

# Contents

[Windows Forms for .NET](#)

[What's new](#)

[Get started](#)

[Overview](#)

[Create an app](#)

[Migration](#)

[Migrate to .NET 5](#)

[Forms](#)

[Event handlers](#)

[Automatic scaling](#)

[Common tasks](#)

[Add a form](#)

[Resize a form](#)

[Position a form](#)

[Controls](#)

[Overview](#)

[Layout options](#)

[Labels](#)

[Events](#)

[Custom controls](#)

[Custom painting and drawing](#)

[Apply accessibility information](#)

[Common tasks](#)

[Add a control to a form](#)

[Create access key shortcuts](#)

[Set the text displayed by a control](#)

[Set the the tab order of a control](#)

[Dock and anchor controls](#)

[Set the image displayed by a control](#)

- Add or remove event handlers

- Make thread-safe calls to controls

## User input - keyboard

- Overview

- Use keyboard events

- Validate input

- Common tasks

- Change the pressed key

- Determine which modifier key is pressed

- Handle input at the form level

- Simulate keyboard events

## User input - mouse

- Overview

- Use mouse events

- Drag-and-drop functionality

- Common tasks

- Distinguish between clicks and double-clicks

- Control and modify the mouse pointer

- Simulate mouse events

# What's new (Windows Forms .NET)

3/9/2021 • 2 minutes to read • [Edit Online](#)

Windows Forms for .NET 5.0 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 5.0. 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)

3/9/2021 • 6 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

There are two implementations of Windows Forms:

1. The open-source implementation hosted on [GitHub](#).

This version runs on .NET 5 and .NET Core 3.1. The Windows Forms Visual Designer requires, at a minimum, [Visual Studio 2019 version 16.8 Preview](#).

2. The .NET Framework 4 implementation that's supported by 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. 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 \(Windows Forms .NET\)](#).

## Why migrate from .NET Framework

Windows Forms for .NET 5.0 provides new features and enhancements over .NET Framework. For more information, see [What's new in Windows Forms for .NET 5](#). To learn how to migrate an app, see [How to migrate a Windows Forms desktop app to .NET 5](#).

## 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 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 \(Windows Forms .NET\)](#)
- [How to add a form to a project \(Windows Forms .NET\)](#)
- [Add a control \(Windows Forms .NET\)](#)

# Tutorial: Create a new WinForms app (Windows Forms .NET)

3/9/2021 • 4 minutes to read • [Edit Online](#)

In this short tutorial, you'll learn how to create a new Windows Forms (WinForms) 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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

In this tutorial, you learn how to:

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

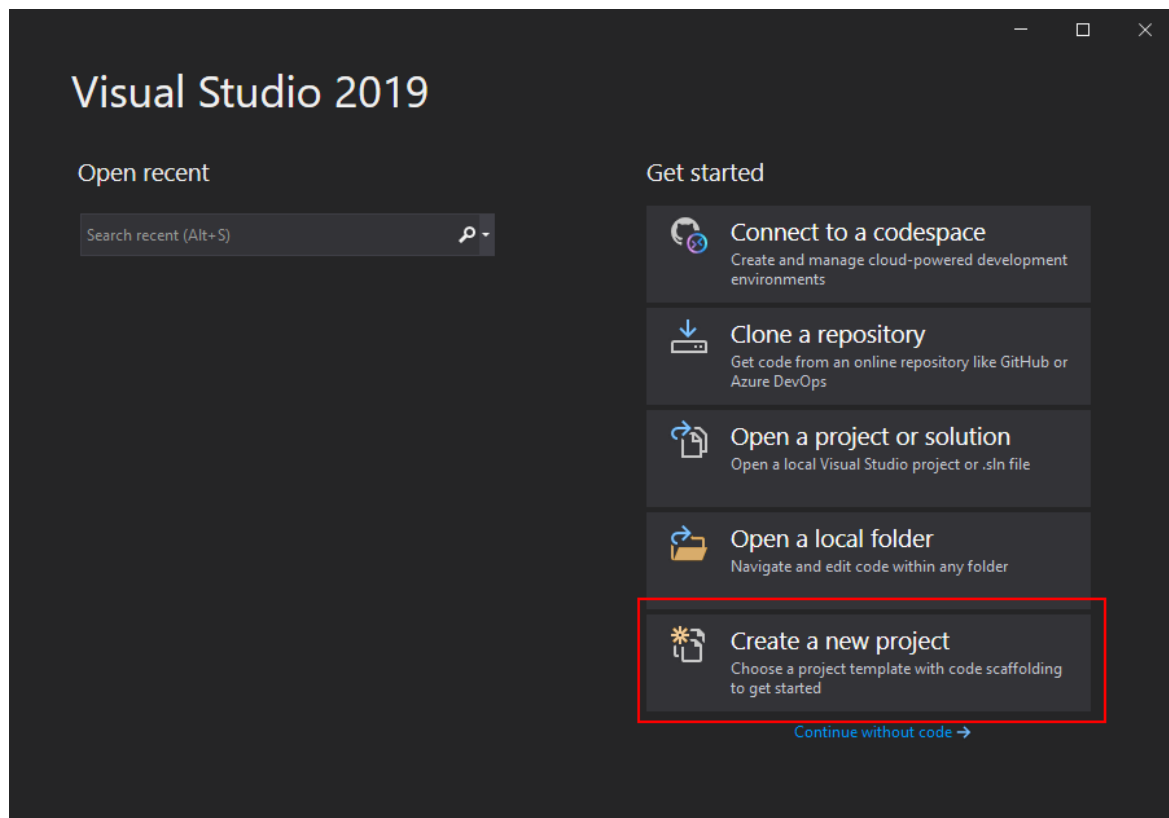
## Prerequisites

- [Visual Studio 2019 version 16.8 or later versions](#)
  - Select the [Visual Studio Desktop workload](#)
  - Select the [.NET 5 individual component](#)

## Create a WinForms app

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

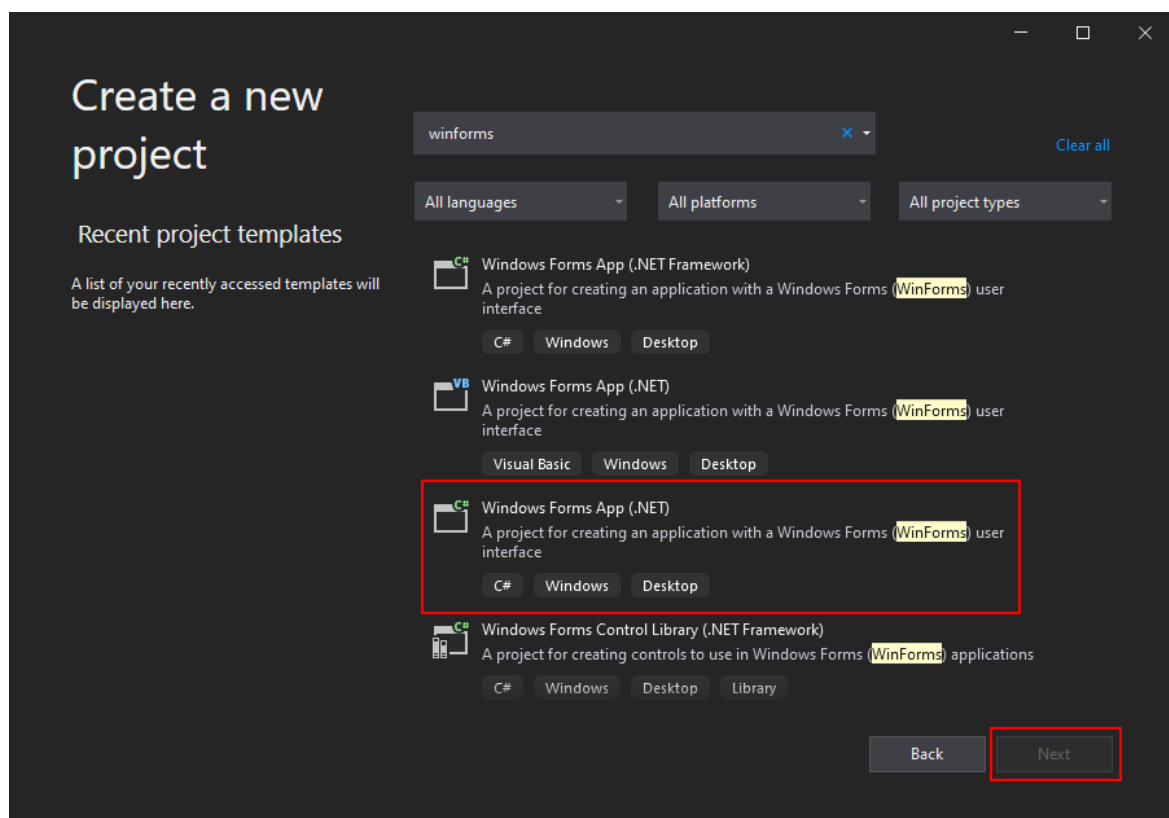
1. Open Visual Studio.
2. Select **Create a new project**.



3. In the **Search for templates** box, type **winforms**, and then press **Enter**.
4. In the **code language** dropdown, choose **C#** or **Visual Basic**.
5. In the templates list, select **Windows Forms App (.NET)** and then click **Next**.

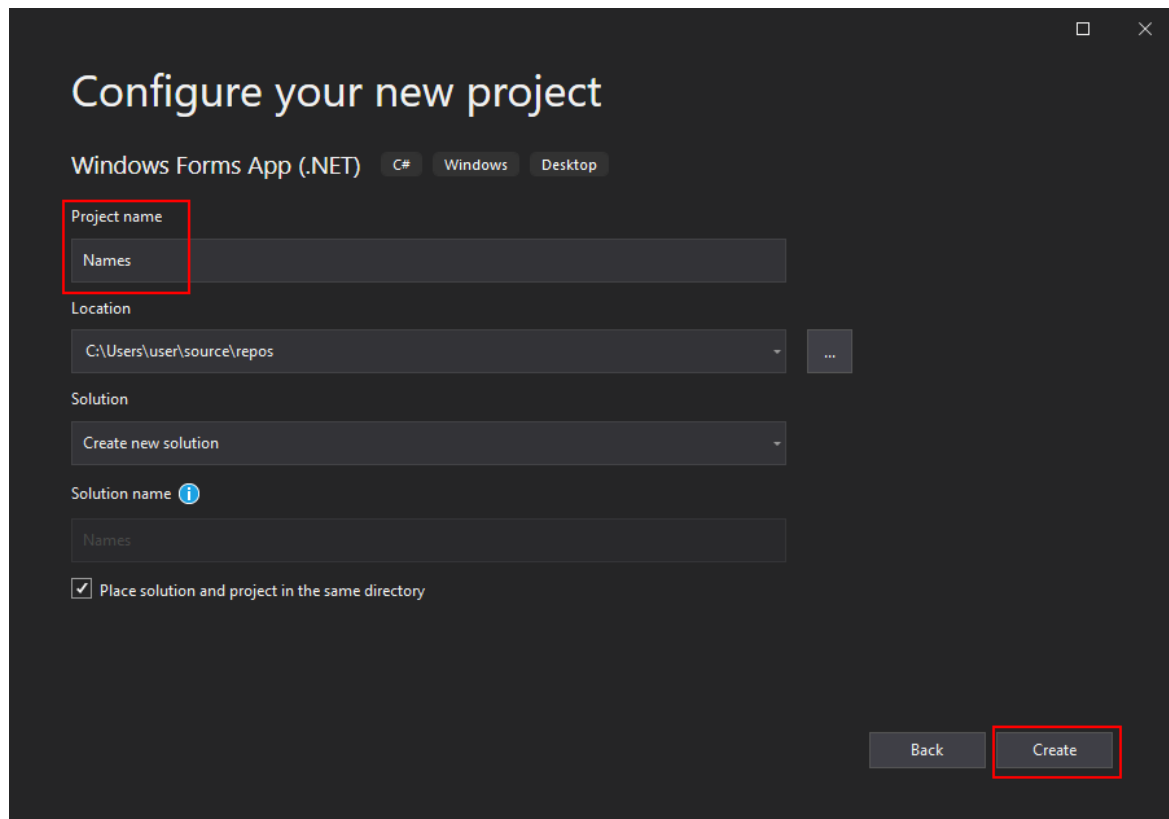
#### IMPORTANT

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



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

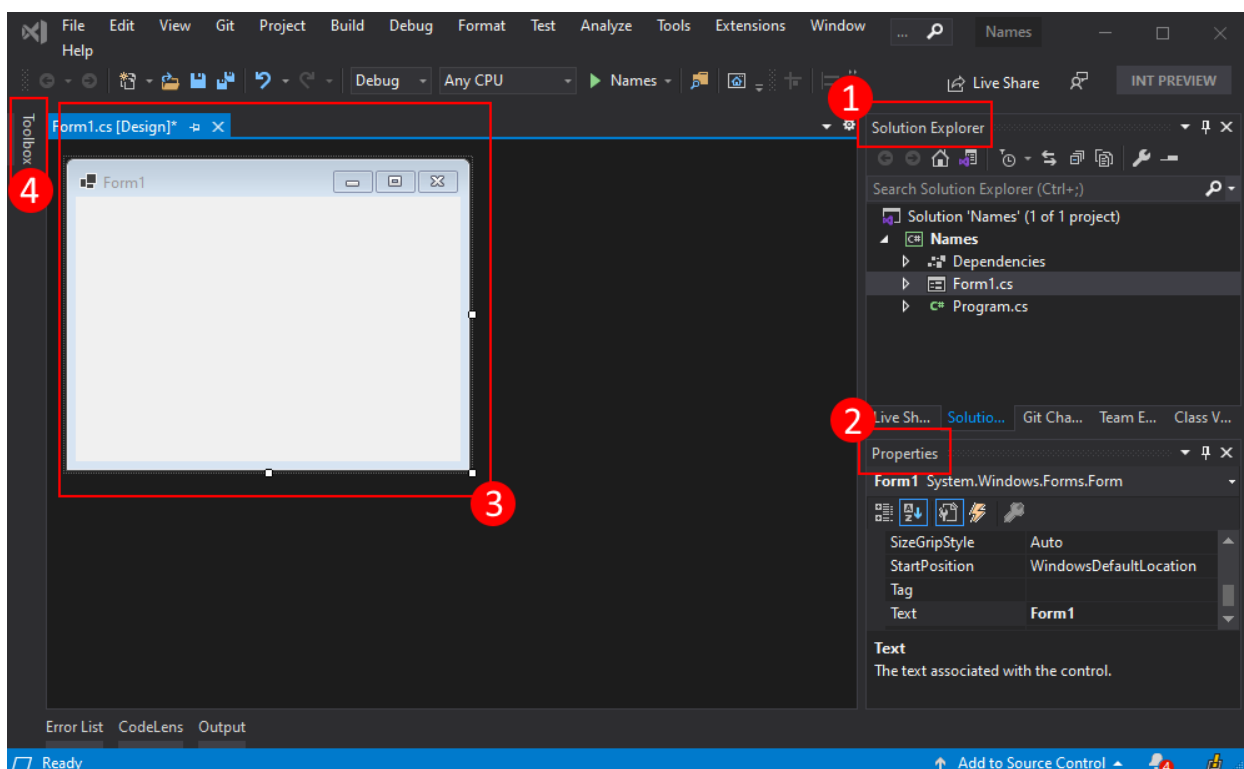
You can also save your project to a different folder by adjusting the **Location** setting.



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 WinForms in Visual Studio has four important components that you'll interact with as you create an app:



1. Solution Explorer

All if 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


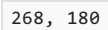


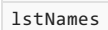




The toolbox contains all of the controls you can add to a form. To add a control to the current form, double-click a control or drag-and-drop the control.

# Add controls to the form

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

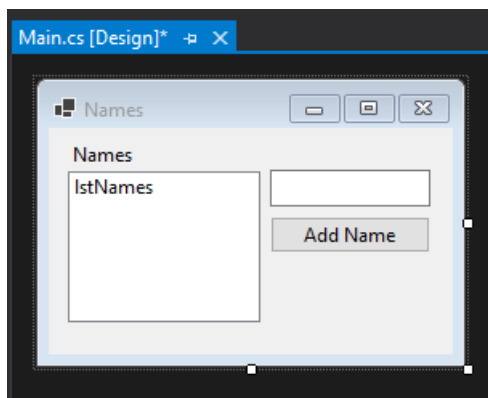
- Label
- Button
- Listbox
- Textbox

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:

OBJECT	SETTING	VALUE
Form	Text	
	Size	
Label	Location	
	Text	
Listbox	Name	
	Location	
	Size	
Textbox	Name	
	Location	


OBJECT	SETTING	VALUE
	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:

```
private void btnAdd_Click(object sender, EventArgs e)
{
}

```

```
Private Sub btnAdd_Click(sender As Object, e As EventArgs) Handles btnAdd.Click
End Sub

```

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.

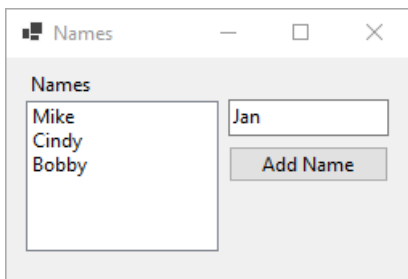
4. The following code demonstrates adding a name to the `lstNames` control:

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

```
Private Sub btnAdd_Click(sender As Object, e As EventArgs) Handles btnAdd.Click
    If Not String.IsNullOrEmpty(txtName.Text) And Not lstNames.Items.Contains(txtName.Text) Then
        lstNames.Items.Add(txtName.Text)
    End If
End Sub
```

## 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](#)

# How to migrate a Windows Forms desktop app to .NET 5

8/12/2021 • 9 minutes to read • [Edit Online](#)

This article describes how to migrate a Windows Forms desktop app from .NET Framework to .NET 5 or later. The .NET SDK includes support for Windows Forms applications. Windows Forms is still a Windows-only framework and only runs on Windows.

Migrating your app from .NET Framework to .NET 5 generally requires a new project file. .NET 5 uses SDK-style project files while .NET Framework typically uses the older Visual Studio project file. If you've ever opened a Visual Studio project file in a text editor, you know how verbose it is. SDK-style projects are smaller and don't require as many entries as the older project file format does.

To learn more about .NET 5, see [Introduction to .NET](#).

## Try the upgrade assistant

The .NET Upgrade Assistant is a command-line tool that can be run on different kinds of .NET Framework apps. It's designed to assist with upgrading .NET Framework apps to .NET 5. After running the tool, in most cases the app will require additional effort to complete the migration. The tool includes the installation of analyzers that can assist with completing the migration.

For more information, see [Upgrade a WPF App to .NET 5 with the .NET Upgrade Assistant](#).

## Prerequisites

- [Visual Studio 2019 version 16.8 Preview](#)
  - Select the [Visual Studio Desktop workload](#).
  - Select the [.NET 5 individual component](#).
- Preview WinForms designer in Visual Studio.

To enable the designer, go to **Tools > Options > Environment > Preview Features** and select the **Use the preview Windows Forms designer for .NET Core apps** option.

- This article uses the [Matching game](#) sample app. If you want to follow along, download and open the application in Visual Studio. Otherwise, use your own app.

### Consider

When migrating a .NET Framework Windows Forms application, there are a few things you must consider.

1. Check that your application is a good candidate for migration.

Use the [.NET Portability Analyzer](#) to determine if your project will migrate to .NET 5. If your project has issues with .NET 5, the analyzer helps you identify those problems. The .NET Portability Analyzer tool can be installed as a Visual Studio extension or used from the command line. For more information, see [.NET Portability Analyzer](#).

2. You're using a different version of Windows Forms.

When .NET Core 3.0 was released, Windows Forms went [open source on GitHub](#). The code for Windows Forms for .NET 5 is a fork of the .NET Framework Windows Forms codebase. It's possible some

differences exist and your app will be difficult to migrate.

3. The [Windows Compatibility Pack](#) may help you migrate.

Some APIs that are available in .NET Framework aren't available in .NET 5. The [Windows Compatibility Pack](#) adds many of these APIs and may help your Windows Forms app become compatible with .NET 5.

4. Update the NuGet packages used by your project.

It's always a good practice to use the latest versions of NuGet packages before any migration. If your application is referencing any NuGet packages, update them to the latest version. Ensure your application builds successfully. After upgrading, if there are any package errors, downgrade the package to the latest version that doesn't break your code.

## Back up your projects

The first step to migrating a project is to back up your project! If something goes wrong, you can restore your code to its original state by restoring your backup. Don't rely on tools such as the .NET Portability Analyzer to back up your project, even if they seem to. It's best to personally create a copy of the original project.

## NuGet packages

If your project is referencing NuGet packages, you probably have a **packages.config** file in your project folder. With SDK-style projects, NuGet package references are configured in the project file. Visual Studio project files can optionally define NuGet packages in the project file too. .NET 5 doesn't use **packages.config** for NuGet packages. NuGet package references must be migrated into the project file before migration.

To migrate the **packages.config** file, do the following steps:

1. In **Solution explorer**, find the project you're migrating.
2. Right-click on **packages.config** > **Migrate packages.config to PackageReference**.
3. Select all of the top-level packages.

A build report is generated to let you know of any issues migrating the NuGet packages.

## Project file

The next step in migrating your app is converting the project file. As previously stated, .NET 5 uses SDK-style project files and won't load the Visual Studio project files that .NET Framework uses. However, there's the possibility that you're already using SDK-style projects. You can easily spot the difference in Visual Studio. Right-click on the project file in **Solution explorer** and look for the **Edit Project File** menu option. If this menu item is missing, you're using the old Visual Studio project format and need to upgrade.

Convert each project in your solution. If you're using the sample app previously referenced, both the **MatchingGame** and **MatchingGame.Logic** projects would be converted.

To convert a project, do the following steps:

1. In **Solution explorer**, find the project you're migrating.
2. Right-click on the project and select **Unload Project**.
3. Right-click on the project and select **Edit Project File**.
4. Copy-and-paste the project XML into a text editor. You'll want a copy so that it's easy to move content into the new project.
5. Erase the content of the file and paste the following XML:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
    <GenerateAssemblyInfo>>false</GenerateAssemblyInfo>
  </PropertyGroup>

</Project>
```

### IMPORTANT

Libraries don't need to define an `<OutputType>` setting. Remove that entry if you're upgrading a library project.

This XML gives you the basic structure of the project. However, it doesn't contain any of the settings from the old project file. Using the old project information you previously copied to a text editor, do the following steps:

1. Copy the following elements from the old project file into the `<PropertyGroup>` element in the new project file:

- `<RootNamespace>`
- `<AssemblyName>`

Your project file should look similar to the following XML:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
    <GenerateAssemblyInfo>>false</GenerateAssemblyInfo>

    <RootNamespace>MatchingGame</RootNamespace>
    <AssemblyName>MatchingGame</AssemblyName>
  </PropertyGroup>

</Project>
```

2. Copy the `<ItemGroup>` elements from the old project file that contain `<ProjectReference>` or `<PackageReference>` into the new file after the `</PropertyGroup>` closing tag.

Your project file should look similar to the following XML:

```

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

  <PropertyGroup>
    (contains settings previously described)
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\MatchingGame.Logic\MatchingGame.Logic.csproj">
      <Project>{36b3e6e2-a9ae-4924-89ae-7f0120ce08bd}</Project>
      <Name>MatchingGame.Logic</Name>
    </ProjectReference>
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="MetroFramework">
      <Version>1.2.0.3</Version>
    </PackageReference>
  </ItemGroup>

</Project>

```

The `<ProjectReference>` elements don't need the `<Project>` and `<Name>` children, so you can remove those settings:

```

<ItemGroup>
  <ProjectReference Include="..\MatchingGame.Logic\MatchingGame.Logic.csproj" />
</ItemGroup>

```

## Resources and settings

One thing to note about the difference between .NET Framework projects and the SDK-style projects used by .NET 5 is that .NET Framework projects use an opt-in model for code files. Any code file you want to compile needs to be explicitly defined in your project file. SDK-style projects are reverse, they default to opt-out behavior: All code files starting from the project's directory and below are automatically included in your project. You don't need to migrate these entries if they are simple and without settings. This is the same for other common files such as *resx*.

Windows Forms projects may also reference the following files:

- *Properties\Settings.settings*
- *Properties\Resources.resx*
- *Properties\app.manifest*

The *app.manifest* file is automatically referenced by your project and you don't need to do anything special to migrate it.

Any *\*.resx* and *\*.settings* files in the *Properties* folder need to be migrated in the project. Copy those entries from the old project file into an `<ItemGroup>` element in the new project. After you copy the entries, change all `<Compile Include="value">` elements to instead use the `Update` attribute instead of `Include`.

- Import the configuration for the *Settings.settings* file.

```
<ItemGroup>
  <None Update="Properties\Settings.settings">
    <Generator>SettingsSingleFileGenerator</Generator>
    <LastGenOutput>Settings.Designer.cs</LastGenOutput>
  </None>
  <Compile Update="Properties\Settings.Designer.cs">
    <AutoGen>True</AutoGen>
    <DependentUpon>Settings.settings</DependentUpon>
    <DesignTimeSharedInput>True</DesignTimeSharedInput>
  </Compile>
</ItemGroup>
```

### IMPORTANT

**Visual Basic** projects typically use the folder *My Project* while C# projects typically use the folder *Properties* for the default project settings file.

- Import the configuration for any *resx* file, such as the *properties\Resources.resx* file. Notice that the `Include` attribute was set to `Update` on the `<Compile>` and `<EmbeddedResource>` element, and `<SubType>` was removed from `<EmbeddedResource>` :

```
<ItemGroup>
  <EmbeddedResource Update="Properties\Resources.resx">
    <Generator>ResXFileCodeGenerator</Generator>
    <LastGenOutput>Resources.Designer.cs</LastGenOutput>
  </EmbeddedResource>
  <Compile Update="Properties\Resources.Designer.cs">
    <AutoGen>True</AutoGen>
    <DependentUpon>Resources.resx</DependentUpon>
    <DesignTime>True</DesignTime>
  </Compile>
</ItemGroup>
```

### IMPORTANT

**Visual Basic** projects typically use the folder *My Project* while C# projects typically use the folder *Properties* for the default project resource file.

## Visual Basic

Visual Basic language projects require extra configuration.

- Import the configuration file *My Project\Application.myapp* setting. Notice that the `<Compile>` element uses the `Update` attribute instead of the `Include` attribute.

```
<ItemGroup>
  <None Include="My Project\Application.myapp">
    <Generator>MyApplicationCodeGenerator</Generator>
    <LastGenOutput>Application.Designer.vb</LastGenOutput>
  </None>
  <Compile Update="My Project\Application.Designer.vb">
    <AutoGen>True</AutoGen>
    <DependentUpon>Application.myapp</DependentUpon>
    <DesignTime>True</DesignTime>
  </Compile>
</ItemGroup>
```

2. Add the `<MyType>WindowsForms</MyType>` setting to the `<PropertyGroup>` element:

```
<PropertyGroup>
  (contains settings previously described)

  <MyType>WindowsForms</MyType>
</PropertyGroup>
```

This setting imports the `My` namespace members Visual Basic programmers are familiar with.

3. Import the namespaces defined by your project.

Visual Basic projects can automatically import namespaces into every code file. Copy the `<ItemGroup>` elements from the old project file that contain `<Import>` into the new file after the `</PropertyGroup>` closing tag.

```
<ItemGroup>
  <Import Include="Microsoft.VisualBasic" />
  <Import Include="System" />
  <Import Include="System.Collections" />
  <Import Include="System.Collections.Generic" />
  <Import Include="System.Data" />
  <Import Include="System.Drawing" />
  <Import Include="System.Diagnostics" />
  <Import Include="System.Windows.Forms" />
  <Import Include="System.Linq" />
  <Import Include="System.Xml.Linq" />
  <Import Include="System.Threading.Tasks" />
</ItemGroup>
```

If you can't find any `<Import>` statements, or your project fails to compile, make sure you at least have the following `<Import>` statements defined in your project:

```
<ItemGroup>
  <Import Include="System.Data" />
  <Import Include="System.Drawing" />
  <Import Include="System.Windows.Forms" />
</ItemGroup>
```

4. From the original project, copy the `<Option*>` and `<StartupObject>` settings to the `<PropertyGroup>` element:

```
<PropertyGroup>
  (contains settings previously described)

  <OptionExplicit>On</OptionExplicit>
  <OptionCompare>Binary</OptionCompare>
  <OptionStrict>Off</OptionStrict>
  <OptionInfer>On</OptionInfer>
  <StartupObject>MatchingGame.My.MyApplication</StartupObject>
</PropertyGroup>
```

## Reload the project

After you convert a project to the new SDK-style format, reload the project in Visual Studio:

1. In **Solution Explorer**, find the project you converted.
2. Right-click on the project and select **Reload Project**.

If the project fails to load, you may have introduced a mistake in the XML of the project. Open the project file for editing and try to identify and fix the mistake. If you can't find a mistake, try starting over.

## Edit App.config

If your app has an *App.config* file, remove the `<supportedRuntime>` element:

```
<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
```

There are some things you should consider with the *App.config* file. The *App.config* file in .NET Framework was used not only to configure the app, but to configure runtime settings and behavior, such as logging. The *App.config* file in .NET 5+ (and .NET Core) is no longer used for runtime configuration. If your *App.config* file has these sections, they won't be respected.

## Add the compatibility package

If your project file is loading correctly, but compilation fails for your project and you receive errors similar to the following:

- The type or namespace `<some name>` could not be found
- The name `<some name>` does not exist in the current context

You may need to add the `Microsoft.Windows.Compatibility` package to your app. This package adds ~21,000 .NET APIs from .NET Framework, such as the `System.Configuration.ConfigurationManager` class and APIs for interacting with the Windows Registry. Add the `Microsoft.Windows.Compatibility` package.

Edit your project file and add the following `<ItemGroup>` element:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Windows.Compatibility" Version="5.0.0" />
</ItemGroup>
```

## Test your app

After you've finished migrating your app, test it!

## Next steps

- Try the [.NET Upgrade Assistant](#) to migrate your app.
- Learn about [breaking changes in Windows Forms](#).
- Read more about the [Windows Compatibility Pack](#).

# Events overview (Windows Forms .NET)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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)

11/3/2020 • 3 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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 through 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 its 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)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

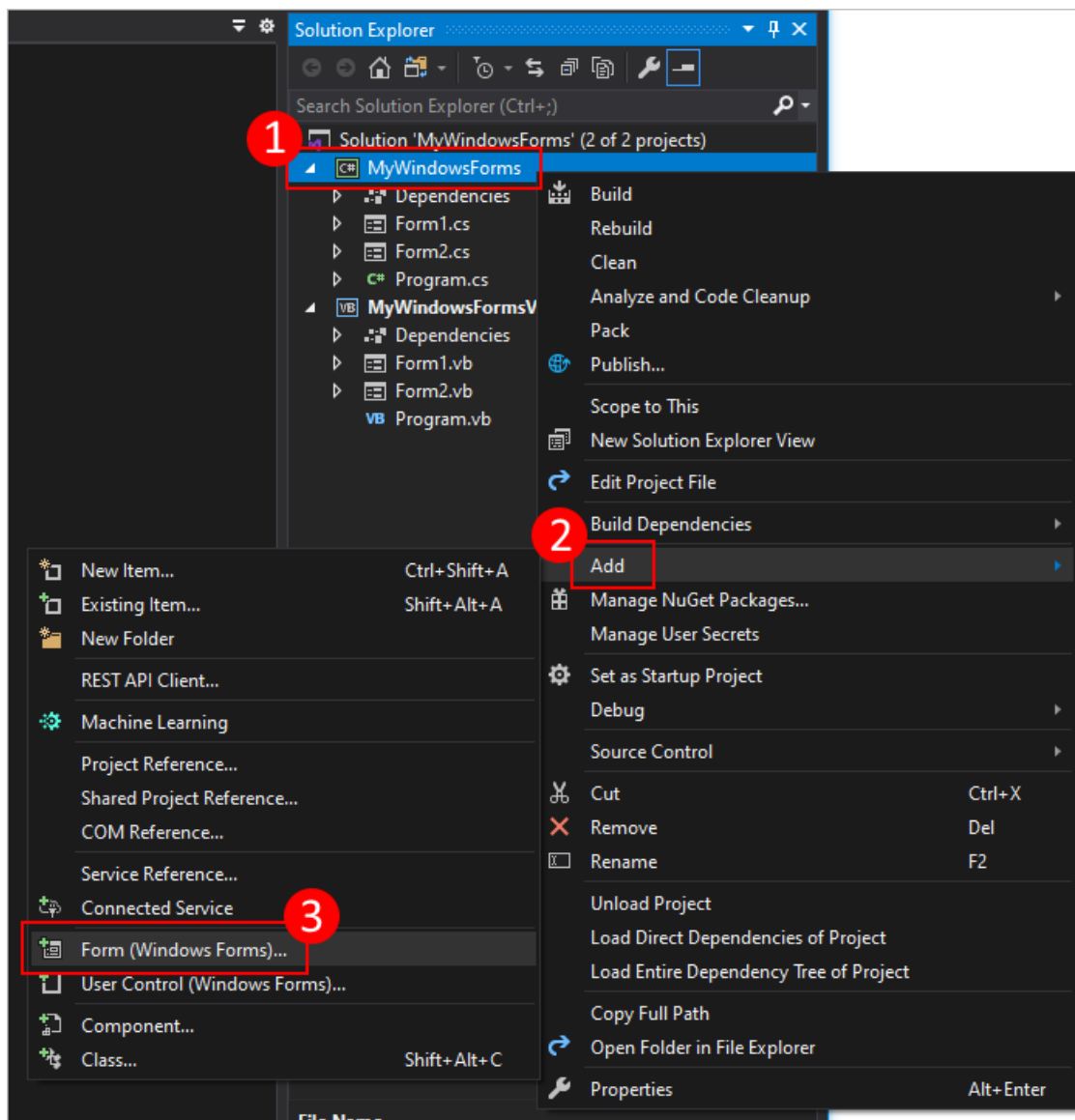
## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## Add a new form

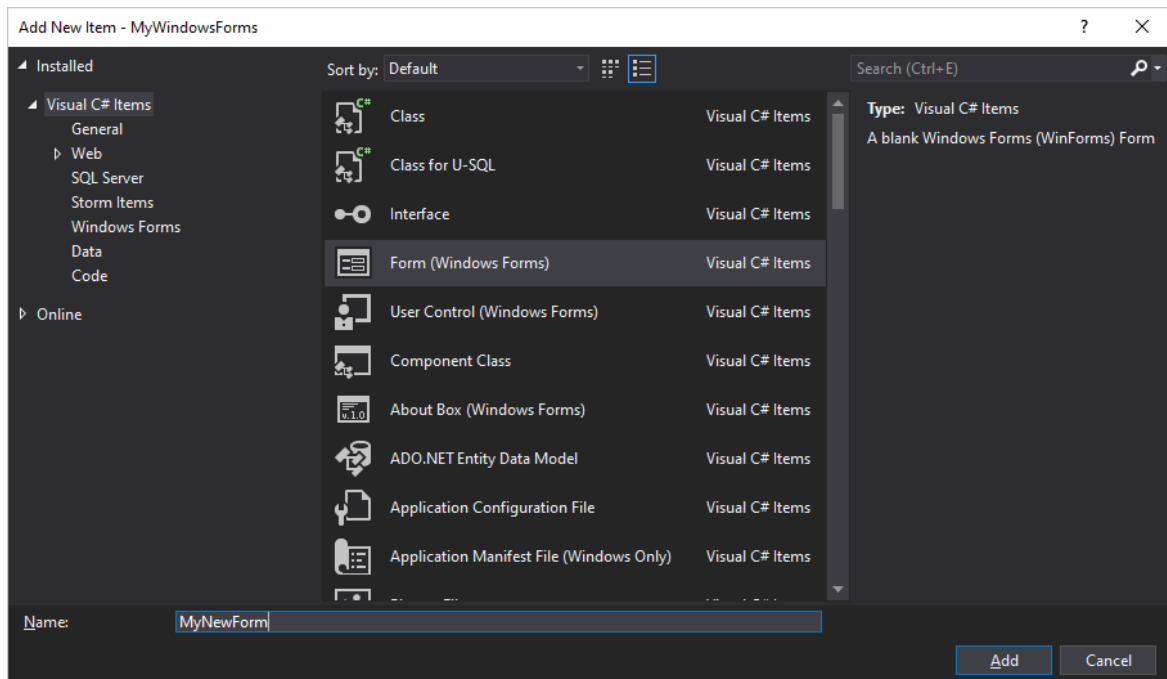
Add a new form with Visual Studio.

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.

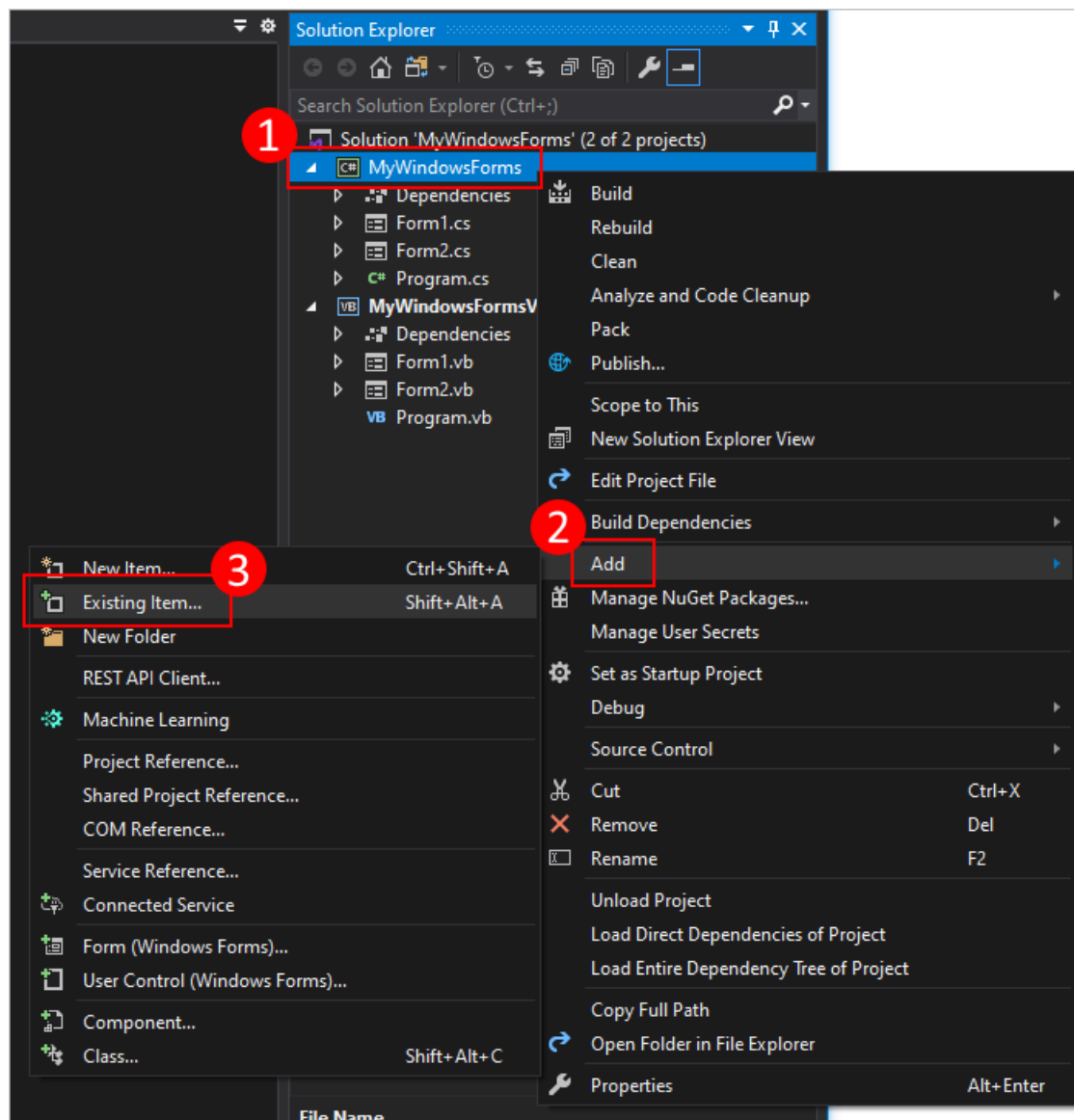
## Add a project reference to a form

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 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 \(Windows Forms .NET\)](#)

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

11/3/2020 • 4 minutes to read • [Edit Online](#)

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.

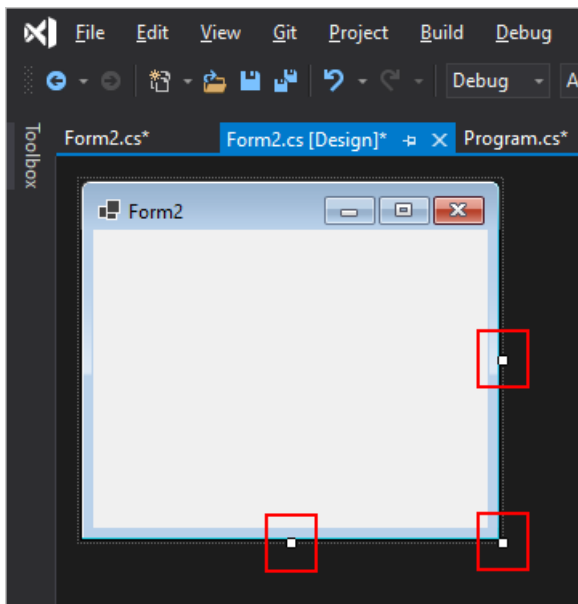
You can change the size of a form at design time with Visual Studio, and at run time with code.

## IMPORTANT

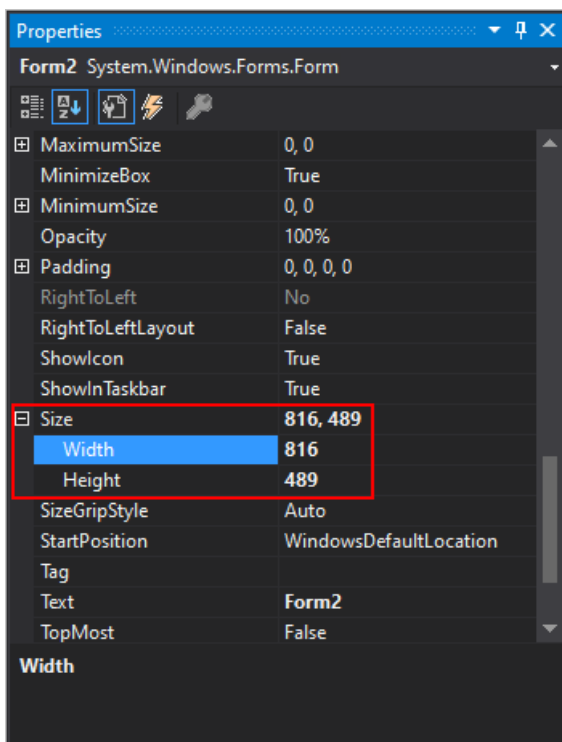
The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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 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:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Size = New Drawing.Size(250, 200)
End Sub
```

### 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:

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

```

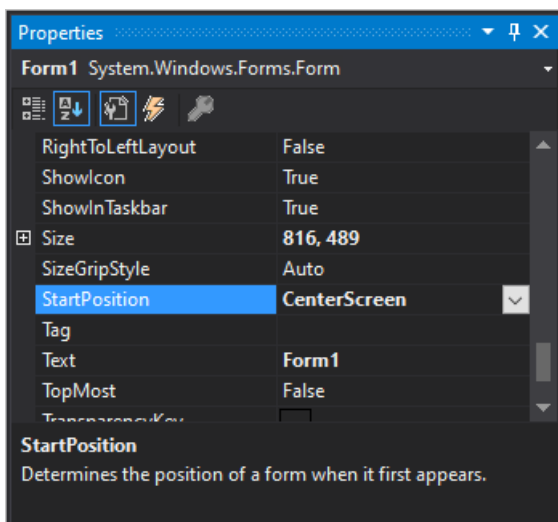
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Dim form = New Form2 With {
        .Size = New Drawing.Size(250, 200)
    }
    form.Show()
End Sub

```

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.



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 <code>Location</code> 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 effect 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:

```

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();

```

```
form.StartPosition = Windows.Forms.FormStartPosition.CenterScreen.Manual
form.Location = New Drawing.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:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Location = New Drawing.Point(0, 0)
End Sub
```

### 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 size:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Dim form = New Form2 With {
        .Size = New Drawing.Size(250, 200)
    }
    form.Show()
End Sub
```

If the `Size` isn't set, the form's default size is what it 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)

11/3/2020 • 4 minutes to read • [Edit Online](#)

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.

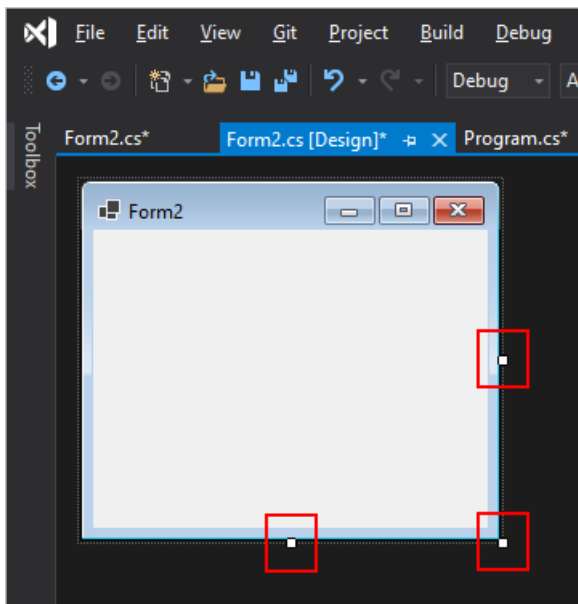
You can change the size of a form at design time with Visual Studio, and at run time with code.

## IMPORTANT

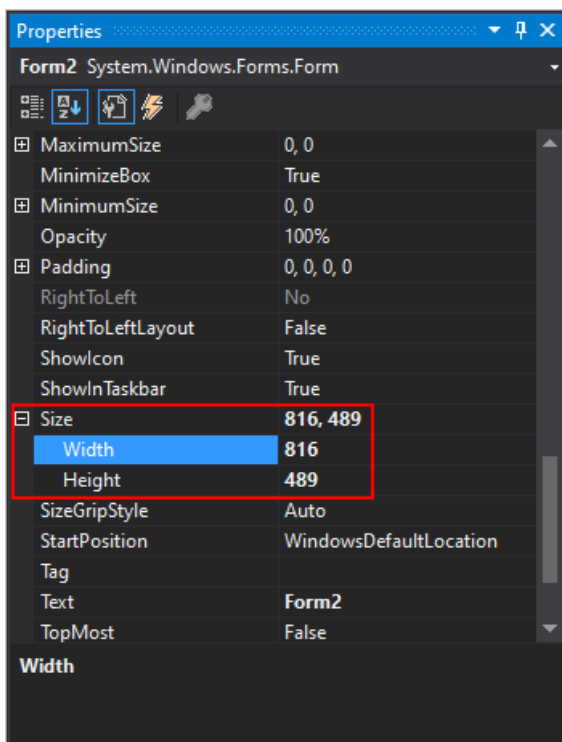
The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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 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:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Size = New Drawing.Size(250, 200)
End Sub
```

### 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:

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

```

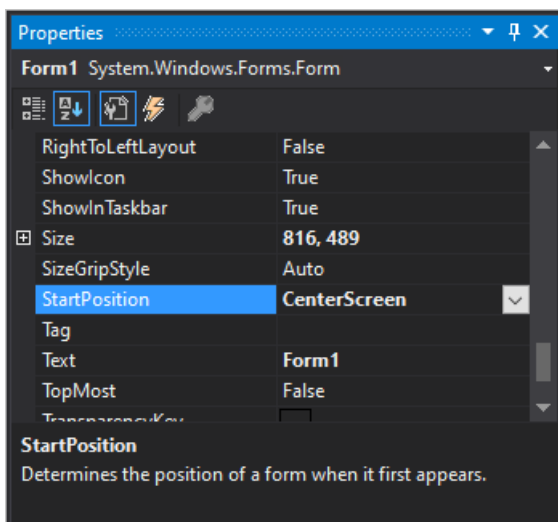
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Dim form = New Form2 With {
        .Size = New Drawing.Size(250, 200)
    }
    form.Show()
End Sub

```

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.



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 <code>Location</code> 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 effect 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:

```

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();

```

```
form.StartPosition = Windows.Forms.FormStartPosition.CenterScreen.Manual
form.Location = New Drawing.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:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Location = New Drawing.Point(0, 0)
End Sub
```

### 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 size:

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    Dim form = New Form2 With {
        .Size = New Drawing.Size(250, 200)
    }
    form.Show()
End Sub
```

If the `Size` isn't set, the form's default size is what it 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)

7/30/2021 • 2 minutes to read • [Edit Online](#)

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](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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 provide a set of common events through the base class: [Control](#). Not every control responds to every event. For example, the [Label](#) control doesn't respond to keyboard input, so the [Control.PreviewKeyDown](#) event isn't raised. Most shared events fall under these categories:

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

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)

7/30/2021 • 12 minutes to read • [Edit Online](#)

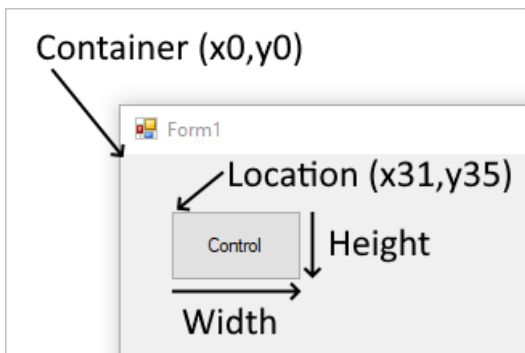
Control placement in Windows Forms is determined not only by the control, but also by the parent of the control. This article describes the different settings provided by controls and the different types of parent containers that affect layout.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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  $(x_0, y_0)$ . 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.

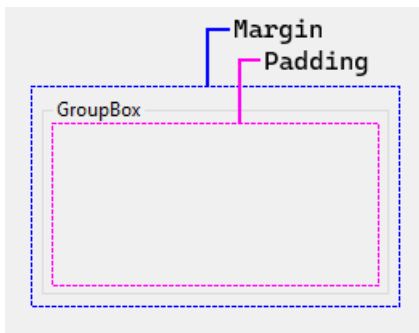
## Margin and Padding

There are two control properties that help with precise placement of controls: [Margin](#) and [Padding](#).

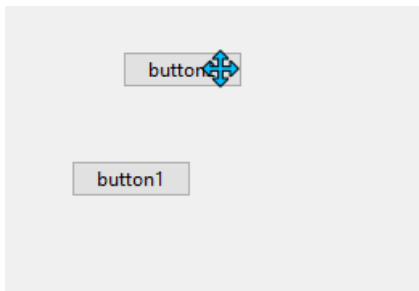
The [Margin](#) property defines the space around the control that keeps other controls a specified distance from the control's borders.

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

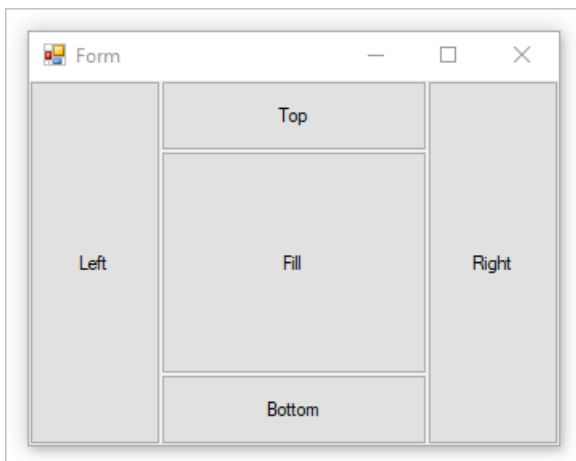
Controls can be automatically placed within their parent. Some parent containers force placement while others respect control settings that guide placement. There are two properties on a control that help automatic placement and size within a parent: [Dock](#) and [Anchor](#).

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.

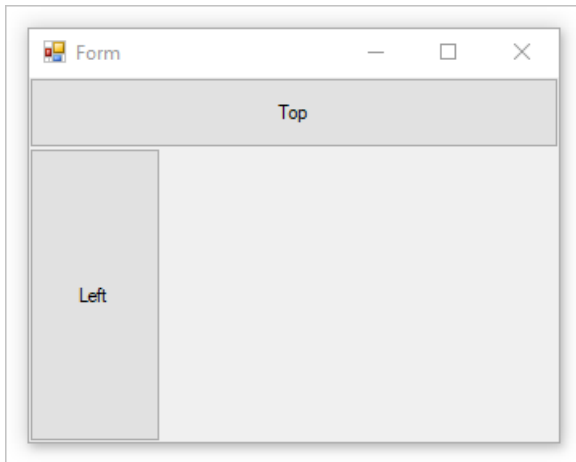


When a control is docked, the container determines the space it should occupy and resizes and places the control. The width and height of the control are still respected based on the docking style. For example, if the control is docked to the top, the [Height](#) of the control is respected but the [Width](#)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

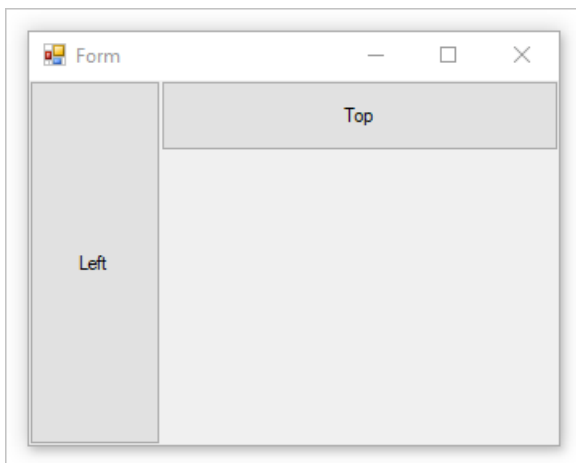
control is docked to the left, the **Width** of the control is respected but the **Height** is automatically adjusted.

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

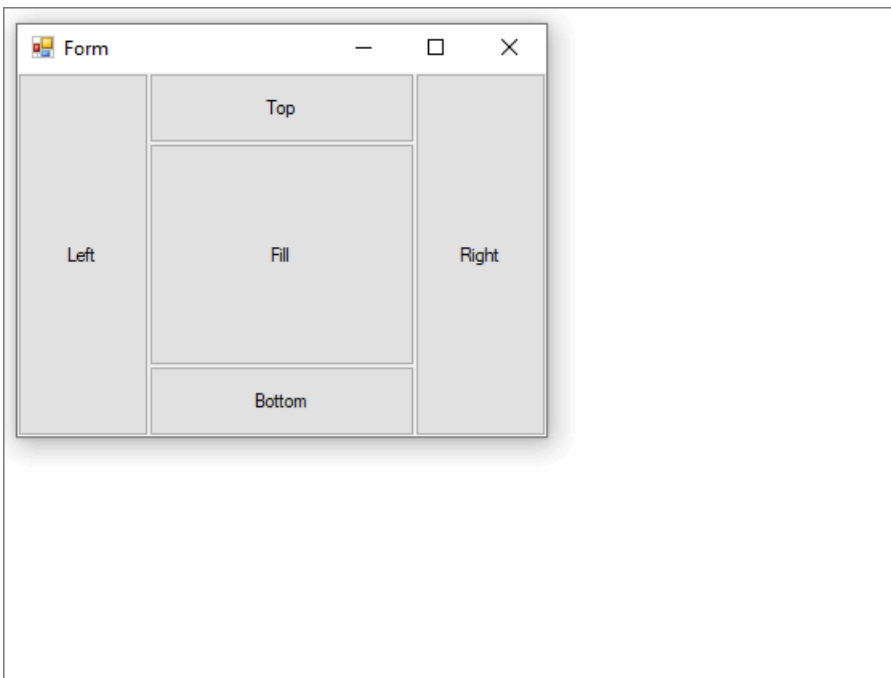
The **Z-order** of the control does affect docking. As docked controls are laid out, they use what space is available to them. For example, if a control is drawn first and docked to the top, it will take up the entire width of the container. If a next control is docked to the left, it has less vertical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



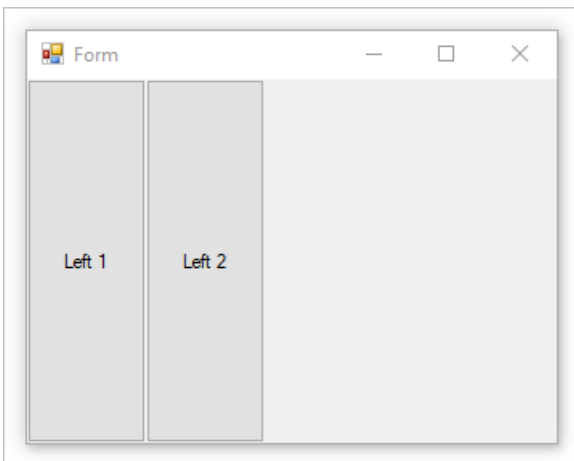
If the control's **Z-order** is reversed, the control that is docked to the left now has more initial space available. The control uses the entire height of the container. The control that is docked to the top has less horizontal space available to it.



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



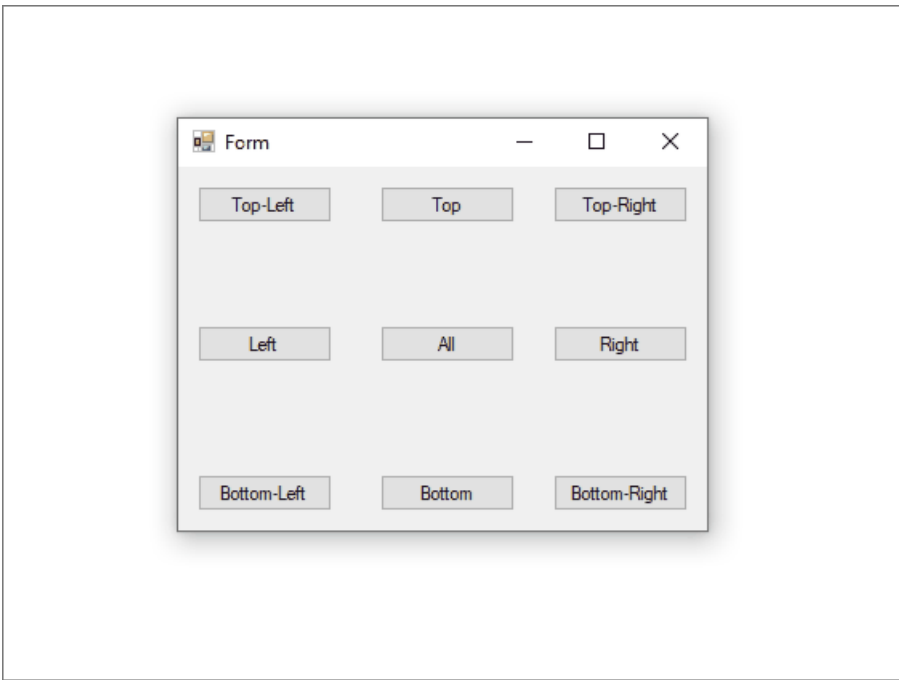
If multiple controls are docked to the same side of the container, they're stacked according to their **Z-order**.



## 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.

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 <a href="#">Location</a> 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 <a href="#">Dock</a> and <a href="#">Anchor</a> properties are honored when <a href="#">AutoSize</a> is <code>true</code> .	<p>The value of the control's <a href="#">Location</a> property is adjusted to the correct value.</p> <p>The <a href="#">Label</a> control is the exception to this rule. When you set the value of a docked <a href="#">Label</a> control's <a href="#">AutoSize</a> property to <code>true</code>, the <a href="#">Label</a> control will not stretch.</p>
A control's <a href="#">MaximumSize</a> and <a href="#">MinimumSize</a> properties are always honored, regardless of the value of its <a href="#">AutoSize</a> property.	The <a href="#">MaximumSize</a> and <a href="#">MinimumSize</a> properties are not affected by the <a href="#">AutoSize</a> property.
There is no minimum size set by default.	This means that if a control is set to shrink under <a href="#">AutoSize</a> and it has no contents, the value of its <a href="#">Size</a> property is <code>(0x,0y)</code> . In this case, your control will shrink to a point, and it will not be readily visible.
If a control does not implement the <a href="#">GetPreferredSize</a> method, the <a href="#">GetPreferredSize</a> method returns last value assigned to the <a href="#">Size</a> property.	This means that setting <a href="#">AutoSize</a> to <code>true</code> will have no effect.

ALWAYS TRUE BEHAVIOR	DESCRIPTION
A control in a <a href="#">TableLayoutPanel</a> cell always shrinks to fit in the cell until its <a href="#">MinimumSize</a> is reached.	This size is enforced as a maximum size. This is not the case when the cell is part of an <a href="#">AutoSize</a> 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:

CONTROL	<code>AUTOSIZE</code> SUPPORTED	<code>AUTOSIZEMODE</code> SUPPORTED
<a href="#">Button</a>	✓	✓
<a href="#">CheckedListBox</a>	✓	✓
<a href="#">FlowLayoutPanel</a>	✓	✓
<a href="#">Form</a>	✓	✓
<a href="#">GroupBox</a>	✓	✓
<a href="#">Panel</a>	✓	✓
<a href="#">TableLayoutPanel</a>	✓	✓
<a href="#">CheckBox</a>	✓	✗
<a href="#">DomainUpDown</a>	✓	✗
<a href="#">Label</a>	✓	✗
<a href="#">LinkLabel</a>	✓	✗
<a href="#">MaskedTextBox</a>	✓	✗

CONTROL	<small>AUTOSIZE</small> SUPPORTED	<small>AUTOSIZEMODE</small> SUPPORTED
NumericUpDown	✓	✗
RadioButton	✓	✗
TextBox	✓	✗
TrackBar	✓	✗
CheckedListBox	✗	✗
ComboBox	✗	✗
DataGridView	✗	✗
DateTimePicker	✗	✗
ListBox	✗	✗
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](#).

AUTO SIZE   SETTING	AUTOSIZEMODE   SETTING	BEHAVIOR
true	Property not available.	The user can't resize the control at design time, except for the following controls:  - <a href="#">TextBox</a> - <a href="#">MaskedTextBox</a> - <a href="#">RichTextBox</a> - <a href="#">TrackBar</a>
true	<a href="#">GrowAndShrink</a>	The user can't resize the control at design time.
true	<a href="#">GrowOnly</a>	The user can resize the control at design time. When the <a href="#">Size</a> property is set, the user can only increase the size of the control.
false or <a href="#">AutoSize</a> 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.

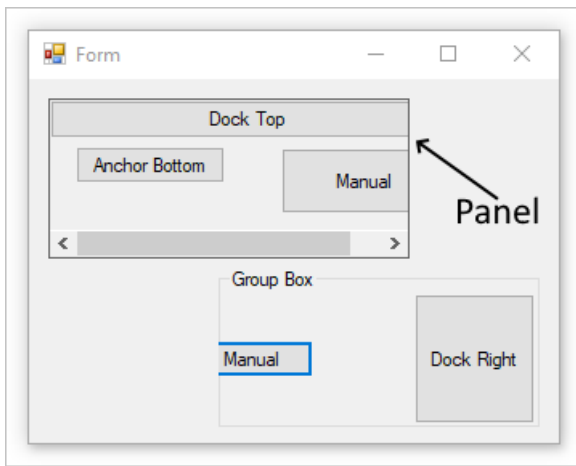
Most of the time a form will have grips on the edges that allow the user to resize the form. The [Anchor](#) property of a control will let the control grow and shrink as the form is resized.

## 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.



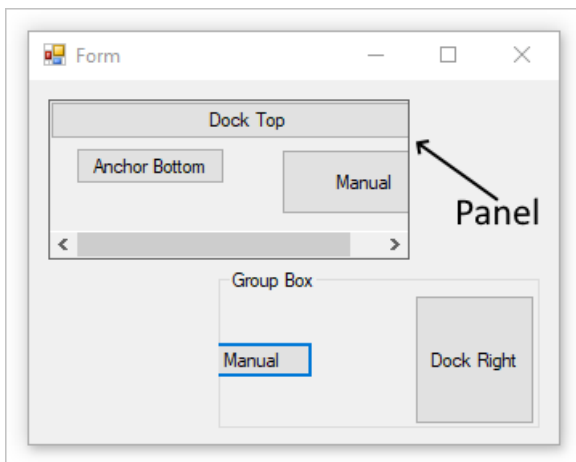
The image above has a panel with the [BorderStyle](#) property set to demonstrate the bounds of the panel.

## 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.

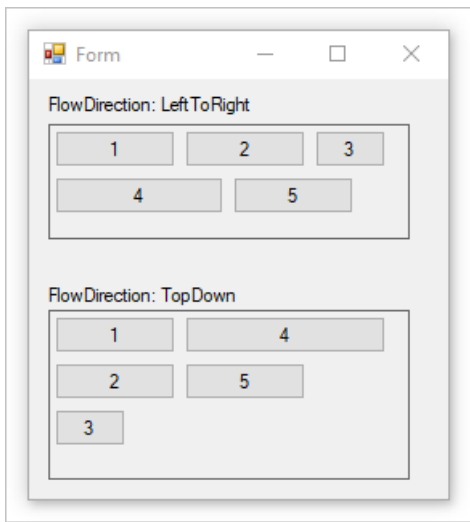


## Container: Flow Layout

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

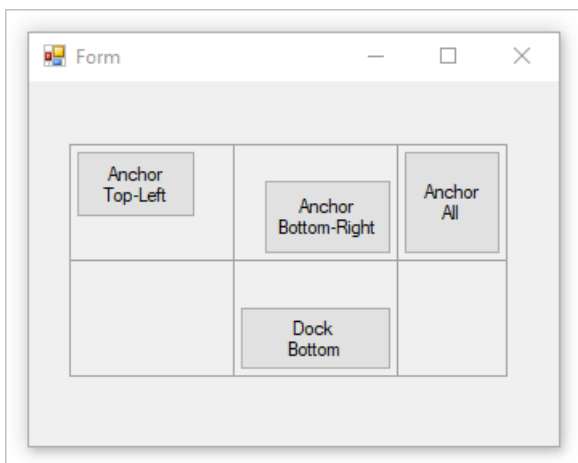
The `TableLayoutPanel` control arranges its contents in a grid. Because the layout is done both at design time and run time, it can change dynamically as the application environment changes. This gives the controls in the panel the ability to resize proportionally, so they can respond to changes such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



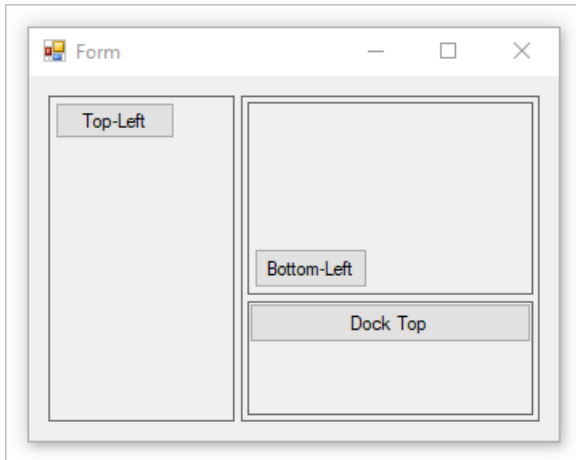
The image above has a table with the `CellBorderStyle` property set to demonstrate the bounds of each cell.

## 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.

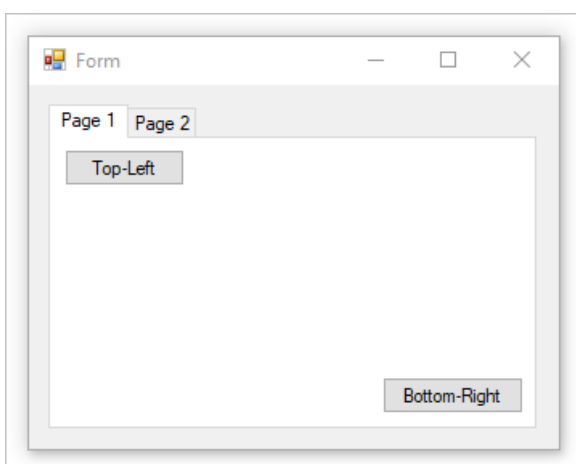


The image above has a split container to create a left and right pane. The right pane contains a second split container with the [Orientation](#) set to [Vertical](#). The [BorderStyle](#) property is set to demonstrate the bounds of each panel.

## 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 [TabPage](#), which contains the individual tabs. Each individual tab is a [TabPage](#) object.



# Label control overview (Windows Forms .NET)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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](#)

# Control events (Windows Forms .NET)

7/20/2021 • 3 minutes to read • [Edit Online](#)

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](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

For more information about how to add or remove a control event handler, see [How to handle an event](#).

## Common events

Controls provide a set of common events through the base class: [Control](#). Not every control responds to every event. For example, the [Label](#) control doesn't respond to keyboard input, so the [Control.PreviewKeyDown](#) event isn't raised. 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:

- [MouseClicked](#)
- [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:

EVENT	DESCRIPTION
<a href="#">BackColorChanged</a>	Occurs when the value of the <a href="#">BackColor</a> property changes.
<a href="#">BackgroundImageChanged</a>	Occurs when the value of the <a href="#">BackgroundImage</a> property changes.
<a href="#">BindingContextChanged</a>	Occurs when the value of the <a href="#">BindingContext</a> property changes.
<a href="#">DockChanged</a>	Occurs when the value of the <a href="#">Dock</a> property changes.
<a href="#">EnabledChanged</a>	Occurs when the <a href="#">Enabled</a> property value has changed.
<a href="#">FontChanged</a>	Occurs when the <a href="#">Font</a> property value changes.
<a href="#">ForeColorChanged</a>	Occurs when the <a href="#">ForeColor</a> property value changes.
<a href="#">LocationChanged</a>	Occurs when the <a href="#">Location</a> property value has changed.
<a href="#">SizeChanged</a>	Occurs when the <a href="#">Size</a> property value changes.
<a href="#">VisibleChanged</a>	Occurs when the <a href="#">Visible</a> 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](#)

# Types of custom controls (Windows Forms .NET)

11/3/2020 • 4 minutes to read • [Edit Online](#)

With Windows Forms, you can develop and implement new controls. You can create a new user control, modify existing controls through inheritance, and write a custom control that does its own painting.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

Deciding which kind of control to create can be confusing. This article highlights the differences among the various kinds of controls from which you can inherit, and provides you with information about how to choose a particular type of control for your project.

IF ...	CREATE A ...
<ul style="list-style-type: none"><li>You want to combine the functionality of several Windows Forms controls into a single reusable unit.</li></ul>	<a href="#">Composite control</a> by inheriting from <a href="#">System.Windows.Forms.UserControl</a> .
<ul style="list-style-type: none"><li>Most of the functionality you need is already identical to an existing Windows Forms control.</li><li>You don't need a custom graphical user interface, or you want to design a new graphical user interface for an existing control.</li></ul>	<a href="#">Extended control</a> by inheriting from a specific Windows Forms control.
<ul style="list-style-type: none"><li>You want to provide a custom graphical representation of your control.</li><li>You need to implement custom functionality that isn't available through standard controls.</li></ul>	<a href="#">Custom control</a> by inheriting from <a href="#">System.Windows.Forms.Control</a> .

## 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](#).
- Provides the security and threading support necessary for a Windows Forms control to act as a Microsoft® ActiveX® control.

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

## Composite Controls

A composite control is a collection of Windows Forms controls encapsulated in a common container. This kind of control is sometimes called a *user control*. The contained controls are called *constituent controls*.

A composite 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 composite control also provides a great deal of default keyboard handling functionality with no extra development effort on your part.

For example, a composite 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.

To author a composite control, derive from the [UserControl](#) class. The [UserControl](#) base class provides keyboard routing for child controls and enables child controls to work as a group.

## 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 much 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, as well as any feature-specific code you need. 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<sup>®</sup> 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.

## ActiveX Controls

Although the Windows Forms infrastructure has been optimized to host Windows Forms controls, you can still use ActiveX controls. There's support for this task in Visual Studio.

## Windowless Controls

The Microsoft Visual Basic® 6.0 and ActiveX technologies support *windowless* controls. Windowless controls aren't supported in Windows Forms.

## 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.](#)

## See also

- [Overview of Using Controls \(Windows Forms .NET\)](#)

# Painting and drawing on controls (Windows Forms .NET)

11/3/2020 • 5 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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.

```
public class PaintEventArgs : EventArgs, IDisposable
{
    public System.Drawing.Rectangle ClipRectangle {get;}
    public System.Drawing.Graphics Graphics {get;}

    // Other properties and methods.
}
```

```

Public Class PaintEventArgs
    Inherits EventArgs
    Implements IDisposable

    Public ReadOnly Property ClipRectangle As System.Drawing.Rectangle
    Public ReadOnly Property Graphics As System.Drawing.Graphics

    ' Other properties and methods.
End Class

```

[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:

```

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);
}

```

```

Protected Overrides Sub OnPaint(e As PaintEventArgs)
    MyBase.OnPaint(e)

    ' Declare and instantiate a drawing pen.
    Using myPen = New System.Drawing.Pen(Color.Aqua)

        ' Create a rectangle that represents the size of the control, minus 1 pixel.
        Dim area = New Rectangle(New Point(0, 0), New Size(Me.Size.Width - 1, Me.Size.Height - 1))

        ' Draw an aqua rectangle in the rectangle represented by the control.
        e.Graphics.DrawRectangle(myPen, area)

    End Using
End Sub

```

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:

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

```
SetStyle(ControlStyles.ResizeRedraw, True)
```

## OnPaintBackground

The base [Control](#) class defines another method that is useful for drawing, the [OnPaintBackground](#) method.

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

```
Protected Overridable Sub OnPaintBackground(e As PaintEventArgs)
```

[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)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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](#)

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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:

```
Button1.AccessibleDefaultActionDescription = "Closes the application."
```

```
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:

```
Button1.AccessibleDescription = "A button with text 'Exit'."
```

```
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:

```
Button1.AccessibleName = "Order"
```

```
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:

```
PictureBox1.AccessibleRole = AccessibleRole.Chart
```

```
pictureBox1.AccessibleRole = AccessibleRole.Chart;
```

## See also

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

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

5/27/2021 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

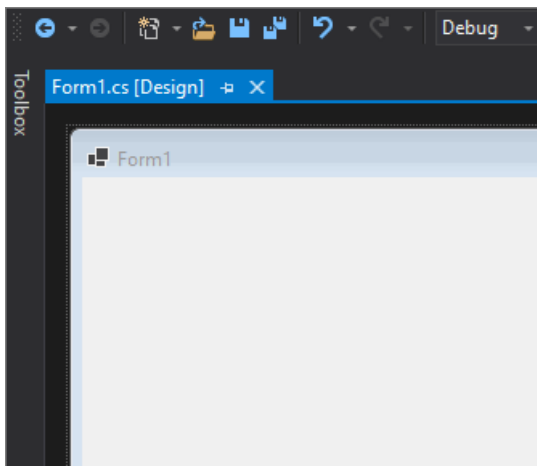
The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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:

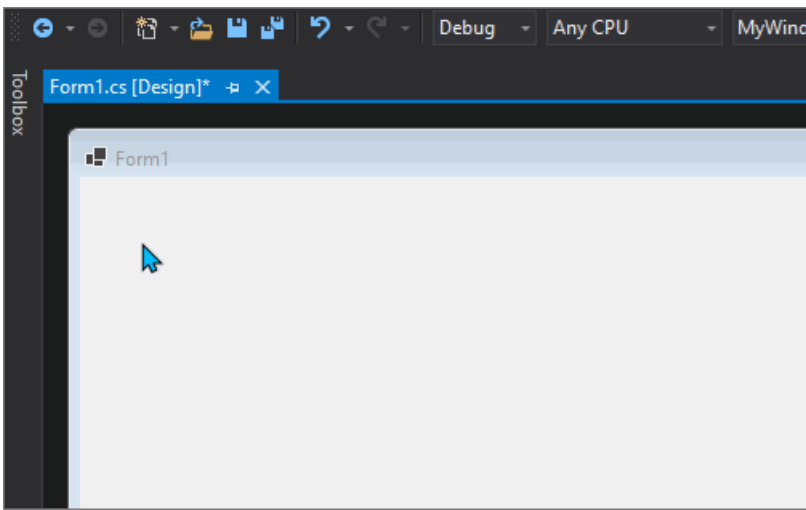
### Add the control by double-clicking

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](#):

```
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);
```

```
Dim label1 As New Label With {.Text = "&First Name",
                              .Location = New Point(10, 10),
                              .TabIndex = 10}

Dim field1 As New TextBox With {.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)

5/27/2021 • 2 minutes to read • [Edit Online](#)

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.

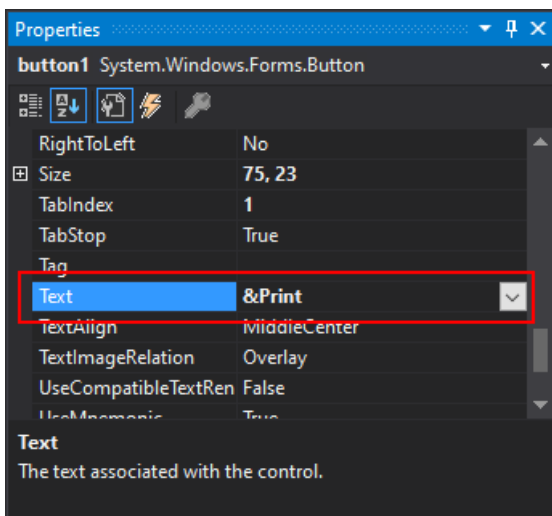
Controls that cannot receive focus can't have access keys, except label controls.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

```
' Set the letter "P" as an access key.  
Button1.Text = "&Print"
```

```
// 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.

```
' Set the letter "P" as an access key.  
Label1.Text = "&Print"  
Label1.TabIndex = 9  
Button1.TabIndex = 10
```

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

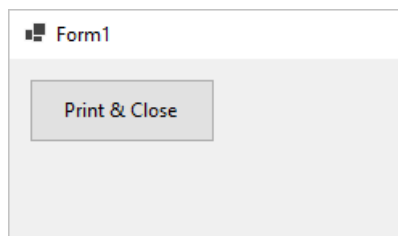
## Display an ampersand

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:

```
' Set the letter "P" as an access key.  
Button1.Text = "Print && Close"
```

```
// 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)

7/20/2021 • 2 minutes to read • [Edit Online](#)

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](#).

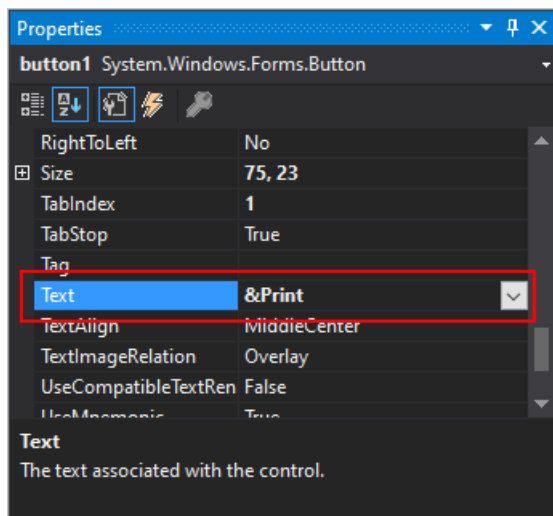
## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

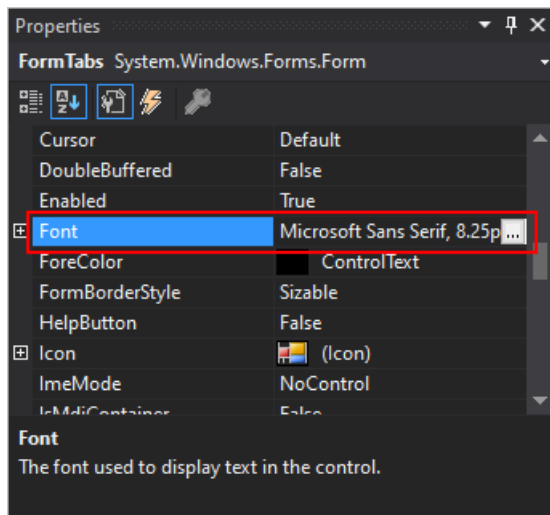
## Designer

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

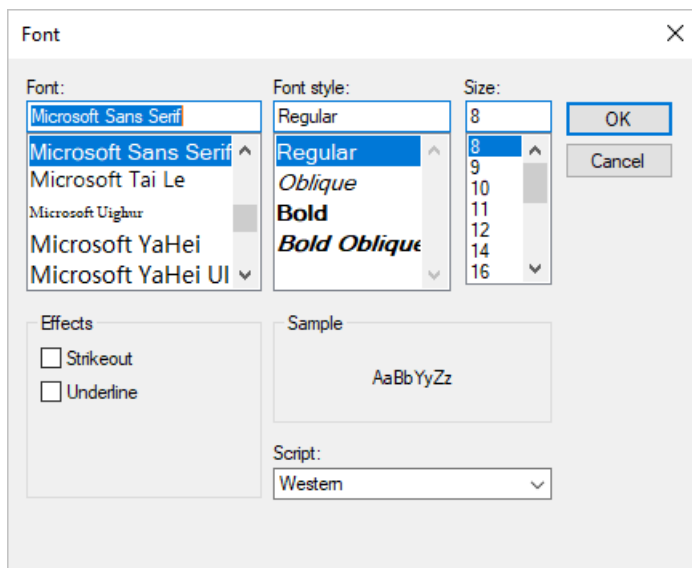
To create an underlined shortcut key, include an ampersand (&) before the letter that will be the shortcut key.



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](#).

```
Button1.Text = "Click here to save changes"
Button1.Font = New Font("Arial", 10, FontStyle.Bold, GraphicsUnit.Point)
```

```
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":

```
MToolStripMenuItem.Text = "&& Now For Something Completely Different"
```

```
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)

6/3/2021 • 2 minutes to read • [Edit Online](#)

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](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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.

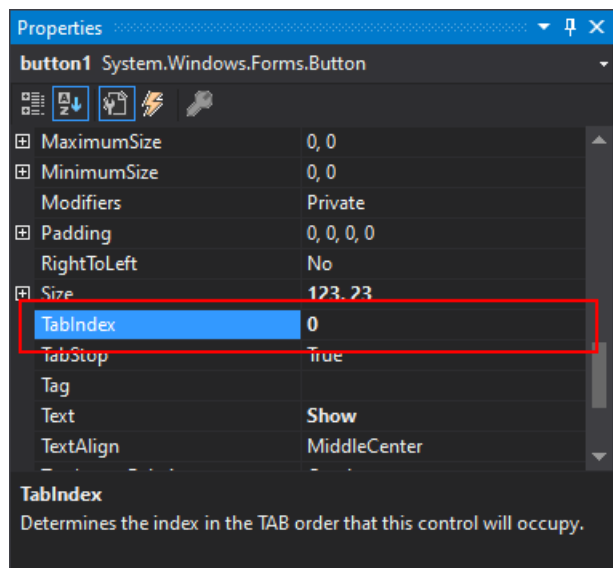
Container controls, such as a [GroupBox](#) control, treat their children as separate from the rest of the form. Each child in the container has its own [TabIndex](#) value. Because a container control can't be focused, when the tab order reaches the container control, the child control of the container 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.

```
Button1.TabIndex = 1
```

```
Button1.TabIndex = 1;
```

## Remove a control from the tab order

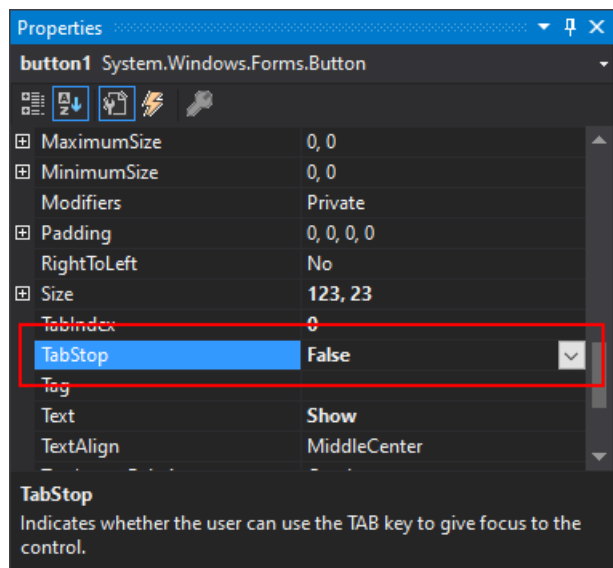
You can prevent a control from receiving focus when the Tab key is pressed, by setting the `TabStop` property to `false`. The control is skipped when you cycle through the controls with the Tab key. The control doesn't lose its tab order when this property is set to `false`.

### 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` .

```
Button1.TabStop = false;
```

```
Button1.TabStop = False
```

### See also

- [Add a control to a form](#)
- [System.Windows.Forms.Control.TabIndex](#)
- [System.Windows.Forms.Control.TabStop](#)

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

6/2/2021 • 3 minutes to read • [Edit Online](#)

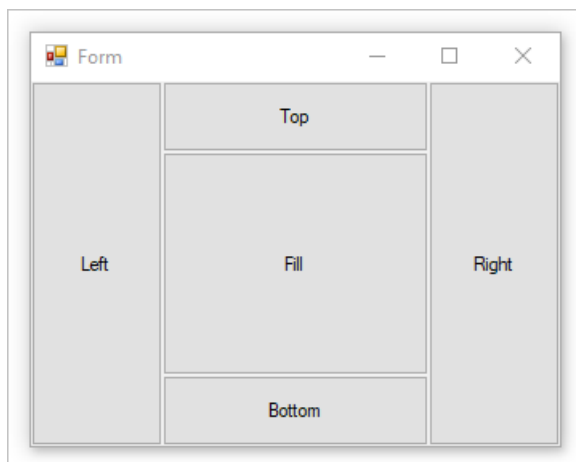
If you're designing a form that the user can resize at run time, the controls on your form should resize and reposition properly. Controls have two properties that help with automatic placement and sizing, when the form changes size.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

- [Control.Dock](#)

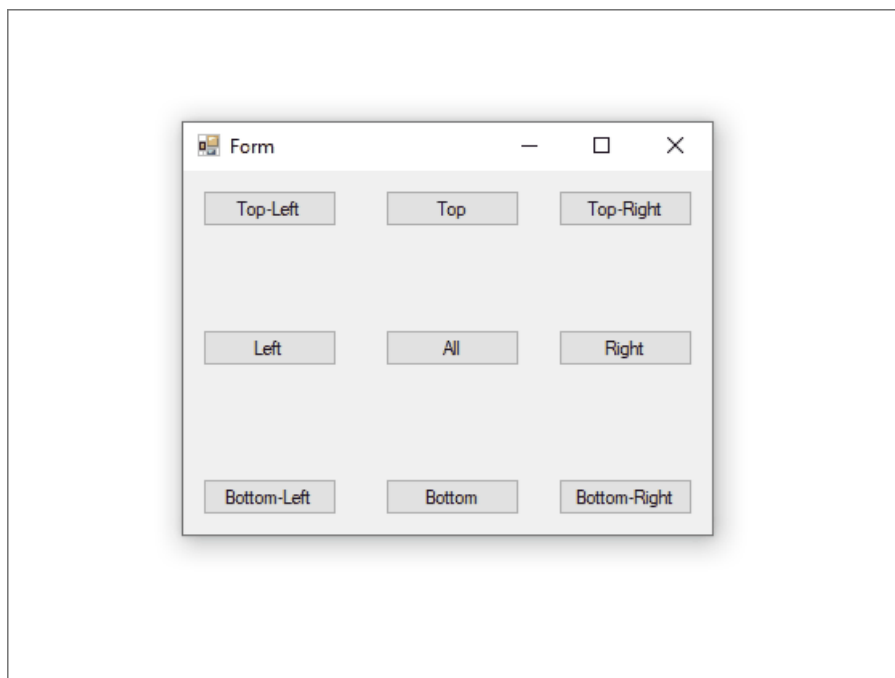
Controls that are docked fill the edges of the control's container, either the form or a container control. For example, Windows Explorer docks its [TreeView](#) control to the left side of the window and its [ListView](#) control to the right side of the window. The docking mode can be any side of the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

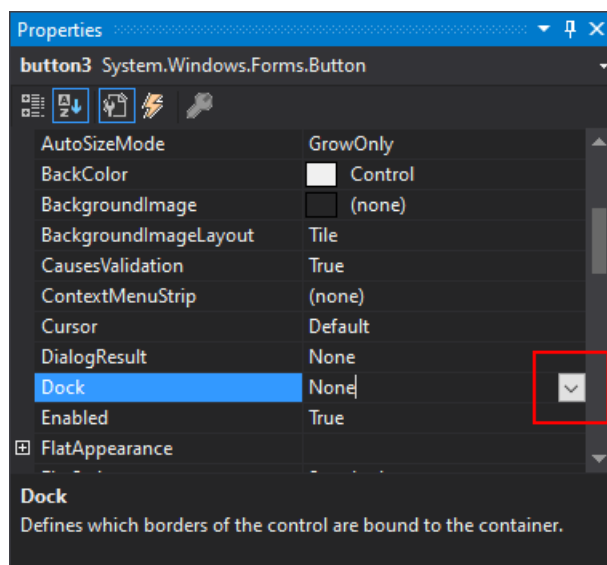
### NOTE

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.

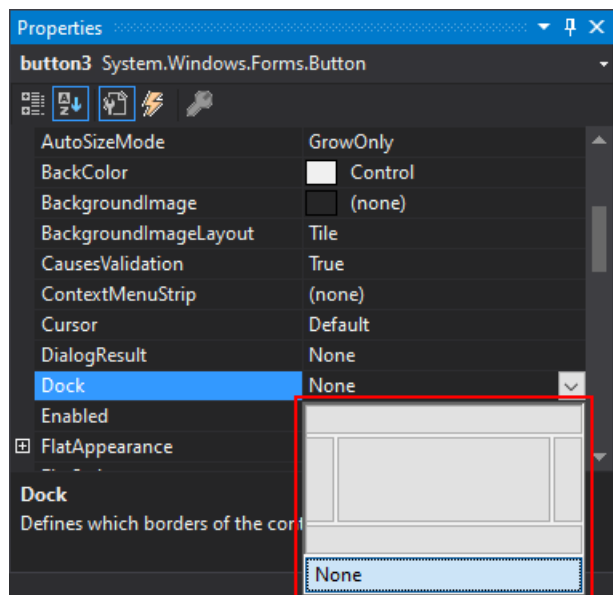
### Use the designer

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:

```
button1.Dock = DockStyle.Right;
```

```
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

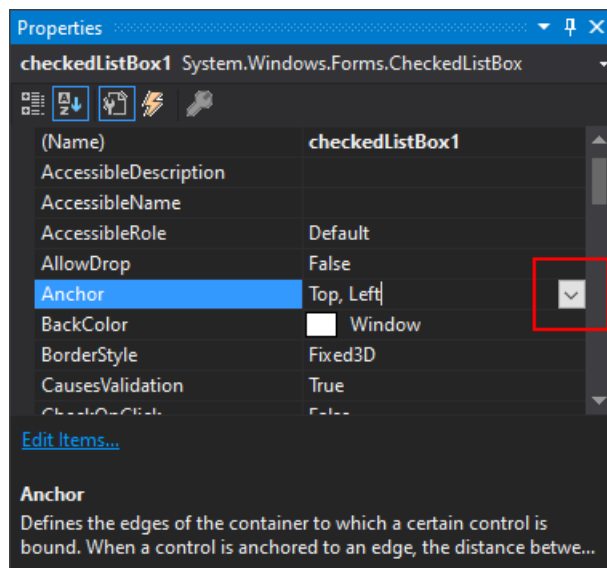
Certain controls, such as the `ComboBox` control, have a limit to their height. Anchoring the control to the bottom of its form or container cannot force the control to exceed its height limit.

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.

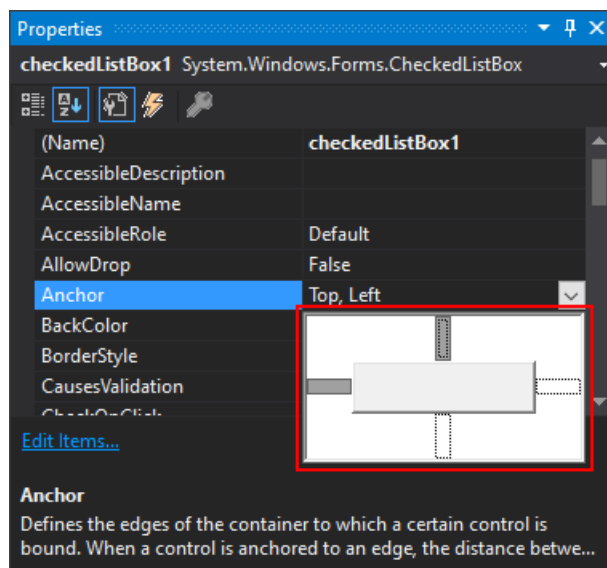
### Use the designer

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:

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

```
button1.Anchor = AnchorStyles.Bottom Or AnchorStyles.Right
```

## See also

- [Position and layout of controls.](#)
- [System.Windows.Forms.Control.Anchor](#)
- [System.Windows.Forms.Control.Dock](#)

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

7/20/2021 • 2 minutes to read • [Edit Online](#)

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.

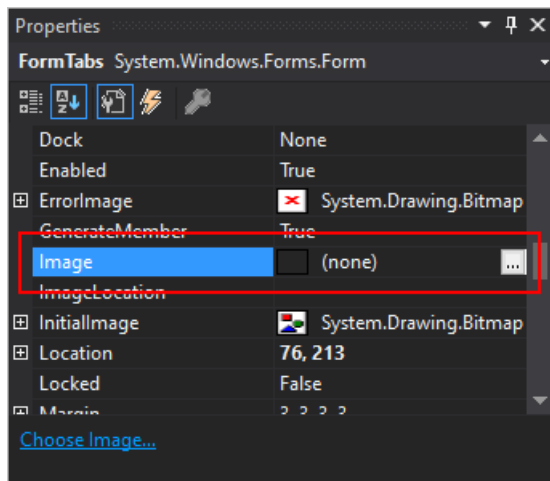
## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

```
// 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");
```

```
' Replace the image named below with your own icon.  
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)

7/30/2021 • 4 minutes to read • [Edit Online](#)

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.

## IMPORTANT


The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

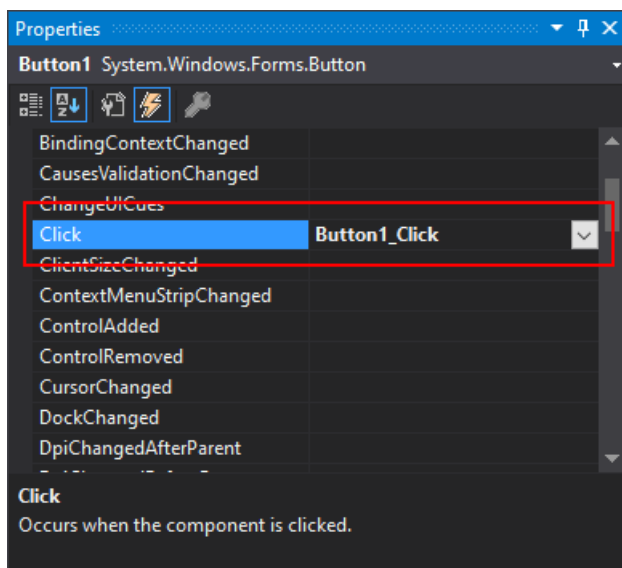
## Handle an event - designer


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:



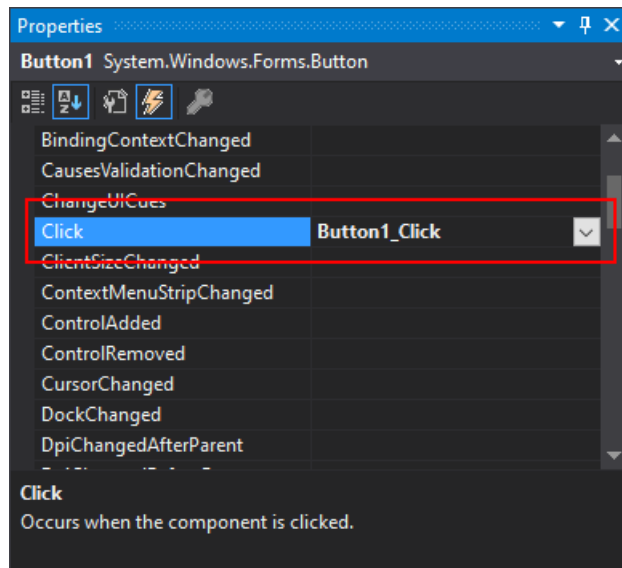
5. Do one of the following:
  - Double-click the event to generate a new handler, it's blank if no handler is assigned. If it's not blank, this action opens the code for the form and navigates to the existing handler.
  - Use the selection box () to choose an existing handler.

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.

```
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)
{
}
}
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    'Create and add the button
    Dim myNewButton As New Button() With {.Location = New Point(10, 10),
                                           .Size = New Size(120, 25),
                                           .Text = "Do work"}


    'Handle the Click event for the new button
    AddHandler myNewButton.Click, AddressOf MyNewButton_Click
    Me.Controls.Add(myNewButton)

    'Remove this button handler so the user cannot do this twice
    RemoveHandler Button1.Click, AddressOf Button1_Click
End Sub

Private Sub MyNewButton_Click(sender As Object, e As EventArgs)

End Sub
```

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:

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

```
RemoveHandler Button1.Click, AddressOf 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:

```
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](#)

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

7/30/2021 • 4 minutes to read • [Edit Online](#)

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](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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.

```
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.";
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim thread2 As New System.Threading.Thread(AddressOf WriteTextUnsafe)
    thread2.Start()
End Sub

Private Sub WriteTextUnsafe()
    TextBox1.Text = "This text was set unsafely."
End Sub
```

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.

```
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;
}
```

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

    Dim threadParameters As New System.Threading.ThreadStart(Sub()
                                                                    WriteTextSafe("This text was set safely.")
                                                                End Sub)

    Dim thread2 As New System.Threading.Thread(threadParameters)
    thread2.Start()

End Sub

Private Sub WriteTextSafe(text As String)

    If (TextBox1.InvokeRequired) Then

        TextBox1.Invoke(Sub()
                        WriteTextSafe($"{text} (THREAD2)")
                        End Sub)

    Else
        TextBox1.Text = text
    End If

End Sub

```

## 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`.

```

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;

```

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

    If (Not BackgroundWorker1.IsBusy) Then
        BackgroundWorker1.RunWorkerAsync()
    End If

End Sub

Private Sub BackgroundWorker1_DoWork(sender As Object, e As ComponentModel.DoWorkEventArgs) Handles
BackgroundWorker1.DoWork

    Dim counter = 0
    Dim max = 10

    While counter <= max

        BackgroundWorker1.ReportProgress(0, counter.ToString())
        System.Threading.Thread.Sleep(1000)

        counter += 1

    End While

End Sub

Private Sub BackgroundWorker1_ProgressChanged(sender As Object, e As
ComponentModel.ProgressChangedEventArgs) Handles BackgroundWorker1.ProgressChanged
    TextBox1.Text = e.UserState
End Sub

```

# Overview of using the keyboard (Windows Forms .NET)

3/9/2021 • 7 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

METHOD	NOTES
<a href="#">PreFilterMessage</a>	This method intercepts queued (also known as posted) Windows messages at the application level.
<a href="#">PreProcessMessage</a>	This method intercepts Windows messages at the form and control level before they have been processed.
<a href="#">WndProc</a>	This method processes Windows messages at the form and control level.
<a href="#">DefWndProc</a>	This method performs the default processing of Windows messages at the form and control level. This provides the minimal functionality of a window.
<a href="#">OnNotifyMessage</a>	This method intercepts messages at the form and control level, after they have been processed. The <a href="#">EnableNotifyMessage</a> 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

ACTION	RELATED METHOD	NOTES
Check for a command key such as an accelerator or menu shortcut.	<a href="#">ProcessCmdKey</a>	This method processes a command key, which takes precedence over regular keys. If this method returns <code>true</code> , the key message is not dispatched and a key event does not occur. If it returns <code>false</code> , <a href="#">IsInputKey</a> is called.
Check for a special key that requires preprocessing or a normal character key that should raise a <a href="#">KeyDown</a> event and be dispatched to a control.	<a href="#">IsInputKey</a>	If the method returns <code>true</code> , it means the control is a regular character and a <a href="#">KeyDown</a> event is raised. If <code>false</code> , <a href="#">ProcessDialogKey</a> is called. <b>Note:</b> To ensure a control gets a key or combination of keys, you can handle the <a href="#">PreviewKeyDown</a> event and set <a href="#">IsInputKey</a> of the <a href="#">PreviewKeyDownEventArgs</a> to <code>true</code> for the key or keys you want.

ACTION	RELATED METHOD	NOTES
Check for a navigation key (ESC, TAB, Return, or arrow keys).	<a href="#">ProcessDialogKey</a>	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 <a href="#">ProcessDialogKey</a> is called on the parent control and so on to the topmost control in the hierarchy. If this method returns <code>true</code> , preprocessing is complete and a key event is not generated. If it returns <code>false</code> , a <a href="#">KeyDown</a> event occurs.

### Preprocessing for a KeyPress event

ACTION	RELATED METHOD	NOTES
Check to see the key is a normal character that should be processed by the control	<a href="#">IsInputChar</a>	If the character is a normal character, this method returns <code>true</code> , the <a href="#">KeyPress</a> event is raised and no further preprocessing occurs. Otherwise <a href="#">ProcessDialogChar</a> will be called.
Check to see if the character is a mnemonic (such as &OK on a button)	<a href="#">ProcessDialogChar</a>	This method, similar to <a href="#">ProcessDialogKey</a> , will be called up the control hierarchy. If the control is a container control, it checks for mnemonics by calling <a href="#">ProcessMnemonic</a> on itself and its child controls. If <a href="#">ProcessDialogChar</a> returns <code>true</code> , a <a href="#">KeyPress</a> 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.

METHOD	NOTES
<a href="#">ProcessKeyMessage</a>	This method processes all keyboard messages that are received by the <a href="#">WndProc</a> method of the control.
<a href="#">ProcessKeyPreview</a>	This method sends the keyboard message to the control's parent. If <a href="#">ProcessKeyPreview</a> returns <code>true</code> , no key event is generated, otherwise <a href="#">ProcessKeyEventArgs</a> is called.
<a href="#">ProcessKeyEventArgs</a>	This method raises the <a href="#">KeyDown</a> , <a href="#">KeyPress</a> , and <a href="#">KeyUp</a> 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 \(Visual Basic\)](#)

TASK	METHOD
Intercept a navigation key and raise a <a href="#">KeyDown</a> event. For example you want TAB and Return to be handled in a text box.	Override <a href="#">IsInputKey</a> . <b>Note:</b> Alternatively, you can handle the <a href="#">PreviewKeyDown</a> event and set <a href="#">IsInputKey</a> of the <a href="#">PreviewKeyDownEventArgs</a> to <code>true</code> 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 <a href="#">ProcessDialogKey</a>
Intercept a navigation key and raise a <a href="#">KeyPress</a> event. For example in a spin-box control you want multiple arrow key presses to accelerate progression through the items.	Override <a href="#">IsInputChar</a> .
Perform special input or navigation handling during a <a href="#">KeyPress</a> event. For example, in a list control holding down the "r" key skips between items that begin with the letter r.	Override <a href="#">ProcessDialogChar</a>
Perform custom mnemonic handling; for example, you want to handle mnemonics on owner-drawn buttons contained in a toolbar.	Override <a href="#">ProcessMnemonic</a> .

## 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 \(Windows Forms .NET\)](#)

# Using keyboard events (Windows Forms .NET)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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 \(Windows Forms .NET\)](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

KEYBOARD EVENT	DESCRIPTION	RESULTS
<a href="#">KeyDown</a>	This event is raised when a user presses a physical key.	<p>The handler for <a href="#">KeyDown</a> receives:</p> <ul style="list-style-type: none"><li>• A <a href="#">KeyEventArgs</a> parameter, which provides the <a href="#">KeyCode</a> property (which specifies a physical keyboard button).</li><li>• The <a href="#">Modifiers</a> property (SHIFT, CTRL, or ALT).</li><li>• The <a href="#">KeyData</a> property (which combines the key code and modifier). The <a href="#">KeyEventArgs</a> parameter also provides:<ul style="list-style-type: none"><li>◦ The <a href="#">Handled</a> property, which can be set to prevent the underlying control from receiving the key.</li><li>◦ The <a href="#">SuppressKeyPress</a> property, which can be used to suppress the <a href="#">KeyPress</a> and <a href="#">KeyUp</a> events for that keystroke.</li></ul></li></ul>

KEYBOARD EVENT	DESCRIPTION	RESULTS
<a href="#">KeyPress</a>	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.	<p><a href="#">KeyPress</a> is raised after <a href="#">KeyDown</a>.</p> <ul style="list-style-type: none"> <li>The handler for <a href="#">KeyPress</a> receives:</li> <li>A <a href="#">KeyPressEventArgs</a> 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.</li> </ul> <p>For example, the "A" key will generate:</p> <ul style="list-style-type: none"> <li>The character code 65, if it is pressed with the SHIFT key</li> <li>Or the CAPS LOCK key, 97 if it is pressed by itself,</li> <li>And 1, if it is pressed with the CTRL key.</li> </ul>
<a href="#">KeyUp</a>	This event is raised when a user releases a physical key.	<p>The handler for <a href="#">KeyUp</a> receives:</p> <ul style="list-style-type: none"> <li>A <a href="#">KeyEventArgs</a> parameter: <ul style="list-style-type: none"> <li>Which provides the <a href="#">KeyCode</a> property (which specifies a physical keyboard button).</li> <li>The <a href="#">Modifiers</a> property (SHIFT, CTRL, or ALT).</li> <li>The <a href="#">KeyData</a> property (which combines the key code and modifier).</li> </ul> </li> </ul>

## 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 \(Windows Forms .NET\)](#)

# Overview of how to validate user input (Windows Forms .NET)

11/3/2020 • 6 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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##`, 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.

CONTROL	DEFAULT VALIDATION BEHAVIOR
<a href="#">ContainerControl</a>	<a href="#">Inherit</a>
<a href="#">Form</a>	<a href="#">EnableAllowFocusChange</a>
<a href="#">PropertyGrid</a>	Property not exposed in Visual Studio
<a href="#">ToolStripContainer</a>	Property not exposed in Visual Studio
<a href="#">SplitContainer</a>	<a href="#">Inherit</a>
<a href="#">UserControl</a>	<a href="#">EnableAllowFocusChange</a>

## 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)

11/3/2020 • 3 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
        e.Handled = true;
}
```

```
Private Sub TextBox1_KeyPress(sender As Object, e As KeyPressEventArgs)
    If e.KeyChar = "a"c Or e.KeyChar = "A"c Then
        e.Handled = True
    End If
End Sub
```

## 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 `!`:

```
private void textBox2_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
    {
        e.KeyChar = '!';
        e.Handled = false;
    }
}
```

```
Private Sub TextBox2_KeyPress(sender As Object, e As KeyPressEventArgs)
    If e.KeyChar = "a" Or e.KeyChar = "A" Then
        e.KeyChar = "!"
        e.Handled = False
    End If
End Sub
```

## 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).

```
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);
}
```

```

Public Overrides Function PreProcessMessage(ByRef m As Message) As Boolean

    Const WM_KEYDOWN = &H100

    If m.Msg = WM_KEYDOWN Then
        Dim keyCode As Keys = CType(m.WParam, Keys) And Keys.KeyCode

        Select Case keyCode
            Case Keys.F1 : m.WParam = CType(Keys.D1, IntPtr)
            Case Keys.F2 : m.WParam = CType(Keys.D2, IntPtr)
            Case Keys.F3 : m.WParam = CType(Keys.D3, IntPtr)
            Case Keys.F4 : m.WParam = CType(Keys.D4, IntPtr)
            Case Keys.F5 : m.WParam = CType(Keys.D5, IntPtr)
            Case Keys.F6 : m.WParam = CType(Keys.D6, IntPtr)
            Case Keys.F7 : m.WParam = CType(Keys.D7, IntPtr)
            Case Keys.F8 : m.WParam = CType(Keys.D8, IntPtr)
            Case Keys.F9 : m.WParam = CType(Keys.D9, IntPtr)
            Case Keys.F10 : m.WParam = CType(Keys.D0, IntPtr)
        End Select
    End If

    Return MyBase.PreProcessMessage(m)
End Function

```

## 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)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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.

```
// 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);
}
```

```
' Event only raised when non-modifier key is pressed
Private Sub TextBox1_KeyPress(sender As Object, e As KeyPressEventArgs)
    If ((Control.ModifierKeys And Keys.Shift) = Keys.Shift) Then
        MessageBox.Show("KeyPress " & [Enum].GetName(GetType(Keys), Keys.Shift))
    End If
End Sub

' Event raised as soon as shift is pressed
Private Sub TextBox1_KeyDown(sender As Object, e As KeyEventArgs)
    If ((Control.ModifierKeys And Keys.Shift) = Keys.Shift) Then
        MessageBox.Show("KeyPress " & [Enum].GetName(GetType(Keys), Keys.Shift))
    End If
End Sub
```

## 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)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

```
// 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;
        }
    }
}
```

```
' 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.
Private Sub Form1_KeyPress(sender As Object, e As KeyPressEventArgs)
    If e.KeyChar >= Chr(48) And e.KeyChar <= Chr(57) Then
        MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' pressed.")

        Select Case e.KeyChar
            Case Chr(49), Chr(52), Chr(55)
                MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' consumed.")
                e.Handled = True
        End Select
    End If
End Sub
```

## See also

- [Overview of using the keyboard \(Windows Forms .NET\)](#)
- [Using keyboard events \(Windows Forms .NET\)](#)
- [Keys](#)
- [ModifierKeys](#)
- [KeyDown](#)
- [KeyPress](#)

# How to simulate keyboard events (Windows Forms .NET)

11/3/2020 • 3 minutes to read • [Edit Online](#)

Windows Forms provides a few options for programmatically simulating keyboard input. This article provides an overview of these options.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

```
<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](#).

```
private void button1_Click(object sender, EventArgs e)
{
    comboBox1.Focus();
    SendKeys.Send("%+{DOWN}");
}
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs)
    ComboBox1.Focus()
    SendKeys.Send("%+{DOWN}")
End Sub
```

## 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.

```
[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=");
}
```

```
<Runtime.InteropServices.DllImport("USER32.DLL", CharSet:=Runtime.InteropServices.CharSet.Unicode)>
Public Shared Function FindWindow(lpClassName As String, lpWindowName As String) As IntPtr : End Function

<Runtime.InteropServices.DllImport("USER32.DLL")>
Public Shared Function SetForegroundWindow(hWnd As IntPtr) As Boolean : End Function

Private Sub Button1_Click(sender As Object, e As EventArgs)
    Dim hCalcWindow As IntPtr = FindWindow(Nothing, "Calculator")

    If SetForegroundWindow(hCalcWindow) Then
        SendKeys.Send("10{+}10=")
    End If
End Sub
```

## 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](#)

# Overview of using the mouse (Windows Forms .NET)

11/3/2020 • 4 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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.

PROPERTY	DESCRIPTION
<a href="#">DoubleClickSize</a>	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.
<a href="#">DoubleClickTime</a>	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.
<a href="#">MouseButtons</a>	Gets the number of buttons on the mouse.
<a href="#">MouseButtonsSwapped</a>	Gets a value indicating whether the functions of the left and right mouse buttons have been swapped.
<a href="#">MouseHoverSize</a>	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.
<a href="#">MouseHoverTime</a>	Gets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle before a mouse hover message is generated.
<a href="#">MousePresent</a>	Gets a value indicating whether a mouse is installed.
<a href="#">MouseSpeed</a>	Gets a value indicating the current mouse speed, from 1 to 20.
<a href="#">MouseWheelPresent</a>	Gets a value indicating whether a mouse with a mouse wheel is installed.
<a href="#">MouseWheelScrollDelta</a>	Gets the amount of the delta value of the increment of a single mouse wheel rotation.
<a href="#">MouseWheelScrollLines</a>	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.

METHOD	NOTES
<a href="#">PreFilterMessage</a>	This method intercepts queued (also known as posted) Windows messages at the application level.
<a href="#">PreProcessMessage</a>	This method intercepts Windows messages at the form and control level before they have been processed.

METHOD	NOTES
<a href="#">WndProc</a>	This method processes Windows messages at the form and control level.
<a href="#">DefWndProc</a>	This method performs the default processing of Windows messages at the form and control level. This provides the minimal functionality of a window.
<a href="#">OnNotifyMessage</a>	This method intercepts messages at the form and control level, after they've been processed. The <a href="#">EnableNotifyMessage</a> 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 \(Windows Forms .NET\)](#)

# Using mouse events (Windows Forms .NET)

11/3/2020 • 6 minutes to read • [Edit Online](#)

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 \(Windows Forms .NET\)](#).

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

MOUSE EVENT	DESCRIPTION
<a href="#">Click</a>	This event occurs when the mouse button is released, typically before the <a href="#">MouseUp</a> event. The handler for this event receives an argument of type <a href="#">EventArgs</a> . Handle this event when you only need to determine when a click occurs.
<a href="#">MouseClicked</a>	This event occurs when the user clicks the control with the mouse. The handler for this event receives an argument of type <a href="#">MouseEventArgs</a> . Handle this event when you need to get information about the mouse when a click occurs.
<a href="#">DoubleClick</a>	This event occurs when the control is double-clicked. The handler for this event receives an argument of type <a href="#">EventArgs</a> . Handle this event when you only need to determine when a double-click occurs.
<a href="#">MouseDoubleClick</a>	This event occurs when the user double-clicks the control with the mouse. The handler for this event receives an argument of type <a href="#">MouseEventArgs</a> . Handle this event when you need to get information about the mouse when a double-click occurs.
<a href="#">MouseDown</a>	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 <a href="#">MouseEventArgs</a> .
<a href="#">MouseEnter</a>	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 <a href="#">EventArgs</a> .
<a href="#">MouseHover</a>	This event occurs when the mouse pointer stops and rests over the control. The handler for this event receives an argument of type <a href="#">EventArgs</a> .

MOUSE EVENT	DESCRIPTION
<a href="#">MouseLeave</a>	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 <a href="#">EventArgs</a> .
<a href="#">MouseMove</a>	This event occurs when the mouse pointer moves while it is over a control. The handler for this event receives an argument of type <a href="#">MouseEventArgs</a> .
<a href="#">MouseUp</a>	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 <a href="#">MouseEventArgs</a> .
<a href="#">MouseWheel</a>	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 <a href="#">MouseEventArgs</a> . You can use the <a href="#">Delta</a> property of <a href="#">MouseEventArgs</a> 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. [MouseClicked](#) 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](#), [MouseClicked](#)
- **Right click:** No click events raised
- **Left double-click:** [Click](#), [MouseClicked](#); [Click](#), [MouseClicked](#)
- **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](#), [MouseClicked](#)
- **Right click:** No click events raised
- **Left double-click:** [Click](#), [MouseClicked](#), [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](#), [MouseDown](#)
- **Right click:** [Click](#), [MouseDown](#)
- **Left double-click:** [Click](#), [MouseDown](#); [DoubleClick](#), [MouseDoubleClick](#)
- **Right double-click:** [Click](#), [MouseDown](#); [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](#), [MouseDown](#)
- **Right click:** [Click](#), [MouseDown](#)
- **Left double-click:** [Click](#), [MouseDown](#); [DoubleClick](#), [MouseDoubleClick](#)
- **Right double-click:** [Click](#), [MouseDown](#); [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)

11/3/2020 • 3 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

MOUSE EVENT	DESCRIPTION
<a href="#">DragEnter</a>	This event occurs when an object is dragged into the control's bounds. The handler for this event receives an argument of type <a href="#">DragEventArgs</a> .
<a href="#">DragOver</a>	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 <a href="#">DragEventArgs</a> .
<a href="#">DragDrop</a>	This event occurs when a drag-and-drop operation is completed. The handler for this event receives an argument of type <a href="#">DragEventArgs</a> .
<a href="#">DragLeave</a>	This event occurs when an object is dragged out of the control's bounds. The handler for this event receives an argument of type <a href="#">EventArgs</a> .

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.

MOUSE EVENT	DESCRIPTION
<a href="#">GiveFeedback</a>	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 <a href="#">GiveFeedbackEventArgs</a> .
<a href="#">QueryContinueDrag</a>	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 <a href="#">QueryContinueDragEventArgs</a> .

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.

```
// 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));
```

```
' Initiate the drag
Private Sub Label1_MouseDown(sender As Object, e As MouseEventArgs)
    DoDragDrop(DirectCast(sender, Label).Text, DragDropEffects.All)
End Sub

' Set the effect filter and allow the drop on this control
Private Sub TextBox1_DragOver(sender As Object, e As DragEventArgs)
    e.Effect = DragDropEffects.All
End Sub

' React to the drop on this control
Private Sub TextBox1_DragDrop(sender As Object, e As DragEventArgs)
    TextBox1.Text = e.Data.GetData(GetType(String))
End Sub
```

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)

11/3/2020 • 4 minutes to read • [Edit Online](#)

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 [MouseDown](#) event is performed before the action tied to the [DoubleClick](#) or [MouseDoubleClick](#) event. This topic demonstrates two solutions to this problem.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

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:

```
public partial class DoubleClickButton : Button
{
    public DoubleClickButton()
    {
        // Set the style so a double click event occurs.
        SetStyle(ControlStyles.StandardClick | ControlStyles.StandardDoubleClick, true);
    }
}
```

```
Public Class DoubleClickButton : Inherits Button

    Public Sub New()
        SetStyle(ControlStyles.StandardClick Or ControlStyles.StandardDoubleClick, True)
    End Sub

End Class
```

The following code demonstrates how a form changes the style of border based on a click or double-click of the new button control:

```

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;
    }
}

```

```

Partial Public Class Form1

    Private _initialStyle As FormBorderStyle
    Private _isDoubleClicking As Boolean

    Public Sub New()
        InitializeComponent()
    End Sub

    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Dim button1 As New DoubleClickButton

        _initialStyle = FormBorderStyle

        button1.Location = New Point(50, 50)
        button1.Size = New Size(200, 23)
        button1.Text = "Click or Double Click"

        AddHandler button1.Click, AddressOf Button1_Click
        AddHandler button1.DoubleClick, AddressOf Button1_DoubleClick

        Controls.Add(button1)

    End Sub

    Private Sub Button1_DoubleClick(sender As Object, e As EventArgs)
        ' This flag prevents the click handler logic from running
        ' A double click raises the click event twice.
        _isDoubleClicking = True
        FormBorderStyle = _initialStyle
    End Sub

    Private Sub Button1_Click(sender As Object, e As EventArgs)
        If _isDoubleClicking Then
            _isDoubleClicking = False
        Else
            FormBorderStyle = FormBorderStyle.FixedToolWindow
        End If
    End Sub

End Class

```

## 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.

```

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();
}
}

```

```

Imports System.Drawing
Imports System.Windows.Forms

Public Class Form2
    Private _lastClick As Date
    Private _inDoubleClick As Boolean
    Private _doubleClickArea As Rectangle
    Private _doubleClickMaxTime As TimeSpan
    Private _singleClickAction As Action
    Private _doubleClickAction As Action
    Private WithEvents _clickTimer As Timer

    Private Sub Form2_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        _doubleClickMaxTime = TimeSpan.FromMilliseconds(SystemInformation.DoubleClickTime)

        _clickTimer = New Timer()
        _clickTimer.Interval = SystemInformation.DoubleClickTime

        _singleClickAction = Sub()
            MessageBox.Show("Single click")
        End Sub

        _doubleClickAction = Sub()
            MessageBox.Show("Double click")
        End Sub
    End Sub

    Private Sub Form2_MouseDown(sender As Object, e As MouseEventArgs) Handles MyBase.MouseDown
        If _inDoubleClick Then

            _inDoubleClick = False

            Dim length As TimeSpan = Date.Now - _lastClick

            ' If double click is valid, respond
            If _doubleClickArea.Contains(e.Location) And length < _doubleClickMaxTime Then
                _clickTimer.Stop()
                Call _doubleClickAction()
            End If

            Return
        End If

        ' Double click was invalid, restart
        _clickTimer.Stop()
        _clickTimer.Start()
        _lastClick = Date.Now
        _inDoubleClick = True
        _doubleClickArea = New Rectangle(e.Location - (SystemInformation.DoubleClickSize / 2),
                                         SystemInformation.DoubleClickSize)
    End Sub

    Private Sub SingleClickTimer_Tick(sender As Object, e As EventArgs) Handles _clickTimer.Tick
        ' Clear double click watcher and timer
        _inDoubleClick = False
        _clickTimer.Stop()

        Call _singleClickAction()
    End Sub
End Class

```

## 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)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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:

```
private void button1_MouseEnter(object sender, EventArgs e) =>
    Cursor.Hide();

private void button1_MouseLeave(object sender, EventArgs e) =>
    Cursor.Show();
```

```
Private Sub Button1_MouseEnter(sender As Object, e As EventArgs) Handles Button1.MouseEnter
    Cursor.Hide()
End Sub

Private Sub Button1_MouseLeave(sender As Object, e As EventArgs) Handles Button1.MouseLeave
    Cursor.Show()
End Sub
```

## 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 by 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:

```
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);
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    PointToScreen(Button2.Location)
End Sub

Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    PointToScreen(Button1.Location)
End Sub
```

## 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:

```
button2.Cursor = System.Windows.Forms.Cursors.Hand;
```

```
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](#)

# How to simulate mouse events (Windows Forms .NET)

11/3/2020 • 2 minutes to read • [Edit Online](#)

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.

## IMPORTANT

The Desktop Guide documentation for .NET 5 (and .NET Core) is under construction.

## 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.

```
button1.PerformClick();
```

```
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`.

```
private void button1_Click(object sender, EventArgs e)
{
    InvokeOnClick(checkBox1, EventArgs.Empty);
}
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    InvokeOnClick(CheckBox1, EventArgs.Empty)
End Sub
```

## 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:

```
[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);
}
```

```
<Runtime.InteropServices.DllImport("USER32.DLL", EntryPoint:="SetCursorPos")>
Public Shared Function SetCursorPos(x As Integer, y As Integer) As Boolean : End Function

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim position As Point = PointToScreen(CheckBox1.Location) + New Size(CheckBox1.Width / 2,
CheckBox1.Height / 2)
    SetCursorPos(position.X, position.Y)
End Sub
```

## 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 \(Windows Forms .NET\)](#)