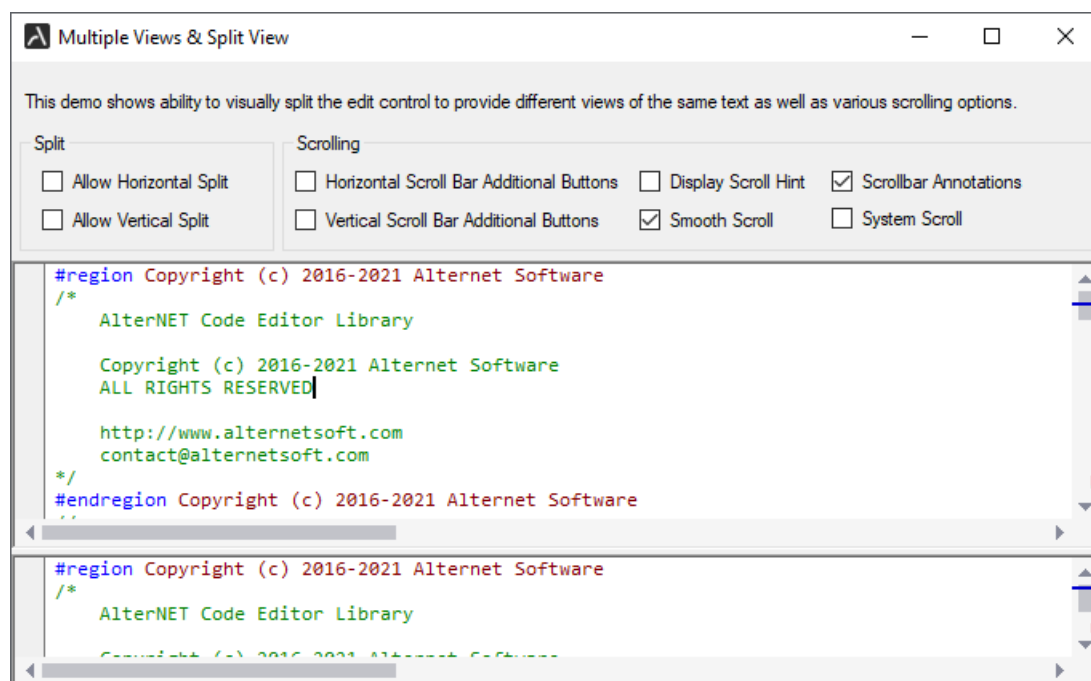
The [ScrollBars](#) property determines which scrollbars and under what conditions appear on the [SyntaxEdit](#). It can take one of the following values:

- [None](#) - neither horizontal, nor vertical scrollbar ever appear
- [Horizontal](#) - horizontal scrollbar appears if necessary, vertical one never appears
- [Vertical](#) - vertical scrollbar appears if necessary, horizontal one never appears
- [Both](#) - both horizontal and vertical scrollbars appear if necessary
- [ForcedHorizontal](#) - horizontal scrollbar is always visible, vertical one never appears
- [ForcedVertical](#) - vertical scrollbar is always visible, horizontal one never appears
- [ForcedBoth](#) - both horizontal and vertical scrollbars are always visible

Behavior of the scrollbars is controlled by *ScrollingOptions*.

You can also use *Scrolling.Options* to allow *SyntaxEdit* to split its content. Note that *SyntaxEdit*'s *Dock* must be set to *DockStyle.Fill*, otherwise this feature will not work. Splitters are displayed in the left-bottom corner for vertical splitting and in the right-top corner for horizontal splitting.

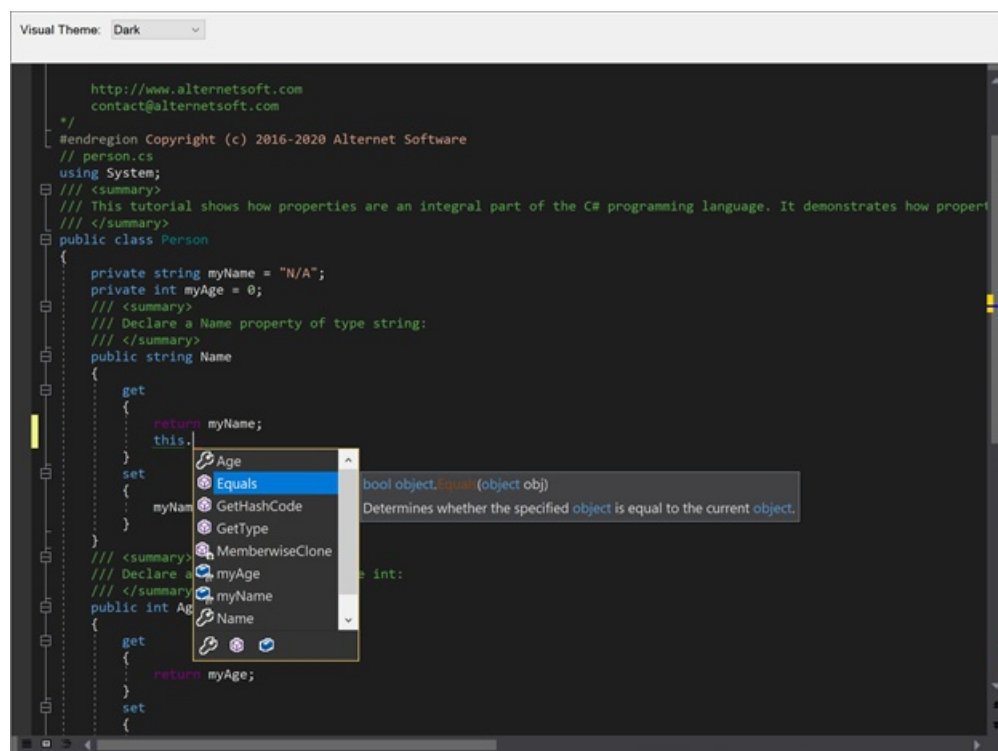
- [SmoothScroll](#) - if set, the display is updated as the user drags the scrollbar, otherwise the display is updated only when the user releases the scrollbar thumb. Disabling this option may improve performance on slow machines.
- [ShowScrollHint](#) - if set, a hint window, showing the new number of the topmost string, is displayed whenever the user drags the scrollbar.
- [UseScrollDelta](#) - if set, editor window content is scrolled by several characters when caret becomes invisible rather than one character
- [SystemScrollbars](#) - if set, system scroll bars are displayed, otherwise custom scrollbars are used.
- [FlatScrollbars](#) - if set, scroll bars are displayed in flat style. This option works only if *SystemScrollBars* is on.
- [AllowSplitHorz](#) - allows displaying horizontal splitting buttons in the scroll area. This option works only if *SystemScrollBars* is off and control has *Dock* property set to *DockStyle.Fill*.
- [AllowSplitVert](#) - allows displaying a vertical splitting button in the scroll area. This option works only if *SystemScrollBars* is off and control has *Dock* property set to *DockStyle.Fill*.
- [HorzButtons](#) - allows displaying additional buttons in the horizontal scrolling area. This option works only if *SystemScrollBars* is off.
- [VertButtons](#) - allows displaying additional buttons in the vertical scrolling area. This option works only if *SystemScrollBars* is off.
- [VerticalScrollBarAnnotations](#) - allows displaying scroll bar annotations that show special items such as line modifications, syntax errors, search results bookmarks and the caret position, throughout the entire document within the scroll bar. Individual annotation kinds are controlled by [Annotations](#) property.



Visual Themes

Visual themes allow to change the appearance of all graphical elements in the editor by setting [VisualThemeType](#) or [VisualTheme](#)

type properties. Light and Dark visual themes are included, and custom appearance can be configured via custom visual theme.



Gutter

The gutter is the area to the left of the text, the purpose of which is to display miscellaneous indicators for the corresponding lines of text. Among these indicators are bookmark indicators, line wrapping indicators, line styles icons, line numbers, outlining buttons and line modification markers.



All the images displayed in the gutter are contained in the gutters image list. The following code gives an example of how to add a custom icon to this list from another image list (for example, the one dropped on the form during design-time):

```
edit.Gutter.Images.Images.Add(imageList1.Images[0]);
```

The mechanism of the line styles icons allows you to define how certain lines of text will be displayed.

The most common use for this is the indication of breakpoint lines and of the current execution point.

For example, the following code defines the style to be used for breakpoints.

```
style_id = edit.LineStyles.AddLineStyle("breakpoint",  
Color.White, Color.Red, Color.Gray, 11, LineStyleOptions.BeyondEol);
```

(Note, in the current version, image # 11 corresponds to the built-in breakpoint indicator image, and #12 corresponds to the current execution point image.

Later on, some line of the text can be assigned the style:

```
edit.Source.LineStyles.SetLineStyle(line_no, style_id);
```

(Note, that here and in the other places of this document line numbers start at 0.)

Note: at any given time, every line can have at most one style. If you need to remove line style for some particular line, call:

```
edit.Source.LineStyles.RemoveLineStyle(line_no);
```

For [SyntaxEdit](#) control appearance of the gutter is controlled by the following properties: [Width](#), [BrushColor](#), [PenColor](#) and [Visible](#). [Width](#) property specifies width of the gutter area, [BrushColor](#) specifies background color of the gutter area, [PenColor](#) specifies color of the gutter line, and [Visible](#) indicates whether or not to draw gutter. Note that gutter can adjust its width if line numbers or outlining is on and painted on the gutter. [SyntaxEdit](#) allows drawing line numbers to visually indicate position of the visible lines inside the document. To enable line numbers you need to set [PaintLineNumbers](#) to true. Turning [PaintLinesOnGutter](#) option on enables drawing line numbers on the gutter area, turning it off causes line numbers to be painted immediately after the gutter area. Appearance of line numbers are controlled by the [IGutter](#)'s properties: [LineNumbersStart](#), [LineNumbersForeColor](#), [LineNumbersBackColor](#), [LineNumbersAlignment](#), [LineNumbersLeftIndent](#) and [LineNumbersRightIndent](#), which are intuitively understandable.

Like Microsoft Visual Studio editor, [SyntaxEdit](#) provides the ability to visually track modified lines. To enable this feature you need to turn [PaintLineModifiers](#) on. When [LineModifiers](#) are on they indicate lines that were changed since last saving. New changes are marked with Yellow color; changes that were done before last saving are marked with Lime color. Colors can be customized using [LineModifierChangedColor](#) and [LineModifierSavedColor](#) properties.

Reaction to mouse clicks and double-clicks on the gutter area can be implemented by assigning handlers to the [GutterClick](#) and [GutterDbClick](#) events.

Bookmarks

Just as with often used reference books, the process of navigating the text can be made more efficient with the usage of bookmarks. Two kinds of bookmarks are supported by the [SyntaxEdit](#): plain and numbered. The former can be toggled for the current line using the [Ctrl+K Ctrl+K](#) key combination sequence, and can be navigated in cyclical manner using the [Ctrl+K Ctrl+N](#) (next bookmark) or [Ctrl+K Ctrl+P](#) (previous bookmark). The numbered bookmarks have a different flavor: there can be up to ten bookmarks, each having a number associated with it.



Toggling the numbered bookmark is performed using the [Ctrl+K Ctrl+#](#), and navigation to the specific bookmark is performed by pressing the [Ctrl+#](#) key combination (where # is any of the digits from 0 to 9). There can be only one plain bookmark in any line. Numbered bookmarks do not have such a limitation, however, only the indicator for the first bookmark in the line will be displayed in the gutter area, if [PaintBookMarks](#) is set to true.

Like most other things in the editor, bookmarks can be manipulated programmatically. Note that the list of bookmarks belongs to the text source, so multiple views of the same source share the same set of bookmarks.

The following code snippet sets the plain bookmark at the current position:

```
System.Drawing.Point pos = edit.Position;
edit.Source.BookMarks.SetBookMark(pos, int.MaxValue);
```

To set the numbered bookmark, replace *int.MaxValue* by the bookmark number (0..9).

To clear all the bookmarks set in the text source, call the [ClearAllBookMarks\(\)](#) method:

```
edit.Source.BookMarks.ClearAllBookMarks();
```

Navigating to the location defined by a particular bookmark can be performed as follows:

```
edit.Source.BookMarks.GotoBookMark(index);
```

Code Editor supports named bookmarks with description and hyperlink. The user may see a description in a tooltip window when moving the cursor over the bookmark, and load the browser with specified url when clicking on the bookmark. Such bookmarks can be set using the following code:

```
edit.Source.BookMarks.SetBookMark(edit.Position, 0,
    "Bookmark1", "This is Named Bookmark", "www.alternetsoft.net");
```

If you need to have custom images, you can change the bookmark indicator images by assigning custom image list:

```
edit.Gutter.BookMarkImageIndex =
edit.Gutter.Images.Images.Count;
edit.Gutter.Images.Images.Add(imageList1.Images[0]);
```

(This code uses the first image from the *imageList1*, which you could, for example, create by just dropping a new Image List from the toolbox on the form. For more examples on working with the gutter, refer to the corresponding section of this manual.)

You can configure bookmarks navigation to work across multiple documents. These documents should be added to the *BookMarkManager* class, and every document should have the *FileName* property assigned.

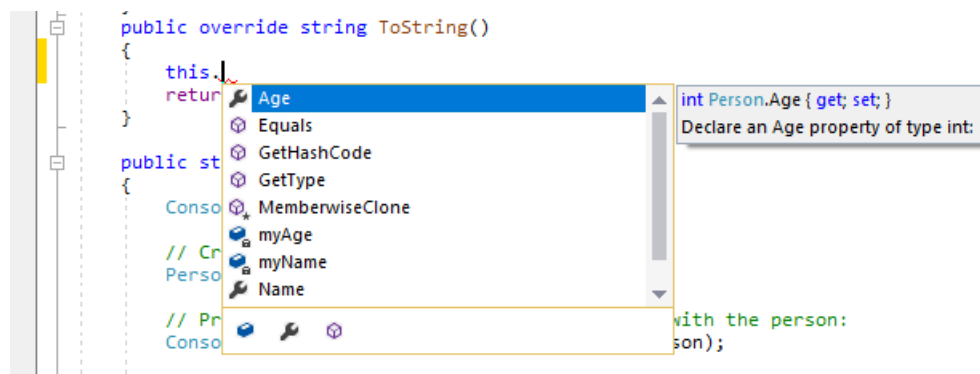
```
BookMarkManager.Register(edit.Source);
BookMarkManager.SharedBookMarks.Activate += new
EventHandler<ActivateEventArgs>(DoActivate);
private void DoActivate(object sender, ActivateEventArgs e)
{
    foreach (var edit in editors.Values)
    {
        if (edit.Source.FileName == e.FileName)
        {
            ActivateEditorTab(editor);
            break;
        }
    }
}
```

In this mode all bookmarks will be stored in a global list inside *BookmarkManager* instead of every individual *SyntaxEdit* control allowing global navigation through them.

Code Completion (Intellisense)

Although the main purpose of an editor is to be a convenient tool for the user to enter the text, quite often a guidance from the editor can significantly improve the effectiveness of the work process. When editing a text which has some structure (i.e. computer program in some language), there are often well-defined sets of input possibilities in certain contexts. For example, for many programming languages, the sequence "someobject." should be followed by one of the existing field names. To assist the user in such situations, the text editor can activate a popup list containing all the methods that can be accessed from the current

scope.



If there is a partial word immediately to the left of the current cursor position, the first entry that starts with that word is highlighted. The user can then continue typing up until the method which he meant is selected or just use up and down arrow keys to navigate the list, and then insert the complete method name by pressing the Enter key.

Automatic Code Completion Invocation

In most cases Code Completion list and Signature Help for method parameters are provided by parser, alongside with the list of characters, such as period "." or open parens "(", which invoke code completion automatically as user types. The task of code completion is to have the list of available choices to appear automatically as user types, for example after user types "someobject." the list of class members for that object is expected to appear, and after they type "somemethod(" the tooltip showing the list of parameters for that function is expected to appear. It can be customized to show those popups only if the user stops input for some short period of time after typing the activating symbol ("." or "(").

The automatic code completion is implemented by Roslyn C# and Visual Basic parsers, TypeScript/JavaScript parsers, as well as by Advanced C#, J#, Visual Basic, VBScript, JavaScript, C, XML and Python parsers.

For example, automatic code completion is attempted after typing a period (".") following a member (member access expression), typing an open brace ("(") following a member (invocation expression or object creation expression), typing a period (".") inside *using* section, typing less sign ("<") inside xml comments, etc. This feature is implemented as close as possible to the Visual Studio .NET editor, so it works in an intuitively understandable way. On top of that Roslyn-based parsers are configured to invoke code completion when the user starts typing identifiers.

When these parsers are used, you still can control some aspects of code completion, for example delay before code completion window appears, using the [NeedCodeCompletion](#) event, which will be discussed later. Moreover, for advanced parsers you can register your own types and objects, namespaces and assemblies for code completion using the `CompletionRepository` property of [SyntaxParser](#).

To make types from most commonly used assemblies such as `System`, `System.Drawing`, and `System.Windows.Forms` to be available for code completion, you can call the following method

```
csParser1.Repository.RegisterDefaultAssemblies();
```

If you need to provide code completion for assemblies declared in other assemblies, you need to register these assemblies this way:

```
csParser1.Repository.RegisterAssembly("System.Xml");
```

You may need to register types for code completion that are not declared in the assembly, but present in the form of source code somewhere else.

For Roslyn-based parsers you can rely on underlying solution/project/document object model:

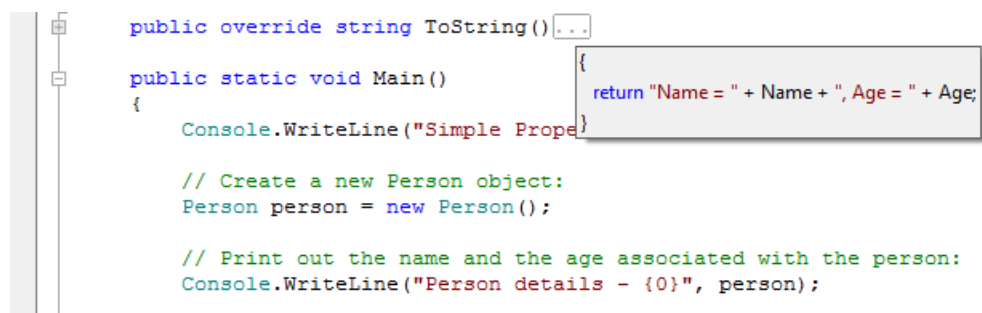
```
csParser1.Repository.RegisterCodeFiles(new string[] { "MyFile.cs" });
```

For one of advanced parsers you need first to create [SyntaxParser](#), load this file into the [Strings](#) object, and then add parsed [SyntaxTree](#) to the code completion repository. The following code demonstrates how it can be accomplished:

```
ISyntaxParser parser = new Altnet.Syntax.Parsers.Advanced.CsParser();
parser.Strings = new TextStrings();
parser.Strings.LoadFile("MyFile.cs")
parser.ReparseText();
csParser1.CompletionRepository.RegisterSyntaxTree(parser.SyntaxTree);
```

Code Outlining

The [SyntaxEdit](#) control supports outlining, which is a text navigation feature that can make navigation of large structured texts more effective. The essence of *outlining* lies in defining sections of the text as structural units that once collapsed, can be visually replaced by a shorter representation, i.e. by ellipsis ("..."). During the text navigation the user can dynamically switch between the collapsed and complete representation of any particular section. Sections can be nested.



```
public override string ToString()...
{
    return "Name = " + Name + ", Age = " + Age;
}

public static void Main()
{
    Console.WriteLine("Simple Propo

// Create a new Person object:
Person person = new Person();

// Print out the name and the age associated with the person:
Console.WriteLine("Person details - {0}", person);
```

The section can be expanded by clicking on the "+" button, by double-clicking the proxy text, or by pressing the Ctrl+M Ctrl+M key sequence (in the default key mapping). The section can be collapsed by clicking on the "-" button, or by pressing the Ctrl+M Ctrl+M key sequence. All the sections can be globally collapsed or expanded using the Ctrl+M Ctrl+L key sequence.

Outlining is the property of the [SyntaxEdit](#) control itself, not of the [TextSource](#), thus it is possible to have two views of the same text one with outlining and another without, or even to have completely different structural parts defined.

All the aspects of the *outlining* are controlled via the [Outlining](#) property of the [SyntaxEdit](#). The *outlining* can be enabled or disabled using the [AllowOutlining](#) property either in design time or at runtime. The look of the outline is controlled by the [OutlineColor](#) and [OutlineOptions](#) properties.

There are two approaches to defining outline sections.

Direct Definition of Outline Sections

Outline sections can be explicitly defined by calling the appropriate methods of the [Outlining](#) property, i.e.:

```
edit.Outlining.Outline(new Point(0, 0), new Point(int.MaxValue, 0), 0, "...").Visible = false;
```

This code snippet defines the section of the first level consisting of the entire first line of the text, using ellipsis ("...") as the proxy text and being in a collapsed state.

While this approach is the simple one, it has one significant drawback: if sections represent structural units defined by the text itself, and the text can be edited by the user, sections have to be somehow constantly kept in sync with the text, which can be a non-trivial task.

Automatic Definition of Outline Sections Using the Syntax Parser

To provide automatic code outlining, the syntax parsing framework has to be employed. This approach may seem to be more complex at the first look, however it provides consistent results. To implement this approach, a class descending from the [SyntaxParser](#) class needs to be defined, and the [Outline](#) method needs to be implemented. This method will be frequently called by the [SyntaxEdit](#) whenever the text changes, so, to provide the user with a smooth editing experience, the implementation should

be relatively fast.

Code Editor includes parsers that support automatic outlining for *C#, Visual Basic, J#, JavaScript, VBScript, Ansi-C, SQL, HTML, XML and Python* languages.

The following example demonstrates how to implement a parser the marks every line starting from the sharp ("#") sign as a separate outline section.

```
private void InitializeComponent()
{
    ...
    this.parser1 = new XParser();
    ...
}
public class XParser: SyntaxParser
{
    public XParser()
    {
        Options = SyntaxOptions.Outline;
    }
    public override int Outline(ICollection<IRange> Ranges)
    {
        Ranges.Clear();
        for(int i = 0; i < Strings.Count; i++)
        {
            if(Strings[i].ToString().StartsWith("#"))
            {
                Ranges.Add(new OutlineRange(
                    new Point(0, i),
                    new Point(int.MaxValue, i),
                    0, "...", false));
            }
        }
        return Ranges.Count;
    }
}
```

Code Editor Extended Features (WinForms)

Code Editor provides advanced text editing functionality such as customizable keyboard mapping, HyperText handling, spell-checking integration, printing and exporting, macro recording and playback and miscellaneous display features.

Keyboard Mapping

While the [SyntaxEdit](#) closely mimics the key-mapping common to most of Microsoft's products, it is completely customizable: you can add or change behavior of certain keys or even define an entirely different key-mapping.

To assign an action to some key combination, use the following code:

```
private void edit_Action()  
{  
    ...  
}  
...  
edit.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new  
    KeyEvent(edit_Action));
```

This would make the *Ctrl+Alt+W* key combination execute the *edit_Action* method.

Or, to pass some object to the key handler:

```
private void edit_Action(object o)  
{  
    ...  
}  
...  
edit.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new  
    KeyEventEx(edit_Action), some_object);
```

To remove some key handler, regardless of whether you have added it yourself, or it is the default one, call:

```
edit.KeyList.Remove(Keys.A | Keys.Control);
```

The code described before is used to manage the key handling in the default state. In fact, the key handling is slightly more complex than that: the [SyntaxEdit](#)'s key handling mechanism can be in different states, other than the default one. Every state has its own key mapping table. Key mapping for bookmark operations can serve as a good example: after the user presses the *Ctrl+K* key combination, combinations *Ctrl+K*, *Ctrl+N*, *Ctrl+P*, *Ctrl+L* (the list is incomplete) obtain the new meaning. If a key combination is pressed for which there is no assignment in some non-default state, then the state is changed to default, and the combination is evaluated in the new context. [SyntaxEdit](#) defines four different non-default states, but you can implement your own:

```
edit.KeyList.Add(Keys.W | Keys.Control, null, 0, 5);  
edit.KeyList.Add(Keys.Tab, new KeyEvent(edit_Action), 5, 5);
```

This code creates a state that is activated by pressing the *Ctrl+W* key combination, and in which the *Tab* key causes the *edit_Action* to be executed. The state is changed back to default when the user presses some key other than the *Tab*. Up until now we have only examined the cases where you add some new functionality, or suppress some existing one. There also might be a case, when you want to use an entirely different key mapping, for example, to simulate some other environment your users are familiar with. To accomplish this, it is necessary to completely clear the current key mapping, and then to assign every function performed by the editor to some key. Note, that this really means every function: even such trivial things as cursor navigation and insertion of a new line are performed according to the key mapping.

For example, the following code assigns the editor's key-mapping to a single action defined: "Select All", which is assigned to the

Ctrl+*X* key combination

```
edit.KeyList.Clear();
edit.KeyList.Add(Keys.X | Keys.Control,
((EventHandlers)edit.KeyList.Handlers)SelectAllEvent);
```

URL handling

The [SyntaxEdit](#) can be set up to handle pieces of text that look like some kind of an URL by setting the [HighlightHyperText](#) property to true. The handling consists of highlighting those pieces of text, and of processing clicks on them. By default, clicking the URL causes the operating system default action to be performed (i.e. launching a browser or an email client), however, you can override this behavior by assigning the [JumpToUrl](#) event handler.

```
private void edit_JumpToUrl(object sender, UrlJumpEventArgs e)
{
    if(is_our_url(e.Text))
    {
        process_url(e.Text);
        e.Handled = true;
    }
}
```

Spellchecker Interface

The [SyntaxEdit](#) supports the spell-as-you-type spell checker integration. To enable spelling for the editor, set its [CheckSpelling](#) property to true and assign the WordSpell event handler.

The following artificial example considers any word longer than 3 characters to be correct:

```
private void edit_WordSpell(object sender, WordSpellEventArgs e)
{
    e.Correct = e.Text.Length > 3;
}
...
this.edit.WordSpell += new WordSpellEvent(this.edit_WordSpell);
```

Incorrect words are displayed with the wiggly underline (the default color is red, but it can be changed using the [SpellColor](#) property). In real-life scenarios you would need to use some third-party software/dictionary to really check the text. Another alternative would be using some word-list file, many of them, including Public Domain or free ones, can be found on the Internet. Refer to a Miscellaneous quick start project, which has one of these dictionaries.

Another useful feature supported by [SyntaxEdit](#) is AutoCorrect, allowing you to auto correct words when typing. To enable this feature you need to set property AutoCorrection to true and handle the AutoCorrect event to provide replacements for words that were typed incorrectly.

Printing and Exporting

[SyntaxEdit](#) includes support for printing, print previewing, and exporting to RTF and HTML.

Exporting can be performed as simple as this:

```
edit.SaveFile(FileName, new RtfExport());
```

Printing tasks are performed and configured via the Printing property of the [SyntaxEdit](#).

For example, to show the print preview dialog, call:

```
edit.Printing.ExecutePrintPreviewDialog();
```

[SyntaxEdit](#) control supports adding user-defined information while printing.

To add some text to the footer:

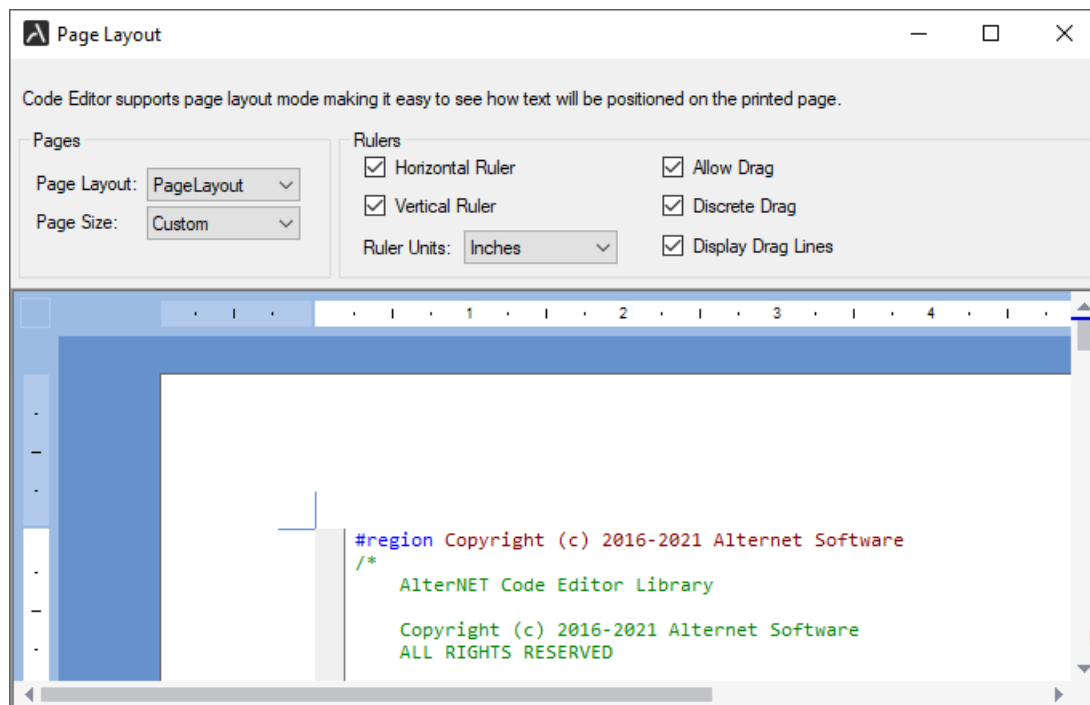
```
edit.Printing.Footer.CenterText = "draft";
```

Text in headers and footers can contain substitution tags. The standard ones are: [page], [pages], [date], [time] and [username]. It is possible to add custom tags by assigning a handler for the [DrawHeader](#) event, for example:

```
private void edit_DrawHeader(object sender, Alternet.Editor.DrawHeaderEventArgs e)
{
    if(e.Tag == "\\[tag]")
    {
        e.Text = "tag replacement text";
        e.Handled = true;
    }
}
```

Page Layout mode

[SyntaxEdit](#) has different ways to get a good view of the editor content. Use normal mode for typing, editing, and formatting text. Working in page layout mode making it easy to see how text will be positioned on the printed page. Page breaks mode is similar to normal mode, but allows to visually separate pages by displaying dotted lines between individual pages. Current mode is controlled by the [PageType](#) property. Use the [DefaultPage](#) property to change bounds and margins of the default page. In Page Layout mode it may be useful to display horizontal and vertical rulers, which will allow users to visually change margins of the current page or selected range of pages. Rulers can be turned on or off using [Rulers](#) property.



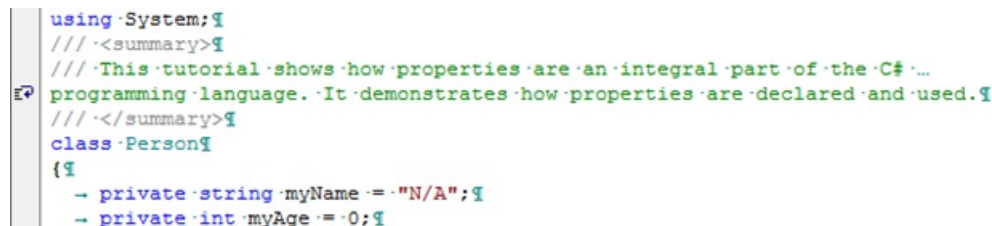
Marco Recording and PlayBack

[SyntaxEdit](#) has macro recording and playback capabilities. It allows recording sequences of keyboard commands and playing them later. Note that mouse input is not recorded.

This feature enables you to store a set of frequently used editing commands. Set *MacroRecording* property to start/finish macro recording. Use the *PlayBack* method to repeat the stored command sequence.

White-space Display

It is sometimes desirable for the user to see the codes which influence the layout of the text and are normally invisible themselves. These codes are space, tab, end-of-line, and the end-of-file (not really a code), and are often collectively referred to as the white-space. The `SyntaxEdit` has the option to display them, and to control their appearance.



```
using System;
/// <summary>
/// This tutorial shows how properties are an integral part of the C# ...
programming language. It demonstrates how properties are declared and used.
/// </summary>
class Person
{
    → private string myName = "N/A";
    → private int myAge = 0;
```

The display of the white-space is enabled using the `Visible` property. The color used to display white-space codes is determined by the `SymbolColor` property, and the characters used to display those codes are determined by `EofSymbol`, `EolSymbol`, `SpaceSymbol`, and `TabSymbol` properties.

Line Separator

It is possible to have lines of the editor to be separated by thin horizontal lines, and to have the current line highlighted. This behavior is controlled by the `LineSeparator` property.

The following options are available:

- `HighlightCurrentLine` specifies that the current line in the editor will be highlighted using the `HighlightColor` for background.
- `HideHighlighting` specifies that the highlighting of the current line should be hidden when the editor loses focus.
- `SeparateLines` specifies that a thin horizontal line of `LineColor` should be drawn between each line of text.
- `SeparateWrapLines` specifies that each visual line of text produced as a result of word-wrap should be separated in the same manner as separate lines (works only if the `SeparateLines` option is also specified).
- `SeparateContent` specifies that line separator will be drawn between sections of the code (for example between methods), if `SyntaxEdit` control is associated with `SyntaxParser` supporting this feature.

Code snippets

The code snippets are the next code completion provider, allowing to insert frequently used fragments of code. Code snippets can be inserted into the editor by pressing `Tab` key after snippet shortcut or by executing code snippet popup window with `Ctrl + K + X` key sequence, or activated programmatically, by calling the `CodeSnippets` method of the `SyntaxEdit`.

The purpose of the code snippets is to permit the user to quickly enter one of the predefined fragments of text. If the code snippet has fields declared, the editor allows modifying their values causing updating field values inside the whole snippet.

The following picture illustrates the usage of the code snippets.

```
public static void Main()
{

    for (int i = 0; i < length; i++)
    {
        Insert Snippet: i
        #if
        #region
        attribute
        exception
        if
        indexer
        interface
        invoke
    }
}
```

Code snippet for if statement
Shortcut: if

Hidden and Read-Only Lines

[SyntaxEdit](#) control can mark certain lines to be readonly or hide them at all so the user can't see them. This can be achieved by using [SetLineHidden](#) and [SetLineReadOnly](#) methods. For hidden lines to take effect, the [AllowHiddenLines](#) property needs to be set to true. Read-only lines can be made visually different from editable lines by setting [ReadOnlyBackColor](#) property. Sometimes it's required to mark certain lines to be both hidden and readonly, this way they can not be deleted if the user selects the outer block containing them and tries to delete it.

```
using System;
public class MyClass1
{
    public string MyFunc()
    {
        Console.WriteLine("Simple Properties");

        // Create a new Form object:
        System.Windows.Forms.Form form1 = new System.Windows.Forms.Form(

        // Print out the name and the text associated with the form:
        Console.WriteLine("Form details - {0}", form1);

        // Set some values on the form object:
        form1.Name = "Form1";
    }
}
```

Structure GuideLines

[SyntaxEdit](#) control can display dashed lines between syntax blocks for some parsers (Roslyn-based, TypeScript and some advanced parsers), helping the user to better understand the structure of the document being edited. This behavior is controlled by a Parser and can be switched off by the StructureGuideLines parser option.

```
public static void Main()
{
    Console.WriteLine("Simple Properties");

    // Create a new Person object:
    Person person = new Person();

    // Print out the name and the age associated with the person:
    Console.WriteLine("Person details - {0}", person);

    // Set some values on the person object:
    person.Name = "Joe";
    person.Age = 99;
    Console.WriteLine("Person details - {0}", person);

    // Increment the Age property:
    person.Age += 1;
}
```


Code Editor Basic Features (WPF)

Code Editor matches most of the features of Visual Studio code editor, such as Selection, Code Completion, Code Outlining, Search/Replace, Navigation and Undo, and many more.

Editing features

Code Editor provides an extended set of methods and properties for the text modifications and navigating within the text content. Most of these methods are called implicitly when the user edits the text in the edit control or presses the arrow and `PageUp`, `PageDown`, `End` or `Home` keys. The most commonly used methods and properties are listed below:

- [Position](#) - gets or sets current position (Column, Row) within the text.
- [NewLine](#) - inserts new line at the current position and number of spaces or tabs according to the indentation options.
- [Insert](#) - inserts string at the current position.
- [DeleteRight](#) - deletes a given number of characters to the right of the current position.
- [DeleteLeft](#) - deletes a given number of characters to the right of the current position.
- [BreakLine](#) - inserts a line break at the given position.
- [UnBreakLine](#) - joins two lines at the end of a current line.
- [MoveCharLeft](#) - moves the current position to one character left.
- [MoveCharRight](#) - moves the current position to one character right.
- [MoveLineDown](#) - moves the current position to one line below.
- [MoveLineUp](#) - moves the current position to one line above.

. * [MoveFileBegin](#) - moves the current position to the beginning of the text.

. * [MoveFileEnd](#) - moves the current position to the end of the text.

All such modifications are translated to the underlying [TextSource](#) component, which also maintains the list of all edits so they can be undone.

Selection

Just like almost any text editor, the [TextEditor](#) supports a concept of text selection and a wide range of operations on it. All the selection related aspects are controlled via the [Selection](#) property. Selections can be of two types: traditional stream-type selection, and block-type selection. The latter can be created by navigating the text with navigation keys holding *Shift* and *Alt* keys held together.

[SelectionBrush](#) and [SelectionForeColor](#) define the background and the foreground colors used to mark the currently selected text. [Selection.InActiveBackColor](#) and [Selection.InActiveForeColor](#) are used when the editor is out of focus.

The [Options](#) controls different aspects of behavior of selections.

- [DisableSelection](#) completely disables selection support in the editor.
- [DisableDragging](#) disables drag-n-drop operations on selection.
- [SelectBeyondEol](#) allows selection in the virtual space (if the [NavigateOptions.BeyondEol](#) is enabled)
- [UseColors](#) instructs the editor to use the same foreground colors for selected text, as the ones used for unselected text (i.e.

any syntax highlighting will be visible). Note for this to be useful, the section background color must be in contrast with all possible foreground colors.

- [HideSelection](#) causes the selection to become invisible when the editor loses focus.
- [SelectLineOnDbClick](#) allows the user to select the entire line by double-clicking on it.
- [DeselectOnCopy](#) causes selection to be removed after the user performs copy selection to clipboard operation.
- [PersistentBlocks](#) causes selection to be retained after the user has finished making it and has started other navigation.
- [OverwriteBlocks](#) causes the new input to overwrite the currently selected text.
- [SmartFormat](#) allows formatting blocks when pasting according to the rules defined by the syntax parser.
- [WordSelect](#) causes whole words to be selected rather than individual characters when using mouse selection.
- [DrawBorder](#) causes Edit control to draw border around selection
- [SelectLineOnTripleClick](#) allows to select whole line rather than single word by triple clicking the mouse
- [DeselectOnDbClick](#) causes selection to be cleared by dblclick.
- [ConvertToSpacesOnPaste](#) specifies that selection should convert all tabs to spaces in the text being pasted when `Lines.UseSpaces` is on.
- [RtfClipboard](#) causes selection to copy its content in text and rtf formats.
- [ClearOnDrag](#) causes selected text to be cleared after dragging from external source
- [CopyLineWhenEmpty](#) allows to copy whole line when selection is empty
- [DisableCodeSnippetOnTab](#) - disables code snippets insertion when pressing Tab key.
- [SelectWordOnCtrlClick](#) causes word under cursor to be selected when user holds Ctrl key
- [ExtendedBlockMode](#) causes text being typed to be inserted into the all selected lines within the rectangular block.

It is possible to programmatically select text by setting [SelectionStart](#) and [SelectionLength](#) properties, or with the help of [SetSelection](#) method.

The selected text can be retrieved or set via the [SelectedText](#) property.

Various operations can be programmatically performed on the current selection. Some of them are:

- [IsEmpty](#) checks whether there is any text selected.
- [SetSelection](#) selects the specified rectangular area.
- [SelectAll\(\)](#) selects the whole document.
- [Cut\(\)/Copy\(\)/Paste\(\)](#) performs standard operations like copying text to the clipboard, cutting text to the clipboard and pasting text from the clipboard.
- [IsPosInSelection](#) checks if the specified position lies within selection.
- [Clear\(\)](#) clears selection (this does not affect the text itself).
- [Move](#) moves or copies the currently selected text to a new location.
- [SmartFormat\(\)](#) formats the selected text according to the rules defined by the syntax parser.

- [LowerCase\(\)/UpperCase\(\)/Capitalize\(\)](#) change the case of the currently selected text.
- [UnIndent\(\)/UnIndent\(\)](#) change the indent of the currently selected text.

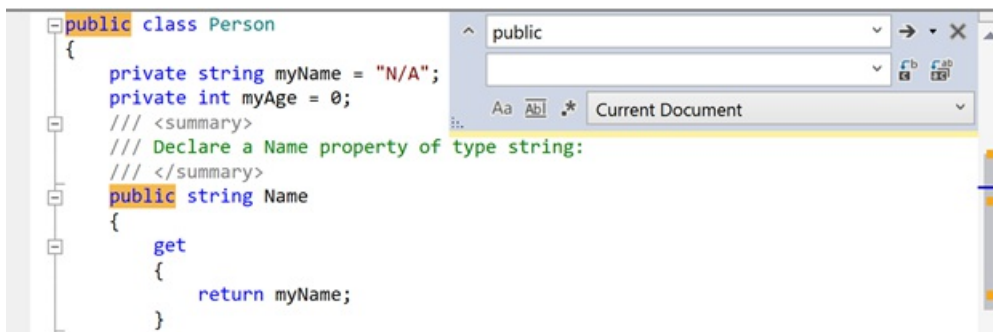
In fact, if some action can be performed by the user, it can also be performed programmatically:

```
if(!edit.Selection.IsEmpty)
edit.Selection.SelectedText =
"(" + edit.Selection.SelectedText + ")";
```

This code encloses the currently selected text in brackets.

Searching and Replacing

Among the operations that can be performed upon the text, there are operations of searching and replacing text strings. Unlike the standard multi-line text editor, which does not implement such a functionality, the [TextEditor](#) control comes with the built-in support for them. It's ready to use out-of-the-box: when the user presses Ctrl+F key combination, the search dialog box appears:



The text-replace dialog can be activated by pressing Ctrl+H. Besides using the UI to control the process, all the operations can be executed programmatically by calling the corresponding methods of the [TextEditor](#).

For example, to find some string, you could use the following code:

```
edit.Find("some string");
```

Or, with regular expressions:

```
edit.Find(" ", SearchOptions.RegularExpressions, new
System.Text.RegularExpressions.Regex("a.?z"));
```

To activate the Search Dialog:

```
edit.DisplaySearchDialog();
```

Moreover, the Search and Replace dialog box functionality is not hardwired: you can replace the dialog box by your own, by implementing the [ISearchDialog](#) interface, and assigning it to the editor by setting its SearchDialog property. The built-in dialog can serve as a good example and a starting point.

If you need to perform the Search and Replace operation without any user interaction, you can use the ReplaceAll method.

I.e.:

```
edit.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |
SearchOptions.EntireScope, out count);
```

After this, every occurrence of "bad" word in the entire text will be replaced by the "good".

Note, that this would move the cursor position to the place where the last replacement has been made, so if you need it to be

truly unnoticeable for the user, you need to enclose this call in the code which saves and restores the current cursor position:

```
System.Drawing.Point pos;
pos = edit.Position;
edit.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |
SearchOptions.EntireScope, out count);
edit.Position = pos;
```

Search/Replace function can work across multiple documents. In order to allow search to find text in multiple editors, you will need to set `SearchManager.Shared` to true and provide list of editors to perform search in its `InitSearch` event handlers and return/navigate to the appropriate editor in `GetSearch` event handler:

```
SearchManager.SharedSearch.Shared = true;
SearchManager.SharedSearch.InitSearch +=
    new InitSearchEvent(DoInitSearch);
SearchManager.SharedSearch.GetSearch +=
    new GetSearchEvent(DoGetSearch);

private void DoInitSearch(object sender, InitSearchEventArgs e)
{
    e.Search = GetActiveTextEditor() as ISearch;
    foreach (var edit in editors.Values)
    {
        edit.SearchGlobal = true;
        e.SearchList.Add(edit.Source.FileName);
    }
}

private void DoGetSearch(object sender, GetSearchEventArgs e)
{
    foreach (var edit in editors.Values)
    {
        if (edit.Source.FileName == e.FileName)
        {
            e.Search = edit as ISearch;
            break;
        }
    }
}
```

Scroll Bars and Split View

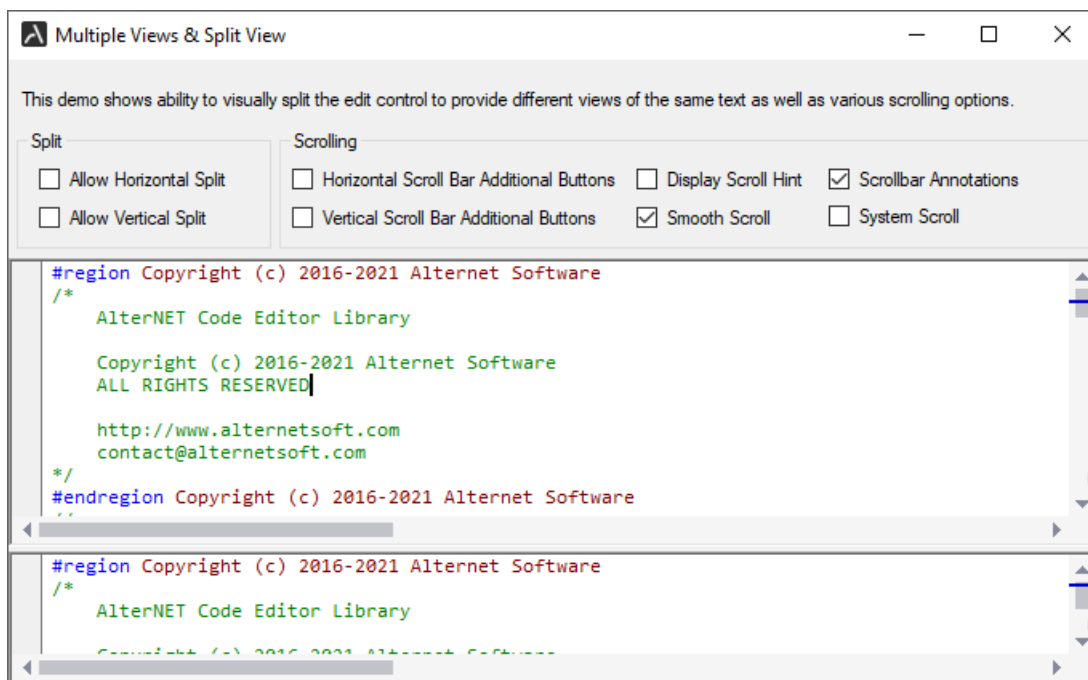
The appearance and behavior of scrollbars is controlled by the [Scrolling](#) property.

Behavior of the scrollbars is controlled by *ScrollingOptions*.

You can also use *Scrolling.Options* to allow [TextEditor](#) to split its content by setting [AllowVerticalEditorSplit](#) property. Splitter is displayed in the left-bottom corner allowing splitting TextEditor's content vertically.

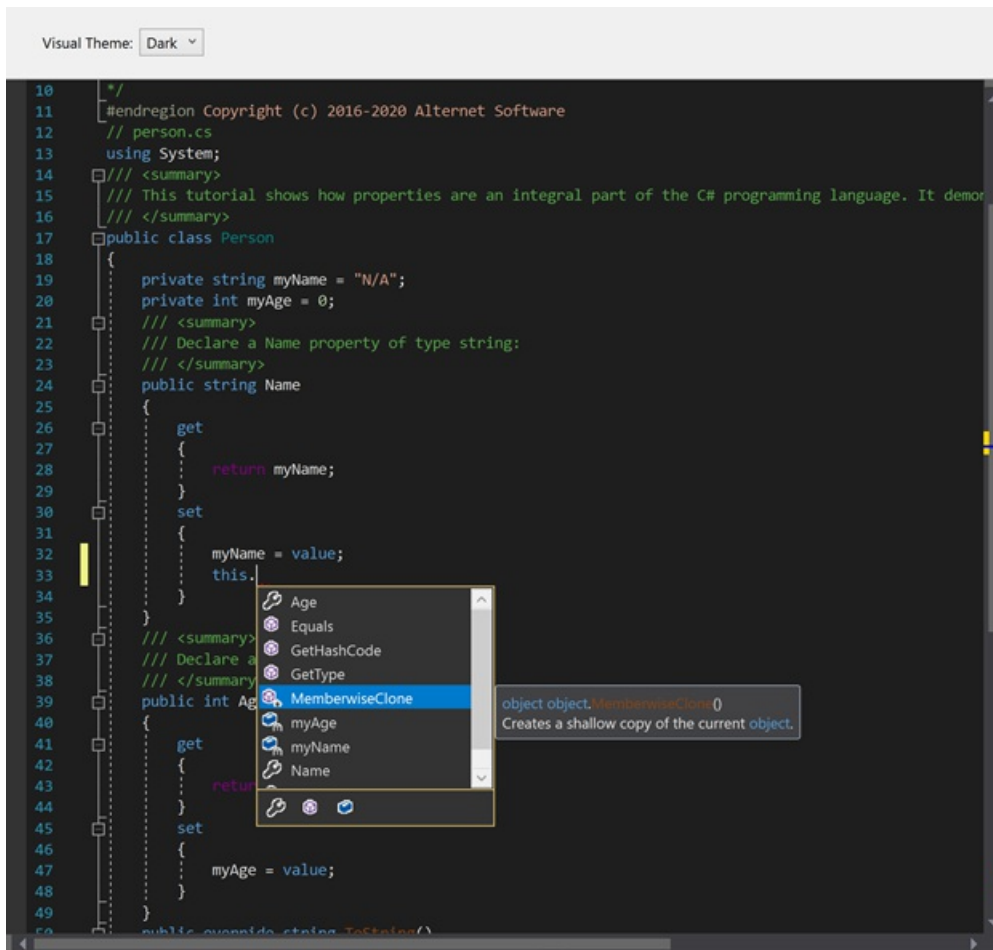
- [SmoothScroll](#) - if set, the display is updated as the user drags the scrollbar, otherwise the display is updated only when the user releases the scrollbar thumb. Disabling this option may improve performance on slow machines.
- [ShowScrollHint](#) - if set, a hint window, showing the new number of the topmost string, is displayed whenever the user drags the scrollbar.
- [UseScrollDelta](#) - if set, editor window content is scrolled by several characters when caret becomes invisible rather than one character
- [SystemScrollbars](#) - if set, system scroll bars are displayed, otherwise custom scrollbars are used.
- [FlatScrollbars](#) - if set, scroll bars are displayed in flat style. This option works only if *SystemScrollBars* is on.

- [AllowSplitHorz](#) - allows displaying horizontal splitting buttons in the scroll area. This option works only if *SystemScrollBars* is off and control has Dock property set to *DockStyle.Fill*.
- [AllowSplitVert](#) - allows displaying a vertical splitting button in the scroll area. This option works only if *SystemScrollBars* is off and control has Dock property set to *DockStyle.Fill*.
- [HorzButtons](#) - allows displaying additional buttons in the horizontal scrolling area. This option works only if *SystemScrollBars* is off.
- [VertButtons](#) - allows displaying additional buttons in the vertical scrolling area. This option works only if *SystemScrollBars* is off.
- *VerticalScrollBarAnnotations* - allows displaying scroll bar annotations that show special items such as line modifications, syntax errors, search results bookmarks and the caret position, throughout the entire document within the scroll bar. Individual annotation kinds are controlled by [Annotations](#) property.



Visual Themes

Visual themes allow to change the appearance of all graphical elements in the editor by setting [VisualThemeType](#) or [VisualTheme](#) type properties. Light and Dark visual themes are included, and custom appearance can be configured via custom visual theme.



Gutter

The gutter is the area to the left of the text, the purpose of which is to display miscellaneous indicators for the corresponding lines of text. Among these indicators are bookmark indicators, line wrapping indicators, line styles icons, line numbers, outlining buttons and line modification markers.



All the images displayed in the gutter are contained in the gutters image list. The following code gives an example of how to add a custom icon to this list from another image list (for example, the one dropped on the form during design-time):

```
edit.Gutter.Images.Images.Add(imageList1.Images[0]);
```

The mechanism of the line styles icons allows you to define how certain lines of text will be displayed.

The most common use for this is the indication of breakpoint lines and of the current execution point.

For example, the following code defines the style to be used for breakpoints.

```
style_id = edit.LineStyles.AddLineStyle("breakpoint",
    Color.White, Color.Red, Color.Gray, 11, LineStyleOptions.BeyondEol);
```

(Note, in the current version, image # 11 corresponds to the built-in breakpoint indicator image, and #12 corresponds to the current execution point image.

Later on, some line of the text can be assigned the style:

```
edit.Source.LineStyles.SetLineStyle(line_no, style_id);
```

(Note, that here and in the other places of this document line numbers start at 0.)

Note: at any given time, every line can have at most one style. If you need to remove line style for some particular line, call:

```
edit.Source.LineStyles.RemoveLineStyle(line_no);
```

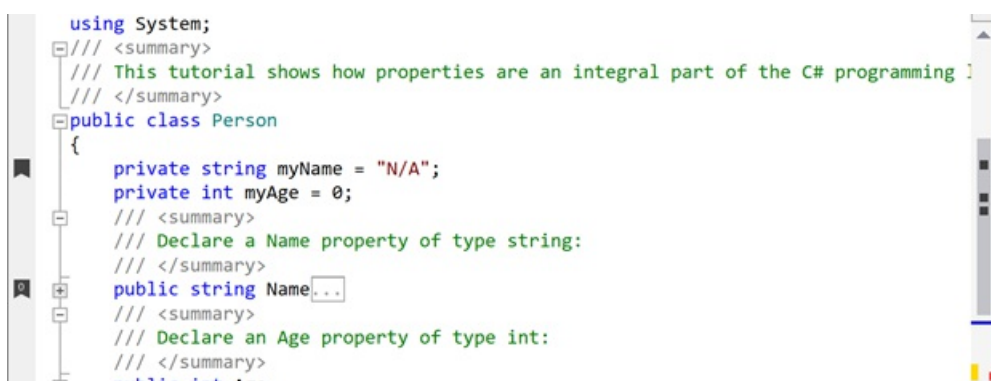
For [TextEditor](#) control appearance of the gutter is controlled by the following properties: [Width](#), [BrushColor](#), [PenColor](#) and [Visible](#). [Width](#) property specifies width of the gutter area, [BrushColor](#) specifies background color of the gutter area, [PenColor](#) specifies color of the gutter line, and [Visible](#) indicates whether or not to draw gutter. Note that gutter can adjust its width if line numbers or outlining is on and painted on the gutter. [TextEditor](#) allows drawing line numbers to visually indicate position of the visible lines inside the document. To enable line numbers you need to set [PaintLineNumbers](#) to true. Turning [PaintLinesOnGutter](#) option on enables drawing line numbers on gutter area, turning it off causes line numbers to be painted immediately after the gutter area. Appearance of line numbers are controlled by the [IGutter](#)'s properties: [LineNumbersStart](#), [LineNumbersForeColor](#), [LineNumbersBackColor](#), [LineNumbersAlignment](#), [LineNumbersLeftIndent](#) and [LineNumbersRightIndent](#), which are intuitively understandable.

Like Microsoft Visual Studio editor, [TextEditor](#) provides the ability to visually track modified lines. To enable this feature you need to turn [PaintLineModifiers](#) on. When [LineModifiers](#) are on they indicate lines that were changed since last saving. New changes are marked with Yellow color; changes that were done before last saving are marked with Lime color. Colors can be customized using [LineModifierChangedColor](#) and [LineModifierSavedColor](#) properties.

Reaction to mouse clicks and double-clicks on the gutter area can be implemented by assigning handlers to the [GutterClick](#) and [GutterDbClick](#) events.

Bookmarks

Just as with often used reference books, the process of navigating the text can be made more efficient with the usage of bookmarks. Two kinds of bookmarks are supported by the [TextEditor](#): plain and numbered. The former can be toggled for the current line using the [Ctrl+K Ctrl+K](#) key combination sequence, and can be navigated in cyclical manner using the [Ctrl+K Ctrl+N](#) (next bookmark) or [Ctrl+K Ctrl+P](#) (previous bookmark). The numbered bookmarks have a different flavor: there can be up to ten bookmarks, each having a number associated with it.



Toggling the numbered bookmark is performed using the [Ctrl+K Ctrl+#](#), and navigation to the specific bookmark is performed by pressing the [Ctrl+#](#) key combination (where # is any of the digits from 0 to 9). There can be only one plain bookmark in any line. Numbered bookmarks do not have such a limitation, however, only the indicator for the first bookmark in the line will be displayed in the gutter area, if [PaintBookMarks](#) is set to true.

Like most other things in the editor, bookmarks can be manipulated programmatically. Note that the list of bookmarks belongs to the text source, so multiple views of the same source share the same set of bookmarks.

The following code snippet sets the plain bookmark at the current position:

```
System.Drawing.Point pos = edit.Position;
edit.Source.BookMarks.SetBookMark(pos, int.MaxValue);
```

To set the numbered bookmark, replace *int.MaxValue* by the bookmark number (0..9).

To clear all the bookmarks set in the text source, call the [ClearAllBookMarks\(\)](#) method:

```
edit.Source.BookMarks.ClearAllBookMarks();
```

Navigating to the location defined by a particular bookmark can be performed as follows:

```
edit.Source.BookMarks.GotoBookMark(index);
```

Code Editor supports named bookmarks with description and hyperlink. The user may see a description in a tooltip window when moving the cursor over the bookmark, and load the browser with specified url when clicking on the bookmark. Such bookmarks can be set using the following code:

```
edit.Source.BookMarks.SetBookMark(edit.Position, 0,
    "Bookmark1", "This is Named Bookmark", "www.alternetsoft.net");
```

If you need to have custom images, you can change the bookmark indicator images by assigning custom image list:

```
edit.Gutter.BookMarkImageIndex =
edit.Gutter.Images.Images.Count;
edit.Gutter.Images.Images.Add(imageList1.Images[0]);
```

(This code uses the first image from the *imageList1*, which you could, for example, create by just dropping a new Image List from the toolbox on the form. For more examples on working with the gutter, refer to the corresponding section of this manual.)

You can configure bookmarks navigation to work across multiple documents. These documents should be added to the *BookMarkManager* class, and every document should have the *FileName* property assigned.

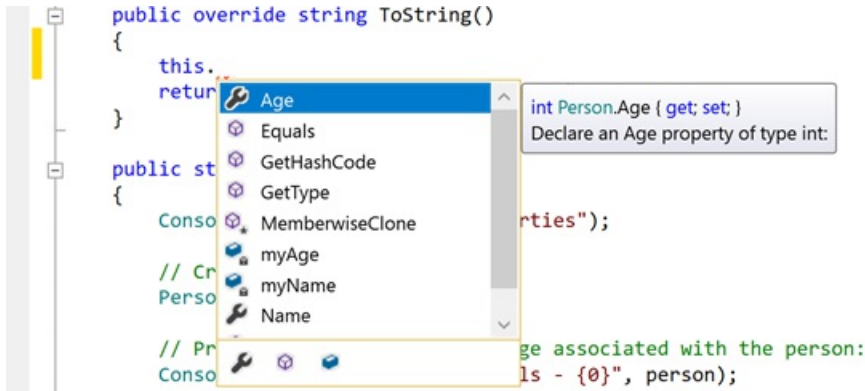
```
BookMarkManager.Register(edit.Source);
BookMarkManager.SharedBookMarks.Activate += new
EventHandler<ActivateEventArgs>(DoActivate);
private void DoActivate(object sender, ActivateEventArgs e)
{
    foreach (var edit in editors.Values)
    {
        if (edit.Source.FileName == e.FileName)
        {
            ActivateEditorTab(editor);
            break;
        }
    }
}
```

In this mode all bookmarks will be stored in a global list inside *BookmarkManager* instead of every individual *TextEditor* control allowing global navigation through them.

Code Completion (Intellisense)

Although the main purpose of an editor is to be a convenient tool for the user to enter the text, quite often a guidance from the editor can significantly improve the effectiveness of the work process. When editing a text which has some structure (i.e. computer program in some language), there are often well-defined sets of input possibilities in certain contexts. For example, for many programming languages, the sequence "someobject." should be followed by one of the existing field names. To assist the user in such situations, the text editor can activate a popup list containing all the methods that can be accessed from the current

scope.



If there is a partial word immediately to the left of the current cursor position, the first entry that starts with that word is highlighted. The user can then continue typing up until the method which he meant is selected or just use up and down arrow keys to navigate the list, and then insert the complete method name by pressing the Enter key.

Automatic Code Completion Invocation

In most cases Code Completion list and Signature Help for method parameters are provided by parser, alongside with the list of characters, such as period "." or open parens "(", which invoke code completion automatically as user types. The task of code completion is to have the list of available choices to appear automatically as user types, for example after user types "someobject." the list of class members for that object is expected to appear, and after they type "somemethod(" the tooltip showing the list of parameters for that function is expected to appear. It can be customized to show those popups only if the user stops input for some short period of time after typing the activating symbol ("." or "(").

The automatic code completion is implemented by Roslyn C# and Visual Basic parsers, TypeScript/JavaScript parsers, as well as by Advanced C#, J#, Visual Basic, VBScript, JavaScript, C, XML and Python parsers.

For example, automatic code completion is attempted after typing a period (".") following a member (member access expression), typing an open brace ("(") following a member (invocation expression or object creation expression), typing a period (".") inside *using* section, typing less sign ("<") inside xml comments, etc. This feature is implemented as close as possible to the Visual Studio .NET editor, so it works in an intuitively understandable way. On top of that Roslyn-based parsers are configured to invoke code completion when the user starts typing identifiers.

When these parsers are used, you still can control some aspects of code completion, for example delay before code completion window appears, using the [NeedCodeCompletion](#) event, which will be discussed later. Moreover, for advanced parsers you can register your own types and objects, namespaces and assemblies for code completion using the `CompletionRepository` property of [SyntaxParser](#).

To make types from most commonly used assemblies such as `System`, `System.Drawing`, and `System.Windows.Forms` to be available for code completion, you can call the following method

```
csParser1.Repository.RegisterDefaultAssemblies();
```

If you need to provide code completion for assemblies declared in other assemblies, you need to register these assemblies this way:

```
csParser1.Repository.RegisterAssembly("System.Xml");
```

You may need to register types for code completion that are not declared in the assembly, but present in the form of source code somewhere else.

For Roslyn-based parsers you can rely on underlying solution/project/document object model:

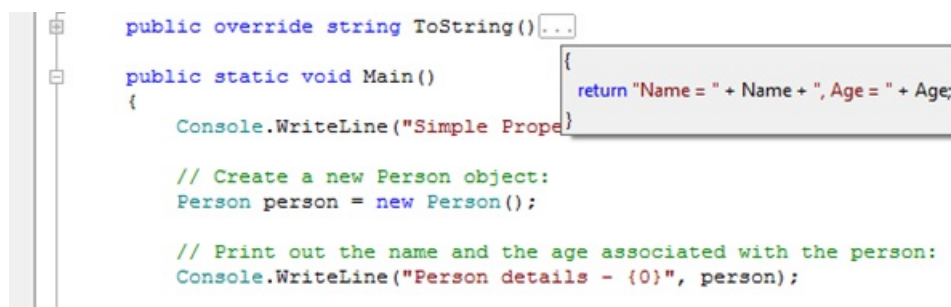
```
csParser1.Repository.RegisterCodeFiles(new string[] { "MyFile.cs" });
```

For one of advanced parsers you need first to create [SyntaxParser](#), load this file into the [Strings](#) object, and then add parsed [SyntaxTree](#) to the code completion repository. The following code demonstrates how it can be accomplished:

```
ISyntaxParser parser = new Alternet.Syntax.Parsers.Advanced.CsParser();  
parser.Strings = new TextStrings();  
parser.Strings.LoadFile("MyFile.cs")  
parser.ReparseText();  
csParser1.CompletionRepository.RegisterSyntaxTree(parser.SyntaxTree);
```

Code Outlining

The [TextEditor](#) control supports outlining, which is a text navigation feature that can make navigation of large structured texts more effective. The essence of *outlining* lies in defining sections of the text as structural units that once collapsed, can be visually replaced by a shorter representation, i.e. by ellipsis ("..."). During the text navigation the user can dynamically switch between the collapsed and complete representation of any particular section. Sections can be nested.



The section can be expanded by clicking on the "+" button, by double-clicking the proxy text, or by pressing the Ctrl+M Ctrl+M key sequence (in the default key mapping). The section can be collapsed by clicking on the "-" button, or by pressing the Ctrl+M Ctrl+M key sequence. All the sections can be globally collapsed or expanded using the Ctrl+M Ctrl+L key sequence.

Outlining is the property of the [TextEditor](#) control itself, not of the [TextSource](#), thus it is possible to have two views of the same text one with outlining and another without, or even to have completely different structural parts defined.

All the aspects of the *outlining* are controlled via the [Outlining](#) property of the [TextEditor](#). The *outlining* can be enabled or disabled using the [AllowOutlining](#) property either in design time or at runtime. The look of the outline is controlled by the [OutlineColor](#) and [OutlineOptions](#) properties.

There are two approaches to defining outline sections.

Direct Definition of Outline Sections

Outline sections can be explicitly defined by calling the appropriate methods of the [Outlining](#) property, i.e.:

```
edit.Outlining.Outline(new Point(0, 0), new Point(int.MaxValue, 0), 0, "...").Visible = false;
```

This code snippet defines the section of the first level consisting of the entire first line of the text, using ellipsis ("...") as the proxy text and being in a collapsed state.

While this approach is the simple one, it has one significant drawback: if sections represent structural units defined by the text itself, and the text can be edited by the user, sections have to be somehow constantly kept in sync with the text, which can be a non-trivial task.

Automatic Definition of Outline Sections Using the Syntax Parser

To provide automatic code outlining, the syntax parsing framework has to be employed. This approach may seem to be more complex at the first look, however it provides consistent results. To implement this approach, a class descending from the

[SyntaxParser](#) class needs to be defined, and the [Outline](#) method needs to be implemented. This method will be frequently called by the [TextEditor](#) whenever the text changes, so, to provide the user with a smooth editing experience, the implementation should be relatively fast.

Code Editor includes parsers that support automatic outlining for *C#, Visual Basic, J#, JavaScript, VBScript, Ansi-C, SQL, HTML, XML and Python* languages.

The following example demonstrates how to implement a parser the marks every line starting from the sharp ("#") sign as a separate outline section.

```
private void InitializeComponent()
{
    ...
    this.parser1 = new XParser();
    ...
}
public class XParser: SyntaxParser
{
    public XParser()
    {
        Options = SyntaxOptions.Outline;
    }
    public override int Outline(IList<IRange> Ranges)
    {
        Ranges.Clear();
        for(int i = 0; i < Strings.Count; i++)
        {
            if(Strings[i].ToString().StartsWith("#"))
            {
                Ranges.Add(new OutlineRange(
                    new Point(0, i),
                    new Point(int.MaxValue, i),
                    0, "...", false));
            }
        }
        return Ranges.Count;
    }
}
```

Code Editor Extended Features (WPF)

Code Editor provides advanced text editing functionality such as customizable keyboard mapping, HyperText handling, spell-checking integration, printing and exporting, macro recording and playback and miscellaneous display features.

Keyboard Mapping

While the [TextEditor](#) closely mimics the key-mapping common to most of Microsoft's products, it is completely customizable: you can add or change behavior of certain keys or even define an entirely different key-mapping.

To assign an action to some key combination, use the following code:

```
private void edit_Action()
{
    ...
}
...
edit.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new
KeyEvent(edit_Action));
```

This would make the *Ctrl+Alt+W* key combination execute the *edit_Action* method.

Or, to pass some object to the key handler:

```
private void edit_Action(object o)
{
    ...
}
...
edit.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new
KeyEventEx(edit_Action), some_object);
```

To remove some key handler, regardless of whether you have added it yourself, or it is the default one, call:

```
edit.KeyList.Remove(Keys.A | Keys.Control);
```

The code described before is used to manage the key handling in the default state. In fact, the key handling is slightly more complex than that: the [TextEditor](#)'s key handling mechanism can be in different states, other than the default one. Every state has its own key mapping table. Key mapping for bookmark operations can serve as a good example: after the user presses the *Ctrl+K* key combination, combinations *Ctrl+K*, *Ctrl+N*, *Ctrl+P*, *Ctrl+L* (the list is incomplete) obtain the new meaning. If a key combination is pressed for which there is no assignment in some non-default state, then the state is changed to default, and the combination is evaluated in the new context. [TextEditor](#) defines four different non-default states, but you can implement your own:

```
edit.KeyList.Add(Keys.W | Keys.Control, null, 0, 5);
edit.KeyList.Add(Keys.Tab, new KeyEvent(edit_Action), 5, 5);
```

This code creates a state that is activated by pressing the *Ctrl+W* key combination, and in which the *Tab* key causes the *edit_Action* to be executed. The state is changed back to default when the user presses some key other than the *Tab*. Up until now we have only examined the cases where you add some new functionality, or suppress some existing one. There also might be a case, when you want to use an entirely different key mapping, for example, to simulate some other environment your users are familiar with. To accomplish this, it is necessary to completely clear the current key mapping, and then to assign every function performed by the editor to some key. Note, that this really means every function: even such trivial things as cursor navigation and insertion of a new line are performed according to the key mapping.

For example, the following code assigns the editor's key-mapping to a single action defined: "Select All", which is assigned to the

Ctrl+X key combination

```
edit.KeyList.Clear();
edit.KeyList.Add(Keys.X | Keys.Control,
((EventHandlers)edit.KeyList.Handlers)SelectAllEvent);
```

URL handling

The [TextEditor](#) can be set up to handle pieces of text that look like some kind of an URL by setting the [HighlightHyperText](#) property to true. The handling consists of highlighting those pieces of text, and of processing clicks on them. By default, clicking the URL causes the operating system default action to be performed (i.e. launching a browser or an email client), however, you can override this behavior by assigning the [JumpToUrl](#) event handler.

```
private void edit_JumpToUrl(object sender, UrlJumpEventArgs e)
{
    if(is_our_url(e.Text))
    {
        process_url(e.Text);
        e.Handled = true;
    }
}
```

Spellchecker Interface

The [TextEditor](#) supports the spell-as-you-type spell checker integration. To enable spelling for the editor, set its [CheckSpelling](#) property to true and assign the WordSpell event handler.

The following artificial example considers any word longer than 3 characters to be correct:

```
private void edit_WordSpell(object sender, WordSpellEventArgs e)
{
    e.Correct = e.Text.Length > 3;
}
...
this.edit.WordSpell += new WordSpellEvent(this.edit_WordSpell);
```

Incorrect words are displayed with the wiggly underline (the default color is red, but it can be changed using the [SpellColor](#) property). In real-life scenarios you would need to use some third-party software/dictionary to really check the text. Another alternative would be using some word-list file, many of them, including Public Domain or free ones, can be found on the Internet. Refer to a Miscellaneous quick start project, which has one of these dictionaries.

Another useful feature supported by [TextEditor](#) is AutoCorrect, allowing you to auto correct words when typing. To enable this feature you need to set property AutoCorrection to true and handle the AutoCorrect event to provide replacements for words that were typed incorrectly.

Printing and Exporting

[TextEditor](#) includes support for printing, print previewing, and exporting to RTF and HTML.

Exporting can be performed as simple as this:

```
edit.SaveFile(FileName, new RtfExport());
```

Printing tasks are performed and configured via the Printing property of the [TextEditor](#).

For example, to show the print preview dialog, call:

```
edit.Printing.ExecutePrintPreviewDialog();
```

[TextEditor](#) control supports adding user-defined information while printing.

To add some text to the footer:

```
edit.Printing.Footer.CenterText = "draft";
```

Text in headers and footers can contain substitution tags. The standard ones are: [page], [pages], [date], [time] and [username].

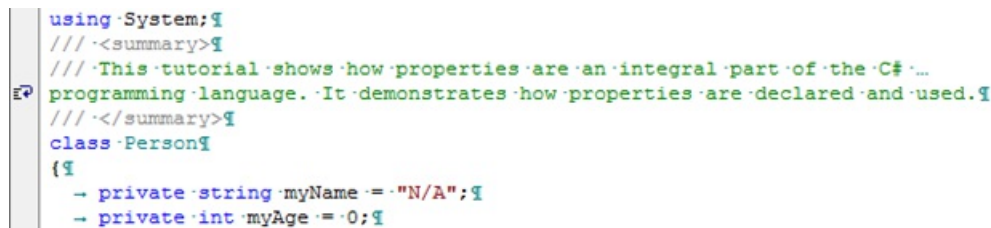
Marco Recording and PlayBack

[TextEditor](#) has macro recording and playback capabilities. It allows recording sequences of keyboard commands and playing them later. Note that mouse input is not recorded.

This feature enables you to store a set of frequently used editing commands. Set *MacroRecording* property to start/finish macro recording. Use the *PlayBack* method to repeat the stored command sequence.

White-space Display

It is sometimes desirable for the user to see the codes which influence the layout of the text and are normally invisible themselves. These codes are space, tab, end-of-line, and the end-of-file (not really a code), and are often collectively referred to as the white-space. The [TextEditor](#) has the option to display them, and to control their appearance.



```
using System;
/// <summary>
/// This tutorial shows how properties are an integral part of the C# ...
programming language. It demonstrates how properties are declared and used.
/// </summary>
class Person
{
    private string myName = "N/A";
    private int myAge = 0;
```

The display of the white-space is enabled using the [Visible](#) property. The color used to display white-space codes is determined by the [SymbolColor](#) property, and the characters used to display those codes are determined by *EofSymbol*, *EolSymbol*, *SpaceSymbol*, and *TabSymbol* properties.

Line Separator

It is possible to have lines of the editor to be separated by thin horizontal lines, and to have the current line highlighted. This behavior is controlled by the [LineSeparator](#) property.

The following options are available:

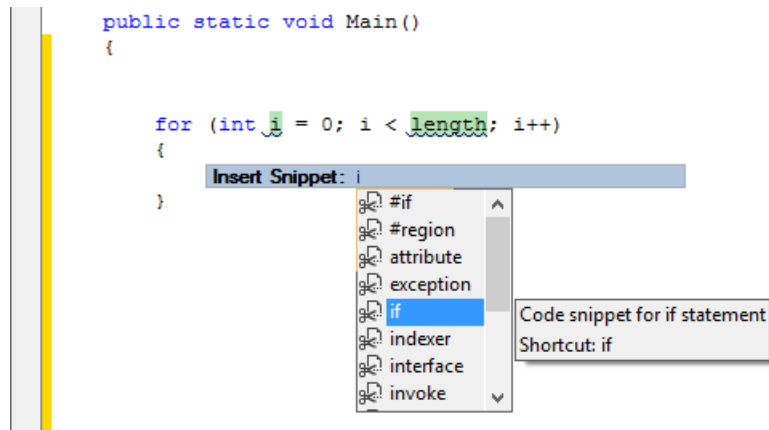
- [HighlightCurrentLine](#) specifies that the current line in the editor will be highlighted using the [HighlightColor](#) for background.
- [HideHighlighting](#) specifies that the highlighting of the current line should be hidden when the editor loses focus.
- [SeparateLines](#) specifies that a thin horizontal line of [LineColor](#) should be drawn between each line of text.
- [SeparateWrapLines](#) specifies that each visual line of text produced as a result of word-wrap should be separated in the same manner as separate lines (works only if the [SeparateLines](#) option is also specified).
- [SeparateContent](#) specifies that line separator will be drawn between sections of the code (for example between methods), if [TextEditor](#) control is associated with [SyntaxParser](#) supporting this feature.

Code snippets

The code snippets are the next code completion provider, allowing to insert frequently used fragments of code. Code snippets can be inserted into the editor by pressing Tab key after snippet shortcut or by executing code snippet popup window with Ctrl + K + X key sequence, or activated programmatically, by calling the CodeSnippets method of the [TextEditor](#).

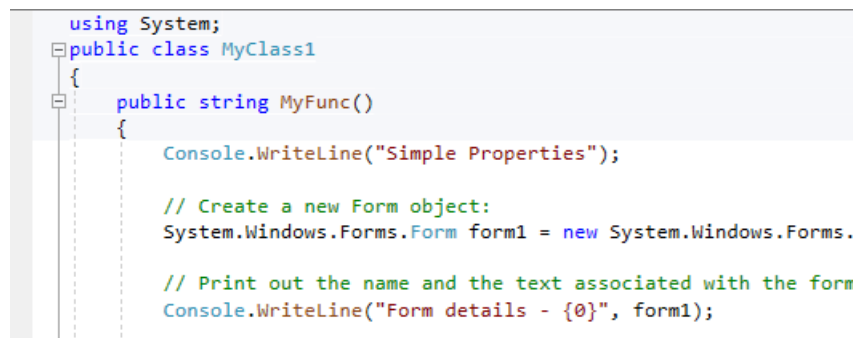
The purpose of the code snippets is to permit the user to quickly enter one of the predefined fragments of text. If the code snippet has fields declared, the editor allows modifying their values causing updating field values inside the whole snippet.

The following picture illustrates the usage of the code snippets.



Hidden and Read-Only Lines

[TextEditor](#) control can mark certain lines to be readonly or hide them at all so the user can't see them. This can be achieved by using [SetLineHidden](#) and [SetLineReadOnly](#) methods. For hidden lines to take effect, the [AllowHiddenLines](#) property needs to be set to true. Read-only lines can be made visually different from editable lines by setting [ReadOnlyBackColor](#) property. Sometimes it's required to mark certain lines to be both hidden and readonly, this way they can not be deleted if the user selects the outer block containing them and tries to delete it.



Structure GuideLines

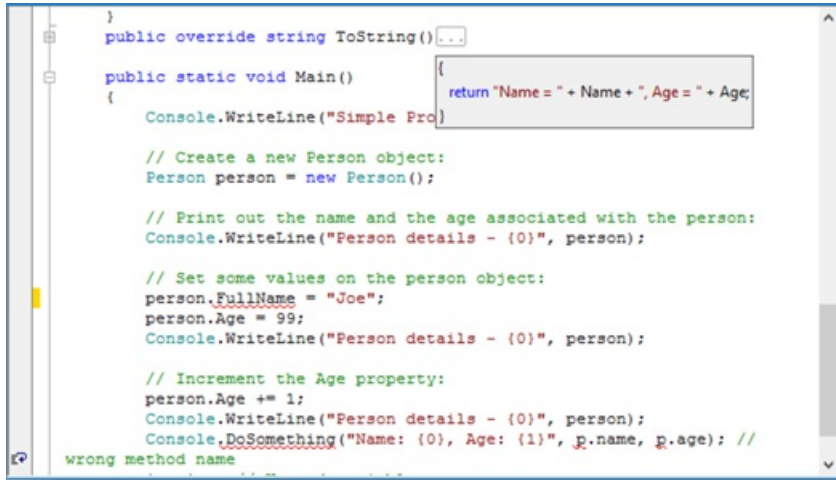
[TextEditor](#) control can display dashed lines between syntax blocks for some parsers (Roslyn-based, TypeScript and some advanced parsers), helping the user to better understand the structure of the document being edited. This behavior is controlled by a Parser and can be switched off by the StructureGuideLines parser option.

```
/// </summary>
public int Age
{
    get
    {
        return myAge;
    }
    set
    {
        myAge = value;
    }
}
public override string ToString()
{

```


Syntax Parsing

Text parsing is performed by one of the [Parser](#) non-visual components. In the very basic version it controls the syntax highlighting, and in case of using more advanced parsers it enables additional features such as code completion, code outlining, code formatting and syntax error underlining.



If no parser is assigned, [SyntaxEdit](#) or [TextEditor](#) do not perform any parsing related functions. To be able to use those features you need to explicitly create a [Parser](#) component and assign it via the `Lexer` property of either these controls directly, or their `TextSource`.

C#/VisualBasic (Roslyn) and TypeScript/JavaScript parsers

The package includes parsers that are based on Microsoft Code Compiler technology (Roslyn) and Microsoft TypeScript compiler.

These parsers allow to have full syntax and semantic model of the text being edited by `SyntaxEdit` control, which enables additional features such as code completion, code outlining, code formatting, highlighting types in different colors and underlying syntax and semantic errors and warnings to be identical to the ones found in Microsoft Visual Studio editor.

LSP Parsers

LangServer-based parsers rely on external servers to provide features like auto complete, go to definition, find all references and alike. The Code Editor package includes parsers based on this technology for C/C++, Java, Python, Lua XML, and PowerShell. There are two variations of each parser - one that relies on the Language server to be installed on the target machine and one that includes all required payload (such as clang libraries for C/C++ or embedded python distribution) in the form of embedded resources. Java embedded parser contains LSP-server files, but not Java installation itself, which needs to be installed on target machines independently.

Advanced Parsers

The Code Editor package comes with several advanced parsers, each one designed to perform syntax highlighting for certain languages. Each of these parsers is derived from the [SyntaxParser](#) class implementing [ISyntaxParser](#) interface and performs syntax analysis of the text in a specific programming language in order to provide advanced code editing features discussed above. These parsers use hard-coded parsing algorithms instead of generic regular-expression based rules, which makes them significantly faster compared to generic parsers (these parsers will be explained further). Currently we have advanced parsers for the following languages: Python, C#, J#, Visual Basic.NET, Ansi-C, VBScript, JavaScript, HTML, SQL, T4, XML and XAML. These parsers perform complete syntax parsing of the source code to build the syntax tree, which is used to implement all mentioned features. Please note that these parsers might not support full language specification, especially language constructs which were added to these programming languages recently.

Python and XAML parsers are implemented in their own namespaces/assemblies, [Alternet.Syntax.Parsers.Python](#) and

[Altenet.Syntax.Parsers.XAML](#) respectively.

For Python/IronPython we have implemented full semantic analysis of the text, which builds a semantic model of the whole text displayed in the editor (and also processes included files). This approach was inspired by studying Microsoft Code Analysis ("Roslyn") implementation, which will be explained below. Semantic model is then used by Code Completion services and for finding declarations and references.

Generic parsers

Writing your own parser to comply with a full specific programming language is non-trivial task, not to mention of keeping it up to date with full language specifications as the language evolves, however for a simple task of syntax highlighting it's normally not required. To perform syntax highlighting you can rely on a generic parsing engine, which is using finite-state automaton rules driven by regular expressions matching the parsed text. Although creating good syntax-highlighting rules for some complex language can still be tricky, you will rarely, if ever, have to do it yourself. The Code Editor is supplied with more than 30 ready-to-use syntax schemes for the most commonly used modern programming languages. In a rare case you need to implement a parser for some custom language, there is a big chance that one of these parsers can be a good starting point. Syntax schemes are stored on disk as .xml files and the built-in visual design-time parser editor is provided to simplify the process of their creation. In case you need most of these languages in your application, you can consider using one of the [SchemeParser](#) descendants which contains appropriate language schemes in the form of an assembly resource.

Creating a new generic parser is described in the [Advanced Topics](#) section of the documentation.

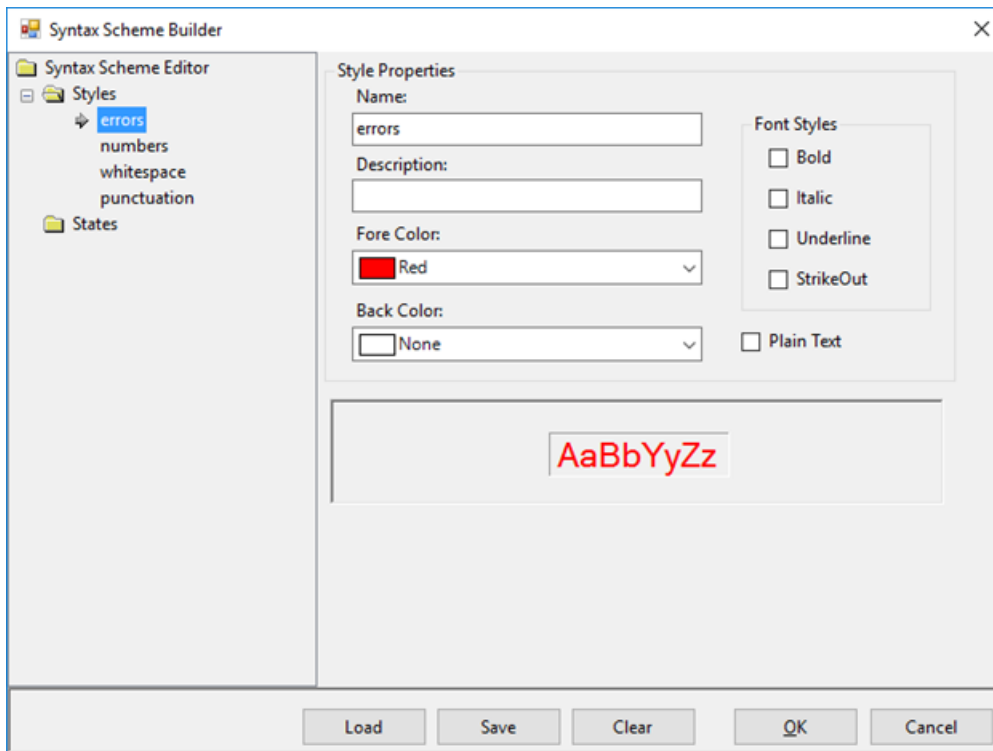
Creating own Generic Parser Schema

Although the Code Editor is supplied with a collection of parsers, it may be sometimes necessary to create a new one. In this chapter we will develop a completely new parser for some trivial fictitious language.

First, let us informally describe the language we are willing to parse. The valid text in this language consists of zero or more groups, enclosed in curly brackets ("{" , "}"), each containing zero or more numbers separated by commas. We want to distinctly highlight punctuation symbols and numbers, and to highlight erroneous input.

The first step is to create a new *Parser* object by dropping it from the toolbox, and assigning it to some [SyntaxEdit](#) by picking the newly created parser from the list of choices appearing for the *Lexer* property of the editor.

After having done this, we can start exploring the *SyntaxBuilder*. It is invoked by pressing the "..." button appearing for the *Scheme* property of the *parser* object.



Now you should see the *SyntaxBuilder* dialog box appears. If you wanted to use some existing scheme, you would have pressed the *Load* button, however, this time we are going to create a new scheme completely from scratch. After completing it, you can press the *Save* button to make it possible to use this new scheme in other projects.

The next thing to do, after entering optional information about the author and the copyright, is to define syntax highlighting styles used in the scheme. This is accomplished by clicking the right mouse button on the *Styles* node to bring the context menu, and choosing the *Add Style* command.

After creating a style you should give it a name and define its visual attributes. For this example we will need four three styles: *number*, *punctuation*, *whitespace*, and *error*. Let us define numbers to have olive color and italic text style, punctuation symbols to be blue, and errors to have red background and white foreground. The *whitespace* style is defined as having no distinct markup at all.

Then we define the states of the parser. For our example language there will be two states: *default* and *block*. States are defined similarly to styles, by choosing the *Add State* command in the context menu appearing to the *States* node. In turn, states contain syntax blocks, created by the *Add Syntax Block* command from the context menu of a state.

The syntax parser is essentially a state machine, driven by the text. Transition conditions are expressed in terms of regular expressions which are checked against the parsed text at the current position up to the next end of line. Expressions are tried in the syntax block definition order. The first successful match determines the syntax block. The text position is advanced by the length of the match, and the text is assigned the style specified for that syntax block. The matched text is additionally matched against the list of the reserved words associated with this syntax block, and if a match occurs, the style defined by the *ResWord Style* is used instead of the one defined by the *Style* property. The state of the state machine is changed according to the *Leave State* property of the syntax block, which can specify any of the states, including the same state, in which the syntax block resides, meaning no state transition is to take place.

The state machine for the language we are parsing is described in the following table, and deserves some comments.

The *whitespace* syntax block is only necessary because of the presence of a match *all error* syntax block. In the more common case where no error highlighting is used, no style (which is the same as the *whitespace* style that we have defined) would be used for the text that does not match any of the syntax blocks. The *error* syntax block is the last in the sequence and matches a single character which has not been matched by any of the preceding rules. The *block* syntax block is matched when the opening curly bracket is met. The bracket itself is assigned the *punctuation* style, and the state machine changes its state into the *block* state (note that state name, style name and syntax block style name coincidences are not required).

In the *block* state, the *whitespace*, and *error* syntax blocks serve the same purpose as in the *default* state. *Number* and *comma* syntax blocks cause numbers and commas to have the corresponding styles, and the *end* syntax block, which matches the closing curly bracket, causes the transition back to the default state.

State	Syntax Block	Regular Expression	Style	Leave State
Default				
	whitespace	\s+	whitespace	<i>default</i>
	block	\{	punctuation	block
	error	.	error	<i>default</i>
Block				
	whitespace	\s+	Whitespace	<i>block</i>
	number	\d+	Number	<i>block</i>
	comma	,	Punctuation	<i>block</i>
	End	\}	Punctuation	default
	Error	.	Error	<i>block</i>

Automatic Code Completion for arbitrary programming language

If there is no [SyntaxParser](#) for your language, you can consider implementing automatic code completion using [NeedCodeCompletion](#) event:

```
private void edit_NeedCodeCompletion(object sender, Altnet.Syntax.CodeCompletionArgs e)
{
    if ((e.CompletionType == CodeCompletionType.ListMembers) ||
        (e.CompletionType == CodeCompletionType.CompleteWord) ||
        ((e.CompletionType == CodeCompletionType.None) && (e.KeyChar == '.')))
    {
        // Look at Manual Code Completion, list members section
        ...
        e.Interval = (e.CompletionType != CodeCompletionType.None)
            ? 0 : 500;
    }
    if(e.CompletionType == CodeCompletionType.ParameterInfo ||
        e.CompletionType == CodeCompletionType.None &&
        e.KeyChar == '(')
    {
        // Look at Manual Code Completion, parameter info section
        ...
        e.Interval = (e.CompletionType != CodeCompletionType.None)
            ? 0 : 500;
    }
}
```

Depending on the kind of the language you are working with, and whether you are using some complete library to work with that language, or do everything yourself, the actual information on symbols will be retrieved in different ways:

- if you are using some third party library, look for something that resembles the name "Symbolic Information API" or like in the manual for that library;
- if you are developing your own language, or at least your own engine for some existing language, you probably already know what exactly to do to acquire the information necessary for code completion to work;
- If you are working with the .NET family of languages, CLR Reflection API should probably be of use for this purpose. The sample program supplied with the package provides a good starting point on working with it.

Manual Code Completion

If you use Code Editor with the parser that does not fully support automatic code completion, you can still provide some guidance to the users as he types by implementing some of the code completion logic manually.

Global Settings

If the application contains more than one instance of the editor, it is quite often desired to share their UI settings, and to provide the user with a centralized facility to manage them. Code Editor is shipped with Customize quick start project that demonstrates how this can be accomplished.

It includes *SyntaxSettings* class which is a holder for the following set of settings:

- The font used to display the text in the editor
- Syntax highlighting styles (i.e. foreground and background colors, font style)
- Whether the following features are enabled or not:
 - Show margin

- Show gutter
 - URL highlighting
 - Outlining
 - Word wrapping
 - Use of spaces instead of tabs for indents
- The width of the gutter area
 - The position of the margin
 - Tab-stop positions
 - Navigation options
 - Selection options
 - Outline options
 - Scrollbar options
 - Color Themes

To use this class, its instance must be created, i.e.:

```
private SyntaxSettings GlobalSettings;  
...  
GlobalSettings = new SyntaxSettings();
```

The settings can be retrieved from some particular [SyntaxEdit](#) controls as follows:

```
GlobalSettings.LoadFromEdit(edit);
```

And then assigned to some other editor like this:

```
GlobalSettings.ApplyToEdit(edit);
```

Settings can be easily stored to some file:

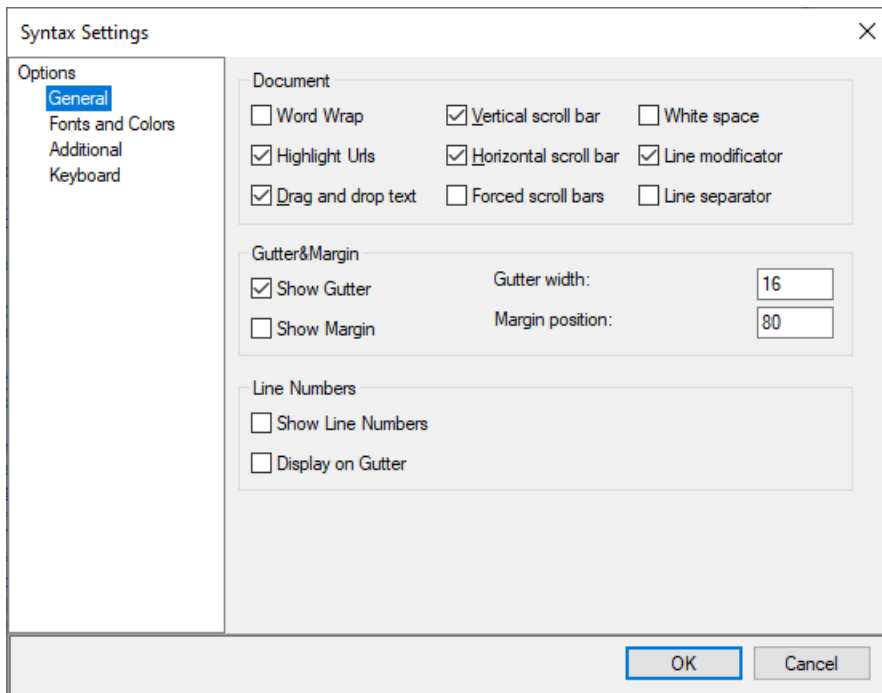
```
GlobalSettings.SaveFile("GlobalSettings.xml");
```

And later on, loaded from that file:

```
GlobalSettings.LoadFile("GlobalSettings.xml");
```

As the name of the file suggests, settings are stored in the XML format. Note, that in the real application you would check for the existence of that file, and also, this file should probably be located somewhere down the user's *Application Data* folder.

To make the handling of the global settings even easier, the *Customize* demo project includes an example settings dialog.



All you need to do to use it, is to declare and construct its instance:

```
using Alternet.Editor.Dialogs;
using Alternet.Editor.Wpf.Dialogs; // for WPF edition
...
private DlgSyntaxSettings Options;
...
Options = new DlgSyntaxSettings();
```

And later on, when the user requests the editor settings dialog perform something similar to the following:

```
Options.SyntaxSettings.Assign(GlobalSettings);
if(Options.ShowDialog() == DialogResult.OK)
{
    GlobalSettings.Assign(Options.SyntaxSettings);
    // for each syntaxEdit or TextEditor used in the application do
    GlobalSettings.ApplyToEdit(edit);
}
```

Localization of dialogs

All string constants used in dialogs are localized to a few foreign languages. CodeEditor supports dialog localization to German, French, Spanish, Russian and Ukrainian languages. The following code demonstrates how to switch to German language:

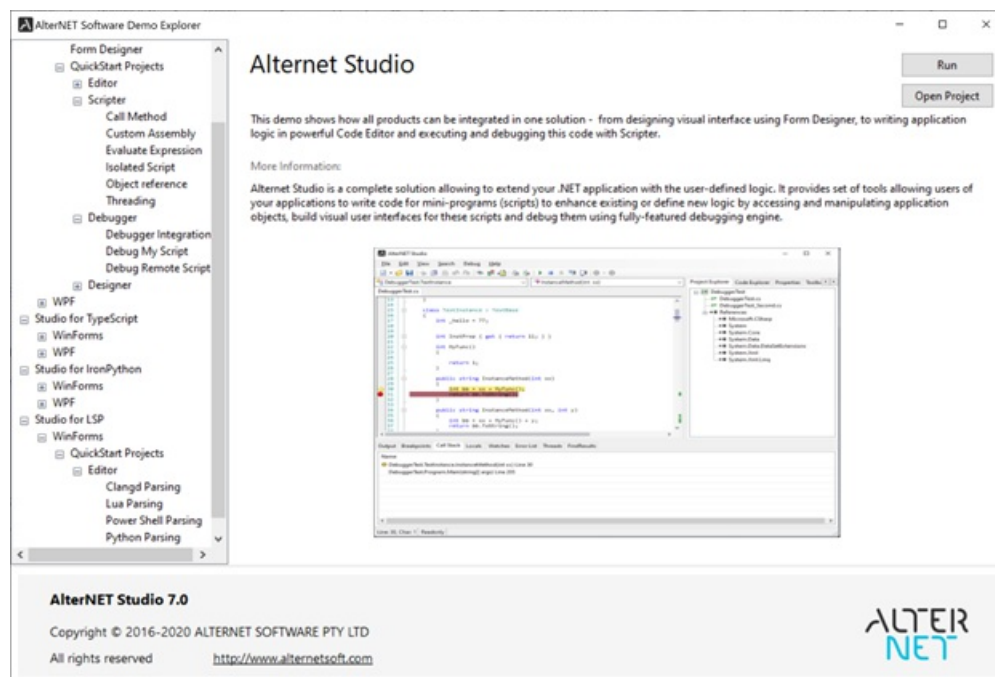
```
Using Alternet.Common;
...
CultureInfo oldcInfo = Thread.CurrentThread.CurrentUICulture;
Thread.CurrentThread.CurrentUICulture = new CultureInfo("de");
try
{
    StringConsts.Localize();
}
finally
{
    Thread.CurrentThread.CurrentUICulture = oldcInfo;
}
```

Scripter Overview

AlterNET Scripter is a component library designed to integrate C#, Visual Basic, TypeScript, JavaScript, Python and IronPython scripts into your WinForms and WPF .NET desktop applications. It allows extending the application logic by implementing custom functionality or automating custom tasks without recompiling and redeploying the application.

Script Execution

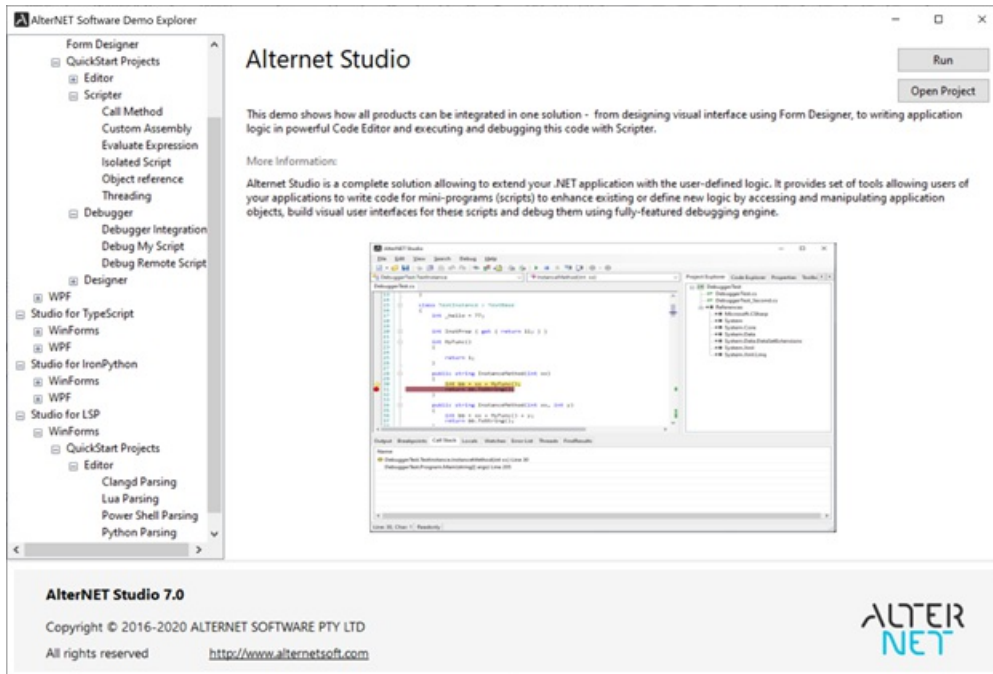
The main components that provide script executing functionality for supported programming languages are: [ScriptRun](#) for C# and VisualBasic; [ScriptRun](#) for Python; [ScriptRun](#) for IronPython and [ScriptRun](#) for TypeScript and JavaScript. These components encapsulate functionality of running standalone C#, Visual Basic, Python, IronPython, TypeScript and JavaScript script files or projects with forms and resources; they allow referencing third-party assemblies and register application-defined objects to be accessible in the scripts.



Script Debugging

The following components provide a fully-featured script debugging engine for the supported programming languages: [ScriptDebugger](#) for C# and VisualBasic; [ScriptDebugger](#) for Python; [ScriptDebugger](#) for IronPython; [ScriptDebugger](#) for TypeScript and JavaScript. [ScriptDebugger](#) for debugger that relies on Debugger Adapter Protocol.

These debuggers support Start, Stop, Break and Continue commands, step by step execution, breakpoints, expression evaluation, viewing local variables and watches, stack tracing and and in case of C#/VisualBasic, multiple thread debugging.



AlterNET Scripter includes a set of quick start projects, each one designed to highlight specific features of the component.

Below is brief overview of these projects, most of them available for C#/Visual Basic, Python, IronPython and TypeScript

AlterNET Studio - demonstrate how to run and debug script files and projects. Example files and projects are located in the demo\resources\debugger folders.

CallMethod - a set of quick start projects for all supported languages which demonstrate how to execute script methods and pass application objects to the script.

CustomAssembly - demonstrates how to use external assemblies in the scripts

EvaluateExpression - shows how ScriptRun can evaluate expression, which again can access some objects defined in the application

Object Reference - shows how application-defined objects can be accessed by bane from the script.

Threading - shows how scripts can be run asynchronously.

DebuggerIntegration - Shows how debugger logic can be integrated in the application to debug scripts (application-independent scripts in case of C#/Visual Basic).

For C#/Visual Basic only:

DebugMyScript - demonstrates how scripts executed by application can be debugged by a separate Script Debugger tool.

DebugRemoteScript - Shows how debugger logic can be embedded in the application to debug scripts that access the application API indirectly.

Isolated Script - Shows how to load a script in the separate AppDomain, so it can be unloaded afterwards and execute methods in it.

For Debugger Adapter Protocol (DAP):

CppLlvmDapDebugger - Shows how debugger logic can be integrated in the application to debug standalone C++ projects.

PythonDapDebugger - Shows how debugger logic can be integrated in the application to debug standalone Python projects.

Creating your first project

To see C# scripting in action, place the [ScriptRun](#) component on the form, and write the following code in Button click event handler:

```
scriptRun1.ScriptSource.FromScriptCode("public class ScriptTest { public static void Main() {  
System.Windows.Forms.MessageBox.Show(\"Hello World \");} }");  
scriptRun1.ScriptSource.WithDefaultReferences();  
scriptRun1.Run();
```

The first line of the code populates the Script source, the second line adds references to most common System assemblies, and the third one runs the code.

C# and Visual Basic Script execution

The basic script execution workflow requires setting a script source, adding references to the assemblies used in the script; registering application-defined objects accessible to the script; compiling script to dynamically-linked library or standalone executable program and running some method in that dll or executing the program.

Setting up Script Source

All properties and methods required to set a script source are encapsulated in [ScriptSource](#) property of the [ScriptRun](#) class; below are the most essential ones:

[Files](#) - specifies a collection of source files to be compiled and executed;

[ScriptCode](#) - specifies source in a form of text string;

[ProjectName](#), [ProjectFileName](#) and [RootNamespace](#) - contain project-related information if [ScriptSource](#) is loaded from the project.

[Imports](#) - contains global namespaces in case Visual Basic is used so you do not need to specify them in the code;

[Conditionals](#) - contains lists conditional compilation symbols;

[References](#) - contains a list of assembly references for types used in the scripts; this can include reference to the calling application.

[SearchPaths](#) - contains search paths to look for the third-party references in case they're not supplied with a full path.

[Resources](#) - contains a list of resx files with resources.

[FromScriptFile](#) - loads Script Source from the single source file;

[FromScriptCode](#) - loads script from code in a form of a text string;

[FromExpression](#) - sets [ScriptSource](#) to the string expression.

[FromScriptProject](#) - loads code from Visual Studio Project

Adding assembly references:

In order to use types in the script, assemblies where these types are declared need to be properly referenced.

The following code populates references with most commonly used assemblies:

```
scriptRun1.ScriptSource.WithDefaultReferences();
```

For technology set to WinForms (which is default option) it contains the following assemblies:

System, System.Drawing, System.WindowsForms;

Note that this list is different in case of .NET Core targets.

You can reference additional assemblies by adding it to the [References](#) property:

```
scriptRun1.ScriptSource.References.Add("System.Data");
```

This method accepts full path, you can also add reference to third-party assemblies in case script uses types from it.

Registering objects to be used in script

Application objects accessible by the script need to be added to the [GlobalItems](#) collection, along with the object's name which will be used in the script and object's type or object itself.

Object value itself is only required during script execution; for script compilation object name and type are sufficient.

[ScriptRun](#) adds references to the assemblies which contain types of the objects being added to [GlobalItems](#) automatically.

Note that [AssemblyKind](#) property needs to be set to Dynamically Linked Library, for it to be loaded in the running application process and be able to access application-defined objects.

Below is sample code which registers application-defined objects in the script.

```
public class MyItem
{
    public MyItem(string text)
    {
        this.Text = text;
    }
    public string Text;
}
scriptRun1.GlobalItems.Add(new ScriptGlobalItem("MyItem", obj: new MyItem("hello")))
```

Script Compilation and Execution

Once [ScriptSource](#) is set, next step is Compile the script; this step is performed implicitly when the script is run the first time, or when Script source is changed (this includes changes of script files externally)

Script compilation engine is implemented by [IScriptHost](#); there are two implementations of [IScriptHost](#) provided, a legacy engine based on [CodeDOMScriptHost](#) wrapper around command-line C# or Visual Basic compiler, or [RoslynScriptHost](#) based on new Microsoft Roslyn Code compiler technology - the last one is used by default and it allows some nice features such as referencing to other script source dynamically by using `#load` directive and gives more control on code parsing and compilation.

Script can be compiled into a dynamically-linked library or in a standalone executable; this is controlled by the [AssemblyKind](#) property. [GenerateModulesOnDisk](#) allows to control whether assembly being compiled will reside in memory or on the disk; and [ModulesDirectoryPath](#) specifies location of compiled assembly where compiled modules will be stored. Platform target (AnyCPU, AnyCpu32BitPreferred, x86, x64 or Auto) is controlled by Platform property (by default it's set to Auto and takes target platform from the application).

Once Compilation is executed, `Compiled` property will be set to true in case compilation was successful, and [IScriptHost](#)'s [ScriptAssembly](#) property will point to the assembly being compiled from the script source. Otherwise [IScriptHost](#)'s properties [CompileFailed](#) will be set to true and [CompilerErrors](#) will be populated with compiler errors. Please note, [CompilerErrors](#) may contain compiler warnings even in case of successful compilation.

Upon successful compilation you can subsequently call [Run](#), [RunMethod](#), or their asynchronous variants: [RunAsync](#) and [RunMethodAsync](#); in case of standalone executable [RunProcess](#) should be used instead.

Python and IronPython Script Execution and Debugging

Scripter contains a set of components that implement python script compilation, execution and debugging. These components are [ScriptRun](#) and [ScriptDebugger](#) for Python and [ScriptRun](#) and [ScriptDebugger](#)

These components are installed on the AlterNet Scripter.Python and AlterNet Scripter.IronPython tabs in the Visual Studio.

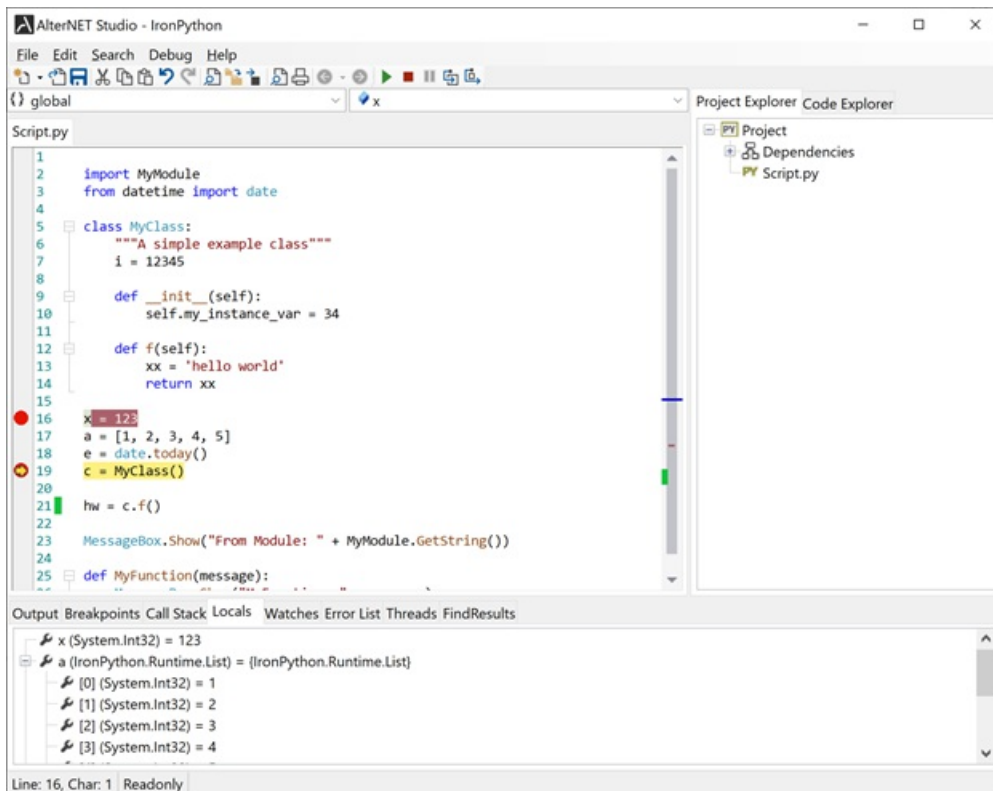
Script execution is based on the Python.NET and IronPython open-source scripting engines which gives Python programmers seamless integration with .NET. These engines support executing Python code and accessing .NET types and objects of the host application from the script.

[ScriptRun](#) and [ScriptRun](#) provides a very similar interface to .NET [ScriptRun](#); former one uses Python itself to execute Python scripts, while latter one creates in-memory .NET assembly out of Python code.

[ScriptRun](#) and [ScriptRun](#) can execute single files, Python projects (which can be loaded/saved to .pyproj file), or evaluate Python expressions.

The main difference between Python.NET and IronPython scripting engine is that Python.NET supports up to Python 3.7 language specification and can use most of the third-party libraries like numpy or pandas that rely on Python/Cython code, while IronPython supports up to Python 2.7 language specification. It also provides multi-threading script execution capabilities, unlike Python.NET which can not execute scripts in the multiple threads simultaneously.

[ScriptDebugger](#) and [ScriptDebugger](#) are based on Microsoft.Scripting debugging engine; they allow incorporating debugging logic in the same application and do not have a limitation of .NET debugger, which requires debugger and script to be debugged running in the separate application processes. It has most of the functionality that .NET Script debugger provides; except for multi-threaded debugging.



TypeScript/JavaScript Script Execution and Debugging

Alongside with component libraries for .NET and Python-based script compilation, execution and debugging, we provide a very similar components for TypeScript/JavaScript: [ScriptRun](#) and [ScriptDebugger](#)

These components are installed on the AlterNet Scripter.TypeScript tab in Visual Studio.

Script execution is based on Microsoft ClearScript which provides v8 high-performance open-source JavaScript engine. It supports executing JavaScript code and accessing .NET types and objects of the host application from the script.

[ScriptRun](#) provides a very similar interface to .NET [ScriptRun](#); the main difference is that it does not create .NET assembly, and executes JavaScript code using ClearScript engine.

The main difference in API is that unlike .NET Script Runner, the collection of referenced objects, types and .NET assemblies is specified via [HostItemsConfiguration](#) property; as opposed to [GlobalItems/References](#) properties; [RunMethod/RunMethodAsync](#) are replaced with [RunFunction/RunFunctionAsync](#).

The following code adds references to most commonly used assemblies and registers RunButton to be accessible from the script:

```
```csharp scriptRun.ScriptHost.HostItemsConfiguration.AddSystemAssemblies().AddObject("RunButton", btNETFromScript);
```

<xref:Alternet.Scripter.TypeScript.ScriptRun> can execute single files, typescript projects (which can be loaded/saved **to json file**), **or** evaluate TypeScript/JavaScript expressions.

**\*\*Note\*\*** that order **of** TypeScript/JavaScript files **in** a project **is** important, as they get executed one by one.

TypeScript compilation service uses host **configuration** to automatically create **all** support files containing typescript definitions. The following line needs **to** be placed **on top of** user's script **to access** .NET types **and** objects from host **configuration**:

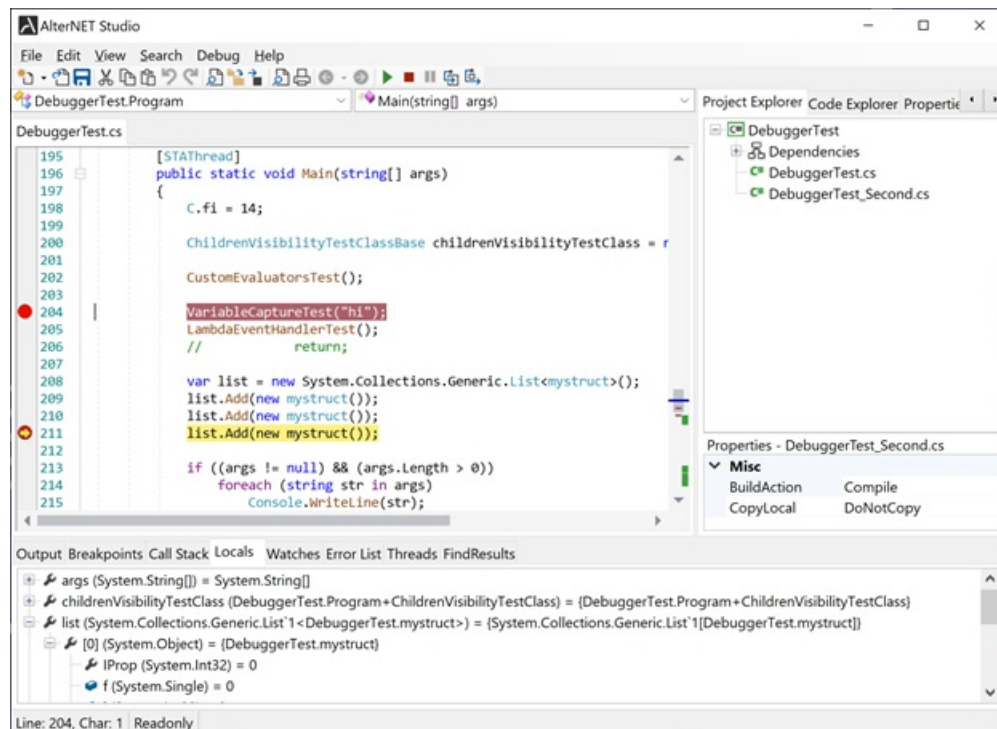
```
///<reference path="clr.d.ts" />
```

<xref:Alternet.Scripter.Debugger.TypeScript.ScriptDebugger> **is** based **on** Google Chrome debugging development tools; it allows incorporating debugging logic **in** the same application **and** does **not** have a limitation **of** .NET debugger, which requires debugger **and** script **to** be debugged running **in** the separate application processes. It has most **of** the functionality that .NET Script debugger provides; except **for** multi-threaded debugging **and** automatic retrieval/evaluation **of** local variables.

```
![TypeScriptDebugger](images/TypeScriptDebugger.png)
```

# Script Debugging

Lot of times script debugging is required for the script the users write. We provide tools to debug script code and a set of UI widgets to build custom debugging interfaces.



## C# and Visual Basic Script Debugging

Script Debugger engine is implemented in [AlterNet.Scripter.Debugger](#) assembly and it is based on CLR debugging COM interfaces low-level API to debug .NET applications.

[https://msdn.microsoft.com/en-US/library/ms404484\(v=vs.110\).aspx](https://msdn.microsoft.com/en-US/library/ms404484(v=vs.110).aspx)

Main component of Script debugging is the [ScriptDebugger](#) class, which provides all commonly used debugging features like step by step execution, stopping on breakpoints, examining local variables, expression evaluations, etc.

Below is a summary of [ScriptDebugger](#) most essential properties, methods and events:

**Methods:**

[StartDebugging\(\)](#) - starts executing the program from the entry point.

[AttachToProcessAsync](#) - Attaches to the already started process which scripts are to be debugged.

[StopDebuggingAsync](#) - Stops the debugging session.

[Break\(\)](#) - Causes the given process to pause its execution so that its current state can be analyzed.

[Continue\(\)](#) - Continues given process to the next breakpoint or until process finishes.

[StepInto\(\)](#) - Executes one statement of code; steps into the next function call, if possible.

[StepOver\(\)](#) - Executes one statement of code; steps over the next function call, if possible.

[StepOut\(\)](#) - Executes remaining lines of the function; steps out of the function currently being executed.

[ActivateThread](#) - Switches debugging to the specified thread.

[SwitchToStackFrame](#) - Switches debugging to the given stack frame.

[SetRunToPositionBreakpoint](#) - Causes the debugger to stop at the specified position.

Following methods may take considerable amount of time, therefore they're implemented asynchronously:

[EvaluateExpressionAsync](#) - Evaluates expression in the current stack frame, with or without child properties.

[EvaluateCurrentExceptionAsync](#) - Evaluates the exception being thrown by the debugger.

[GetStackFramesAsync](#) - Gets a list of method calls that are currently on a stack.

[GetThreadsAsync](#) - Gets a list of active threads.

[GetVariablesInScopeAsync](#) - Gets all local variables in the given stack frame.

[TrySetNextStatementAsync](#) - Sets the execution point to the specified line.

[GetExecutionPositionAsync](#) - Gets the current execution point.

Properties:

[IsStarted](#) - Indicates whether the debug process has started.

[State](#) - Gets current debugger state.

[ScriptRun](#) - in case Debugger used to debug standalone executable, contains all information required to compile and run the script.

[GeneratedModulesPath](#) - Specifies directory where assemblies for the scripts being debugged are located.

[Breakpoints](#) - Returns collection of debugger breakpoints.

[EventsSyncAction](#) - A function which could be provided by the application to sync raised debugger events if required (for example, perform `Control.Invoke`)

Events:

[ActiveThreadChanged](#) - Occurs when thread to be debugged changes.

[DebuggerErrorOccured](#) - Occurs when debugger encounters error during debugging session.

[DebuggingStarted](#) - Occurs when the debugging session is started.

[DebuggingStopped](#) - Occurs when the debugging session is stopped.

[ExecutionResumed](#) - Occurs when debugging is resumed after being paused.

[ExecutionStopped](#) - Occurs when debugging is paused.

[LogMessageReceived](#) - Occurs when a debug message is received.

[StackFrameSwitched](#) - Occurs when the debugger is switched to the stack frame.

[StateChanged](#) - Occurs when debugging state is changed (when debugger is started, stopped or paused)

## .NET Script Debugging best practices

The main issue that we've faced with debugging is that it's not quite possible to embed debugging logic in the same process where scripts are being executed, as the debugger process will need to freeze itself when debugging. Refer to the following blog for more details:

<https://blogs.msdn.microsoft.com/jmstall/2005/11/05/you-cant-debug-yourself/>



Therefore we see two main options for script debugging to work:

1. Script is compiled as a dynamically linked library and is linked to the calling application (which is the most straightforward way of scripts to be able to access application-defined objects). In this case Script debugger must be a separate process which attaches to the main application process and allows to debug script code in it. The script debugger can be made look like it belongs to the same application (which is outside of the scope of this tutorial), but it has to be in a separate process.

In this mode Script Debugger does not compile or execute script itself; instead it relies on the main application to do so. It receives the main application process id, source and project file along with the name of assembly to be debugged via command-line arguments; attaches to the main process and communicates with it by sending Start Debug or Stop Commands and receiving a list of compilation errors or script completion events.

Refer to [DebugMyScript](#) quickstart projects for more details.

Note that you cannot debug the main application under Visual Studio and have Script Debugger to attach to it at the same time, as Visual Studio will attach its own debugger.

Note that the target platform of debugger and debugee process need to be the same (for AlterNET Studio demo it's set to AnyCPU, 32-bit preferred).

1. Script to be compiled in the separate executable; and debugging logic is embedded in the application itself. This option requires either the script to be application-independent (which is not useful if scripts are intended to extend application logic), or access application-defined objects via interprocess-communication. Please refer to our [DebuggerIntegration/DebugRemoteScript](#) quickstart projects for more details.

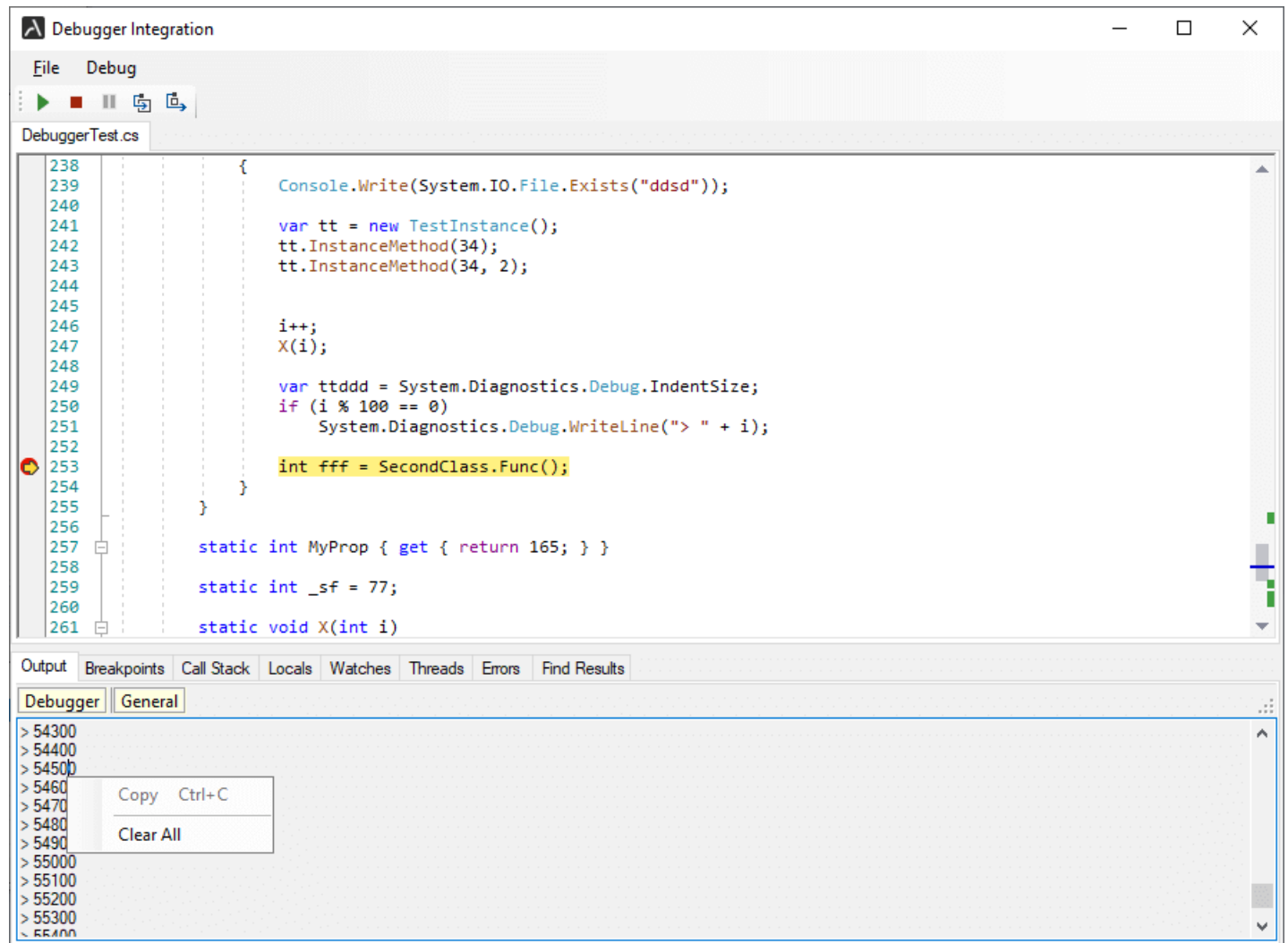
These limitations do not apply to Python, IronPython and TypeScript script debuggers.

# Script Debugging Widgets

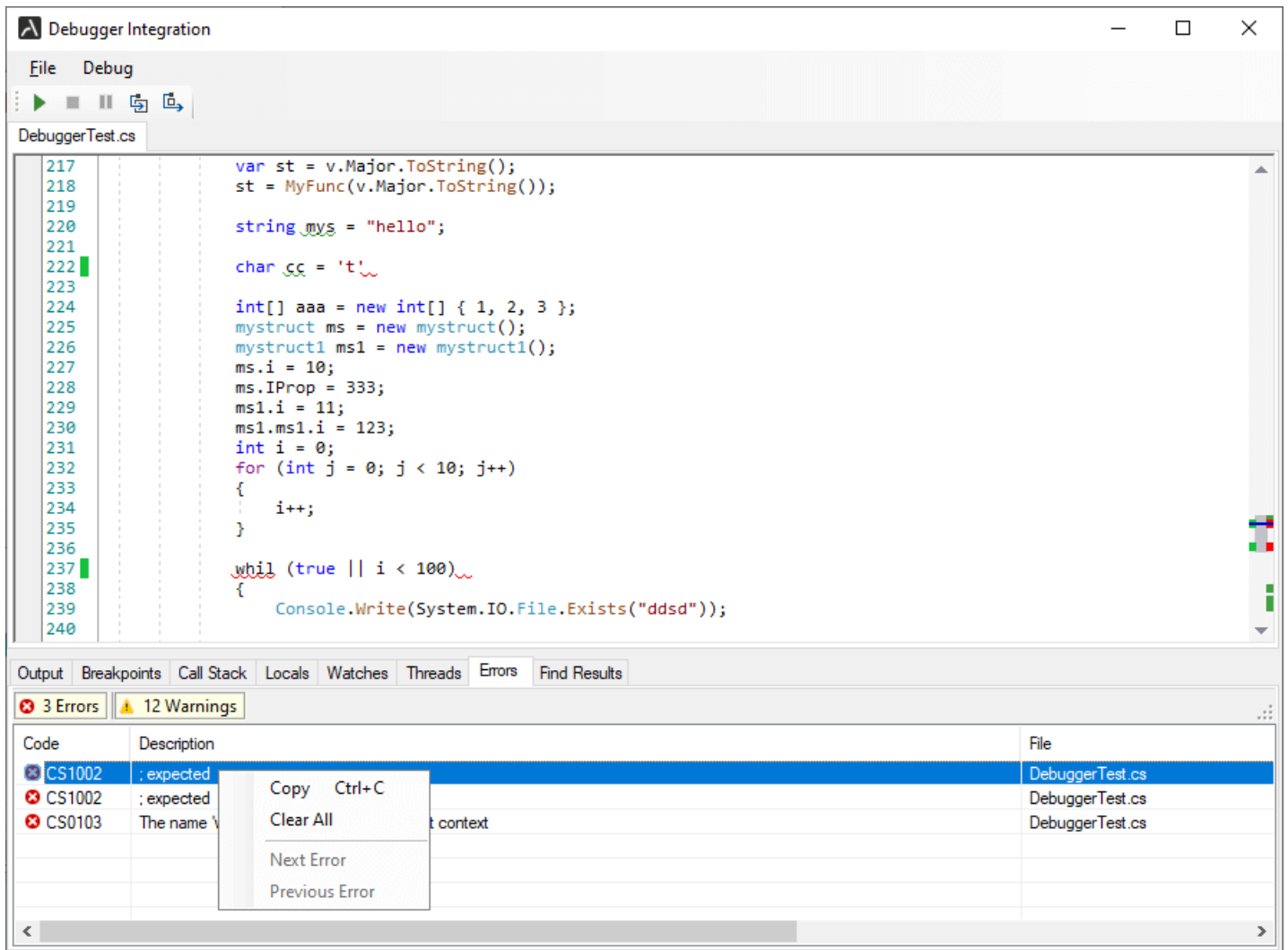
Scripter package includes a set of debugger widgets, toolbars, menus and code editors available both for WinForms and WPF and can be linked to any Debugger components for C#/VisualBasic, Python/IronPython, TypeScript/JavaScript and Debug Server Protocol-based debuggers.

These widgets include:

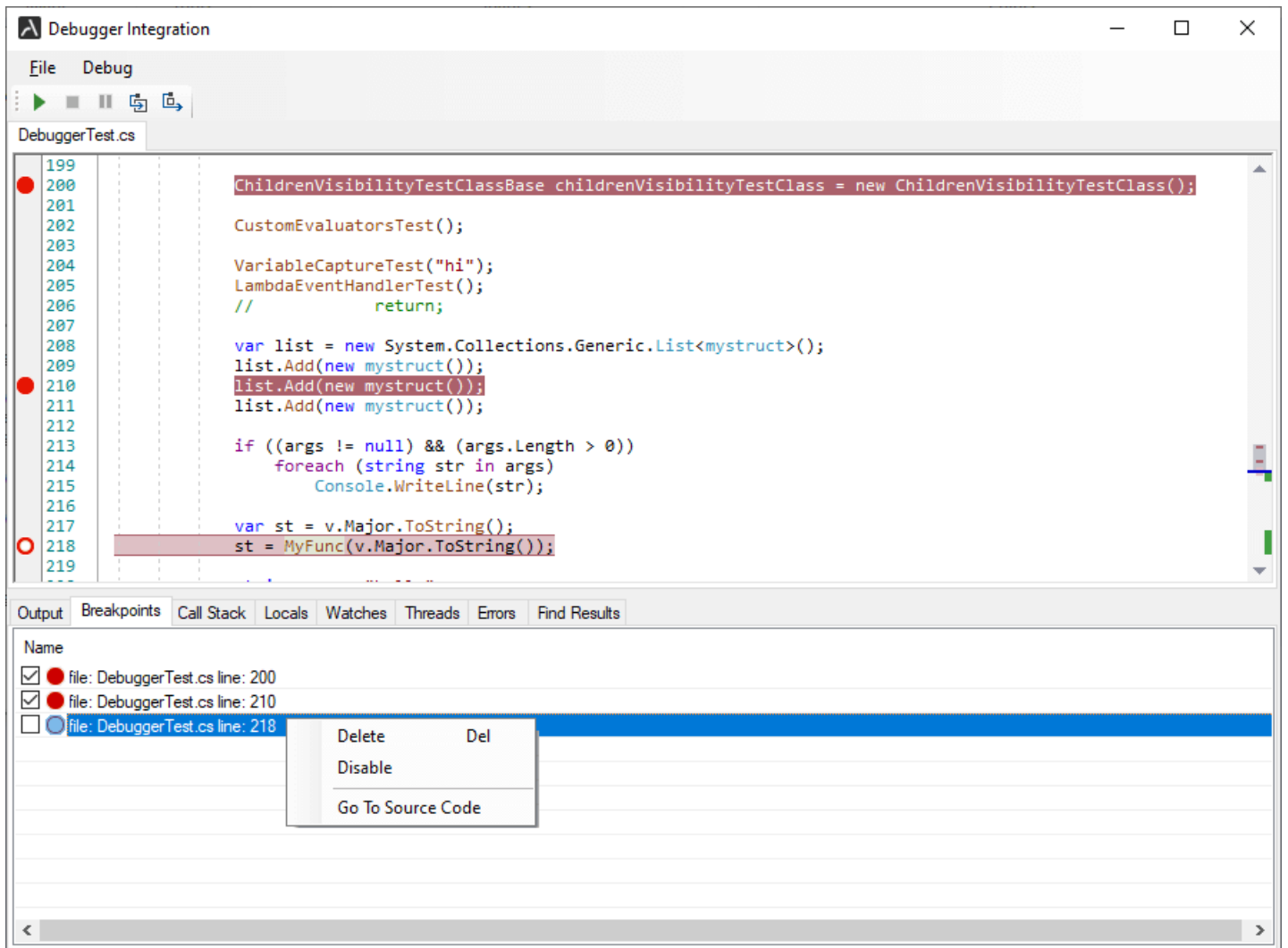
[Output \(Output\)](#) - to log debugger events or application-specific messages



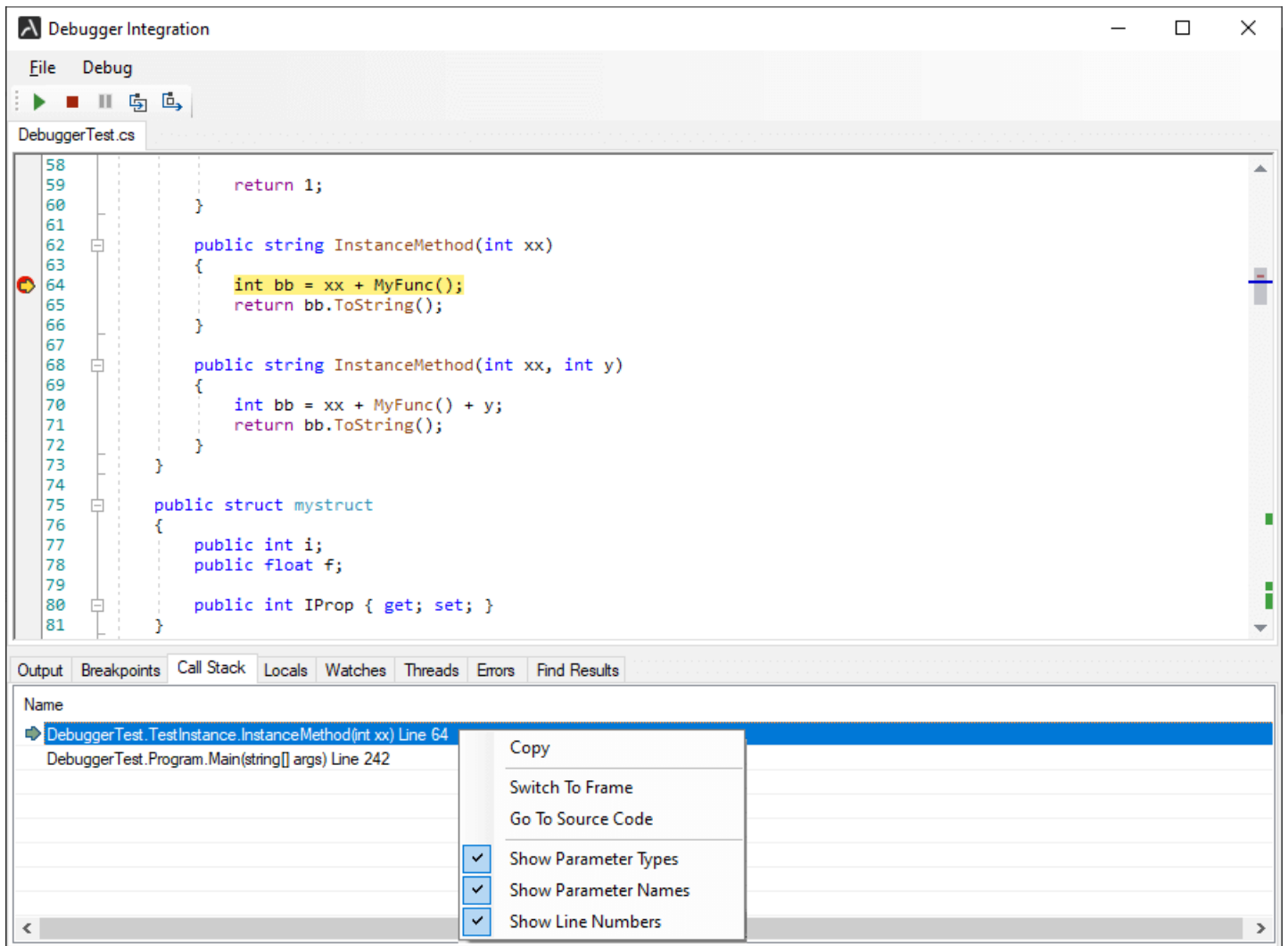
[Errors \(Errors\)](#) - to display a list of compilation errors.



**Breakpoints (Breakpoints)**- to display and navigate through the list of breakpoints set in the source;



**CallStack (CallStack)**- to display and navigate through the list of method calls that are currently on stack.



**Locals (Locals)**- to examine values of local variables once debugging code step-by-step.

Debugger Integration

File Debug

DebuggerTest.cs

```
220 string mys = "hello";
221
222 char cc = 't';
223
224 int[] aaa = new int[] { 1, 2, 3 };
225 mystruct ms = new mystruct();
226 mystruct1 ms1 = new mystruct1();
227 ms.i = 10;
228 ms.IProp = 333;
229 ms1.i = 11;
230 ms1.ms1.i = 123;
231 int i = 0;
232 for (int j = 0; j < 10; j++)
233 {
234 i++;
235 }
236
237 while (true || i < 100)
238 {
239 Console.Write(System.IO.File.Exists("ddsd"));
240 }
241
242 var tt = new TestInstance();
243 tt.InstanceMethod(34);
244 tt.InstanceMethod(34, 2);
```

Output Breakpoints Call Stack Locals Watches Threads Errors Find Results

args (System.String[]) = System.String[]

childrenVisibilityTestClass (DebuggerTest.Program+ChildrenVisibilityTestClass) = {DebuggerTest.Program+ChildrenVisibilityTestClass}

list (System.Collections.Generic.List`1<DebuggerTest.mystruct>) = {DebuggerTest.mystruct}

st (System.String) = "0"

mys (System.String) = "hello"

cc (System.Char) = t

aaa (System.Int32[]) = System.Int32[]

ms (DebuggerTest.mystruct) = {DebuggerTest.mystruct}

ms1 (DebuggerTest.mystruct1) = {DebuggerTest.mystruct1}

i (System.Int32) = 0

j (System.Int32) = 0

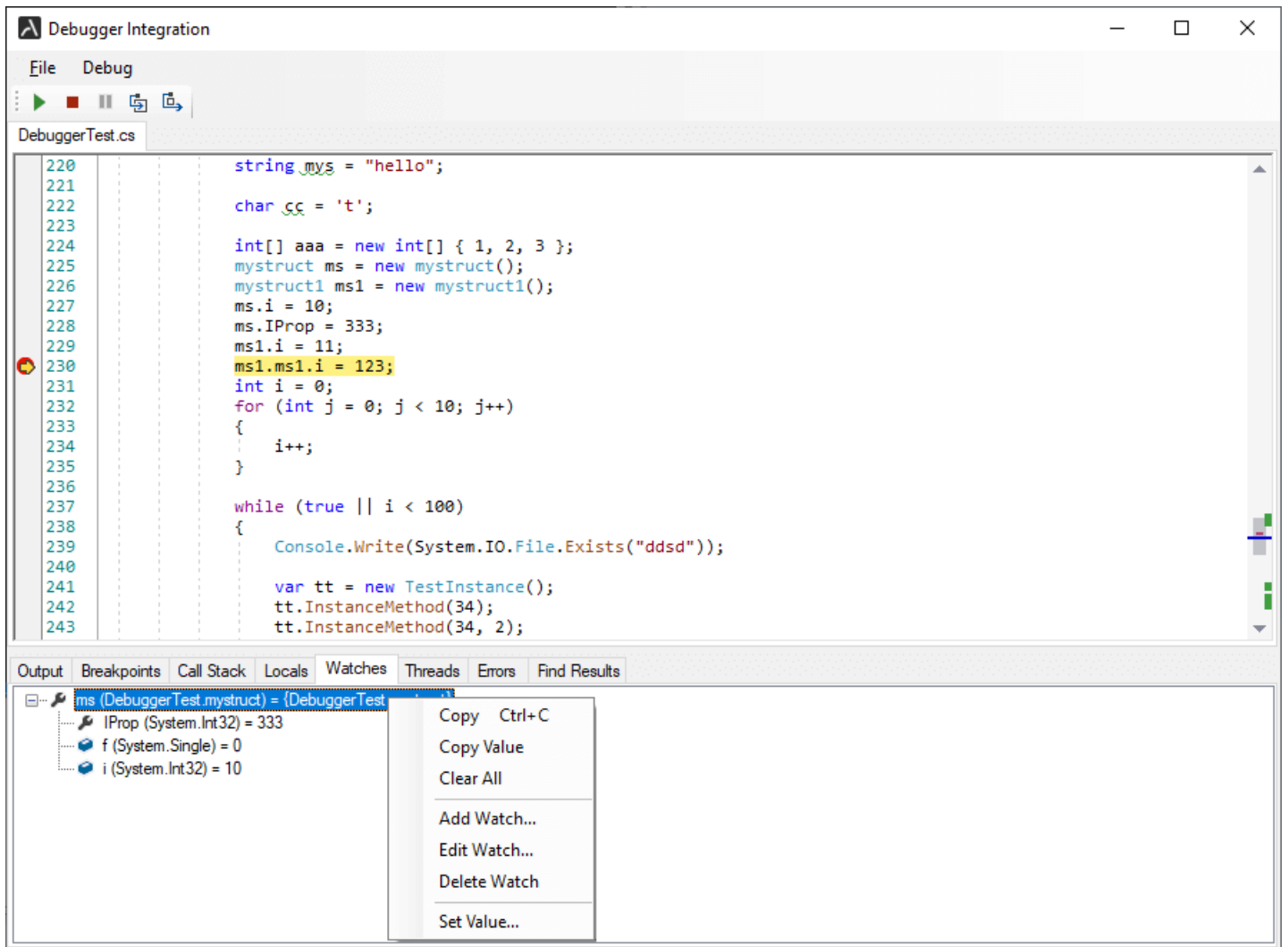
Copy Ctrl+C

Copy Value

Add to Watch

Set Value...

**Watches (Watches)**- to examine values of watch expressions when debugging



[Threads \(Threads\)](#)- to display active threads and switch debugging between them.

Debugger Integration

File Debug

Threads.cs

```

38
39
40 lock (consoleSync)
41 Console.WriteLine(value);
42 }
43 }
44
45 [STAThread]
46 public static void Main()
47 {
48 const int ProducerCount = 4;
49 const int ConsumerCount = 3;
50
51 for (int i = 0; i < ProducerCount; i++)
52 new Thread(ProducerThreadProc) { IsBackground = true }.Start();
53
54 for (int i = 0; i < ConsumerCount; i++)
55 new Thread(ConsumerThreadProc) { IsBackground = true }.Start();
56
57 Thread.Sleep(Timeout.Infinite);
58 }
59
60 }

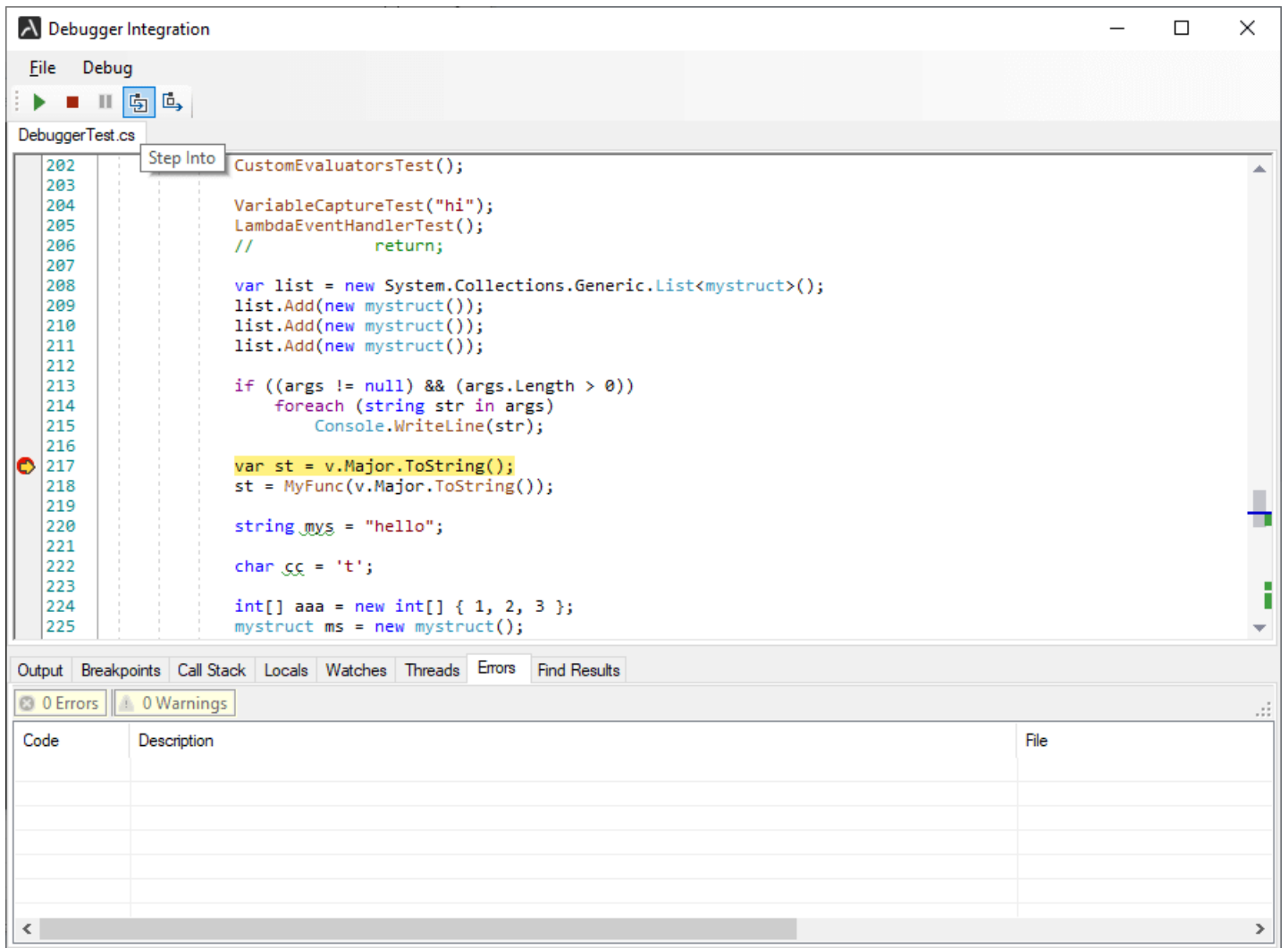
```

Output Breakpoints Call Stack Locals Watches Threads Errors Find Results

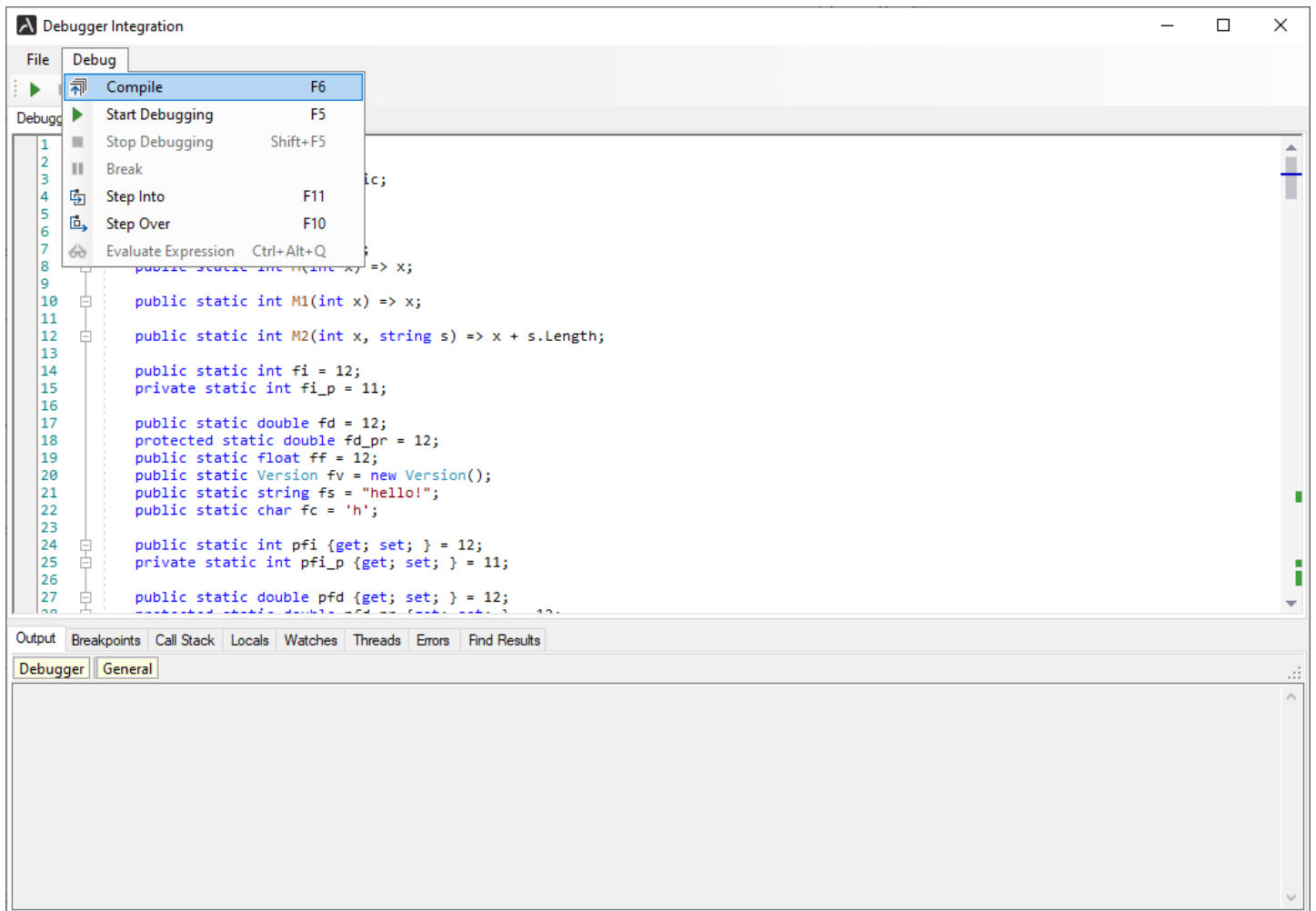
Name	ID
<b>Threads.cs</b>	<b>20532</b>
Threads.cs	22460
Threads.cs	22128

[DebuggerControlToolbar](#) ([DebuggerControlToolbar](#))- a toolbar with buttons executing Run/Stop/StepInto/StepOver commands.





[DebugMenu \(DebugMenu\)](#) - menu with menu items executing Run/Stop/StepInto/StepOver commands.



[DebugCodeEdit](#) ([DebugCodeEdit](#)) code-editing controls designed to work with the Script debugger; these controls allow user to set or remove breakpoints and evaluate expressions by hovering mouse over the symbol during debugging.

Debugger Integration

File

Debug

DebuggerTest.cs

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

VariableCaptureTest("hi");

LambdaEventHandlerTest();

// return;

var list = new System.Collections.Generic.List<mystruct>();

list.Add(new mystruct());

list.Add(new mystruct());

list.Add(new mystruct());

if ((args != null) && (args.Length > 0))

foreach (string str in args)

Console.WriteLine(str);

var st = v.Major.ToString();

st = MyFu int System.Version.Major

Gets the value of the major component of the version number for the current System.Version object.

string my = "hello";

char cc = 't';

int[] aaa = new int[] { 1, 2, 3 };

mystruct ms = new mystruct();

mystruct1 ms1 = new mystruct1();

ms.i = 10;

Output

Breakpoints

Call Stack

Locals

Watches

Threads

Errors

Find Results

Debugger

General

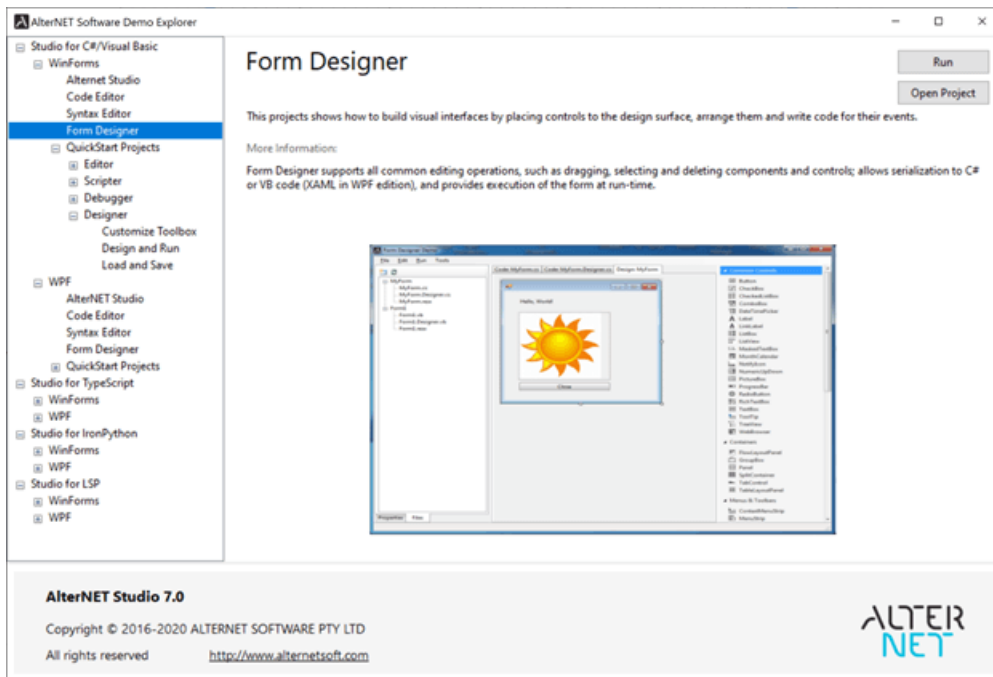
# Form Designer Overview

AlterNET Form Designer is a .NET component library providing a quick and convenient way for creating visual user interfaces. It allows placing controls to the design surfaces, setting their initial properties and writing event handlers for their events.

The main components in the package are [FormDesignerControl](#) for WinForms and [FormDesignerControl](#) for WPF. These controls represent a design surface allowing users to add controls to a form, arrange them, and write code for their events.

Form Designer supports all common editing operations such as dragging, selecting and deleting components and controls; changing their size and z-order, align them horizontally or vertically, copy and paste controls. Like Visual Studio Form Designer, it serializes its content into C#/VisualBasic or TypeScript/JavaScript source code.

Form Designer includes a set of demos and quick start projects that show how to place controls from the toolbox, load and save forms being designed, write code in control's event handlers, and how to run these forms.



Below is brief overview of these projects:

*FormDesigner* - This project shows how to build visual interfaces by placing controls to the design surface, arrange them and write code for their events.

*LoadAndSave* - demonstrates how to save/load forms being designed.

*DesignAndRun* - shows how to write event handlers code and run the form being designed.

*CustomizeToolbox* - shows how to rearrange toolbox tabs and install third-party assemblies on the toolbox.

## Creating your first project

The first thing to do after creating a new WinForms or WPF application is to place the [FormDesignerControl](#) or [FormDesignerControl](#) controls on your form. You would also need to place [ToolboxControl](#) or [ToolboxControl](#) controls and assign their [FormDesignerControl](#) property so you can drag and drop controls and components from it to the design surface.

When you run the application, you will see a new empty form with the design surface, where you can drag controls from the toolbox, resize, arrange them, etc.

If you'd like to examine and change properties of the selected controls, you will need to place [PropertyGridControl](#) or [PropertyGridControl](#) controls, set their [FormDesignerControl](#) property so it displays a component or a control being currently selected in the designer.

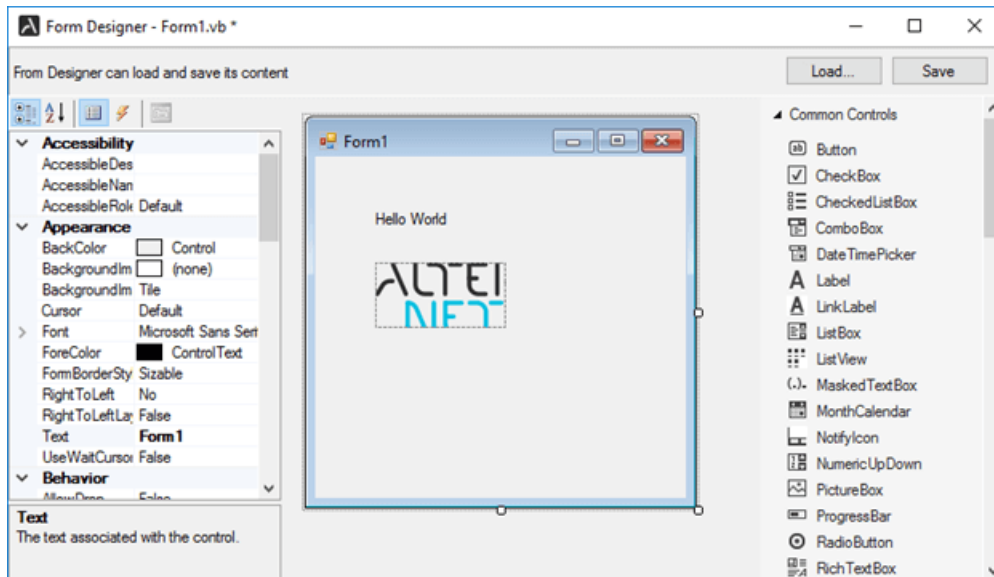
Similarly, for navigation through form's control you can place [OutlineControl](#) or [OutlineControl](#) controls and set their `FormDesignerControl` property.

# WinForms Form Designer

WinForms Form Designer provides a design-time surface, where the user can place controls from the toolbox, arrange them, examine and change their properties and write event handlers for their events.

## FormDesignerControl Control

[FormDesignerControl](#) is based on .NET Framework FormDesigner services, which are implemented in System.ComponentModel.Design namespace and included in .NET framework. This allows AlterNET FormDesigner to look and feel very similar to Visual Studio WinForms Form Designer.



Below are most essential properties, methods and events of FormDesignerControl class:

Properties:

[DesignerCommands](#) - provides an interface to Form Designer commands, such as Copy/Paste, Undo/Redo, Aligning and Arranging controls, etc.

[DesignerHost](#) - Provides an interface for managing designer transactions and components.

[IsModified](#) - Indicates whether designer content has been modified since last save.

[SelectedComponents](#) - contains list of selected components or controls

[PrimarySelection](#) - gets the first selected component or control.

[Source](#) - gets or sets FormDesigner Source.

[ToolboxControl](#) - gets or sets toolbox control associated with the designer.

[ReferencedAssemblies](#) - gets a collection of assemblies where the controls and components used on the form being designed are declared.

[ImportedNamespaces](#) - in the case of Visual Basic, gets a collection of globally available namespaces.

[Options](#) - allows to change Form Designer appearance by specifying whether to display snap lines, smart tags, form designer grid and change grid size.

Events:

[DesignerHostChanged](#) - occurs when the designer host changes, for example if a new form is loaded.

[NavigateToUserMethodRequested](#) - occurs when form designer is requested to navigate to the event handler. For example, when a user double clicks on the control.

[SelectionChanged](#) - occurs when a user selects a different control in the designer.

[CommandStateChanged](#) - occurs when state of designer commands changes (for example when undo stack becomes available)

[DesignedContentChanged](#) - occurs when a user modifies any aspect of the control being designed.

[LoadingErrorOccured](#) - occurs when there is a parse error of the design code during loading.

[CompilerErrorClick](#) - occurs when a user clicks on a compiler error on the Form Designer surface.

[DesignSurfaceKeyDown](#) - when a user presses a key when the design surface is focused.

#### Methods:

[Reload\(\)](#) - reloads form to be designed from the source.

[Save\(\)](#) - serializes designer to C# / Visual Basic file, Python or TypeScript / JavaScript code.

Form Designer works with three different source files simultaneously: one containing design-time code, another one containing user-written event-handlers and resource file for saving/loading form's resources (such as images). Most often users will need to edit at least a file containing event handlers, which will require setting up [FormDesignerControl](#) control's source to the one supporting integration with the editor. Refer to [FormDesignerTextSource](#) class for the implementation of the source which integrates with [SyntaxEdit](#) control included in our demo projects.

#### Error Handling

WinForms FormDesigner loads and saves its content into C#/VisualBasic, Python or TypeScript/JavaScript code; this code needs to be correct both from syntax and semantic point of view. In case FormDesigner loader encounters an error in the code, it switches to error mode and displays a list of found errors, allowing a user to click and navigate to the error source by handling [CompilerErrorClick](#) event.

**Note:** specific language assembly handling code serialization needs to be included in the project to save/load Form designer content into the code.

C#/VisualBasic language services are implemented in Alternet.FormDesigner.Roslyn.v8 assembly;

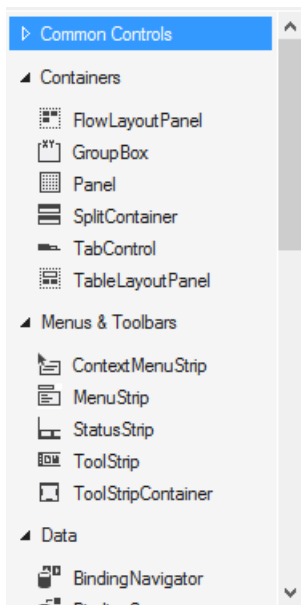
Python language services are implemented in Alternet.FormDesigner.Python.v8 assembly.

TypeScript/JavaScript language services are implemented in Alternet.FormDesigner.TypeScript.v8 assembly.



## ToolboxControl

The [ToolboxControl](#) control displays components and controls that you can place onto the design surface. It provides a set of foldable tabs helping to organize controls by categories and allowing to specify which components and controls, including third-party controls, appear on the toolbox, on which tabs and sort order.



ToolBox control can be linked to [FormDesignerControl](#) by setting [FormDesignerControl](#) property which allows dragging components and controls to the Form Designer.

Below are essential properties and methods to manipulate Toolbox control:

### Properties

[CategoryNames](#) - gets a collection of Categories (Tabs) displayed by the toolbox.

[SelectedTool](#) - returns currently selected toolbox item.

### Methods

[AddCategory](#) - adds a new category to the toolbox.

[AddItem](#) - places a toolbox item onto the specified toolbox tab.

[ClearItemsInCategory](#) - clears items in the specified tab.

[GetAllTools\(\)](#) - gets all toolbox items.

[GetToolsFromCategory](#) - gets toolbox items on the specific tab.

[SelectPointer\(\)](#) - deselects currently selected toolbox item and selects pointer tool.

[RemoveCategory](#) - removes a specific tab.

[RemoveItem](#) - removes a toolbox item from the category.

[SetSelectedItem](#) - selects toolbox item.

[AddItemForType](#) - adds toolbox item from type name

[AddItemsFromAssembly](#) - add all types that can appear on the toolbox from the assembly.

[ScrollToCategory](#) - scrolls toolbox control to the specified category.

[ScrollToItem](#) - scrolls toolbox control to the specified item.

[BeginUpdate\(\)](#) - prevents repainting of the toolbox until EndUpdate is called

[EndUpdate\(\)](#) - re-enables toolbox repainting.

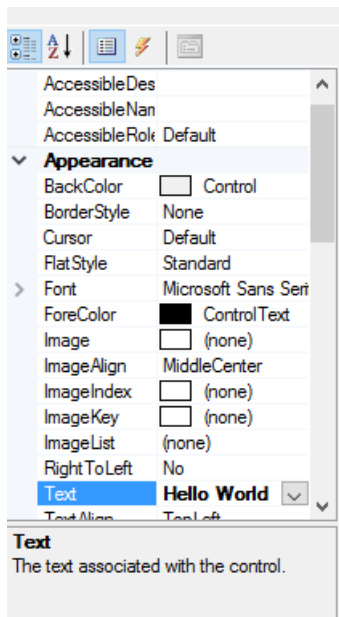
[Save](#) - saves the toolbox content to the specified stream.



[Load](#) - loads the toolbox content from the specified stream.

## PropertyGridControl

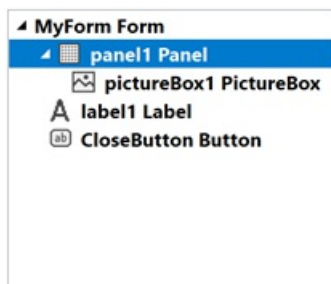
The [PropertyGridControl](#) control allows the user to view and change the design-time properties and events of the controls or components selected in the designer.



PropertyGrid control can be linked to [FormDesignerControl](#) by setting [FormDesignerControl](#) property which allows to view and change properties and events for the controls being selected in the Form Designer.

## OutlineControl

[OutlineControl](#) displays Form's layout as a tree view, providing an easy way to navigate, show/hide and re-arrange controls on the form.



OutlineControlcontrol can be linked to [FormDesignerControl](#) by setting [FormDesignerControl](#) property which allows navigating through the controls displayed by the Form Designer.

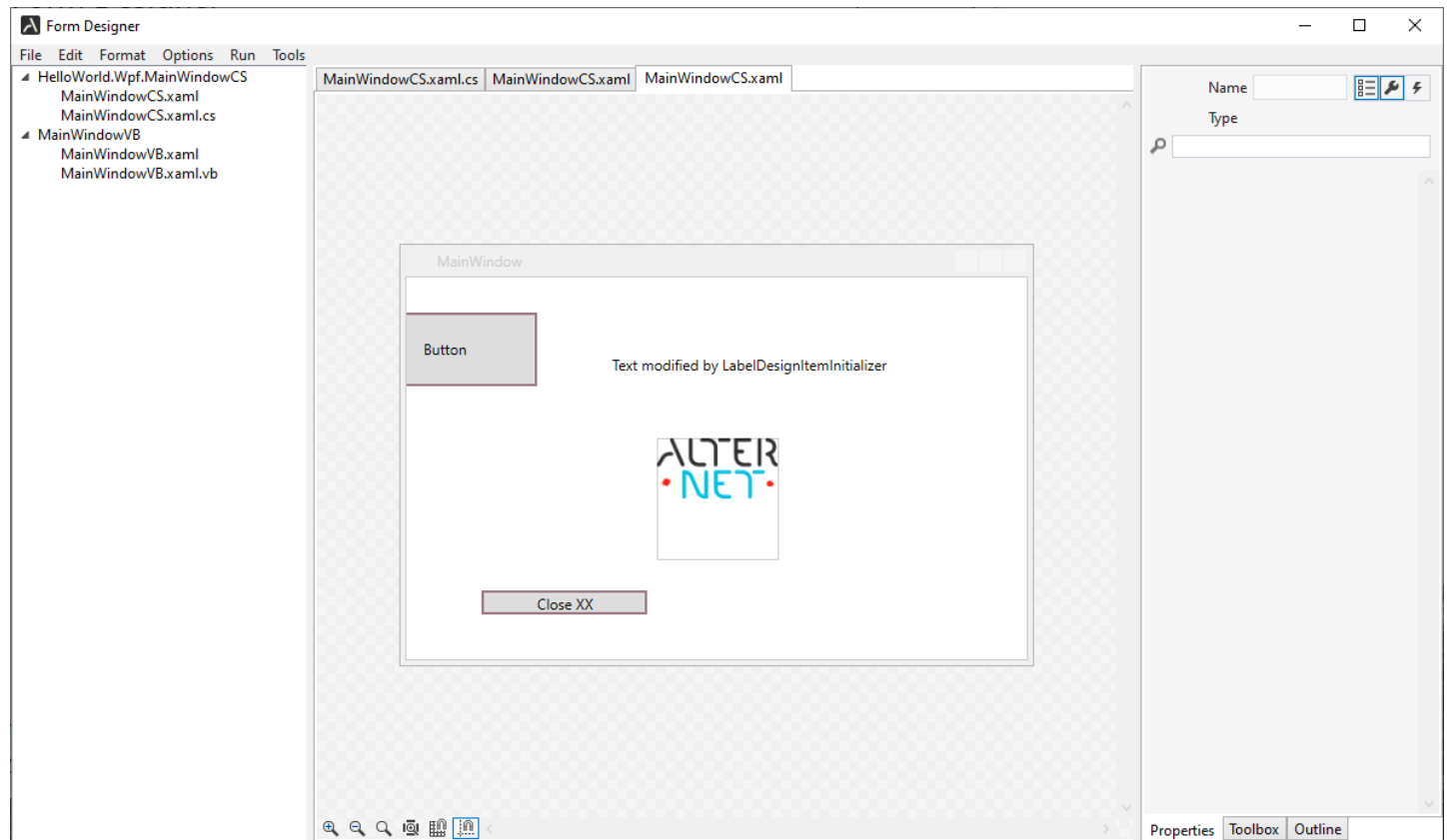
# WPF Form Designer

WPF Form Designer provides a design-time surface, where the user can place controls from the toolbox, arrange them, examine and change their properties and write event handlers for their events.

## FormDesignerControl Control

[FormDesignerControl](#) is based on SharpDevelop open-source FormDesigner package:

<https://github.com/icsharpcode/WpfDesigner>



Below are most essential properties, methods and events of `FormDesignerControl` class:

### Properties:

[DesignerCommands](#) - provides an interface to Form Designer commands, such as Copy/Paste, Undo/Redo, Aligning and Arranging controls, etc.

[IsModified](#) - Indicates whether designer content has been modified since last save.

[SelectedItems](#) - contains list of selected controls

[Source](#) - gets or sets FormDesigner Source.

[ReferencedAssemblies](#) - gets a collection of assemblies where the controls used on the form being designed are declared.

[CurrentTool](#) - gets or sets currently selected tool in the Form Designer.

### Events:

[NavigateToUserMethodRequested](#) - occurs when form designer is requested to navigate to the event handler. For example, when a user double clicks on the control.

[SelectionChanged](#) - occurs when a user selects a different control in the designer.

[DesignedContentChanged](#) - occurs when a user modifies any aspect of the control being designed.

## Methods:

[Reload\(\)](#) - reloads form to be designed from the source.

[Save\(\)](#) - serializes designer content to the XAML code.

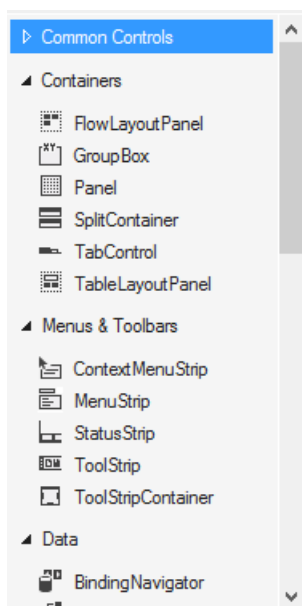
Form Designer works with two different source files simultaneously: XAML containing design-time code, and C# or Visual Basic source containing user-written event-handlers. Most often users will need to edit a file containing event handlers, which will require setting up [FormDesignerControl](#) control's source to the one supporting integration with the editor. Refer to [FormDesignerTextSource](#) class for the implementation of the source which integrates with [TextEditor](#) control included in our demo projects.

## Error Handling

WinForms FormDesigner loads and saves its content into C#/VisualBasic, Python or TypeScript/JavaScript code; this code needs to be correct both from syntax and semantic point of view. In case FormDesigner loader encounters an error in the code, it switches to error mode and displays a list of found errors, allowing a user to click and navigate to the error source by handling [CompilerErrorClick](#) event.

# ToolboxControl

The [ToolboxControl](#) control displays components and controls that you can place onto the design surface. It provides a set of foldable tabs helping to organize controls by categories and allowing to specify which components and controls, including third-party controls, appear on the toolbox, on which tabs and sort order.



ToolBox control can be linked to [FormDesignerControl](#) by setting [FormDesigner](#) property which allows dragging components and controls to the Form Designer.

Below are essential properties and methods to manipulate Toolbox control:

## Properties

[CategoryNames](#) - gets a collection of Categories (Tabs) displayed by the toolbox.

[SelectedToolboxItem](#) - returns currently selected toolbox item.

## Methods

[AddCategory](#) - adds a new category to the toolbox.

[AddItem](#) - places a toolbox item onto the specified toolbox tab.

[ClearItemsInCategory](#) - clears items in the specified tab.

[GetAllTools\(\)](#) - gets all toolbox items.

[GetToolsFromCategory](#) - gets toolbox items on the specific tab.

[SelectPointer\(\)](#) - deselects currently selected toolbox item and selects pointer tool.

[RemoveCategory](#) - removes a specific tab.

[RemoveItem](#) - removes a toolbox item from the category.

[SetSelectedItem](#) - selects toolbox item.

[AddItemForType](#) - adds toolbox item from type name

[AddItemsFromAssembly](#) - add all types that can appear on the toolbox from the assembly.

[BeginUpdate\(\)](#) - prevents repainting of the toolbox until EndUpdate is called

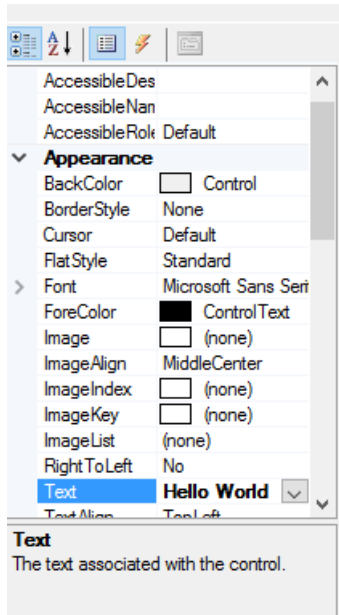
[EndUpdate\(\)](#) - re-enables toolbox repainting.

[Save](#) - saves the toolbox content to the specified stream.

[Load](#) - loads the toolbox content from the specified stream.

## PropertyGridControl

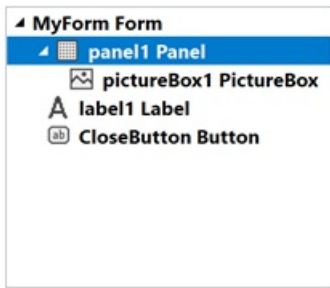
The [PropertyGridControl](#) control allows the user to view and change the design-time properties and events of the controls or components selected in the designer.



PropertyGrid control can be linked to [FormDesignerControl](#) by setting [FormDesignerControl](#) property which allows to view and change properties and events for the controls being selected in the Form Designer.

## OutlineControl

[OutlineControl](#) displays Form's layout as a tree view, providing an easy way to navigate, show/hide and re-arrange controls on the form.



OutlineControl can be linked to [FormDesignerControl](#) by setting [FormDesigner](#) property which allows navigating through the controls displayed by the Form Designer.