Windows Forms documentation

Learn about using Windows Forms, an open-source, graphical user interface for Windows, on .NET.

Learn about Windows Forms

WHAT'S NEW

What's new

OVERVIEW

Windows Forms overview

TUTORIAL

Create a new app

Migrate from .NET Framework

DOWNLOAD

Visual Studio 2022

Controls

OVERVIEW

About controls

CONCEPT

Layout

HOW-TO GUIDE

Add a control

Add access (shortcut) keys

Keyboard input

OVERVIEW

About the keyboard

CONCEPT

Keyboard events

HOW-TO GUIDE

Handle keyboard input

Modify key events

Mouse input

OVERVIEW

About the mouse

CONCEPT

Mouse events

Drag-and-drop

HOW-TO GUIDE

Manage the cursor

Simulate mouse events

What's new for .NET 8 (Windows Forms .NET)

Article • 12/15/2023

This article describes some of the new Windows Forms features and enhancements in .NET 8.

There are a few breaking changes you should be aware of when migrating from .NET Framework to .NET 8. For more information, see Breaking changes in Windows Forms.

Data binding improvements

A new data binding engine was in preview with .NET 7, and is now fully enabled in .NET 8. Though not as extensive as the existing Windows Forms data binding engine, this new engine is modeled after WPF, which makes it easier to implement MVVM design principles.

The enhanced data binding capabilities make it simpler to fully utilize the MVVM pattern and employ object-relational mappers from ViewModels in Windows Forms. This reduces the amount of code in code-behind files. More importantly, it enables code sharing between Windows Forms and other .NET GUI frameworks like WPF, UWP/WinUI, and .NET MAUI. It's important to note that while the previously mentioned GUI frameworks use XAML as a UI technology, XAML isn't coming to Windows Forms.

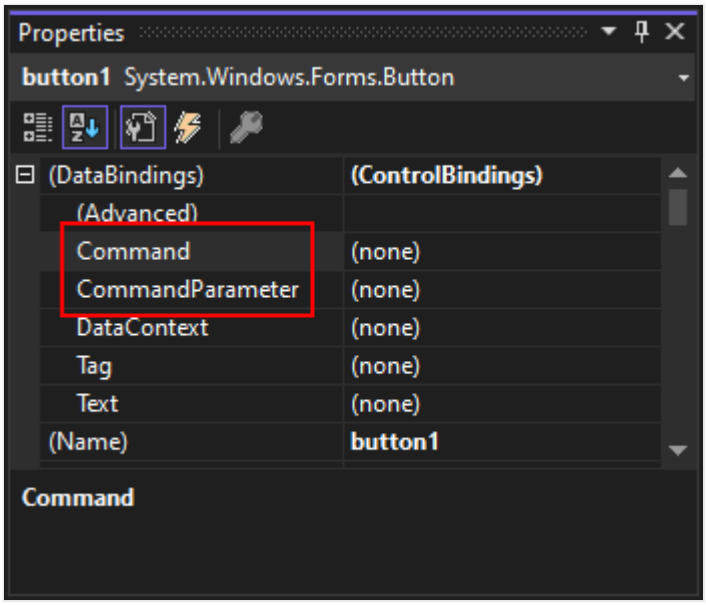
The IBindableComponent interface and the BindableComponent class drive the new binding system. Control implements the interface and provides new data binding capabilities to Windows Forms.

Button commands

Button commands were in preview with .NET 7, and is now fully enabled in .NET 8. Similar to WPF, the instance of an object that implements the ICommand interface can be assigned to the button's Command property. When the button is clicked, the command is invoked.

An optional parameter can be provided when the command is invoked, by the specifying a value for the button's CommandParameter property.

The `Command` and `CommandParameter` properties are set in the designer through the **Properties** window, under **(DataBindings)**, as illustrated by the following image.



Buttons also listen to the ICommand.CanExecuteChanged event, which causes the control to query the ICommand.CanExecute method. When that method returns `true`, the control is enabled; the control is disabled when `false` is returned.

Visual Studio DPI improvements

Visual Studio 2022 17.8 Introduces DPI-unwaware designer tabs. Previously, the Windows Designer tab in Visual Studio ran at the DPI of Visual Studio. This causes problems when you're designing a DPI-unaware Windows Forms app. Now you can ensure that the designer runs at the same scale as you want the app to run, either DPI-aware or not. Before this feature was introduced, you had to run Visual Studio in DPI-unaware mode, which made Visual Studio itself blurry when scaling was applied in Windows. Now you can leave Visual Studio alone and let the designer run DPI-unaware.

You can enable the DPI-unaware designer for the Windows Forms project by adding `<ForceDesignerDPIUnaware>` to the project file, and setting the value to `true`.

XML

```
<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net8.0-windows</TargetFramework>
  <Nullable>enable</Nullable>
  <UseWindowsForms>true</UseWindowsForms>
  <ImplicitUsings>enable</ImplicitUsings>
  <ForceDesignerDPIUnaware>false</ForceDesignerDPIUnaware>
  <ApplicationHighDpiMode>DpiUnawareGdiScaled</ApplicationHighDpiMode>
</PropertyGroup>
```

> ⓘ **Important**

Visual Studio reads this setting when the project is loaded, and not when it's changed. After changing this setting, unload and reload your project to get Visual Studio to respect it.

High DPI improvements

High DPI rendering with PerMonitorV2 has been improved:

- Correctly scale nested controls. For example, a button that's in a panel, which is placed on a tab page.
- Scale Form.MaximumSize and Form.MinimumSize properties based on the current monitor DPI settings.

  Starting with .NET 8, this feature is enabled by default and you need to opt out of it to revert to the previous behavior.

  To disable the feature, add `System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi` to the `configProperties` setting in *runtimeconfig.json*, and set the value to false:

  JSON

  ```
  {
    "runtimeOptions": {
      "tfm": "net8.0",
      "frameworks": [
          ...
      ],
      "configProperties": {
        "System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi": false,
      }
    }
  }
  ```

Miscellaneous improvements

Here are some other notable changes:

- The code that handled `FolderBrowserDialog` was improved, fixing a few memory leaks.
- The code base for Windows Forms has been slowly enabling C# nullability, rooting out any potential null-reference errors.

- The `System.Drawing` source code was migrated to the Windows Forms GitHub repository.
- Modern Windows icons can be accessed by a new API, System.Drawing.SystemIcons.GetStockIcon. The System.Drawing.StockIconId enumeration lists all of the available system icons.
- More designers are available at run-time now. For more information, see GitHub issue #4908.

What's new for .NET 7 (Windows Forms .NET)

Article • 02/14/2023

This article describes some of the new Windows Forms features and enhancements in .NET 7.

There are a few breaking changes you should be aware of when migrating from .NET Framework to .NET 7. For more information, see Breaking changes in Windows Forms.

High DPI improvements

High DPI rendering with PerMonitorV2 has been improved:

- Correctly scale nested controls. For example, a button that's in a panel, which is placed on a tab page.
- Scale Form.MaximumSize and Form.MinimumSize properties based on the current monitor DPI settings for applications that run `ApplicationHighDpiMode` set to `PerMonitorV2`.

  In .NET 7, this feature is disabled by default and you must opt in to receive this change. Starting with .NET 8, this feature is enabled by default and you need to opt out of it to revert to the previous behavior.

  To enable feature, set the `configProperties` setting in *runtimeconfig.json*:

  JSON

  ```
  {
    "runtimeOptions": {
      "tfm": "net7.0",
      "frameworks": [
          ...
      ],
      "configProperties": {
        "System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi": true,
      }
    }
  }
  ```

Accessibility improvements and fixes

This release adds further improvements to accessibility, including but not limited to the following items:

- Many announcement-related issues observed in screen readers have been addressed, ensuring the information about controls is correct. For example, ListView now correctly announces when a group is expanded or collapsed.
- More controls now provide UI Automation support:
  - TreeView
  - DateTimePicker
  - ToolStripContainer
  - ToolStripPanel
  - FlowLayoutPanel
  - TableLayoutPanel
  - SplitContainer
  - PrintPreviewControl
  - WebBrowser
- Memory leaks related to running a Windows Forms application under assistive tools, such as Narrator, have been fixed.
- Assistive tools now accurately draw focus indicators and report correct bounding rectangles for nested forms and some elements of composite controls, such as DataGridView, ListView, and TabControl.
- The Automation UI ExpandCollapse Control Pattern has been properly implemented in ListView, TreeView, and PropertyGrid controls, and only activates for expandable items.
- Various color contrast ratio corrections in controls.
- Visibility improvements for ToolStripTextBox and ToolStripButton in high-contrast themes.

Data binding improvements (preview)

While Windows Forms already had a powerful binding engine, a more modern form of data binding, similar to data binding provided by WPF, is being introduced.

The new data binding features allow you to fully embrace the MVVM pattern and the use of object-relational mappers from ViewModels in Windows Forms easier than before. This, in turn, makes it possible to reduce code in the code-behind files, and opens new testing possibilities. More importantly, it enables code sharing between

Windows Forms and other .NET GUI frameworks such as WPF, UWP/WinUI, and .NET MAUI. And to clarify a commonly asked question, there aren't any plans to introduce XAML in Windows Forms.

These new data binding features are in preview for .NET 7, and more work on this feature will happen in .NET 8.

To enable the new binding, add the `EnablePreviewFeatures` setting to your project file. This is supported in both C# and Visual Basic.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <!-- other settings -->

  <PropertyGroup>
    <EnablePreviewFeatures>true</EnablePreviewFeatures>
  </PropertyGroup>

</Project>
```

The following code snippet demonstrates the new properties, events, and methods added to the various classes in Windows Forms. Even though the following code example is in C#, it also applies to Visual Basic.

C#

```
public class Control  {
    [BindableAttribute(true)]
    public virtual object DataContext { get; set; }
    [BrowsableAttribute(true)]
    public event EventHandler DataContextChanged;
    protected virtual void OnDataContextChanged(EventArgs e);
    protected virtual void OnParentDataContextChanged(EventArgs e);
}

[RequiresPreviewFeaturesAttribute]
public abstract class BindableComponent : Component, IBindableComponent,
IComponent, IDisposable {
    protected BindableComponent();
    public BindingContext? BindingContext { get; set; }
    public ControlBindingsCollection DataBindings { get; }
    public event EventHandler BindingContextChanged;
    protected virtual void OnBindingContextChanged(EventArgs e);
}

public abstract class ButtonBase : Control {
    [BindableAttribute(true)]
    [RequiresPreviewFeaturesAttribute]
```

```
    public ICommand? Command { get; set; }
    [BindableAttribute(true)]
    public object? CommandParameter { [RequiresPreviewFeaturesAttribute]
get; [RequiresPreviewFeaturesAttribute] set; }
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler? CommandCanExecuteChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler? CommandChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler? CommandParameterChanged;
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandCanExecuteChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandParameterChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnRequestCommandExecute(EventArgs e);
}

public abstract class ToolStripItem : BindableComponent, IComponent,
IDisposable, IDropTarget {
    [BindableAttribute(true)]
    [RequiresPreviewFeaturesAttribute]
    public ICommand Command { get; set; }
    [BindableAttribute(true)]
    public object CommandParameter { [RequiresPreviewFeaturesAttribute] get;
[RequiresPreviewFeaturesAttribute] set; }
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandCanExecuteChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandParameterChanged;
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandCanExecuteChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandParameterChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnRequestCommandExecute(EventArgs e);
}
```

Miscellaneous improvements

Here are some other notable changes:

- Drag-and-drop handling matches the Windows drag-and-drop functionality with richer display effects such as icons and text labels.
- Folder and file dialogs allow for more options:

    - Add to recent
    - Check write access
    - Expanded mode
    - OK requires interaction
    - Select read-only
    - Show hidden files
    - Show pinned places
    - Show preview
- ErrorProvider has a HasErrors property now.
- Form's snap layout is fixed for Windows 11.

See also

- .NET Blog - What's new in Windows Forms in .NET 7
- Breaking changes in Windows Forms
- Tutorial: Create a new WinForms app
- How to migrate a Windows Forms desktop app to .NET 5

What's new for .NET 6 (Windows Forms .NET)

Article • 06/02/2023

This article describes some of the new Windows Forms features and enhancements in .NET 6.

There are a few breaking changes you should be aware of when migrating from .NET Framework to .NET 6. For more information, see Breaking changes in Windows Forms.

Updated templates for C#

.NET 6 introduced many changes to the standard console application templates. In line with those changes, the Windows Forms templates for C# have been updated to enable global using directives, file-scoped namespaces, and nullable reference types by default.

One feature of the new C# templates that has not been carried forward with Windows Forms is top-level statements. The typical Windows Forms application requires the `[STAThread]` attribute and consists of multiple types split across multiple files, such as the designer code files, so using **top-level statements** doesn't make sense.

New application bootstrap

The templates that generate a new Windows Forms application create a `Main` method which serves as the entry point for your application when it runs. This method contains code that configures Windows Forms and displays the first form, known as the bootstrap code:

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

In .NET 6, these templates have been modified to use the new bootstrap code, invoked by the `ApplicationConfiguration.Initialize` method.

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
    }
}
```

This method is automatically generated at compile time and contains the code to configure Windows Forms. The project file can control these settings now too, and you can avoid configuring it in code. For example, the generated method looks similar to the following code:

C#

```
public static void Initialize()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.SetHighDpiMode(HighDpiMode.SystemAware);
}
```

The new bootstrap code is used by Visual Studio to configure the Windows Forms Visual Designer. If you opt-out of using the new bootstrap code, by restoring the old code and bypassing the `ApplicationConfiguration.Initialize` method, the Windows Forms Visual Designer won't respect the bootstrap settings you set.

The settings generated in the `Initialize` method are controlled by the project file.

Project-level application settings

To complement the new application bootstrap feature of Windows Forms, a few `Application` settings previously set in the startup code of the application should be set in the project file. The project file can configure the following application settings:

Project setting	Default value	Corresponding API
ApplicationVisualStyles	`true`	`Application.EnableVisualStyles`

Project setting	Default value	Corresponding API
ApplicationUseCompatibleTextRendering	`false`	`Application.SetCompatibleTextRenderingDefault`
ApplicationHighDpiMode	`SystemAware`	`Application.SetHighDpiMode`
ApplicationDefaultFont	`Segoe UI, 9pt`	`Application.SetDefaultFont`

The following example demonstrates a project file that sets these application-related properties:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net6.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWindowsForms>true</UseWindowsForms>
    <ImplicitUsings>enable</ImplicitUsings>

    <ApplicationVisualStyles>true</ApplicationVisualStyles>

<ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRen
dering>
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
    <ApplicationDefaultFont>Microsoft Sans Serif,
8.25pt</ApplicationDefaultFont>

  </PropertyGroup>

</Project>
```

The Windows Forms Visual Designer uses these settings. For more information, see the Visual Studio designer improvements section.

Change the default font

Windows Forms on .NET Core 3.0 introduced a new default font for Windows Forms: **Segoe UI, 9pt**. This font better aligned to the Windows user experience (UX) guidelines. However, .NET Framework uses **Microsoft Sans Serif, 8.25pt** as the default font. This change made it harder for some customers to migrate their large applications that utilized a pixel-perfect layout from .NET Framework to .NET. The only way to change the

font for the whole application was to edit every form in the project, setting the Font property to an alternate font.

The default font can now be set in two ways:

- Set the default font in the project file to be used by the application bootstrap code:

  > **Important**
  >
  > This is the preferred way. Using the project to configure the new application bootstrap system allows Visual Studio to use these settings in the designer.

  In the following example, the project file configures Windows Forms to use the same font that .NET Framework uses.

  XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <!-- other settings -->

  <PropertyGroup>
    <ApplicationDefaultFont>Microsoft Sans Serif,
8.25pt</ApplicationDefaultFont>
  </PropertyGroup>

</Project>
```

- or -

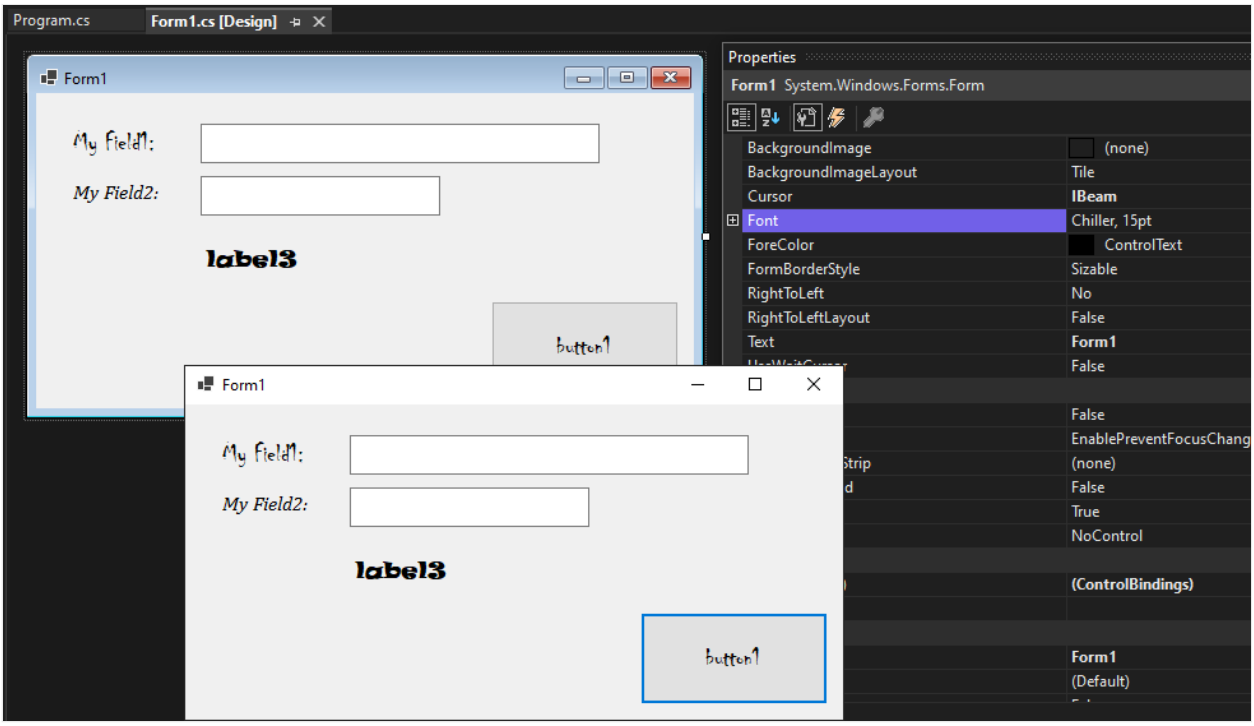
- Call the the Application.SetDefaultFont API in the old way (but with no designer support):

  C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.SetHighDpiMode(HighDpiMode.SystemAware);
        Application.SetDefaultFont(new Font(new FontFamily("Microsoft
Sans Serif"), 8.25f));
        Application.Run(new Form1());
    }
}
```

Visual Studio designer improvements

The Windows Forms Visual Designer now accurately reflects the default font. Previous versions of Windows Forms for .NET didn't properly display the **Segoe UI** font in the Visual Designer, and was actually designing the form with the .NET Framework's default font. Because of the new new application bootstrap feature, the Visual Designer accurately reflects the default font. Additionally, the Visual Designer respects the default font that's set in the project file.



More runtime designers

Designers that existed in the .NET Framework and enabled building a general-purpose designer, for example building a report designer, have been added to .NET 6:

- System.ComponentModel.Design.ComponentDesigner
- System.Windows.Forms.Design.ButtonBaseDesigner
- System.Windows.Forms.Design.ComboBoxDesigner
- System.Windows.Forms.Design.ControlDesigner
- System.Windows.Forms.Design.DocumentDesigner
- System.Windows.Forms.Design.DocumentDesigner
- System.Windows.Forms.Design.FormDocumentDesigner
- System.Windows.Forms.Design.GroupBoxDesigner
- System.Windows.Forms.Design.LabelDesigner
- System.Windows.Forms.Design.ListBoxDesigner
- System.Windows.Forms.Design.ListViewDesigner
- System.Windows.Forms.Design.MaskedTextBoxDesigner

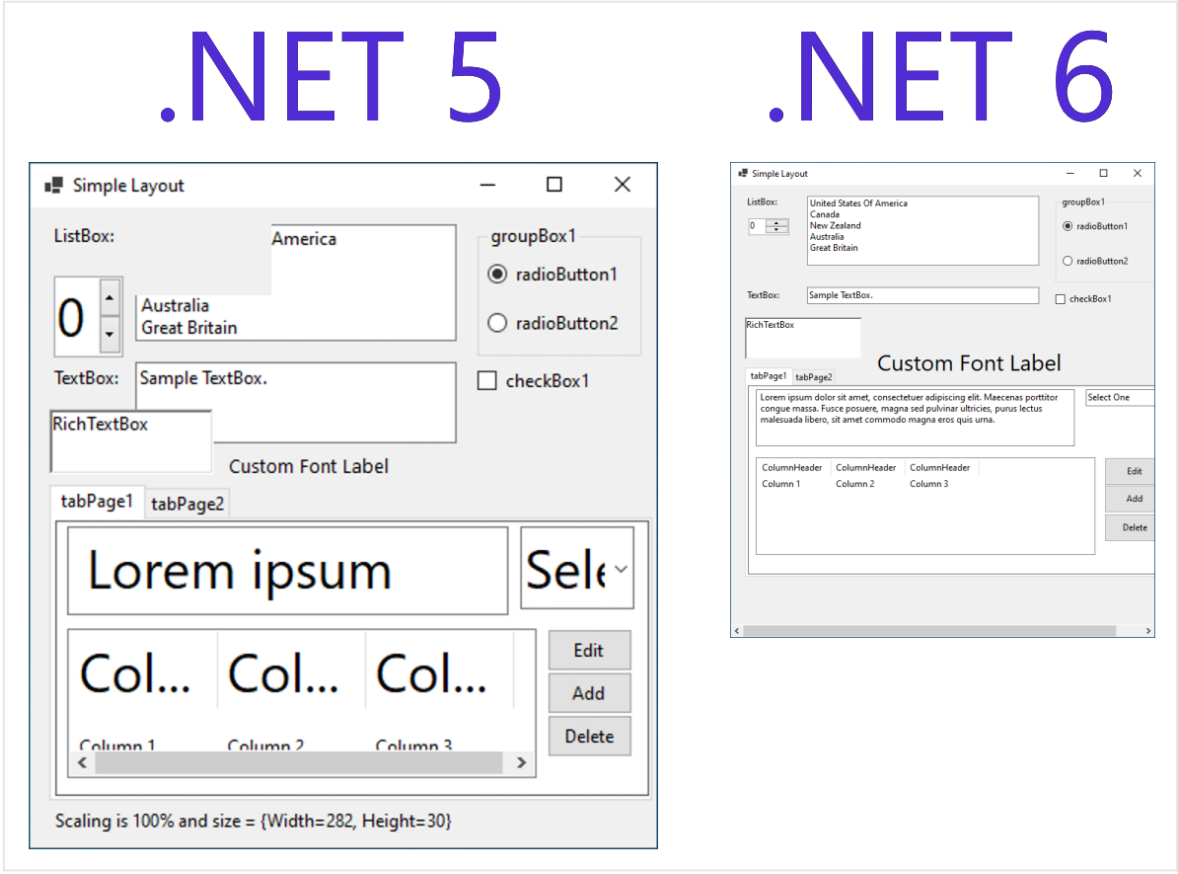
- System.Windows.Forms.Design.PanelDesigner
- System.Windows.Forms.Design.ParentControlDesigner
- System.Windows.Forms.Design.ParentControlDesigner
- System.Windows.Forms.Design.PictureBoxDesigner
- System.Windows.Forms.Design.RadioButtonDesigner
- System.Windows.Forms.Design.RichTextBoxDesigner
- System.Windows.Forms.Design.ScrollableControlDesigner
- System.Windows.Forms.Design.ScrollableControlDesigner
- System.Windows.Forms.Design.TextBoxBaseDesigner
- System.Windows.Forms.Design.TextBoxDesigner
- System.Windows.Forms.Design.ToolStripDesigner
- System.Windows.Forms.Design.ToolStripDropDownDesigner
- System.Windows.Forms.Design.ToolStripItemDesigner
- System.Windows.Forms.Design.ToolStripMenuItemDesigner
- System.Windows.Forms.Design.TreeViewDesigner
- System.Windows.Forms.Design.UpDownBaseDesigner
- System.Windows.Forms.Design.UserControlDocumentDesigner

High DPI improvements for PerMonitorV2

High DPI rendering with PerMonitorV2 have been improved:

- Controls are created with the same DPI awareness as the application.
- Container controls and MDI child windows have improved scaling behaviors.

  For example, in .NET 5, moving a Windows Forms app from a monitor with 200% scaling to a monitor with 100% scaling would result in misplaced controls. This has been greatly improved in .NET 6:



New APIs

- System.Windows.Forms.Application.SetDefaultFont
- System.Windows.Forms.Control.IsAncestorSiteInDesignMode
- System.Windows.Forms.ProfessionalColors.StatusStripBorder
- System.Windows.Forms.ProfessionalColorTable.StatusStripBorder

New Visual Basic APIs

- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventHandler
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.MinimumSplashScreenDisplayTime
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.MinimumSplashScreenDisplayTime
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.Font
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.Font
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.HighDpiMode
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.HighDpiMode

- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.ApplyApplicationDefaults
- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.HighDpiMode
- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.HighDpiMode

Updated APIs

- System.Windows.Forms.Control.Invoke now accepts System.Action and System.Func<TResult> as input parameters.
- System.Windows.Forms.Control.BeginInvoke now accepts System.Action as an input parameter.
- System.Windows.Forms.DialogResult is extended with the following members:
  - `TryAgain`
  - `Continue`
- System.Windows.Forms.Form has a new property: MdiChildrenMinimizedAnchorBottom
- System.Windows.Forms.MessageBoxButtons is extended with the following member:
  - `CancelTryContinue`
- System.Windows.Forms.MessageBoxDefaultButton is extended with the following member:
  - `Button4`
- System.Windows.Forms.LinkClickedEventArgs has now a new constructor and extended with the following properties:
  - System.Windows.Forms.LinkClickedEventArgs.LinkLength
  - System.Windows.Forms.LinkClickedEventArgs.LinkStart
- System.Windows.Forms.NotifyIcon.Text is now limited to 127 characters (from 63).

Improved accessibility

Microsoft UI Automation patterns work better with accessibility tools like Narrator and Jaws.

See also

- Breaking changes in Windows Forms
- Tutorial: Create a new WinForms app
- How to upgrade a Windows Forms desktop app to .NET 7

What's new for .NET 5 (Windows Forms .NET)

Article • 11/12/2021

Windows Forms for .NET 5 adds the following features and enhancements over .NET Framework.

There are a few breaking changes you should be aware of when migrating from .NET Framework to .NET 5. For more information, see Breaking changes in Windows Forms.

Enhanced features

- Microsoft UI Automation patterns work better with accessibility tools like Narrator and Jaws.
- Improved performance.
- The VB.NET project template defaults to DPI SystemAware settings for high DPI resolutions such as 4k monitors.
- The default font matches the current Windows design recommendations.

  > ⊗ **Caution**
  >
  > This may impact the layout of apps migrated from .NET Framework.

New controls

The following controls have been added since Windows Forms was ported to .NET Framework:

- System.Windows.Forms.TaskDialog

  A task dialog is a dialog box that can be used to display information and receive simple input from the user. Like a message box, it's formatted by the operating system according to parameters you set. Task dialog has more features than a message box. For more information, see the Task dialog sample.

- Microsoft.Web.WebView2.WinForms.WebView2

A new web browser control with modern web support. Based on Edge (Chromium). For more information, see Getting started with WebView2 in Windows Forms.

Enhanced controls

- System.Windows.Forms.ListView
  - Supports collapsible groups
  - Footers
  - Group subtitle, task, and title images

- System.Windows.Forms.FolderBrowserDialog

  This dialog has been upgraded to use the modern Windows experience instead of the old Windows 7 experience.

- System.Windows.Forms.FileDialog

  - Added support for ClientGuid.

    `ClientGuid` enables a calling application to associate a GUID with a dialog's persisted state. A dialog's state can include factors such as the last visited folder and the position and size of the dialog. Typically, this state is persisted based on the name of the executable file. With `ClientGuid`, an application can persist different states of the dialog within the same application.

- System.Windows.Forms.TextRenderer

  Support added for ReadOnlySpan<T> to enhance performance of rendering text.

See also

- Breaking changes in Windows Forms
- Tutorial: Create a new WinForms app (Windows Forms .NET)
- How to migrate a Windows Forms desktop app to .NET 5

Desktop Guide (Windows Forms .NET)

Article • 06/02/2023

Welcome to the Desktop Guide for Windows Forms, a UI framework that creates rich desktop client apps for Windows. The Windows Forms development platform supports a broad set of app development features, including controls, graphics, data binding, and user input. Windows Forms features a drag-and-drop visual designer in Visual Studio to easily create Windows Forms apps.

There are two implementations of Windows Forms:

1. The open-source implementation hosted on GitHub.

   This version runs on .NET 6+. The Windows Forms Visual Designer requires, at a minimum, Visual Studio 2019 version 16.8.

   The latest version is Windows Forms for .NET 7 using Visual Studio 2022 version 17.4.

2. The .NET Framework 4 implementation that's supported by Visual Studio 2022, Visual Studio 2019, and Visual Studio 2017.

   .NET Framework 4 is a Windows-only version of .NET and is considered a Windows Operating System component. This version of Windows Forms is distributed with .NET Framework.

This Desktop Guide is written for Windows Forms on .NET 5 and later versions. For more information about the .NET Framework version of Windows Forms, see Windows Forms for .NET Framework.

Introduction

Windows Forms is a UI framework for building Windows desktop apps. It provides one of the most productive ways to create desktop apps based on the visual designer provided in Visual Studio. Functionality such as drag-and-drop placement of visual controls makes it easy to build desktop apps.

With Windows Forms, you develop graphically rich apps that are easy to deploy, update, and work while offline or while connected to the internet. Windows Forms apps can access the local hardware and file system of the computer where the app is running.

To learn how to create a Windows Forms app, see Tutorial: Create a new WinForms app.

Why migrate from .NET Framework

Windows Forms for .NET provides new features and enhancements over .NET Framework. For more information, see What's new in Windows Forms for .NET 7. To learn how to upgrade an app, see How to upgrade a Windows Forms desktop app to .NET 7

Build rich, interactive user interfaces

Windows Forms is a UI technology for .NET, a set of managed libraries that simplify common app tasks such as reading and writing to the file system. When you use a development environment like Visual Studio, you can create Windows Forms smart-client apps that display information, request input from users, and communicate with remote computers over a network.

In Windows Forms, a *form* is a visual surface on which you display information to the user. You ordinarily build Windows Forms apps by adding controls to forms and developing responses to user actions, such as mouse clicks or key presses. A *control* is a discrete UI element that displays data or accepts data input.

When a user does something to your form or one of its controls, the action generates an event. Your app reacts to these events with code, and processes the events when they occur.

Windows Forms contains a variety of controls that you can add to forms: controls that display text boxes, buttons, drop-down boxes, radio buttons, and even webpages. If an existing control doesn't meet your needs, Windows Forms also supports creating your own custom controls using the UserControl class.

Windows Forms has rich UI controls that emulate features in high-end apps like Microsoft Office. When you use the ToolStrip and MenuStrip controls, you can create toolbars and menus that contain text and images, display submenus, and host other controls such as text boxes and combo boxes.

With the drag-and-drop **Windows Forms Designer** in Visual Studio, you can easily create Windows Forms apps. Just select the controls with your cursor and place them where you want on the form. The designer provides tools such as gridlines and snap lines to take the hassle out of aligning controls. You can use the FlowLayoutPanel, TableLayoutPanel, and SplitContainer controls to create advanced form layouts in less time.

Finally, if you must create your own custom UI elements, the System.Drawing namespace contains a large selection of classes to render lines, circles, and other shapes directly on a form.

Create forms and controls

For step-by-step information about how to use these features, see the following Help topics.

- How to add a form to a project
- How to add Controls to a form

Display and manipulate data

Many apps must display data from a database, XML or JSON file, web service, or other data source. Windows Forms provides a flexible control that is named the DataGridView control for displaying such tabular data in a traditional row and column format, so that every piece of data occupies its own cell. When you use DataGridView, you can customize the appearance of individual cells, lock arbitrary rows and columns in place, and display complex controls inside cells, among other features.

Connecting to data sources over a network is a simple task with Windows Forms. The BindingSource component represents a connection to a data source, and exposes methods for binding data to controls, navigating to the previous and next records, editing records, and saving changes back to the original source. The BindingNavigator control provides a simple interface over the BindingSource component for users to navigate between records.

You can create data-bound controls easily by using the Data Sources window in Visual Studio. The window displays data sources such as databases, web services, and objects in your project. You can create data-bound controls by dragging items from this window onto forms in your project. You can also data-bind existing controls to data by dragging objects from the Data Sources window onto existing controls.

Another type of data binding you can manage in Windows Forms is *settings*. Most apps must retain some information about their run-time state, such as the last-known size of forms, and retain user preference data, such as default locations for saved files. The Application Settings feature addresses these requirements by providing an easy way to store both types of settings on the client computer. After you define these settings by using either Visual Studio or a code editor, the settings are persisted as XML and automatically read back into memory at run time.

Deploy apps to client computers

After you have written your app, you must send the app to your users so that they can install and run it on their own client computers. When you use the ClickOnce technology, you can deploy your apps from within Visual Studio by using just a few clicks, and provide your users with a URL pointing to your app on the web. ClickOnce manages all the elements and dependencies in your app, and ensures that the app is correctly installed on the client computer.

ClickOnce apps can be configured to run only when the user is connected to the network, or to run both online and offline. When you specify that an app should support offline operation, ClickOnce adds a link to your app in the user's **Start** menu. The user can then open the app without using the URL.

When you update your app, you publish a new deployment manifest and a new copy of your app to your web server. ClickOnce will detect that there is an update available and upgrade the user's installation. No custom programming is required to update old apps.

See also

- Tutorial: Create a new WinForms app
- How to add a form to a project
- Add a control

Tutorial: Create a Windows Forms app with .NET

Article • 02/08/2023

In this short tutorial, you'll learn how to create a new Windows Forms app with Visual Studio. Once the initial app has been generated, you'll learn how to add controls and how to handle events. By the end of this tutorial, you'll have a simple app that adds names to a list box.

In this tutorial, you learn how to:

- ✓ Create a new Windows Forms app
- ✓ Add controls to a form
- ✓ Handle control events to provide app functionality
- ✓ Run the app

Prerequisites

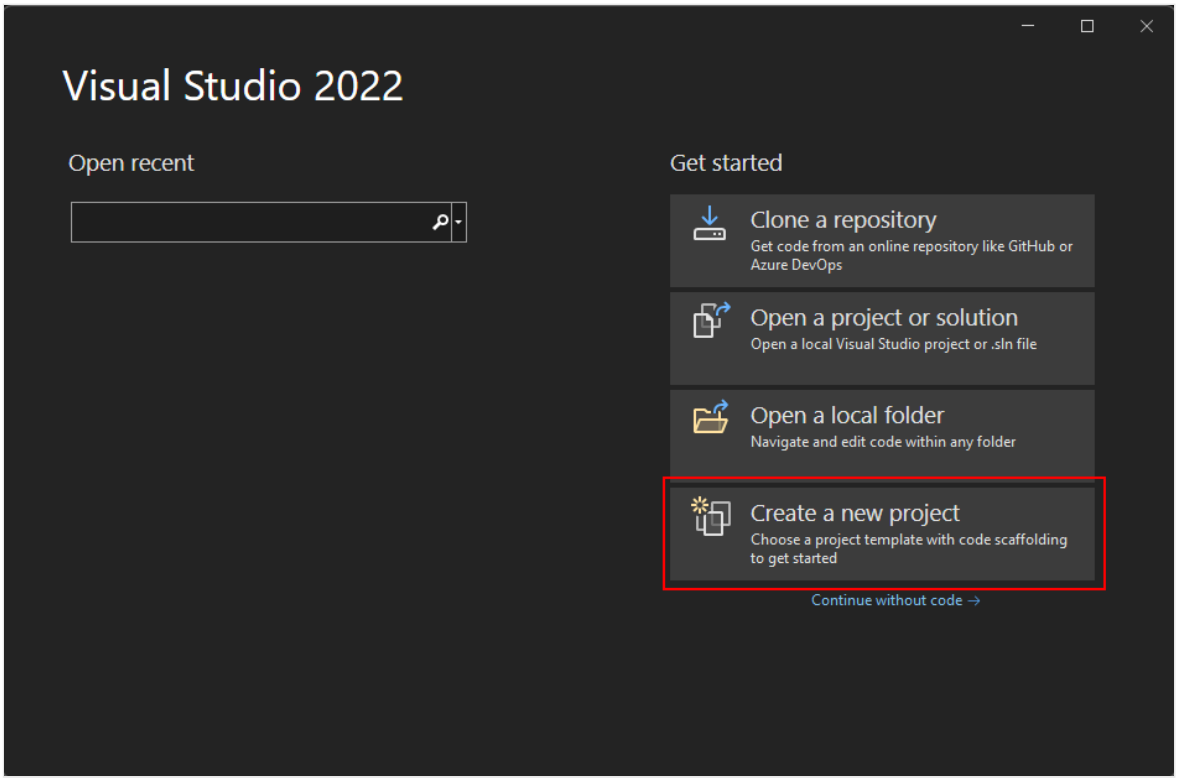
- Visual Studio 2022 version 17.0 or later versions
  - Select the .NET desktop development workload
  - Select the .NET 6 individual component

> **Tip**
>
> Use Visual Studio 2022 version 17.4 or later and install both the .NET 7 and .NET 6 individual components. Support for .NET 7 was added in Visual Studio 2022 version 17.4.

Create a Windows Forms app

The first step to creating a new app is opening Visual Studio and generating the app from a template.

1. Open Visual Studio.
2. Select **Create a new project**.



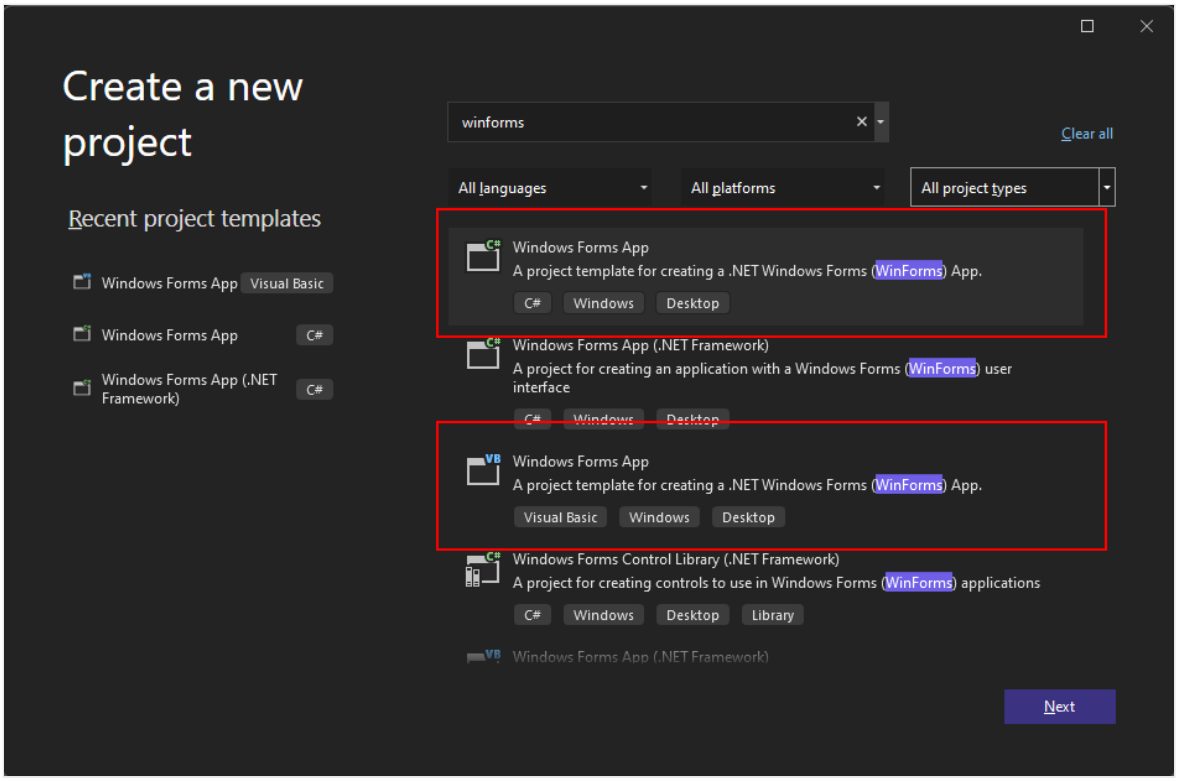
3. In the **Search for templates** box, type **winforms**, and wait for the search results to appear.

4. In the **code language** dropdown, choose **C#** or **Visual Basic**.

5. In the list of templates, select **Windows Forms App** and then click **Next**.

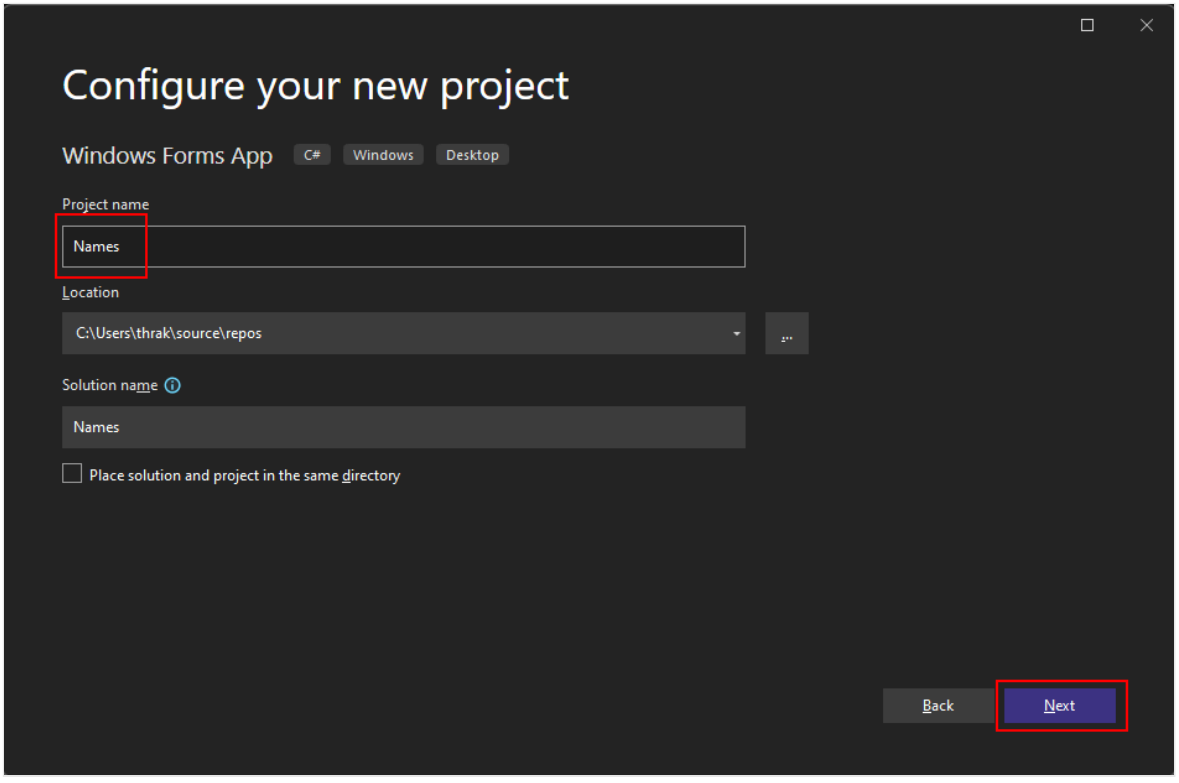
   > **ⓘ Important**
   >
   > Don't select the **Windows Forms App (.NET *Framework*)** template.

   The following image shows both C# and Visual Basic .NET project templates. If you applied the **code language** filter, you'll see the corresponding template.

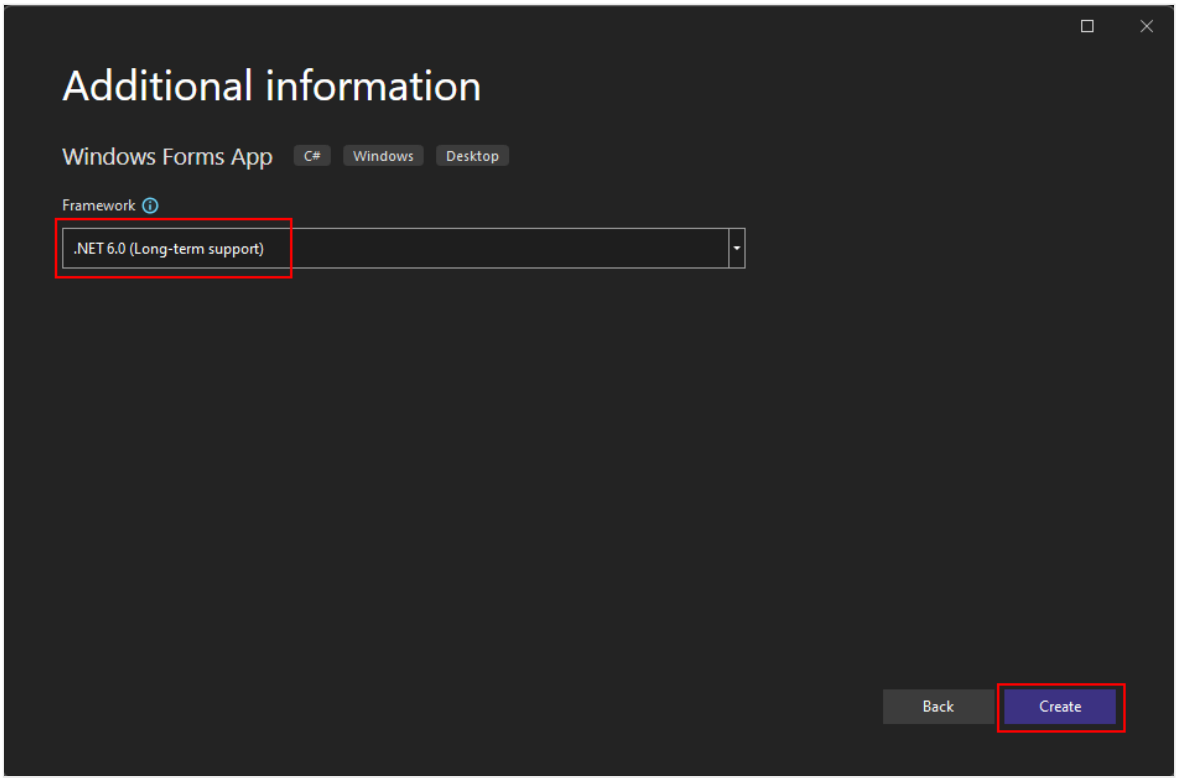


6. In the **Configure your new project** window, set the **Project name** to *Names* and click **Next**.

   You can also save your project to a different folder by adjusting the **Location** path.



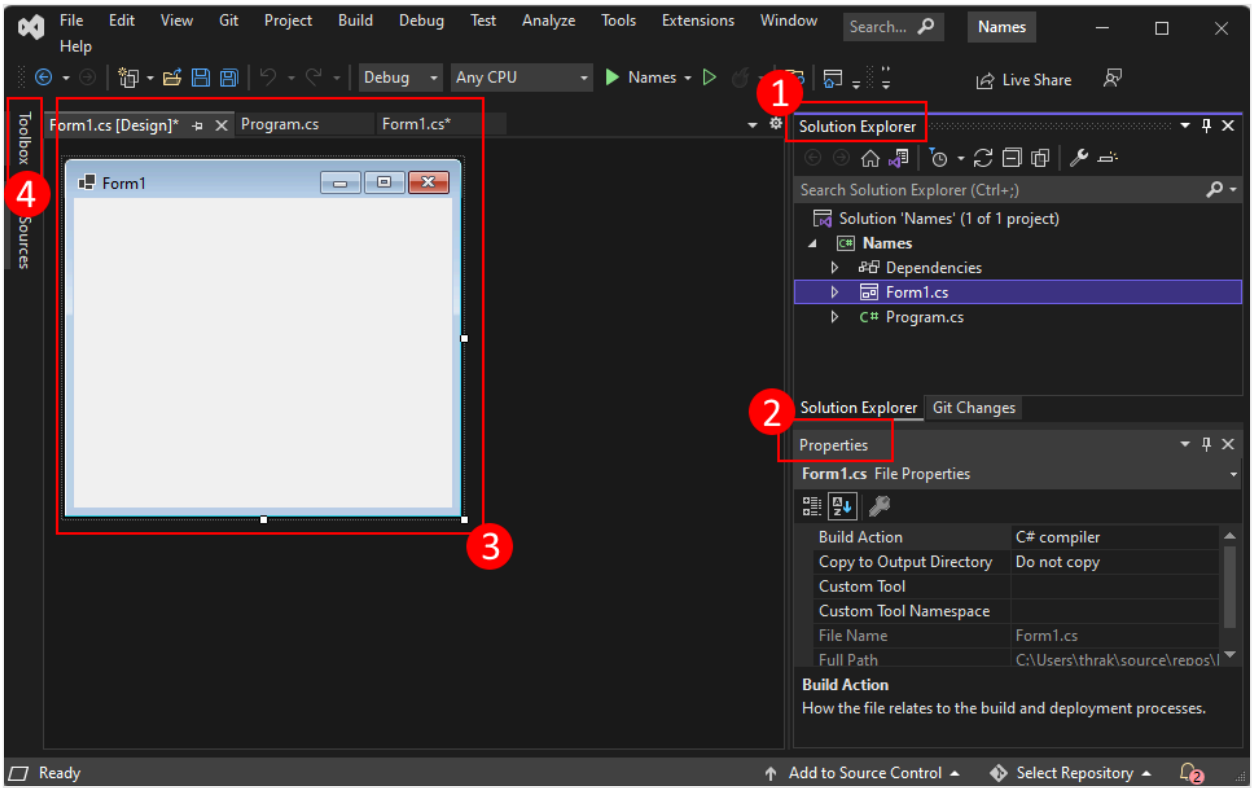
7. Finally, in the **Additional information** window, select **.NET 6.0 (Long-term support)** for the **Framework** setting, and then click **Create**.



Once the app is generated, Visual Studio should open the designer pane for the default form, *Form1*. If the form designer isn't visible, double-click on the form in the **Solution Explorer** pane to open the designer window.

Important parts of Visual Studio

Support for Windows Forms in Visual Studio has four important components that you'll interact with as you create an app:



1. Solution Explorer

   All of your project files, code, forms, resources, will appear in this pane.

2. Properties

   This pane shows property settings you can configure based on the item selected. For example, if you select an item from **Solution Explorer**, you'll see property settings related to the file. If you select an object in the **Designer**, you'll see settings for the control or form.

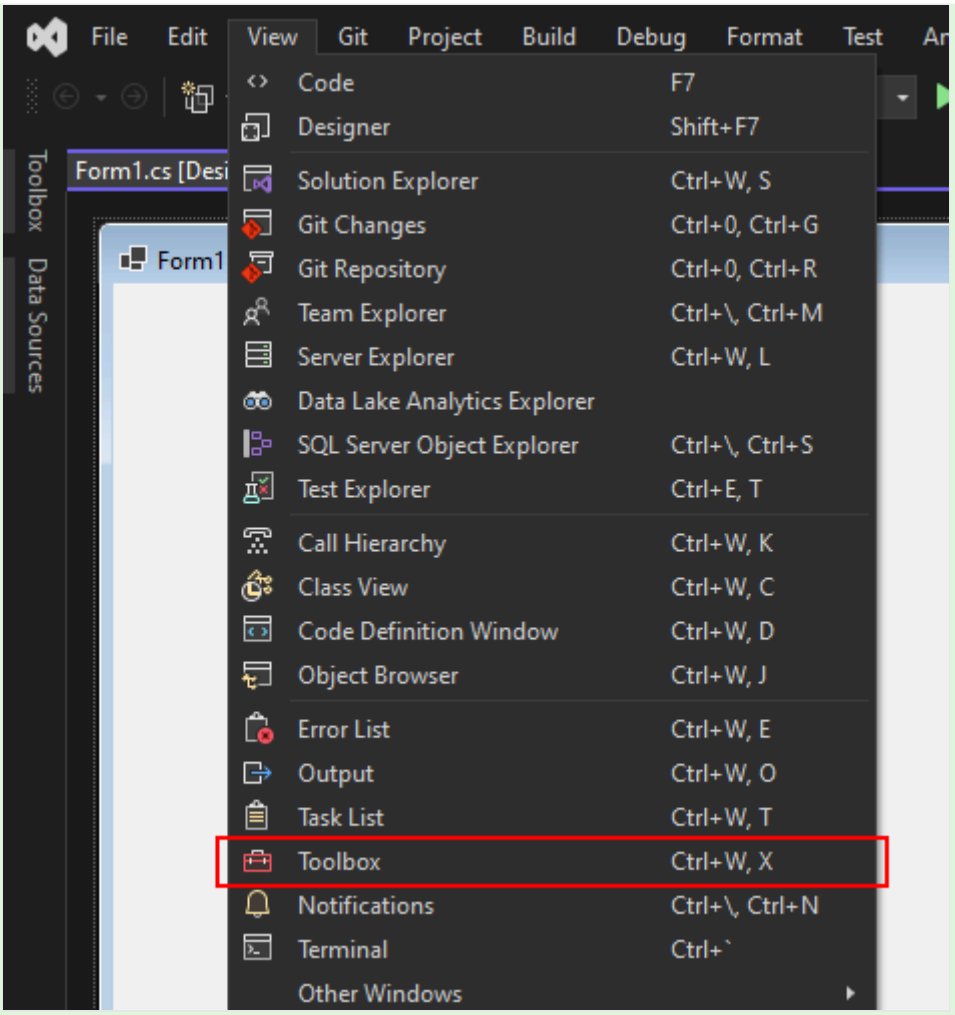
3. Form Designer

   This is the designer for the form. It's interactive and you can drag-and-drop objects from the **Toolbox**. By selecting and moving items in the designer, you can visually compose the user interface (UI) for your app.

4. Toolbox

   The toolbox contains all of the controls you can add to a form. To add a control to the current form, double-click a control or drag-and-drop the control.

> **Tip**
>
> If the toolbox isn't visible, you can display it through the **View** > **Toolbox** menu item.



Add controls to the form

With the *Form1* form designer open, use the **Toolbox** pane to add the following controls to the form:

- Label
- Button
- Listbox
- Textbox

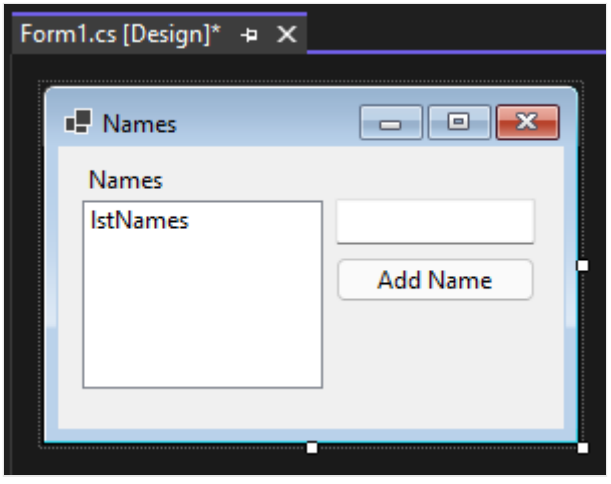
You can position and size the controls according to the following settings. Either visually move them to match the screenshot that follows, or click on each control and configure the settings in the **Properties** pane. You can also click on the form title area to select the form:

Expand table

Object	Setting	Value
Form	Text	`Names`

Object	Setting	Value
	Size	`268, 180`
Label	Location	`12, 9`
	Text	`Names`
Listbox	Name	`lstNames`
	Location	`12, 27`
	Size	`120, 94`
Textbox	Name	`txtName`
	Location	`138, 26`
	Size	`100, 23`
Button	Name	`btnAdd`
	Location	`138, 55`
	Size	`100, 23`
	Text	`Add Name`

You should have a form in the designer that looks similar to the following:



Handle events

Now that the form has all of its controls laid out, you need to handle the events of the controls to respond to user input. With the form designer still open, perform the following steps:

1. Select the button control on the form.

2. In the **Properties** pane, click on the events icon  to list the events of the button.

3. Find the **Click** event and double-click it to generate an event handler.

   This action adds the following code to the the form:

   ```
   C#

   private void btnAdd_Click(object sender, EventArgs e)
   {

   }
   ```

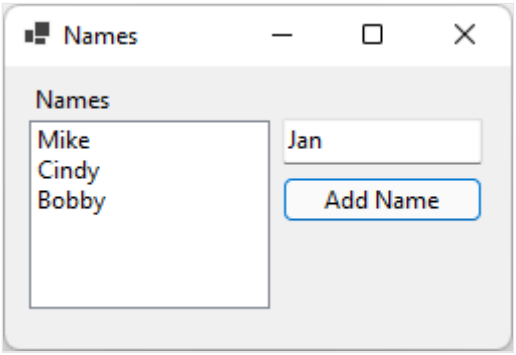
   The code we'll put in this handler will add the name specified by the `txtName` textbox control to the `lstNames` listbox control. However, we want there to be two conditions to adding the name: the name provided must not be blank, and the name must not already exist.

4. The following code demonstrates adding a name to the `lstNames` control:

   ```
   C#

   private void btnAdd_Click(object sender, EventArgs e)
   {
       if (!string.IsNullOrWhiteSpace(txtName.Text) &&
   !lstNames.Items.Contains(txtName.Text))
           lstNames.Items.Add(txtName.Text);
   }
   ```

Run the app

Now that the event has been coded, you can run the app by pressing the `F5` key or by selecting **Debug** > **Start Debugging** from the menu. The form displays and you can enter a name in the textbox and then add it by clicking the button.



Next steps

Learn more about Windows Forms

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Upgrade a .NET Framework Windows Forms desktop app to .NET 8

Article • 04/11/2024

This article describes how to upgrade a Windows Forms desktop app to .NET 8 using the Upgrade Assistant. Even though Windows Forms runs on .NET, a cross-platform technology, Windows Forms is still a Windows-only framework. The following Windows Forms-related project types can be upgraded with the .NET Upgrade Assistant:

- Windows Forms project
- Control library
- .NET library

Prerequisites

- Windows Operating System.
- Download and extract the demo app used with this article.
- Visual Studio 2022 version 17.8 or later to target .NET 8.
- .NET Upgrade Assistant extension for Visual Studio.

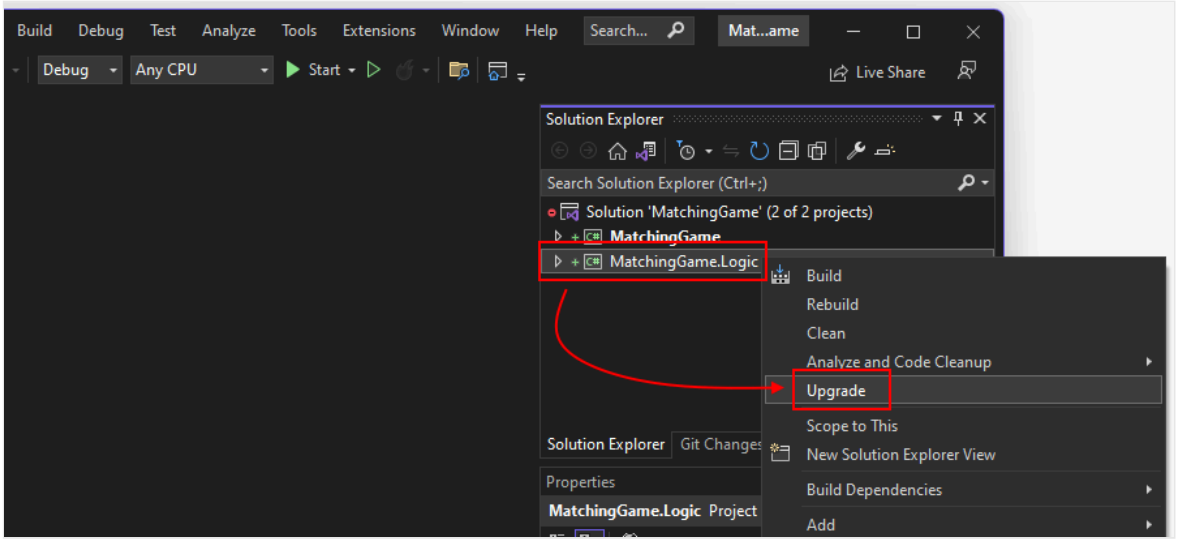
Upgrade the dependencies first

If you're upgrading multiple projects, start with projects that have no dependencies. In the Matching Game sample, the **MatchingGame** project depends on the **MatchingGame.Logic** library, so **MatchingGame.Logic** should be upgraded first.

> **Tip**
>
> Be sure to have a backup of your code, such as in source control or a copy.

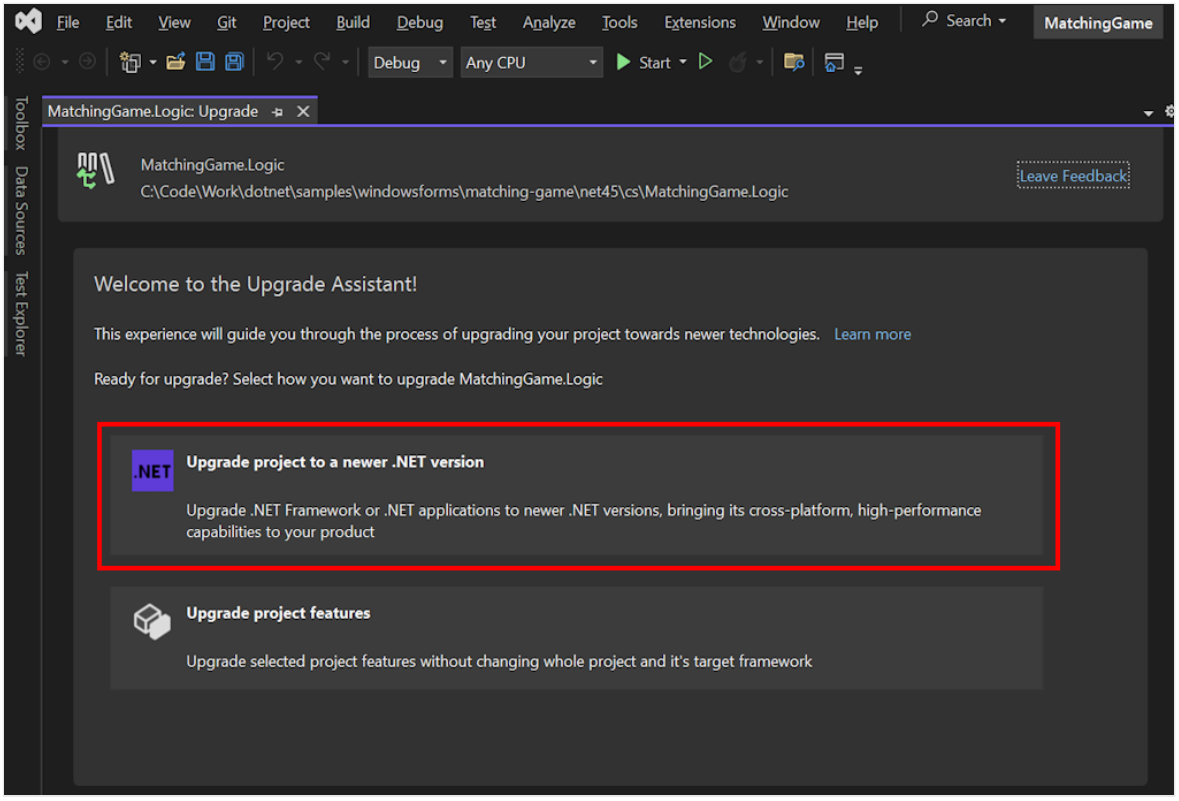
Use the following steps to upgrade a project in Visual Studio:

1. Right-click on the **MatchingGame.Logic** project in the **Solution Explorer** window and select **Upgrade**:

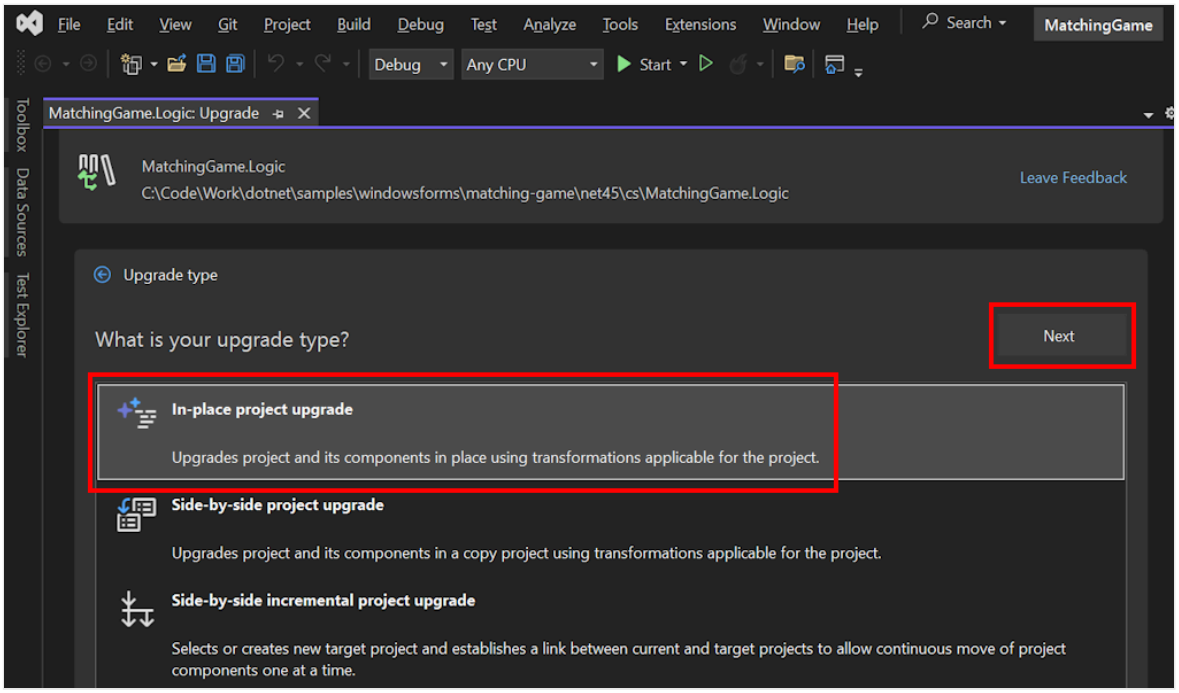


A new tab is opened that prompts you to choose which type of upgrade you want to perform.

2. Select **Upgrade project to a newer .NET version**.

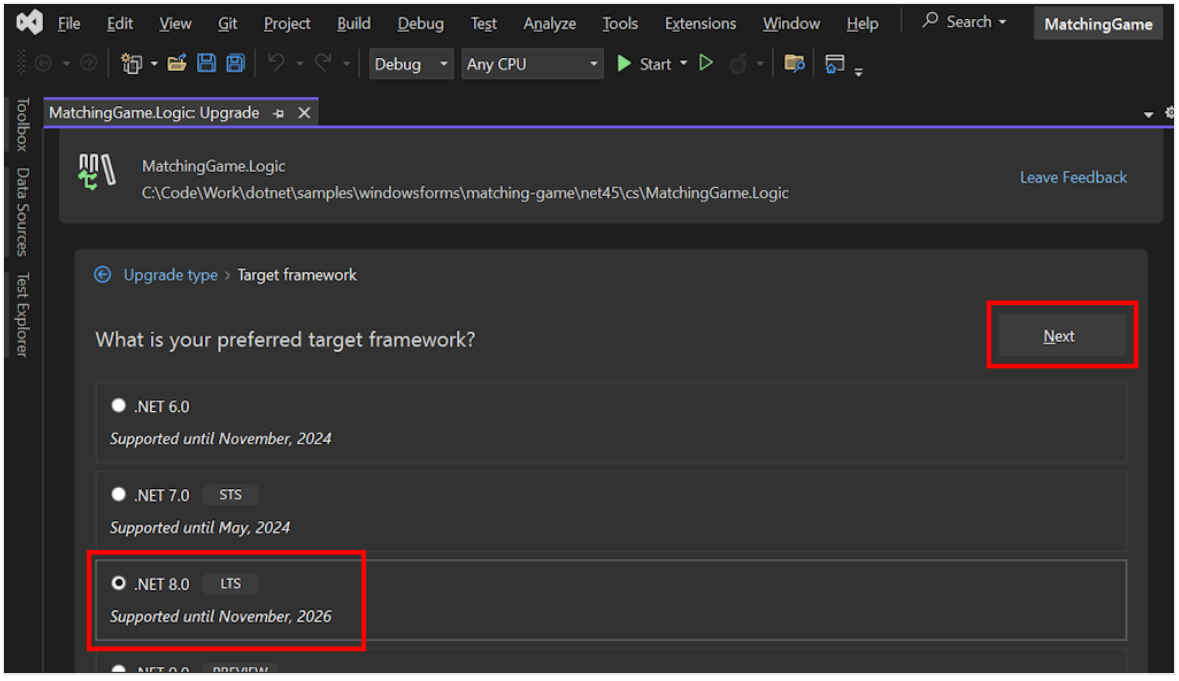


3. Select **In-place project upgrade**.



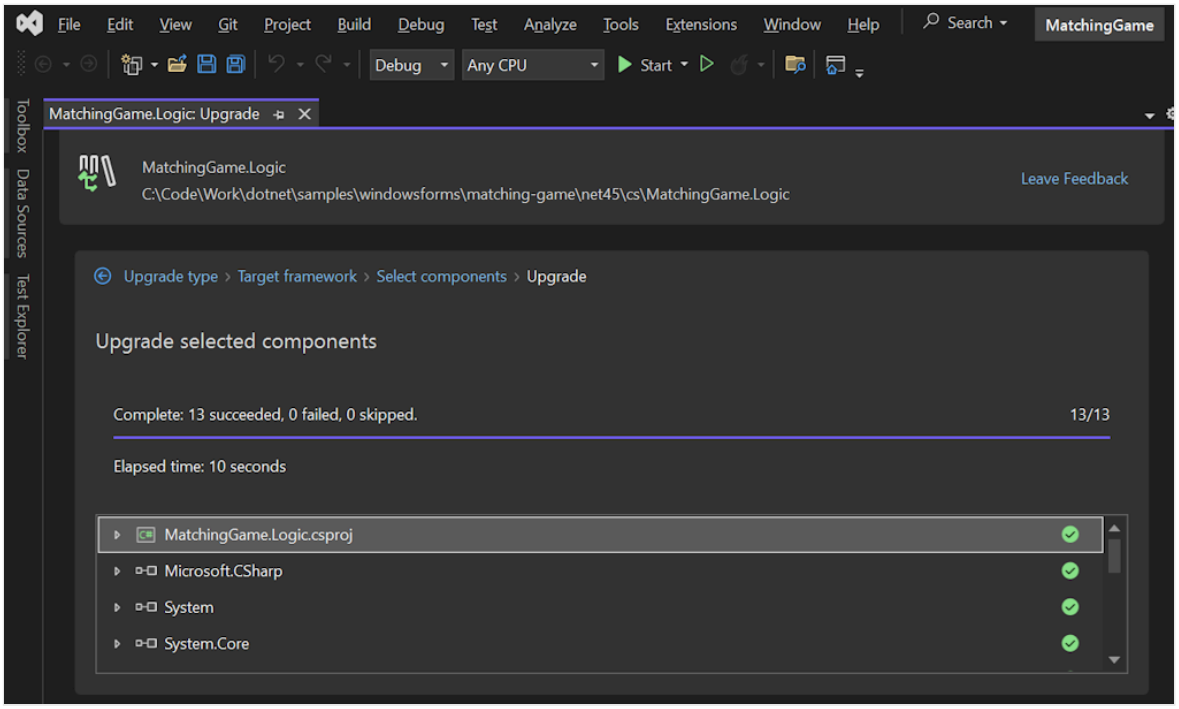
4. Next, select the target framework. Based on the type of project you're upgrading, you're presented with different options. **.NET Standard 2.0** can be used by both .NET Framework and .NET. This is a good choice if the library doesn't rely on a desktop technology like Windows Forms. However, the latest .NET releases provide many language and compiler improvements over .NET Standard.

   Select **.NET 8.0** and then select **Next**.



5. A tree is shown with all of the artifacts related to the project, such as code files and libraries. You can upgrade individual artifacts or the entire project, which is the default. Select **Upgrade selection** to start the upgrade.

6. When the upgrade is finished, the results are displayed:



Artifacts with a solid green circle were upgraded while empty green circles were skipped. Skipped artifacts mean that the upgrade assistant didn't find anything to upgrade.

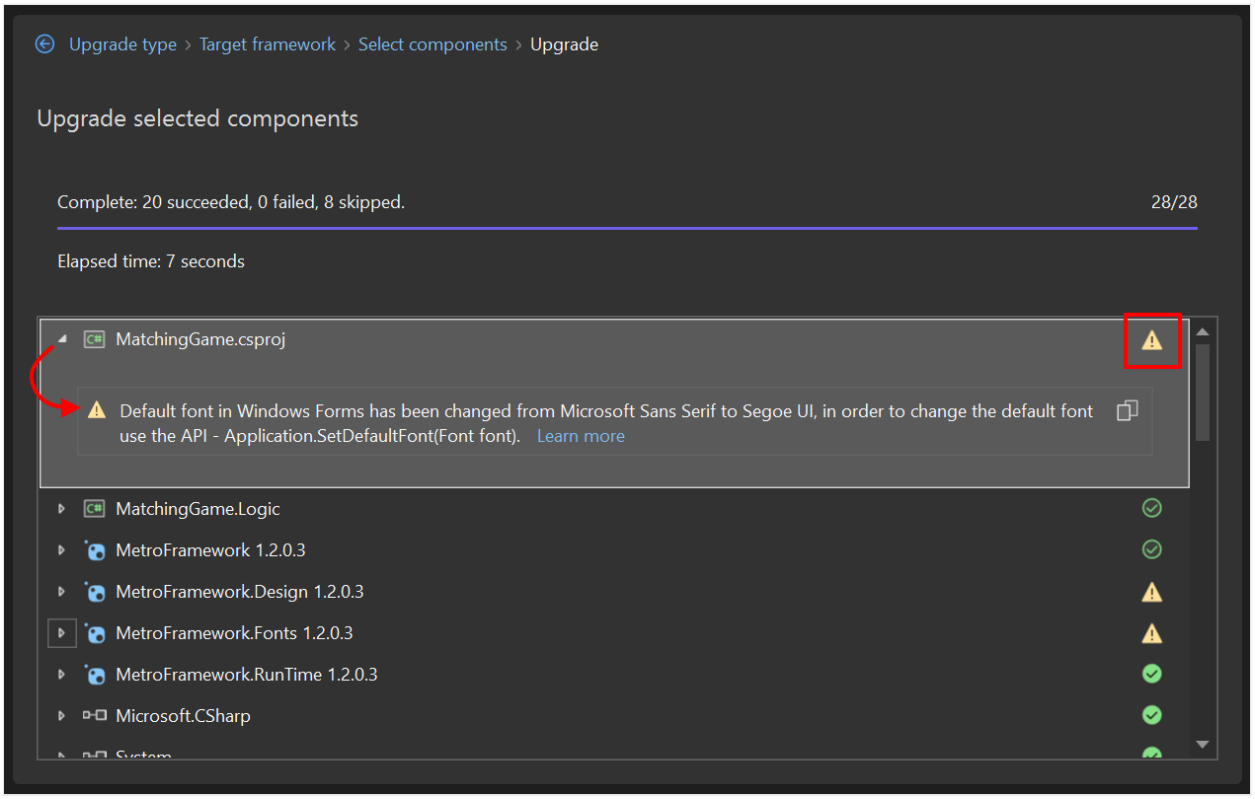
Now that the app's supporting library is upgraded, upgrade the main app.

Upgrade the main project

Once all of the supporting libraries are upgraded, the main app project can be upgraded. With the example app, there's only one library project to upgrade, which was upgraded in the previous section.

1. Right-click on the **MatchingGame** project in the **Solution Explorer** window and select **Upgrade**:
2. Select **Upgrade project to a newer .NET version**.
3. Select **In-place project upgrade** as the upgrade mode.
4. Select **.NET 8.0** for the target framework and select **Next**.
5. Leave all of the artifacts selected and select **Upgrade selection**.

After the upgrade is complete, the results are shown. Notice how the Windows Forms project has a warning symbol. Expand that item and more information is shown about that step:



Notice that the project upgrade component mentions that the default font has changed. Because the font may affect control layout, you need to check every form and custom control in your project to ensure the UI is arranged correctly.

Generate a clean build

After your main project is upgraded, clean and compile it.

1. Right-click on the **MatchingGame** project in the **Solution Explorer** window and select **Clean**.
2. Right-click on the **MatchingGame** project in the **Solution Explorer** window and select **Build**.

If your application encountered any errors, you can find them in the **Error List** window with a recommendation how to fix them.

The **Windows Forms Matching Game Sample** project is now upgraded to .NET 8.

Related content

- Porting from .NET Framework to .NET.

  The porting guide provides an overview of what you should consider when porting your code from .NET Framework to .NET. The complexity of your projects dictates how much work you'll do after the initial migration of the project files.

- Modernize after upgrading to .NET from .NET Framework.

  The world of .NET has changed a lot since .NET Framework. This link provides some information about how to modernize your app after you've upgraded it.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Events overview (Windows Forms .NET)

Article • 10/28/2020

An event is an action that you can respond to, or "handle," in code. Events can be generated by a user action, such as clicking the mouse or pressing a key, by program code, or by the system.

Event-driven applications execute code in response to an event. Each form and control exposes a predefined set of events that you can program against. If one of these events occurs and there's code an associated event handler, that code is invoked.

The types of events raised by an object vary, but many types are common to most controls. For example, most objects will handle a Click event. If a user clicks a form, code in the form's Click event handler is executed.

> **Note**
>
> Many events occur in conjunction with other events. For example, in the course of the DoubleClick event occurring, the MouseDown, MouseUp, and Click events occur.

For information about how to raise and consume an event, see Handling and raising events.

Delegates and their role

Delegates are classes commonly used within .NET to build event-handling mechanisms. Delegates roughly equate to function pointers, commonly used in Visual C++ and other object-oriented languages. Unlike function pointers however, delegates are object-oriented, type-safe, and secure. Also, where a function pointer contains only a reference to a particular function, a delegate consists of a reference to an object, and references to one or more methods within the object.

This event model uses *delegates* to bind events to the methods that are used to handle them. The delegate enables other classes to register for event notification by specifying a handler method. When the event occurs, the delegate calls the bound method. For more information about how to define delegates, see Handling and raising events.

Delegates can be bound to a single method or to multiple methods, referred to as multicasting. When creating a delegate for an event, you typically create a multicast event. A rare exception might be an event that results in a specific procedure (such as

displaying a dialog box) that wouldn't logically repeat multiple times per event. For information about how to create a multicast delegate, see How to combine delegates (Multicast Delegates).

A multicast delegate maintains an invocation list of the methods it's bound to. The multicast delegate supports a Combine method to add a method to the invocation list and a Remove method to remove it.

When an event is recorded by the application, the control raises the event by invoking the delegate for that event. The delegate in turn calls the bound method. In the most common case (a multicast delegate), the delegate calls each bound method in the invocation list in turn, which provides a one-to-many notification. This strategy means that the control doesn't need to maintain a list of target objects for event notification—the delegate handles all registration and notification.

Delegates also enable multiple events to be bound to the same method, allowing a many-to-one notification. For example, a button-click event and a menu-command–click event can both invoke the same delegate, which then calls a single method to handle these separate events the same way.

The binding mechanism used with delegates is dynamic: a delegate can be bound at run-time to any method whose signature matches that of the event handler. With this feature, you can set up or change the bound method depending on a condition and to dynamically attach an event handler to a control.

See also

- Handling and raising events

Automatic scaling (Windows Forms .NET)

Article • 10/28/2020

Automatic scaling enables a form and its controls, designed on one machine with a certain display resolution or font, to be displayed appropriately on another machine with a different display resolution or font. It assures that the form and its controls will intelligently resize to be consistent with native windows and other applications on both the users' and other developers' machines. Automatic scaling and visual styles enable Windows Forms applications to maintain a consistent look-and-feel when compared to native Windows applications on each user's machine.

For the most part, automatic scaling works as expected in Windows Forms. However, font scheme changes can be problematic.

Need for automatic scaling

Without automatic scaling, an application designed for one display resolution or font will either appear too small or too large when that resolution or font is changed. For example, if the application is designed using Tahoma 9 point as a baseline, without adjustment it will appear too small if run on a machine where the system font is Tahoma 12 point. Text elements, such as titles, menus, text box contents, and so on will render smaller than other applications. Furthermore, the size of user interface (UI) elements that contain text, such as the title bar, menus, and many controls are dependent on the font used. In this example, these elements will also appear relatively smaller.

An analogous situation occurs when an application is designed for a certain display resolution. The most common display resolution is 96 dots per inch (DPI), which equals 100% display scaling, but higher resolution displays supporting 125%, 150%, 200% (which respectively equal 120, 144 and 192 DPI) and above are becoming more common. Without adjustment, an application, especially a graphics-based one, designed for one resolution will appear either too large or too small when run at another resolution.

Automatic scaling seeks to address these problems by automatically resizing the form and its child controls according to the relative font size or display resolution. The Windows operating system supports automatic scaling of dialog boxes using a relative unit of measurement called dialog units. A dialog unit is based on the system font and its relationship to pixels can be determined though the Win32 SDK function `GetDialogBaseUnits`. When a user changes the theme used by Windows, all dialog boxes

are automatically adjusted accordingly. In addition, Windows Forms supports automatic scaling either according to the default system font or the display resolution. Optionally, automatic scaling can be disabled in an application.

> ⊗ **Caution**
>
> Arbitrary mixtures of DPI and font scaling modes are not supported. Although you may scale a user control using one mode (for example, DPI) and place it on a form using another mode (Font) with no issues, but mixing a base form in one mode and a derived form in another can lead to unexpected results.

Automatic scaling in action

Windows Forms uses the following logic to automatically scale forms and their contents:

1. At design time, each ContainerControl records the scaling mode and it current resolution in the AutoScaleMode and AutoScaleDimensions, respectively.
2. At run time, the actual resolution is stored in the CurrentAutoScaleDimensions property. The AutoScaleFactor property dynamically calculates the ratio between the run-time and design-time scaling resolution.
3. When the form loads, if the values of CurrentAutoScaleDimensions and AutoScaleDimensions are different, then the PerformAutoScale method is called to scale the control and its children. This method suspends layout and calls the Scale method to perform the actual scaling. Afterwards, the value of AutoScaleDimensions is updated to avoid progressive scaling.
4. PerformAutoScale is also automatically invoked in the following situations:
    - In response to the OnFontChanged event if the scaling mode is Font.
    - When the layout of the container control resumes and a change is detected in the AutoScaleDimensions or AutoScaleMode properties.
    - As implied above, when a parent ContainerControl is being scaled. Each container control is responsible for scaling its children using its own scaling factors and not the one from its parent container.
5. Child controls can modify their scaling behavior through several means:
    - The ScaleChildren property can be overridden to determine if their child controls should be scaled or not.

- The GetScaledBounds method can be overridden to adjust the bounds that the control is scaled to, but not the scaling logic.
- The ScaleControl method can be overridden to change the scaling logic for the current control.

See also

- AutoScaleMode
- Scale
- PerformAutoScale
- AutoScaleDimensions

How to add a form to a project (Windows Forms .NET)

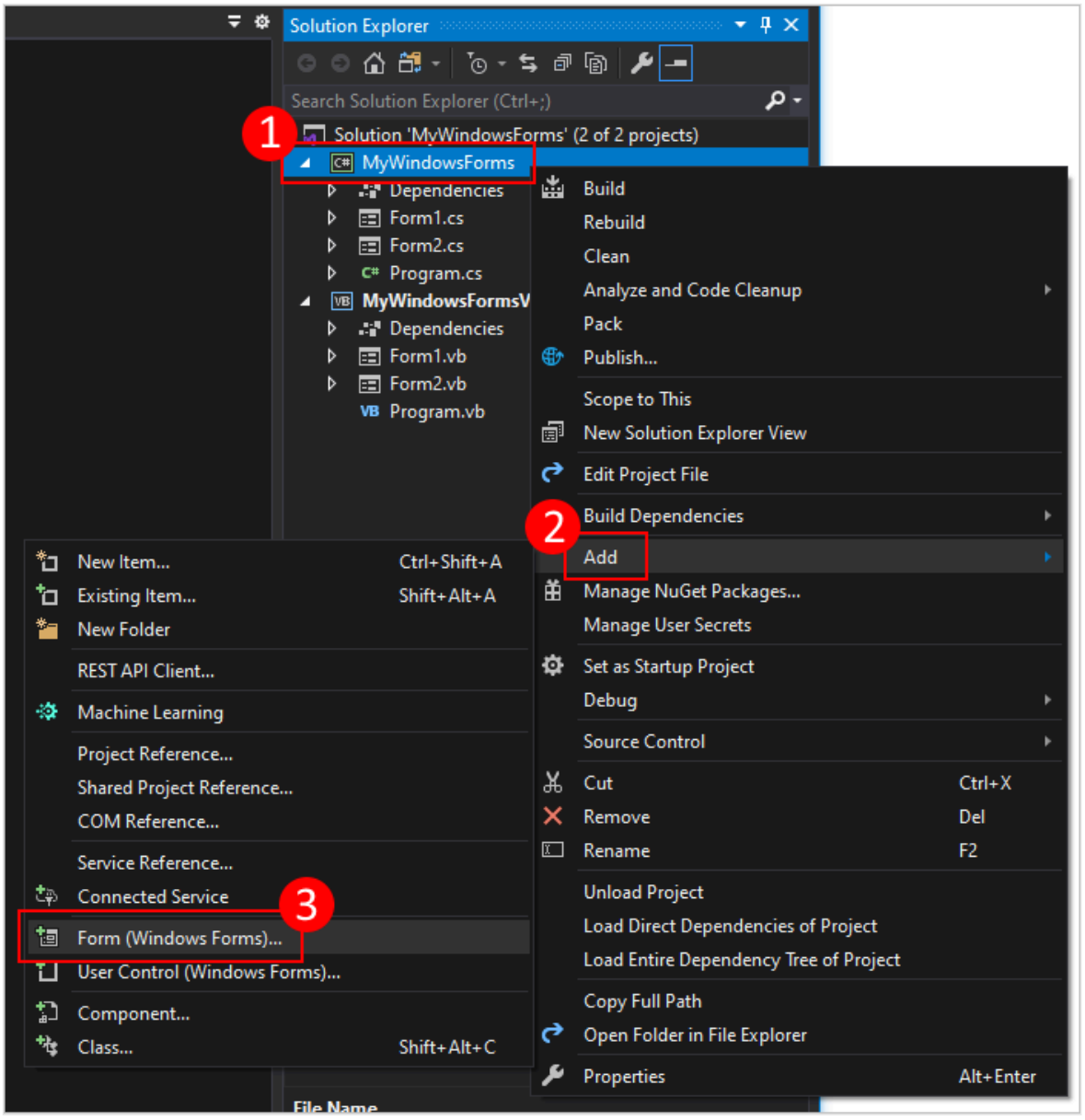
Article • 04/19/2024

Add forms to your project with Visual Studio. When your app has multiple forms, you can choose which is the startup form for your app, and you can display multiple forms at the same time.

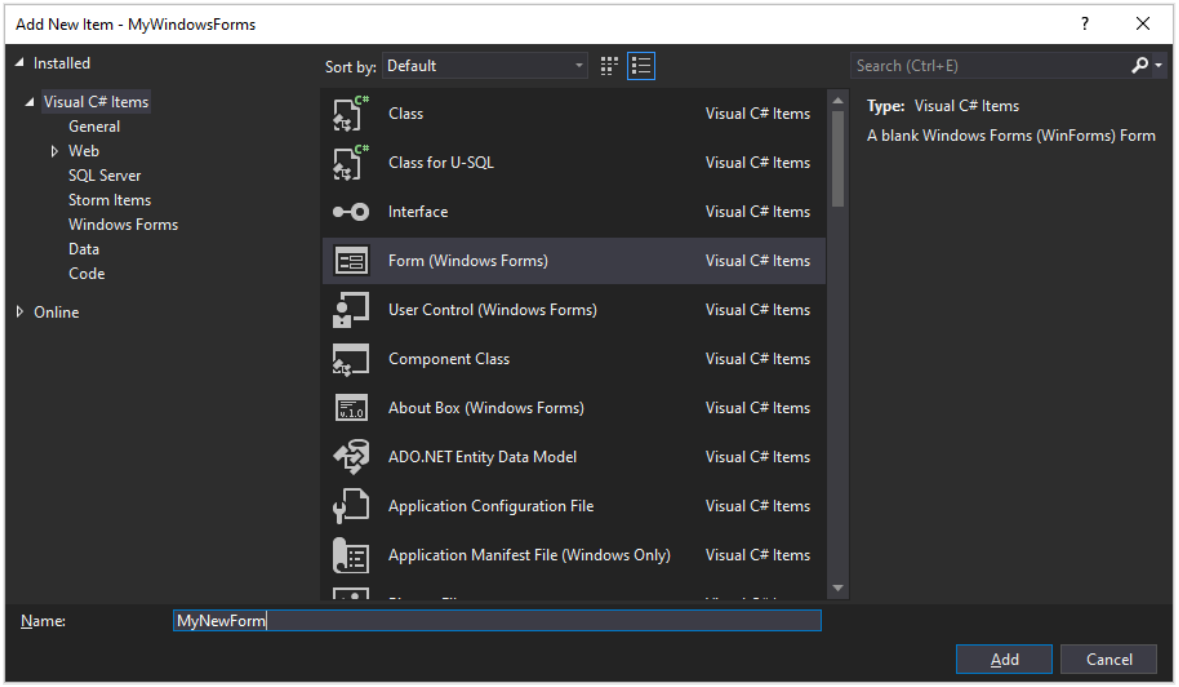
Add a new form

Add a new form with Visual Studio.

1. In Visual Studio, find the **Project Explorer** pane. Right-click on the project and choose **Add** > **Form (Windows Forms)**.



2. In the **Name** box, type a name for your form, such as *MyNewForm*. Visual Studio will provide a default and unique name that you may use.



Once the form has been added, Visual Studio opens the form designer for the form.

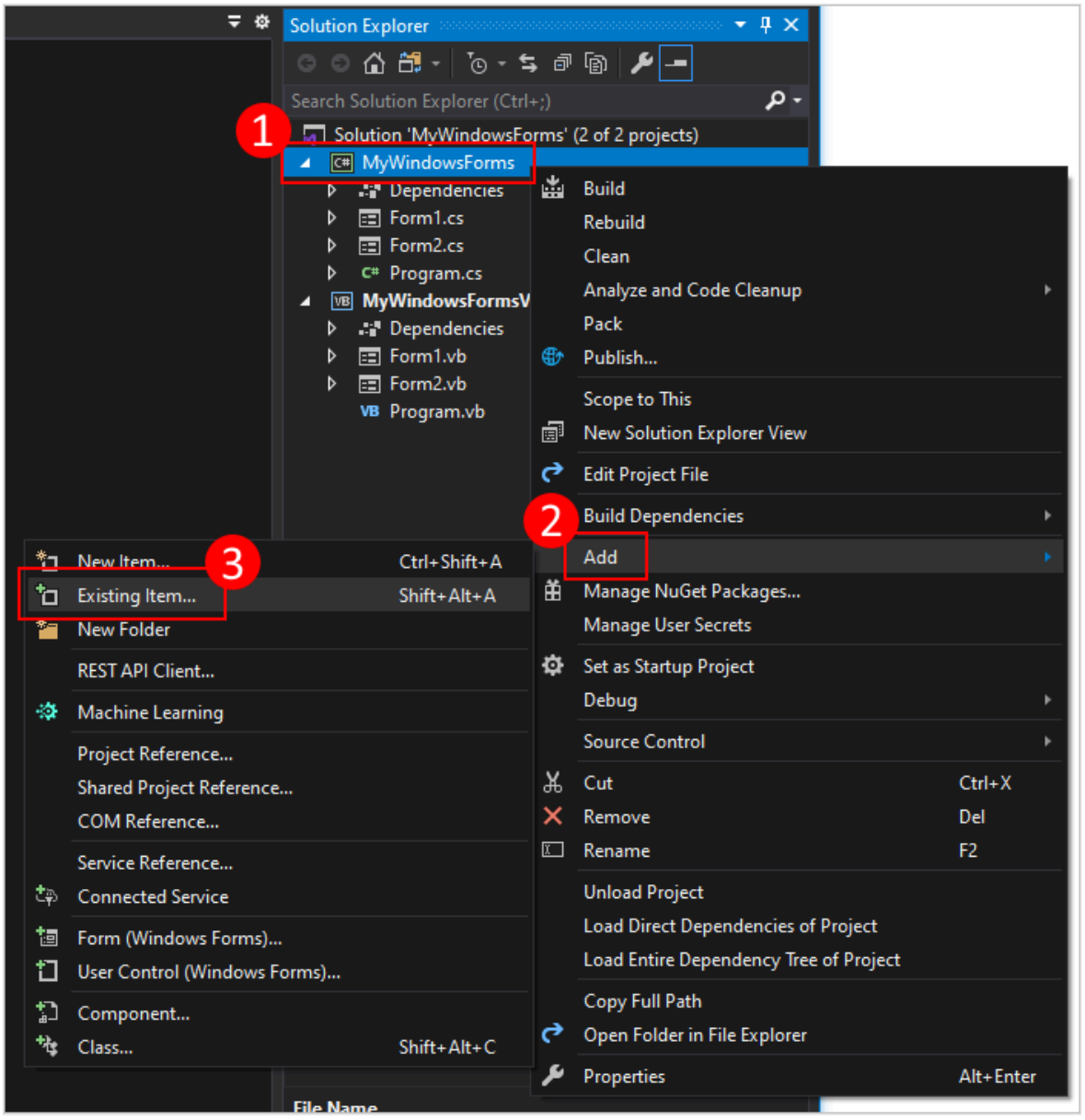
Add a project reference to a form

If you have the source files to a form, you can add the form to your project by copying the files into the same folder as your project. The project automatically references any code files that are in the same folder or child folder of your project.

Forms are made up of two files that share the same name: *form2.cs* (*form2* being an example of a file name) and *form2.Designer.cs*. Sometimes a resource file exists, sharing the same name, *form2.resx*. In in the previous example, *form2* represents the base file name. You'll want to copy all related files to your project folder.

Alternatively, you can use Visual Studio to import a file into your project. When you add an existing file to your project, the file is copied into the same folder as your project.

1. In Visual Studio, find the **Project Explorer** pane. Right-click on the project and choose **Add** > **Existing Item**.



2. Navigate to the folder containing your form files.

3. Select the *form2.cs* file, where *form2* is the base file name of the related form files. Don't select the other files, such as *form2.Designer.cs*.

See also

- How to position and size a form (Windows Forms .NET)
- Events overview (Windows Forms .NET)
- Position and layout of controls (Windows Forms .NET)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

How to position and size a form (Windows Forms .NET)

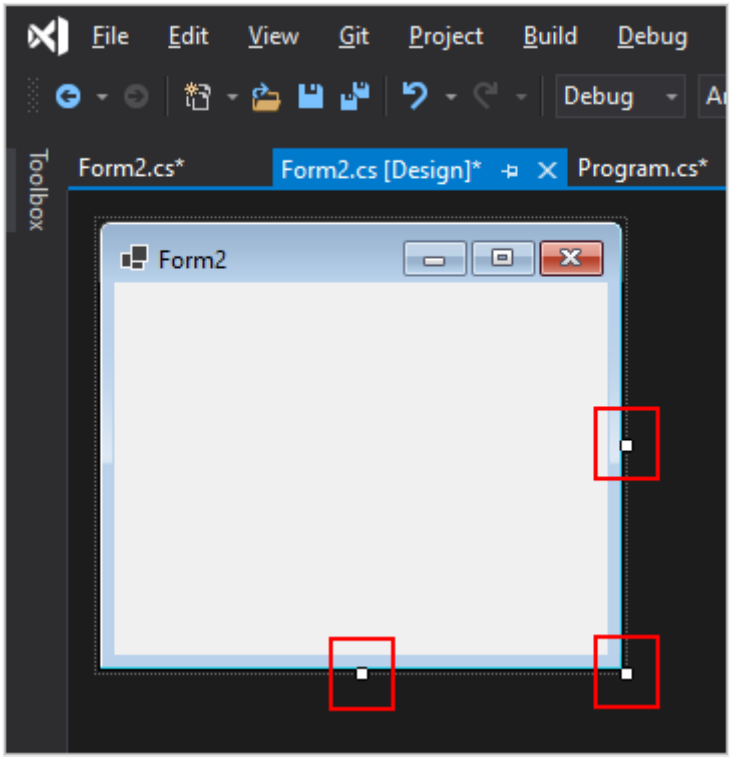
Article • 09/23/2022

When a form is created, the size and location is initially set to a default value. The default size of a form is generally a width and height of *800x500* pixels. The initial location, when the form is displayed, depends on a few different settings.

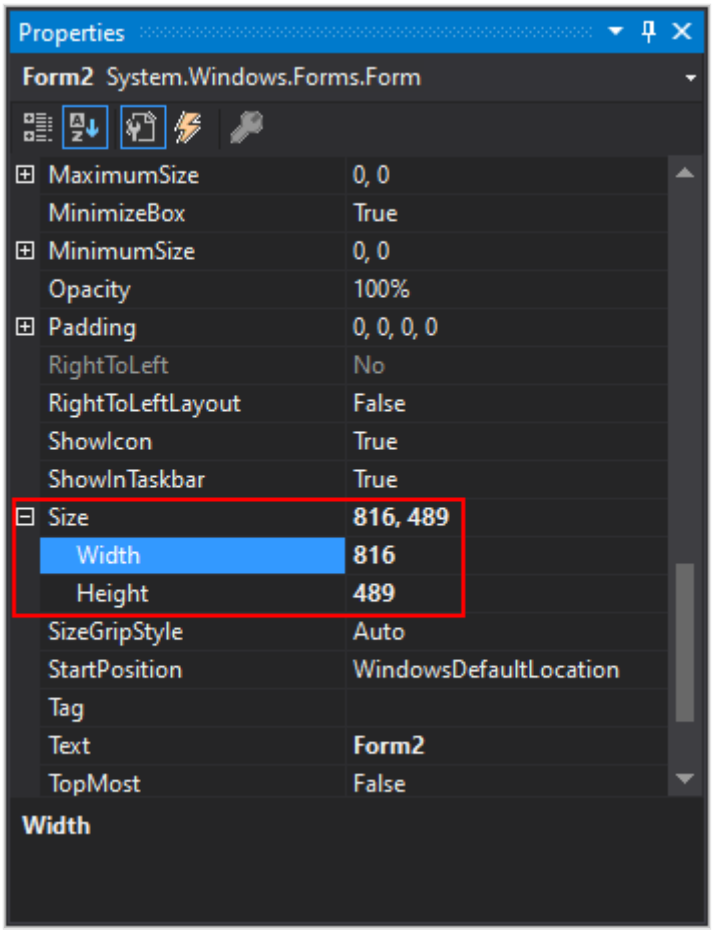
You can change the size of a form at design time with Visual Studio, and at run time with code.

Resize with the designer

After adding a new form to the project, the size of a form is set in two different ways. First, you can set it is with the size grips in the designer. By dragging either the right edge, bottom edge, or the corner, you can resize the form.



The second way you can resize the form while the designer is open, is through the properties pane. Select the form, then find the **Properties** pane in Visual Studio. Scroll down to **size** and expand it. You can set the **Width** and **Height** manually.



Resize in code

Even though the designer sets the starting size of a form, you can resize it through code. Using code to resize a form is useful when something about your application determines that the default size of the form is insufficient.

To resize a form, change the Size, which represents the width and height of the form.

Resize the current form

You can change the size of the current form as long as the code is running within the context of the form. For example, if you have `Form1` with a button on it, that when clicked invokes the `Click` event handler to resize the form:

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Size = new Size(250, 200);
```

Resize a different form

You can change the size of another form after it's created by using the variable referencing the form. For example, let's say you have two forms, `Form1` (the startup form in this example) and `Form2`. `Form1` has a button that when clicked, invokes the `Click` event. The handler of this event creates a new instance of the `Form2` form, sets the size, and then displays it:

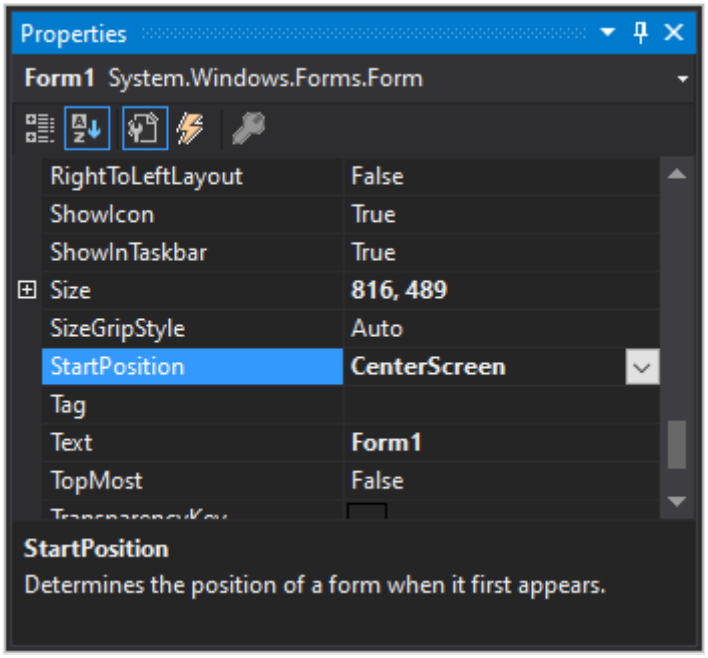
C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Size = new Size(250, 200);
    form.Show();
}
```

If the `Size` isn't manually set, the form's default size is what it was set to during design-time.

Position with the designer

When a form instance is created and displayed, the initial location of the form is determined by the StartPosition property. The Location property holds the current location the form. Both properties can be set through the designer.



Expand table

FormStartPosition Enum	Description
CenterParent	The form is centered within the bounds of its parent form.
CenterScreen	The form is centered on the current display.
Manual	The position of the form is determined by the Location property.
WindowsDefaultBounds	The form is positioned at the Windows default location and is resized to the default size determined by Windows.
WindowsDefaultLocation	The form is positioned at the Windows default location and isn't resized.

The CenterParent value only works with forms that are either a multiple document interface (MDI) child form, or a normal form that is displayed with the ShowDialog method. `CenterParent` has no affect on a normal form that is displayed with the Show method. To center a form (`form` variable) to another form (`parentForm` variable), use the following code:

C#

```
form.StartPosition = FormStartPosition.Manual;
form.Location = new Point(parentForm.Width / 2 - form.Width / 2 +
parentForm.Location.X,
                          parentForm.Height / 2 - form.Height / 2 +
parentForm.Location.Y);
form.Show();
```

Position with code

Even though the designer can be used to set the starting location of a form, you can use code either change the starting position mode or set the location manually. Using code to position a form is useful if you need to manually position and size a form in relation to the screen or other forms.

Move the current form

You can move the current form as long as the code is running within the context of the form. For example, if you have `Form1` with a button on it, that when clicked invokes the `Click` event handler. The handler in this example changes the location of the form to the top-left of the screen by setting the Location property:

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Location = new Point(0, 0);
```

Position a different form

You can change the location of another form after it's created by using the variable referencing the form. For example, let's say you have two forms, `Form1` (the startup form in this example) and `Form2`. `Form1` has a button that when clicked, invokes the `Click` event. The handler of this event creates a new instance of the `Form2` form and sets the location:

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Location = new Point(0, 0);
    form.Show();
}
```

If the `Location` isn't set, the form's default position is based on what the `StartPosition` property was set to at design-time.

See also

- How to add a form to a project (Windows Forms .NET)
- Events overview (Windows Forms .NET)
- Position and layout of controls (Windows Forms .NET)

How to position and size a form (Windows Forms .NET)

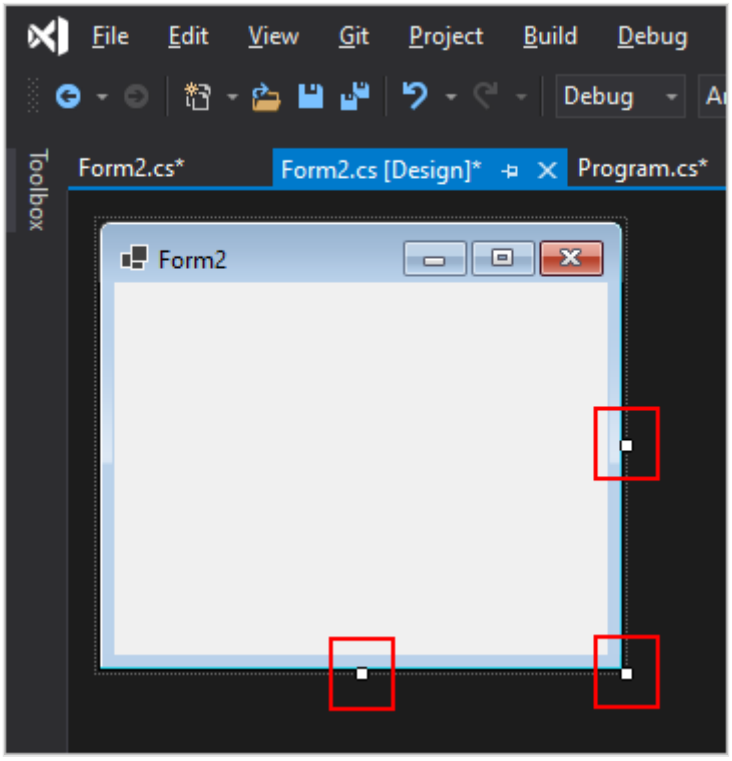
Article • 09/23/2022

When a form is created, the size and location is initially set to a default value. The default size of a form is generally a width and height of *800x500* pixels. The initial location, when the form is displayed, depends on a few different settings.

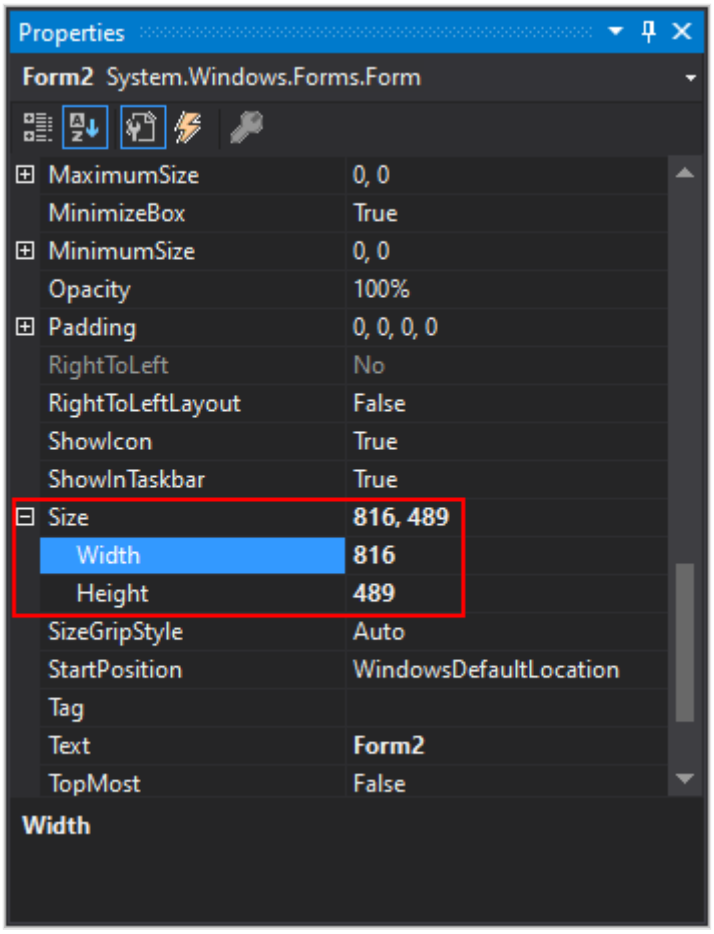
You can change the size of a form at design time with Visual Studio, and at run time with code.

Resize with the designer

After adding a new form to the project, the size of a form is set in two different ways. First, you can set it is with the size grips in the designer. By dragging either the right edge, bottom edge, or the corner, you can resize the form.



The second way you can resize the form while the designer is open, is through the properties pane. Select the form, then find the **Properties** pane in Visual Studio. Scroll down to **size** and expand it. You can set the **Width** and **Height** manually.



Resize in code

Even though the designer sets the starting size of a form, you can resize it through code. Using code to resize a form is useful when something about your application determines that the default size of the form is insufficient.

To resize a form, change the Size, which represents the width and height of the form.

Resize the current form

You can change the size of the current form as long as the code is running within the context of the form. For example, if you have `Form1` with a button on it, that when clicked invokes the `Click` event handler to resize the form:

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Size = new Size(250, 200);
```

Resize a different form

You can change the size of another form after it's created by using the variable referencing the form. For example, let's say you have two forms, `Form1` (the startup form in this example) and `Form2`. `Form1` has a button that when clicked, invokes the `Click` event. The handler of this event creates a new instance of the `Form2` form, sets the size, and then displays it:

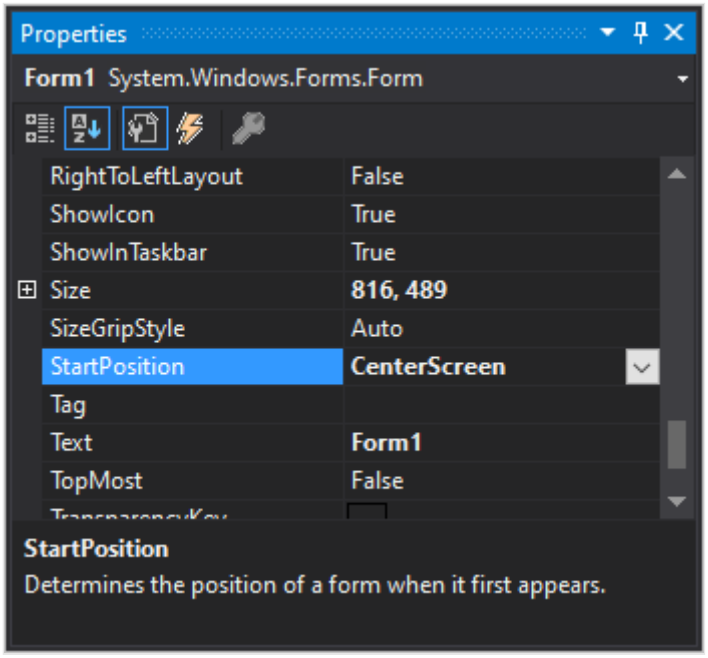
C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Size = new Size(250, 200);
    form.Show();
}
```

If the `Size` isn't manually set, the form's default size is what it was set to during design-time.

Position with the designer

When a form instance is created and displayed, the initial location of the form is determined by the StartPosition property. The Location property holds the current location the form. Both properties can be set through the designer.



Expand table

FormStartPosition Enum	Description
CenterParent	The form is centered within the bounds of its parent form.
CenterScreen	The form is centered on the current display.
Manual	The position of the form is determined by the Location property.
WindowsDefaultBounds	The form is positioned at the Windows default location and is resized to the default size determined by Windows.
WindowsDefaultLocation	The form is positioned at the Windows default location and isn't resized.

The CenterParent value only works with forms that are either a multiple document interface (MDI) child form, or a normal form that is displayed with the ShowDialog method. `CenterParent` has no affect on a normal form that is displayed with the Show method. To center a form (`form` variable) to another form (`parentForm` variable), use the following code:

C#

```
form.StartPosition = FormStartPosition.Manual;
form.Location = new Point(parentForm.Width / 2 - form.Width / 2 +
parentForm.Location.X,
                          parentForm.Height / 2 - form.Height / 2 +
parentForm.Location.Y);
form.Show();
```

Position with code

Even though the designer can be used to set the starting location of a form, you can use code either change the starting position mode or set the location manually. Using code to position a form is useful if you need to manually position and size a form in relation to the screen or other forms.

Move the current form

You can move the current form as long as the code is running within the context of the form. For example, if you have `Form1` with a button on it, that when clicked invokes the `Click` event handler. The handler in this example changes the location of the form to the top-left of the screen by setting the Location property:

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Location = new Point(0, 0);
```

Position a different form

You can change the location of another form after it's created by using the variable referencing the form. For example, let's say you have two forms, `Form1` (the startup form in this example) and `Form2`. `Form1` has a button that when clicked, invokes the `Click` event. The handler of this event creates a new instance of the `Form2` form and sets the location:

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Location = new Point(0, 0);
    form.Show();
}
```

If the `Location` isn't set, the form's default position is based on what the `StartPosition` property was set to at design-time.

See also

- How to add a form to a project (Windows Forms .NET)
- Events overview (Windows Forms .NET)
- Position and layout of controls (Windows Forms .NET)

Overview of using controls (Windows Forms .NET)

Article • 08/14/2023

Windows Forms controls are reusable components that encapsulate user interface functionality and are used in client-side, Windows-based applications. Not only does Windows Forms provide many ready-to-use controls, it also provides the infrastructure for developing your own controls. You can combine existing controls, extend existing controls, or author your own custom controls. For more information, see Types of custom controls.

Adding controls

Controls are added through the Visual Studio Designer. With the Designer, you can place, size, align, and move controls. Alternatively, controls can be added through code. For more information, see Add a control (Windows Forms).

Layout options

The position a control appears on a parent is determined by the value of the Location property relative to the top-left of the parent surface. The top-left position coordinate in the parent is `(x0,y0)`. The size of the control is determined by the Size property and represents the width and height of the control.

Besides manual positioning and sizing, various container controls are provided that help with automatic placement of controls.

For more information, see Position and layout of controls and How to dock and anchor controls.

Control events

Controls provides more than 60 events through the base class Control. These include the Paint event, which causes a control to be drawn, events related to displaying a window, such as the Resize and Layout events, and low-level mouse and keyboard events. Some low-level events are synthesized by Control into semantic events such as Click and DoubleClick. Most shared events fall under these categories:

- Mouse events

- Keyboard events
- Property changed events
- Other events

Not every control responds to every event. For example, the Label control doesn't respond to keyboard input, and the Control.PreviewKeyDown event isn't raised.

Often, a control is a wrapper for an underlaying Win32 control, and using the Paint event to draw on top of the control may be limited or do nothing at all, since the control is ultimately drawn by Windows.

For more information, see Control events and How to handle a control event.

Control accessibility

Windows Forms has accessibility support for screen readers and voice input utilities for verbal commands. However, you must design your UI with accessibility in mind. Windows Forms controls expose various properties to handle accessibility. For more information about these properties, see Providing Accessibility Information for Controls.

See also

- Position and layout of controls
- Label control overview
- Control events
- Types of custom controls
- Painting and drawing on controls
- Providing Accessibility Information for Controls

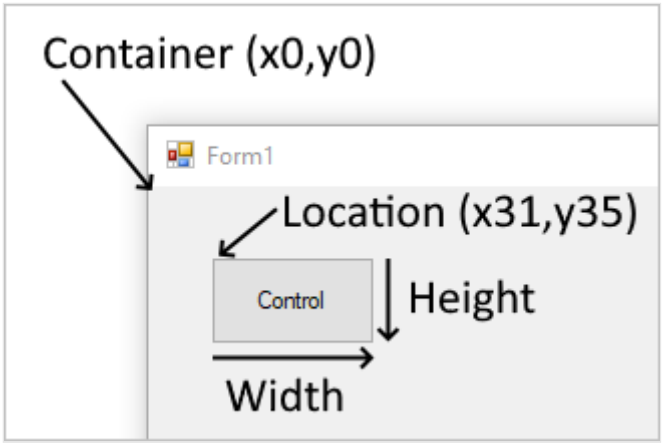
Position and layout of controls (Windows Forms .NET)

Article • 07/30/2021

Control placement in Windows Forms is determined not only by the control, but also by the parent of the control. This article describes the different settings provided by controls and the different types of parent containers that affect layout.

Fixed position and size

The position a control appears on a parent is determined by the value of the Location property relative to the top-left of the parent surface. The top-left position coordinate in the parent is `(x0,y0)`. The size of the control is determined by the Size property and represents the width and height of the control.



When a control is added to a parent that enforces automatic placement, the position and size of the control is changed. In this case, the position and size of the control may not be manually adjusted, depending on the type of parent.

The MaximumSize and MinimumSize properties help set the minimum and maximum space a control can use.

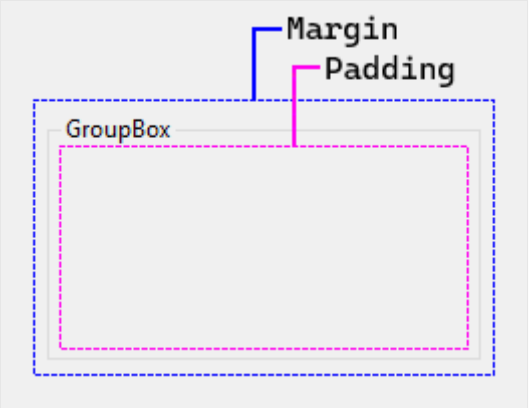
Margin and Padding

There are two control properties that help with precise placement of controls: Margin and Padding.

The Margin property defines the space around the control that keeps other controls a specified distance from the control's borders.

The Padding property defines the space in the interior of a control that keeps the control's content (for example, the value of its Text property) a specified distance from the control's borders.

The following figure shows the Margin and Padding properties on a control.



The Visual Studio Designer will respect these properties when you're positioning and resizing controls. Snaplines appear as guides to help you remain outside the specified margin of a control. For example, Visual Studio displays the snapline when you drag a control next to another control:

Automatic placement and size

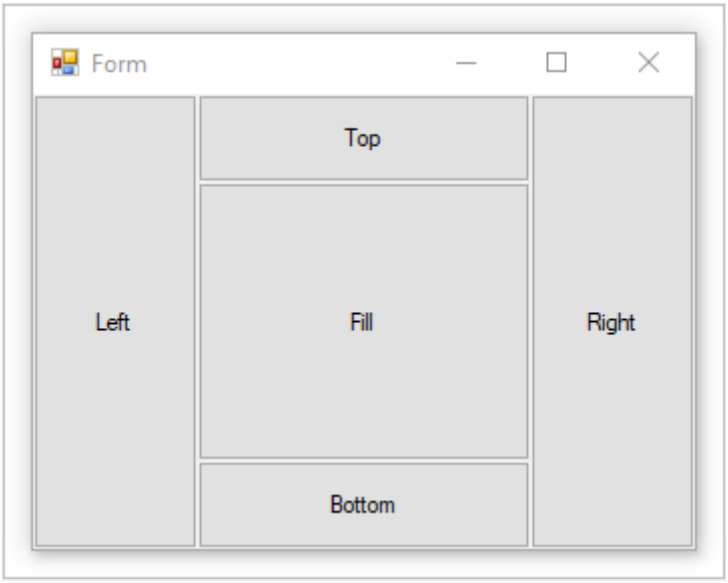
Controls can be automatically placed within their parent. Some parent containers force placement while others respect control settings that guide placement. There are two properties on a control that help automatic placement and size within a parent: Dock and Anchor.

Drawing order can affect automatic placement. The order in which a control is drawn determined by the control's index in the parent's Controls collection. This index is known as the **Z-order**. Each control is drawn in the reverse order they appear in the collection. Meaning, the collection is a first-in-last-drawn and last-in-first-drawn collection.

The MinimumSize and MaximumSize properties help set the minimum and maximum space a control can use.

Dock

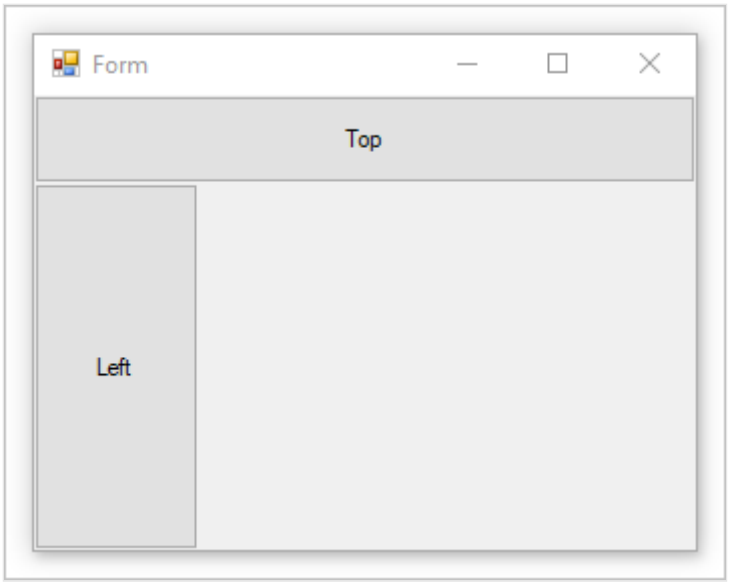
The `Dock` property sets which border of the control is aligned to the corresponding side of the parent, and how the control is resized within the parent.



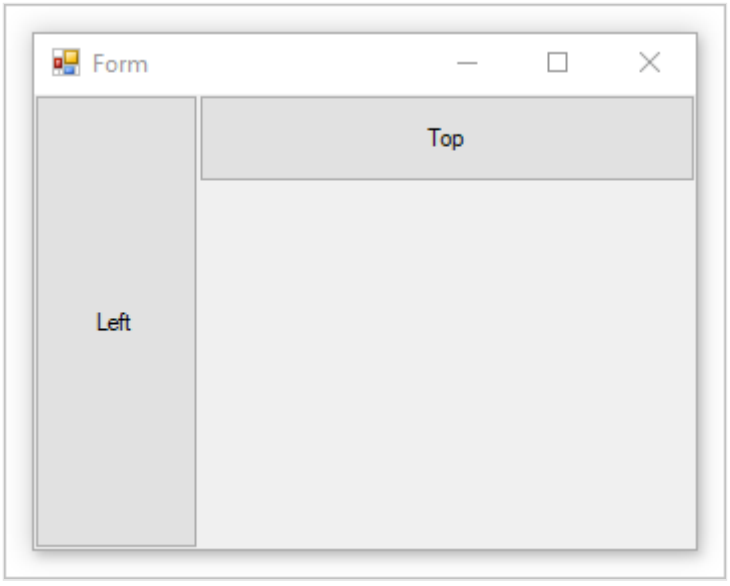
When a control is docked, the container determines the space it should occupy and resizes and places the control. The width and height of the control are still respected based on the docking style. For example, if the control is docked to the top, the Height of the control is respected but the Width is automatically adjusted. If a control is docked to the left, the Width of the control is respected but the Height is automatically adjusted.

The Location of the control can't be manually set as docking a control automatically controls its position.

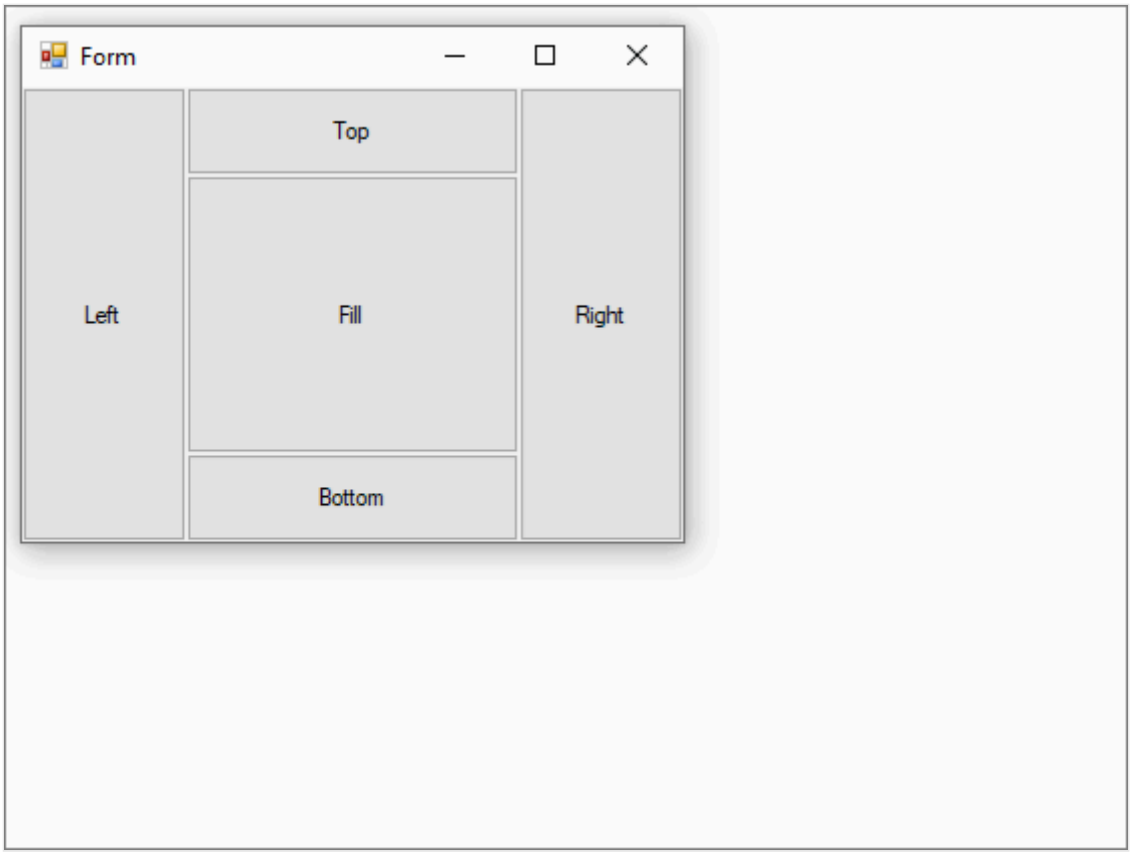
The **Z-order** of the control does affect docking. As docked controls are laid out, they use what space is available to them. For example, if a control is drawn first and docked to the top, it will take up the entire width of the container. If a next control is docked to the left, it has less vertical space available to it.



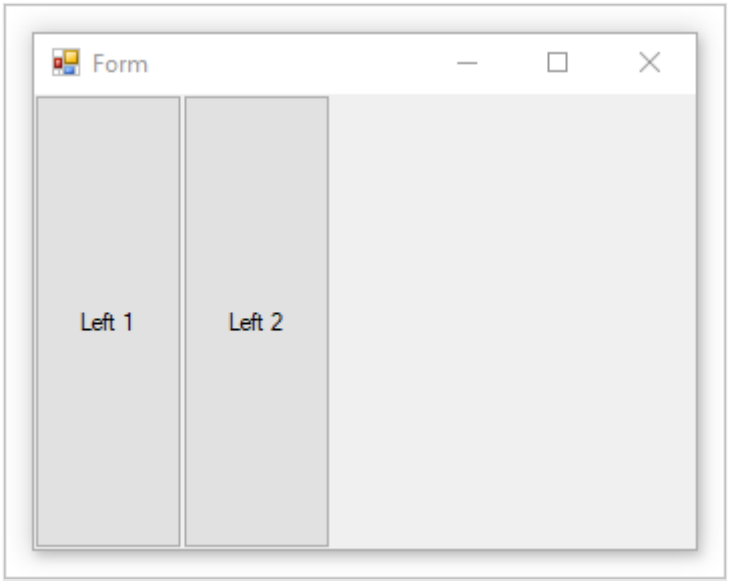
If the control's **Z-order** is reversed, the control that is docked to the left now has more initial space available. The control uses the entire height of the container. The control that is docked to the top has less horizontal space available to it.



As the container grows and shrinks, the controls docked to the container are repositioned and resized to maintain their applicable positions and sizes.



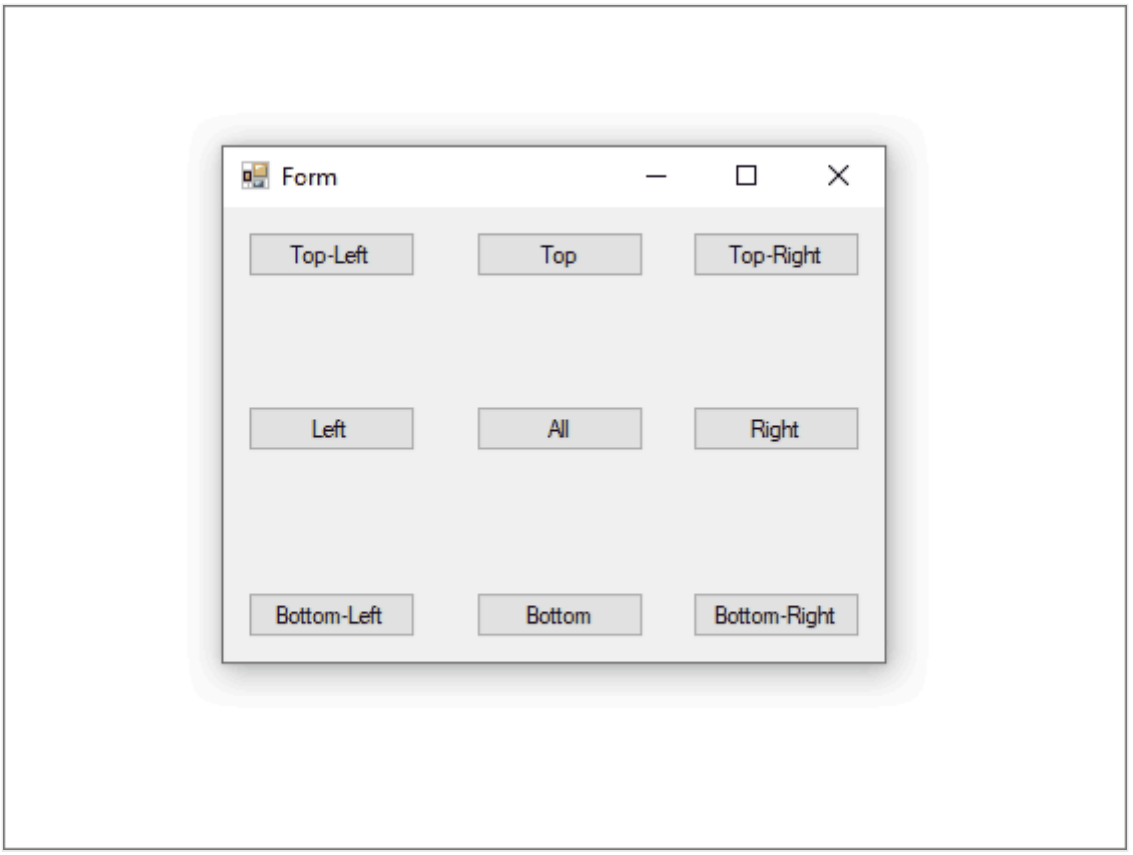
If multiple controls are docked to the same side of the container, they're stacked according to their **Z-order**.



Anchor

Anchoring a control allows you to tie the control to one or more sides of the parent container. As the container changes in size, any child control will maintain its distance to the anchored side.

A control can be anchored to one or more sides, without restriction. The anchor is set with the Anchor property.



Automatic sizing

The AutoSize property enables a control to change its size, if necessary, to fit the size specified by the PreferredSize property. You adjust the sizing behavior for specific controls by setting the `AutoSizeMode` property.

Only some controls support the AutoSize property. In addition, some controls that support the AutoSize property also supports the `AutoSizeMode` property.

Expand table

Always true behavior	Description
Automatic sizing is a run-time feature.	This means it never grows or shrinks a control and then has no further effect.
If a control changes size, the value of its Location property always remains constant.	When a control's contents cause it to grow, the control grows toward the right and downward. Controls do not grow to the left.
The Dock and Anchor properties are honored when AutoSize is `true`.	The value of the control's Location property is adjusted to the correct value. The Label control is the exception to this rule. When you set the value of a docked Label control's

Always true behavior	Description
	AutoSize property to `true`, the Label control will not stretch.
A control's MaximumSize and MinimumSize properties are always honored, regardless of the value of its AutoSize property.	The MaximumSize and MinimumSize properties are not affected by the AutoSize property.
There is no minimum size set by default.	This means that if a control is set to shrink under AutoSize and it has no contents, the value of its Size property is `(0x,0y)`. In this case, your control will shrink to a point, and it will not be readily visible.
If a control does not implement the GetPreferredSize method, the GetPreferredSize method returns last value assigned to the Size property.	This means that setting AutoSize to `true` will have no effect.
A control in a TableLayoutPanel cell always shrinks to fit in the cell until its MinimumSize is reached.	This size is enforced as a maximum size. This is not the case when the cell is part of an AutoSize row or column.

AutoSizeMode property

The AutoSizeMode property provides more fine-grained control over the default AutoSize behavior. The `AutoSizeMode` property specifies how a control sizes itself to its content. The content, for example, could be the text for a Button control or the child controls for a container.

The following list shows the `AutoSizeMode` values and its behavior.

- AutoSizeMode.GrowAndShrink

  The control grows or shrinks to encompass its contents.

  The MinimumSize and MaximumSize values are honored, but the current value of the Size property is ignored.

  This is the same behavior as controls with the AutoSize property and no `AutoSizeMode` property.

- AutoSizeMode.GrowOnly

  The control grows as much as necessary to encompass its contents, but it will not shrink smaller than the value specified by its Size property.

This is the default value for `AutoSizeMode`.

Controls that support the AutoSize property

The following table describes the level of auto sizing support by control:

Expand table

Control	`AutoSize` supported	`AutoSizeMode` supported
Button	✔	✔
CheckedListBox	✔	✔
FlowLayoutPanel	✔	✔
Form	✔	✔
GroupBox	✔	✔
Panel	✔	✔
TableLayoutPanel	✔	✔
CheckBox	✔	✘
DomainUpDown	✔	✘
Label	✔	✘
LinkLabel	✔	✘
MaskedTextBox	✔	✘
NumericUpDown	✔	✘
RadioButton	✔	✘
TextBox	✔	✘
TrackBar	✔	✘
CheckedListBox	✘	✘
ComboBox	✘	✘
DataGridView	✘	✘
DateTimePicker	✘	✘
ListBox	✘	✘

Control	`AutoSize` supported	`AutoSizeMode` supported
ListView	✗	✗
MaskedTextBox	✗	✗
MonthCalendar	✗	✗
ProgressBar	✗	✗
PropertyGrid	✗	✗
RichTextBox	✗	✗
SplitContainer	✗	✗
TabControl	✗	✗
TabPage	✗	✗
TreeView	✗	✗
WebBrowser	✗	✗
ScrollBar	✗	✗

AutoSize in the design environment

The following table describes the sizing behavior of a control at design time, based on the value of its AutoSize and `AutoSizeMode` properties.

Override the SelectionRules property to determine whether a given control is in a user-resizable state. In the following table, "can't resize" means Moveable only, "can resize" means AllSizeable and Moveable.

Expand table

`AutoSize` setting	`AutoSizeMode` setting	Behavior
`true`	Property not available.	The user can't resize the control at design time, except for the following controls: - TextBox - MaskedTextBox - RichTextBox - TrackBar
`true`	GrowAndShrink	The user can't resize the control at design time.

`AutoSize` setting	`AutoSizeMode` setting	Behavior
`true`	GrowOnly	The user can resize the control at design time. When the Size property is set, the user can only increase the size of the control.
`false` or `AutoSize` is hidden	Not applicable.	User can resize the control at design time.

> **Note**
>
> To maximize productivity, the Windows Forms Designer in Visual Studio shadows the **AutoSize** property for the **Form** class. At design time, the form behaves as though the **AutoSize** property is set to `false`, regardless of its actual setting. At runtime, no special accommodation is made, and the **AutoSize** property is applied as specified by the property setting.

Container: Form

The Form is the main object of Windows Forms. A Windows Forms application will usually have a form displayed at all times. Forms contain controls and respect the Location and Size properties of the control for manual placement. Forms also respond to the Dock property for automatic placement.

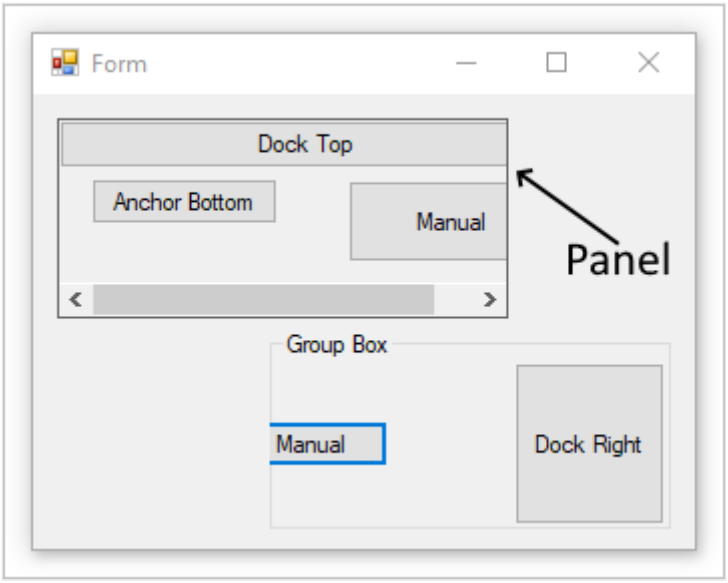
Most of the time a form will have grips on the edges that allow the user to resize the form. The Anchor property of a control will let the control grow and shrink as the form is resized.

Container: Panel

The Panel control is similar to a form in that it simply groups controls together. It supports the same manual and automatic placement styles that a form does. For more information, see the Container: Form section.

A panel blends in seamlessly with the parent, and it does cut off any area of a control that falls out of bounds of the panel. If a control falls outside the bounds of the panel and AutoScroll is set to `true`, scroll bars appear and the user can scroll the panel.

Unlike the group box control, a panel doesn't have a caption and border.



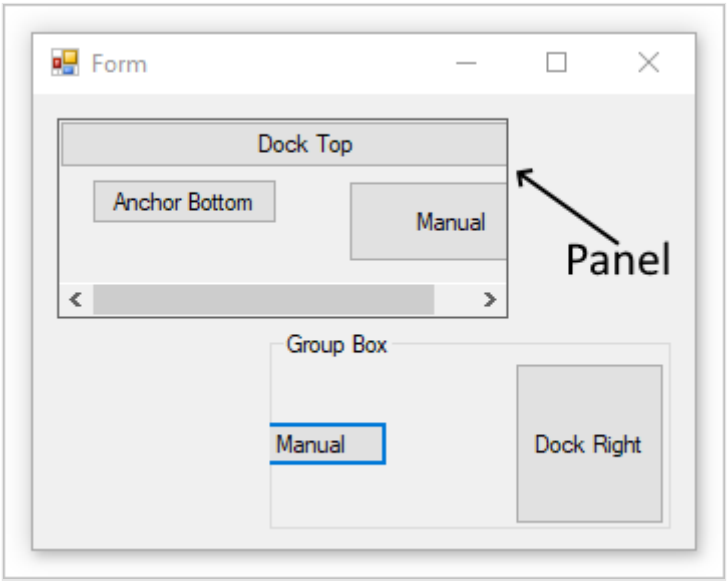
The image above has a panel with the BorderStyle property set to demonstrate the bounds of the panel.

Container: Group box

The GroupBox control provides an identifiable grouping for other controls. Typically, you use a group box to subdivide a form by function. For example, you may have a form representing personal information and the fields related to an address would be grouped together. At design time, it's easy to move the group box around along with its contained controls.

The group box supports the same manual and automatic placement styles that a form does. For more information, see the Container: Form section. A group box also cuts off any portion of a control that falls out of bounds of the panel.

Unlike the panel control, a group box doesn't have the capability to scroll content and display scroll bars.

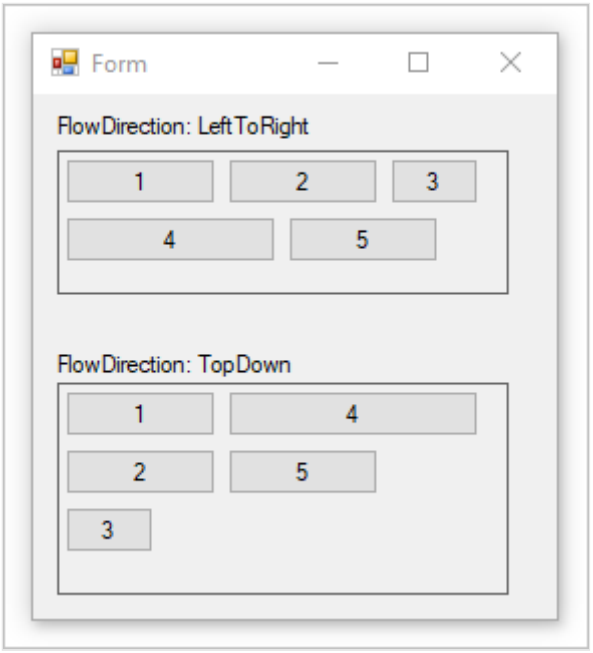


Container: Flow Layout

The FlowLayoutPanel control arranges its contents in a horizontal or vertical flow direction. You can wrap the control's contents from one row to the next, or from one column to the next. Alternately, you can clip instead of wrap its contents.

You can specify the flow direction by setting the value of the FlowDirection property. The FlowLayoutPanel control correctly reverses its flow direction in Right-to-Left (RTL) layouts. You can also specify whether the FlowLayoutPanel control's contents are wrapped or clipped by setting the value of the WrapContents property.

The FlowLayoutPanel control automatically sizes to its contents when you set the AutoSize property to `true`. It also provides a `FlowBreak` property to its child controls. Setting the value of the `FlowBreak` property to `true` causes the FlowLayoutPanel control to stop laying out controls in the current flow direction and wrap to the next row or column.



The image above has two `FlowLayoutPanel` controls with the BorderStyle property set to demonstrate the bounds of the control.

Container: Table layout

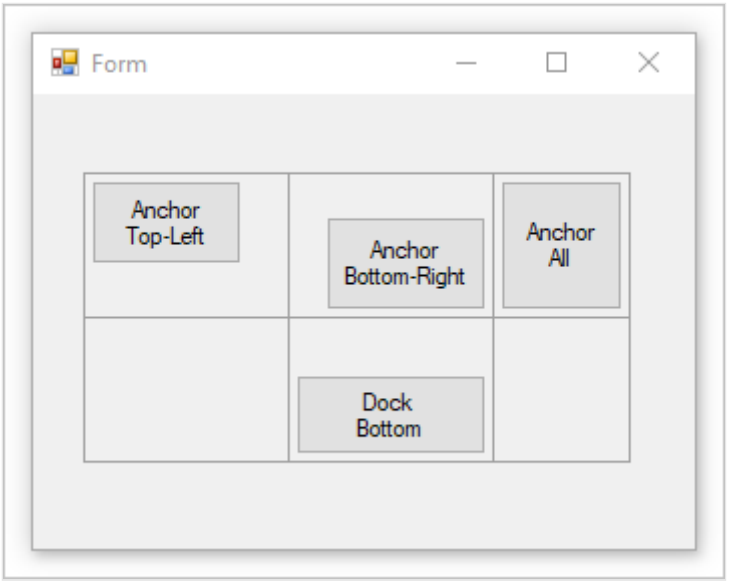
The TableLayoutPanel control arranges its contents in a grid. Because the layout is done both at design time and run time, it can change dynamically as the application environment changes. This gives the controls in the panel the ability to resize proportionally, so they can respond to changes such as the parent control resizing or text length changing because of localization.

Any Windows Forms control can be a child of the TableLayoutPanel control, including other instances of TableLayoutPanel. This allows you to construct sophisticated layouts that adapt to changes at run time.

You can also control the direction of expansion (horizontal or vertical) after the TableLayoutPanel control is full of child controls. By default, the TableLayoutPanel control expands downward by adding rows.

You can control the size and style of the rows and columns by using the RowStyles and ColumnStyles properties. You can set the properties of rows or columns individually.

The TableLayoutPanel control adds the following properties to its child controls: `Cell`, `Column`, `Row`, `ColumnSpan`, and `RowSpan`.

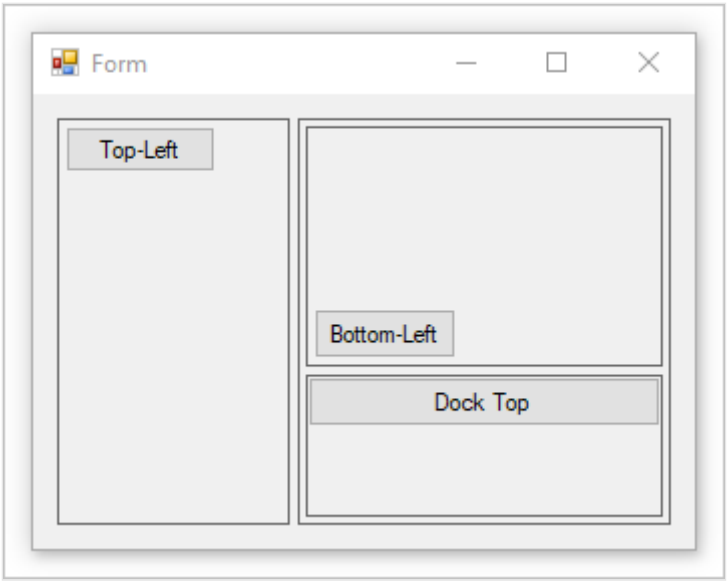


The image above has a table with the CellBorderStyle property set to demonstrate the bounds of each cell.

Container: Split container

The Windows Forms SplitContainer control can be thought of as a composite control; it's two panels separated by a movable bar. When the mouse pointer is over the bar, the pointer changes shape to show that the bar is movable.

With the SplitContainer control, you can create complex user interfaces; often, a selection in one panel determines what objects are shown in the other panel. This arrangement is effective for displaying and browsing information. Having two panels lets you aggregate information in areas, and the bar, or "splitter," makes it easy for users to resize the panels.

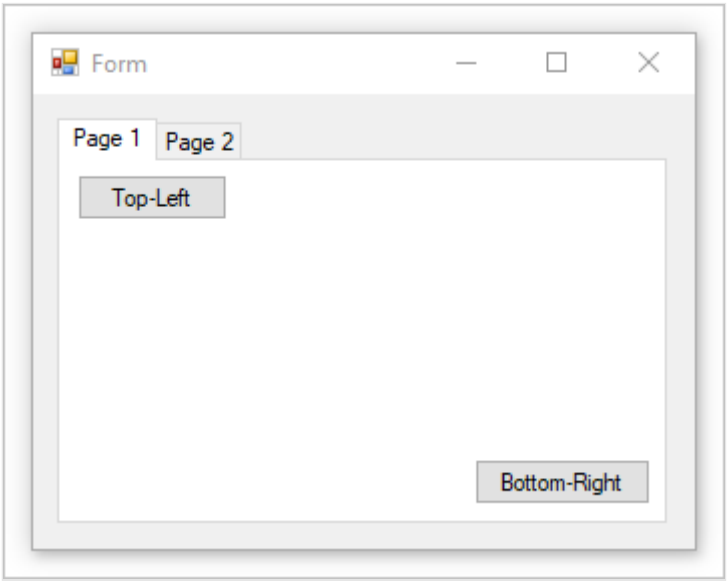


The image above has a split container to create a left and right pane. The right pane contains a second split container with the Orientation set to Vertical. The BorderStyle property is set to demonstrate the bounds of each panel.

Container: Tab control

The TabControl displays multiple tabs, like dividers in a notebook or labels in a set of folders in a filing cabinet. The tabs can contain pictures and other controls. Use the tab control to produce the kind of multiple-page dialog box that appears many places in the Windows operating system, such as the Control Panel and Display Properties. Additionally, the TabControl can be used to create property pages, which are used to set a group of related properties.

The most important property of the TabControl is TabPages, which contains the individual tabs. Each individual tab is a TabPage object.



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Label control overview (Windows Forms .NET)

Article • 04/19/2024

Windows Forms Label controls are used to display text that cannot be edited by the user. They're used to identify objects on a form and to provide a description of what a certain control represents or does. For example, you can use labels to add descriptive captions to text boxes, list boxes, combo boxes, and so on. You can also write code that changes the text displayed by a label in response to events at run time.

Working with the Label Control

Because the Label control can't receive focus, it can be used to create access keys for other controls. An access key allows a user to focus the next control in tab order by pressing the `Alt` key with the chosen access key. For more information, see Use a label to focus a control.

The caption displayed in the label is contained in the Text property. The TextAlign property allows you to set the alignment of the text within the label. For more information, see How to: Set the Text Displayed by a Windows Forms Control.

See also

- Use a label to focus a control (Windows Forms .NET)
- How to: Set the text displayed by a control (Windows Forms .NET)
- AutoScaleMode
- Scale
- PerformAutoScale
- AutoScaleDimensions

more information, see our contributor guide.

Provide product feedback

Control events (Windows Forms .NET)

Article • 08/14/2023

Controls provide events that are raised when the user interacts with the control or when the state of the control changes. This article describes the common events shared by most controls, events raised by user interaction, and events unique to specific controls. For more information about events in Windows Forms, see Events overview and Handling and raising events.

For more information about how to add or remove a control event handler, see How to handle an event.

Common events

Controls provides more than 60 events through the base class Control. These include the Paint event, which causes a control to be drawn, events related to displaying a window, such as the Resize and Layout events, and low-level mouse and keyboard events. Some low-level events are synthesized by Control into semantic events such as Click and DoubleClick. Most shared events fall under these categories:

- Mouse events
- Keyboard events
- Property changed events
- Other events

Mouse events

Considering Windows Forms is a User Interface (UI) technology, mouse input is the primary way users interact with a Windows Forms application. All controls provide basic mouse-related events:

- MouseClick
- MouseDoubleClick
- MouseDown
- MouseEnter
- MouseHover
- MouseLeave
- MouseMove
- MouseUp
- MouseWheel

- Click

For more information, see Using mouse events.

Keyboard events

If the control responds to user input, such as a TextBox or Button control, the appropriate input event is raised for the control. The control must be focused to receive keyboard events. Some controls, such as the Label control, can't be focused and can't receive keyboard events. The following is a list of keyboard events:

- KeyDown
- KeyPress
- KeyUp

For more information, see Using keyboard events.

Property changed events

Windows Forms follows the *PropertyNameChanged* pattern for properties that have change events. The data binding engine provided by Windows Forms recognizes this pattern and integrates well with it. When creating your own controls, implement this pattern.

This pattern implements the following rules, using the property `FirstName` as an example:

- Name your property: `FirstName`.
- Create an event for the property using the pattern `PropertyNameChanged`: `FirstNameChanged`.
- Create a private or protected method using the pattern `OnPropertyNameChanged`: `OnFirstNameChanged`.

If the `FirstName` property set modifies the backing value, the `OnFirstNameChanged` method is called. The `OnFirstNameChanged` method raises the `FirstNameChanged` event.

Here are some of the common property changed events for a control:

Expand table

Event	Description
BackColorChanged	Occurs when the value of the BackColor property changes.
BackgroundImageChanged	Occurs when the value of the BackgroundImage property changes.
BindingContextChanged	Occurs when the value of the BindingContext property changes.
DockChanged	Occurs when the value of the Dock property changes.
EnabledChanged	Occurs when the Enabled property value has changed.
FontChanged	Occurs when the Font property value changes.
ForeColorChanged	Occurs when the ForeColor property value changes.
LocationChanged	Occurs when the Location property value has changed.
SizeChanged	Occurs when the Size property value changes.
VisibleChanged	Occurs when the Visible property value changes.

For a full list of events, see the **Events** section of the Control Class.

Other events

Controls will also raise events based on the state of the control, or other interactions with the control. For example, the HelpRequested event is raised if the control has focus and the user presses the F1 key. This event is also raised if the user presses the context-sensitive **Help** button on a form, and then presses the help cursor on the control.

Another example is when a control is changed, moved, or resized, the Paint event is raised. This event provides the developer with the opportunity to draw on the control and change its appearance.

For a full list of events, see the **Events** section of the Control Class.

See also

- How to handle an event
- Events overview
- Using mouse events
- Using keyboard events
- System.Windows.Forms.Control
- System.Windows.Forms.Control.Click
- System.Windows.Forms.Button

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Custom controls (Windows Forms .NET)

Article • 08/04/2023

With Windows Forms, you can create new controls or modify existing controls through inheritance. This article highlights the differences among the ways of creating new controls, and provides you with information about how to choose a particular type of control for your project.

Base control class

The Control class is the base class for Windows Forms controls. It provides the infrastructure required for visual display in Windows Forms applications and provides the following capabilities:

- Exposes a window handle.
- Manages message routing.
- Provides mouse and keyboard events, and many other user interface events.
- Provides advanced layout features.
- Contains many properties specific to visual display, such as ForeColor, BackColor, Height, and Width.

Because so much of the infrastructure is provided by the base class, it's relatively easy to develop your own Windows Forms controls.

Create your own control

There are three types of custom controls you can create: user controls, extended controls, and custom controls. The following table helps you decide which type of control you should create:

Expand table

If ...	Create a ...
• You want to combine the functionality of several Windows Forms controls into a single reusable unit.	Design a user control by inheriting from System.Windows.Forms.UserControl.
• Most of the functionality you need is already identical to an existing Windows Forms control.	Extend a control by inheriting from a specific Windows Forms control.

If ...	Create a ...
• You don't need a custom graphical user interface, or you want to design a new graphical user interface for an existing control.	
• You want to provide a custom graphical representation of your control. • You need to implement custom functionality that isn't available through standard controls.	Create a custom control by inheriting from System.Windows.Forms.Control.

User controls

A user control is a collection of Windows Forms controls presented as a single control to the consumer. This kind of control is referred to as a *composite control*. The contained controls are called *constituent controls*.

A user control holds all of the inherent functionality associated with each of the contained Windows Forms controls and enables you to selectively expose and bind their properties. A user control also provides a great deal of default keyboard handling functionality with no extra development effort on your part.

For example, a user control could be built to display customer address data from a database. This control would include a DataGridView control to display the database fields, a BindingSource to handle binding to a data source, and a BindingNavigator control to move through the records. You could selectively expose data binding properties, and you could package and reuse the entire control from application to application.

For more information, see User control overview.

Extended controls

You can derive an inherited control from any existing Windows Forms control. With this approach, you can keep all of the inherent functionality of a Windows Forms control, and then extend that functionality by adding custom properties, methods, or other features. With this option, you can override the base control's paint logic, and then extend its user interface by changing its appearance.

For example, you can create a control derived from the Button control that tracks how many times a user has clicked it.

In some controls, you can also add a custom appearance to the graphical user interface of your control by overriding the OnPaint method of the base class. For an extended button that tracks clicks, you can override the OnPaint method to call the base implementation of OnPaint, and then draw the click count in one corner of the Button control's client area.

Custom controls

Another way to create a control is to create one substantially from the beginning by inheriting from Control. The Control class provides all of the basic functionality required by controls, including mouse and keyboard handling events, but no control-specific functionality or graphical interface.

Creating a control by inheriting from the Control class requires more thought and effort than inheriting from UserControl or an existing Windows Forms control. Because a great deal of implementation is left for you, your control can have greater flexibility than a composite or extended control, and you can tailor your control to suit your exact needs.

To implement a custom control, you must write code for the OnPaint event of the control, which controls how the control is visually drawn. You must also write any feature-specific behaviors for your control. You can also override the WndProc method and handle windows messages directly. This is the most powerful way to create a control, but to use this technique effectively, you need to be familiar with the Microsoft Win32® API.

An example of a custom control is a clock control that duplicates the appearance and behavior of an analog clock. Custom painting is invoked to cause the hands of the clock to move in response to Tick events from an internal Timer component.

Custom design experience

If you need to implement a custom design-time experience, you can author your own designer. For composite controls, derive your custom designer class from the ParentControlDesigner or the DocumentDesigner classes. For extended and custom controls, derive your custom designer class from the ControlDesigner class.

Use the DesignerAttribute to associate your control with your designer.

The following information is out of date but may help you.

- (Visual Studio 2013) Extending Design-Time Support.
- (Visual Studio 2013) How to: Create a Windows Forms Control That Takes Advantage of Design-Time Features.

Painting and drawing on controls (Windows Forms .NET)

Article • 06/02/2023

Custom painting of controls is one of the many complicated tasks made easy by Windows Forms. When authoring a custom control, you have many options available to handle your control's graphical appearance. If you're authoring a custom control, that is, a control that inherits from Control, you must provide code to render its graphical representation.

If you're creating a composite control, that is a control that inherits from UserControl or one of the existing Windows Forms controls, you may override the standard graphical representation and provide your own graphics code.

If you want to provide custom rendering for an existing control without creating a new control, your options become more limited. However, there are still a wide range of graphical possibilities for your controls and applications.

The following elements are involved in control rendering:

- The drawing functionality provided by the base class System.Windows.Forms.Control.
- The essential elements of the GDI graphics library.
- The geometry of the drawing region.
- The procedure for freeing graphics resources.

Drawing provided by control

The base class Control provides drawing functionality through its Paint event. A control raises the Paint event whenever it needs to update its display. For more information about events in the .NET, see Handling and raising events.

The event data class for the Paint event, PaintEventArgs, holds the data needed for drawing a control - a handle to a graphics object and a rectangle that represents the region to draw in.

C#

```
public class PaintEventArgs : EventArgs, IDisposable
{

    public System.Drawing.Rectangle ClipRectangle {get;}
```

```
    public System.Drawing.Graphics Graphics {get;}

    // Other properties and methods.
}
```

Graphics is a managed class that encapsulates drawing functionality, as described in the discussion of GDI later in this article. The ClipRectangle is an instance of the Rectangle structure and defines the available area in which a control can draw. A control developer can compute the ClipRectangle using the ClipRectangle property of a control, as described in the discussion of geometry later in this article.

OnPaint

A control must provide rendering logic by overriding the OnPaint method that it inherits from Control. OnPaint gets access to a graphics object and a rectangle to draw in through the Graphics and the ClipRectangle properties of the PaintEventArgs instance passed to it.

The following code uses the `System.Drawing` namespace:

C#

```
protected override void OnPaint(PaintEventArgs e)
{
    // Call the OnPaint method of the base class.
    base.OnPaint(e);

    // Declare and instantiate a new pen that will be disposed of at the end
of the method.
    using var myPen = new Pen(Color.Aqua);

    // Create a rectangle that represents the size of the control, minus 1
pixel.
    var area = new Rectangle(new Point(0, 0), new Size(this.Size.Width - 1,
this.Size.Height - 1));

    // Draw an aqua rectangle in the rectangle represented by the control.
    e.Graphics.DrawRectangle(myPen, area);
}
```

The OnPaint method of the base Control class doesn't implement any drawing functionality but merely invokes the event delegates that are registered with the Paint event. When you override OnPaint, you should typically invoke the OnPaint method of the base class so that registered delegates receive the Paint event. However, controls that paint their entire surface shouldn't invoke the base class's OnPaint, as this introduces flicker.

> **Note**
>
> Don't invoke OnPaint directly from your control; instead, invoke the Invalidate method (inherited from Control) or some other method that invokes Invalidate. The Invalidate method in turn invokes OnPaint. The Invalidate method is overloaded, and, depending on the arguments supplied to Invalidate `e`, redraws either some or all of its screen area.

The code in the OnPaint method of your control will execute when the control is first drawn, and whenever it is refreshed. To ensure that your control is redrawn every time it is resized, add the following line to the constructor of your control:

```
C#

SetStyle(ControlStyles.ResizeRedraw, true);
```

OnPaintBackground

The base Control class defines another method that is useful for drawing, the OnPaintBackground method.

```
C#

protected virtual void OnPaintBackground(PaintEventArgs e);
```

OnPaintBackground paints the background (and in that way, the shape) of the window and is guaranteed to be fast, while OnPaint paints the details and might be slower because individual paint requests are combined into one Paint event that covers all areas that have to be redrawn. You might want to invoke the OnPaintBackground if, for instance, you want to draw a gradient-colored background for your control.

While OnPaintBackground has an event-like nomenclature and takes the same argument as the `OnPaint` method, `OnPaintBackground` is not a true event method. There is no `PaintBackground` event and `OnPaintBackground` doesn't invoke event delegates. When overriding the `OnPaintBackground` method, a derived class is not required to invoke the `OnPaintBackground` method of its base class.

GDI+ Basics

The Graphics class provides methods for drawing various shapes such as circles, triangles, arcs, and ellipses, and methods for displaying text. The System.Drawing namespace contains namespaces and classes that encapsulate graphics elements such as shapes (circles, rectangles, arcs, and others), colors, fonts, brushes, and so on.

Geometry of the Drawing Region

The ClientRectangle property of a control specifies the rectangular region available to the control on the user's screen, while the ClipRectangle property of PaintEventArgs specifies the area that is painted. A control might need to paint only a portion of its available area, as is the case when a small section of the control's display changes. In those situations, a control developer must compute the actual rectangle to draw in and pass that to Invalidate. The overloaded versions of Invalidate that take a Rectangle or Region as an argument use that argument to generate the ClipRectangle property of PaintEventArgs.

Freeing Graphics Resources

Graphics objects are expensive because they use system resources. Such objects include instances of the System.Drawing.Graphics class and instances of System.Drawing.Brush, System.Drawing.Pen, and other graphics classes. It's important that you create a graphics resource only when you need it and release it soon as you're finished using it. If you create an instance of a type that implements the IDisposable interface, call its Dispose method when you're finished with it to free resources.

See also

- Types of custom controls

Providing Accessibility Information for Controls (Windows Forms .NET)

Article • 10/28/2020

Accessibility aids are specialized programs and devices that help people with disabilities use computers more effectively. Examples include screen readers for people who are blind and voice input utilities for people who provide verbal commands instead of using the mouse or keyboard. These accessibility aids interact with the accessibility properties exposed by Windows Forms controls. These properties are:

- System.Windows.Forms.AccessibleObject
- System.Windows.Forms.Control.AccessibleDefaultActionDescription
- System.Windows.Forms.Control.AccessibleDescription
- System.Windows.Forms.Control.AccessibleName
- System.Windows.Forms.AccessibleRole

AccessibilityObject Property

This read-only property contains an AccessibleObject instance. The `AccessibleObject` implements the IAccessible interface, which provides information about the control's description, screen location, navigational abilities, and value. The designer sets this value when the control is added to the form.

AccessibleDefaultActionDescription Property

This string describes the action of the control. It does not appear in the Properties window and may only be set in code. The following example sets the AccessibleDefaultActionDescription property for a button control:

C#

```
button1.AccessibleDefaultActionDescription = "Closes the application.";
```

AccessibleDescription Property

This string describes the control. The AccessibleDescription property may be set in the Properties window, or in code as follows:

```
C#

button1.AccessibleDescription = "A button with text 'Exit'";
```

AccessibleName Property

This is the name of a control reported to accessibility aids. The AccessibleName property may be set in the Properties window, or in code as follows:

```
C#

button1.AccessibleName = "Order";
```

AccessibleRole Property

This property, which contains an AccessibleRole enumeration, describes the user interface role of the control. A new control has the value set to `Default`. This would mean that by default, a `Button` control acts as a `Button`. You may want to reset this property if a control has another role. For example, you may be using a `PictureBox` control as a `Chart`, and you may want accessibility aids to report the role as a `Chart`, not as a `PictureBox`. You may also want to specify this property for custom controls you have developed. This property may be set in the Properties window, or in code as follows:

```
C#

pictureBox1.AccessibleRole = AccessibleRole.Chart;
```

See also

- Label control overview (Windows Forms .NET)
- AccessibleObject
- Control.AccessibilityObject
- Control.AccessibleDefaultActionDescription
- Control.AccessibleDescription
- Control.AccessibleName
- Control.AccessibleRole
- AccessibleRole

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Add a control to a form (Windows Forms .NET)

Article • 04/19/2024

Most forms are designed by adding controls to the surface of the form to define a user interface (UI). A *control* is a component on a form used to display information or accept user input.

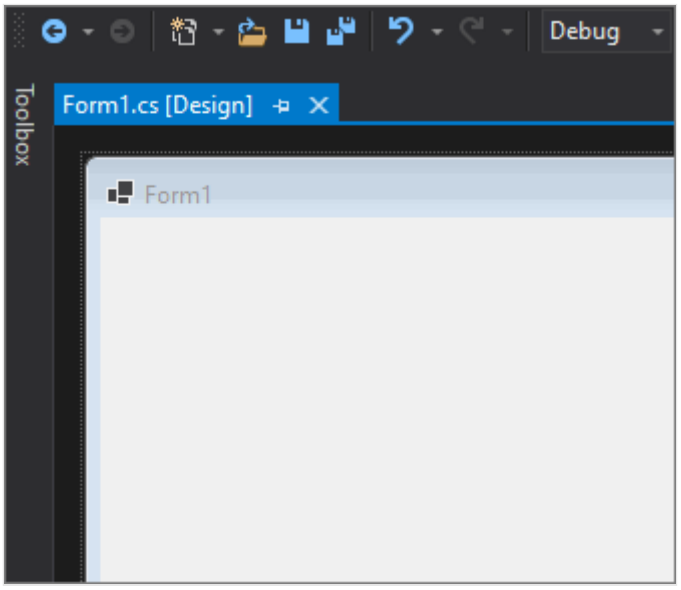
The primary way a control is added to a form is through the Visual Studio Designer, but you can also manage the controls on a form at run time through code.

Add with Designer

Visual Studio uses the Forms Designer to design forms. There is a Controls pane which lists all the controls available to your app. You can add controls from the pane in two ways:

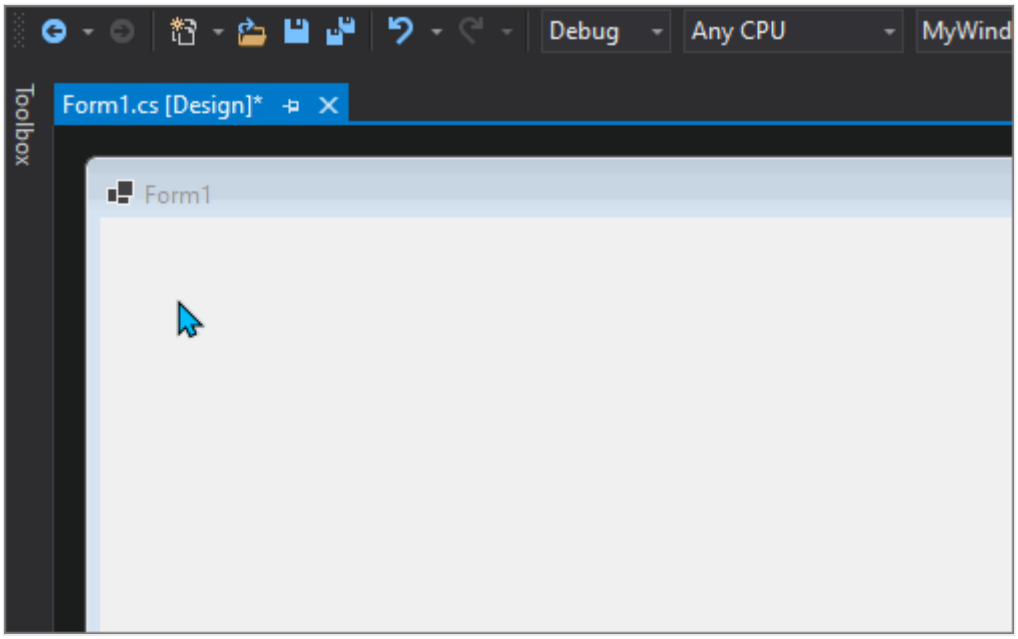
Add the control by double-clicking

When a control is double-clicked, it is automatically added to the current open form with default settings.



Add the control by drawing

Select the control by clicking on it. In your form, drag-select a region. The control will be placed to fit the size of the region you selected.



Add with code

Controls can be created and then added to a form at run time with the form's Controls collection. This collection can also be used to remove controls from a form.

The following code adds and positions two controls, a Label and a TextBox:

C#

```
Label label1 = new Label()
{
    Text = "&First Name",
    Location = new Point(10, 10),
    TabIndex = 10
};

TextBox field1 = new TextBox()
{
    Location = new Point(label1.Location.X, label1.Bounds.Bottom +
Padding.Top),
    TabIndex = 11
};

Controls.Add(label1);
Controls.Add(field1);
```

See also

- Set the Text Displayed by a Windows Forms Control
- Add an access key shortcut to a control
- System.Windows.Forms.Label

- System.Windows.Forms.TextBox
- System.Windows.Forms.Button

Add an access key shortcut to a control (Windows Forms .NET)

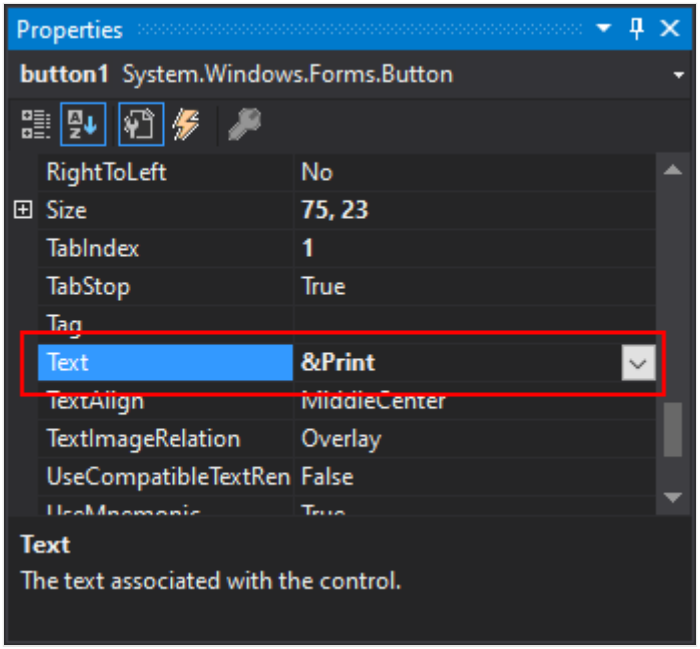
Article • 04/19/2024

An *access key* is an underlined character in the text of a menu, menu item, or the label of a control such as a button. With an access key, the user can "click" a button by pressing the `Alt` key in combination with the predefined access key. For example, if a button runs a procedure to print a form, and therefore its `Text` property is set to "Print," adding an ampersand (&) before the letter "P" causes the letter "P" to be underlined in the button text at run time. The user can run the command associated with the button by pressing `Alt`.

Controls that cannot receive focus can't have access keys, except label controls.

Designer

In the **Properties** window of Visual Studio, set the **Text** property to a string that includes an ampersand (&) before the letter that will be the access key. For example, to set the letter "P" as the access key, enter **&Print**.



Programmatic

Set the `Text` property to a string that includes an ampersand (&) before the letter that will be the shortcut.

C#

```
// Set the letter "P" as an access key.
button1.Text = "&Print";
```

Use a label to focus a control

Even though a label cannot be focused, it has the ability to focus the next control in the tab order of the form. Each control is assigned a value to the TabIndex property, generally in ascending sequential order. When the access key is assigned to the Label.Text property, the next control in the sequential tab order is focused.

Using the example from the Programmatic section, if the button didn't have any text set, but instead presented an image of a printer, you could use a label to focus the button.

C#

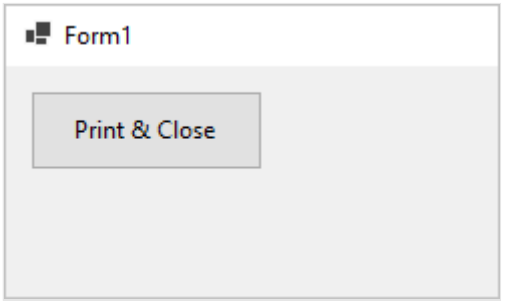
```
// Set the letter "P" as an access key.
label1.Text = "&Print";
label1.TabIndex = 9
button1.TabIndex = 10
```

Display an ampersand

When setting the text or caption of a control that interprets an ampersand (&) as an access key, use two consecutive ampersands (&&) to display a single ampersand. For example, the text of a button set to `"Print && Close"` displays in the caption of `Print & Close`:

C#

```
// Set the letter "P" as an access key.
button1.Text = "Print && Close";
```



See also

- Set the text displayed by a Windows Forms control
- System.Windows.Forms.Button
- System.Windows.Forms.Label

How to: Set the text displayed by a control (Windows Forms .NET)

Article • 04/19/2024

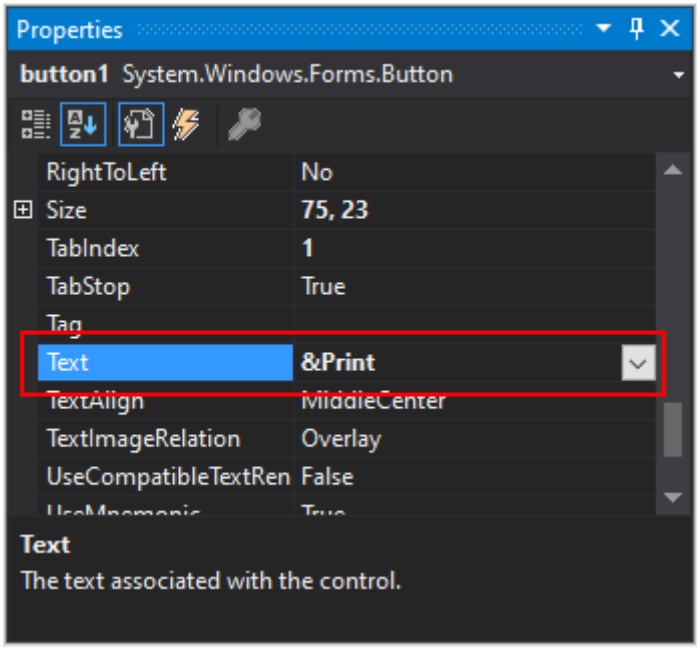
Windows Forms controls usually display some text that's related to the primary function of the control. For example, a Button control usually displays a caption indicating what action will be performed if the button is clicked. For all controls, you can set or return the text by using the Text property. You can change the font by using the Font property.

You can also set the text by using the designer.

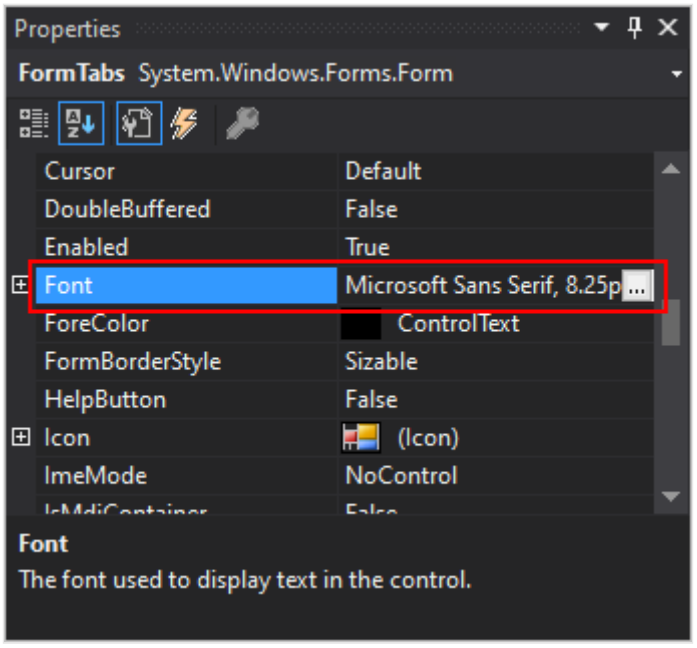
Designer

1. In the **Properties** window in Visual Studio, set the **Text** property of the control to an appropriate string.

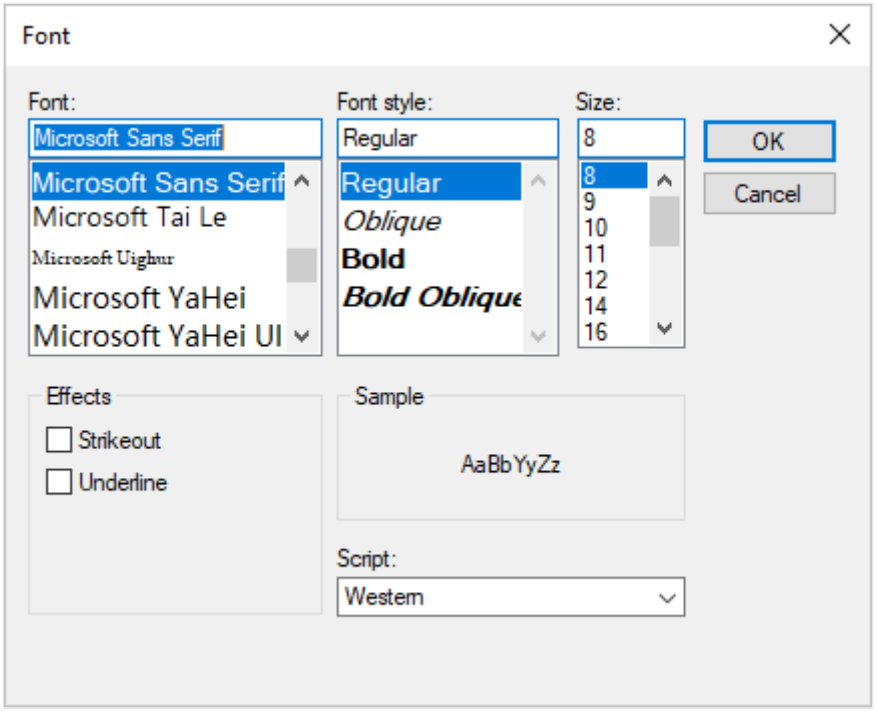
   To create an underlined shortcut key, include an ampersand (&) before the letter that will be the shortcut key.



2. In the **Properties** window, select the ellipsis button (...) next to the **Font** property.



In the standard font dialog box, adjust the font with settings such as type, size, and style.



Programmatic

1. Set the Text property to a string.

   To create an underlined access key, include an ampersand (&) before the letter that will be the access key.

2. Set the Font property to an object of type Font.

```
C#
```

```
button1.Text = "Click here to save changes";
button1.Font = new Font("Arial", 10, FontStyle.Bold,
GraphicsUnit.Point);
```

> **Note**
>
> You can use an escape character to display a special character in user-interface elements that would normally interpret them differently, such as menu items. For example, the following line of code sets the menu item's text to read "& Now For Something Completely Different":

C#

```
mpMenuItem.Text = "&& Now For Something Completely Different";
```

See also

- Create Access Keys for Windows Forms Controls
- System.Windows.Forms.Control.Text

How to set the tab order on Windows Forms (Windows Forms .NET)

Article • 04/19/2024

The tab order is the order in which a user moves focus from one control to another by pressing the `Tab` key. Each form has its own tab order. By default, the tab order is the same as the order in which you created the controls. Tab-order numbering begins with zero and ascends in value, and is set with the TabIndex property.

You can also set the tab order by using the designer.

Tab order can be set in the **Properties** window of the designer using the TabIndex property. The `TabIndex` property of a control determines where it's positioned in the tab order. By default, the first control added to the designer has a `TabIndex` value of 0, the second has a `TabIndex` of 1, and so on. Once the highest `TabIndex` has been focused, pressing `Tab` will cycle and focus the control with the lowest `TabIndex` value.

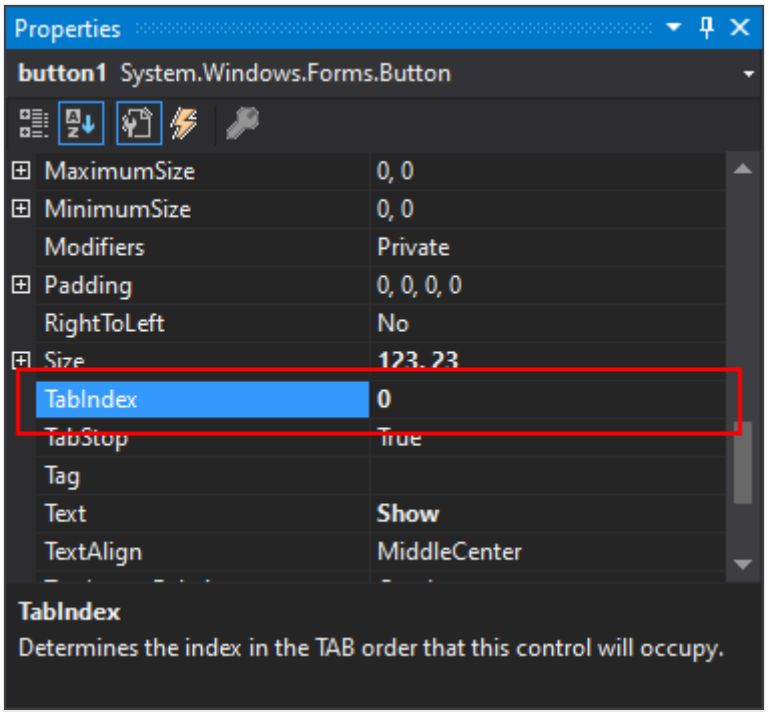
Container controls, such as a GroupBox control, treat their children as separate from the rest of the form. Each child in the container has its own TabIndex value. Because a container control can't be focused, when the tab order reaches the container control, the child control of the container with the lowest `TabIndex` is focused. As the `Tab` is pressed, each child control is focused according to its `TabIndex` value until the last control. When `Tab` is pressed on the last control, focus resumes to the next control in the parent of the container, based on the next `TabIndex` value.

Any control of the many on your form can be skipped in the tab order. Usually, pressing `Tab` successively at run time selects each control in the tab order. By turning off the TabStop property, a control is passed over in the tab order of the form.

Designer

Use the Visual Studio designer **Properties** window to set the tab order of a control.

1. Select the control in the designer.
2. In the **Properties** window in Visual Studio, set the **TabIndex** property of the control to an appropriate number.



Programmatic

1. Set the `TabIndex` property to a numerical value.

```
C#

Button1.TabIndex = 1;
```

Remove a control from the tab order

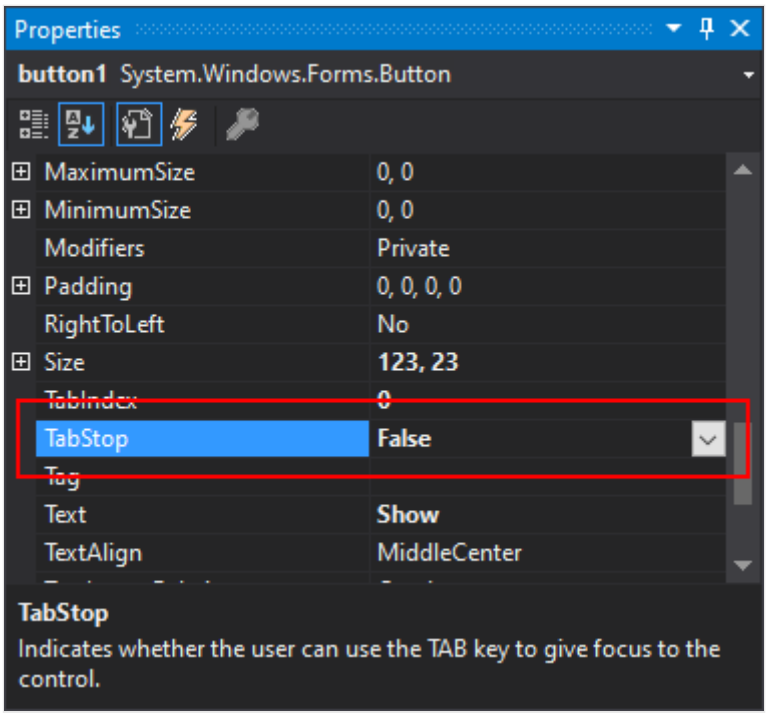
You can prevent a control from receiving focus when the `Tab` key is pressed, by setting the TabStop property to `false`. The control is skipped when you cycle through the controls with the `Tab` key. The control doesn't lose its tab order when this property is set to `false`.

> ⓘ Note
>
> A radio button group has a single tab stop at run-time. The selected button, the button with its Checked property set to `true`, has its TabStop property automatically set to `true`. Other buttons in the radio button group have their `TabStop` property set to `false`.

Set TabStop with the designer

1. Select the control in the designer.

2. In the **Properties** window in Visual Studio, set the **TabStop** property to `False`.



Set TabStop programmatically

1. Set the `TabStop` property to `false`.

```
C#

Button1.TabStop = false;
```

See also

- Add a control to a form
- System.Windows.Forms.Control.TabIndex
- System.Windows.Forms.Control.TabStop

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

more information, see our contributor guide.

Open a documentation issue

Provide product feedback

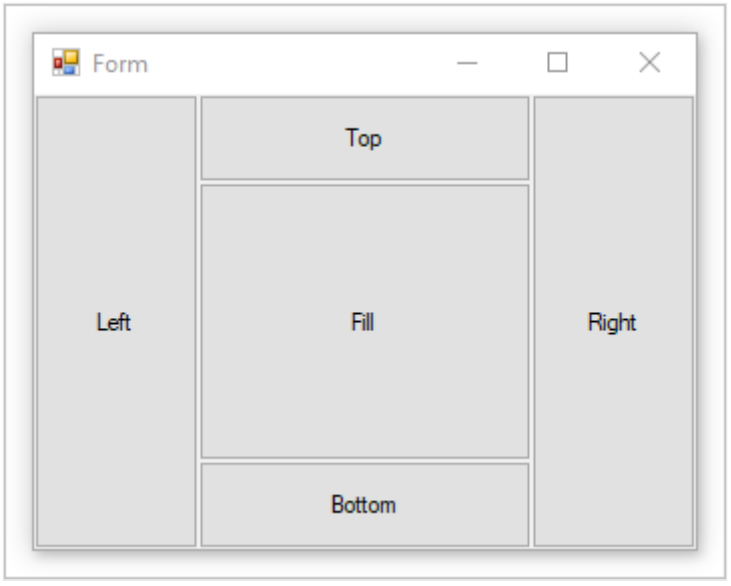
How to dock and anchor controls (Windows Forms .NET)

Article • 06/02/2021

If you're designing a form that the user can resize at run time, the controls on your form should resize and reposition properly. Controls have two properties that help with automatic placement and sizing, when the form changes size.

- Control.Dock

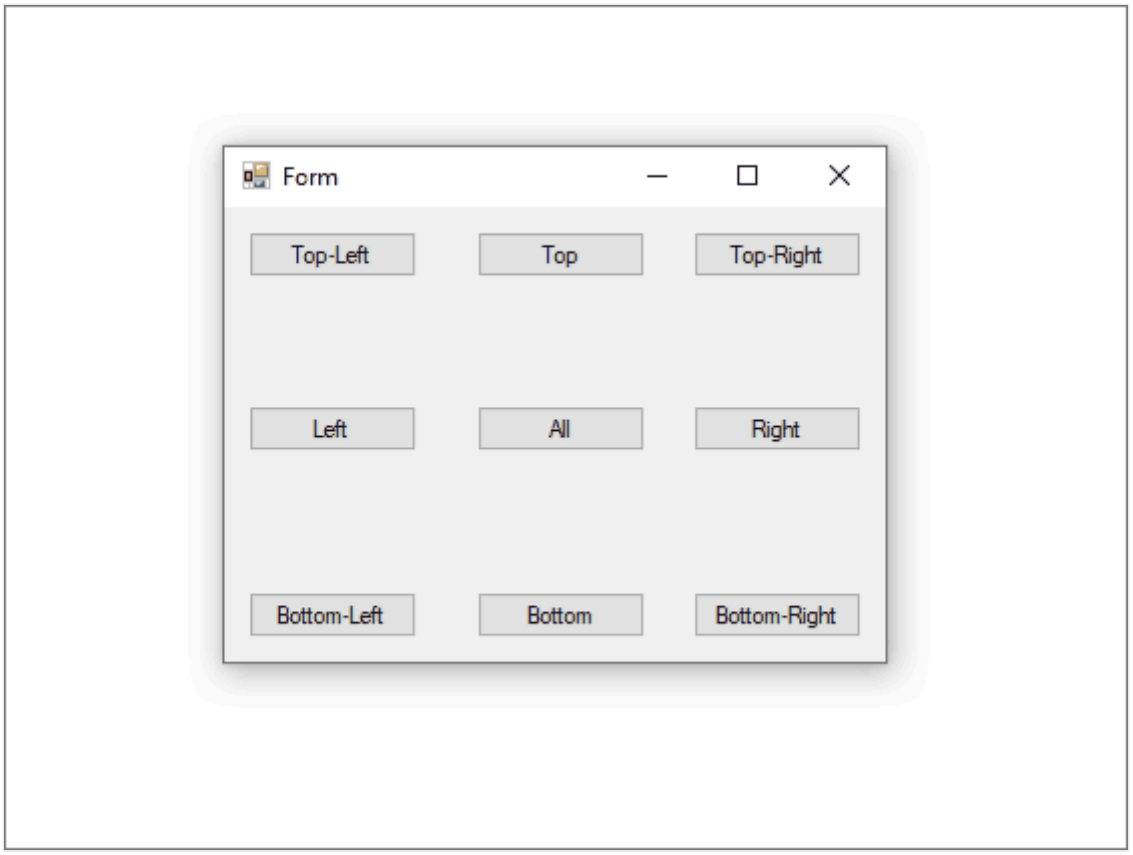
  Controls that are docked fill the edges of the control's container, either the form or a container control. For example, Windows Explorer docks its TreeView control to the left side of the window and its ListView control to the right side of the window. The docking mode can be any side of the control's container, or set to fill the remaining space of the container.

  Controls are docked in reverse z-order and the Dock property interacts with the AutoSize property. For more information, see Automatic sizing.

- Control.Anchor

  When an anchored control's form is resized, the control maintains the distance between the control and the anchor positions. For example, if you have a TextBox control that is anchored to the left, right, and bottom edges of the form, as the form is resized, the TextBox control resizes horizontally so that it maintains the same distance from the right and left sides of the form. The control also positions itself vertically so that its location is always the same distance from the bottom edge of the form. If a control isn't anchored and the form is resized, the position of the control relative to the edges of the form is changed.



For more information, see Position and layout of controls.

Dock a control

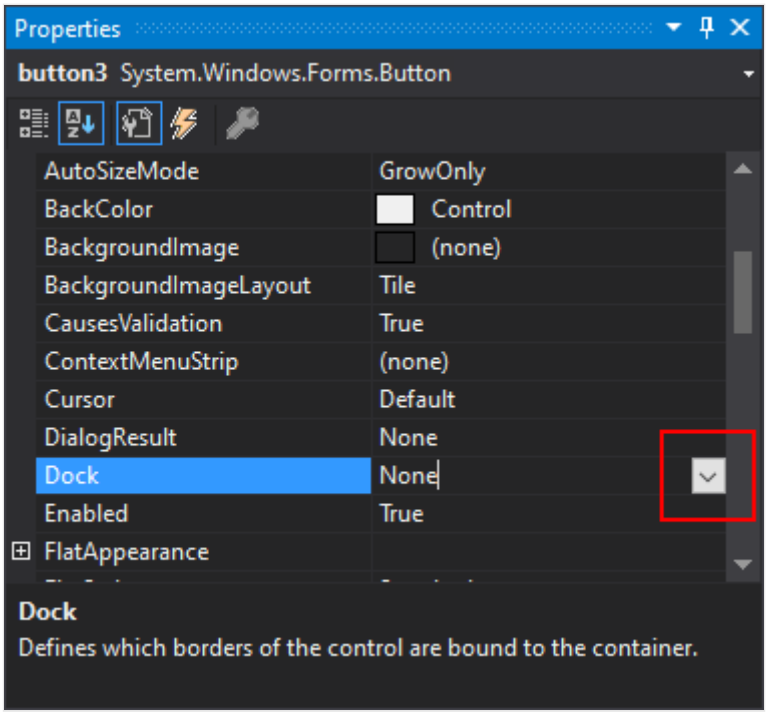
A control is docked by setting its Dock property.

> **Note**
>
> Inherited controls must be `Protected` to be able to be docked. To change the access level of a control, set its **Modifier** property in the **Properties** window.

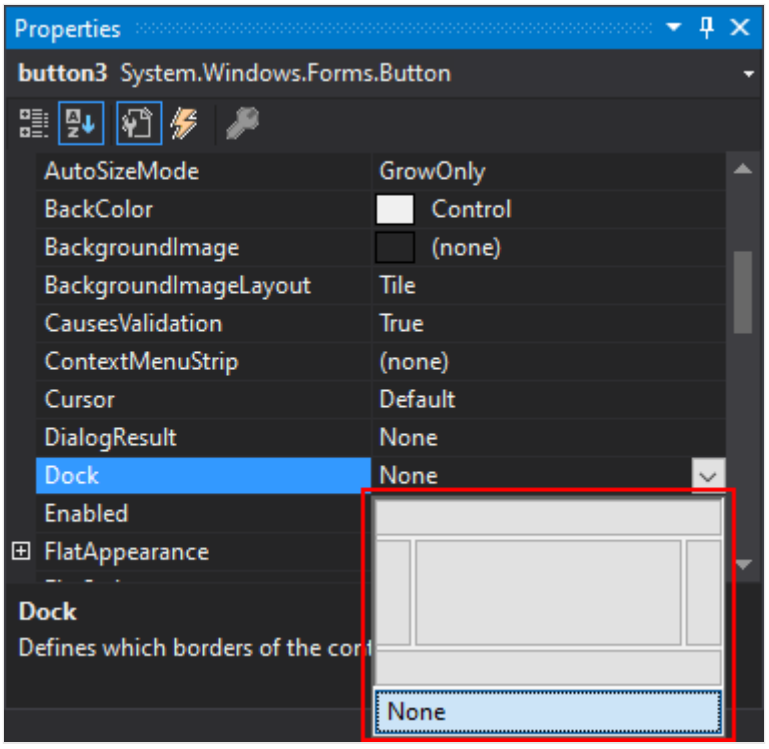
Use the designer

Use the Visual Studio designer **Properties** window to set the docking mode of a control.

1. Select the control in the designer.
2. In the **Properties** window, select the arrow to the right of the **Dock** property.



3. Select the button that represents the edge of the container where you want to dock the control. To fill the contents of the control's form or container control, press the center box. Press **(none)** to disable docking.



The control is automatically resized to fit the boundaries of the docked edge.

Set Dock programmatically

1. Set the `Dock` property on a control. In this example, a button is docked to the right side of its container:

```
C#

button1.Dock = DockStyle.Right;
```

Anchor a control

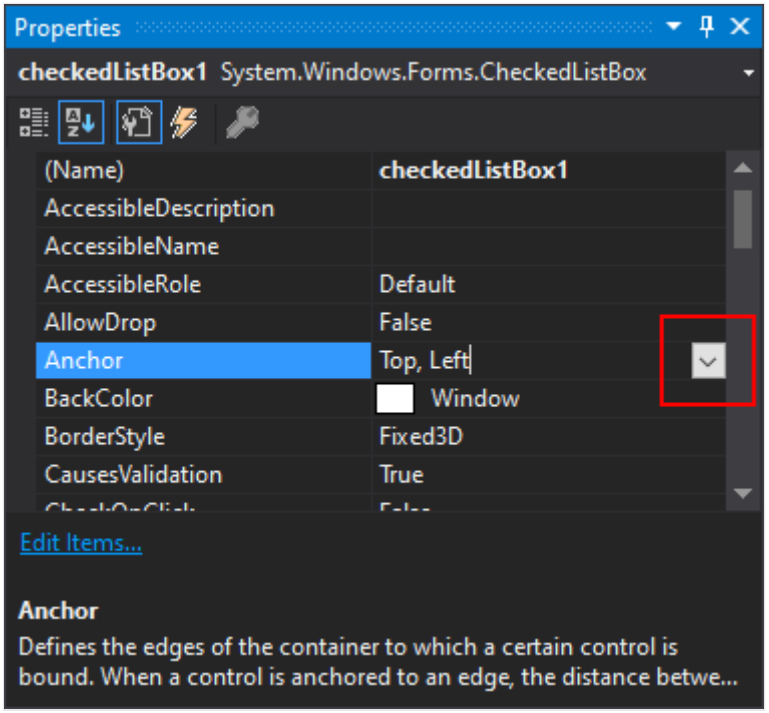
A control is anchored to an edge by setting its Anchor property to one or more values.

> **Note**
>
> Certain controls, such as the ComboBox control, have a limit to their height. Anchoring the control to the bottom of its form or container cannot force the control to exceed its height limit.
>
> Inherited controls must be `Protected` to be able to be anchored. To change the access level of a control, set its `Modifiers` property in the **Properties** window.

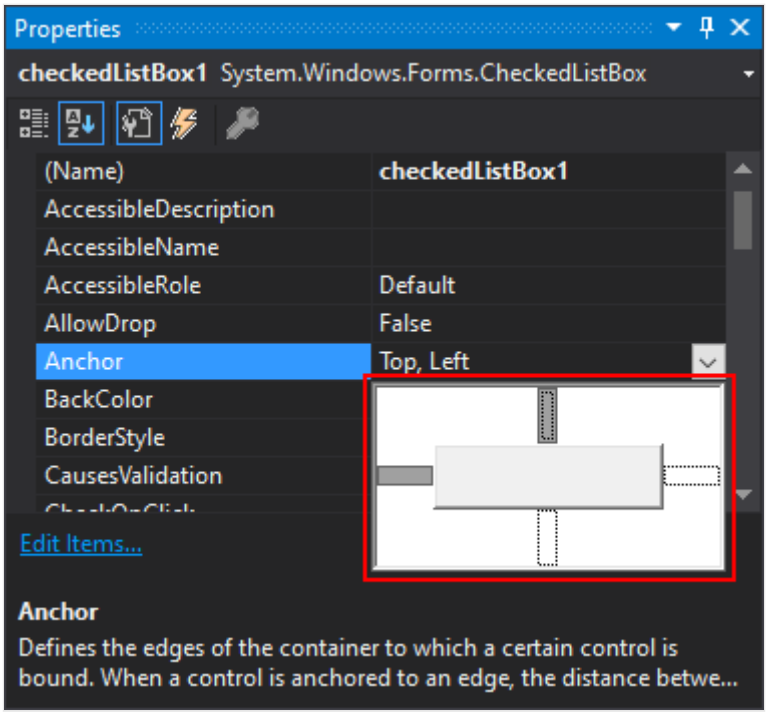
Use the designer

Use the Visual Studio designer **Properties** window to set the anchored edges of a control.

1. Select the control in the designer.
2. In the **Properties** window, select the arrow to the right of the **Anchor** property.



3. To set or unset an anchor, select the top, left, right, or bottom arm of the cross.



Set Anchor programmatically

1. Set the `Anchor` property on a control. In this example, a button is anchored to the right and bottom sides of its container:

```
C#
```

```
button1.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
```

See also

- Position and layout of controls.
- System.Windows.Forms.Control.Anchor
- System.Windows.Forms.Control.Dock

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

How to display an image on a control (Windows Forms .NET)

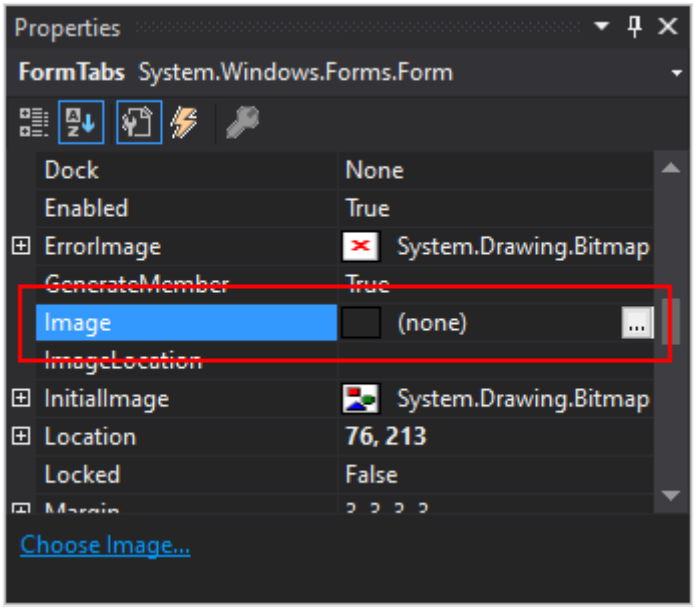
Article • 04/19/2024

Several Windows Forms controls can display images. These images can be icons that clarify the purpose of the control, such as a diskette icon on a button denoting the Save command. Alternatively, the icons can be background images to give the control the appearance and behavior you want.

Display an image - designer

In Visual Studio, use the Visual Designer to display an image.

1. Open the Visual Designer of the form containing the control to change.
2. Select the control.
3. In the **Properties** pane, select the **Image** or **BackgroundImage** property of the control.
4. Select the ellipsis (...) to display the **Select Resource** dialog box and then select the image you want to display.



Display an image - code

Set the control's `Image` or `BackgroundImage` property to an object of type Image. Generally, you'll load the image from a file by using the FromFile method.

In the following code example, the path set for the location of the image is the **My Pictures** folder. Most computers running the Windows operating system include this directory. This also enables users with minimal system access levels to run the application safely. The following code example requires that you already have a form with a PictureBox control added.

C#

```
// Replace the image named below with your own icon.
// Note the escape character used (@) when specifying the path.
pictureBox1.Image = Image.FromFile
   (System.Environment.GetFolderPath
   (System.Environment.SpecialFolder.MyPictures)
   + @"\Image.gif");
```

See also

- System.Drawing.Image.FromFile
- System.Drawing.Image
- System.Windows.Forms.Control.BackgroundImage

How to handle a control event (Windows Forms .NET)

Article • 07/30/2021

Events for controls (and for forms) are generally set through the Visual Studio Visual Designer for Windows Forms. Setting an event through the Visual Designer is known as handling an event at design-time. You can also handle events dynamically in code, known as handling events at run-time. An event created at run-time allows you to connect event handlers dynamically based on what your app is currently doing.

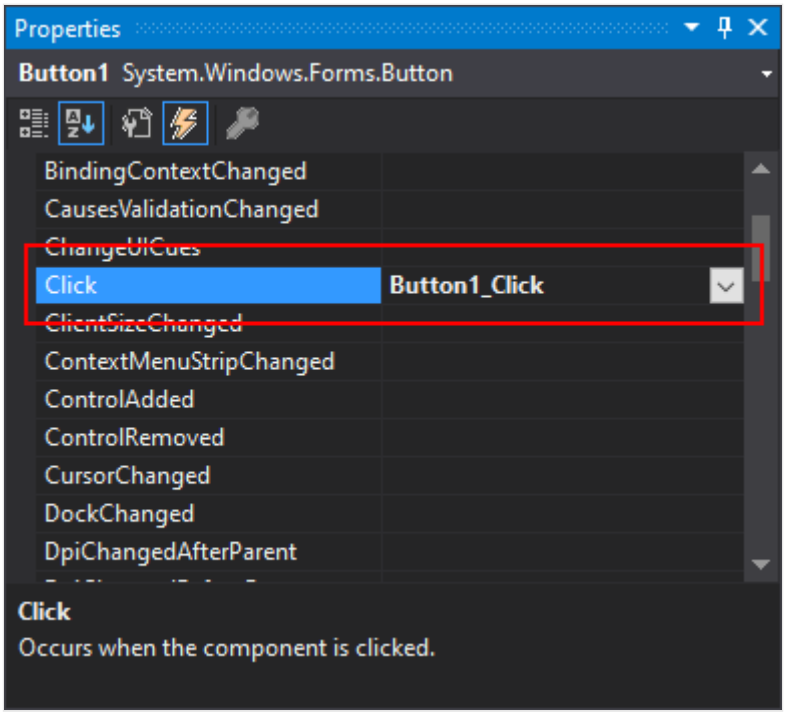
Handle an event - designer

In Visual Studio, use the Visual Designer to manage handlers for control events. The Visual Designer will generate the handler code and add it to the event for you.

Set the handler

Use the **Properties** pane to add or set the handler of an event:

1. Open the Visual Designer of the form containing the control to change.
2. Select the control.
3. Change the **Properties** pane mode to **Events** by pressing the events button ().
4. Find the event you want to add a handler to, for example, the **Click** event:



5. Do one of the following:

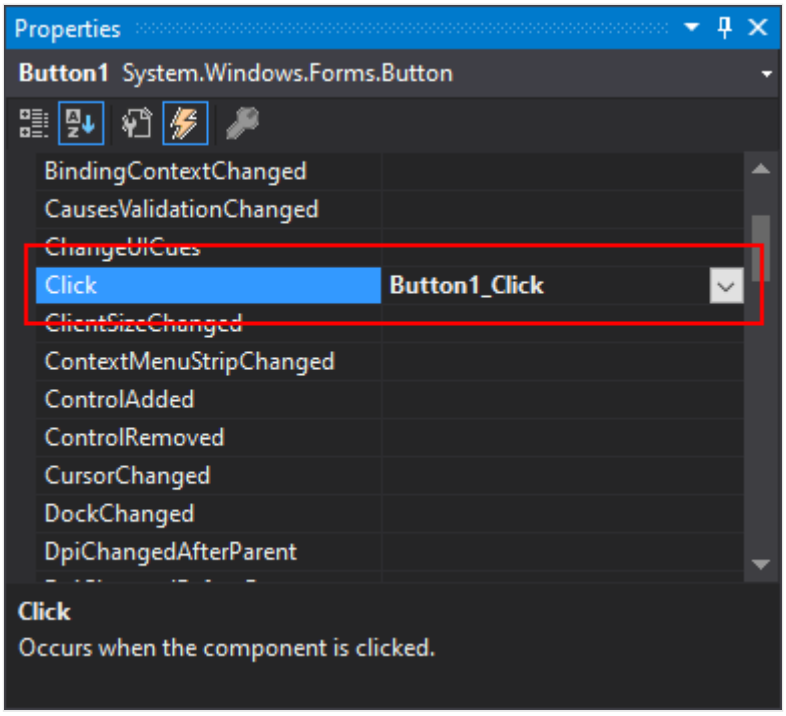
   - Double-click the event to generate a new handler, it's blank if no handler is assigned. If it's not blank, this action opens the code for the form and navigates to the existing handler.
   - Use the selection box ( ) to choose an existing handler.

     The selection box will list all methods that have a compatible method signature for the event handler.

Clear the handler

To remove an event handler, you can't just delete handler code that is in the form's code-behind file, it's still referenced by the event. Use the **Properties** pane to remove the handler of an event:

1. Open the Visual Designer of the form containing the control to change.
2. Select the control.
3. Change the **Properties** pane mode to **Events** by pressing the events button ( ).
4. Find the event containing the handler you want to remove, for example, the **Click** event:



5. Right-click on the event and choose **Reset**.

Handle an event - code

You typically add event handlers to controls at design-time through the Visual Designer. You can, though, create controls at run-time, which requires you to add event handlers in code. Adding handlers in code also gives you the chance to add multiple handlers to the same event.

Add a handler

The following example shows how to create a control and add an event handler. This control is created in the Button.Click event handler a different button. When **Button1** is pressed. The code moves and sizes a new button. The new button's `Click` event is handled by the `MyNewButton_Click` method. To get the new button to appear, it's added to the form's `Controls` collection. There's also code to remove the `Button1.Click` event's handler, this is discussed in the Remove the handler section.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    // Create and add the button
    Button myNewButton = new()
    {
        Location = new Point(10, 10),
        Size = new Size(120, 25),
        Text = "Do work"
```

```
    };

    // Handle the Click event for the new button
    myNewButton.Click += MyNewButton_Click;
    this.Controls.Add(myNewButton);

    // Remove this button handler so the user cannot do this twice
    button1.Click -= button1_Click;
}

private void MyNewButton_Click(object sender, EventArgs e)
{

}
```

To run this code, do the following to a form with the Visual Studio Visual Designer:

1. Add a new button to the form and name it **Button1**.
2. Change the **Properties** pane mode to **Events** by pressing the event button ().
3. Double-click the **Click** event to generate a handler. This action opens the code window and generates a blank `Button1_Click` method.
4. Replace the method code with the previous code above.

For more information about C# events, see Events (C#) For more information about Visual Basic events, see Events (Visual Basic)

Remove the handler

The Add a handler section used some code to demonstrate adding a handler. That code also contained a call to remove a handler:

C#

```
button1.Click -= button1_Click;
```

This syntax can be used to remove any event handler from any event.

For more information about C# events, see Events (C#) For more information about Visual Basic events, see Events (Visual Basic)

How to use multiple events with the same handler

With the Visual Studio Visual Designer's **Properties** pane, you can select the same handler already in use by a different event. Follow the directions in the Set the handler section to select an existing handler instead of creating a new one.

In C#, the handler is attached to a control's event in the form's designer code, which changed through the Visual Designer. For more information about C# events, see Events (C#)

Visual Basic

In Visual Basic, the handler is attached to a control's event in the form's code-behind file, where the event handler code is declared. Multiple `Handles` keywords can be added to the event handler code to use it with multiple events. The Visual Designer will generate the `Handles` keyword for you and add it to the event handler. However, you can easily do this yourself to any control's event and event handler, as long as the signature of the handler method matches the event. For more information about Visual Basic events, see Events (Visual Basic)

This code demonstrates how the same method can be used as a handler for two different Button.Click events:

VB

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click, Button2.Click
    'Do some work to handle the events
End Sub
```

See also

- Control events
- Events overview
- Using mouse events
- Using keyboard events
- System.Windows.Forms.Button

can also create and review issues and pull requests. For more information, see our contributor guide.

provide feedback:

- Open a documentation issue
- Provide product feedback

How to make thread-safe calls to controls (Windows Forms .NET)

Article • 04/19/2024

Multithreading can improve the performance of Windows Forms apps, but access to Windows Forms controls isn't inherently thread-safe. Multithreading can expose your code to serious and complex bugs. Two or more threads manipulating a control can force the control into an inconsistent state and lead to race conditions, deadlocks, and freezes or hangs. If you implement multithreading in your app, be sure to call cross-thread controls in a thread-safe way. For more information, see Managed threading best practices.

There are two ways to safely call a Windows Forms control from a thread that didn't create that control. Use the System.Windows.Forms.Control.Invoke method to call a delegate created in the main thread, which in turn calls the control. Or, implement a System.ComponentModel.BackgroundWorker, which uses an event-driven model to separate work done in the background thread from reporting on the results.

Unsafe cross-thread calls

It's unsafe to call a control directly from a thread that didn't create it. The following code snippet illustrates an unsafe call to the System.Windows.Forms.TextBox control. The `Button1_Click` event handler creates a new `WriteTextUnsafe` thread, which sets the main thread's TextBox.Text property directly.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    var thread2 = new System.Threading.Thread(WriteTextUnsafe);
    thread2.Start();
}

private void WriteTextUnsafe() =>
    textBox1.Text = "This text was set unsafely.";
```

The Visual Studio debugger detects these unsafe thread calls by raising an InvalidOperationException with the message, **Cross-thread operation not valid. Control accessed from a thread other than the thread it was created on.** The InvalidOperationException always occurs for unsafe cross-thread calls during Visual Studio debugging, and may occur at app runtime. You should fix the issue, but you can

disable the exception by setting the Control.CheckForIllegalCrossThreadCalls property to `false`.

Safe cross-thread calls

The following code examples demonstrate two ways to safely call a Windows Forms control from a thread that didn't create it:

1. The System.Windows.Forms.Control.Invoke method, which calls a delegate from the main thread to call the control.
2. A System.ComponentModel.BackgroundWorker component, which offers an event-driven model.

In both examples, the background thread sleeps for one second to simulate work being done in that thread.

Example: Use the Invoke method

The following example demonstrates a pattern for ensuring thread-safe calls to a Windows Forms control. It queries the System.Windows.Forms.Control.InvokeRequired property, which compares the control's creating thread ID to the calling thread ID. If they're different, you should call the Control.Invoke method.

The `WriteTextSafe` enables setting the TextBox control's Text property to a new value. The method queries InvokeRequired. If InvokeRequired returns `true`, `WriteTextSafe` recursively calls itself, passing the method as a delegate to the Invoke method. If InvokeRequired returns `false`, `WriteTextSafe` sets the TextBox.Text directly. The `Button1_Click` event handler creates the new thread and runs the `WriteTextSafe` method.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    var threadParameters = new System.Threading.ThreadStart(delegate {
WriteTextSafe("This text was set safely."); });
    var thread2 = new System.Threading.Thread(threadParameters);
    thread2.Start();
}

public void WriteTextSafe(string text)
{
    if (textBox1.InvokeRequired)
    {
```

```
        // Call this same method but append THREAD2 to the text
        Action safeWrite = delegate { WriteTextSafe($"{text} (THREAD2)"); };
        textBox1.Invoke(safeWrite);
    }
    else
        textBox1.Text = text;
}
```

Example: Use a BackgroundWorker

An easy way to implement multithreading is with the System.ComponentModel.BackgroundWorker component, which uses an event-driven model. The background thread raises the BackgroundWorker.DoWork event, which doesn't interact with the main thread. The main thread runs the BackgroundWorker.ProgressChanged and BackgroundWorker.RunWorkerCompleted event handlers, which can call the main thread's controls.

To make a thread-safe call by using BackgroundWorker, handle the DoWork event. There are two events the background worker uses to report status: ProgressChanged and RunWorkerCompleted. The `ProgressChanged` event is used to communicate status updates to the main thread, and the `RunWorkerCompleted` event is used to signal that the background worker has completed its work. To start the background thread, call BackgroundWorker.RunWorkerAsync.

The example counts from 0 to 10 in the `DoWork` event, pausing for one second between counts. It uses the ProgressChanged event handler to report the number back to the main thread and set the TextBox control's Text property. For the ProgressChanged event to work, the BackgroundWorker.WorkerReportsProgress property must be set to `true`.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    if (!backgroundWorker1.IsBusy)
        backgroundWorker1.RunWorkerAsync();
}

private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    int counter = 0;
    int max = 10;

    while (counter <= max)
    {
        backgroundWorker1.ReportProgress(0, counter.ToString());
        System.Threading.Thread.Sleep(1000);
```

```
        counter++;
    }
}

private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e) =>
    textBox1.Text = (string)e.UserState;
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Custom controls (Windows Forms .NET)

Article • 08/04/2023

With Windows Forms, you can create new controls or modify existing controls through inheritance. This article highlights the differences among the ways of creating new controls, and provides you with information about how to choose a particular type of control for your project.

Base control class

The Control class is the base class for Windows Forms controls. It provides the infrastructure required for visual display in Windows Forms applications and provides the following capabilities:

- Exposes a window handle.
- Manages message routing.
- Provides mouse and keyboard events, and many other user interface events.
- Provides advanced layout features.
- Contains many properties specific to visual display, such as ForeColor, BackColor, Height, and Width.

Because so much of the infrastructure is provided by the base class, it's relatively easy to develop your own Windows Forms controls.

Create your own control

There are three types of custom controls you can create: user controls, extended controls, and custom controls. The following table helps you decide which type of control you should create:

Expand table

If ...	Create a ...
• You want to combine the functionality of several Windows Forms controls into a single reusable unit.	Design a user control by inheriting from System.Windows.Forms.UserControl.
• Most of the functionality you need is already identical to an existing Windows Forms control.	Extend a control by inheriting from a specific Windows Forms control.

If ...	Create a ...
• You don't need a custom graphical user interface, or you want to design a new graphical user interface for an existing control.	
• You want to provide a custom graphical representation of your control. • You need to implement custom functionality that isn't available through standard controls.	Create a custom control by inheriting from System.Windows.Forms.Control.

User controls

A user control is a collection of Windows Forms controls presented as a single control to the consumer. This kind of control is referred to as a *composite control*. The contained controls are called *constituent controls*.

A user control holds all of the inherent functionality associated with each of the contained Windows Forms controls and enables you to selectively expose and bind their properties. A user control also provides a great deal of default keyboard handling functionality with no extra development effort on your part.

For example, a user control could be built to display customer address data from a database. This control would include a DataGridView control to display the database fields, a BindingSource to handle binding to a data source, and a BindingNavigator control to move through the records. You could selectively expose data binding properties, and you could package and reuse the entire control from application to application.

For more information, see User control overview.

Extended controls

You can derive an inherited control from any existing Windows Forms control. With this approach, you can keep all of the inherent functionality of a Windows Forms control, and then extend that functionality by adding custom properties, methods, or other features. With this option, you can override the base control's paint logic, and then extend its user interface by changing its appearance.

For example, you can create a control derived from the Button control that tracks how many times a user has clicked it.

In some controls, you can also add a custom appearance to the graphical user interface of your control by overriding the OnPaint method of the base class. For an extended button that tracks clicks, you can override the OnPaint method to call the base implementation of OnPaint, and then draw the click count in one corner of the Button control's client area.

Custom controls

Another way to create a control is to create one substantially from the beginning by inheriting from Control. The Control class provides all of the basic functionality required by controls, including mouse and keyboard handling events, but no control-specific functionality or graphical interface.

Creating a control by inheriting from the Control class requires more thought and effort than inheriting from UserControl or an existing Windows Forms control. Because a great deal of implementation is left for you, your control can have greater flexibility than a composite or extended control, and you can tailor your control to suit your exact needs.

To implement a custom control, you must write code for the OnPaint event of the control, which controls how the control is visually drawn. You must also write any feature-specific behaviors for your control. You can also override the WndProc method and handle windows messages directly. This is the most powerful way to create a control, but to use this technique effectively, you need to be familiar with the Microsoft Win32® API.

An example of a custom control is a clock control that duplicates the appearance and behavior of an analog clock. Custom painting is invoked to cause the hands of the clock to move in response to Tick events from an internal Timer component.

Custom design experience

If you need to implement a custom design-time experience, you can author your own designer. For composite controls, derive your custom designer class from the ParentControlDesigner or the DocumentDesigner classes. For extended and custom controls, derive your custom designer class from the ControlDesigner class.

Use the DesignerAttribute to associate your control with your designer.

The following information is out of date but may help you.

- (Visual Studio 2013) Extending Design-Time Support.
- (Visual Studio 2013) How to: Create a Windows Forms Control That Takes Advantage of Design-Time Features.

Guidelines to designing user controls (Windows Forms .NET)

Article • 04/19/2024

This article provides guidelines for creating user controls. We recommend that you follow these guidelines to make sure that you design a user control that's consistent with other Windows Forms controls.

Defining events

Events commonly communicate state change and alert you to how the user is interacting with a Windows Forms control. For more information about events and delegates, see Handle and raise events.

When defining your own events, follow these suggestions:

- Use the EventHandler event delegate when you define an event that doesn't have any associated data. Use the EventHandler<TEventArgs> event delegate when you do have associated data.
- Derive from EventArgs and extend it with your data, when you raise an event with associated data.
- Pass the control instance as the `sender` parameter.
- Create a method named `On{EventName}` that raises the event, which is marked as `protected` and `virtual` (in C#) or `Protected` and `Overridable` (in Visual Basic).

C#

```
// The event
public event EventHandler AllowInteractionChanged;

// The backing field for the property
private bool _allowInteraction;

// The property
public bool AllowInteraction
{
    get => _allowInteraction;
    set
    {
        // When the value has changed, call the method to raise the event
        if (_allowInteraction != value)
        {
            _allowInteraction = value;
            OnAllowInteractionChanged();
```

```
        }
    }
}

// Raises the event
public virtual void OnAllowInteractionChanged() =>
    AllowInteractionChanged?.Invoke(this, EventArgs.Empty);
```

Property changed events

If you want your control to send notifications when a property changes, define an event named `{PropertyName}Changed`. This is the naming convention used in Windows Forms. The associated event delegate type for property-changed events is EventHandler, and the event data type is EventArgs. The base class Control defines many property-changed events, such as BackColorChanged, BackgroundImageChanged, FontChanged, LocationChanged. When your property follows this naming convention, it receives bi-directional data binding support.

The same suggestions in the Defining events section apply here, as well.

C#

```
public class ProgressReportEventArgs : EventArgs
{
    public readonly int Value;
    public readonly int Maximum;

    public ProgressReportEventArgs(int value, int maximum) =>
        (Value, Maximum) = (value, maximum);
}

public event EventHandler<ProgressReportEventArgs> ProgressChanged;

public virtual void OnProgressChanged(int value, int maximum) =>
    ProgressChanged?.Invoke(this, new ProgressReportEventArgs(value,
maximum));
```

Properties

Control properties should support the Windows Forms Visual Designer. The **Properties** window interacts with control properties, and users expect to use this to change properties of the control. Either add the DefaultValueAttribute to the property, or create corresponding `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods. For more information, see DefaultValueAttribute and Reset and ShouldSerialize.

Properties that you don't want exposed to the Windows Forms Visual Designer should add the BrowsableAttribute to the property, passing false for the parameter of the attribute. This hides the property from the **Properties** window. For more information, see Define a property and Attributes for properties.

Extend an existing control

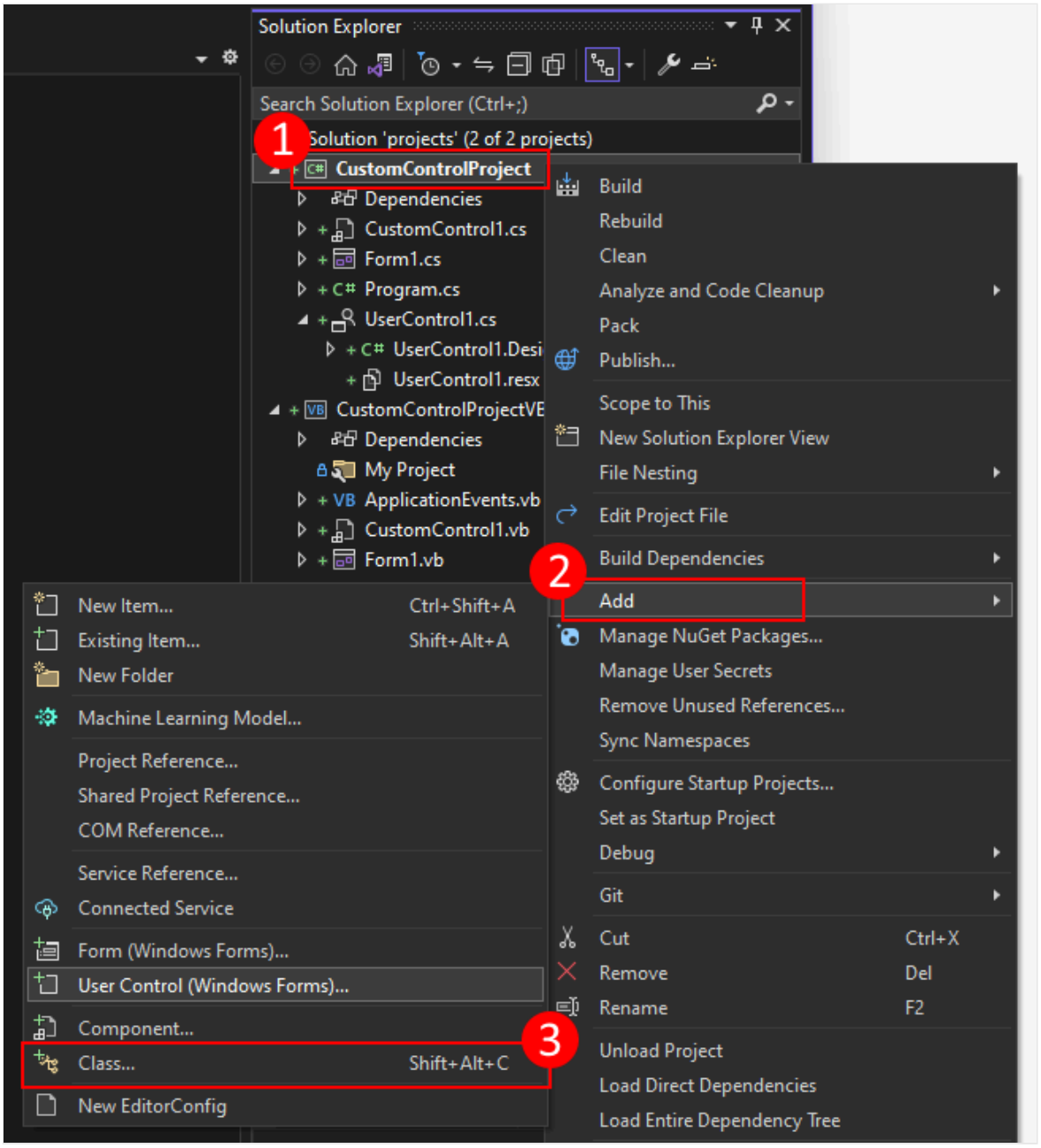
Article • 05/01/2024

If you want to add more features to an existing control, you can create a control that inherits from an existing control. The new control contains all of the capabilities and visual aspect of the base control, but gives you opportunity to extend it. For example, if you created a control that inherits Button, your new control would look and act exactly like a button. You could create new methods and properties to customize the behavior of the control. Some controls allow you to override the OnPaint method to change the way the control looks.

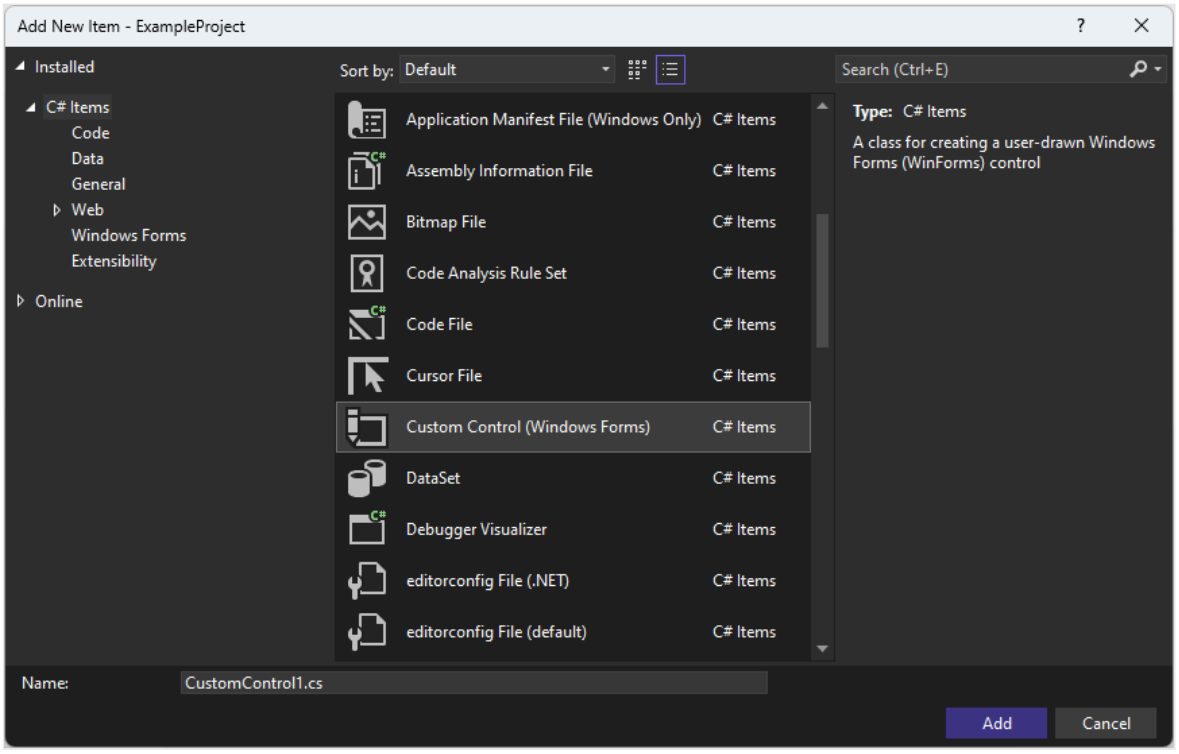
Add a custom control to a project

After creating a new project, use the Visual Studio templates to create a user control. The following steps demonstrate how to add a user control to your project:

1. In Visual Studio, find the **Project Explorer** pane. Right-click on the project and choose **Add** > **Class**.



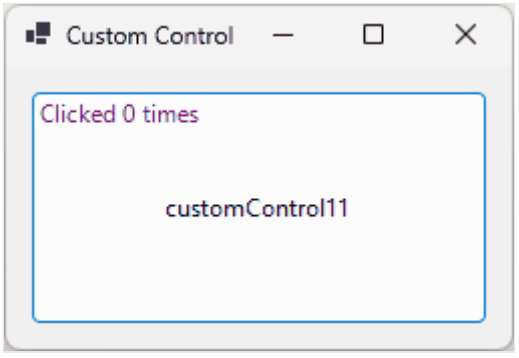
2. In the **Name** box, type a name for your user control. Visual Studio provides a default and unique name that you may use. Next, press **Add**.



After the user control is created, Visual Studio opens the code-editor for the control. The next step is to turn this custom control into a button and extend it.

Change the custom control to a button

In this section, you learn how to change a custom control into a button that counts and displays the number of times it's clicked.



After you add a custom control to your project named `CustomControl1`, the control designer should be opened. If it's not, double-click on the control in the **Solution Explorer**. Follow these steps to convert the custom control into a control that inherits from `Button` and extends it:

1. With the control designer opened, press `F7` or right-click on the designer window and select **View Code**.

2. In the code-editor, you should see a class definition:

C#

```
namespace CustomControlProject
{
    public partial class CustomControl2 : Control
    {
        public CustomControl2()
        {
            InitializeComponent();
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            base.OnPaint(pe);
        }
    }
}
```

3. Change the base class from `Control` to `Button`.

> **Important**
>
> If you're using **Visual Basic**, the base class is defined in the **.designer.vb* file of your control. The base class to use in Visual Basic is `System.Windows.Forms.Button`.

4. Add a class-scoped variable named `_counter`.

C#

```
private int _counter = 0;
```

5. Override the `OnPaint` method. This method draws the control. The control should draw a string on top of the button, so you must call the base class' `OnPaint` method first, then draw a string.

C#

```
protected override void OnPaint(PaintEventArgs pe)
{
    // Draw the control
    base.OnPaint(pe);

    // Paint our string on top of it
    pe.Graphics.DrawString($"Clicked {_counter} times", Font,
```

```
Brushes.Purple, new PointF(3, 3));
}
```

6. Lastly, override the `OnClick` method. This method is called every time the control is pressed. The code is going to increase the counter, and then call the `Invalidate` method, which forces the control to redraw itself.

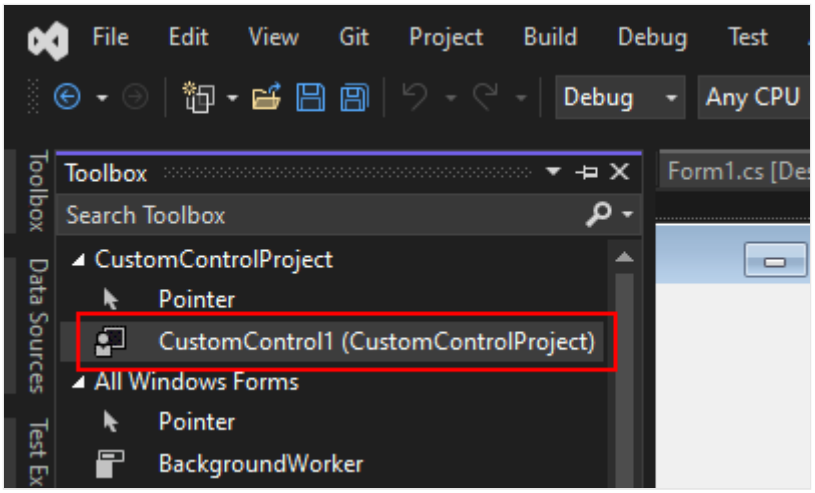
```
C#

protected override void OnClick(EventArgs e)
{
    // Increase the counter and redraw the control
    _counter++;
    Invalidate();

    // Call the base method to invoke the Click event
    base.OnClick(e);
}
```

The final code should look like the following snippet:

```
C#

public partial class CustomControl1 : Button
{
    private int _counter = 0;

    public CustomControl1()
    {
        InitializeComponent();
    }

    protected override void OnPaint(PaintEventArgs pe)
    {
        // Draw the control
        base.OnPaint(pe);

        // Paint our string on top of it
        pe.Graphics.DrawString($"Clicked {_counter} times", Font,
Brushes.Purple, new PointF(3, 3));
    }

    protected override void OnClick(EventArgs e)
    {
        // Increase the counter and redraw the control
        _counter++;
        Invalidate();

        // Call the base method to invoke the Click event
        base.OnClick(e);
```

```
    }
}
```

Now that the control is created, compile the project to populate the **Toolbox** window with the new control. Open a form designer and drag the control to the form. Run the project and press the button. Each press increases the number of clicks by one. The total clicks are printed as text on top of the button.



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

User control overview (Windows Forms .NET)

Article • 04/19/2024

A user control is a collection of Windows Forms controls encapsulated in a common container. This kind of control is referred to as a *composite control*. The contained controls are called *constituent controls*. User controls derive from the UserControl class.

User controls are designed like Forms, with a visual designer. You create, arrange, and modify, the constituent controls through the visual designer. The control events and logic are written exactly the same way as when you're designing a Form. The user control is placed on a Form just like any other control.

User controls are usable by the project in which they're created, or in other projects that have reference to the user control's library.

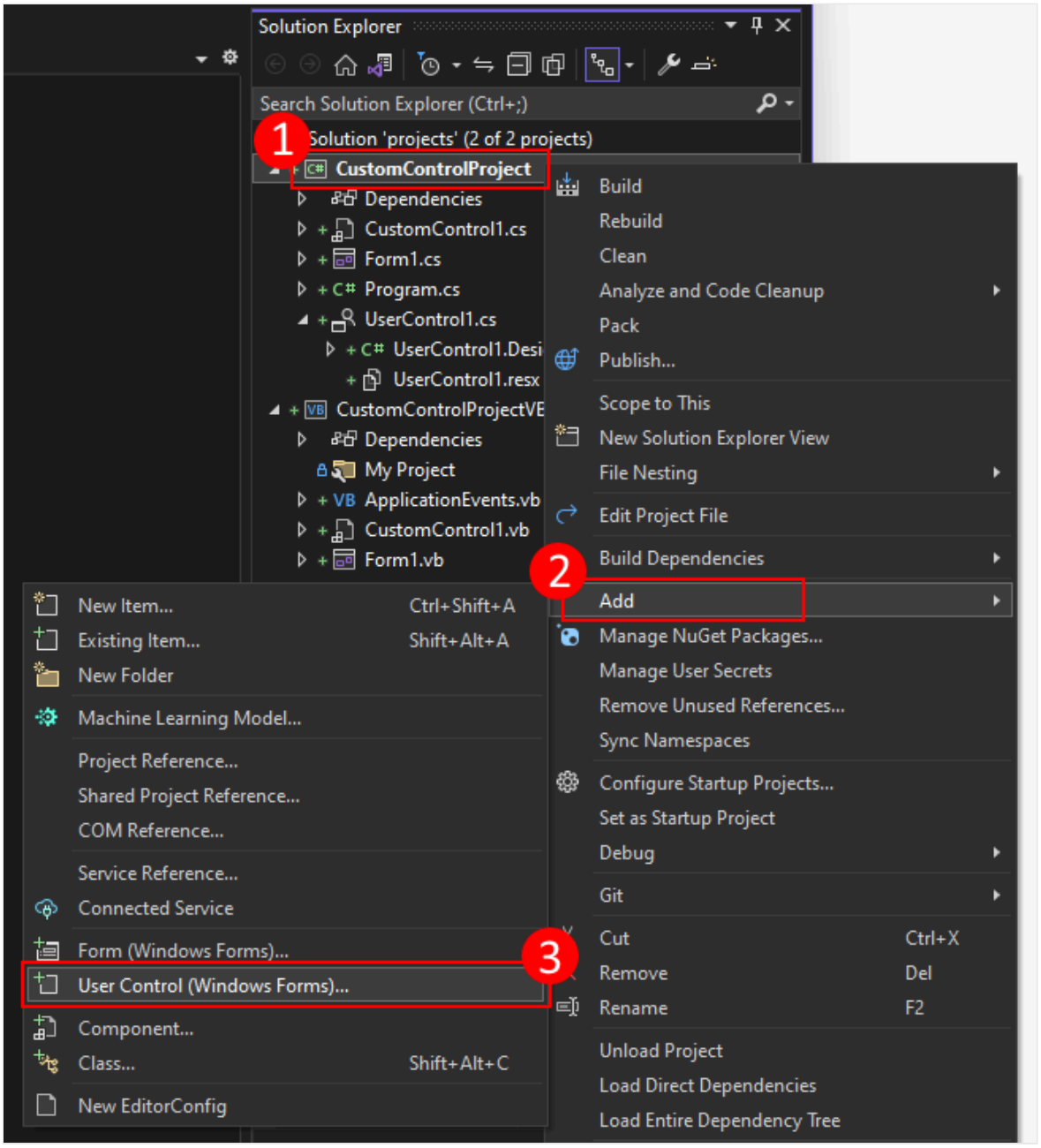
Constituent controls

The constituent controls are available to the user control, and the app user can interact with them all individually at runtime, but the properties and methods declared by the constituent controls aren't exposed to the consumer. For example, if you place a `TextBox` and `Button` control on the user control, the button's `Click` event is handled internally by the user control, but not by the Form where the user control is placed.

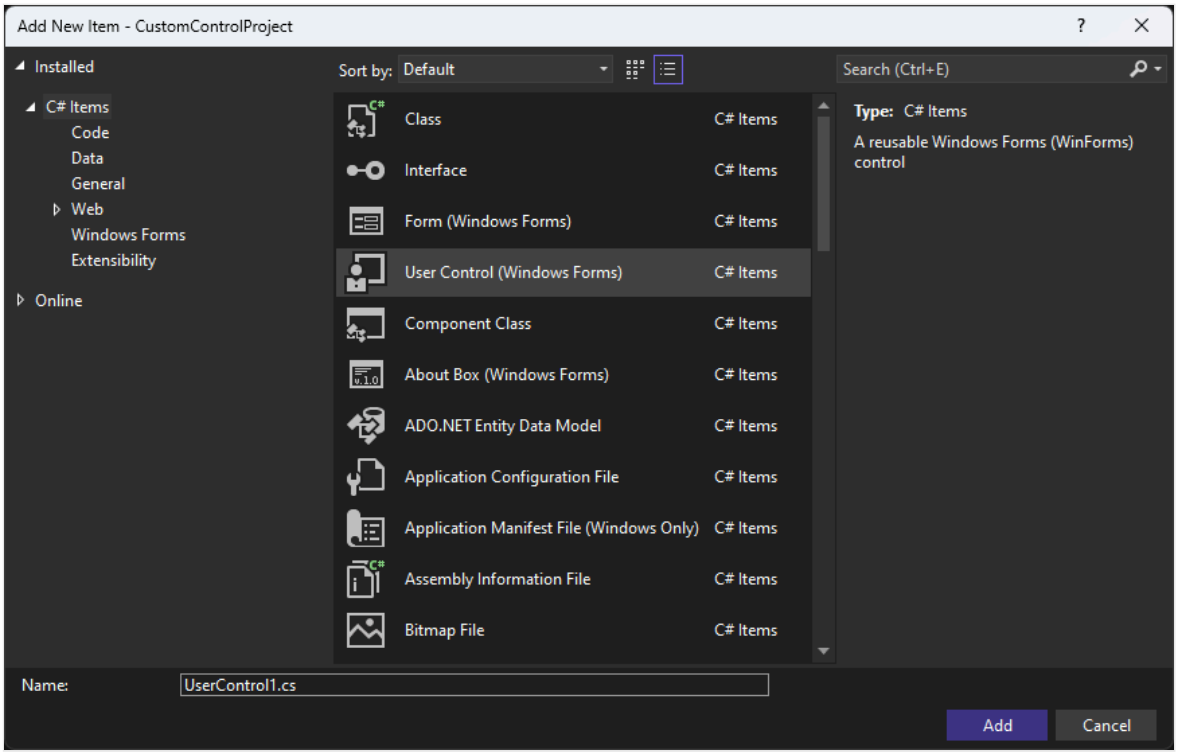
Add a user control to a project

After creating a new project, use the Visual Studio templates to create a user control. The following steps demonstrate how to add a user control to your project:

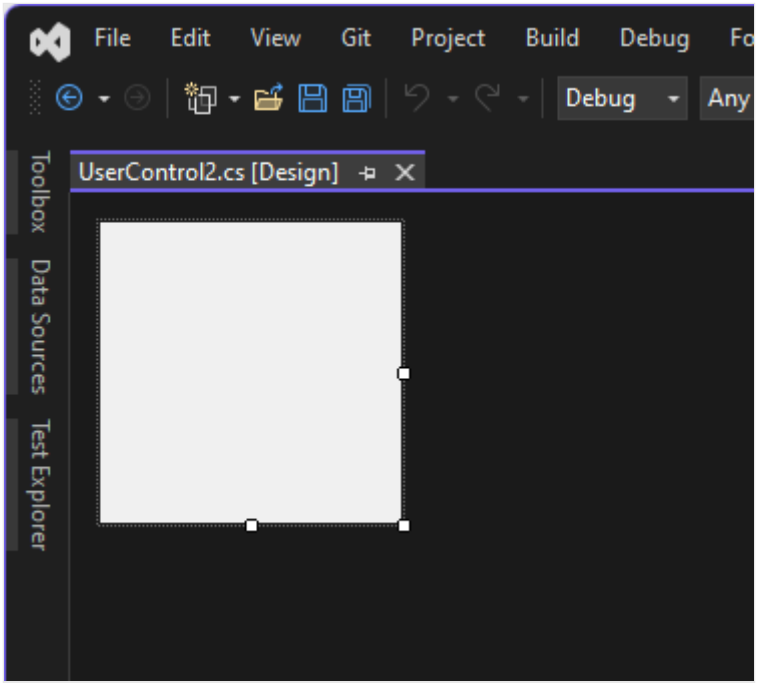
1. In Visual Studio, find the **Project Explorer** pane. Right-click on the project and choose **Add** > **User Control (Windows Forms)**.



2. In the **Name** box, type a name for your user control. Visual Studio provides a default and unique name that you may use. Next, press **Add**.



After the user control is created, Visual Studio opens the designer:



For an example of a working user control, see How to create a user control.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

more information, see our contributor guide.

Open a documentation issue

Provide product feedback

How to create a user control (Windows Forms .NET)

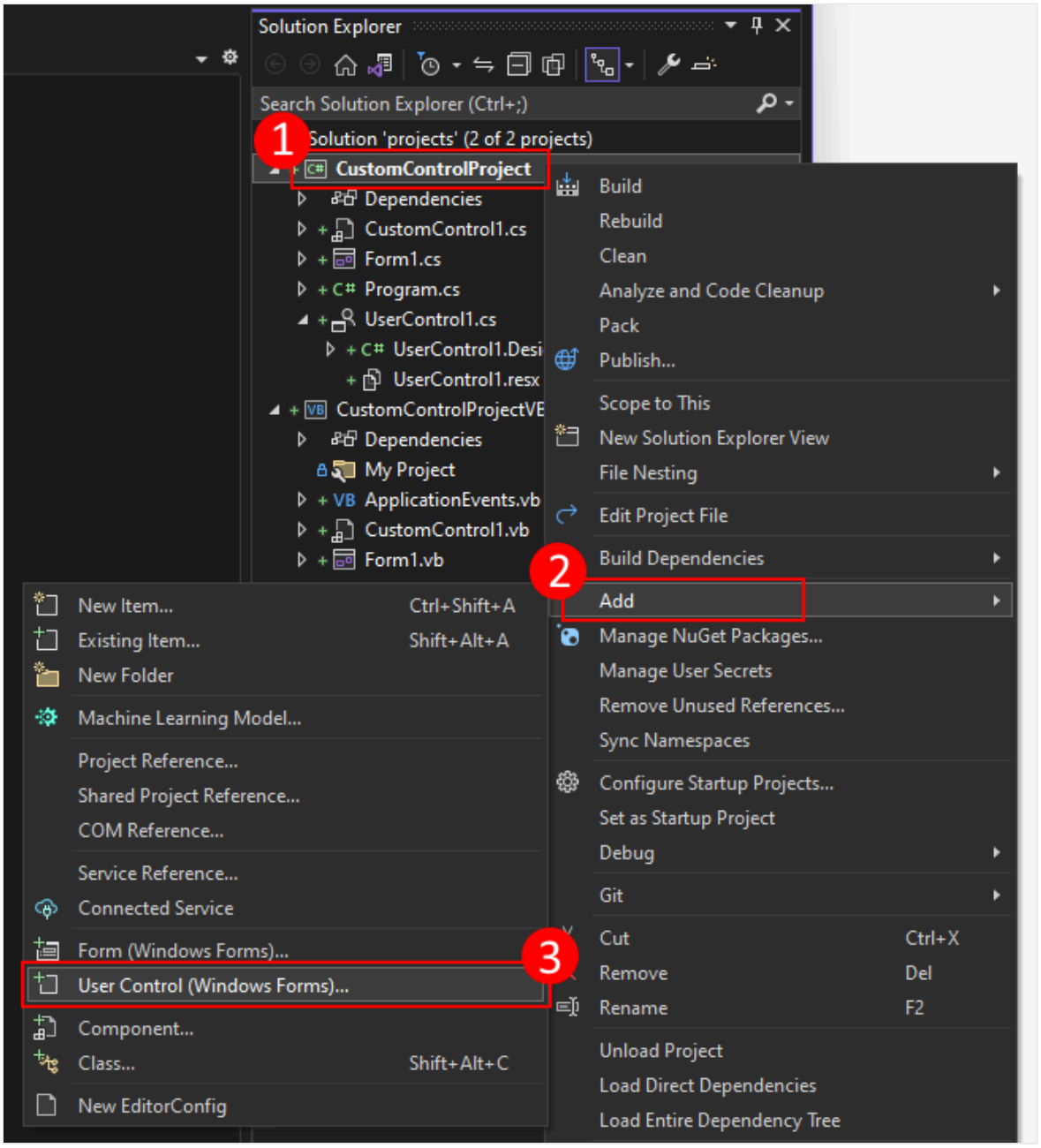
Article • 08/04/2023

This article teaches you how to add a user control to your project and then add that user control to a form. You'll create a reusable user control that's both visually appealing and functional. The new control groups a TextBox control with a Button control. When the user selects the button, the text in the text box is cleared. For more information about user controls, see User control overview.

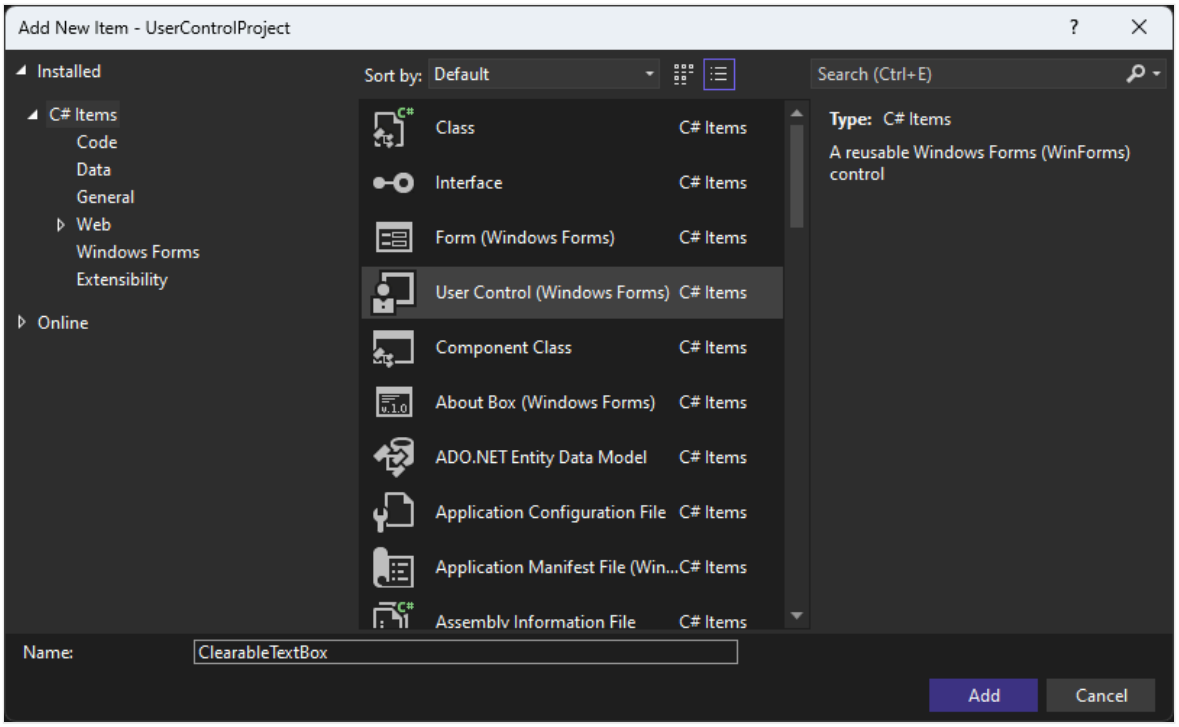
Add a user control to a project

After opening your Windows Forms project in Visual Studio, use the Visual Studio templates to create a user control:

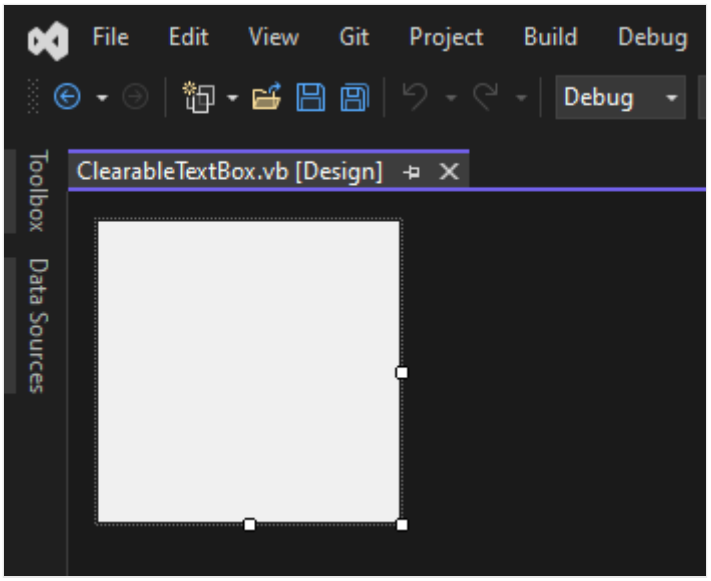
1. In Visual Studio, find the **Project Explorer** window. Right-click on the project and choose **Add** > **User Control (Windows Forms)**.



2. Set the **Name** of the control to **ClearableTextBox**, and press **Add**.



After the user control is created, Visual Studio opens the designer:



Design the clearable text box

The user control is made up of *constituent controls*, which are the controls you create on the design surface, just like how you design a form. Follow these steps to add and configure the user control and its constituent controls:

1. With the designer open, the user control design surface should be the selected object. If it's not, click on the design surface to select it. Set the following properties in the **Properties** window:

Expand table

Property	Value
MinimumSize	`84, 53`
Size	`191, 53`

2. Add a **Label** control. Set the following properties:

Expand table

Property	Value
Name	`lblTitle`
Location	`3, 5`

3. Add a **TextBox** control. Set the following properties:

Expand table

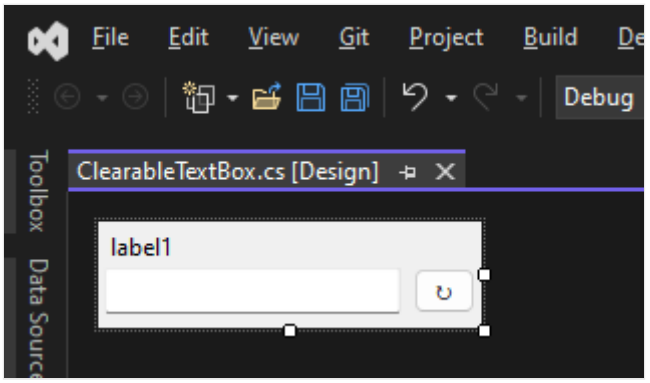
Property	Value
Name	`txtValue`
Anchor	`Top, Left, Right`
Location	`3, 23`
Size	`148, 23`

4. Add a **Button** control. Set the following properties:

Expand table

Property	Value
Name	`btnClear`
Anchor	`Top, Right`
Location	`157, 23`
Size	`31, 23`
Text	`↻`

The control should look like the following image:



5. Press `F7` to open the code editor for the `ClearableTextBox` class.

6. Make the following code changes:

   a. At the top of the code file, import the `System.ComponentModel` namespace.

   b. Add the `DefaultEvent` attribute to the class. This attribute sets which event is generated by the consumer when the control is double-clicked in the designer. The consumer being the object declaring and using this control. For more information about attributes, see Attributes (C#) or Attributes overview (Visual Basic).

   C#

   ```
   using System.ComponentModel;

   namespace UserControlProject
   {
       [DefaultEvent(nameof(TextChanged))]
       public partial class ClearableTextBox : UserControl
   ```

   c. Add an event handler that forwards the TextBox.TextChanged event to the consumer:

   C#

   ```
   [Browsable(true)]
   public new event EventHandler? TextChanged
   {
       add => txtValue.TextChanged += value;
       remove => txtValue.TextChanged -= value;
   }
   ```

   Notice that the event has the `Browsable` attribute declared on it. When the `Browsable` is applied to an event or property, it controls whether or not the item is visible in the **Properties** window when the control is selected in the designer.

In this case, `true` is passed as a parameter to the attribute indicating that the event should be visible.

d. Add a string property named `Text`, which forwards the TextBox.Text property to the consumer:

```
C#

[Browsable(true)]
public new string Text
{
    get => txtValue.Text;
    set => txtValue.Text = value;
}
```

e. Add a string property named `Title`, which forwards the Label.Text property to the consumer:

```
C#

[Browsable(true)]
public string Title
{
    get => lblTitle.Text;
    set => lblTitle.Text = value;
}
```

7. Switch back to the `ClearableTextBox` designer and double-click the `btnClear` control to generate a handler for the `Click` event. Add the following code for the handler, which clears the `txtValue` text box:

```
C#

private void btnClear_Click(object sender, EventArgs e) =>
    Text = "";
```

8. Finally, build the project by right-clicking the project in the **Solution Explorer** window, and selecting **Build**. There shouldn't be any errors, and after the build is finished, the `ClearableTextBox` control appears in the toolbox for use.

The next step is using the control in a form.

Sample application

If you created a new project in the last section, you have a blank Form named **Form1**, otherwise, create a new form.

1. In the **Solution Explorer** window, double-click the form to open the designer. The form's design surface should be selected.
2. Set the form's `Size` property to `432, 315`.
3. Open the **Toolbox** window, and double-click the **ClearableTextBox** control. This control should be listed under a section named after your project.
4. Again, double-click on the **ClearableTextBox** control to generate a second control.
5. Move back to the designer and separate the controls so that you can see both of them.
6. Select one control and set the following properties:

Expand table

Property	Value
Name	`ctlFirstName`
Location	`12, 12`
Size	`191, 53`
Title	`First Name`

7. Select the other control and set the following properties:

Expand table

Property	Value
Name	`ctlLastName`
Location	`12, 71`
Size	`191, 53`
Title	`Last Name`

8. Back in the **Toolbox** window, add a label control to the form, and set the following properties:

Expand table

Property	Value
Name	`lblFullName`
Location	`12, 252`

9. Next, you need to generate the event handlers for the two user controls. In the designer, double-click on the `ctlFirstName` control. This action generates the event handler for the `TextChanged` event, and opens the code editor.

10. Swap back to the designer and double-click the `ctlLastName` control to generate the second event handler.

11. Swap back to the designer and double-click on the form's title bar. This action generates an event handler for the `Load` event.

12. In the code editor, add a method named `UpdateNameLabel`. This method combines both names to create a message, and assigns the message to the `lblFullName` control.

```
C#
```

```
private void UpdateNameLabel()
{
    if (string.IsNullOrWhiteSpace(ctlFirstName.Text) ||
string.IsNullOrWhiteSpace(ctlLastName.Text))
        lblFullName.Text = "Please fill out both the first name and the
last name.";
    else
        lblFullName.Text = $"Hello {ctlFirstName.Text}
{ctlLastName.Text}, I hope you're having a good day.";
}
```

13. For both `TextChanged` event handlers, call the `UpdateNameLabel` method:

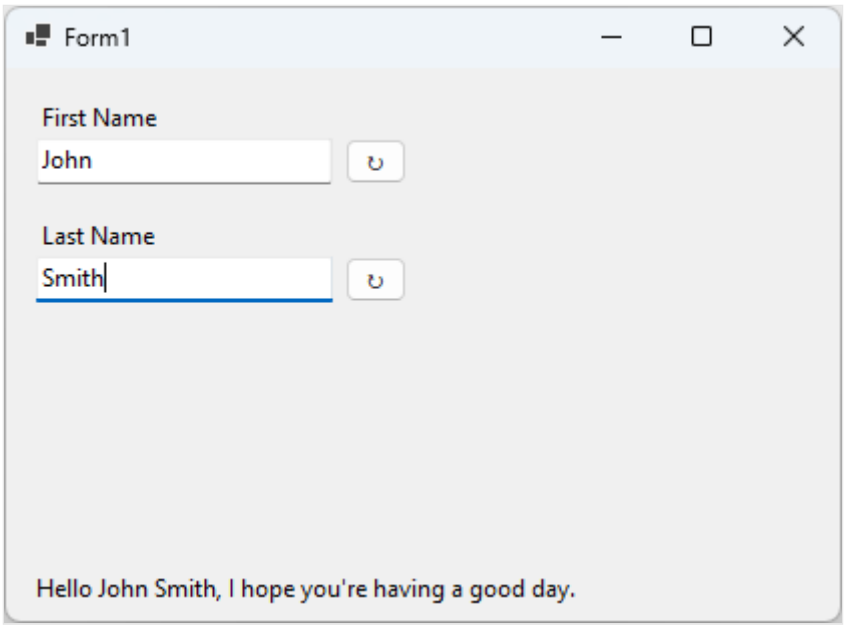
```
C#
```

```
private void ctlFirstName_TextChanged(object sender, EventArgs e) =>
    UpdateNameLabel();

private void ctlLastName_TextChanged(object sender, EventArgs e) =>
    UpdateNameLabel();
```

14. Finally, call the `UpdateNameLabel` method from the form's `Load` event:

```C#
private void Form1_Load(object sender, EventArgs e) =>
    UpdateNameLabel();
```

Run the project and enter a first and last name:



Try pressing the ↻ button to reset one of the text boxes.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Visual Studio design-time support for custom controls (Windows Forms .NET)

Article • 08/04/2023

As you've noticed when interacting with the Windows Forms designer, there are many different design-time features offered by the Windows Forms controls. Some of the features offered by the Visual Studio Designer include snap lines, action items, and the property grid. All of these features offer you an easier way to interact and customize a control during design-time. This article is an overview about what kind of support you can add to your custom controls, to make the design-time experience better for the consumers of your controls.

What's different from .NET Framework

Many basic design elements of custom controls have remained the same from .NET Framework. However, if you use more advanced designer customization features, such as action lists, type converters, custom dialogs, you have some unique scenarios to handle.

Visual Studio is a .NET Framework-based application, and as such, the Visual Designer you see for Windows Forms is also based on .NET Framework. With a .NET Framework project, both the Visual Studio environment and the Windows Forms app being designed run within the same process, **devenv.exe**. This poses a problem when you're working with a Windows Forms .NET (not .NET Framework) app. Both .NET and .NET Framework can't work within the same process. As a result, Windows Forms .NET uses a different designer, the "out-of-process" designer.

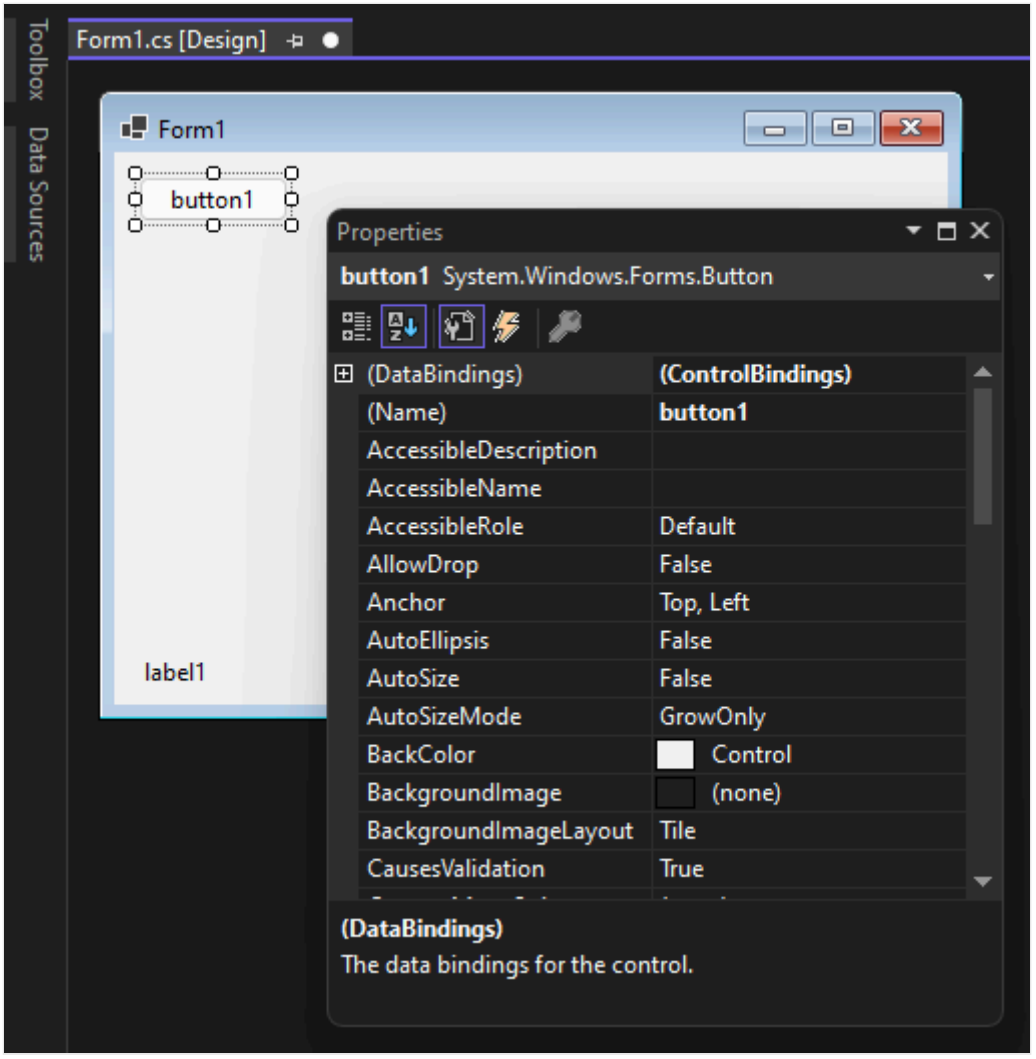
The out-of-process designer is a process called **DesignToolsServer.exe**, and is run along-side Visual Studio's **devenv.exe** process. The **DesignToolsServer.exe** process runs in the same version and platform, such as .NET 7 and x64, of .NET that your app is targeting. When your custom control needs to display UI in the **devenv.exe** your custom control must implement a client-server architecture to facilitate the communication to and from **devenv.exe**. For more information, see The designer changes since .NET Framework (Windows Forms .NET).

Property window

The Visual Studio **Properties** window displays the properties and events for the selected control or form. This is usually the first point of customization you perform on a custom

control or component.

The following image shows a `Button` control selected in the Visual Designer, and the properties grid showing the button's properties:



You can control some aspects of how information about your custom control appears in the properties grid. Attributes are applied either to the custom control class or class properties.

Attributes for classes

The following table shows the attributes you can apply to specify the behavior of your custom controls and components at design time.

Expand table

Attribute	Description
DefaultEventAttribute	Specifies the default event for a component.
DefaultPropertyAttribute	Specifies the default property for a component.

Attribute	Description
DesignerAttribute	Specifies the class used to implement design-time services for a component.
DesignerCategoryAttribute	Specifies that the designer for a class belongs to a certain category.
ToolboxItemAttribute	Represents an attribute of a toolbox item.
ToolboxItemFilterAttribute	Specifies the filter string and filter type to use for a Toolbox item.

Attributes for properties

The following table shows the attributes you can apply to properties or other members of your custom controls and components.

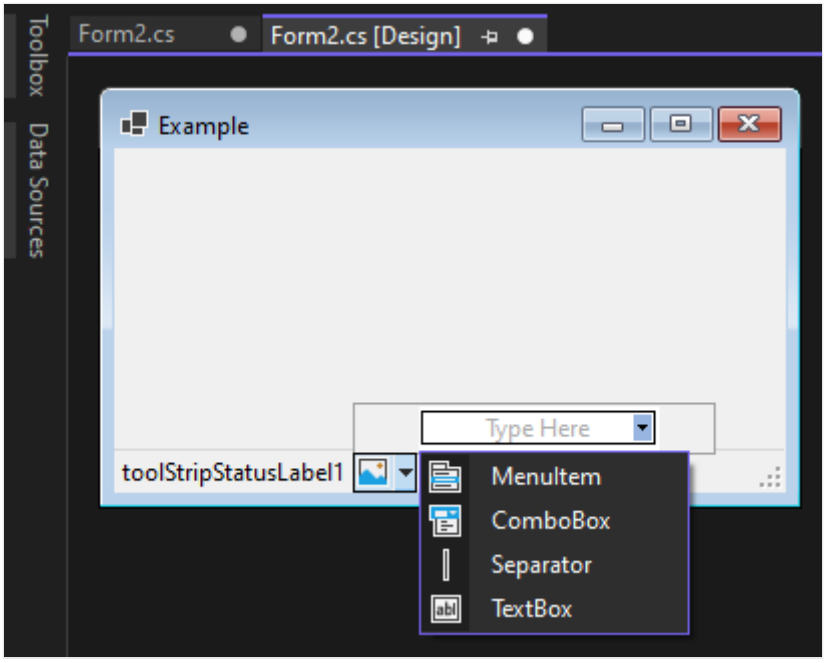
Expand table

Attribute	Description
AmbientValueAttribute	Specifies the value to pass to a property to cause the property to get its value from another source. This is known as *ambience*.
BrowsableAttribute	Specifies whether a property or event should be displayed in a **Properties** window.
CategoryAttribute	Specifies the name of the category in which to group the property or event when displayed in a PropertyGrid control set to Categorized mode.
DefaultValueAttribute	Specifies the default value for a property.
DescriptionAttribute	Specifies a description for a property or event.
DisplayNameAttribute	Specifies the display name for a property, event, or a public method that doesn't return a value and takes no arguments.
EditorAttribute	Specifies the editor to use to change a property.
EditorBrowsableAttribute	Specifies that a property or method is viewable in an editor.
HelpKeywordAttribute	Specifies the context keyword for a class or member.
LocalizableAttribute	Specifies whether a property should be localized.
PasswordPropertyTextAttribute	Indicates that an object's text representation is hidden by characters such as asterisks.
ReadOnlyAttribute	Specifies whether the property this attribute is bound to is read-only or read/write at design time.

Attribute	Description
RefreshPropertiesAttribute	Indicates that the property grid should refresh when the associated property value changes.
TypeConverterAttribute	Specifies what type to use as a converter for the object this attribute is bound to.

Custom control designers

The design-time experience for custom controls can be enhanced by authoring an associated custom designer. By default, your custom control is displayed on the host's design surface, and it looks the same as it does during run-time. With a custom designer you can enhance the design-time view of the control, add action items, snap lines, and other items, which can assist the user in determining how to lay out and configure the control. For example, at design-time the ToolStrip designer adds extra controls for the user to add, remove, and configure the individual items, as demonstrated in the following image:



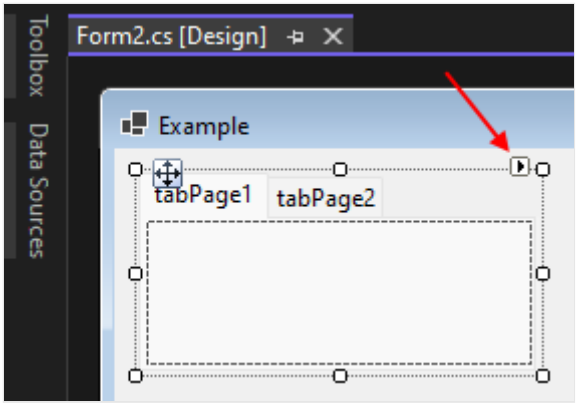
You can create your own custom designers by performing the following steps:

1. Adding reference to the Microsoft.WinForms.Designer.SDK NuGet package.
2. Create a type inherits from the `Microsoft.DotNet.DesignTools.Designers.ControlDesigner` class.
3. In your user control class, mark the class with the System.ComponentModel.DesignerAttribute class attribute, passing the type you created in the previous step.

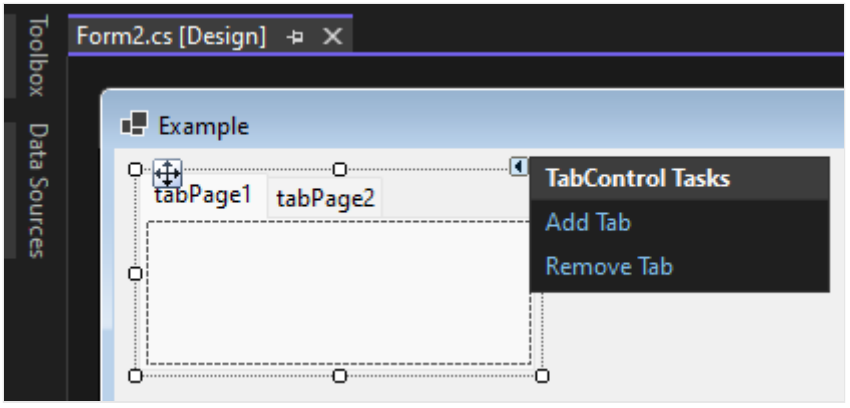
For more information, see the What's different from .NET Framework section.

Action items

Designer actions are context-sensitive menus that allow the user to perform common tasks quickly. For example, if you add a TabControl to a form, you're going to add and remove tabs to and from the control. Tabs are managed in the **Properties** window, through the TabPages property, which displays a tab collection editor. Instead of forcing the user to always sift through the **Properties** list looking for the `TabPages` property, the `TabControl` provides a smart tag button that's only visible when the control is selected, as shown in the following images:



When the smart tag is selected, the action list is displayed:



By adding the **Add Tab** and **Remove Tab** actions, the control's designer makes it so that you can quickly add or remove a tab.

Creating an action item list

Action item lists are provided by the `ControlDesigner` type you create. The following steps are a basic guide to creating your own action list:

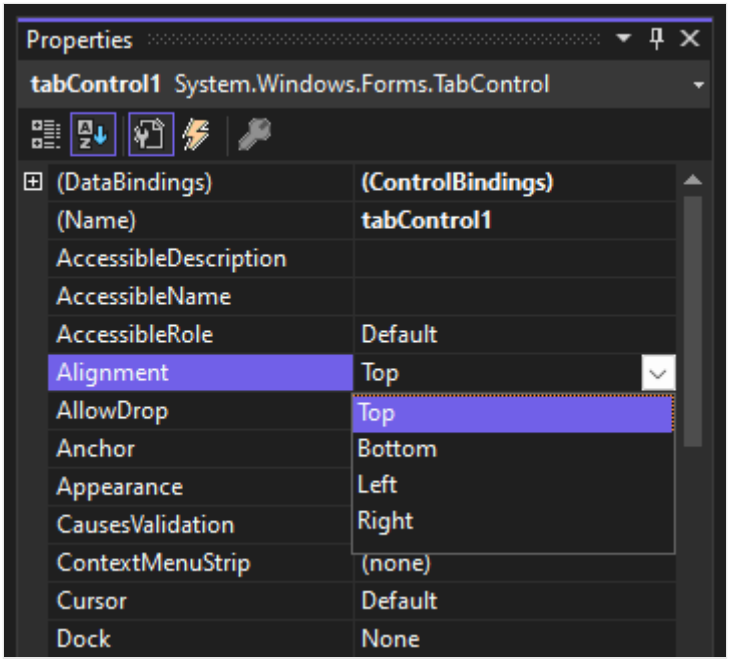
1. Adding reference to the Microsoft.WinForms.Designer.SDK NuGet package.

2. Create a new action list class that inherits from `Microsoft.DotNet.DesignTools.Designers.Actions.DesignerActionList`.
3. Add the properties to the action list you want the user to access. For example, adding a `bool` or `Boolean` (in Visual Basic) property to the class creates a CheckBox control in the action list.
4. Follow the steps in the Custom control designers section to create a new designer.
5. In the designer class, override the `ActionLists` property, which returns a `Microsoft.DotNet.DesignTools.Designers.Actions.DesignerActionListCollection` type.
6. Add your action list to a `DesignerActionListCollection` instance, and return it.

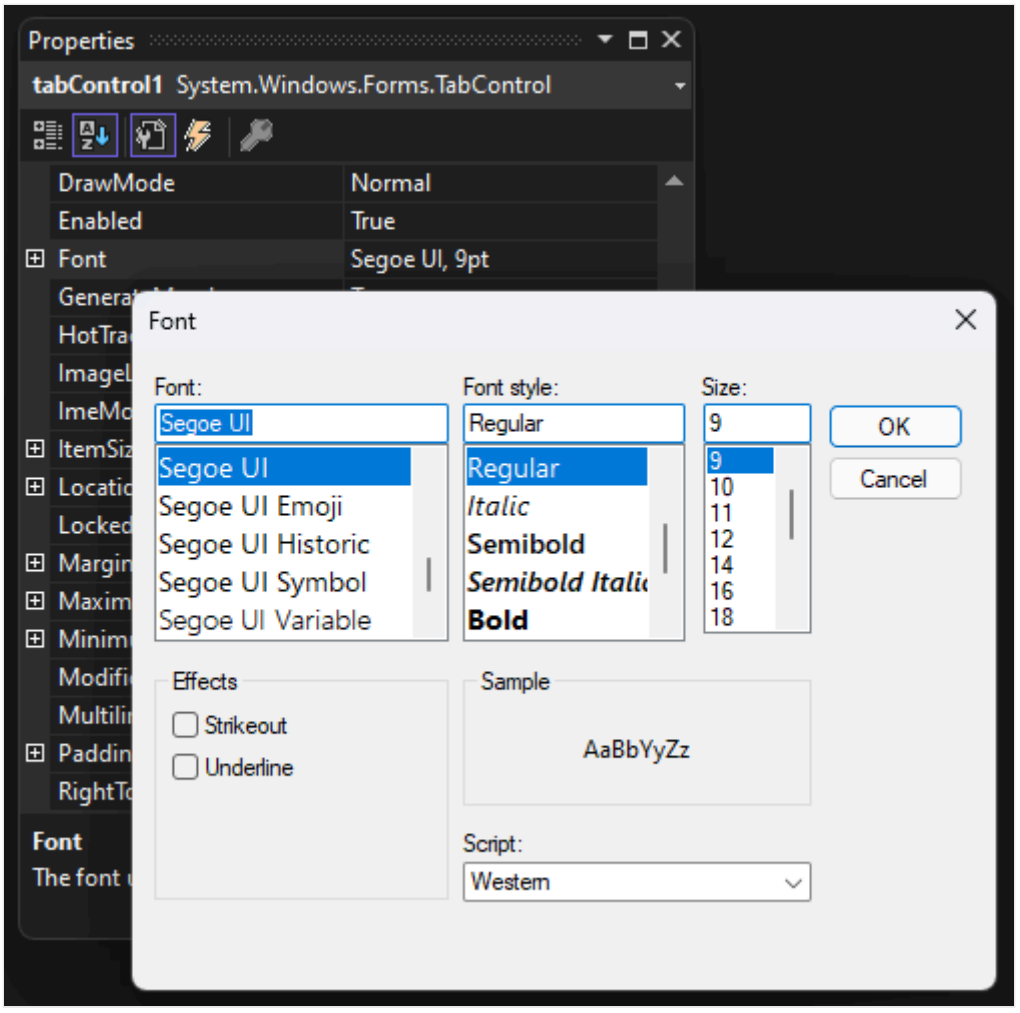
For an example of an action list, see the Windows Forms Designer Extensibility Documents & Samples GitHub repository, specifically the TileRepeater.Designer.Server/ControlDesigner folder.

Modal dialog type editors

In the **Properties** window, most properties are easily edited in the grid, such as when the property's backing type is an enumeration, boolean, or number.



Sometimes, a property is more complex and requires a custom dialog that the user can use to change the property. For example, the Font property is a System.Drawing.Font type, which contains many properties that alter what the font looks like. This isn't easily presentable in the **Properties** window, so this property uses a custom dialog to edit the font:



If your custom control properties are using the built-in type editors provided by Windows Forms, you can use the EditorAttribute to mark your properties with the corresponding .NET Framework editor you want Visual Studio to use. By using the built-in editors, you avoid the requirement of replicating the proxy-object client-server communication provided by the out-of-process designer.

When referencing a built-in type editor, use the .NET Framework type, not the .NET type:

C#

```
[Editor("System.Windows.Forms.Design.FileNameEditor, System.Design,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a",
        "System.Drawing.Design.UITypeEditor, System.Drawing,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]
public string? Filename { get; set; }
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to

can also create and review issues and pull requests. For more information, see our contributor guide.

provide feedback:

- Open a documentation issue
- Provide product feedback

The designer changes since .NET Framework (Windows Forms .NET)

Article • 08/04/2023

The Visual Designer for Windows Forms for .NET has had some improvements and changes since .NET Framework. These changes largely affect custom control designers. This article describes the key differences from .NET Framework.

Visual Studio is a .NET Framework-based application, and as such, the Visual Designer you see for Windows Forms is also based on .NET Framework. With a .NET Framework project, both the Visual Studio environment and the Windows Forms app being designed, run within the same process: **devenv.exe**. This poses a problem when you're working with a Windows Forms .NET (not .NET Framework) app. Both .NET and .NET Framework code can't work within the same process. As a result, Windows Forms .NET uses a different designer, the "out-of-process" designer.

Out-of-process designer

The out-of-process designer is a process called **DesignToolsServer.exe**, and is run along-side Visual Studio's **devenv.exe** process. The **DesignToolsServer.exe** process runs on the same version and platform of .NET that your app has been set up to target, such as .NET 7 and x64.

In the Visual Studio designer, .NET Framework proxy objects are created for each component and control on the designer, which communicate with the real .NET objects from your project in the **DesignToolsServer.exe** designer.

Control designers

For .NET, control designers need to be coded with the `Microsoft.WinForms.Designer.SDK` API, available on NuGet. This library is a refactoring of the .NET Framework designers for .NET. All of the designer types have moved to different namespaces but the type names are mostly the same. To update your .NET Framework designers for .NET, you must adjust the namespaces a bit.

- Designer classes and other related types, such as `ControlDesigner` and `ComponentTray`, have moved from the `System.Windows.Forms.Design` namespace to the `Microsoft.DotNet.DesignTools.Designers` namespace.

- Action list-related types in the `System.ComponentModel.Design` namespace have moved to the `Microsoft.DotNet.DesignTools.Designers.Actions` namespace.
- Behavior-related types, such as adorners and snaplines, in the `System.Windows.Forms.Design.Behavior` namespace have moved to the `Microsoft.DotNet.DesignTools.Designers.Behaviors` namespace.

Custom type editors

Custom type editors are more complicated than the control designers. Because the Visual Studio process is .NET Framework-based, any UI shown within the context of Visual Studio must be .NET Framework-based too. This design poses a problem, for example, when you're creating a .NET control that shows a custom type editor invoked by clicking on the `...` button in the property grid. The dialog can't be shown within the context of Visual Studio.

The out-of-process designer handles most of the control designer features, such as adorners, built-in type editors, and custom painting. Anytime you need to show a custom modal dialog, such as displaying new type editor, you need to replicate that proxy-object client-server communication that the out-of-process designer provides. This creates a lot more overhead than the old .NET Framework system.

If your custom control properties are using type editors provided by Windows Forms, you can use the EditorAttribute to mark your properties with the corresponding .NET Framework editor you want Visual Studio to use. By using the built-in editors, you avoid the requirement of replicating the proxy-object client-server communication provided by the out-of-process designer.

C#

```
[Editor("System.Windows.Forms.Design.FileNameEditor, System.Design,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a",
        "System.Drawing.Design.UITypeEditor, System.Drawing,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]
public string? Filename { get; set; }
```

Create a type editor

To create custom designers that provide type editors, you'll need a variety of projects, as described in the following list:

- `Control`: This project is your custom control library that contains the code for your controls. This is the library a user would reference when they want to use your

controls.

- `Control.Client`: A Windows Forms for .NET Framework project that contains your custom designer UI dialogs.
- `Control.Server`: A Windows Forms for .NET project that contains the custom designer code for your controls.
- `Control.Protocol`: A .NET Standard project that contains the communication classes used both by the `Control.Client` and `Control.Server` projects.
- `Control.Package`: A NuGet package project that contains all of the other projects. This package is formatted in a way that lets the Visual Studio Windows Forms for .NET tooling host and use your control library and designers.

Even if your type editor derives from an existing editor, such as ColorEditor or FileNameEditor, you still have to create that proxy-object client-server communication because you've provided a new UI class type that you want to display in the context of Visual Studio. However, the code to implement that type editor into Visual Studio is much simpler.

> **ⓘ Important**
>
> Documentation that describes this scenario in detail is in progress. Until that documentation is published, use the following blog post and sample to guide you in creating, publishing, and using this project structure:
>
> - Blog: Custom Controls for WinForm's Out-Of-Process Designer
> - TileRepeater control example

Design-time properties for custom controls (Windows Forms .NET)

Article • 08/04/2023

This article teaches you about how properties are handled for controls in the Windows Forms Visual Designer in Visual Studio.

Every control inherits many properties from the base class System.Windows.Forms.Control, such as:

- Enabled
- Font
- ForeColor
- Focused
- Visible
- Width

When creating a control, you can define new properties and control how they appear in the designer.

Define a property

Any public property with a **get** accessor defined by a control is automatically visible in the Visual Studio **Properties** window. If the property also defines a **set** accessor, the property can be changed in the **Properties** window. However, properties can be explicitly displayed or hidden from the **Properties** window by applying the BrowsableAttribute. This attribute takes a single boolean parameter to indicate whether or not it's displayed. For more information about attributes, see Attributes (C#) or Attributes overview (Visual Basic).

C#

```
[Browsable(false)]
public bool IsSelected { get; set; }
```

> [NOTE] Complex properties that can't be implicitly converted to and from a string require a type converter.

Serialized properties

Properties set on a control are serialized into the designer's code-behind file. This happens when the value of a property is set to something other than its default value.

When the designer detects a change to a property, it evaluates all properties for the control and serializes any property whose value doesn't match the default value for the property. The value of a property is serialized into the designer's code-behid file. Default values help the designer to determine which property values should be serialized.

Default values

A property is considered to have a default value when it either applies the DefaultValueAttribute attribute, or the property's class contains property-specific `Reset` and `ShouldSerialize` methods. For more information about attributes, see Attributes (C#) or Attributes overview (Visual Basic).

By setting a default value, you enable the following:

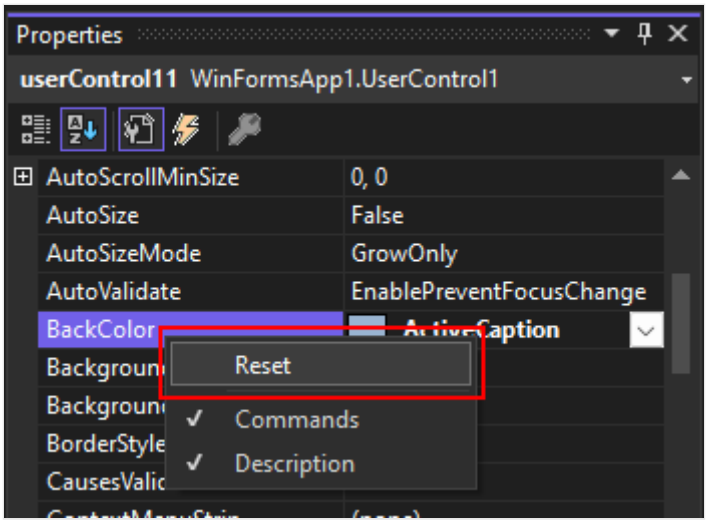
- The property provides visual indication in the **Properties** window if it has been modified from its default value.
- The user can right-click on the property and choose **Reset** to restore the property to its default value.
- The designer generates more efficient code.

If a property uses a simple type, such as a primitive type, the default value can be set by applying the `DefaultValueAttribute` to the property. However, properties with this attribute don't automatically start with that assigned value. You must set the property's backing field to the same default value. You can set the property on the declaration or in the class's constructor.

When a property is a complex type, or you want to control the designer's reset and serialization behavior, define the `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods on the class. For example, if the control defines an `Age` property, the methods are named `ResetAge` and `ShouldSerializeAge`.

> **ⓘ Important**
>
> Either apply the `DefaultValueAttribute` to the property, or provide both `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods. Don't use both.

Properties can be "reset" to their default values through the **Properties** window by right-clicking on the property name and selecting **Reset**.



The availability of the **Properties** > **Right-click** > **Reset** context menu option is enabled when:

- The property has the DefaultValueAttribute attribute applied, and the value of the property doesn't match the attribute's value.
- The property's class defines a `Reset<PropertyName>` method without a `ShouldSerialize<PropertyName>`.
- The property's class defines a `Reset<PropertyName>` method and the `ShouldSerialize<PropertyName>` returns true.

DefaultValueAttribute

If a property's value doesn't match the value provided by DefaultValueAttribute, the property is considered changed and can be reset through the **Properties** window.

> **ⓘ Important**
>
> This attribute shouldn't be used on properties that have corresponding `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods.

The following code declares two properties, an enumeration with a default value of `North` and an integer with a default value of 10.

C#

```
[DefaultValue(typeof(Directions), "North")]
public Directions PointerDirection { get; set; } = Directions.North;

[DefaultValue(10)]
public int DistanceInFeet { get; set; } = 10;
```

Reset and ShouldSerialize

As previously mentioned, the `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods provide the opportunity to guide not only the reset behavior of a property, but also in determining if a value is changed and should be serialized into the designer's code-behind file. Both methods work together and you shouldn't define one without the other.

> **ⓘ Important**
>
> The `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods shouldn't be created for a property that has a DefaultValueAttribute.

When `Reset<PropertyName>` is defined, the **Properties** window displays a **Reset** context menu option for that property. When **Reset** is selected, the `Reset<PropertyName>` method is invoked. The **Reset** context menu option is enabled or disabled by what is returned by the `ShouldSerialize<PropertyName>` method. When `ShouldSerialize<PropertyName>` returns `true`, it indicates that the property has changed from its default value and should be serialized into the code-behind file and enables the **Reset** context menu option. When `false` is returned, the **Reset** context menu option is disabled and the code-behind has the property-set code removed.

> **Tip**
>
> Both methods can and should be defined with private scope so that they don't make up the public API of the control.

The following code snippet declares a property named `Direction`. This property's designer behavior is controlled by the `ResetDirection` and `ShouldSerializeDirection` methods.

C#

```
public Directions Direction { get; set; } = Directions.None;

private void ResetDirection() =>
    Direction = Directions.None;

private bool ShouldSerializeDirection() =>
    Direction != Directions.None;
```

Type converters

While type converters typically convert one type to another, they also provide string-to-value conversion for the property grid and other design-time controls. String-to-value conversion allows complex properties to be represented in these design-time controls.

Most built-in data types (numbers, enumerations, and others) have default type converters that provide string-to-value conversions and perform validation checks. The default type converters are in the `System.ComponentModel` namespace and are named after the type being converted. The converter type names use the following format: `{type name}Converter`. For example, StringConverter, TimeSpanConverter, and Int32Converter.

Type converters are used extensively at design-time with the **Properties** window. A type converter can be applied to a property or a type, using the TypeConverterAttribute.

The **Properties** window uses converters to display the property as a string value when the `TypeConverterAttribute` is declared on the property. When the `TypeConverterAttribute` is declared on a type, the **Properties** window uses the converter on every property of that type. The type converter also helps with serializing the property value in the designer's code-behind file.

Type editors

The **Properties** window automatically uses a type editor for a property when the type of the property is a built-in or known type. For example, a boolean value is edited as a combo box with **True** and **False** values and the DateTime type uses a calendar dropdown.

> **Important**
>
> Custom type editors have changed since .NET Framework. For more information, see The designer changes since .NET Framework (Windows Forms .NET).

issues and pull requests. For more information, see our contributor guide.

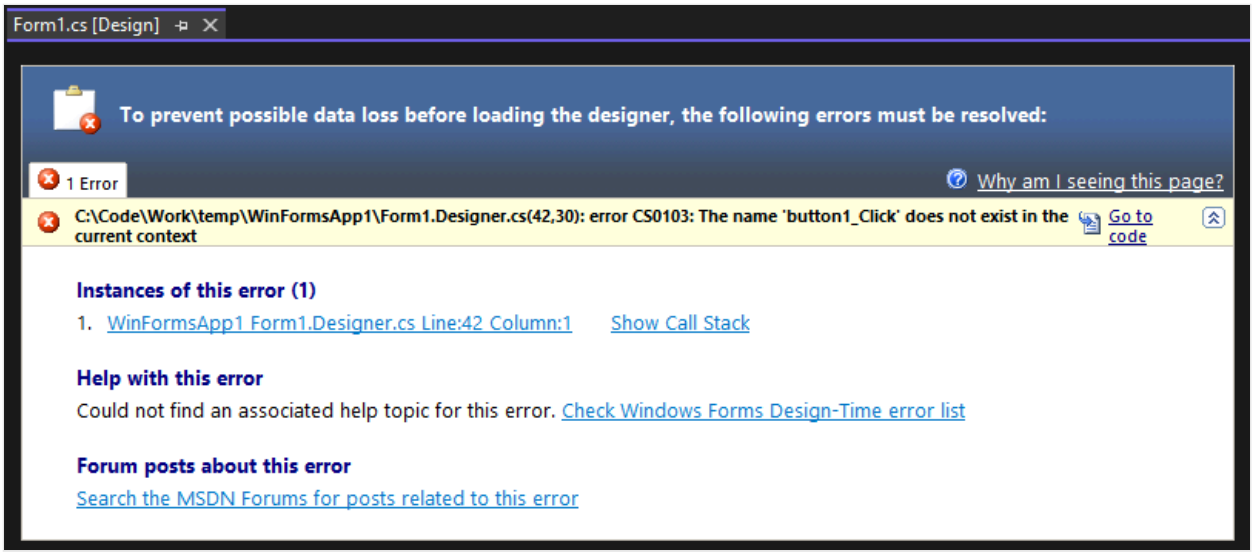
Open a documentation issue

Provide product feedback

Windows Forms Designer error page (Windows Forms .NET)

Article • 08/04/2023

If the Windows Forms Designer fails to load due to an error in your code, in a third-party component, or elsewhere, an error page is presented instead of the designer. This error page doesn't necessarily signify a bug in the designer. The bug may be somewhere in the code-behind file. Errors appear in collapsible, yellow bars with a link to jump to the location of the error on the code page.



Error window

The error window is made up of various parts.

- Yellow bar

  The yellow collapsible bar is created for every error, grouped by description. The bar describes the compiler error preventing the designer from loading property. It includes these details:
  - The file the error resides in.
  - The column and row in the file where the error occurs.
  - An error code.
  - A description of the error.
  - A link to navigate directly to the error.

- Instances of this error

When the yellow error bar is expanded, each instance of the error is listed. Many error types include an exact location in the following format: *<project name>* *<form name>* Line:*<line number>* Column:*<column number>*. If a call stack is associated with the error, you can select the **Show Call Stack** link to see it. Examining the call stack might further help you resolve the error.

> **Important**
>
> The elements of an error might vary based on the code language you're using.

- Help with this error

  If a help article for the error is available, select the **MSDN Help** link to navigate directly to the help page.

- Forum posts about this error

  Select the **Search the MSDN Forums for posts related to this error** link to navigate to the old Microsoft Developer Network forums. You might want to search or ask a question on the Microsoft Q&A or StackOverflow forums.

What to try first

You can often clear an error by cleaning and rebuilding the project or solution.

1. Find the **Solution Explorer** window.
2. Right-click on the solution or project, and select **Clean**.
3. Right-click on the solution or project, and select **Rebuild**.

You can also try to delete the *bin* and *obj* folders from the project folder. This might clear a temporary file or cause a **restore** action to happen, fixing a bad dependency.

Use the following section to triage common design-time errors.

Common design-time errors

This section lists some of the errors you may encounter.

- The name '<name>' does not exist in the current context
- '<identifier name>' is not a valid identifier
- '<name>' already exists in '<project name>'

- '<Toolbox tab name>' is not a toolbox category
- A requested language parser is not installed
- A service required for generating and parsing source code is missing
- An exception occurred while trying to create an instance of '<object name>'
- Another editor has '<document name>' open in an incompatible mode
- Another editor has made changes to '<document name>'
- Another editor has the file open in an incompatible mode
- Array rank '<rank in array>' is too high
- Assembly '<assembly name>' could not be opened
- Bad element type. This serializer expects an element of type '<type name>'
- Cannot access the Visual Studio Toolbox at this time
- Cannot bind an event handler to the '<event name>' event because it is read-only
- Cannot create a method name for the requested component because it is not a member of the design container
- Cannot name the object '<name>' because it is already named '<name>'
- Cannot remove or destroy inherited component '<component name>'
- Category '<Toolbox tab name>' does not have a tool for class '<class name>'
- Class '<class name>' has no matching constructor
- Code generation for property '<property name>' failed
- Component '<component name>' did not call Container.Add in its constructor
- Component name cannot be empty
- Could not access the variable '<variable name>' because it has not been initialized yet
- Could not find type '<type name>'
- Could not load type '<type name>'
- Could not locate the project item templates for inherited components
- Delegate class '<class name>' has no invoke method. Is this class a delegate
- Duplicate declaration of member '<member name>'
- Error reading resources from the resource file for the culture '<culture name>'
- Error reading resources from the resource file for the default culture '<culture name>'
- Failed to parse method '<method name>'
- Invalid component name: '<component name>'
- The type '<class name>' is made of several partial classes in the same file
- The assembly '<assembly name>' could not be found
- The assembly name '<assembly name>' is invalid
- The base class '<class name>' cannot be designed
- The base class '<class name>' could not be loaded
- The class '<class name>' cannot be designed in this version of Visual Studio
- The class name is not a valid identifier for this language

- The component cannot be added because it contains a circular reference to '<reference name>'
- The designer cannot be modified at this time
- The designer could not be shown for this file because none of the classes within it can be designed
- The designer for base class '<class name>' is not installed
- The designer must create an instance of type '<type name>', but it can't because the type is declared as abstract
- The file could not be loaded in the designer
- The language for this file does not support the necessary code parsing and generation services
- The language parser class '<class name>' is not implemented properly
- The name '<name>' is already used by another object
- The object '<object name>' does not implement the IComponent interface
- The object '<object name>' returned null for the property '<property name>' but this is not allowed
- The serialization data object is not of the proper type
- The service '<service name>' is required, but could not be located
- The service instance must derive from or implement '<interface name>'
- The text in the code window could not be modified
- The Toolbox enumerator object only supports retrieving one item at a time
- The Toolbox item for '<component name>' could not be retrieved from the Toolbox
- The Toolbox item for '<Toolbox item name>' could not be retrieved from the Toolbox
- The type '<type name>' could not be found
- The type resolution service may only be called from the main application thread
- The variable '<variable name>' is either undeclared or was never assigned
- There is already a command handler for the menu command '<menu command name>'
- There is already a component named '<component name>'
- There is already a Toolbox item creator registered for the format '<format name>'
- This language engine does not support a CodeModel with which to load a designer
- Type '<type name>' does not have a constructor with parameters of types '<parameter type names>'
- Unable to add reference '<reference name>' to the current application
- Unable to check out the current file
- Unable to find page named '<Options dialog box tab name>'
- Unable to find property '<property name>' on page '<Options dialog box tab name>'

- Visual Studio cannot open a designer for the file because the class within it does not inherit from a class that can be visually designed
- Visual Studio cannot save or load instances of the type '<type name>'
- Visual Studio is unable to open '<document name>' in Design view
- Visual Studio was unable to find a designer for classes of type '<type name>'

The name '<name>' does not exist in the current context

Most commonly you see this error when you delete or rename an event handler in the code-behind file that's reference by the designer file. Open the *<form>.designer.<langauge>* code file and delete the event handler from the form or control.

'<identifier name>' is not a valid identifier

This error indicates that a field, method, event, or object is improperly named.

'<name>' already exists in '<project name>'

You've specified a name for an inherited form that already exists in the project. To correct this error, give the inherited form a unique name.

'<Toolbox tab name>' is not a toolbox category

A third-party designer tried to access a tab on the Toolbox that doesn't exist. Contact the component vendor.

A requested language parser is not installed

Visual Studio attempted to a load a designer that's registered for the file type but could not. This is most likely because of an error that occurred during setup. Contact the vendor of the language you're using for a fix.

A service required for generating and parsing source code is missing

This error is a problem with a third-party component. Contact the component vendor.

An exception occurred while trying to create an instance of '<object name>'

A third-party designer requested that Visual Studio create an object, but the object raised an error. Contact the component vendor.

Another editor has '<document name>' open in an incompatible mode

This error arises if you try to open a file that is already opened in another editor. The editor that already has the file open is shown. To correct this error, close the editor that has the file open, and try again.

Another editor has made changes to '<document name>'

Close and reopen the designer for the changes to take effect. Normally, Visual Studio automatically reloads a designer after changes are made. However, other designers, such as third-party component designers, may not support reload behavior. In this case, Visual Studio prompts you to close and reopen the designer manually.

Another editor has the file open in an incompatible mode

This message is similar to "Another editor has '<document name>' open in an incompatible mode," but Visual Studio is unable to determine the file name. To correct this error, close the editor that has the file open, and try again.

Array rank '<rank in array>' is too high

Visual Studio only supports single-dimension arrays in the code block that's parsed by the designer. Multidimensional arrays are valid outside this area.

Assembly '<assembly name>' could not be opened

This error message arises when you try to open a file that could not be opened. Verify that the file exists and is a valid assembly.

Bad element type. This serializer expects an element of type '<type name>'

This error is a problem with a third-party component. Contact the component vendor.

Cannot access the Visual Studio Toolbox at this time

Visual Studio made a call to the Toolbox, which was not available. If you see this error, please log an issue by using Report a Problem.

Cannot bind an event handler to the '<event name>' event because it is read-only

This error most often arises when you've tried to connect an event to a control that's inherited from a base class. If the control's member variable is private, Visual Studio cannot connect the event to the method. Privately inherited controls cannot have extra events bound to them.

Cannot create a method name for the requested component because it is not a member of the design container

Visual Studio has tried to add an event handler to a component that does not have a member variable in the designer. Contact the component vendor.

Cannot name the object '<name>' because it is already named '<name>'

This error is an internal error in the Visual Studio serializer. It indicates that the serializer has tried to name an object twice, which is not supported. If you see this error, please log an issue by using Report a Problem.

Cannot remove or destroy inherited component '<component name>'

Inherited controls are under the ownership of their inheriting class. Changes to the inherited control must be made in the class from which the control originates. Thus, you cannot rename or destroy it.

Category '<Toolbox tab name>' does not have a tool for class '<class name>'

The designer tried to reference a class on a particular Toolbox tab, but the class does not exist. Contact the component vendor.

Class '<class name>' has no matching constructor

A third-party designer has asked Visual Studio to create an object with particular parameters in the constructor that does not exist. Contact the component vendor.

Code generation for property '<property name>' failed

This error is a generic wrapper for an error. The error string that accompanies this message gives more details about the error message and have a link to a more specific help article. To correct this error, address the error specified in the error message appended to this error.

Component '<component name>' did not call Container.Add() in its constructor

This message is related to an error in the component you loaded or placed on the form. It indicates that the component did not add itself to its container control (whether that is another control or a form). The designer continues to work, but there may be problems with the component at run time.

To correct the error, contact the component vendor. Or, if it is a component you created, call the `IContainer.Add` method in the component's constructor.

Component name cannot be empty

This error arises when you try to rename a component to an empty value.

Could not access the variable '<variable name>' because it has not been initialized yet

This error can arise because of two scenarios. Either a third-party component vendor has a problem with a control or component they have distributed, or the code you have written has recursive dependencies between components.

To correct this error, ensure that your code does not have a recursive dependency. If it is free of such problems, note the exact text of the error message and contact the component vendor.

Could not find type '<type name>'

> Error message: "Could not find type '<type name>'. Please make sure that the assembly that contains this type is referenced. If this type is a part of your development project, make sure that the project has been successfully built."

This error occurred because a reference was not found. Make sure the type indicated in the error message is referenced, and that any assemblies that the type requires are also referenced. Often, the problem is that a control in the solution has not been built. To build, select **Build Solution** from the **Build** menu. Otherwise, if the control has already been built, add a reference manually from the right-click menu of the **References** or **Dependencies** folder in Solution Explorer.

Could not load type '<type name>'

Visual Studio attempted to wire up an event-handling method and could not find one or more parameter types for the method. This error is usually caused by a missing reference. To correct this error, add the reference containing the type to the project and try again.

Could not locate the project item templates for inherited components

The templates for inherited forms in Visual Studio are not available. If you see this error, please log an issue by using Report a Problem.

Delegate class '<class name>' has no invoke method. Is this class a delegate

Visual Studio has tried to create an event handler, but there is something wrong with the event type. This error can happen if the event was created by a non-CLS-compliant language. Contact the component vendor.

Duplicate declaration of member '<member name>'

This error arises because a member variable has been declared twice (for example, two controls named `Button1` are declared in the code). Names must be unique across inherited forms. Additionally, names cannot differ only by case.

Error reading resources from the resource file for the culture '<culture name>'

This error can occur if there is a bad .resx file in the project.

To correct this error:

1. Select the **Show All Files** button in Solution Explorer to view the .resx files associated with the solution.
2. Load the .resx file in the XML Editor by right-clicking the .resx file and choosing **Open**.
3. Edit the .resx file manually to address the errors.

Error reading resources from the resource file for the default culture '<culture name>'

This error can occur if there is a bad .resx file in the project for the default culture.

To correct this error:

1. Select the **Show All Files** button in Solution Explorer to view the .resx files associated with the solution.
2. Load the .resx file in the XML Editor by right-clicking the .resx file and choosing **Open**.
3. Edit the .resx file manually to address the errors.

Failed to parse method '<method name>'

> Error message: "Failed to parse method '<method name>'. The parser reported the following error: '<error string>'. Please look in the Task List for potential errors."

This is a general error message for problems that arise during parsing. These errors are often due to syntax errors. See the Task List for specific messages related to the error.

Invalid component name: '<component name>'

You've tried to rename a component to an invalid value for that language. To correct this error, name the component such that it complies with the naming rules for that language.

The type '<class name>' is made of several partial classes in the same file

When you define a class in multiple files by using the partial keyword, you can only have one partial definition in each file.

To correct this error, remove all but one of the partial definitions of your class from the file.

The assembly '<assembly name>' could not be found

This error is similar to "The type '<type name>' could not be found," but this error usually happens because of a metadata attribute. To correct this error, check that all assemblies used by attributes are referenced.

The assembly name '<assembly name>' is invalid

A component has requested a particular assembly, but the name provided by the component is not a valid assembly name. Contact the component vendor.

The base class '<class name>' cannot be designed

Visual Studio loaded the class, but the class cannot be designed because the implementer of the class did not provide a designer. If the class supports a designer, make sure there are no problems that would cause issues with displaying it in a designer, such as compiler errors. Also, make sure that all references to the class are correct and all class names are correctly spelled. Otherwise, if the class is not designable, edit it in Code view.

The base class '<class name>' could not be loaded

The class is not referenced in the project, so Visual Studio can't load it. To correct this error, add a reference to the class in the project, and close and reopen the Windows Forms Designer window.

The class '<class name>' cannot be designed in this version of Visual Studio

The designer for this control or component does not support the same types that Visual Studio does. Contact the component vendor.

The class name is not a valid identifier for this language

The source code created by the user has a class name that isn't valid for the language being used. To correct this error, name the class such that it conforms to the language requirements.

The component cannot be added because it contains a circular reference to '<reference name>'

You cannot add a control or component to itself. Another situation where this might occur is if there is code in the InitializeComponent method of a form (for example, `Form1`) that creates another instance of `Form1`.

The designer cannot be modified at this time

This error occurs when the file in the editor is marked as read-only. Ensure that the file is not marked read-only and the application is not running.

The designer could not be shown for this file because none of the classes within it can be designed

This error occurs when Visual Studio cannot find a base class that satisfies designer requirements. Forms and controls must derive from a base class that supports designers. If you're deriving from an inherited form or control, make sure the project has been built.

The designer for base class '<class name>' is not installed

Visual Studio could not load the designer for the class. If you see this error, please log an issue by using Report a Problem.

The designer must create an instance of type '<type name>', but it can't because the type is declared as abstract

This error occurred because the base class of the object being passed to the designer is abstract, which is not allowed.

The file could not be loaded in the designer

The base class of this file does not support any designers. As a workaround, use Code view to work on the file. Right-click the file in Solution Explorer and choose **View Code**.

The language for this file does not support the necessary code parsing and generation services

Error message: "The language for this file does not support the necessary code parsing and generation services. Ensure the file you are opening is a member of a project and then try to open the file again."

This error most likely resulted from opening a file that's in a project that does not support designers.

The language parser class '<class name>' is not implemented properly

Error message: "The language parser class '<class name>' is not implemented properly. Contact the vendor for an updated parser module."

The language in use has registered a designer class that doesn't derive from the correct base class. Contact the vendor of the language you're using.

The name '<name>' is already used by another object

This is an internal error in the Visual Studio serializer. If you see this error, please log an issue by using Report a Problem.

The object '<object name>' does not implement the `IComponent` interface

Visual Studio tried to create a component, but the object created does not implement the IComponent interface. Contact the component vendor for a fix.

The object '<object name>' returned null for the property '<property name>' but this is not allowed

There are some .NET properties that should always return an object. For example, the **Controls** collection of a form should always return an object, even when there are no controls in it.

To correct this error, ensure that the property specified in the error is not null.

The serialization data object is not of the proper type

A data object offered by the serializer is not an instance of a type that matches the current serializer being used. Contact the component vendor.

The service '<service name>' is required, but could not be located

A service required by Visual Studio is unavailable. If you tried to load a project that doesn't support that designer, use the Code Editor to make the changes instead. Otherwise, If you see this error, please log an issue by using Report a Problem.

The service instance must derive from or implement '<interface name>'

This error indicates that a component or component designer has called the **AddService** method, which requires an interface and object, but the object specified does not implement the interface specified. Contact the component vendor.

The text in the code window could not be modified

This error occurs when Visual Studio is unable to edit a file due to disk space or memory problems, or the file is marked read-only.

The Toolbox enumerator object only supports retrieving one item at a time

If you see this error, If you see this error, please log an issue by using Report a Problem.

The Toolbox item for '<component name>' could not be retrieved from the Toolbox

The component in question threw an exception when Visual Studio accessed it. Contact the component vendor.

The Toolbox item for '<Toolbox item name>' could not be retrieved from the Toolbox

This error occurs if the data within the Toolbox item becomes corrupted or the version of the component has changed. Try removing the item from the Toolbox and adding it back again.

The type '<type name>' could not be found

When the designer is loaded, Visual Studio failed to find a type. Ensure that the assembly containing the type is referenced. If the assembly is part of the current development project, ensure that the project has been built.

The type resolution service may only be called from the main application thread

Visual Studio attempted to access required resources from the wrong thread. This error is displayed when the code used to create the designer has called the type resolution service from a thread other than the main application thread. To correct this error, call the service from the correct thread or contact the component vendor.

The variable '<variable name>' is either undeclared or was never assigned

The source code has a reference to a variable, such as **Button1**, that isn't declared or assigned. If the variable has not been assigned, this message appears as a warning, not an error.

There is already a command handler for the menu command '<menu command name>'

This error arises if a third-party designer adds a command that already has a handler to the command table. Contact the component vendor.

There is already a component named '<component name>'

> Error message: "There is already a component named '<component name>'. Components must have unique names, and names must not be case-sensitive. A name also cannot conflict with the name of any component in an inherited class."

This error message arises when there has been a change to the name of a component in the Properties window. To correct this error, ensure that all component names are unique, are not case-sensitive, and do not conflict with the names of any components in the inherited classes.

There is already a Toolbox item creator registered for the format '<format name>'

A third-party component made a callback to an item on a Toolbox tab, but the item already contained a callback. Contact the component vendor.

This language engine does not support a CodeModel with which to load a designer

This message is similar to "The language for this file does not support the necessary code parsing and generation services," but this message involves an internal registration problem. If you see this error, If you see this error, please log an issue by using Report a Problem.

Type '<type name>' does not have a constructor with parameters of types '<parameter type names>'

Visual Studio could not find a constructor that had matching parameters. This error may be the result of supplying a constructor with types other than those that are required. For example, a **Point** constructor might take two integers. If you provided float types, this error is raised.

To correct this error, use a different constructor or explicitly cast the parameter types such that they match those provided by the constructor.

Unable to add reference '<reference name>' to the current application

Visual Studio is unable to add a reference. To correct this error, check that a different version of the reference is not already referenced.

Unable to check out the current file

This error arises when you change a file that's currently checked in to source-code control. Usually, Visual Studio presents the file checkout dialog box so that the user can

check out the file. This time, the file was not checked out, perhaps because of a merge conflict during checkout. To correct this error, ensure that the file is not locked, and then try to check out the file manually.

Unable to find page named '<Options dialog box tab name>'

This error arises when a component designer requests access to a page from the Options dialog box by using a name that does not exist. Contact the component vendor.

Unable to find property '<property name>' on page '<Options dialog box tab name>'

This error arises when a component designer requests access to a particular value on a page from the Options dialog box, but that value does not exist. Contact the component vendor.

Visual Studio cannot open a designer for the file because the class within it does not inherit from a class that can be visually designed

Visual Studio loaded the class, but the designer for that class could not be loaded. Visual Studio requires that designers use the first class in a file. To correct this error, move the class code so that it is the first class in the file, and then load the designer again.

Visual Studio cannot save or load instances of the type '<type name>'

This is a problem with a third-party component. Contact the component vendor.

Visual Studio is unable to open '<document name>' in Design view

This error indicates that the language of the project does not support a designer and arises when you attempt to open a file in the Open File dialog box or from Solution Explorer. Instead, edit the file in Code view.

Visual Studio was unable to find a designer for classes of type '<type name>'

Visual Studio loaded the class, but the class cannot be designed. Instead, edit the class in Code view by right-clicking the class and choosing **View Code**.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Troubleshoot 32-bit problems (Windows Forms .NET)

Article • 03/28/2024

After upgrading to Visual Studio 2022, you might run into a problem where the design-time experience of your app stops working. This could be related to referencing a 32-bit component. Visual Studio 2022 is a 64-bit process and can't load 32-bit components, regardless of the underlying technology, such as .NET Framework, .NET, or COM\ActiveX. You might not realize you have references to 32-bit components until you try to upgrade Visual Studio. References that compile to 64-bit or target `AnyCPU` continue to work. You'll also run into the same problem if a component you're referencing compiles to `AnyCPU` but happens to reference something 32-bit.

What's the problem

Windows Forms code runs in two modes: design-time and run-time. At run-time, you're running in whatever mode you compiled for: 32-bit or 64-bit; .NET Framework or .NET. At design-time, your code is run inside Visual Studio, which is a 64-bit .NET Framework process. If your project code doesn't match that environment, it can't run in the designer. For example, if your project is targeting 32-bit .NET Framework or 64-bit .NET, it doesn't match Visual Studio's 64-bit .NET Framework process. And that's the problem: the Windows Forms designer in Visual Studio can't instantiate 32-bit components or .NET components directly, it can only instantiate 64-bit .NET Framework components. To fix these integration issues, the Windows Forms team created the **out-of-process designer** for Visual Studio that acts like a translation layer for the Windows Forms designer. The out-of-process designer communicates with the Visual Studio 64-bit .NET Framework designer on behalf of your code so that you can use the designer with .NET projects.

Previous versions of Visual Studio targeted 32-bit, and your project probably compiled to `AnyCPU`, which would pick 32-bit while in design mode to match Visual Studio. 32-bit specific references worked, but if you had a 64-bit specific reference, you might have run into a problem with the designer. With Visual Studio 2022, the problem reversed. Visual Studio 2022 is only available in 64-bit. Components and libraries that were compiled as `AnyCPU` work in both 32-bit and 64-bit and don't have an issue running in Visual Studio 2022 64-bit. But, after upgrading to Visual Studio 2022, your projects might fail to run at design-time if the project relies on a 32-bit specific component. This

is even the case when your referenced component is compiled for `AnyCPU`, but happens to reference a 32-bit component or 32-bit COM\ActiveX library directly.

To summarize, 32-bit components can't be used by the Windows Forms designer in Visual Studio 2022, which is a 64-bit app. The **out-of-process designer** was created to help Windows Forms for .NET apps during design-time for both 32-bit and 64-bit. This designer now helps with loading 32-bit and 64-bit .NET Framework components.

What can you do

There are a few design changes you should consider, which might help your project.

- Upgrade from .NET Framework to .NET 8+.

  .NET uses the out-of-process designer, which helps with 32-bit designer problems.

- With .NET Framework, set your app to target `AnyCPU`.

  If you target `AnyCPU` and enable `Prefer 32-bit`, your app runs under 64-bit when in Visual Studio design-time, but compiles to 32-bit for run-time.

- Recompile the 32-bit component for `AnyCPU` or 64-bit.

  If you have access to the source code for the 32-bit component, try compiling it for `AnyCPU` or 64-bit and reference that new version.

- Find a 64-bit alternative component.

  If you're using a component owned by someone else, check to see if they offer a 64-bit version, and reference that.

- Try the out-of-process designer.

  Your final option would be to enable the out-of-process designer for .NET Framework.

Out-of-process designer

If your project targets .NET, you're already using the out-of-process designer. However, if you're still using .NET Framework, you need to enable the out-of-process designer.

> ⚠ **Warning**

The updated out-of-process 32-Bit .NET Framework Designer doesn't achieve full parity with the old in-process .NET Framework Designer due to the same architectural differences. Highly customized control designers aren't compatible. If you use custom control libraries from 3rd parties, check if they offer versions that support the out-of-process .NET Framework Designer.

The **out-of-process designer**, with some limitations, handles 32-bit issues with Visual Studio 2022:

- .NET Framework benefits from improved type resolution.
- ActiveX and COM references are supported in both .NET Framework and .NET.
- The in-process designer in Visual Studio detects 32-bit assembly load failures and can suggest enabling the out-of-process designer.

Use the out-of-process designer

Support for 32-bit references requires **Visual Studio 17.9 or later**. Enable it by adding the following `<PropertyGroup>` setting to your project file:

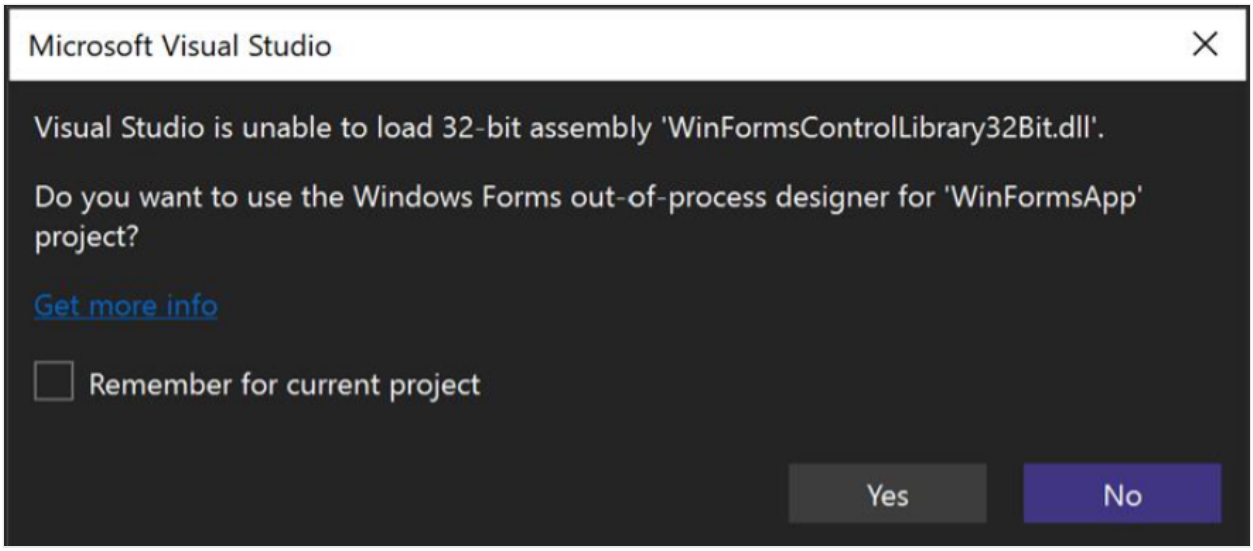
XML

```
<PropertyGroup>
    <UseWinFormsOutOfProcDesigner>True</UseWinFormsOutOfProcDesigner>
</PropertyGroup>
```

After modifying the project file, reload the project.

32-bit issue detection

Currently, when Visual Studio detects that a 32-bit reference fails to load, it prompts you to enable the Windows Forms out-of-process designer. If you agree to enable it, the project is updated for you and then reloaded.



This detection feature is controlled in the Visual Studio menu, under **Tools** > **Options** > **Preview Features**.

See also

- .NET Blog – WinForms Designer Selection for 32-bit .NET Framework Projects

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Use Reset and ShouldSerialize to control a property (Windows Forms .NET)

Article • 04/19/2024

In this article, you learn how to create the `Reset<PropertyName>` and `ShouldSerialize<PropertyName>` methods to manage a property for the **Properties** window in Visual Studio. `Reset` and `ShouldSerialize` are optional methods that you can provide for a property, if the property doesn't have a simple default value. If the property has a simple default value, you should apply the DefaultValueAttribute and supply the default value to the attribute class constructor instead. Either of these mechanisms enables the following features in the designer:

- The property provides visual indication in the property browser if it has been modified from its default value.
- The user can right-click on the property and choose **Reset** to restore the property to its default value.
- The designer generates more efficient code.

For more information about properties, see Reset and ShouldSerialize.

Supporting code

This article demonstrates the `Reset` and `ShouldSerialize` methods by creating a compass rose control. If you're working with your own user control, you can skip this section.

1. Add the following enumeration to your code:

```
C#

public enum Directions
{
    None,
    North,
    NorthEast,
    East,
    SouthEast,
    South,
    SouthWest,
    West,
    NorthWest,
}
```

2. Add a user control named `CompassRose`.

3. Add a property named `Direction` of type `Directions` to the user control.

```
C#

public Directions Direction { get; set; } = Directions.None;
```

Reset

The `Reset<PropertyName>` method resets the corresponding `<PropertyName>` property to its default value.

The following code resets the `Direction` property to `None`, which is considered the default value for the compass rose control:

```
C#

private void ResetDirection() =>
    Direction = Directions.None;
```

ShouldSerialize

The `ShouldSerialize<PropertyName>` method returns a boolean value that indicates whether or not the backing property has changed from its default value and should be serialized into the designer's code.

The following code returns true when the `Direction` property doesn't equal `None`, indicating that a direction has been chosen:

```
C#

private bool ShouldSerializeDirection() =>
    Direction != Directions.None;
```

Example

The following code shows the `Reset` and `ShouldSerialize` methods for the `Direction` property:

```
C#
```

```
public partial class CompassRose : UserControl
{
    public Directions Direction { get; set; } = Directions.None;

    public CompassRose() =>
        InitializeComponent();

    private void ResetDirection() =>
        Direction = Directions.None;

    private bool ShouldSerializeDirection() =>
        Direction != Directions.None;
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Set the icon of a control in the Toolbox (Windows Forms .NET)

Article • 04/19/2024

Controls that you create always receive a generic icon for the **Toolbox** window in Visual Studio. However, when you change the icon, it adds a sense of professionalism to your control, and makes it stand out in the toolbox. This article teaches you how to set the icon for your control.

Bitmap icon

Icons for the **Toolbox** window in Visual Studio must conform to certain standards, otherwise they're ignored or are displayed incorrectly.

- **Size**: Icons for a control must be a 16x16 bitmap image.
- **File type**: The icon can be either a Bitmap (*.bmp*) or a Windows Icon (*.ico*) file.
- **Transparency**: The magenta color (RGB: `255,0,255`, Hex: `0xFF00FF`) is rendered transparent.
- **Themes**: Visual Studio has multiple themes, but each theme is considered either dark or light. Your icon should be designed for the light theme. When Visual Studio uses a dark theme, the dark and light colors in the icon are automatically inverted.

How to assign an icon

Icons are assigned to a control with the ToolboxBitmapAttribute attribute. For more information about attributes, see Attributes (C#) or Attributes overview (Visual Basic).

> **Tip**
>
> You can download a sample icon from **GitHub**.

The attribute is set on the control's class, and has three different constructors:

- ToolboxBitmapAttribute(Type)—This constructor takes a single type reference, and from that type, tries to find an embedded resource to use as the icon.

  The type's FullName value is used to try to find a corresponding embedded resource based on the name of the type. For example, if the `MyProject.MyNamespace.CompassRose` type is referenced, the attribute looks for a

resource named `MyProject.MyNamespace.CompassRose.bmp` or `MyProject.MyNamespace.CompassRose.ico`. If the resource is found, it's used as the control's icon.

C#

```
// Looks for a CompassRose.bmp or CompassRose.ico embedded resource in
the
// same namespace as the CompassRose type.
[ToolboxBitmap(typeof(CompassRose))]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

- ToolboxBitmapAttribute(Type, String)—This constructor takes two parameters. The first parameter is a type, and the second is the namespace and name of the resource in the assembly of that type.

C#

```
// Loads the icon from the WinFormsApp1.Resources.CompassRoseIcon.bmp
resource
// in the assembly containing the type CompassRose
[ToolboxBitmap(typeof(CompassRose),
"WinFormsApp1.Resources.CompassRoseIcon.bmp")]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

- ToolboxBitmapAttribute(String)—This constructor takes a single string parameter, the absolute path to the icon file.

C#

```
// Loads the icon from a file on disk
[ToolboxBitmap(@"C:\Files\Resources\MyIcon.bmp")]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Overview of using the keyboard (Windows Forms .NET)

Article • 07/06/2022

In Windows Forms, user input is sent to applications in the form of Windows messages. A series of overridable methods process these messages at the application, form, and control level. When these methods receive keyboard messages, they raise events that can be handled to get information about the keyboard input. In many cases, Windows Forms applications will be able to process all user input simply by handling these events. In other cases, an application may need to override one of the methods that process messages in order to intercept a particular message before it is received by the application, form, or control.

Keyboard events

All Windows Forms controls inherit a set of events related to mouse and keyboard input. For example, a control can handle the KeyPress event to determine the character code of a key that was pressed. For more information, see Using keyboard events.

Methods that process user input messages

Forms and controls have access to the IMessageFilter interface and a set of overridable methods that process Windows messages at different points in the message queue. These methods all have a Message parameter, which encapsulates the low-level details of Windows messages. You can implement or override these methods to examine the message and then either consume the message or pass it on to the next consumer in the message queue. The following table presents the methods that process all Windows messages in Windows Forms.

Expand table

Method	Notes
PreFilterMessage	This method intercepts queued (also known as posted) Windows messages at the application level.
PreProcessMessage	This method intercepts Windows messages at the form and control level before they have been processed.
WndProc	This method processes Windows messages at the form and control level.

Method	Notes
DefWndProc	This method performs the default processing of Windows messages at the form and control level. This provides the minimal functionality of a window.
OnNotifyMessage	This method intercepts messages at the form and control level, after they have been processed. The EnableNotifyMessage style bit must be set for this method to be called.

Keyboard and mouse messages are also processed by an additional set of overridable methods that are specific to those types of messages. For more information, see the Preprocessing keys section. .

Types of keys

Windows Forms identifies keyboard input as virtual-key codes that are represented by the bitwise Keys enumeration. With the Keys enumeration, you can combine a series of pressed keys to result in a single value. These values correspond to the values that accompany the **WM_KEYDOWN** and **WM_SYSKEYDOWN** Windows messages. You can detect most physical key presses by handling the KeyDown or KeyUp events. Character keys are a subset of the Keys enumeration and correspond to the values that accompany the **WM_CHAR** and **WM_SYSCHAR** Windows messages. If the combination of pressed keys results in a character, you can detect the character by handling the KeyPress event.

Order of keyboard events

As listed previously, there are 3 keyboard related events that can occur on a control. The following sequence shows the general order of the events:

1. The user pushes the "a" key, the key is preprocessed, dispatched, and a KeyDown event occurs.
2. The user holds the "a" key, the key is preprocessed, dispatched, and a KeyPress event occurs. This event occurs multiple times as the user holds a key.
3. The user releases the "a" key, the key is preprocessed, dispatched and a KeyUp event occurs.

Preprocessing keys

Like other messages, keyboard messages are processed in the WndProc method of a form or control. However, before keyboard messages are processed, the

PreProcessMessage method calls one or more methods that can be overridden to handle special character keys and physical keys. You can override these methods to detect and filter certain keys before the messages are processed by the control. The following table shows the action that is being performed and the related method that occurs, in the order that the method occurs.

Preprocessing for a KeyDown event

Expand table

Action	Related method	Notes
Check for a command key such as an accelerator or menu shortcut.	ProcessCmdKey	This method processes a command key, which takes precedence over regular keys. If this method returns `true`, the key message is not dispatched and a key event does not occur. If it returns `false`, IsInputKey is called.
Check for a special key that requires preprocessing or a normal character key that should raise a KeyDown event and be dispatched to a control.	IsInputKey	If the method returns `true`, it means the control is a regular character and a KeyDown event is raised. If `false`, ProcessDialogKey is called. **Note:** To ensure a control gets a key or combination of keys, you can handle the PreviewKeyDown event and set IsInputKey of the PreviewKeyDownEventArgs to `true` for the key or keys you want.
Check for a navigation key (ESC, TAB, Return, or arrow keys).	ProcessDialogKey	This method processes a physical key that employs special functionality within the control, such as switching focus between the control and its parent. If the immediate control does not handle the key, the ProcessDialogKey is called on the parent control and so on to the topmost control in the hierarchy. If this method returns `true`, preprocessing is complete and a key event is not generated. If it returns `false`, a KeyDown event occurs.

Preprocessing for a KeyPress event

Expand table

Action	Related method	Notes
Check to see the key is a normal character that	IsInputChar	If the character is a normal character, this method returns `true`, the KeyPress event is raised and no

Action	Related method	Notes
should be processed by the control		further preprocessing occurs. Otherwise ProcessDialogChar will be called.
Check to see if the character is a mnemonic (such as &OK on a button)	ProcessDialogChar	This method, similar to ProcessDialogKey, will be called up the control hierarchy. If the control is a container control, it checks for mnemonics by calling ProcessMnemonic on itself and its child controls. If ProcessDialogChar returns `true`, a KeyPress event does not occur.

Processing keyboard messages

After keyboard messages reach the WndProc method of a form or control, they are processed by a set of methods that can be overridden. Each of these methods returns a Boolean value specifying whether the keyboard message has been processed and consumed by the control. If one of the methods returns `true`, then the message is considered handled, and it is not passed to the control's base or parent for further processing. Otherwise, the message stays in the message queue and may be processed in another method in the control's base or parent. The following table presents the methods that process keyboard messages.

Expand table

Method	Notes
ProcessKeyMessage	This method processes all keyboard messages that are received by the WndProc method of the control.
ProcessKeyPreview	This method sends the keyboard message to the control's parent. If ProcessKeyPreview returns `true`, no key event is generated, otherwise ProcessKeyEventArgs is called.
ProcessKeyEventArgs	This method raises the KeyDown, KeyPress, and KeyUp events, as appropriate.

Overriding keyboard methods

There are many methods available for overriding when a keyboard message is preprocessed and processed; however, some methods are much better choices than others. Following table shows tasks you might want to accomplish and the best way to override the keyboard methods. For more information on overriding methods, see Inheritance (C# Programming Guide) or Inheritance (Visual Basic)

Expand table

Task	Method
Intercept a navigation key and raise a KeyDown event. For example you want TAB and Return to be handled in a text box.	Override IsInputKey. **Note:** Alternatively, you can handle the PreviewKeyDown event and set IsInputKey of the PreviewKeyDownEventArgs to `true` for the key or keys you want.
Perform special input or navigation handling on a control. For example, you want the use of arrow keys in your list control to change the selected item.	Override ProcessDialogKey
Intercept a navigation key and raise a KeyPress event. For example in a spin-box control you want multiple arrow key presses to accelerate progression through the items.	Override IsInputChar.
Perform special input or navigation handling during a KeyPress event. For example, in a list control holding down the "r" key skips between items that begin with the letter r.	Override ProcessDialogChar
Perform custom mnemonic handling; for example, you want to handle mnemonics on owner-drawn buttons contained in a toolbar.	Override ProcessMnemonic.

See also

- Keys
- WndProc
- PreProcessMessage
- Using keyboard events (Windows Forms .NET)
- How to modify keyboard key events (Windows Forms .NET)
- How to Check for modifier key presses (Windows Forms .NET)
- How to simulate keyboard events (Windows Forms .NET)
- How to handle keyboard input messages in the form (Windows Forms .NET)
- Add a control (Windows Forms .NET)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Using keyboard events (Windows Forms .NET)

Article • 04/19/2024

Most Windows Forms programs process keyboard input by handling the keyboard events. This article provides an overview of the keyboard events, including details on when to use each event and the data that is supplied for each event. For more information about events in general, see Events overview (Windows Forms .NET).

Keyboard events

Windows Forms provides two events that occur when a user presses a keyboard key and one event when a user releases a keyboard key:

- The KeyDown event occurs once.
- The KeyPress event, which can occur multiple times when a user holds down the same key.
- The KeyUp event occurs once when a user releases a key.

When a user presses a key, Windows Forms determines which event to raise based on whether the keyboard message specifies a character key or a physical key. For more information about character and physical keys, see Keyboard overview, keyboard events.

The following table describes the three keyboard events.

Expand table

Keyboard event	Description	Results
KeyDown	This event is raised when a user presses a physical key.	The handler for KeyDown receives: • A KeyEventArgs parameter, which provides the KeyCode property (which specifies a physical keyboard button). • The Modifiers property (SHIFT, CTRL, or ALT). • The KeyData property (which combines the key code and modifier). The KeyEventArgs parameter also provides:

Keyboard event	Description	Results
		◦ The Handled property, which can be set to prevent the underlying control from receiving the key. ◦ The SuppressKeyPress property, which can be used to suppress the KeyPress and KeyUp events for that keystroke.
KeyPress	This event is raised when the key or keys pressed result in a character. For example, a user presses SHIFT and the lowercase "a" keys, which result in a capital letter "A" character.	KeyPress is raised after KeyDown. • The handler for KeyPress receives: • A KeyPressEventArgs parameter, which contains the character code of the key that was pressed. This character code is unique for every combination of a character key and a modifier key. For example, the "A" key will generate: ◦ The character code 65, if it is pressed with the SHIFT key ◦ Or the CAPS LOCK key, 97 if it is pressed by itself, ◦ And 1, if it is pressed with the CTRL key.
KeyUp	This event is raised when a user releases a physical key.	The handler for KeyUp receives: • A KeyEventArgs parameter: ◦ Which provides the KeyCode property (which specifies a physical keyboard button). ◦ The Modifiers property (SHIFT, CTRL, or ALT). ◦ The KeyData property (which combines the key code and modifier).

See also

- Overview of using the keyboard (Windows Forms .NET)
- How to modify keyboard key events (Windows Forms .NET)

- How to Check for modifier key presses (Windows Forms .NET)
- How to simulate keyboard events (Windows Forms .NET)
- How to handle keyboard input messages in the form (Windows Forms .NET)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Overview of how to validate user input (Windows Forms .NET)

Article • 10/28/2020

When users enter data into your application, you may want to verify that the data is valid before your application uses it. You may require that certain text fields not be zero-length, that a field formatted as a telephone number, or that a string doesn't contain invalid characters. Windows Forms provides several ways for you to validate input in your application.

MaskedTextBox Control

If you need to require users to enter data in a well-defined format, such as a telephone number or a part number, you can accomplish this quickly and with minimal code by using the MaskedTextBox control. A *mask* is a string made up of characters from a masking language that specifies which characters can be entered at any given position in the text box. The control displays a set of prompts to the user. If the user types an incorrect entry, for example, the user types a letter when a digit is required, the control will automatically reject the input.

The masking language that is used by MaskedTextBox is flexible. It allows you to specify required characters, optional characters, literal characters, such as hyphens and parentheses, currency characters, and date separators. The control also works well when bound to a data source. The Format event on a data binding can be used to reformat incoming data to comply with the mask, and the Parse event can be used to reformat outgoing data to comply with the specifications of the data field.

Event-driven validation

If you want full programmatic control over validation, or need complex validation checks, you should use the validation events that are built into most Windows Forms controls. Each control that accepts free-form user input has a Validating event that will occur whenever the control requires data validation. In the Validating event-handling method, you can validate user input in several ways. For example, if you have a text box that must contain a postal code, you can do the validation in the following ways:

- If the postal code must belong to a specific group of zip codes, you can do a string comparison on the input to validate the data entered by the user. For example, if

the postal code must be in the set `{10001, 10002, 10003}`, then you can use a string comparison to validate the data.

- If the postal code must be in a specific form, you can use regular expressions to validate the data entered by the user. For example, to validate the form `#####` or `#####-####`, you can use the regular expression `^(\d{5})(-\d{4})?$`. To validate the form `A#A #A#`, you can use the regular expression `[A-Z]\d[A-Z] \d[A-Z]\d`. For more information about regular expressions, see .NET Regular Expressions and Regular Expression Examples.
- If the postal code must be a valid United States Zip code, you could call a Zip code Web service to validate the data entered by the user.

The Validating event is supplied an object of type CancelEventArgs. If you determine that the control's data isn't valid, cancel the Validating event by setting this object's Cancel property to `true`. If you don't set the Cancel property, Windows Forms will assume that validation succeeded for that control and raise the Validated event.

For a code example that validates an email address in a TextBox, see the Validating event reference.

Event-driven validation data-bound controls

Validation is useful when you have bound your controls to a data source, such as a database table. By using validation, you can make sure that your control's data satisfies the format required by the data source, and that it doesn't contain any special characters such as quotation marks and back slashes that might be unsafe.

When you use data binding, the data in your control is synchronized with the data source during execution of the Validating event. If you cancel the Validating event, the data won't be synchronized with the data source.

> **ⓘ Important**
>
> If you have custom validation that takes place after the **Validating** event, it won't affect the data binding. For example, if you have code in a **Validated** event that attempts to cancel the data binding, the data binding will still occur. In this case, to perform validation in the **Validated** event, change the control's **Binding.DataSourceUpdateMode** property from **DataSourceUpdateMode.OnValidation** to **DataSourceUpdateMode.Never**, and

add `your-control.DataBindings["field-name"].WriteValue()` to your validation code.

Implicit and explicit validation

So when does a control's data get validated? This is up to you, the developer. You can use either implicit or explicit validation, depending on the needs of your application.

Implicit validation

The implicit validation approach validates data as the user enters it. Validate the data by reading the keys as they're pressed, or more commonly whenever the user takes the input focus away from the control. This approach is useful when you want to give the user immediate feedback about the data as they're working.

If you want to use implicit validation for a control, you must set that control's AutoValidate property to EnablePreventFocusChange or EnableAllowFocusChange. If you cancel the Validating event, the behavior of the control will be determined by what value you assigned to AutoValidate. If you assigned EnablePreventFocusChange, canceling the event will cause the Validated event not to occur. Input focus will remain on the current control until the user changes the data to a valid format. If you assigned EnableAllowFocusChange, the Validated event won't occur when you cancel the event, but focus will still change to the next control.

Assigning Disable to the AutoValidate property prevents implicit validation altogether. To validate your controls, you'll have to use explicit validation.

Explicit validation

The explicit validation approach validates data at one time. You can validate the data in response to a user action, such as clicking a **Save** button or a **Next** link. When the user action occurs, you can trigger explicit validation in one of the following ways:

- Call Validate to validate the last control to have lost focus.
- Call ValidateChildren to validate all child controls in a form or container control.
- Call a custom method to validate the data in the controls manually.

Default implicit validation behavior for controls

Different Windows Forms controls have different defaults for their AutoValidate property. The following table shows the most common controls and their defaults.

Expand table

Control	Default Validation Behavior
ContainerControl	Inherit
Form	EnableAllowFocusChange
PropertyGrid	Property not exposed in Visual Studio
ToolStripContainer	Property not exposed in Visual Studio
SplitContainer	Inherit
UserControl	EnableAllowFocusChange

Closing the form and overriding Validation

When a control maintains focus because the data it contains is invalid, it's impossible to close the parent form in one of the usual ways:

- By clicking the **Close** button.
- By selecting the **System** > **Close** menu.
- By calling the Close method programmatically.

However, in some cases, you might want to let the user close the form regardless of whether the values in the controls are valid. You can override validation and close a form that still contains invalid data by creating a handler for the form's FormClosing event. In the event, set the Cancel property to `false`. This forces the form to close. For more information and an example, see Form.FormClosing.

> **Note**
>
> If you force the form to close in this manner, any data in the form's controls that has not already been saved is lost. In addition, modal forms don't validate the contents of controls when they're closed. You can still use control validation to lock focus to a control, but you don't have to be concerned about the behavior associated with closing the form.

See also

- Using keyboard events (Windows Forms .NET)
- Control.Validating
- Form.FormClosing
- System.Windows.Forms.FormClosingEventArgs

How to modify keyboard key events (Windows Forms .NET)

Article • 04/19/2024

Windows Forms provides the ability to consume and modify keyboard input. Consuming a key refers to handling a key within a method or event handler so that other methods and events further down the message queue don't receive the key value. And, modifying a key refers to modifying the value of a key so that methods and event handlers further down the message queue receive a different key value. This article shows how to accomplish these tasks.

Consume a key

In a KeyPress event handler, set the Handled property of the KeyPressEventArgs class to `true`.

-or-

In a KeyDown event handler, set the Handled property of the KeyEventArgs class to `true`.

> **Note**
>
> Setting the Handled property in the KeyDown event handler does not prevent the KeyPress and KeyUp events from being raised for the current keystroke. Use the SuppressKeyPress property for this purpose.

The following example handles the KeyPress event to consume the `A` and `a` character keys. Those keys can't be typed into the text box:

C#

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
        e.Handled = true;
}
```

Modify a standard character key

In a KeyPress event handler, set the KeyChar property of the KeyPressEventArgs class to the value of the new character key.

The following example handles the KeyPress event to change any `A` and `a` character keys to `!`:

C#

```
private void textBox2_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
    {
        e.KeyChar = '!';
        e.Handled = false;
    }
}
```

Modify a non-character key

You can only modify non-character key presses by inheriting from the control and overriding the PreProcessMessage method. As the input Message is sent to the control, it's processed before the control raising events. You can intercept these messages to modify or block them.

The following code example demonstrates how to use the WParam property of the Message parameter to change the key pressed. This code detects a key from `F1` through `F10` and translates the key into a numeric key ranging from `0` through `9` (where `F10` maps to `0`).

C#

```
public override bool PreProcessMessage(ref Message m)
{
    const int WM_KEYDOWN = 0x100;

    if (m.Msg == WM_KEYDOWN)
    {
        Keys keyCode = (Keys)m.WParam & Keys.KeyCode;

        // Detect F1 through F9.
        m.WParam = keyCode switch
        {
            Keys.F1 => (IntPtr)Keys.D1,
            Keys.F2 => (IntPtr)Keys.D2,
            Keys.F3 => (IntPtr)Keys.D3,
            Keys.F4 => (IntPtr)Keys.D4,
            Keys.F5 => (IntPtr)Keys.D5,
```

```
            Keys.F6 => (IntPtr)Keys.D6,
            Keys.F7 => (IntPtr)Keys.D7,
            Keys.F8 => (IntPtr)Keys.D8,
            Keys.F9 => (IntPtr)Keys.D9,
            Keys.F10 => (IntPtr)Keys.D0,
            _ => m.WParam
        };
    }

    // Send all other messages to the base method.
    return base.PreProcessMessage(ref m);
}
```

See also

- Overview of using the keyboard (Windows Forms .NET)
- Using keyboard events (Windows Forms .NET)
- Keys
- KeyDown
- KeyPress

How to check for modifier key presses (Windows Forms .NET)

Article • 04/19/2024

As the user types keys into your application, you can monitor for pressed modifier keys such as the `SHIFT`, `ALT`, and `CTRL`. When a modifier key is pressed in combination with other keys or even a mouse click, your application can respond appropriately. For example, pressing the `S` key may cause an "s" to appear on the screen. If the keys `CTRL+S` are pressed, instead, the current document may be saved.

If you handle the KeyDown event, the KeyEventArgs.Modifiers property received by the event handler specifies which modifier keys are pressed. Also, the KeyEventArgs.KeyData property specifies the character that was pressed along with any modifier keys combined with a bitwise OR.

If you're handling the KeyPress event or a mouse event, the event handler doesn't receive this information. Use the ModifierKeys property of the Control class to detect a key modifier. In either case, you must perform a bitwise AND of the appropriate Keys value and the value you're testing. The Keys enumeration offers variations of each modifier key, so it's important that you do the bitwise AND check with the correct value.

For example, the `SHIFT` key is represented by the following key values:

- Keys.Shift
- Keys.ShiftKey
- Keys.RShiftKey
- Keys.LShiftKey

The correct value to test `SHIFT` as a modifier key is Keys.Shift. Similarly, to test for `CTRL` and `ALT` as modifiers you should use the Keys.Control and Keys.Alt values, respectively.

Detect modifier key

Detect if a modifier key is pressed by comparing the ModifierKeys property and the Keys enumeration value with a bitwise AND operator.

The following code example shows how to determine whether the `SHIFT` key is pressed within the KeyPress and KeyDown event handlers.

```
C#
```

```
// Event only raised when non-modifier key is pressed
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((Control.ModifierKeys & Keys.Shift) == Keys.Shift)
        MessageBox.Show("KeyPress " + Keys.Shift);
}

// Event raised as soon as shift is pressed
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    if ((Control.ModifierKeys & Keys.Shift) == Keys.Shift)
        MessageBox.Show("KeyDown " + Keys.Shift);
}
```

See also

- Overview of using the keyboard (Windows Forms .NET)
- Using keyboard events (Windows Forms .NET)
- Keys
- ModifierKeys
- KeyDown
- KeyPress

How to handle keyboard input messages in the form (Windows Forms .NET)

Article • 04/19/2024

Windows Forms provides the ability to handle keyboard messages at the form level, before the messages reach a control. This article shows how to accomplish this task.

Handle a keyboard message

Handle the KeyPress or KeyDown event of the active form and set the KeyPreview property of the form to `true`. This property causes the keyboard to be received by the form before they reach any controls on the form. The following code example handles the KeyPress event by detecting all of the number keys and consuming `1`, `4`, and `7`.

C#

```
// Detect all numeric characters at the form level and consume 1,4, and 7.
// Form.KeyPreview must be set to true for this event handler to be called.
void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar >= 48 && e.KeyChar <= 57)
    {
        MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' pressed.");

        switch (e.KeyChar)
        {
            case (char)49:
            case (char)52:
            case (char)55:
                MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' consumed.");
                e.Handled = true;
                break;
        }
    }
}
```

See also

- Overview of using the keyboard (Windows Forms .NET)
- Using keyboard events (Windows Forms .NET)
- Keys

- ModifierKeys
- KeyDown
- KeyPress

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

How to simulate keyboard events (Windows Forms .NET)

Article • 10/28/2020

Windows Forms provides a few options for programmatically simulating keyboard input. This article provides an overview of these options.

Use SendKeys

Windows Forms provides the System.Windows.Forms.SendKeys class for sending keystrokes to the active application. There are two methods to send keystrokes to an application: SendKeys.Send and SendKeys.SendWait. The difference between the two methods is that `SendWait` blocks the current thread when the keystroke is sent, waiting for a response, while `Send` doesn't. For more information about `SendWait`, see To send a keystroke to a different application.

> **⊗ Caution**
>
> If your application is intended for international use with a variety of keyboards, the use of **SendKeys.Send** could yield unpredictable results and should be avoided.

Behind the scenes, `SendKeys` uses an older Windows implementation for sending input, which may fail on modern Windows where it's expected that the application isn't running with administrative rights. If the older implementation fails, the code automatically tries the newer Windows implementation for sending input. Additionally, when the SendKeys class uses the new implementation, the SendWait method no longer blocks the current thread when sending keystrokes to another application.

> **ⓘ Important**
>
> If your application relies on consistent behavior regardless of the operating system, you can force the **SendKeys** class to use the new implementation by adding the following application setting to your app.config file.

XML

```
<appSettings>
  <add key="SendKeys" value="SendInput"/>
</appSettings>
```

To force the **SendKeys** class to *only* use the previous implementation, use the value `"JournalHook"` instead.

To send a keystroke to the same application

Call the SendKeys.Send or SendKeys.SendWait method of the SendKeys class. The specified keystrokes will be received by the active control of the application.

The following code example uses `Send` to simulate pressing the `ALT` and `DOWN` keys together. These keystrokes cause the ComboBox control to display its dropdown. This example assumes a Form with a Button and ComboBox.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    comboBox1.Focus();
    SendKeys.Send("%+{DOWN}");
}
```

To send a keystroke to a different application

The SendKeys.Send and SendKeys.SendWait methods send keystrokes to the active application, which is usually the application you're sending keystrokes from. To send keystrokes to another application, you first need to activate it. Because there's no managed method to activate another application, you must use native Windows methods to focus the other application. The following code example uses platform invoke to call the `FindWindow` and `SetForegroundWindow` methods to activate the Calculator application window, and then calls `Send` to issue a series of calculations to the Calculator application.

The following code example uses `Send` to simulate pressing keys into the Windows 10 Calculator application. It first searches for an application window with title of `Calculator` and then activates it. Once activated, the keystrokes are sent to calculate 10 plus 10.

C#

```
[DllImport("USER32.DLL", CharSet = CharSet.Unicode)]
public static extern IntPtr FindWindow(string lpClassName, string
lpWindowName);

[DllImport("USER32.DLL")]
public static extern bool SetForegroundWindow(IntPtr hWnd);
```

```
private void button1_Click(object sender, EventArgs e)
{
    IntPtr calcWindow = FindWindow(null, "Calculator");

    if (SetForegroundWindow(calcWindow))
        SendKeys.Send("10{+}10=");
}
```

Use OnEventName methods

The easiest way to simulate keyboard events is to call a method on the object that raises the event. Most events have a corresponding method that invokes them, named in the pattern of `On` followed by `EventName`, such as `OnKeyPress`. This option is only possible within custom controls or forms, because these methods are protected and can't be accessed from outside the context of the control or form.

These protected methods are available to simulate keyboard events.

- `OnKeyDown`
- `OnKeyPress`
- `OnKeyUp`

For more information about these events, see Using keyboard events (Windows Forms .NET).

See also

- Overview of using the keyboard (Windows Forms .NET)
- Using keyboard events (Windows Forms .NET)
- Using mouse events (Windows Forms .NET)
- SendKeys
- Keys
- KeyDown
- KeyPress

can also create and review issues and pull requests. For more information, see our contributor guide.

provide feedback:

- Open a documentation issue
- Provide product feedback

Overview of using the mouse (Windows Forms .NET)

Article • 10/28/2020

Receiving and handling mouse input is an important part of every Windows application. You can handle mouse events to carry out an action in your application, or use mouse location information to perform hit testing or other actions. Also, you can change the way the controls in your application handle mouse input. This article describes these mouse events in detail, and how to obtain and change system settings for the mouse.

In Windows Forms, user input is sent to applications in the form of Windows messages. A series of overridable methods process these messages at the application, form, and control level. When these methods receive mouse messages, they raise events that can be handled to get information about the mouse input. In many cases, Windows Forms applications can process all user input simply by handling these events. In other cases, an application may override one of the methods that process messages to intercept a particular message before it's received by the application, form, or control.

Mouse Events

All Windows Forms controls inherit a set of events related to mouse and keyboard input. For example, a control can handle the MouseClick event to determine the location of a mouse click. For more information on the mouse events, see Using mouse events.

Mouse location and hit-testing

When the user moves the mouse, the operating system moves the mouse pointer. The mouse pointer contains a single pixel, called the hot spot, which the operating system tracks and recognizes as the position of the pointer. When the user moves the mouse or presses a mouse button, the Control that contains the HotSpot raises the appropriate mouse event.

You can obtain the current mouse position with the Location property of the MouseEventArgs when handling a mouse event or by using the Position property of the Cursor class. You can then use mouse location information to carry out hit-testing, and then perform an action based on the location of the mouse. Hit-testing capability is built in to several controls in Windows Forms such as the ListView, TreeView, MonthCalendar and DataGridView controls.

Used with the appropriate mouse event, MouseHover for example, hit-testing is very useful for determining when your application should do a specific action.

Changing mouse input settings

You can detect and change the way a control handles mouse input by deriving from the control and using the GetStyle and SetStyle methods. The SetStyle method takes a bitwise combination of ControlStyles values to determine whether the control will have standard click, double-click behavior, or if the control will handle its own mouse processing. Also, the SystemInformation class includes properties that describe the capabilities of the mouse and specify how the mouse interacts with the operating system. The following table summarizes these properties.

Expand table

Property	Description
DoubleClickSize	Gets the dimensions, in pixels, of the area in which the user must click twice for the operating system to consider the two clicks a double-click.
DoubleClickTime	Gets the maximum number of milliseconds that can elapse between a first click and a second click for the mouse action to be considered a double-click.
MouseButtons	Gets the number of buttons on the mouse.
MouseButtonsSwapped	Gets a value indicating whether the functions of the left and right mouse buttons have been swapped.
MouseHoverSize	Gets the dimensions, in pixels, of the rectangle within which the mouse pointer has to stay for the mouse hover time before a mouse hover message is generated.
MouseHoverTime	Gets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle before a mouse hover message is generated.
MousePresent	Gets a value indicating whether a mouse is installed.
MouseSpeed	Gets a value indicating the current mouse speed, from 1 to 20.
MouseWheelPresent	Gets a value indicating whether a mouse with a mouse wheel is installed.
MouseWheelScrollDelta	Gets the amount of the delta value of the increment of a single mouse wheel rotation.
MouseWheelScrollLines	Gets the number of lines to scroll when the mouse wheel is rotated.

Methods that process user input messages

Forms and controls have access to the IMessageFilter interface and a set of overridable methods that process Windows messages at different points in the message queue. These methods all have a Message parameter, which encapsulates the low-level details of Windows messages. You can implement or override these methods to examine the message and then either consume the message or pass it on to the next consumer in the message queue. The following table presents the methods that process all Windows messages in Windows Forms.

Expand table

Method	Notes
PreFilterMessage	This method intercepts queued (also known as posted) Windows messages at the application level.
PreProcessMessage	This method intercepts Windows messages at the form and control level before they have been processed.
WndProc	This method processes Windows messages at the form and control level.
DefWndProc	This method performs the default processing of Windows messages at the form and control level. This provides the minimal functionality of a window.
OnNotifyMessage	This method intercepts messages at the form and control level, after they've been processed. The EnableNotifyMessage style bit must be set for this method to be called.

See also

- Using mouse events (Windows Forms .NET)
- Drag-and-drop mouse behavior overview (Windows Forms .NET)
- Manage mouse pointers (Windows Forms .NET)

Provide product feedback

Using mouse events (Windows Forms .NET)

Article • 10/28/2020

Most Windows Forms programs process mouse input by handling the mouse events. This article provides an overview of the mouse events, including details on when to use each event and the data that is supplied for each event. For more information about events in general, see Events overview (Windows Forms .NET).

Mouse events

The primary way to respond to mouse input is to handle mouse events. The following table shows the mouse events and describes when they're raised.

Expand table

Mouse Event	Description
Click	This event occurs when the mouse button is released, typically before the MouseUp event. The handler for this event receives an argument of type EventArgs. Handle this event when you only need to determine when a click occurs.
MouseClick	This event occurs when the user clicks the control with the mouse. The handler for this event receives an argument of type MouseEventArgs. Handle this event when you need to get information about the mouse when a click occurs.
DoubleClick	This event occurs when the control is double-clicked. The handler for this event receives an argument of type EventArgs. Handle this event when you only need to determine when a double-click occurs.
MouseDoubleClick	This event occurs when the user double-clicks the control with the mouse. The handler for this event receives an argument of type MouseEventArgs. Handle this event when you need to get information about the mouse when a double-click occurs.
MouseDown	This event occurs when the mouse pointer is over the control and the user presses a mouse button. The handler for this event receives an argument of type MouseEventArgs.
MouseEnter	This event occurs when the mouse pointer enters the border or client area of the control, depending on the type of control. The handler for this event receives an argument of type EventArgs.

Mouse Event	Description
MouseHover	This event occurs when the mouse pointer stops and rests over the control. The handler for this event receives an argument of type EventArgs.
MouseLeave	This event occurs when the mouse pointer leaves the border or client area of the control, depending on the type of the control. The handler for this event receives an argument of type EventArgs.
MouseMove	This event occurs when the mouse pointer moves while it is over a control. The handler for this event receives an argument of type MouseEventArgs.
MouseUp	This event occurs when the mouse pointer is over the control and the user releases a mouse button. The handler for this event receives an argument of type MouseEventArgs.
MouseWheel	This event occurs when the user rotates the mouse wheel while the control has focus. The handler for this event receives an argument of type MouseEventArgs. You can use the Delta property of MouseEventArgs to determine how far the mouse has scrolled.

Mouse information

A MouseEventArgs is sent to the handlers of mouse events related to clicking a mouse button and tracking mouse movements. MouseEventArgs provides information about the current state of the mouse, including the location of the mouse pointer in client coordinates, which mouse buttons are pressed, and whether the mouse wheel has scrolled. Several mouse events, such as those that are raised when the mouse pointer has entered or left the bounds of a control, send an EventArgs to the event handler with no further information.

If you want to know the current state of the mouse buttons or the location of the mouse pointer, and you want to avoid handling a mouse event, you can also use the MouseButtons and MousePosition properties of the Control class. MouseButtons returns information about which mouse buttons are currently pressed. The MousePosition returns the screen coordinates of the mouse pointer and is equivalent to the value returned by Position.

Converting Between Screen and Client Coordinates

Because some mouse location information is in client coordinates and some is in screen coordinates, you may need to convert a point from one coordinate system to the other.

You can do this easily by using the PointToClient and PointToScreen methods available on the Control class.

Standard Click event behavior

If you want to handle mouse click events in the proper order, you need to know the order in which click events are raised in Windows Forms controls. All Windows Forms controls raise click events in the same order when any supported mouse button is pressed and released, except where noted in the following list for individual controls. The following list shows the order of events raised for a single mouse-button click:

1. MouseDown event.
2. Click event.
3. MouseClick event.
4. MouseUp event.

The following is the order of events raised for a double mouse-button click:

1. MouseDown event.
2. Click event.
3. MouseClick event.
4. MouseUp event.
5. MouseDown event.
6. DoubleClick event.

   This can vary, depending on whether the control in question has the StandardDoubleClick style bit set to `true`. For more information about how to set a ControlStyles bit, see the SetStyle method.
7. MouseDoubleClick event.
8. MouseUp event.

Individual controls

The following controls don't conform to the standard mouse click event behavior:

- Button
- CheckBox

- ComboBox
- RadioButton

  > **Note**
  >
  > For the **ComboBox** control, the event behavior detailed later occurs if the user clicks on the edit field, the button, or on an item within the list.

  - **Left click**: Click, MouseClick
  - **Right click**: No click events raised
  - **Left double-click**: Click, MouseClick; Click, MouseClick
  - **Right double-click**: No click events raised
- TextBox, RichTextBox, ListBox, MaskedTextBox, and CheckedListBox controls

  > **Note**
  >
  > The event behavior detailed later occurs when the user clicks anywhere within these controls.

  - **Left click**: Click, MouseClick
  - **Right click**: No click events raised
  - **Left double-click**: Click, MouseClick, DoubleClick, MouseDoubleClick
  - **Right double-click**: No click events raised
- ListView control

  > **Note**
  >
  > The event behavior detailed later occurs only when the user clicks on the items in the **ListView** control. No events are raised for clicks anywhere else on the control. In addition to the events described later, there are the **BeforeLabelEdit** and **AfterLabelEdit** events, which may be of interest to you if you want to use validation with the **ListView** control.

  - **Left click**: Click, MouseClick
  - **Right click**: Click, MouseClick
  - **Left double-click**: Click, MouseClick; DoubleClick, MouseDoubleClick
  - **Right double-click**: Click, MouseClick; DoubleClick, MouseDoubleClick
- TreeView control

> ⓘ Note
>
> The event behavior detailed later occurs only when the user clicks on the items themselves or to the right of the items in the **TreeView** control. No events are raised for clicks anywhere else on the control. In addition to those described later, there are the **BeforeCheck**, **BeforeSelect**, **BeforeLabelEdit**, **AfterSelect**, **AfterCheck**, and **AfterLabelEdit** events, which may be of interest to you if you want to use validation with the **TreeView** control.

- **Left click**: Click, MouseClick
- **Right click**: Click, MouseClick
- **Left double-click**: Click, MouseClick; DoubleClick, MouseDoubleClick
- **Right double-click**: Click, MouseClick; DoubleClick, MouseDoubleClick

Painting behavior of toggle controls

Toggle controls, such as the controls deriving from the ButtonBase class, have the following distinctive painting behavior in combination with mouse click events:

1. The user presses the mouse button.
2. The control paints in the pressed state.
3. The MouseDown event is raised.
4. The user releases the mouse button.
5. The control paints in the raised state.
6. The Click event is raised.
7. The MouseClick event is raised.
8. The MouseUp event is raised.

> ⓘ Note
>
> If the user moves the pointer out of the toggle control while the mouse button is down (such as moving the mouse off the **Button** control while it is pressed), the toggle control will paint in the raised state and only the **MouseUp** event occurs. The **Click** or **MouseClick** events will not occur in this situation.

See also

- Overview of using the mouse (Windows Forms .NET)
- Manage mouse pointers (Windows Forms .NET)
- How to simulate mouse events (Windows Forms .NET)
- System.Windows.Forms.Control

Drag-and-drop mouse behavior overview (Windows Forms .NET)

Article • 10/28/2020

Windows Forms includes a set of methods, events, and classes that implement drag-and-drop behavior. This topic provides an overview of the drag-and-drop support in Windows Forms.

Drag-and-drop events

There are two categories of events in a drag and drop operation: events that occur on the current target of the drag-and-drop operation, and events that occur on the source of the drag and drop operation. To perform drag-and-drop operations, you must handle these events. By working with the information available in the event arguments of these events, you can easily facilitate drag-and-drop operations.

Events on the current drop target

The following table shows the events that occur on the current target of a drag-and-drop operation.

Expand table

Mouse Event	Description
DragEnter	This event occurs when an object is dragged into the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragOver	This event occurs when an object is dragged while the mouse pointer is within the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragDrop	This event occurs when a drag-and-drop operation is completed. The handler for this event receives an argument of type DragEventArgs.
DragLeave	This event occurs when an object is dragged out of the control's bounds. The handler for this event receives an argument of type EventArgs.

The DragEventArgs class provides the location of the mouse pointer, the current state of the mouse buttons and modifier keys of the keyboard, the data being dragged, and

DragDropEffects values that specify the operations allowed by the source of the drag event and the target drop effect for the operation.

Events on the drop source

The following table shows the events that occur on the source of the drag-and-drop operation.

Expand table

Mouse Event	Description
GiveFeedback	This event occurs during a drag operation. It provides an opportunity to give a visual cue to the user that the drag-and-drop operation is occurring, such as changing the mouse pointer. The handler for this event receives an argument of type GiveFeedbackEventArgs.
QueryContinueDrag	This event is raised during a drag-and-drop operation and enables the drag source to determine whether the drag-and-drop operation should be canceled. The handler for this event receives an argument of type QueryContinueDragEventArgs.

The QueryContinueDragEventArgs class provides the current state of the mouse buttons and modifier keys of the keyboard, a value specifying whether the ESC key was pressed, and a DragAction value that can be set to specify whether the drag-and-drop operation should continue.

Performing drag-and-drop

Drag-and-drop operations always involve two components, the **drag source** and the **drop target**. To start a drag-and-drop operation, designate a control as the source and handle the MouseDown event. In the event handler, call the DoDragDrop method providing the data associated with the drop and the a DragDropEffects value.

Set the target control's AllowDrop property set to `true` to allow that control to accept a drag-and-drop operation. The target handles two events, first an event in response to the drag being over the control, such as DragOver. And a second event which is the drop action itself, DragDrop.

The following example demonstrates a drag from a Label control to a TextBox. When the drag is completed, the `TextBox` responds by assigning the label's text to itself.

```
C#
```

```
// Initiate the drag
private void label1_MouseDown(object sender, MouseEventArgs e) =>
    DoDragDrop(((Label)sender).Text, DragDropEffects.All);

// Set the effect filter and allow the drop on this control
private void textBox1_DragOver(object sender, DragEventArgs e) =>
    e.Effect = DragDropEffects.All;

// React to the drop on this control
private void textBox1_DragDrop(object sender, DragEventArgs e) =>
    textBox1.Text = (string)e.Data.GetData(typeof(string));
```

For more information about the drag effects, see Data and AllowedEffect.

See also

- Overview of using the mouse (Windows Forms .NET)
- Control.DragDrop
- Control.DragEnter
- Control.DragLeave
- Control.DragOver

How to distinguish between clicks and double-clicks (Windows Forms .NET)

Article • 04/19/2024

Typically, a single *click* initiates a user interface action and a *double-click* extends the action. For example, one click usually selects an item, and a double-click edits the selected item. However, the Windows Forms click events do not easily accommodate a scenario where a click and a double-click perform incompatible actions, because an action tied to the Click or MouseClick event is performed before the action tied to the DoubleClick or MouseDoubleClick event. This topic demonstrates two solutions to this problem.

One solution is to handle the double-click event and roll back the actions in the handling of the click event. In rare situations you may need to simulate click and double-click behavior by handling the MouseDown event and by using the DoubleClickTime and DoubleClickSize properties of the SystemInformation class. You measure the time between clicks and if a second click occurs before the value of DoubleClickTime is reached and the click is within a rectangle defined by DoubleClickSize, perform the double-click action; otherwise, perform the click action.

To roll back a click action

Ensure that the control you are working with has standard double-click behavior. If not, enable the control with the SetStyle method. Handle the double-click event and roll back the click action as well as the double-click action. The following code example demonstrates a how to create a custom button with double-click enabled, as well as how to roll back the click action in the double-click event handling code.

This code example uses a new button control that enables double-clicks:

C#

```
public partial class DoubleClickButton : Button
{
    public DoubleClickButton()
    {
        // Set the style so a double click event occurs.
        SetStyle(ControlStyles.StandardClick |
ControlStyles.StandardDoubleClick, true);
    }
}
```

The following code demonstrates how a form changes the style of border based on a click or double-click of the new button control:

C#

```
public partial class Form1 : Form
{
    private FormBorderStyle _initialStyle;
    private bool _isDoubleClicking;

    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        _initialStyle = this.FormBorderStyle;

        var button1 = new DoubleClickButton();
        button1.Location = new Point(50, 50);
        button1.Size = new Size(200, 23);
        button1.Text = "Click or Double Click";
        button1.Click += Button1_Click;
        button1.DoubleClick += Button1_DoubleClick;

        Controls.Add(button1);
    }

    private void Button1_DoubleClick(object sender, EventArgs e)
    {
        // This flag prevents the click handler logic from running
        // A double click raises the click event twice.
        _isDoubleClicking = true;
        FormBorderStyle = _initialStyle;
    }

    private void Button1_Click(object sender, EventArgs e)
    {
        if (_isDoubleClicking)
            _isDoubleClicking = false;
        else
            FormBorderStyle = FormBorderStyle.FixedToolWindow;
    }
}
```

To distinguish between clicks

Handle the MouseDown event and determine the location and time span between clicks using the SystemInformation property and a Timer component. Perform the appropriate

action depending on whether a click or double-click takes place. The following code example demonstrates how this can be done.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

namespace project
{
    public partial class Form2 : Form
    {
        private DateTime _lastClick;
        private bool _inDoubleClick;
        private Rectangle _doubleClickArea;
        private TimeSpan _doubleClickMaxTime;
        private Action _doubleClickAction;
        private Action _singleClickAction;
        private Timer _clickTimer;

        public Form2()
        {
            InitializeComponent();
            _doubleClickMaxTime =
TimeSpan.FromMilliseconds(SystemInformation.DoubleClickTime);

            _clickTimer = new Timer();
            _clickTimer.Interval = SystemInformation.DoubleClickTime;
            _clickTimer.Tick += ClickTimer_Tick;

            _singleClickAction = () => MessageBox.Show("Single clicked");
            _doubleClickAction = () => MessageBox.Show("Double clicked");
        }

        private void Form2_MouseDown(object sender, MouseEventArgs e)
        {
            if (_inDoubleClick)
            {
                _inDoubleClick = false;

                TimeSpan length = DateTime.Now - _lastClick;

                // If double click is valid, respond
                if (_doubleClickArea.Contains(e.Location) && length <
_doubleClickMaxTime)
                {
                    _clickTimer.Stop();
                    _doubleClickAction();
                }

                return;
```

```
            }

            // Double click was invalid, restart
            _clickTimer.Stop();
            _clickTimer.Start();
            _lastClick = DateTime.Now;
            _inDoubleClick = true;
            _doubleClickArea = new Rectangle(e.Location -
(SystemInformation.DoubleClickSize / 2),

SystemInformation.DoubleClickSize);
        }

        private void ClickTimer_Tick(object sender, EventArgs e)
        {
            // Clear double click watcher and timer
            _inDoubleClick = false;
            _clickTimer.Stop();

            _singleClickAction();
        }
    }
}
```

See also

- Overview of using the mouse (Windows Forms .NET)
- Using mouse events (Windows Forms .NET)
- Manage mouse pointers (Windows Forms .NET)
- How to simulate mouse events (Windows Forms .NET)
- Control.Click
- Control.MouseDown
- Control.SetStyle

Manage mouse pointers (Windows Forms .NET)

Article • 10/28/2020

The mouse *pointer*, which is sometimes referred to as the cursor, is a bitmap that specifies a focus point on the screen for user input with the mouse. This topic provides an overview of the mouse pointer in Windows Forms and describes some of the ways to modify and control the mouse pointer.

Accessing the mouse pointer

The mouse pointer is represented by the Cursor class, and each Control has a Control.Cursor property that specifies the pointer for that control. The Cursor class contains properties that describe the pointer, such as the Position and HotSpot properties, and methods that can modify the appearance of the pointer, such as the Show, Hide, and DrawStretched methods.

The following example hides the cursor when the cursor is over a button:

C#

```
private void button1_MouseEnter(object sender, EventArgs e) =>
    Cursor.Hide();

private void button1_MouseLeave(object sender, EventArgs e) =>
    Cursor.Show();
```

Controlling the mouse pointer

Sometimes you may want to limit the area in which the mouse pointer can be used or change the position the mouse. You can get or set the current location of the mouse using the Position property of the Cursor. In addition, you can limit the area the mouse pointer can be used be setting the Clip property. The clip area, by default, is the entire screen.

The following example positions the mouse pointer between two buttons when they are clicked:

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Cursor.Position = PointToScreen(button2.Location);

private void button2_Click(object sender, EventArgs e) =>
    Cursor.Position = PointToScreen(button1.Location);
```

Changing the mouse pointer

Changing the mouse pointer is an important way of providing feedback to the user. For example, the mouse pointer can be modified in the handlers of the MouseEnter and MouseLeave events to tell the user that computations are occurring and to limit user interaction in the control. Sometimes, the mouse pointer will change because of system events, such as when your application is involved in a drag-and-drop operation.

The primary way to change the mouse pointer is by setting the Control.Cursor or DefaultCursor property of a control to a new Cursor. For examples of changing the mouse pointer, see the code example in the Cursor class. In addition, the Cursors class exposes a set of Cursor objects for many different types of pointers, such as a pointer that resembles a hand.

The following example changes the cursor of the mouse pointer for a button to a hand:

C#

```
button2.Cursor = System.Windows.Forms.Cursors.Hand;
```

To display the wait pointer, which resembles an hourglass, whenever the mouse pointer is on the control, use the UseWaitCursor property of the Control class.

See also

- Overview of using the mouse (Windows Forms .NET)
- Using mouse events (Windows Forms .NET)
- How to distinguish between clicks and double-clicks (Windows Forms .NET)
- How to simulate mouse events (Windows Forms .NET)
- System.Windows.Forms.Cursor
- Cursor.Position

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback feedback

.NET Desktop feedback is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

How to simulate mouse events (Windows Forms .NET)

Article • 04/19/2024

Simulating mouse events in Windows Forms isn't as straight forward as simulating keyboard events. Windows Forms doesn't provide a helper class to move the mouse and invoke mouse-click actions. The only option for controlling the mouse is to use native Windows methods. If you're working with a custom control or a form, you can simulate a mouse event, but you can't directly control the mouse.

Events

Most events have a corresponding method that invokes them, named in the pattern of `On` followed by `EventName`, such as `OnMouseMove`. This option is only possible within custom controls or forms, because these methods are protected and can't be accessed from outside the context of the control or form. The disadvantage to using a method such as `OnMouseMove` is that it doesn't actually control the mouse or interact with the control, it simply raises the associated event. For example, if you wanted to simulate hovering over an item in a ListBox, `OnMouseMove` and the `ListBox` doesn't visually react with a highlighted item under the cursor.

These protected methods are available to simulate mouse events.

- `OnMouseDown`
- `OnMouseEnter`
- `OnMouseHover`
- `OnMouseLeave`
- `OnMouseMove`
- `OnMouseUp`
- `OnMouseWheel`
- `OnMouseClick`
- `OnMouseDoubleClick`

For more information about these events, see Using mouse events (Windows Forms .NET)

Invoke a click

Considering most controls do something when clicked, like a button calling user code, or checkbox change its checked state, Windows Forms provides an easy way to trigger the click. Some controls, such as a combobox, don't do anything special when clicked and simulating a click has no effect on the control.

PerformClick

The System.Windows.Forms.IButtonControl interface provides the PerformClick method which simulates a click on the control. Both the System.Windows.Forms.Button and System.Windows.Forms.LinkLabel controls implement this interface.

C#

```
button1.PerformClick();
```

InvokeClick

With a form a custom control, use the InvokeOnClick method to simulate a mouse click. This is a protected method that can only be called from within the form or a derived custom control.

For example, the following code clicks a checkbox from `button1`.

C#

```
private void button1_Click(object sender, EventArgs e)
{
    InvokeOnClick(checkBox1, EventArgs.Empty);
}
```

Use native Windows methods

Windows provides methods you can call to simulate mouse movements and clicks such as User32.dll SendInput and User32.dll SetCursorPos. The following example moves the mouse cursor to the center of a control:

C#

```
[DllImport("user32.dll", EntryPoint = "SetCursorPos")]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern bool SetCursorPos(int x, int y);

private void button1_Click(object sender, EventArgs e)
```

```
{
    Point position = PointToScreen(checkBox1.Location) + new
Size(checkBox1.Width / 2, checkBox1.Height / 2);
    SetCursorPos(position.X, position.Y);
}
```

See also

- Overview of using the mouse (Windows Forms .NET)
- Using mouse events (Windows Forms .NET)
- How to distinguish between clicks and double-clicks (Windows Forms .NET)
- Manage mouse pointers (Windows Forms .NET)

PrintDialog component overview (Windows Forms .NET)

Article • 06/18/2022

Printing in Windows Forms consists primarily of using the PrintDocument component to enable the user to print. The PrintPreviewDialog control, PrintDialog and PageSetupDialog components provide a familiar graphical interface to Windows operating system users.

The `PrintDialog` component is a pre-configured dialog box used to select a printer, choose the pages to print, and determine other print-related settings in Windows-based applications. It's a simple solution for printer and print-related settings instead of configuring your own dialog box. You can enable users to print many parts of their documents: print all, print a selected page range, or print a selection. By relying on standard Windows dialog boxes, you create applications whose basic functionality is immediately familiar to users. The PrintDialog component inherits from the CommonDialog class.

Typically, you create a new instance of the PrintDocument component and set the properties that describe what to print using the PrinterSettings and PageSettings classes. Call to the Print method actually prints the document.

Working with the component

Use the PrintDialog.ShowDialog method to display the dialog at run time. This component has properties that relate to either a single print job (PrintDocument class) or the settings of an individual printer (PrinterSettings class). One of the two, in turn, may be shared by multiple printers.

The show dialog box method helps you to add print dialog box to the form. The PrintDialog component appears in the tray at the bottom of the Windows Forms Designer in Visual Studio.

How to capture user input from a PrintDialog at run time

You can set options related to printing at design time. Sometimes you may want to change these options at run time, most likely because of choices made by the user. You can capture user input for printing a document using the PrintDialog and the

PrintDocument components. The following steps demonstrate displaying the print dialog for a document:

1. Add a PrintDialog and a PrintDocument component to your form.
2. Set the Document property of the PrintDialog to the PrintDocument added to the form.

   C#

   ```
   printDialog1.Document = printDocument1;
   ```

3. Display the PrintDialog component by using the ShowDialog method.

   C#

   ```
   // display show dialog and if user selects "Ok" document is printed
   if (printDialog1.ShowDialog() == DialogResult.OK)
       printDocument1.Print();
   ```

4. The user's printing choices from the dialog will be copied to the PrinterSettings property of the PrintDocument component.

How to create print jobs

The foundation of printing in Windows Forms is the PrintDocument component—more specifically, the PrintPage event. By writing code to handle the PrintPage event, you can specify what to print and how to print it. The following steps demonstrate creating print job:

1. Add a PrintDocument component to your form.
2. Write code to handle the PrintPage event.

   You'll have to code your own printing logic. Additionally, you'll have to specify the material to be printed.

   As a material to print, in the following code example, a sample graphic in the shape of a red rectangle is created in the PrintPage event handler.

   C#

   ```
   private void PrintDocument1_PrintPage(object sender,
   System.Drawing.Printing.PrintPageEventArgs e) =>
   ```

```
    e.Graphics.FillRectangle(Brushes.Red, new Rectangle(100, 100, 100,
100));
```

You may also want to write code for the BeginPrint and EndPrint events. It will help to include an integer representing the total number of pages to print that is decremented as each page prints.

> **Note**
>
> You can add a **PrintDialog** component to your form to provide a clean and efficient user interface (UI) to your users. Setting the **Document** property of the **PrintDialog** component enables you to set properties related to the print document you're working with on your form.

For more information about the specifics of Windows Forms print jobs, including how to create a print job programmatically, see PrintPageEventArgs.

How to complete print jobs

Frequently, word processors and other applications that involve printing will provide the option to display a message to users that a print job is complete. You can provide this functionality in your Windows Forms by handling the EndPrint event of the PrintDocument component.

The following procedure requires that you've created a Windows-based application with a PrintDocument component on it. The procedure given below is the standard way of enabling printing from a Windows-based application. For more information about printing from Windows Forms using the PrintDocument component, see How to create print jobs.

1. Set the DocumentName property of the PrintDocument component.

   ```
   C#

   printDocument1.DocumentName = "SamplePrintApp";
   ```

2. Write code to handle the EndPrint event.

   In the following code example, a message box is displayed, indicating that the document has finished printing.

   ```
   C#
   ```

```
private void PrintDocument1_EndPrint(object sender,
System.Drawing.Printing.PrintEventArgs e) =>
    MessageBox.Show(printDocument1.DocumentName + " has finished
printing.");
```

Print a multi-page text file (Windows Forms .NET)

Article • 06/18/2022

It's common for Windows-based applications to print text. The Graphics class provides methods for drawing objects (graphics or text) to a device, such as a screen or printer. The following section describes in detail the process to print text file. This method doesn't support printing non-plain text files, such as an Office Word document or a *PDF* file.

> ⓘ **Note**
>
> The **DrawText** methods of **TextRenderer** are not supported for printing. You should always use the **DrawString** methods of **Graphics**, as shown in the following code example, to draw text for printing purposes.

To print text

1. In Visual Studio, double-click the form you want to print from, in the **Solution Explorer** pane. This opens the Visual Designer.
2. From the **Toolbox**, double-click the PrintDocument component to add it to the form. This should create a `PrintDocument` component with the name `printDocument1`.
3. Either add a `Button` to the form, or use a button that is already on the form.
4. In the Visual Designer of the form, select the button. In the **Properties** pane, select the **Event** filter button and then double-click the `Click` event to generate an event handler.
5. The `Click` event code should be visible. Outside the scope of the event handler, add a private string variable to the class named `stringToPrint`.

```
C#

private string stringToPrint="";
```

6. Back in the `Click` event handler code, set the DocumentName property to the name of the document. This information is sent to the printer. Next, read the document text content and store it in the `stringToPrint` string. Finally, call the Print method to raise the PrintPage event. The `Print` method is highlighted below.

```
C#

private void button1_Click(object sender, EventArgs e)
{
    string docName = "testPage.txt";
    string docPath = @"C:\";
    string fullPath = System.IO.Path.Combine(docPath, docName);

    printDocument1.DocumentName = docName;

    stringToPrint = System.IO.File.ReadAllText(fullPath);

    printDocument1.Print();
}
```

7. Go back to the Visual Designer of the form and select the `PrintDocument` component. On the **Properties** pane, select the **Event** filter and then double-click the `PrintPage` event to generate an event handler.

8. In the PrintPage event handler, use the Graphics property of the PrintPageEventArgs class and the document contents to calculate line length and lines per page. After each page is drawn, check if it's the last page, and set the HasMorePages property of the `PrintPageEventArgs` accordingly. The `PrintPage` event is raised until `HasMorePages` is `false`.

   In the following code example, the event handler is used to print the contents of the "testPage.txt" file in the same font as it's used on the form.

```
C#

private void PrintDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    int charactersOnPage = 0;
    int linesPerPage = 0;

    // Sets the value of charactersOnPage to the number of characters
    // of stringToPrint that will fit within the bounds of the page.
    e.Graphics.MeasureString(stringToPrint, this.Font,
        e.MarginBounds.Size, StringFormat.GenericTypographic,
        out charactersOnPage, out linesPerPage);

    // Draws the string within the bounds of the page
```

```
    e.Graphics.DrawString(stringToPrint, this.Font, Brushes.Black,
        e.MarginBounds, StringFormat.GenericTypographic);

    // Remove the portion of the string that has been printed.
    stringToPrint = stringToPrint.Substring(charactersOnPage);

    // Check to see if more pages are to be printed.
    e.HasMorePages = (stringToPrint.Length > 0);
}
```

See also

- Graphics
- Brush
- PrintDialog component overview

How to print a Form (Windows Forms .NET)

Article • 05/10/2022

As part of the development process, you typically will want to print a copy of your Windows Form. The following code example shows how to print a copy of the current form by using the CopyFromScreen method.

In the following example, a button named **Button1** is added to the form. When the **Button1** button is clicked, it saves the form to an image in memory, and then sends it to the print object.

Example

C#

```
namespace Sample_print_win_form1
{
    public partial class Form1 : Form
    {
        Bitmap memoryImage;
        public Form1()
        {
            InitializeComponent();
        }

        private void Button1_Click(object sender, EventArgs e)
        {
            Graphics myGraphics = this.CreateGraphics();
            Size s = this.Size;
            memoryImage = new Bitmap(s.Width, s.Height, myGraphics);
            Graphics memoryGraphics = Graphics.FromImage(memoryImage);
            memoryGraphics.CopyFromScreen(this.Location.X, this.Location.Y,
0, 0, s);

            printDocument1.Print();

        }
        private void PrintDocument1_PrintPage(System.Object sender,
           System.Drawing.Printing.PrintPageEventArgs e) =>
               e.Graphics.DrawImage(memoryImage, 0, 0);

    }
}
```

Robust programming

The following conditions may cause an exception:

- You don't have permission to access the printer.
- There's no printer installed.

.NET security

To run this code example, you must have permission to access the printer you use with your computer.

See also

- PrintDocument
- How to: Render Images with GDI+
- How to: Print Graphics in Windows Forms

Print using print preview (Windows Forms .NET)

Article • 06/02/2023

It's common in Windows Forms programming to offer print preview in addition to printing services. An easy way to add print preview services to your application is to use a PrintPreviewDialog control in combination with the PrintPage event-handling logic for printing a file.

To preview a text document with a PrintPreviewDialog control

1. In Visual Studio, use the **Solution Explorer** pane and double-click the form you want to print from. This opens the Visual Designer.
2. From the **Toolbox** pane, double-click both the PrintDocument component and the PrintPreviewDialog component, to add them to the form.
3. Either add a `Button` to the form, or use a button that is already on the form.
4. In the Visual Designer of the form, select the button. In the **Properties** pane, select the **Event** filter button and then double-click the `Click` event to generate an event handler.
5. The `Click` event code should be visible. Outside the scope of the event handler, add two private string variables to the class named `documentContents` and `stringToPrint`:

   C#

   ```
   // Declare a string to hold the entire document contents.
   private string documentContents="";

   // Declare a variable to hold the portion of the document that
   // is not printed.
   private string stringToPrint="";
   ```

6. Back in the `Click` event handler code, set the DocumentName property to the document you wish to print, and open and read the document's contents to the string you added previously.

C#

```
string docName = "testPage.txt";
string docPath = @"C:\";
string fullPath = System.IO.Path.Combine(docPath, docName);
printDocument1.DocumentName = docName;
stringToPrint = System.IO.File.ReadAllText(fullPath);
```

7. As you would for printing the document, in the PrintPage event handler, use the Graphics property of the PrintPageEventArgs class and the file contents to calculate lines per page and render the document's contents. After each page is drawn, check to see if it's the last page, and set the HasMorePages property of the `PrintPageEventArgs` accordingly. The `PrintPage` event is raised until `HasMorePages` is `false`. When the document has finished rendering, reset the string to be rendered. Also, ensure that the `PrintPage` event is associated with its event-handling method.

> **Note**
>
> If you have implemented printing in your application, you may have already completed step 5 and 6.

In the following code example, the event handler is used to print the "testPage.txt" file in the same font used on the form.

C#

```
void PrintDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
    int charactersOnPage = 0;
    int linesPerPage = 0;

    // Sets the value of charactersOnPage to the number of characters
    // of stringToPrint that will fit within the bounds of the page.
    e.Graphics.MeasureString(stringToPrint, this.Font,
        e.MarginBounds.Size, StringFormat.GenericTypographic,
        out charactersOnPage, out linesPerPage);

    // Draws the string within the bounds of the page.
    e.Graphics.DrawString(stringToPrint, this.Font, Brushes.Black,
    e.MarginBounds, StringFormat.GenericTypographic);

    // Remove the portion of the string that has been printed.
    stringToPrint = stringToPrint.Substring(charactersOnPage);

    // Check to see if more pages are to be printed.
    e.HasMorePages = (stringToPrint.Length > 0);
```

```
    // If there are no more pages, reset the string to be printed.
    if (!e.HasMorePages)
        stringToPrint = documentContents;
}
```

8. Set the Document property of the PrintPreviewDialog control to the PrintDocument component on the form.

```
C#

printPreviewDialog1.Document = printDocument1;
```

9. Call the ShowDialog method on the PrintPreviewDialog control. Note the highlighted code given below, you would typically call ShowDialog from the Click event-handling method of a button. Calling ShowDialog raises the PrintPage event and renders the output to the `PrintPreviewDialog` control. When the user selects the print icon on the dialog, the `PrintPage` event is raised again, sending the output to the printer instead of the preview dialog. Hence, the string is reset at the end of the rendering process in step 4.

   The following code example shows the Click event-handling method for a button on the form. The event-handling method calls the methods to read the document and show the print preview dialog.

```
C#

private void Button1_Click(object sender, EventArgs e)
{
    string docName = "testPage.txt";
    string docPath = @"C:\";
    string fullPath = System.IO.Path.Combine(docPath, docName);
    printDocument1.DocumentName = docName;
    stringToPrint = System.IO.File.ReadAllText(fullPath);

    printPreviewDialog1.Document = printDocument1;

    printPreviewDialog1.ShowDialog();
}
```

See also

- PrintDialog component overview
- Print a multi-page text file
- More Secure Printing in Windows Forms

Data binding overview (Windows Forms .NET)

Article • 06/18/2022

In Windows Forms, you can bind to not just traditional data sources, but also to almost any structure that contains data. You can bind to an array of values that you calculate at run time, read from a file, or derive from the values of other controls.

In addition, you can bind any property of any control to the data source. In traditional data binding, you typically bind the display property—for example, the Text property of a TextBox control—to the data source. With .NET, you also have the option of setting other properties through binding. You might use binding to perform the following tasks:

- Setting the graphic of an image control.
- Setting the background color of one or more controls.
- Setting the size of controls.

Essentially, data binding is an automatic way of setting any run-time accessible property of any control on a form.

Interfaces related to data binding

ADO.NET lets you create many different data structures to suit the binding needs of your application and the data you're working with. You might want to create your own classes that provide or consume data in Windows Forms. These objects can offer varying levels of functionality and complexity. From basic data binding, to providing design-time support, error checking, change notification, or even support for a structured rollback of the changes made to the data itself.

Consumers of data binding interfaces

The following sections describe two groups of interface objects. The first group of interface is implemented on data sources by data source authors. The data source consumers such as the Windows Forms controls or components implement these interfaces. The second group of interface is designed to use by component authors. Component authors use these interfaces when they're creating a component that supports data binding to be consumed by the Windows Forms data binding engine. You can implement these interfaces within classes associated with your form to enable data

binding. Each case presents a class that implements an interface that enables interaction with data. Visual Studio rapid application development (RAD) data design experience tools already take advantage of this functionality.

Interfaces for implementation by data source authors

The Windows Forms controls implement following interfaces:

- IList interface

  A class that implements the IList interface could be an Array, ArrayList, or CollectionBase. These are indexed lists of items of type Object and the lists must contain homogenous types, because the first item of the index determines the type. `IList` would be available for binding only at run time.

  > **Note**
  >
  > If you want to create a list of business objects for binding with Windows Forms, you should consider using the **BindingList<T>**. The `BindingList` is an extensible class that implements the primary interfaces required for two-way Windows Forms data binding.

- IBindingList interface

  A class that implements the IBindingList interface provides a much higher level of data-binding functionality. This implementation offers you basic sorting capabilities and change notification. Both are useful when the list items change, and when the list itself changes. Change notification is important if you plan to have multiple controls bound to the same data. It helps you to make data changes made in one of the controls to propagate to the other bound controls.

  > **Note**
  >
  > Change notification is enabled for the **IBindingList** interface through the **SupportsChangeNotification** property which, when `true`, raises a **ListChanged** event, indicating the list changed or an item in the list changed.

  The type of change is described by the ListChangedType property of the ListChangedEventArgs parameter. Hence, whenever the data model is updated, any dependent views, such as other controls bound to the same data source, will

also be updated. However, objects contained within the list will have to notify the list when they change so that the list can raise the ListChanged event.

> **Note**
>
> The **BindingList<T>** provides a generic implementation of the **IBindingList** interface.

- IBindingListView interface

  A class that implements the IBindingListView interface provides all the functionality of an implementation of IBindingList, along with filtering and advanced sorting functionality. This implementation offers string-based filtering, and multi-column sorting with property descriptor-direction pairs.

- IEditableObject interface

  A class that implements the IEditableObject interface allows an object to control when changes to that object are made permanent. This implementation supports the BeginEdit, EndEdit, and CancelEdit methods, which enable you to roll back changes made to the object. Following is a brief explanation of the functioning of the `BeginEdit`, `EndEdit`, and `CancelEdit` methods and how they work with one another to enable a possible rollback of changes made to the data:

  - The BeginEdit method signals the start of an edit on an object. An object that implements this interface will need to store any updates after the `BeginEdit` method call in such a way that the updates can be discarded if the CancelEdit method is called. In data binding Windows Forms, you can call `BeginEdit` multiple times within the scope of a single edit transaction (for example, `BeginEdit`, `BeginEdit`, EndEdit). Implementations of IEditableObject should keep track of whether `BeginEdit` has already been called and ignore subsequent calls to `BeginEdit`. Because this method can be called multiple times, it's important that subsequent calls to it are nondestructive. Subsequent `BeginEdit` calls can't destroy the updates that have been made or change the data that was saved on the first `BeginEdit` call.

  - The EndEdit method pushes any changes since BeginEdit was called into the underlying object, if the object is currently in edit mode.

  - The CancelEdit method discards any changes made to the object.

For more information about how the BeginEdit, EndEdit, and CancelEdit methods work, see Save data back to the database.

This transactional notion of data functionality is used by the DataGridView control.

- ICancelAddNew interface

  A class that implements the ICancelAddNew interface usually implements the IBindingList interface and allows you to roll back an addition made to the data source with the AddNew method. If your data source implements the `IBindingList` interface, you should also have it implement the `ICancelAddNew` interface.

- IDataErrorInfo interface

  A class that implements the IDataErrorInfo interface allows objects to offer custom error information to bound controls:

  - The Error property returns general error message text (for example, "An error has occurred").
  - The Item[] property returns a string with the specific error message from the column (for example, "The value in the `State` column is invalid").

- IEnumerable interface

  A class that implements the IEnumerable interface is typically consumed by ASP.NET. Windows Forms support for this interface is only available through the BindingSource component.

  > ⓘ **Note**
  >
  > The **BindingSource** component copies all **IEnumerable** items into a separate list for binding purposes.

- ITypedList interface

  A collections class that implements the ITypedList interface provides the feature to control the order and the set of properties exposed to the bound control.

  > ⓘ **Note**
  >
  > When you implement the **GetItemProperties** method, and the **PropertyDescriptor** array is not null, the last entry in the array will be the

property descriptor that describes the list property that is another list of items.

- ICustomTypeDescriptor interface

  A class that implements the ICustomTypeDescriptor interface provides dynamic information about itself. This interface is similar to ITypedList but is used for objects rather than lists. This interface is used by DataRowView to project the schema of the underlying rows. A simple implementation of `ICustomTypeDescriptor` is provided by the CustomTypeDescriptor class.

  > ⓘ **Note**
  >
  > To support design-time binding to types that implement **ICustomTypeDescriptor**, the type must also implement **IComponent** and exist as an instance on the Form.

- IListSource interface

  A class that implements the IListSource interface enables list-based binding on non-list objects. The GetList method of `IListSource` is used to return a bindable list from an object that doesn't inherit from IList. `IListSource` is used by the DataSet class.

- IRaiseItemChangedEvents interface

  A class that implements the IRaiseItemChangedEvents interface is a bindable list that also implements the IBindingList interface. This interface is used to indicate if your type raises ListChanged events of type ItemChanged through its RaisesItemChangedEvents property.

  > ⓘ **Note**
  >
  > You should implement the **IRaiseItemChangedEvents** if your data source provides the property to list event conversion described previously and is interacting with the **BindingSource** component. Otherwise, the `BindingSource` will also perform property to list event conversion resulting in slower performance.

- ISupportInitialize interface

A component that implements the ISupportInitialize interface takes advantages of batch optimizations for setting properties and initializing co-dependent properties. The `ISupportInitialize` contains two methods:

  - BeginInit signals that object initialization is starting.
  - EndInit signals that object initialization is finishing.

- ISupportInitializeNotification interface

  A component that implements the ISupportInitializeNotification interface also implements the ISupportInitialize interface. This interface allows you to notify other `ISupportInitialize` components that initialization is complete. The `ISupportInitializeNotification` interface contains two members:

  - IsInitialized returns a `boolean` value indicating whether the component is initialized.
  - Initialized occurs when EndInit is called.

- INotifyPropertyChanged interface

  A class that implements this interface is a type that raises an event when any of its property values change. This interface is designed to replace the pattern of having a change event for each property of a control. When used in a BindingList<T>, a business object should implement the INotifyPropertyChanged interface and the BindingList`1 will convert PropertyChanged events to ListChanged events of type ItemChanged.

  > **Note**
  >
  > For change notification to occur in a binding between a bound client and a data source, your bound data-source type should either implement the **INotifyPropertyChanged** interface (preferred) or you can provide *propertyName*`Changed` events for the bound type, but you shouldn't do both.

Interfaces for implementation by component authors

The following interfaces are designed for consumption by the Windows Forms data-binding engine:

- IBindableComponent interface

A class that implements this interface is a non-control component that supports data binding. This class returns the data bindings and binding context of the component through the DataBindings and BindingContext properties of this interface.

> **Note**
>
> If your component inherits from **Control**, you don't need to implement the **IBindableComponent** interface.

- ICurrencyManagerProvider interface

  A class that implements the ICurrencyManagerProvider interface is a component that provides its own CurrencyManager to manage the bindings associated with this particular component. Access to the custom `CurrencyManager` is provided by the CurrencyManager property.

  > **Note**
  >
  > A class that inherits from **Control** manages bindings automatically through its **BindingContext** property, so cases in which you need to implement the **ICurrencyManagerProvider** are fairly rare.

Data sources supported by Windows Forms

Traditionally, data binding has been used within applications to take advantage of data stored in databases. With Windows Forms data binding, you can access data from databases and data in other structures, such as arrays and collections, so long as certain minimum requirements have been met.

Structures to bind To

In Windows Forms, you can bind to a wide variety of structures, from simple objects (simple binding) to complex lists such as ADO.NET data tables (complex binding). For simple binding, Windows Forms support binding to the public properties on the simple object. Windows Forms list-based binding generally requires that the object supports the IList interface or the IListSource interface. Additionally, if you're binding with through a BindingSource component, you can bind to an object that supports the IEnumerable interface.

The following list shows the structures you can bind to in Windows Forms.

- BindingSource

  A BindingSource is the most common Windows Forms data source and acts a proxy between a data source and Windows Forms controls. The general `BindingSource` usage pattern is to bind your controls to the `BindingSource` and bind the `BindingSource` to the data source (for example, an ADO.NET data table or a business object). The `BindingSource` provides services that enable and improve the level of data binding support. For example, Windows Forms list based controls such as the DataGridView and ComboBox don't directly support binding to IEnumerable data sources however, you can enable this scenario by binding through a `BindingSource`. In this case, the `BindingSource` will convert the data source to an IList.

- Simple objects

  Windows Forms support data binding control properties to public properties on the instance of an object using the Binding type. Windows Forms also support binding list based controls, such as a ListControl to an object instance when a BindingSource is used.

- Array or Collection

  To act as a data source, a list must implement the IList interface; one example would be an array that is an instance of the Array class. For more information on arrays, see How to: Create an Array of Objects (Visual Basic).

  In general, you should use BindingList<T> when you create lists of objects for data binding. `BindingList` is a generic version of the IBindingList interface. The `IBindingList` interface extends the IList interface by adding properties, methods and events necessary for two-way data binding.

- IEnumerable

  Windows Forms controls can be bound to data sources that only support the IEnumerable interface if they're bound through a BindingSource component.

- ADO.NET data objects

  ADO.NET provides many data structures suitable for binding to. Each varies in its sophistication and complexity.

  - DataColumn

A DataColumn is the essential building block of a DataTable, in that multiple columns comprise a table. Each `DataColumn` has a DataType property that determines the kind of data the column holds (for example, the make of an automobile in a table describing cars). You can simple-bind a control (such as a TextBox control's Text property) to a column within a data table.

- DataTable

  A DataTable is the representation of a table, with rows and columns, in ADO.NET. A data table contains two collections: DataColumn, representing the columns of data in a given table (which ultimately determine the kinds of data that can be entered into that table), and DataRow, representing the rows of data in a given table. You can complex-bind a control to the information contained in a data table (such as binding the DataGridView control to a data table). However, when you bind to a `DataTable`, you're a binding to the table's default view.

- DataView

  A DataView is a customized view of a single data table that may be filtered or sorted. A data view is the data "snapshot" used by complex-bound controls. You can simple-bind or complex-bind to the data within a data view, but note that you're binding to a fixed "picture" of the data rather than a clean, updating data source.

- DataSet

  A DataSet is a collection of tables, relationships, and constraints of the data in a database. You can simple-bind or complex-bind to the data within a dataset, but note that you're binding to the default DataViewManager for the `DataSet` (see the next bullet point).

- DataViewManager

  A DataViewManager is a customized view of the entire DataSet, analogous to a DataView, but with relations included. With a DataViewSettings collection, you can set default filters and sort options for any views that the `DataViewManager` has for a given table.

Types of data binding

Windows Forms can take advantage of two types of data binding: simple binding and complex binding. Each offers different advantages.

Type of data binding	Description
Simple data binding	The ability of a control to bind to a single data element, such as a value in a column in a dataset table. Simple data binding is the type of binding typical for controls such as a TextBox control or Label control, which are controls that typically only display a single value. In fact, any property on a control can be bound to a field in a database. There's extensive support for this feature in Visual Studio. For more information, see Navigate data and Create a simple-bound control (Windows Forms .NET).
Complex data binding	The ability of a control to bind to more than one data element, typically more than one record in a database. Complex binding is also called list-based binding. Examples of controls that support complex binding are the DataGridView, ListBox, and ComboBox controls. For an example of complex data binding, see How to: Bind a Windows Forms ComboBox or ListBox Control to Data.

Binding source component

To simplify data binding, Windows Forms enables you to bind a data source to the BindingSource component and then bind controls to the `BindingSource`. You can use the `BindingSource` in simple or complex binding scenarios. In either case, the `BindingSource` acts as an intermediary between the data source and bound controls providing change notification currency management and other services.

Common scenarios that employ data binding

Nearly every commercial application uses information read from data sources of one type or another, usually through data binding. The following list shows a few of the most common scenarios that utilize data binding as the method of data presentation and manipulation.

Scenario	Description
Reporting	Reports provide a flexible way for you to display and summarize your data in a printed document. It's common to create a report that prints selected contents of a data source either to the screen or to a printer. Common reports include lists, invoices, and summaries. Items are formatted into columns of lists, with subitems organized under each list item, but you should choose the layout that best suits the data.

Scenario	Description
Data entry	A common way to enter large amounts of related data or to prompt users for information is through a data entry form. Users can enter information or select choices using text boxes, option buttons, drop-down lists, and check boxes. Information is then submitted and stored in a database, whose structure is based on the information entered.
Master/detail relationship	A master/detail application is one format for looking at related data. Specifically, there are two tables of data with a relation connecting in the classic business example, a "Customers" table and an "Orders" table with a relationship between them linking customers and their respective orders. For more information about creating a master/detail application with two Windows Forms DataGridView controls, see How to: Create a Master/Detail Form Using Two Windows Forms DataGridView Controls
Lookup Table	Another common data presentation/manipulation scenario is the table lookup. Often, as part of a larger data display, a ComboBox control is used to display and manipulate data. The key is that the data displayed in the `ComboBox` control is different than the data written to the database. For example, if you have a `ComboBox` control displaying the items available from a grocery store, you would probably like to see the names of the products (bread, milk, eggs). However, to ease information retrieval within the database and for database normalization, you would probably store the information for the specific items of a given order as item numbers (#501, #603, and so on). Thus, there's an implicit connection between the "friendly name" of the grocery item in the `ComboBox` control on your form and the related item number that is present in an order. It's the essence of a table lookup. For more information, see How to: Create a Lookup Table with the Windows Forms BindingSource Component.

See also

- Binding
- Data binding overview
- Design great data sources with change notification
- Create a simple-bound control (Windows Forms .NET)
- BindingSource Component
- How to: Bind the Windows Forms DataGrid Control to a Data Source

Design great data sources with change notification (Windows Forms .NET)

Article • 06/18/2022

One of the most important concepts of Windows Forms data binding is the *change notification*. To ensure that your data source and bound controls always have the most recent data, you must add change notification for data binding. Specifically, you want to ensure that bound controls are notified of changes that were made to their data source. The data source is notified of changes that were made to the bound properties of a control.

There are different kinds of change notifications, depending on the kind of data binding:

- Simple binding, in which a single control property is bound to a single instance of an object.
- List-based binding, which can include a single control property bound to the property of an item in a list or a control property bound to a list of objects.

Additionally, if you're creating Windows Forms controls that you want to use for data binding, you must apply the *PropertyName*Changed pattern to the controls. Applying pattern to the controls results in changes to the bound property of a control are propagated to the data source.

Change notification for simple binding

For simple binding, business objects must provide change notification when the value of a bound property changes. You can provide change notification by exposing a PropertyNameChanged event for each property of your business object. It also requires binding the business object to controls with the BindingSource or the preferred method in which your business object implements the INotifyPropertyChanged interface and raises a PropertyChanged event when the value of a property changes. When you use objects that implement the `INotifyPropertyChanged` interface, you don't have to use the `BindingSource` to bind the object to a control. But using the `BindingSource` is recommended.

Change notification for list-based binding

Windows Forms depends on a bound list to provide *property change* and *list change* information to bound controls. The *property change* is a list item property value change and *list change* is an item deleted or added to the list. Therefore, lists used for data binding must implement the IBindingList, which provides both types of change notification. The BindingList<T> is a generic implementation of `IBindingList` and is designed for use with Windows Forms data binding. You can create a `BindingList` that contains a business object type that implements INotifyPropertyChanged and the list will automatically convert the PropertyChanged events to ListChanged events. If the bound list isn't an `IBindingList`, you must bind the list of objects to Windows Forms controls by using the BindingSource component. The `BindingSource` component will provide property-to-list conversion similar to that of the `BindingList`. For more information, see How to: Raise Change Notifications Using a BindingSource and the INotifyPropertyChanged Interface.

Change notification for custom controls

Finally, from the control side you must expose a *PropertyName*Changed event for each property designed to be bound to data. The changes to the control property are then propagated to the bound data source. For more information, see Apply the PropertyNameChanged pattern.

Apply the PropertyNameChanged pattern

The following code example demonstrates how to apply the *PropertyName*Changed pattern to a custom control. Apply the pattern when you implement custom controls that are used with the Windows Forms data binding engine.

C#

```
// This class implements a simple user control
// that demonstrates how to apply the propertyNameChanged pattern.
[ComplexBindingProperties("DataSource", "DataMember")]
public class CustomerControl : UserControl
{
    private DataGridView dataGridView1;
    private Label label1;
    private DateTime lastUpdate = DateTime.Now;

    public EventHandler DataSourceChanged;

    public object DataSource
    {
        get
        {
```

```
            return this.dataGridView1.DataSource;
        }
        set
        {
            if (DataSource != value)
            {
                this.dataGridView1.DataSource = value;
                OnDataSourceChanged();
            }
        }
    }

    public string DataMember
    {
        get { return this.dataGridView1.DataMember; }

        set { this.dataGridView1.DataMember = value; }
    }

    private void OnDataSourceChanged()
    {
        if (DataSourceChanged != null)
        {
            DataSourceChanged(this, new EventArgs());
        }
    }

    public CustomerControl()
    {
        this.dataGridView1 = new System.Windows.Forms.DataGridView();
        this.label1 = new System.Windows.Forms.Label();
        this.dataGridView1.ColumnHeadersHeightSizeMode =

System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
        this.dataGridView1.ImeMode = System.Windows.Forms.ImeMode.Disable;
        this.dataGridView1.Location = new System.Drawing.Point(100, 100);
        this.dataGridView1.Size = new System.Drawing.Size(500,500);

        this.dataGridView1.TabIndex = 1;
        this.label1.AutoSize = true;
        this.label1.Location = new System.Drawing.Point(50, 50);
        this.label1.Name = "label1";
        this.label1.Size = new System.Drawing.Size(76, 13);
        this.label1.TabIndex = 2;
        this.label1.Text = "Customer List:";
        this.Controls.Add(this.label1);
        this.Controls.Add(this.dataGridView1);
        this.Size = new System.Drawing.Size(450, 250);
    }
}
```

Implement the INotifyPropertyChanged interface

The following code example demonstrates how to implement the INotifyPropertyChanged interface. Implement the interface on business objects that are used in Windows Forms data binding. When implemented, the interface communicates to a bound control the property changes on a business object.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Runtime.CompilerServices;
using System.Windows.Forms;

// Change the namespace to the project name.
namespace binding_control_example
{
    // This form demonstrates using a BindingSource to bind
    // a list to a DataGridView control. The list does not
    // raise change notifications. However the DemoCustomer1 type
    // in the list does.
    public partial class Form3 : Form
    {
        // This button causes the value of a list element to be changed.
        private Button changeItemBtn = new Button();

        // This DataGridView control displays the contents of the list.
        private DataGridView customersDataGridView = new DataGridView();

        // This BindingSource binds the list to the DataGridView control.
        private BindingSource customersBindingSource = new BindingSource();

        public Form3()
        {
            InitializeComponent();

            // Set up the "Change Item" button.
            this.changeItemBtn.Text = "Change Item";
            this.changeItemBtn.Dock = DockStyle.Bottom;
            this.changeItemBtn.Height = 100;
            //this.changeItemBtn.Click +=
              //  new EventHandler(changeItemBtn_Click);
            this.Controls.Add(this.changeItemBtn);

            // Set up the DataGridView.
            customersDataGridView.Dock = DockStyle.Top;
            this.Controls.Add(customersDataGridView);
```

```
            this.Size = new Size(400, 200);
        }

        private void Form3_Load(object sender, EventArgs e)
        {
            this.Top = 100;
            this.Left = 100;
            this.Height = 600;
            this.Width = 1000;

            // Create and populate the list of DemoCustomer objects
            // which will supply data to the DataGridView.
            BindingList<DemoCustomer1> customerList = new ();
            customerList.Add(DemoCustomer1.CreateNewCustomer());
            customerList.Add(DemoCustomer1.CreateNewCustomer());
            customerList.Add(DemoCustomer1.CreateNewCustomer());

            // Bind the list to the BindingSource.
            this.customersBindingSource.DataSource = customerList;

            // Attach the BindingSource to the DataGridView.
            this.customersDataGridView.DataSource =
                this.customersBindingSource;
        }

        // Change the value of the CompanyName property for the first
        // item in the list when the "Change Item" button is clicked.
        void changeItemBtn_Click(object sender, EventArgs e)
        {
            // Get a reference to the list from the BindingSource.
            BindingList<DemoCustomer1>? customerList =
                this.customersBindingSource.DataSource as
BindingList<DemoCustomer1>;

            // Change the value of the CompanyName property for the
            // first item in the list.
            customerList[0].CustomerName = "Tailspin Toys";
            customerList[0].PhoneNumber = "(708)555-0150";
        }

    }

    // This is a simple customer class that
    // implements the IPropertyChange interface.
    public class DemoCustomer1 : INotifyPropertyChanged
    {
        // These fields hold the values for the public properties.
        private Guid idValue = Guid.NewGuid();
        private string customerNameValue = String.Empty;
        private string phoneNumberValue = String.Empty;

        public event PropertyChangedEventHandler PropertyChanged;

        // This method is called by the Set accessor of each property.
        // The CallerMemberName attribute that is applied to the optional
```

```
propertyName
        // parameter causes the property name of the caller to be
substituted as an argument.
        private void NotifyPropertyChanged([CallerMemberName] String
propertyName = "")
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
            }
        }

        // The constructor is private to enforce the factory pattern.
        private DemoCustomer1()
        {
            customerNameValue = "Customer";
            phoneNumberValue = "(312)555-0100";
        }

        // This is the public factory method.
        public static DemoCustomer1 CreateNewCustomer()
        {
            return new DemoCustomer1();
        }

        // This property represents an ID, suitable
        // for use as a primary key in a database.
        public Guid ID
        {
            get
            {
                return this.idValue;
            }
        }

        public string CustomerName
        {
            get
            {
                return this.customerNameValue;
            }

            set
            {
                if (value != this.customerNameValue)
                {
                    this.customerNameValue = value;
                    NotifyPropertyChanged();
                }
            }
        }

        public string PhoneNumber
        {
```

```
            get
            {
                return this.phoneNumberValue;
            }

            set
            {
                if (value != this.phoneNumberValue)
                {
                    this.phoneNumberValue = value;
                    NotifyPropertyChanged();
                }
            }
        }
    }
}
```

Synchronize bindings

During implementation of data binding in Windows Forms, multiple controls are bound to the same data source. In some cases, it may be necessary to take extra steps to ensure that the bound properties of the controls remain synchronized with each other and the data source. These steps are necessary in two situations:

- If the data source doesn't implement IBindingList, and therefore generate ListChanged events of type ItemChanged.
- If the data source implements IEditableObject.

In the former case, you can use a BindingSource to bind the data source to the controls. In the latter case, you use a `BindingSource` and handle the BindingComplete event and call EndCurrentEdit on the associated BindingManagerBase.

For more information on implementing this concept, see the BindingComplete API reference page.

See also

- Data Binding
- BindingSource
- INotifyPropertyChanged
- BindingList<T>

Navigate data (Windows Forms .NET)

Article • 06/18/2022

The easiest way to navigate through records in a data source is to bind a BindingSource component to the data source and then bind controls to the `BindingSource`. You can then use the built-in navigation method of the `BindingSource`, such as MoveNext, MoveLast, MovePrevious, and MoveFirst. Using these methods will adjust the Position and Current properties of the `BindingSource` appropriately. You can also find a record and set it as the current record by setting the `Position` property.

To increment the record position in a data source

Set the Position property of the BindingSource for your bound data to the record position to go to the required record position. The following example illustrates using the MoveNext method of the `BindingSource` to increment the `Position` property when you select the `nextButton`. The `BindingSource` is associated with the `Customers` table of a dataset `Northwind`.

C#

```
private void nextButton_Click(object sender, System.EventArgs e)
{
    this.customersBindingSource.MoveNext();
}
```

> **Note**
>
> Setting the **Position** property to a value beyond the first or last record does not result in an error, as Windows Forms won't set the position to a value outside the bounds of the list. If it's important to know whether you have gone past the first or last record, include logic to test whether you'll exceed the data element count.

To check whether you've exceeded the first or last record

Create an event handler for the PositionChanged event. In the handler, you can test whether the proposed position value has exceeded the actual data element count.

The following example illustrates how you can test whether you've reached the last data element. In the example, if you are at the last element, the **Next** button on the form is disabled.

C#

```
void customersBindingSource_PositionChanged(object sender, EventArgs e)
{
    if (customersBindingSource.Position == customersBindingSource.Count - 1)
        nextButton.Enabled = false;
    else
        nextButton.Enabled = true;
}
```

> **Note**
>
> Be aware that, if you change the list you are navigating in code, you should re-enable the **Next** button so that users might browse the entire length of the new list. Additionally, be aware that the above **PositionChanged** event for the specific **BindingSource** you are working with needs to be associated with its event-handling method.

To find a record and set it as the current item

Find the record you wish to set as the current item. Use the Find method of the BindingSource as shown in the example, if your data source implements IBindingList. Some examples of data sources that implement `IBindingList` are BindingList<T> and DataView.

C#

```
void findButton_Click(object sender, EventArgs e)
{
    int foundIndex = customersBindingSource.Find("CustomerID", "ANTON");
    customersBindingSource.Position = foundIndex;
}
```

To ensure the selected row in a child table remains at the correct position

When you work with data binding in Windows Forms, you'll display data in a parent/child or master/detail view. It's a data-binding scenario where data from the same source is displayed in two controls. Changing the selection in one control causes the data displayed in the second control to change. For example, the first control might contain a list of customers and the second a list of orders related to the selected customer in the first control.

When you display data in a parent/child view, you might have to take extra steps to ensure that the currently selected row in the child table isn't reset to the first row of the table. In order to do this, you'll have to cache the child table position and reset it after the parent table changes. Typically, the child table reset occurs the first time a field in a row of the parent table changes.

To cache the current child table position

1. Declare an integer variable to store the child table position and a Boolean variable to store whether to cache the child table position.

   C#

   ```
   private int cachedPosition = -1;
   private bool cacheChildPosition = true;
   ```

2. Handle the ListChanged event for the binding's CurrencyManager and check for a ListChangedType of Reset.

3. Check the current position of the CurrencyManager. If it's greater than the first entry in the list (typically 0), save it to a variable.

   C#

   ```
   void relatedCM_ListChanged(object sender, ListChangedEventArgs e)
   {
       // Check to see if this is a caching situation.
       if (cacheChildPosition && cachePositionCheckBox.Checked)
       {
           // If so, check to see if it is a reset situation, and the
   current
           // position is greater than zero.
           CurrencyManager relatedCM = sender as CurrencyManager;
           if (e.ListChangedType == ListChangedType.Reset &&
   relatedCM.Position > 0)

               // If so, cache the position of the child table.
               cachedPosition = relatedCM.Position;
   ```

```
    }
}
```

4. Handle the parent list's CurrentChanged event for the parent currency manager. In the handler, set the Boolean value to indicate it isn't a caching scenario. If the `CurrentChanged` occurs, the change to the parent is a list position change and not an item value change.

```
C#

void bindingSource1_CurrentChanged(object sender, EventArgs e)
{
    // If the CurrentChanged event occurs, this is not a caching
    // situation.
    cacheChildPosition = false;
}
```

To reset the child table position

1. Handle the PositionChanged event for the child table binding's CurrencyManager.

2. Reset the child table position to the cached position saved in the previous procedure.

```
C#

void relatedCM_PositionChanged(object sender, EventArgs e)
{
    // Check to see if this is a caching situation.
    if (cacheChildPosition && cachePositionCheckBox.Checked)
    {
        CurrencyManager relatedCM = sender as CurrencyManager;

        // If so, check to see if the current position is
        // not equal to the cached position and the cached
        // position is not out of bounds.
        if (relatedCM.Position != cachedPosition && cachedPosition
            > 0 && cachedPosition < relatedCM.Count)
        {
            relatedCM.Position = cachedPosition;
            cachedPosition = -1;
        }
    }
}
```

To test the code example, perform the following steps:

1. Run the example.
2. Ensure the **Cache and reset position** checkbox is selected.
3. Select the **Clear parent field** button to cause a change in a field of the parent table. Notice that the selected row in the child table doesn't change.
4. Close and rerun the example. You need to run it again because the reset behavior occurs only on the first change in the parent row.
5. Clear the **Cache and reset position** checkbox.
6. Select the **Clear parent field** button. Notice that the selected row in the child table changes to the first row.

See also

- Data binding overview
- Data sources supported by Windows Forms
- Change Notification in Windows Forms Data Binding
- How to synchronize multiple controls to the same data source
- BindingSource Component

Create a simple-bound control (Windows Forms .NET)

Article • 06/18/2022

With simple data binding, you can display a single data element, such as a column value from a dataset table to a control on a form. You can simple-bind any property of a control to a data value.

To simple-bind a control

1. Connect to a data source.
2. In Visual Studio, select the control on the form and display the **Properties** window.
3. Expand the **DataBindings** property.

   The properties that are bound are displayed under the **DataBindings** property. For example, in most controls, the **Text** property is frequently bound.
4. If the property you want to bind isn't one of the commonly bound properties, select the **Ellipsis** button () in the **Advanced** box to display the **Formatting and Advanced Binding** dialog with a complete list of properties for that control.
5. Select the property you want to bind and select the drop-down arrow under **Binding**. A list of available data sources is displayed.
6. Expand the data source you want to bind to until you find the single data element you want. For example, if you're binding to a column value in a dataset table, expand the name of the dataset, and then expand the table name to display column names.
7. Select the name of an element to bind to.
8. If you're working in the **Formatting and Advanced Binding** dialog, select **OK** to return to the **Properties** window.
9. If you want to bind more properties of the control, repeat steps 3 to 7.

> **Note**
>
> As simple-bound controls show only a single data element, it's typical to include navigation logic in a Windows Form with simple-bound controls.

To create a bound control and format the displayed data

With Windows Forms data binding, you can format the data displayed in a data-bound control by using the **Formatting and Advanced Binding** dialog.

1. Connect to a data source.
2. In Visual Studio, select the control on the form and then open the **Properties** window.
3. Expand the **DataBindings** property, and then in the **Advanced** box, select the ellipsis button () to display the **Formatting and Advanced Binding** dialog, which has a complete list of properties for that control.
4. Select the property you want to bind, and then select the **Binding** arrow.

   A list of available data sources is displayed.
5. Expand the data source you want to bind the property to until you find the single data element you want.

   For example, if you're binding to a column value in a dataset's table, expand the name of the dataset, and then expand the table name to display column names.
6. Select the name of an element to bind to.
7. In the **Format type** box, select the format you want to apply to the data displayed in the control.

   In every case, you can specify the value displayed in the control if the data source contains DBNull. Otherwise, the options vary slightly, depending on the format type you select. The following table shows the format types and options.

Format type	Formatting option
No Formatting	No options.
Numeric	Specify number of decimal places by using **Decimal places** up-down control.
Currency	Specify the number of decimal places by using **Decimal places** up-down control.

Format type	Formatting option
Date Time	Choose how the date and time should be displayed by selecting one of the items in the **Type** selection box.
Scientific	Specify the number of decimal places by using **Decimal places** up-down control.
Custom	Specify a custom format string. For more information, see Formatting Types. **Note:** Custom format strings aren't guaranteed to successfully round trip between the data source and bound control. Instead, handle the Parse or Format event for the binding and apply custom formatting in the event-handling code.

8. Select **OK** to close the **Formatting and Advanced Binding** dialog and return to the **Properties** window.

See also

- Binding
- Data Binding
- User Input Validation in Windows Forms

Synchronize multiple controls to the same data source (Windows Forms .NET)

Article • 06/18/2022

During the implementation of data binding in Windows Forms, multiple controls are bound to the same data source. In the following situations, it's necessary to ensure that the bound properties of the control remain synchronized with each other and the data source:

- If the data source doesn't implement IBindingList, and therefore generates ListChanged events of type ItemChanged.
- If the data source implements IEditableObject.

In the former case, you can use a BindingSource to bind the data source to the controls. In the latter case, you use a `BindingSource` and handle the BindingComplete event and call EndCurrentEdit on the associated BindingManagerBase.

Example of bind controls using BindingSource

The following code example demonstrates how to bind three controls, two textbox controls, and a DataGridView control to the same column in a DataSet using a BindingSource component. The example demonstrates how to handle the BindingComplete event. It ensures that when the text value of one textbox is changed, the other textbox and the `DataGridView` control are updated with the correct value.

The example uses a BindingSource to bind the data source and the controls. Alternatively, you can bind the controls directly to the data source and retrieve the BindingManagerBase for the binding from the form's BindingContext and then handle the BindingComplete event for the `BindingManagerBase`. For more information on binding the data source and the controls, see the help page about the `BindingComplete` event of `BindingManagerBase`.

C#

```
public Form1()
{
    InitializeComponent();
    set1.Tables.Add("Menu");
    set1.Tables[0].Columns.Add("Beverages");
```

```
    // Add some rows to the table.
    set1.Tables[0].Rows.Add("coffee");
    set1.Tables[0].Rows.Add("tea");
    set1.Tables[0].Rows.Add("hot chocolate");
    set1.Tables[0].Rows.Add("milk");
    set1.Tables[0].Rows.Add("orange juice");

    // Set the data source to the DataSet.
    bindingSource1.DataSource = set1;

    //Set the DataMember to the Menu table.
    bindingSource1.DataMember = "Menu";

    // Add the control data bindings.
    dataGridView1.DataSource = bindingSource1;
    textBox1.DataBindings.Add("Text", bindingSource1,
        "Beverages", true, DataSourceUpdateMode.OnPropertyChanged);
    textBox2.DataBindings.Add("Text", bindingSource1,
        "Beverages", true, DataSourceUpdateMode.OnPropertyChanged);
    bindingSource1.BindingComplete +=
        new BindingCompleteEventHandler(bindingSource1_BindingComplete);
}

void bindingSource1_BindingComplete(object sender, BindingCompleteEventArgs
e)
{
    // Check if the data source has been updated, and that no error has
occurred.
    if (e.BindingCompleteContext ==
        BindingCompleteContext.DataSourceUpdate && e.Exception == null)

        // If not, end the current edit.
        e.Binding.BindingManagerBase.EndCurrentEdit();
}
```

See also

- Design great data sources with change notification
- Data Binding
- How to: Share Bound Data Across Forms Using the BindingSource Component

WFAC001: Only projects with 'OutputType=WindowsApplication' supported

Article • 11/23/2023

Only projects with `OutputType` set to `Exe` or `WinExe` are supported, because only application projects define an application entry point, where the application bootstrap code must reside.

How to fix

Remove the call to `ApplicationConfiguration.Initialize` or change the project type to an executable.

WFAC002: Unsupported property value

Article • 11/23/2023

The related application configuration values defined in the project file are invalid. The following snippet demonstrates valid values:

XML

```
<PropertyGroup>

  <ApplicationVisualStyles>true</ApplicationVisualStyles>

<ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe
ndering>
  <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
  <ApplicationDefaultFont>Microsoft Sans Serif,
8.25pt</ApplicationDefaultFont>

</PropertyGroup>
```

How to fix

Change the invalid setting to a valid value.

Obsolete Windows Forms features in .NET 7+

Article • 06/02/2023

Starting in .NET 7, some Windows Forms APIs are marked as obsolete (or otherwise produce a warning) with custom diagnostic IDs of the format `WFDEVXXX`.

If you encounter build warnings or errors due to usage of an obsolete API, follow the specific guidance provided for the diagnostic ID listed in the Reference section. Warnings or errors for these obsoletions *can't* be suppressed using the standard diagnostic ID (CS0618) for obsolete types or members; use the custom `WFDEVXXX` diagnostic ID values instead. For more information, see Suppress warnings.

Reference

The following table provides an index to the `WFDEVXXX` obsoletions and warnings in .NET 7+.

Diagnostic ID	Warning or error	Description
WFDEV001	Warning	Casting to/from IntPtr is unsafe. Use `WParamInternal`, `LParamInternal`, or `ResultInternal` instead.
WFDEV002	Warning/error	System.Windows.Forms.DomainUpDown.DomainUpDownAccessibleObject is no longer used to provide accessible support for DomainUpDown controls. Use AccessibleObject instead.
WFDEV003	Warning	System.Windows.Forms.DomainUpDown.DomainItemAccessibleObject is no longer used to provide accessible support for DomainUpDown items. Use AccessibleObject instead.

Suppress warnings

It's recommended that you use an available workaround whenever possible. However, if you cannot change your code, you can suppress warnings through a `#pragma` directive or a `<NoWarn>` project setting. If you must use the obsolete APIs and the `WFDEVXXX` diagnostic does not surface as an error, you can suppress the warning in code or in your project file.

To suppress the warnings in code:

C#

```
// Disable the warning.
#pragma warning disable WFDEV001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore WFDEV001
```

To suppress the warnings in a project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net7.0</TargetFramework>
   <!-- NoWarn below suppresses WFDEV001 project-wide -->
   <NoWarn>$(NoWarn);WFDEV001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements
-->
   <NoWarn>$(NoWarn);WFDEV001</NoWarn>
   <NoWarn>$(NoWarn);WFDEV003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a
semicolon-delimited list -->
   <NoWarn>$(NoWarn);WFDEV001;WFDEV003</NoWarn>
  </PropertyGroup>
</Project>
```

> **Note**
>
> Suppressing warnings in this way only disables the obsoletion warnings you specify. It doesn't disable any other warnings, including obsoletion warnings with different diagnostic IDs.

See also

- Obsolete .NET features in .NET 5+

WFAC010: Unsupported high DPI configuration.

Article • 11/23/2023

Windows Forms applications should specify application DPI-awareness via the application configuration or with the Application.SetHighDpiMode API.

Workarounds

Using C#

Use the new bootstrap API by calling the `ApplicationConfiguration.Initialize` method before Application.Run().

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
    }
}
```

The `ApplicationConfiguration.Initialize` method is generated when your app compiles, based on the settings in the app's project file. For example, look at the following `<Application*>` settings:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWindowsForms>true</UseWindowsForms>
    <ImplicitUsings>enable</ImplicitUsings>

    <ApplicationVisualStyles>true</ApplicationVisualStyles>

<ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe
```

```
ndering>
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
    <ApplicationDefaultFont>Microsoft Sans Serif,
8.25pt</ApplicationDefaultFont>

  </PropertyGroup>

</Project>
```

These settings generate the following method:

C#

```
[CompilerGenerated]
internal static partial class ApplicationConfiguration
{
    public static void Initialize()
    {
        global::System.Windows.Forms.Application.EnableVisualStyles();

global::System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(f
alse);

global::System.Windows.Forms.Application.SetHighDpiMode(HighDpiMode.SystemAw
are);
        global::System.Windows.Forms.Application.SetDefaultFont(new Font(new
FontFamily("Microsoft Sans Serif"), 8.25f, (FontStyle)0, (GraphicsUnit)3));
    }
}
```

Using Visual Basic

Visual Basic operates a little differently than C# at the moment. The project file settings are required for Visual Studio to detect the application settings, but you must also configure the settings in the project's property page **Application** > **Application Framework** (which affects the *My Project\Application.myapp* file) or in the application startup events.

> **ⓘ Important**
>
> Font isn't settable in the project properties.

The following code example demonstrates handling the ApplyApplicationDefaults event to configure the default font and HighDPI mode:

VB

```
Imports Microsoft.VisualBasic.ApplicationServices

Namespace My
    Partial Friend Class MyApplication
        Private Sub MyApplication_ApplyApplicationDefaults(sender As Object,
e As ApplyApplicationDefaultsEventArgs) Handles Me.ApplyApplicationDefaults
            e.Font = New Font("Microsoft Sans Serif", 8.25)
            e.HighDpiMode = HighDpiMode.SystemAware
        End Sub
    End Class
End Namespace
```

Suppress a warning

If you must use the obsolete APIs, you can suppress the warning in code or in your project file.

To suppress only a single violation, add preprocessor directives to your source file to disable and then re-enable the warning.

C#

```
// Disable the warning.
#pragma warning disable WFAC010

// Code that uses the API.
// ...

// Re-enable the warning.
#pragma warning restore WFAC010
```

To suppress all the `WFAC010` warnings in your project, add a `<NoWarn>` property to your project file.

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   ...
   <NoWarn>$(NoWarn);WFAC010</NoWarn>
  </PropertyGroup>
</Project>
```

For more information, see Suppress warnings.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

- Open a documentation issue
- Provide product feedback

WFDEV001: WParam, LParam, and Message.Result are obsolete

Article • 09/12/2022

To reduce the risk of cast and overflow exceptions associated with IntPtr on different platforms, the Windows Forms SDK disallows direct use of Message.WParam, Message.LParam, and Message.Result. Projects that use the `DEBUG` build of the Windows Forms SDK and that reference WParam, LParam, or Result will fail to compile due to warning `WFDEV001`.

Workarounds

Update your code to use the new internal properties, either `WParamInternal`, `LParamInternal`, or `ResultInternal` depending on the situation.

Suppress a warning

If you must use the obsolete APIs, you can suppress the warning in code or in your project file.

To suppress only a single violation, add preprocessor directives to your source file to disable and then re-enable the warning.

C#

```
// Disable the warning.
#pragma warning disable WFDEV001

// Code that uses obsolete API.
// ...

// Re-enable the warning.
#pragma warning restore WFDEV001
```

To suppress all the `WFDEV001` warnings in your project, add a `<NoWarn>` property to your project file.

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
```

```
    <NoWarn>$(NoWarn);WFDEV001</NoWarn>
  </PropertyGroup>
</Project>
```

For more information, see Suppress warnings.

WFDEV002: DomainUpDownAccessibleObject should not be used

Article • 02/06/2023

Any reference to System.Windows.Forms.DomainUpDown.DomainUpDownAccessibleObject will result in warning `WFDEV002`. This warning states that DomainUpDown.DomainUpDownAccessibleObject is no longer used to provide accessible support for DomainUpDown controls. The DomainUpDown.DomainUpDownAccessibleObject type was never intended for public use.

> **Note**
>
> This warning was promoted to an error starting in .NET 8, and you can no longer suppress the error. For more information, see **WFDEV002 obsoletion is now an error**.

Workarounds

- Update your code to use AccessibleObject instead of DomainUpDown.DomainUpDownAccessibleObject.
- If you're using .NET 7, you can suppress the warning and your code will continue to compile and run.

Suppress a warning (.NET 7 only)

If you must use the obsolete API, you can suppress the warning in code or in your project file.

To suppress only a single violation, add preprocessor directives to your source file to disable and then re-enable the warning.

C#

```
// Disable the warning.
#pragma warning disable WFDEV002
```

```
// Code that uses obsolete API.
// ...

// Re-enable the warning.
#pragma warning restore WFDEV002
```

To suppress all the `WFDEV002` warnings in your project, add a `<NoWarn>` property to your project file.

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   ...
   <NoWarn>$(NoWarn);WFDEV002</NoWarn>
  </PropertyGroup>
</Project>
```

For more information, see Suppress warnings.

WFDEV003: DomainItemAccessibleObject should not be used

Article • 09/12/2022

Any reference to System.Windows.Forms.DomainUpDown.DomainItemAccessibleObject will result in warning `WFDEV003`. This warning states that DomainUpDown.DomainItemAccessibleObject is no longer used to provide accessible support for items in DomainUpDown controls. This type was never intended for public use.

Previously, objects of this type were served to accessibility tools that navigated the hierarchy of a DomainUpDown control. In .NET 7 and later versions, instances of type AccessibleObject are used to represent items in a DomainUpDown control for accessibility tools.

Workarounds

Remove invocations of the public constructor for the DomainUpDown.DomainItemAccessibleObject type. Use System.Windows.Forms.AccessibleObject instead.

Suppress a warning

If you must use the obsolete APIs, you can suppress the warning in code or in your project file.

To suppress only a single violation, add preprocessor directives to your source file to disable and then re-enable the warning.

C#

```
// Disable the warning.
#pragma warning disable WFDEV003

// Code that uses obsolete API.
// ...

// Re-enable the warning.
#pragma warning restore WFDEV003
```

To suppress all the `WFDEV003` warnings in your project, add a `<NoWarn>` property to your project file.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   ...
   <NoWarn>$(NoWarn);WFDEV003</NoWarn>
  </PropertyGroup>
</Project>
```

For more information, see Suppress warnings.