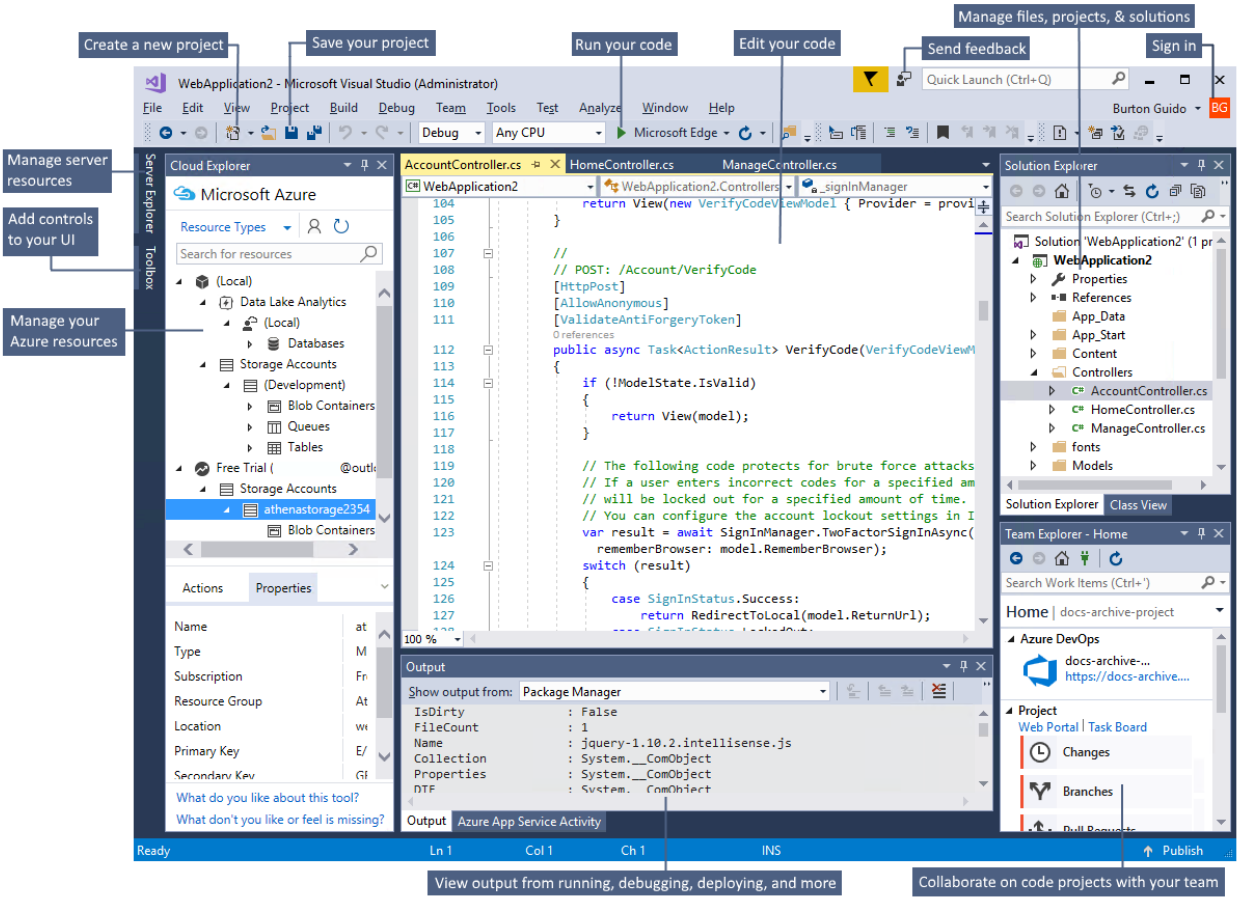
Contents

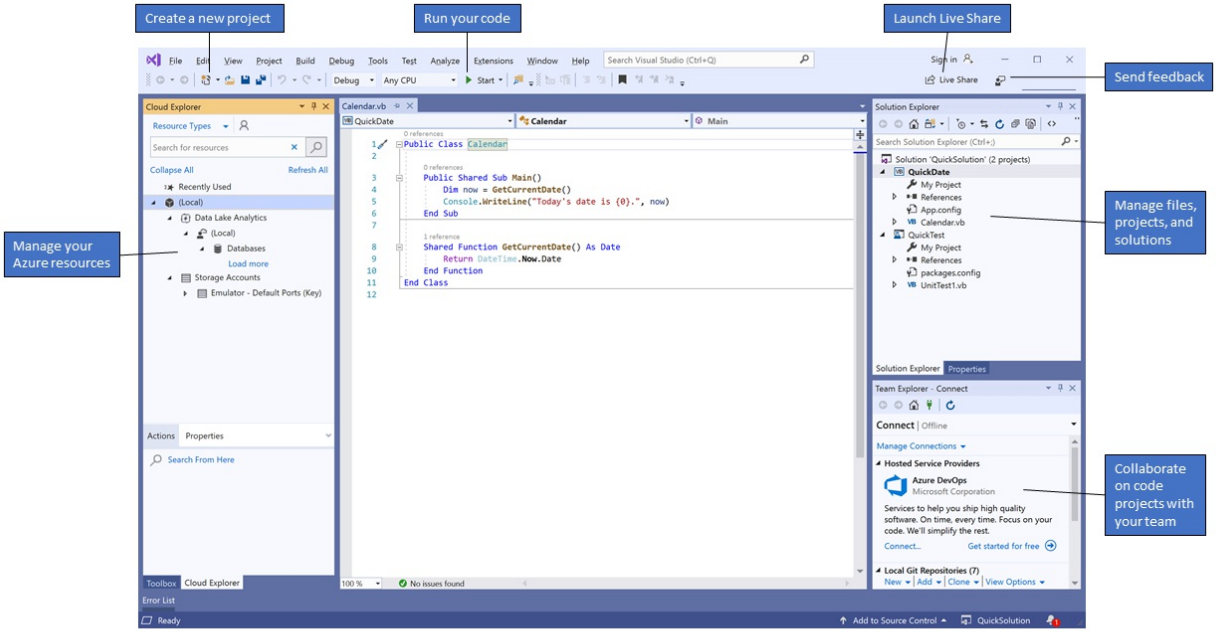
- Visual Basic Tutorials
- Overview
  - About Visual Studio
  - About the code editor
  - About projects and solutions
  - More Visual Studio features
- Create an app
  - Create a console app
  - Create a UWP app
  - Create a WPF app
  - Create a web app
  - Create a Windows Forms app
- Learn Visual Studio
  - Open a project from a repo
  - Write and edit code
  - Compile and build
  - Debug your code
  - Unit testing
  - Deploy your project
  - Access data

Welcome to the Visual Studio IDE | Visual Basic

6/10/2019 • 13 minutes to read • Edit Online

The Visual Studio *integrated development environment* is a creative launching pad that you can use to edit, debug, and build code, and then publish an app. An integrated development environment (IDE) is a feature-rich program that can be used for many aspects of software development. Over and above the standard editor and debugger that most IDEs provide, Visual Studio includes compilers, code completion tools, graphical designers, and many more features to ease the software development process.





This image shows Visual Studio with an open project and several key tool windows you'll likely use:

- Solution Explorer (top right) lets you view, navigate, and manage your code files. **Solution Explorer** can help organize your code by grouping the files into solutions and projects.
- The editor window (center), where you'll likely spend a majority of your time, displays file contents. This is where you can edit code or design a user interface such as a window with buttons and text boxes.
- The Output window (bottom center) is where Visual Studio sends notifications such as debugging and error messages, compiler warnings, publishing status messages, and more. Each message source has its own tab.
- Team Explorer (bottom right) lets you track work items and share code with others using version control technologies such as Git and Team Foundation Version Control (TFVC).

Editions

Visual Studio is available for Windows and Mac. Visual Studio for Mac has many of the same features as Visual Studio 2017, and is optimized for developing cross-platform and mobile apps. This article focuses on the Windows version of Visual Studio 2017.

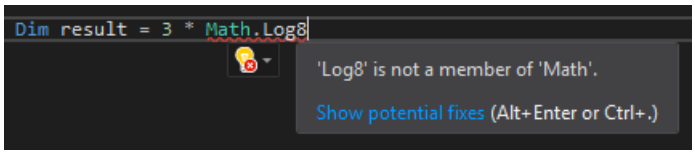
There are three editions of Visual Studio 2017: Community, Professional, and Enterprise. See Compare Visual Studio 2017 IDEs to learn about which features are supported in each edition.

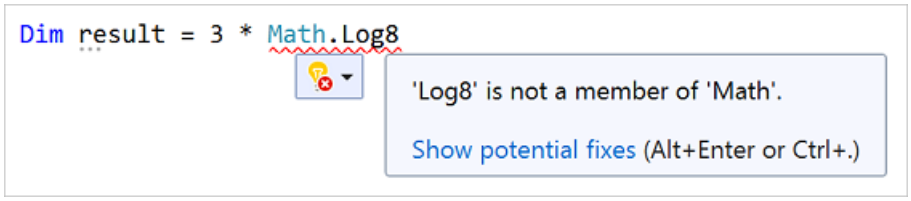
Popular productivity features

Some of the popular features in Visual Studio that help you to be more productive as you develop software include:

- Squiggles and Quick Actions

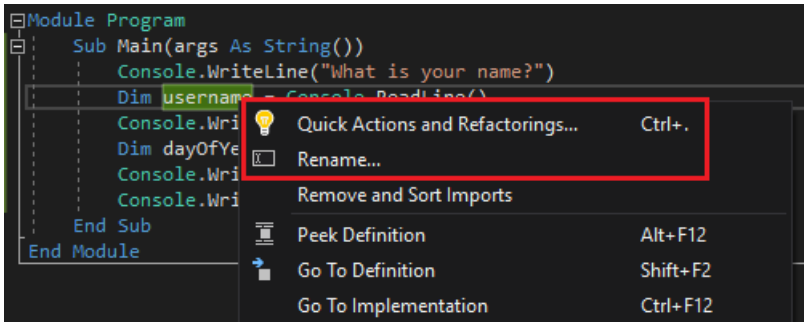
  Squiggles are wavy underlines that alert you to errors or potential problems in your code as you type. These visual clues enable you to fix problems immediately without waiting for the error to be discovered during build or when you run the program. If you hover over a squiggle, you see additional information about the error. A light bulb may also appear in the left margin with actions, known as Quick Actions, to fix the error.

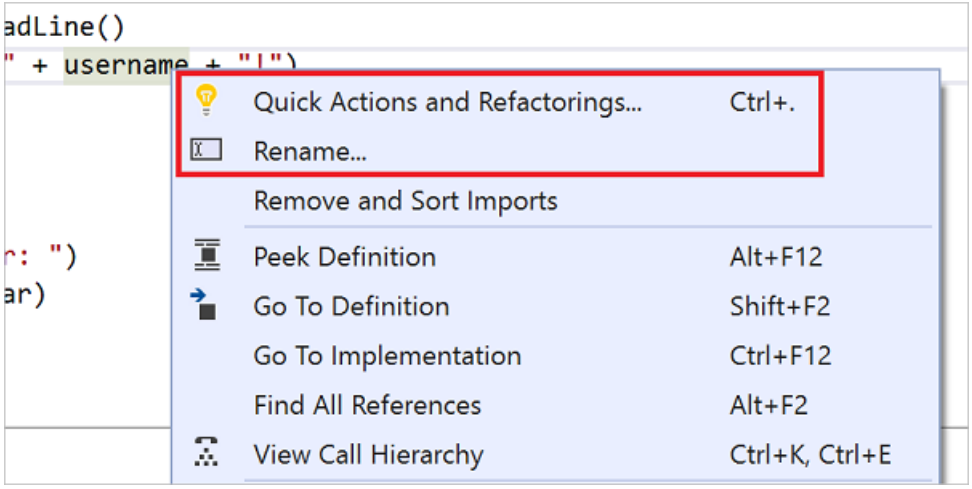




- Refactoring

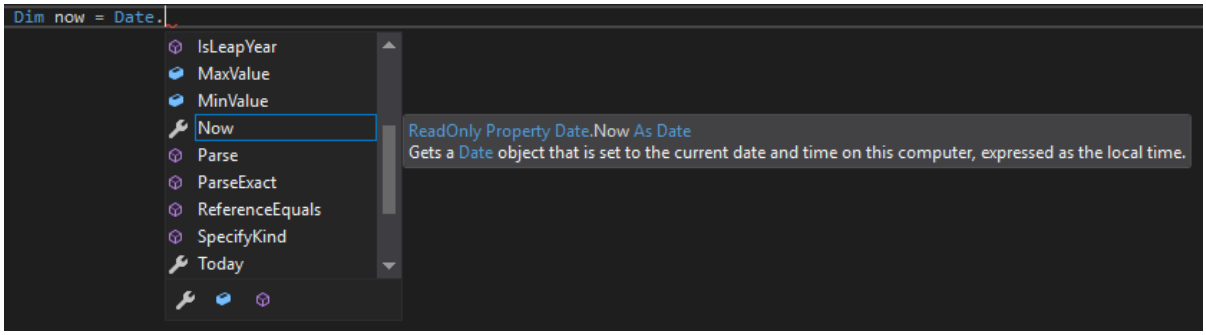
  Refactoring includes operations such as intelligent renaming of variables, extracting one or more lines of code into a new method, changing the order of method parameters, and more.





- IntelliSense

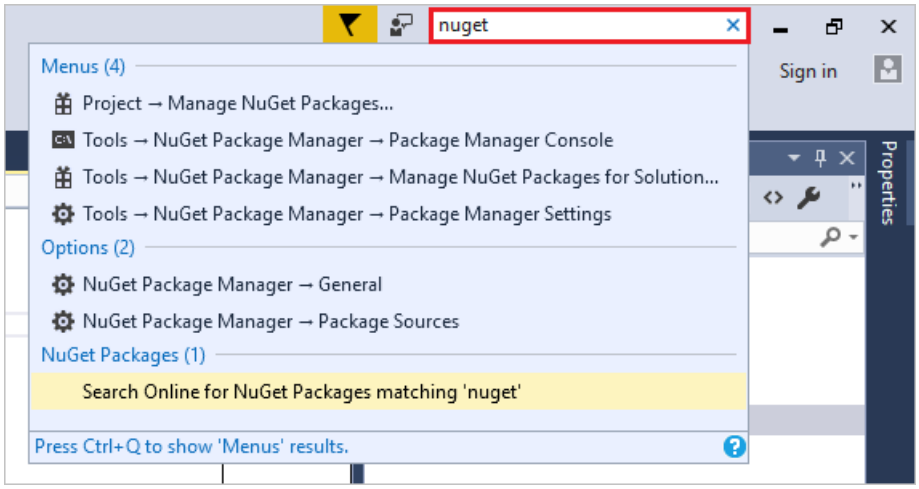
  IntelliSense is a term for a set of features that displays information about your code directly in the editor and, in some cases, write small bits of code for you. It's like having basic documentation inline in the editor, which saves you from having to look up type information elsewhere. IntelliSense features vary by language. For more information, see C# IntelliSense, Visual C++ IntelliSense, JavaScript IntelliSense, and Visual Basic IntelliSense. The following illustration shows how IntelliSense displays a member list for a type:



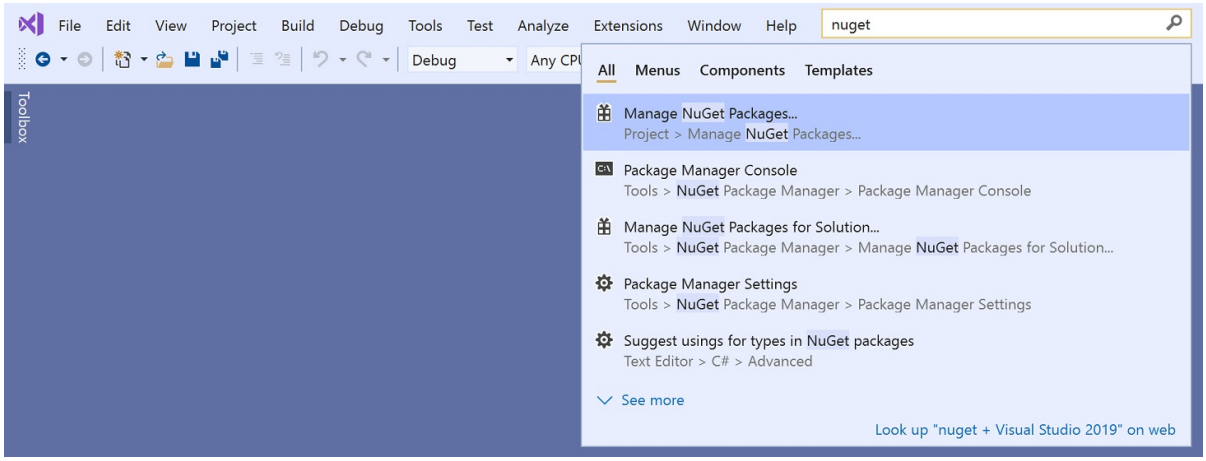
- Search box

  Visual Studio can seem overwhelming at times with so many menus, options, and properties. The search box is a great way to rapidly find what you need in Visual Studio. When you start typing the name of something you're looking for, Visual Studio lists results that take you exactly where you need to go. If you need to add functionality to Visual Studio, for example to add support for an additional programming language, the search box provides results that open Visual Studio Installer to install a workload or individual component.

  > **TIP**
  >
  > Press **Ctrl**+**Q** as a shortcut to the search box.



For more information, see Quick Launch.



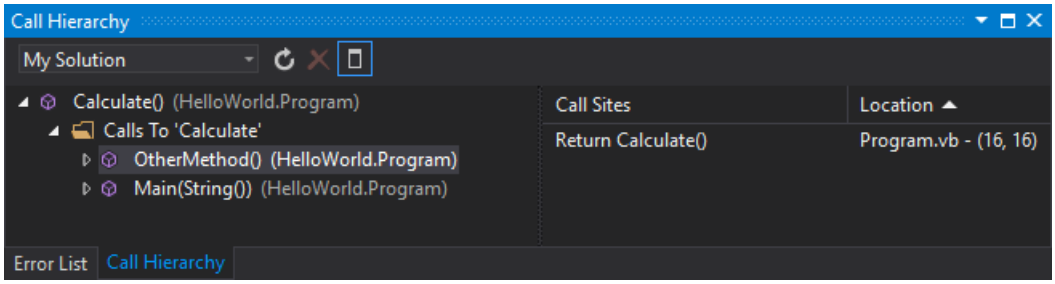
- Live Share

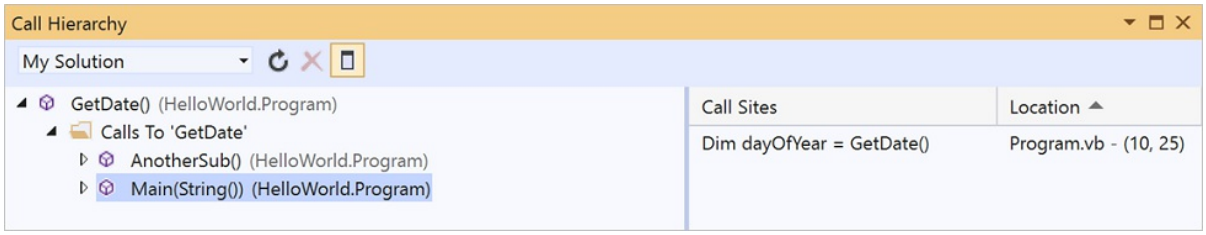
  Collaboratively edit and debug with others in real time, regardless of what your app type or programming

language. You can instantly and securely share your project and, as needed, debugging sessions, terminal instances, localhost web apps, voice calls, and more.

- Call Hierarchy

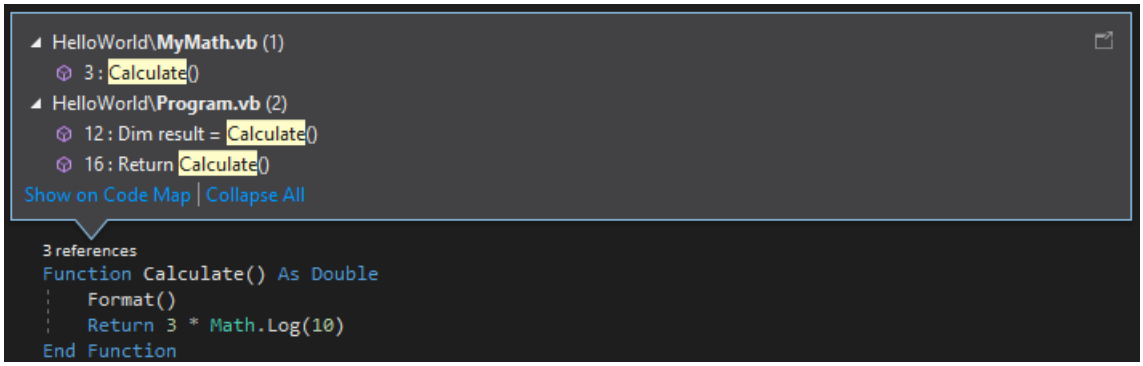
  The **Call Hierarchy** window shows the methods that call a selected method. This can be useful information when you're thinking about changing or removing the method, or when you're trying to track down a bug.

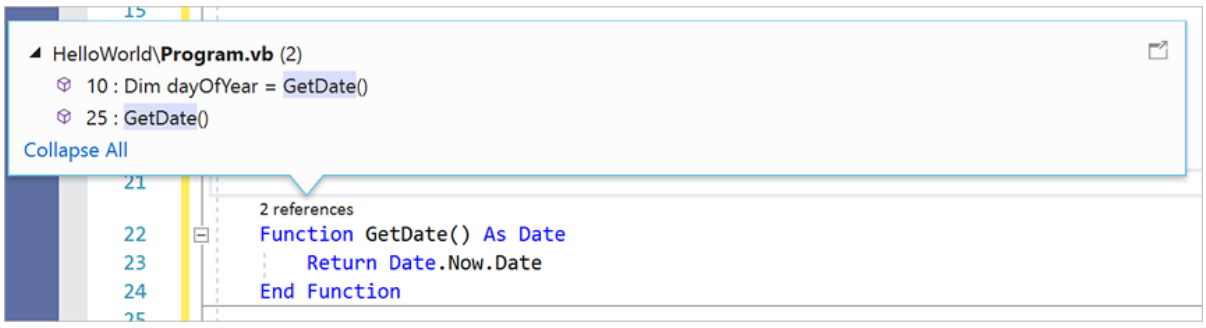




- CodeLens

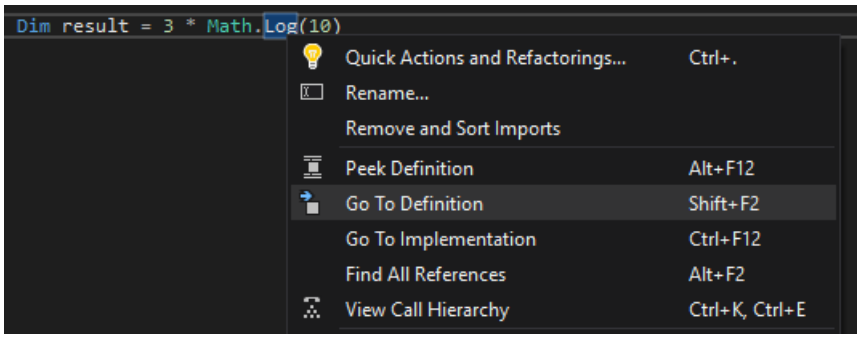
  CodeLens helps you find references to your code, changes to your code, linked bugs, work items, code reviews, and unit tests, all without leaving the editor.

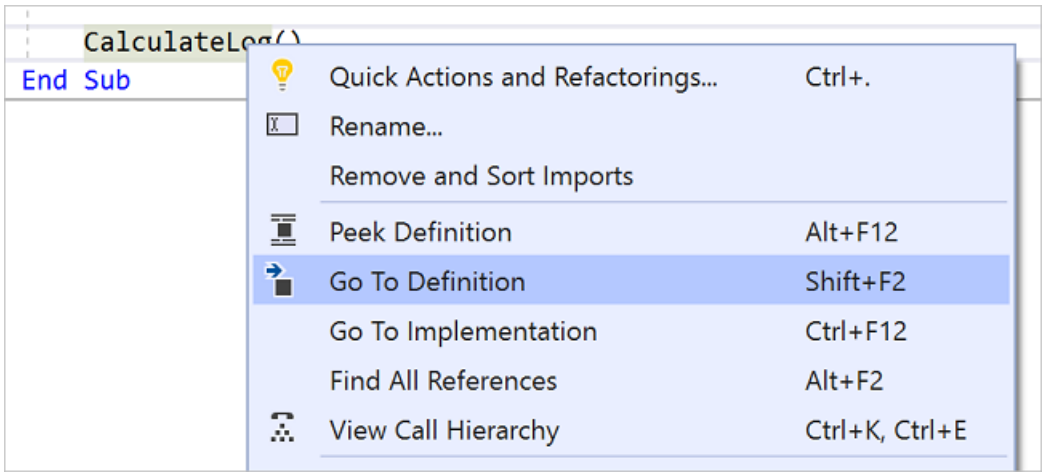




- Go To Definition

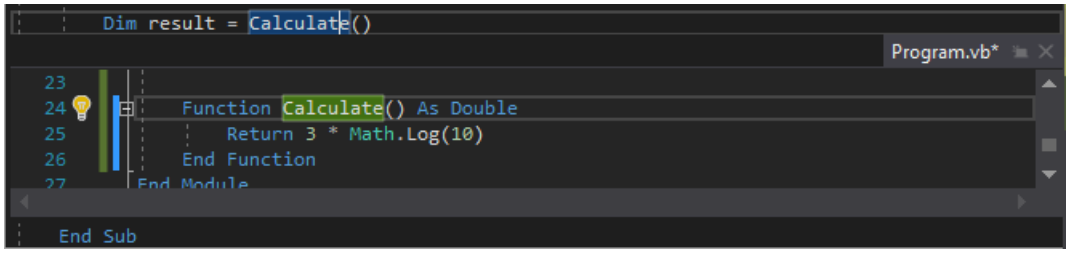
  The Go To Definition feature takes you directly to the location where a function or type is defined.

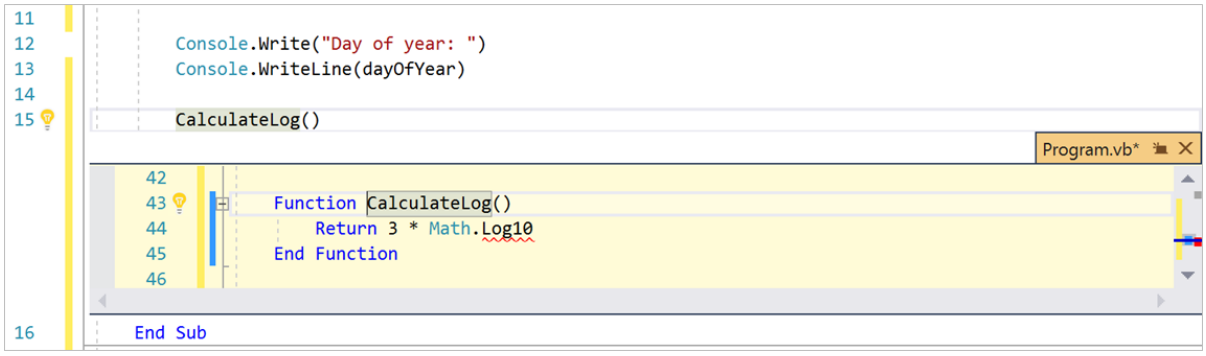




- Peek Definition

  The **Peek Definition** window shows the definition of a method or type without actually opening a separate file.



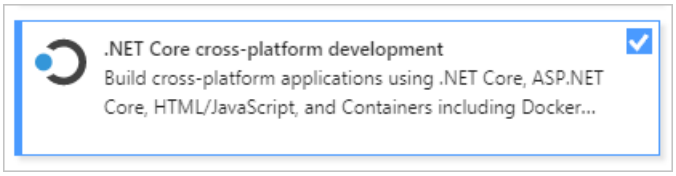


Install the Visual Studio IDE

In this section, you'll create a simple project to try out some of the things you can do with Visual Studio. You'll change the color theme, use IntelliSense as a coding aid, and debug an app to see the value of a variable during the program's execution.

To get started, download Visual Studio and install it on your system. The modular installer enables you to choose and install *workloads*, which are groups of features needed for the programming language or platform you prefer. To follow the steps for creating a program, be sure to select the **.NET Core cross-platform development** workload during installation.

To get started, download Visual Studio and install it on your system. The modular installer enables you to choose and install *workloads*, which are groups of features needed for the programming language or platform you prefer. To follow the steps for creating a program, be sure to select the **.NET Core cross-platform development** workload during installation.



When you open Visual Studio for the first time, you can optionally sign in using your Microsoft account or your work or school account.

Customize Visual Studio

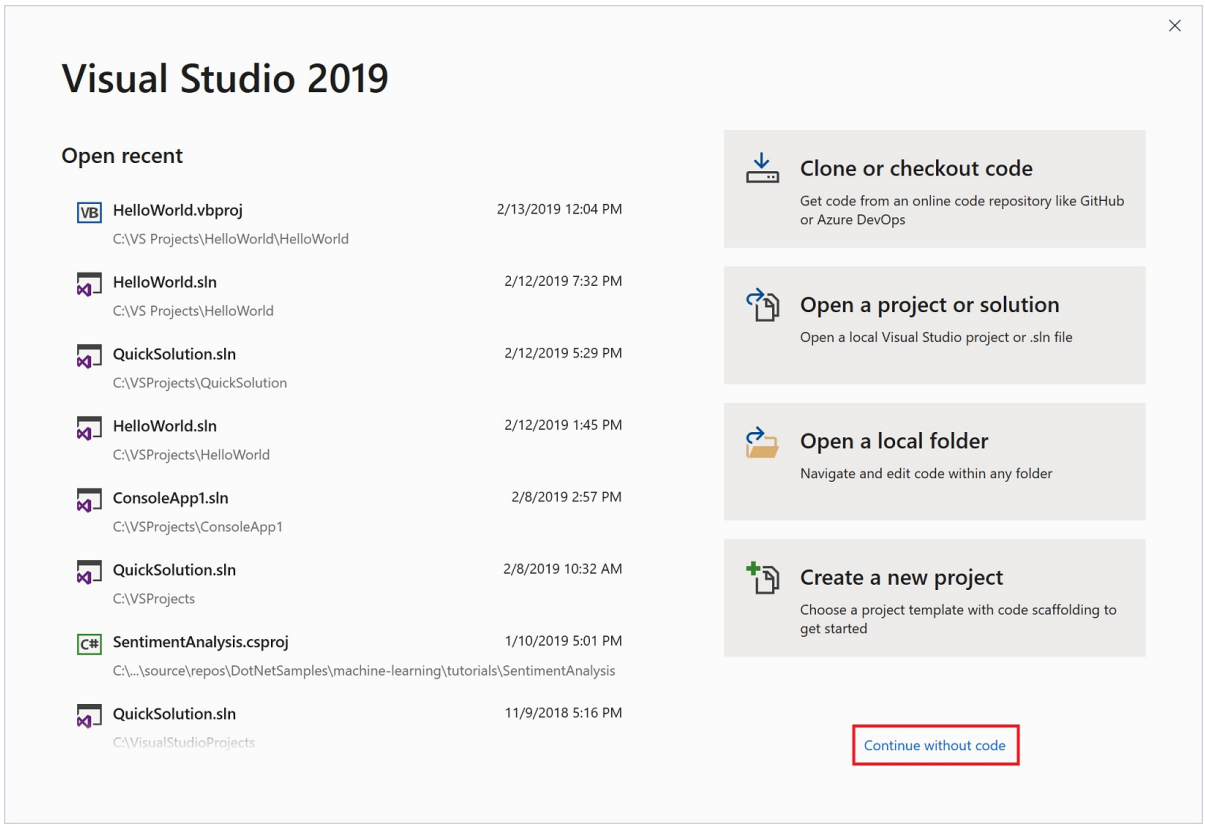
You can personalize the Visual Studio user interface, including change the default color theme.

Change the color theme

To change to the **Dark** theme:

1. Open Visual Studio.

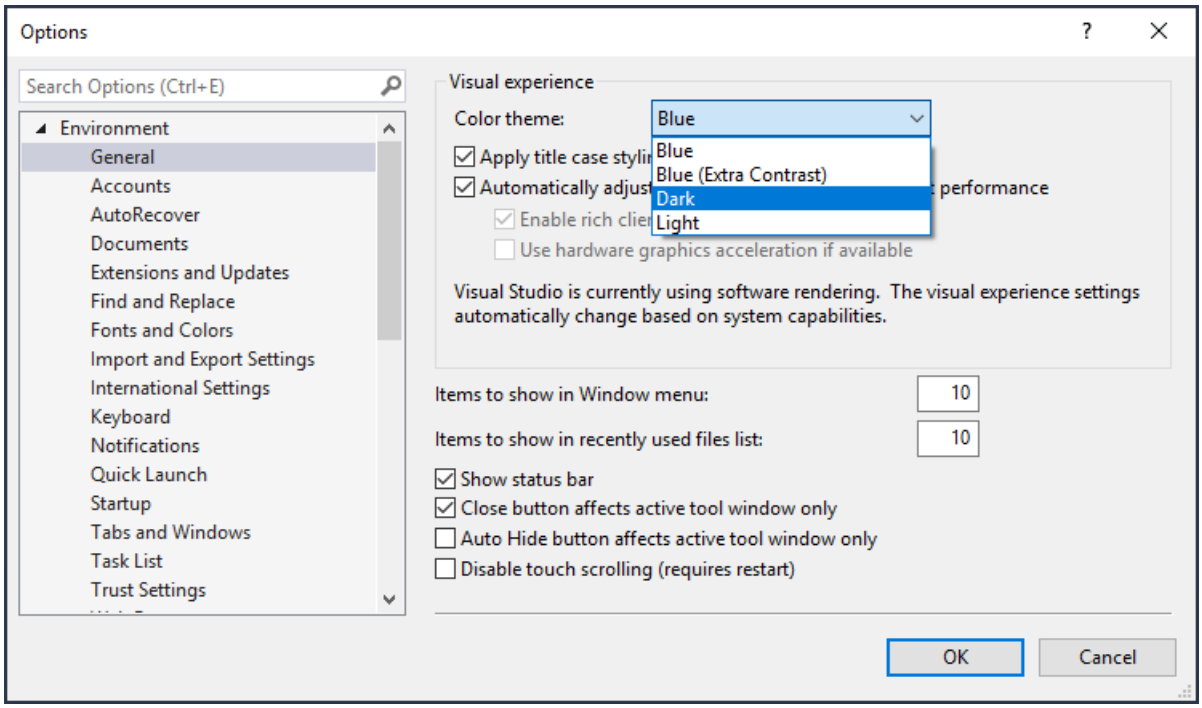
1. Open Visual Studio. On the start window, choose **Continue without code**.



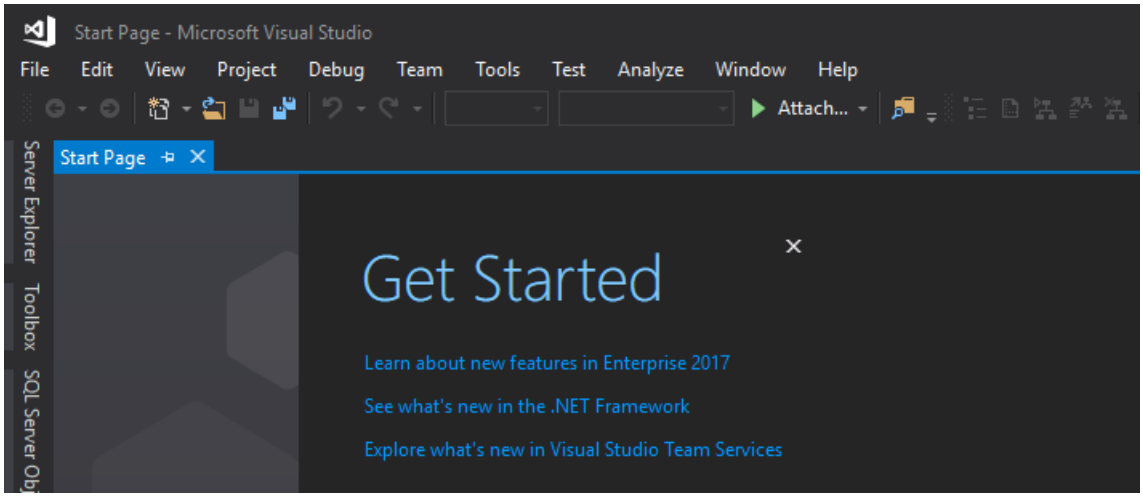
   The IDE opens.

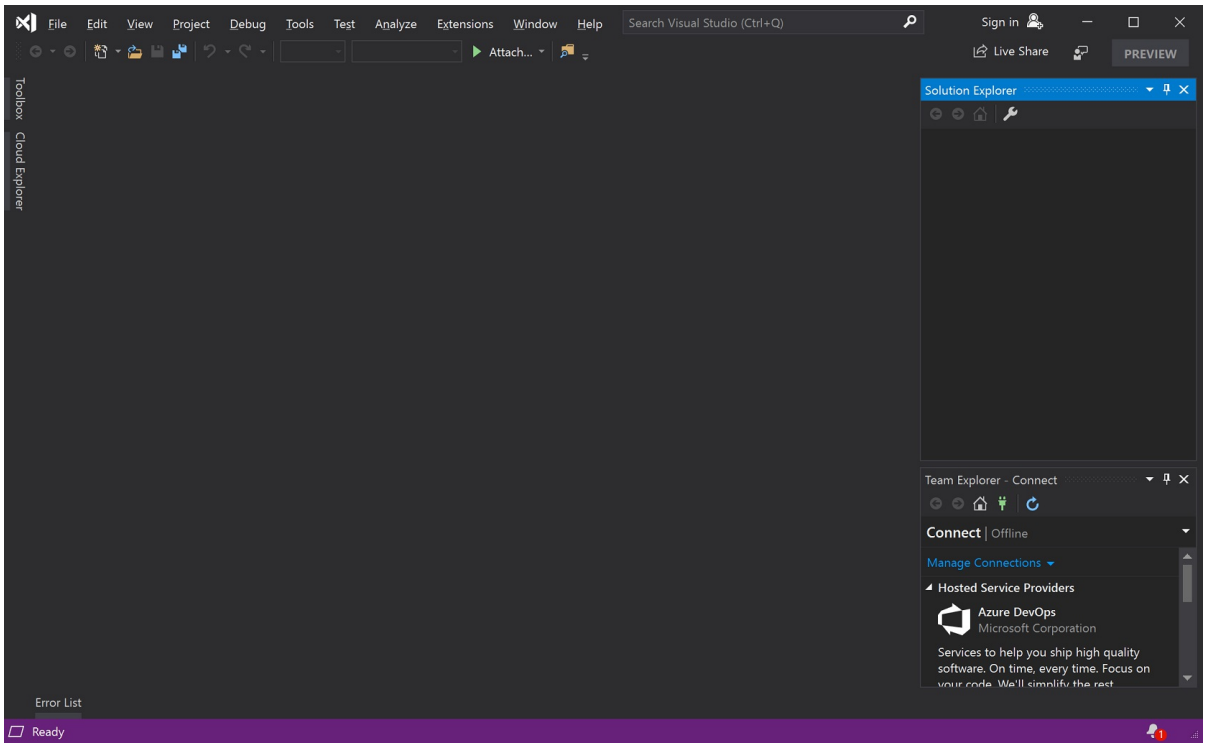
2. On the menu bar, choose **Tools** > **Options** to open the **Options** dialog.

3. On the **Environment** > **General** options page, change the **Color theme** selection to **Dark**, and then choose **OK**.



The color theme for the entire IDE changes to **Dark**.





Select environment settings

Next we'll configure Visual Studio to use environment settings tailored to Visual Basic developers.

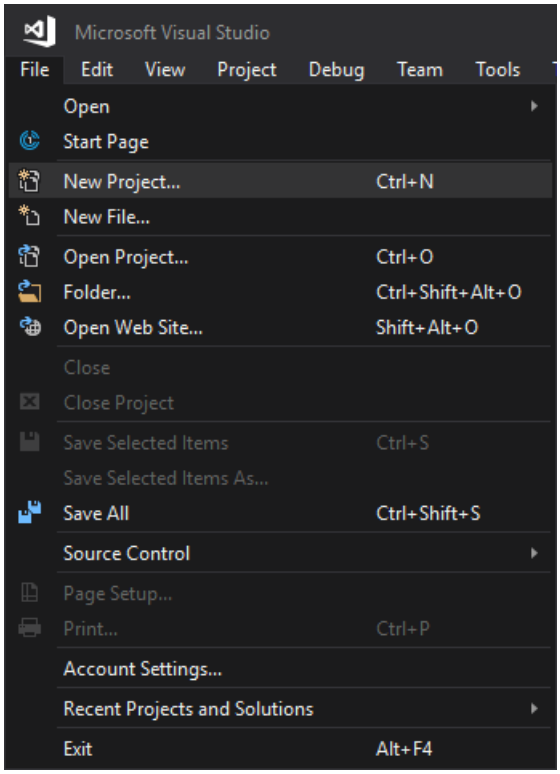
1. On the menu bar, choose **Tools** > **Import and Export Settings**.
2. In the **Import and Export Settings Wizard**, select **Reset all settings** on the first page, and then choose **Next**.
3. On the **Save Current Settings** page, select an option to save your current settings or not, and then choose **Next**. (If you haven't customized any settings, select **No, just reset settings, overwriting my current settings**.)
4. On the **Choose a Default Collection of Settings** page, choose **Visual Basic**, and then choose **Finish**.
5. On the **Reset Complete** page, choose **Close**.

To learn about other ways you can personalize the IDE, see Personalize Visual Studio.

Create a program

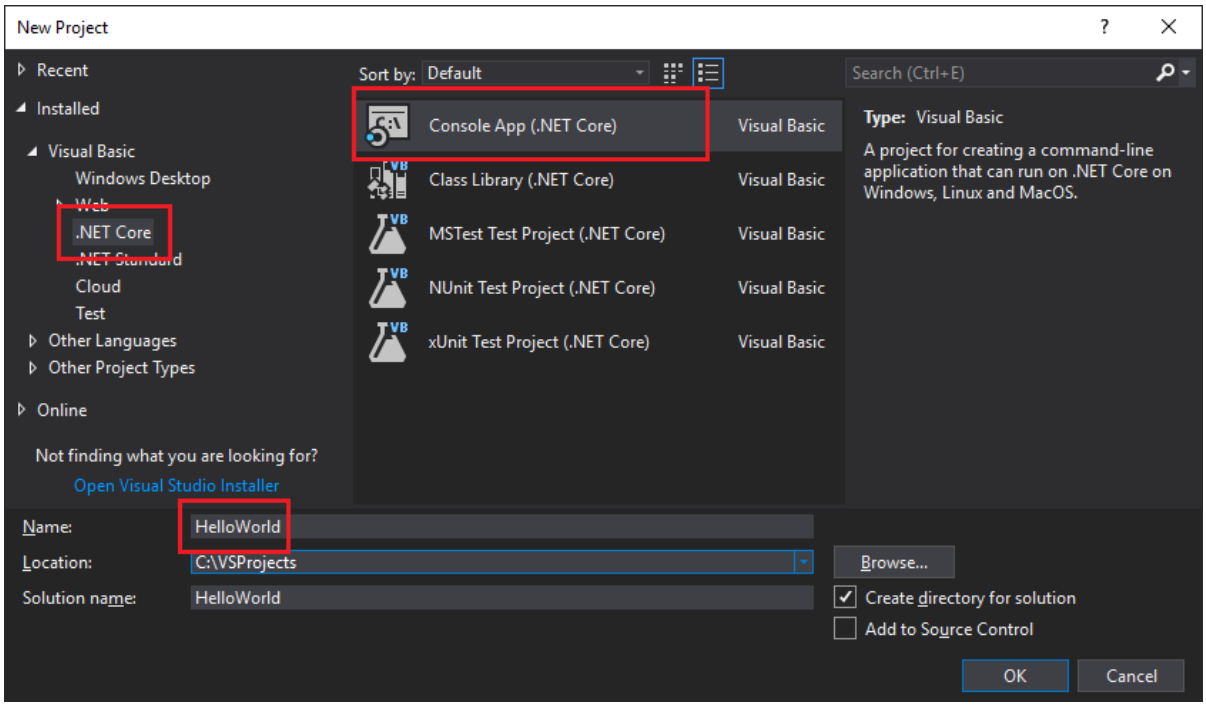
Let's dive in and create a simple program.

1. On the Visual Studio menu bar, choose **File** > **New Project**.



   The **New Project** dialog box shows several project *templates*. A template contains the basic files and settings needed for a given project type.

2. Choose the **.NET Core** category under **Visual Basic**, and then choose the **Console App (.NET Core)** template. In the **Name** text box, type **HelloWorld**, and then select the **OK** button.

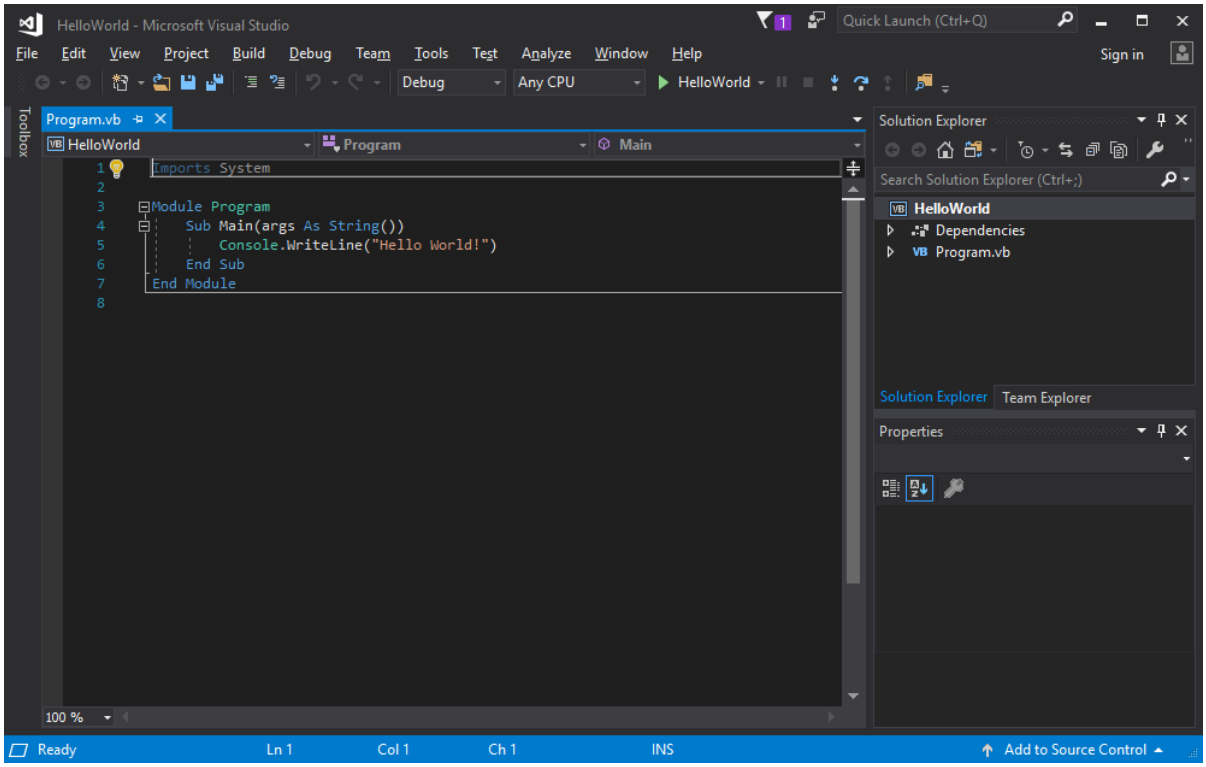


NOTE

If you don't see the **.NET Core** category, you need to install the **.NET Core cross-platform development** workload. To do this, choose the **Open Visual Studio Installer** link on the bottom left of the **New Project** dialog. After Visual Studio Installer opens, scroll down and select the **.NET Core cross-platform development** workload, and then select **Modify**.

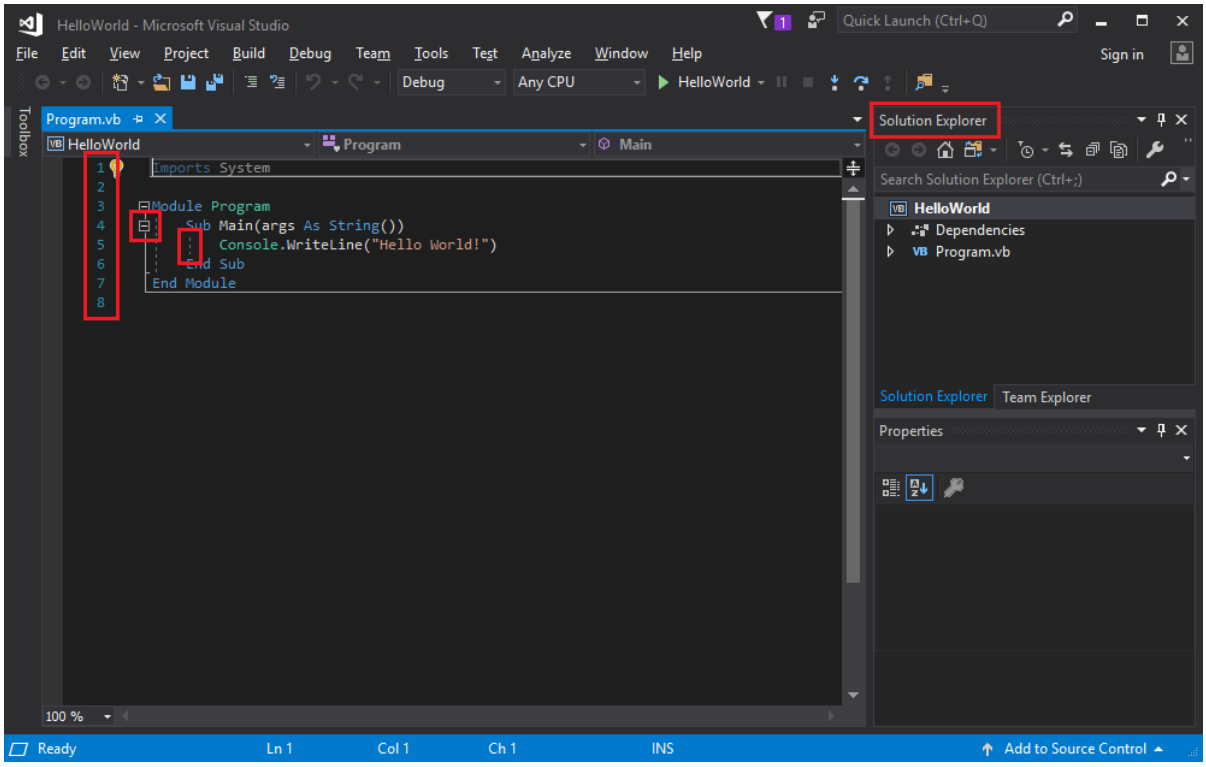
Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following:



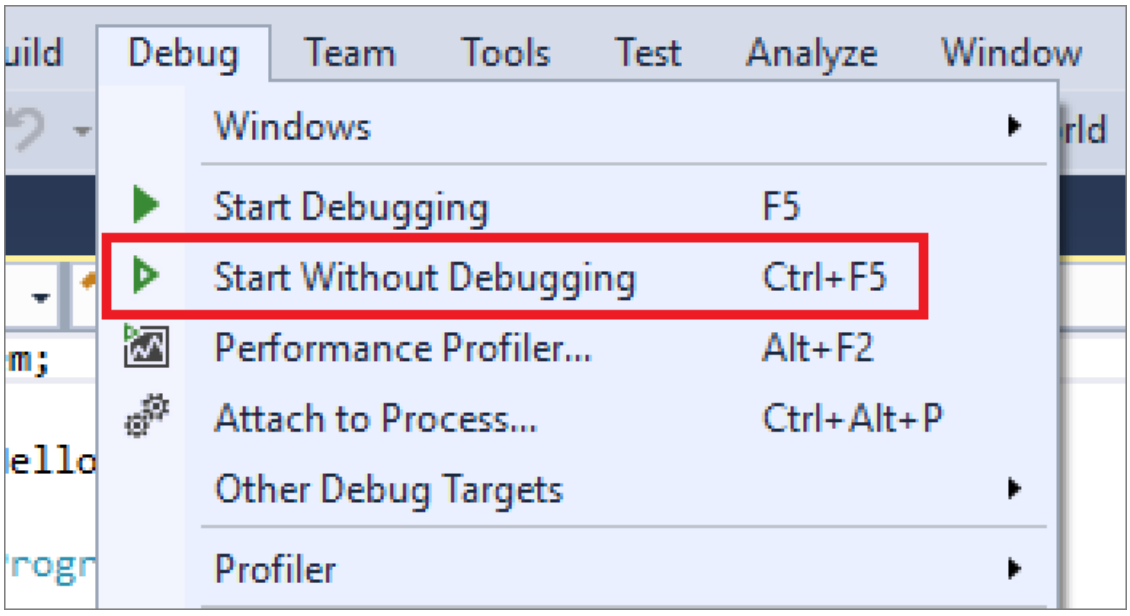
The Visual Basic code for the app appears in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, and line numbers

help you locate code later. You can choose the small, boxed minus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need, helping to minimize onscreen clutter. The project files are listed on the right side in a window called **Solution Explorer**.

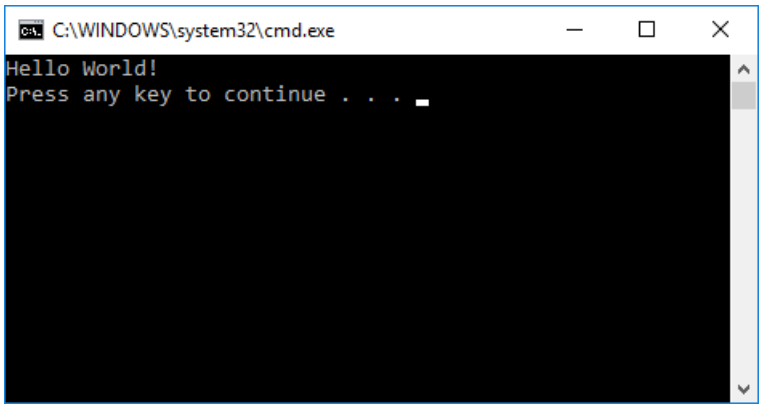


There are other menus and tool windows available, but let's move on for now.

3. Now, start the app. You can do this by choosing **Start Without Debugging** from the **Debug** menu on the menu bar. You can also press **Ctrl**+**F5**.



Visual Studio builds the app, and a console window opens with the message **Hello World!**. You now have a running app!



4. To close the console window, press any key on your keyboard.

5. Let's add some additional code to the app. Add the following Visual Basic code before the line that says `Console.WriteLine("Hello World!")`:

   ```
   Console.WriteLine("What is your name?")
   Dim name = Console.ReadLine()
   ```

   This code displays **What is your name?** in the console window, and then waits until the user enters some text followed by the **Enter** key.

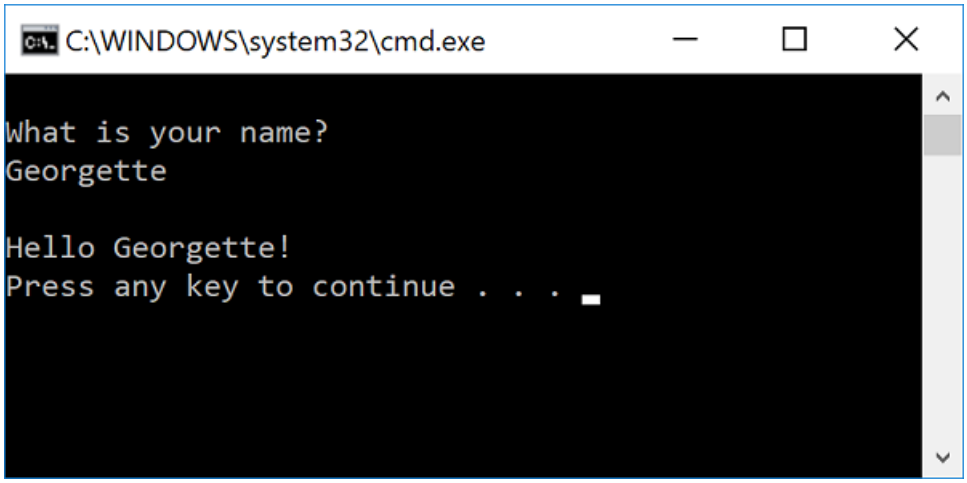
6. Change the line that says `Console.WriteLine("Hello World!")` to the following code:

   ```
   Console.WriteLine("Hello " + name + "!")
   ```

7. Run the app again by pressing **Ctrl**+**F5**.

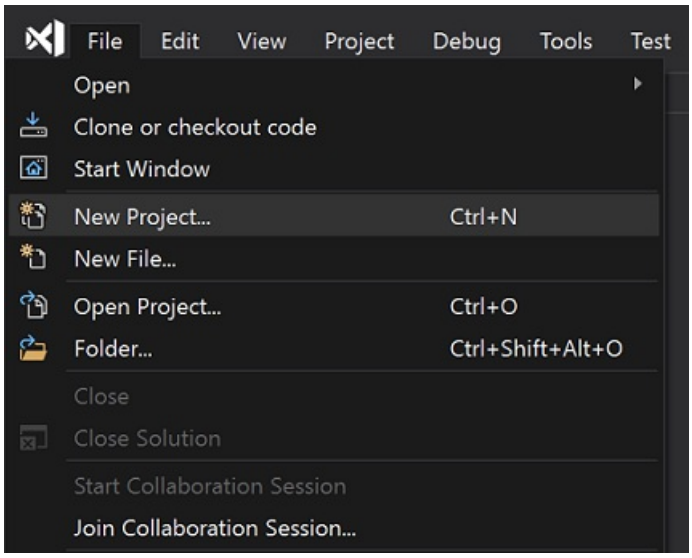
   Visual Studio rebuilds the app, and a console window opens and prompts you for your name.

8. Enter your name in the console window and press **Enter**.



9. Press any key to close the console window and stop the running program.

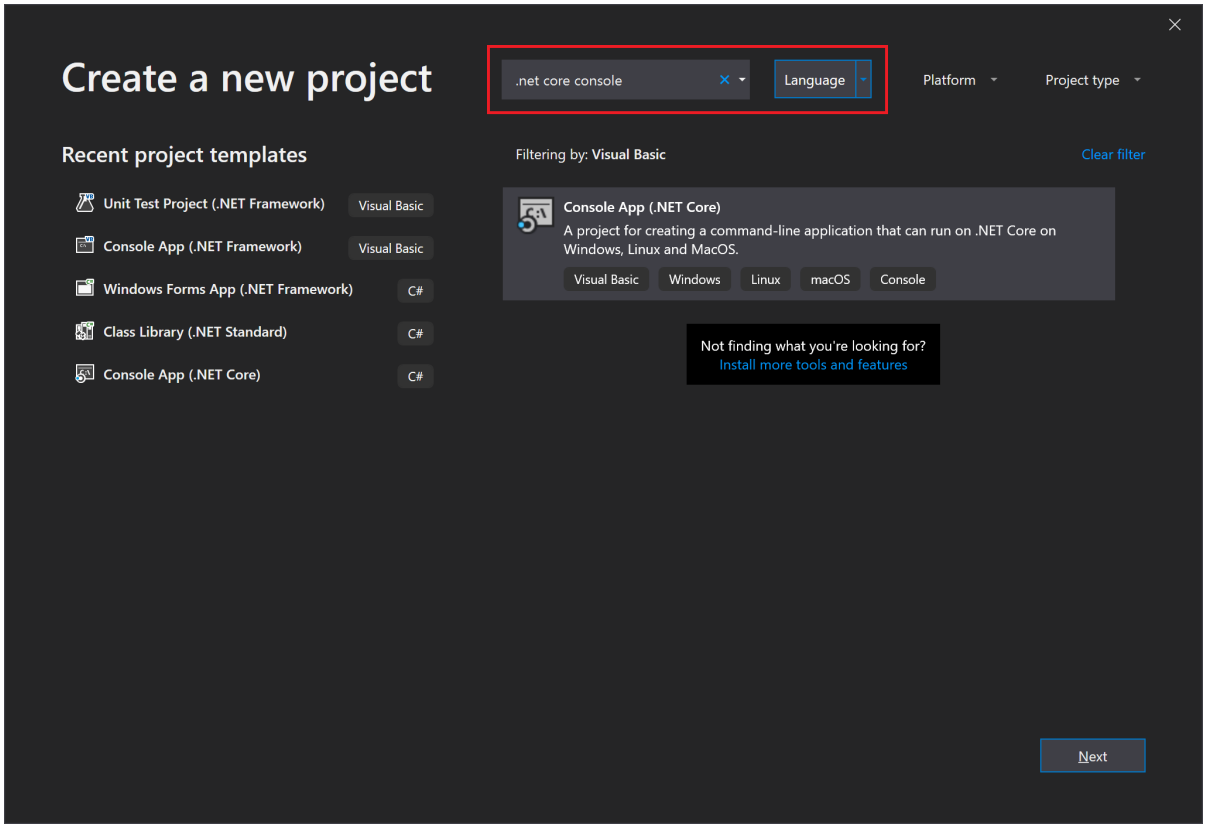
1. On the Visual Studio menu bar, choose **File** > **New Project**.



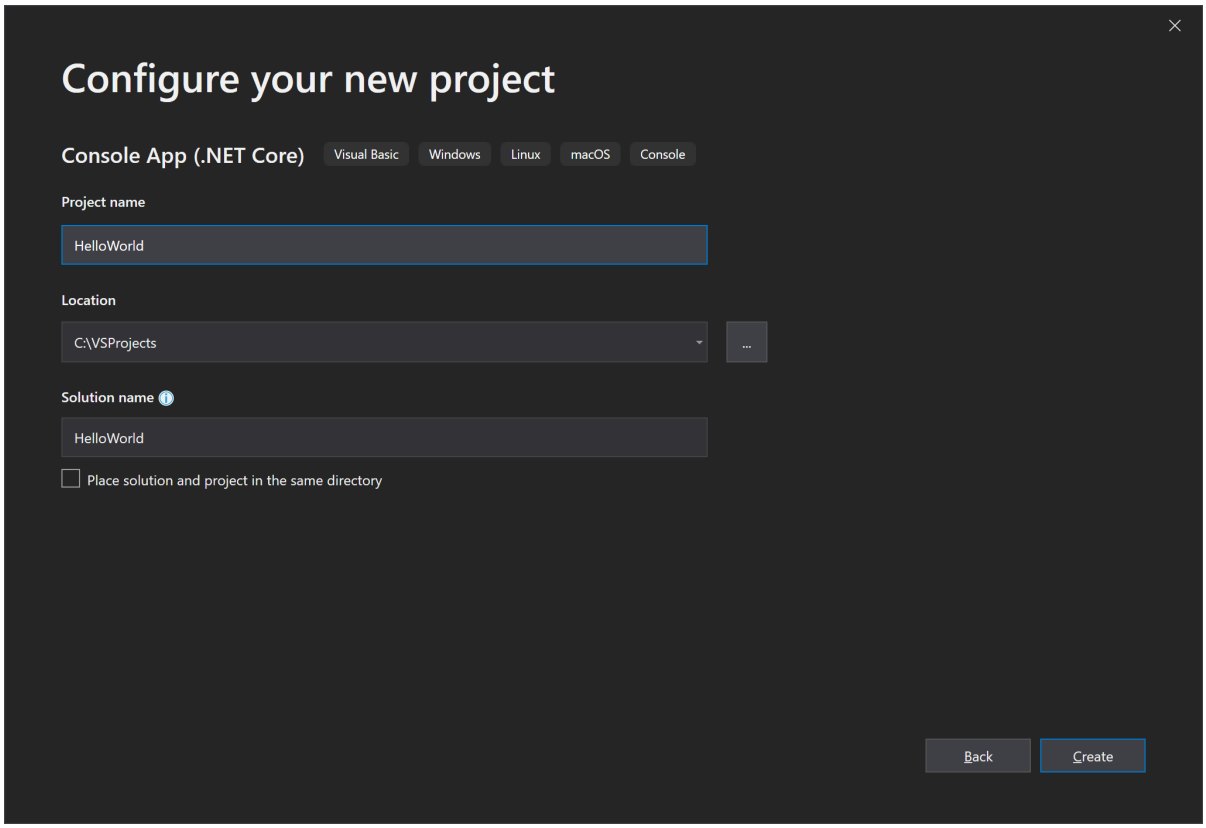
The **Create a new project** window opens and shows several project *templates*. A template contains the basic files and settings needed for a given project type.

2. To find the template we want, type or enter **.net core console** in the search box. The list of available templates is automatically filtered based on the keywords you entered. You can further filter the template results by choosing **Visual Basic** from the **Language** drop-down list.

3. Select the **Console App (.NET Core)** template, and then choose **Next**.

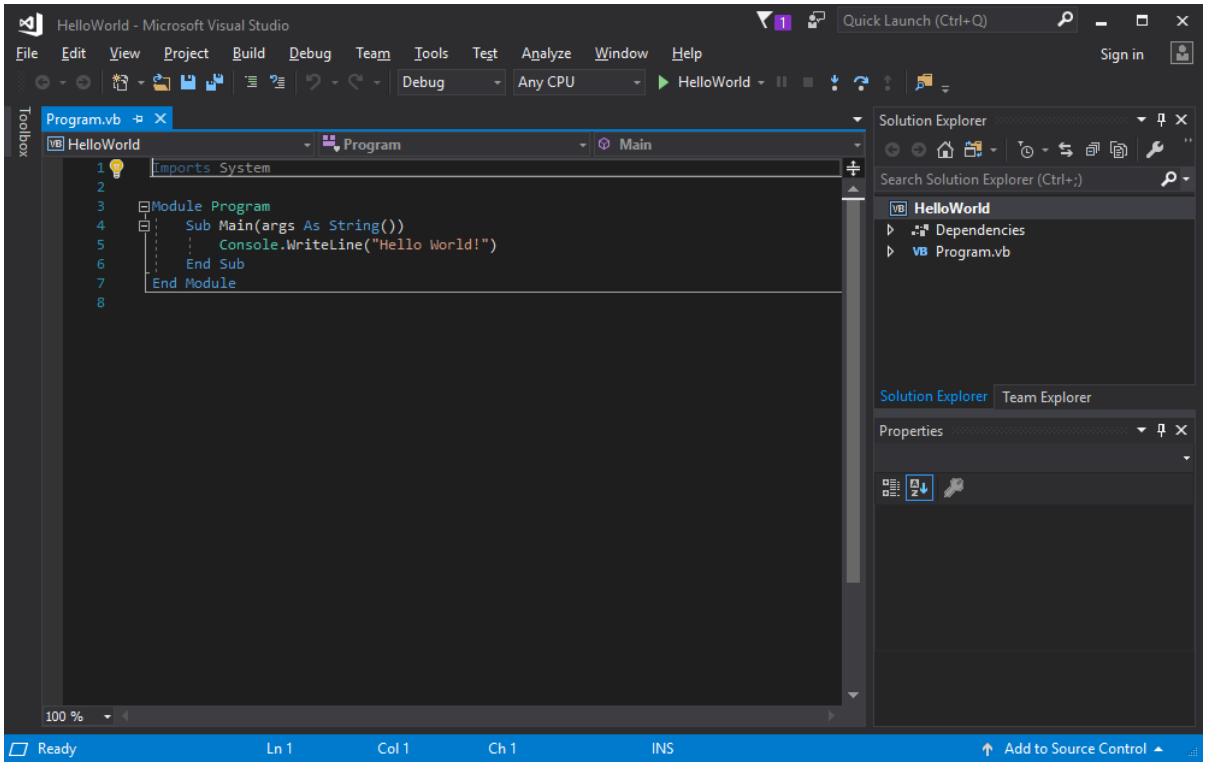


4. In the **Configure your new project** window, enter **HelloWorld** in the **Project name** box, optionally change the directory location for your project files, and then choose **Create**.

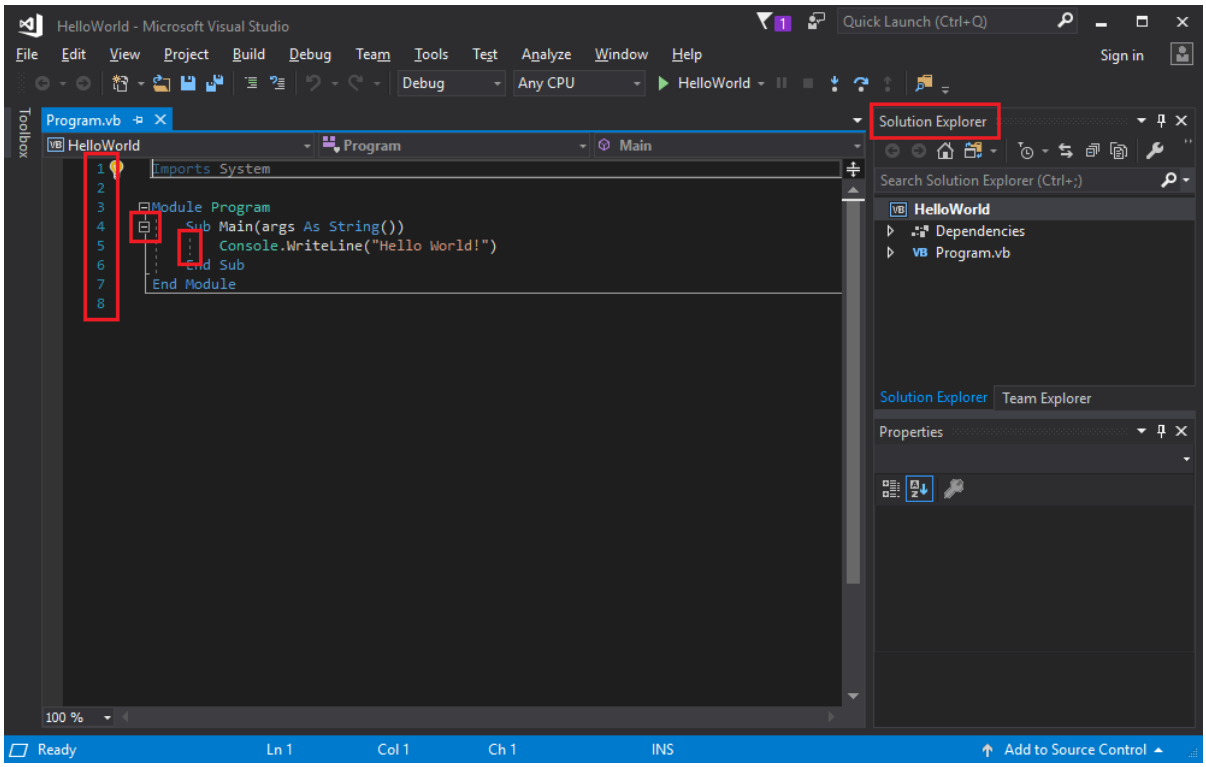


Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following:

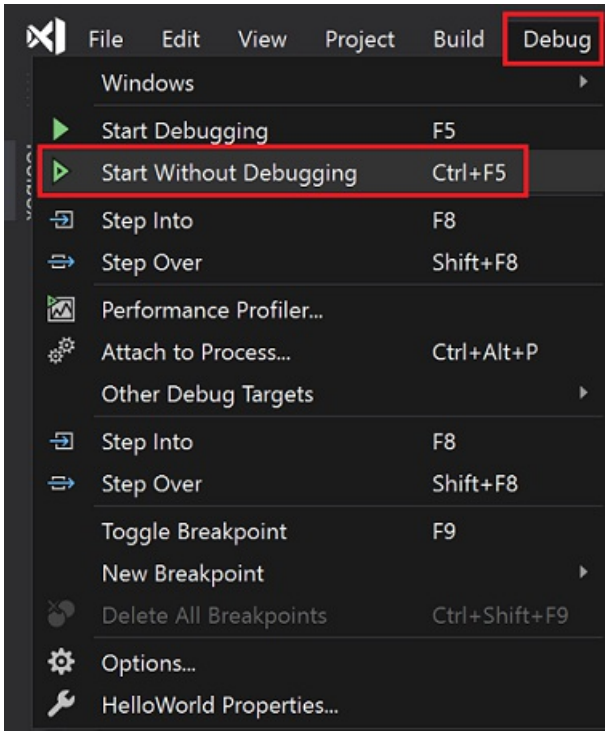


The Visual Basic code for the app appears in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, and line numbers help you locate code later. You can choose the small, boxed minus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need, helping to minimize onscreen clutter. The project files are listed on the right side in a window called **Solution Explorer**.

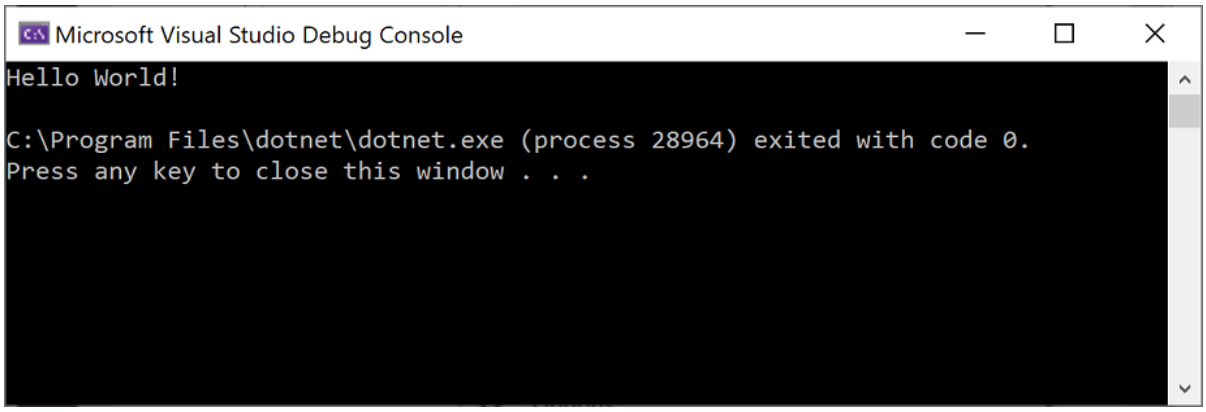


There are other menus and tool windows available, but let's move on for now.

5. Now, start the app. You can do this by choosing **Start Without Debugging** from the **Debug** menu on the menu bar. You can also press **Ctrl**+**F5**.



Visual Studio builds the app, and a console window opens with the message **Hello World!**. You now have a running app!



6. To close the console window, press any key on your keyboard.

7. Let's add some additional code to the app. Add the following Visual Basic code before the line that says `Console.WriteLine("Hello World!")`:

   ```
   Console.WriteLine("What is your name?")
   Dim name = Console.ReadLine()
   ```

   This code displays **What is your name?** in the console window, and then waits until the user enters some text followed by the **Enter** key.

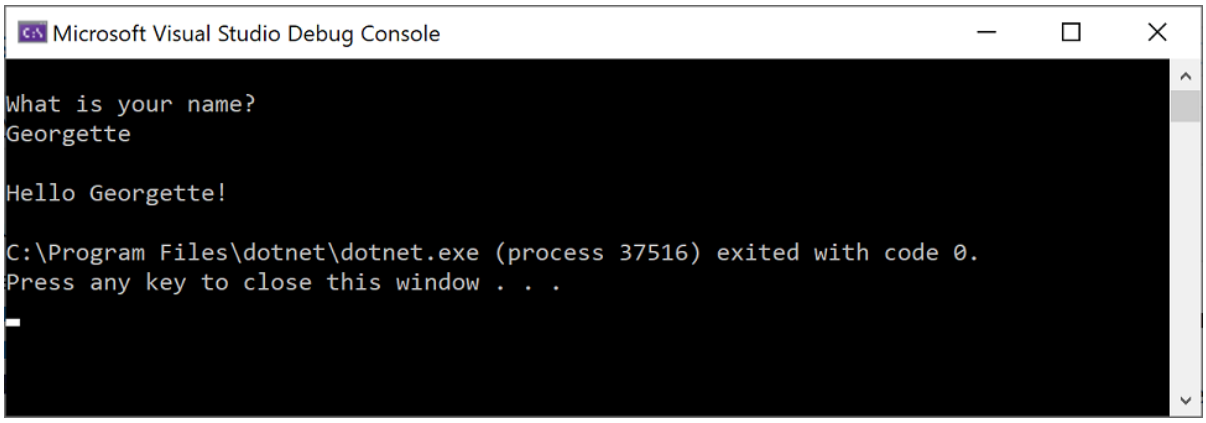
8. Change the line that says `Console.WriteLine("Hello World!")` to the following code:

   ```
   Console.WriteLine("Hello " + name + "!")
   ```

9. Run the app again by pressing **Ctrl**+**F5**.

   Visual Studio rebuilds the app, and a console window opens and prompts you for your name.

10. Enter your name in the console window and press **Enter**.



11. Press any key to close the console window and stop the running program.

Use refactoring and IntelliSense

Let's look at a couple of the ways that refactoring and IntelliSense can help you code more efficiently.

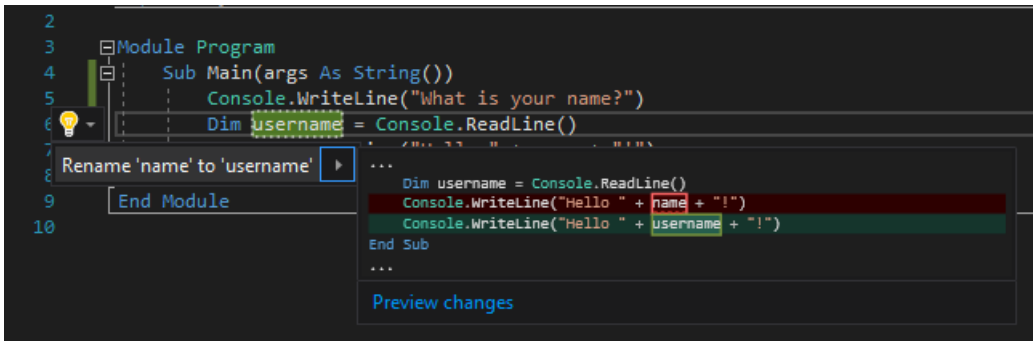
First, let's rename the `name` variable:

1. Double-click the `name` variable to select it.

2. Type in the new name for the variable, **username**.

Notice that a gray box appears around the variable, and a light bulb appears in the margin.

3. Select the light bulb icon to show the available Quick Actions. Select **Rename 'name' to 'username'**.

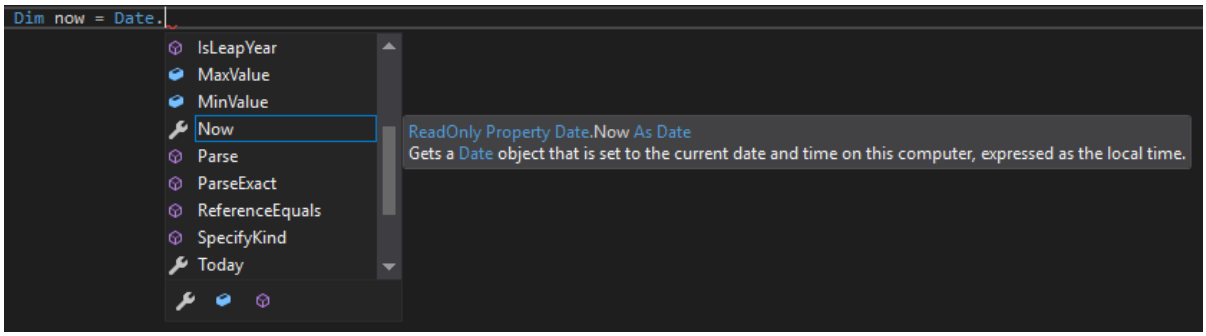


The variable is renamed across the project, which in our case is only two places.

4. Now let's take a look at IntelliSense. Below the line that says `Console.WriteLine("Hello " + username + "!")`, type the following code fragment:

```
Dim now = Date.
```

A box displays the members of the DateTime class. In addition, the description of the currently selected member displays in a separate box.



5. Select the member named **Now**, which is a property of the class, by double-clicking on it or selecting it using the up or down arrow keys and then pressing **Tab**.

6. Below that, type in or paste the following lines of code:

```
Dim dayOfYear = now.DayOfYear
Console.Write("Day of year: ")
Console.WriteLine(dayOfYear)
```

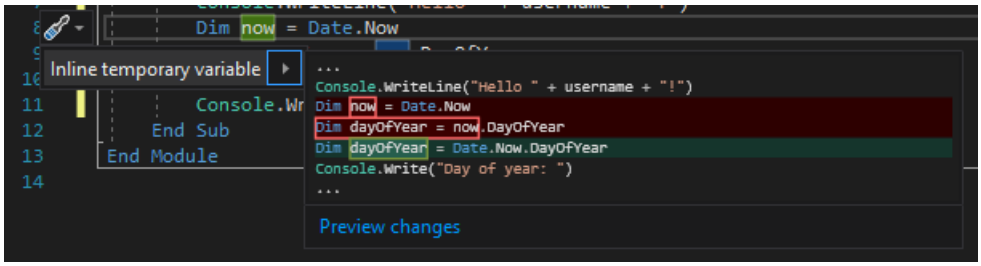
> **TIP**
>
> Console.Write is a little different to Console.WriteLine in that it doesn't add a line terminator after it prints. That means that the next piece of text that's sent to the output will print on the same line. You can hover over each of these methods in your code to see their description.

7. Next, we'll use refactoring again to make the code a little more concise. Click on the variable `now` in the line `Dim now = Date.Now`.

Notice that a little screwdriver icon appears in the margin on that line.

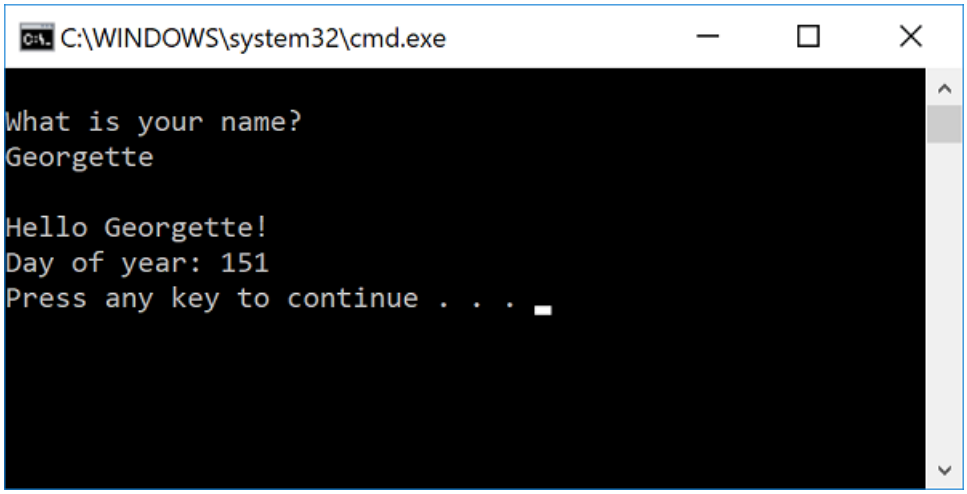
8. Click the screwdriver icon to see what suggestions Visual Studio has available. In this case, it's showing the Inline temporary variable refactoring to remove a line of code without changing the overall behavior of the

code:

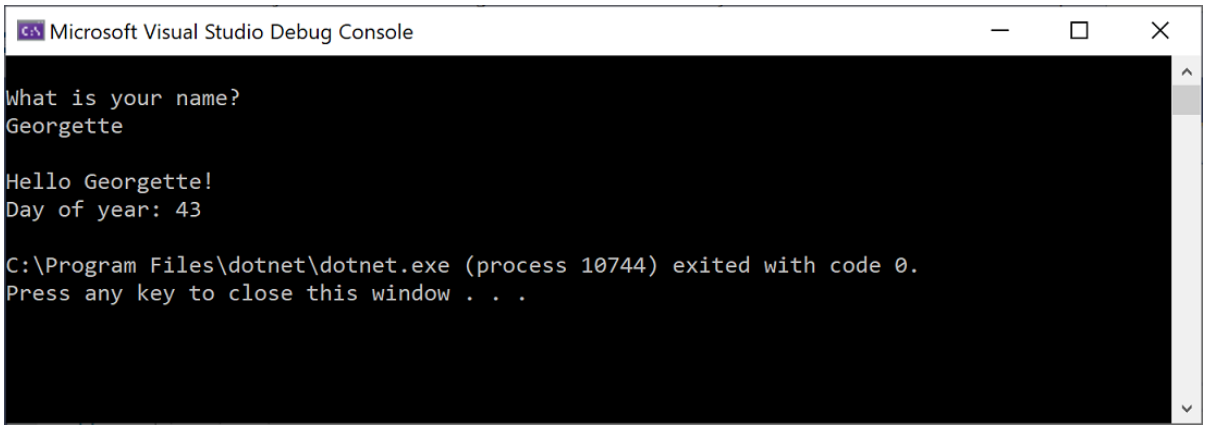


9. Click **Inline temporary variable** to refactor the code.

10. Run the program again by pressing **Ctrl**+**F5**. The output looks something like this:



10. Run the program again by pressing **Ctrl**+**F5**. The output looks something like this:



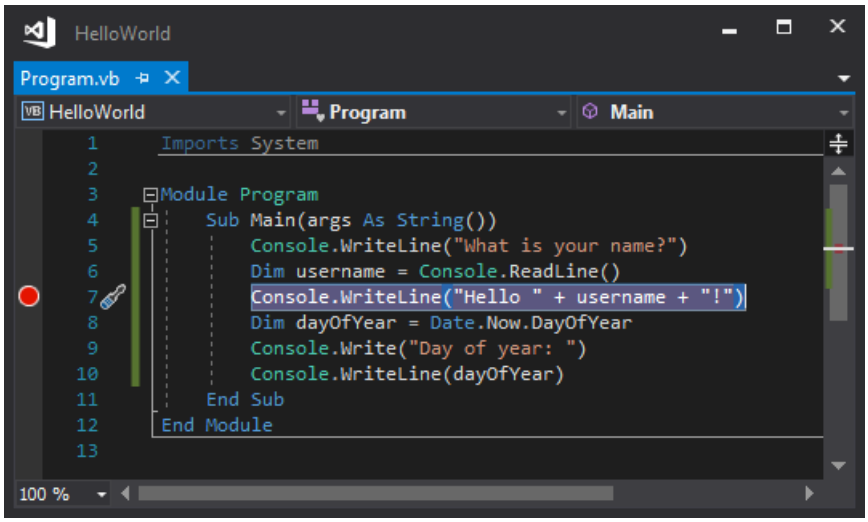
Debug code

When you write code, you need to run it and test it for bugs. Visual Studio's debugging system lets you step through code one statement at a time and inspect variables as you go. You can set *breakpoints* that stop execution of the code at a particular line. You can observe how the value of a variable changes as the code runs, and more.

Let's set a breakpoint to see the value of the `username` variable while the program is "in flight".

1. Find the line of code that says `Console.WriteLine("Hello " + username + "!")`. To set a breakpoint on this line of code, that is, to make the program pause execution at this line, click in the far left margin of the editor. You can also click anywhere on the line of code and then press **F9**.

   A red circle appears in the far left margin, and the code is highlighted in red.

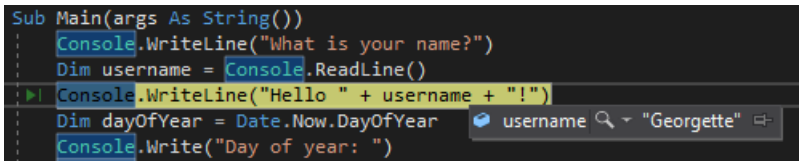


2. Start debugging by selecting **Debug** > **Start Debugging** or by pressing **F5**.

3. When the console window appears and asks for your name, type it in and press **Enter**.

   The focus returns to the Visual Studio code editor and the line of code with the breakpoint is highlighted in yellow. This signifies that it's the next line of code that the program will execute.

4. Hover your mouse over the `username` variable to see its value. Alternatively, you can right-click on `username` and select **Add Watch** to add the variable to the **Watch** window, where you can also see its value.



5. To let the program run to completion, press **F5** again.

To get more details about debugging in Visual Studio, see Debugger feature tour.

Next steps

Explore Visual Studio further by following along with one of these introductory articles:

Learn to use the code editor

Learn about projects and solutions

See also

- Discover more Visual Studio features
- Visit visualstudio.microsoft.com
- Read The Visual Studio blog

Learn to use the code editor

4/17/2019 • 6 minutes to read • Edit Online

In this 10-minute introduction to the code editor in Visual Studio, we'll add code to a file to look at some of the ways that Visual Studio makes writing, navigating, and understanding code easier.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

This article assumes you're already familiar with Visual Basic. If you aren't, we suggest you look at a tutorial such as Get started with Visual Basic in Visual Studio first.

> **TIP**
> To follow along with this article, make sure you have the Visual Basic settings selected for Visual Studio. For information about selecting settings for the integrated development environment (IDE), see Select environment settings.

Create a new code file

Start by creating a new file and adding some code to it.

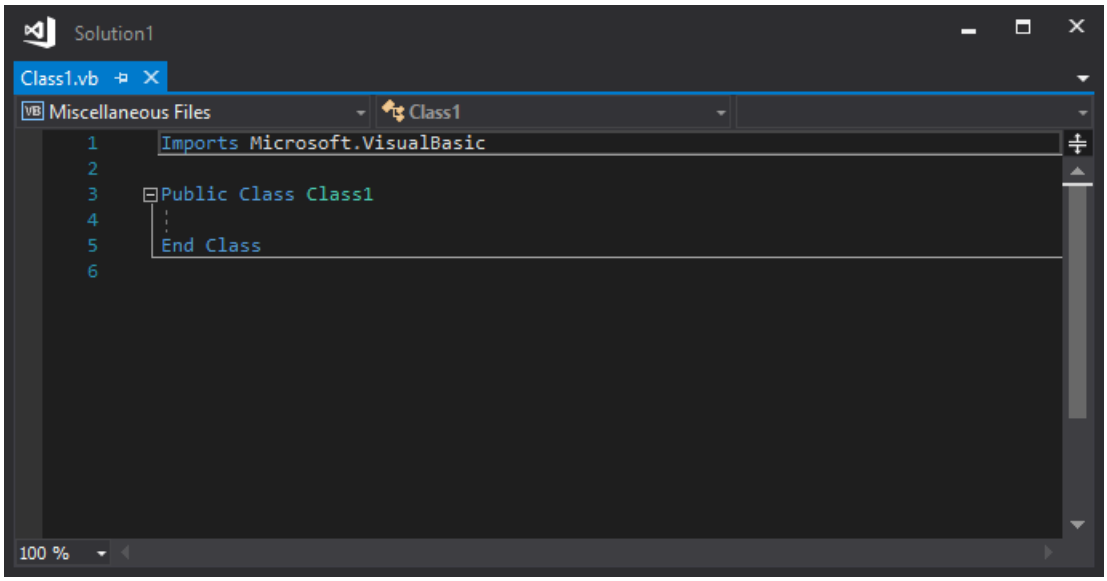
1. Open Visual Studio.

1. Open Visual Studio. Press **Esc** or click **Continue without code** on the start window to open the development environment.

2. From the **File** menu on the menu bar, choose **New File**.

3. In the **New File** dialog box, under the **General** category, choose **Visual Basic Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a Visual Basic class. (You can already notice that you don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers, such as syntax highlighting. All you need is a code file!)

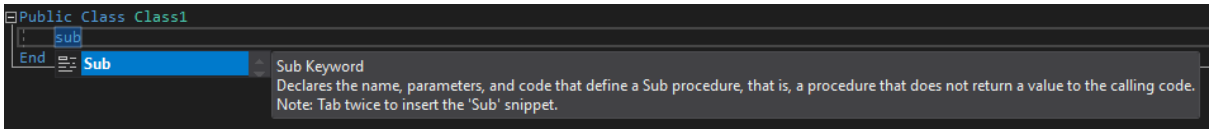


Use code snippets

Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including Visual Basic, C#, and C++. Let's add the Visual Basic **Sub** snippet to our file.

1. Place your cursor above the line that says `End Class`, and type **sub**.

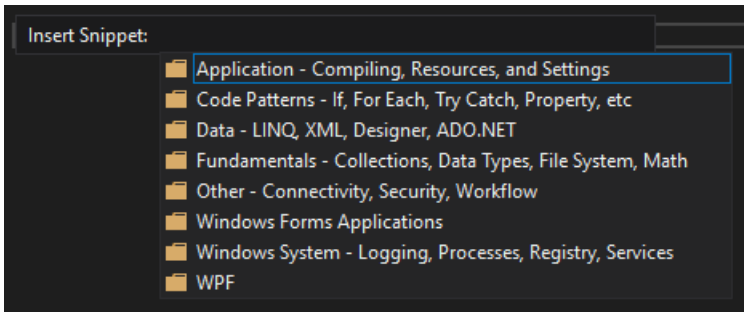
   A pop-up dialog box appears with information about the `Sub` keyword and how to insert the **Sub** code snippet.



2. Press **Tab** twice to insert the code snippet.

   The outline for the Sub procedure `MySub()` is added to the file.

The available code snippets vary for different programming languages. You can look at the available code snippets for Visual Basic by choosing **Edit** > **IntelliSense** > **Insert Snippet** (or press **Ctrl**+**K**, **Ctrl**+**X**). For Visual Basic, code snippets are available for the following categories:



There are snippets for determining if a file exists on the computer, writing to a text file, reading a registry value, executing a SQL query, creating a For Each...Next statement, and many more.

Comment out code

The toolbar, which is the row of buttons under the menu bar in Visual Studio, can help make you more productive as you code. For example, you can toggle IntelliSense completion mode, increase or decrease a line indent, or comment out code that you don't want to compile. (IntelliSense is a coding aid that displays a list of matching

methods, amongst other things.) In this section, we'll comment out some code.

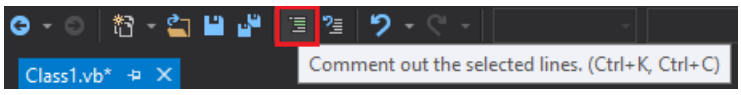
1. Paste the following code into the `MySub()` procedure body.

```
' _words is a string array that we'll sort alphabetically
Dim _words = New String() {
"the",
"quick",
"brown",
"fox",
"jumps"
}

Dim morewords = New String() {
"over",
"the",
"lazy",
"dog"
}

Dim query = From word In _words
            Order By word.Length
            Select word
```

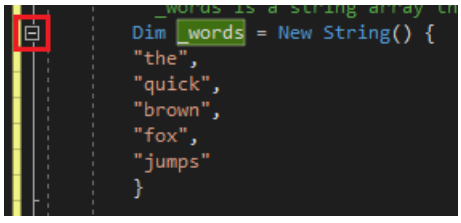
2. We're not using the `morewords` array, but we may use it later so we don't want to completely delete it. Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing curly brace, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**K**, **Ctrl**+**C**.

   The Visual Basic comment character `'` is added to the beginning of each selected line to comment out the code.

Collapse code blocks

You can collapse sections of code to focus just on the parts that are of interest to you. To practice, let's collapse the `_words` array to one line of code. Choose the small gray box with the minus sign inside it in the margin of the line that says `Dim _words = New String() {`. Or, if you're a keyboard user, place the cursor anywhere in the array definition and press **Ctrl**+**M**, **Ctrl**+**M**.



The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

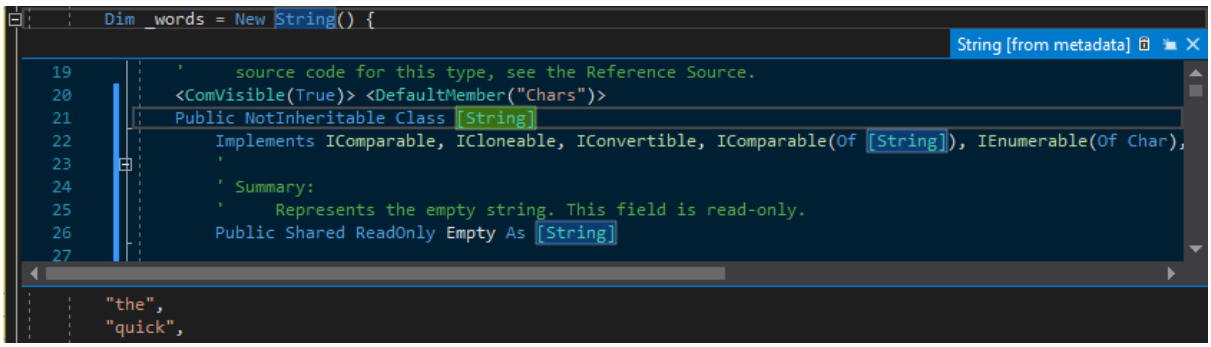
View symbol definitions

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing **Go to Definition** anywhere the symbol is referenced. An

even quicker way that doesn't move your focus away from the file you're working in is to use Peek Definition. Let's peek at the definition of the `String` type.

1. Right-click on the word `String` and choose **Peek Definition** from the content menu. Or, press **Alt**+**F12**.

   A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

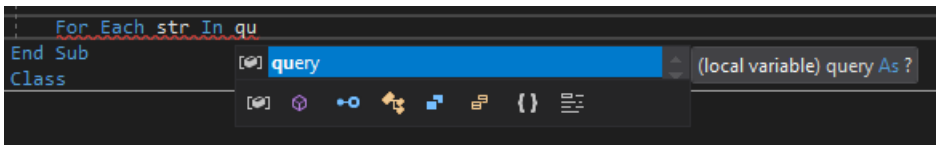
Use IntelliSense to complete words

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it. Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
For Each str In qu
```

   You see IntelliSense show you **Quick Info** about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense's word completion functionality, press **Tab**.

3. Finish off the code block to look like the following code.

```
For Each str In query
    Console.WriteLine(str)
Next
```

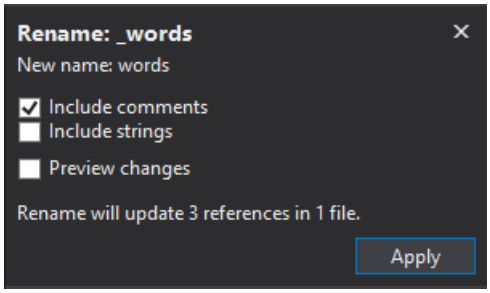
Refactor a name

Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `_words` variable to `words`.

1. Place your cursor over the definition of the `_words` variable and choose **Rename** from the right-click or context menu.

   A pop-up **Rename** dialog box appears at the top right of the editor.

2. With the variable `_words` still selected, type in the desired name of **words**. Notice that the reference to `words` in the query is also automatically renamed. Before you press **Enter** or click **Apply**, select the **Include comments** checkbox in the **Rename** pop-up box.



3. Press **Enter** or click **Apply**.

   Both occurrences of `words` are renamed, as well as the reference to `words` in the code comment.

Next steps

Learn about projects and solutions

See also

- Code snippets
- Navigate code
- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

Learn about projects and solutions using Visual Basic

7/23/2019 • 8 minutes to read • Edit Online

In this introductory article, we'll explore what it means to create a *solution* and a *project* in Visual Studio. A solution is a container that's used to organize one or more related code projects, for example a class library project and a corresponding test project. We'll look at the properties of a project and some of the files it can contain. We'll also create a reference from one project to another.

> **TIP**
>
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

> **TIP**
>
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

We'll construct a solution and project from scratch as an educational exercise to understand the concept of a project. In your general use of Visual Studio, you'll likely use some of the various project *templates* that Visual Studio offers when you create a new project.

> **NOTE**
>
> Solutions and projects aren't required to develop apps in Visual Studio. You can also just open a folder that contains code and start coding, building, and debugging. For example, if you clone a GitHub repo, it might not contain Visual Studio projects and solutions. For more information, see Develop code in Visual Studio without projects or solutions.

Solutions and projects

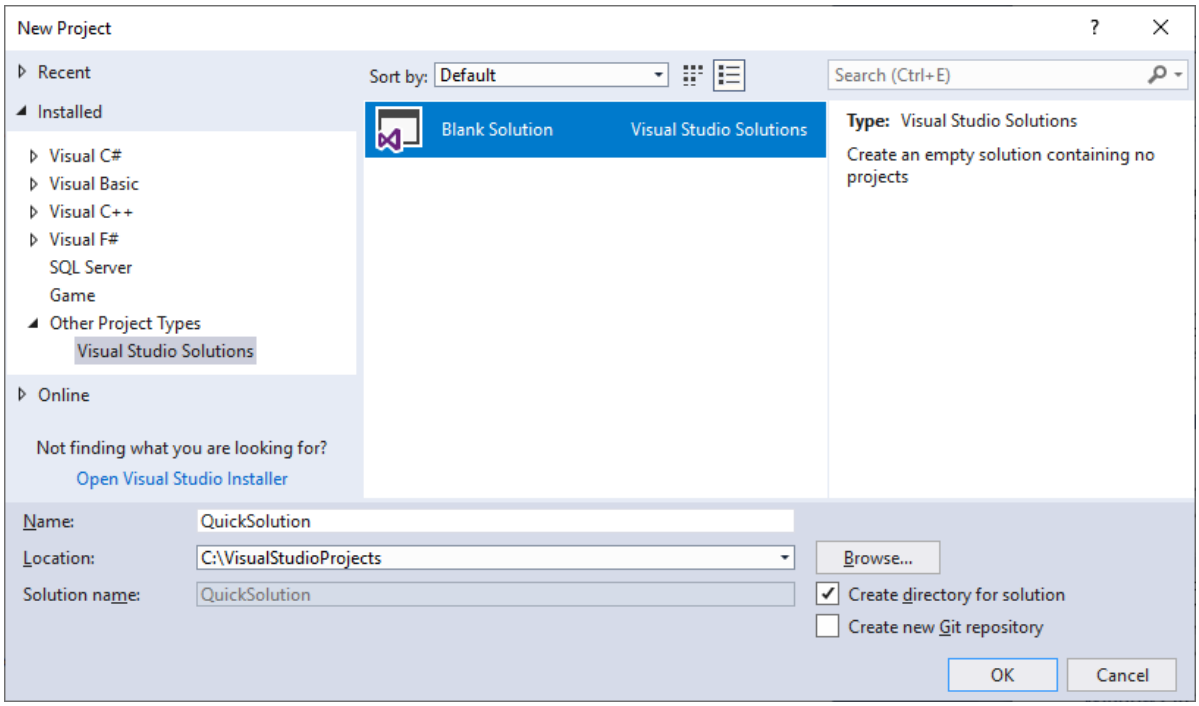
Despite its name, a solution is not an "answer". A solution is simply a container used by Visual Studio to organize one or more related projects. When you open a solution in Visual Studio, it automatically loads all the projects that the solution contains.

Create a solution

We'll start our exploration by creating an empty solution. After you get to know Visual Studio, you probably won't find yourself creating empty solutions very often. When you create a new project, Visual Studio automatically creates a solution to house the project if there's not a solution already open.

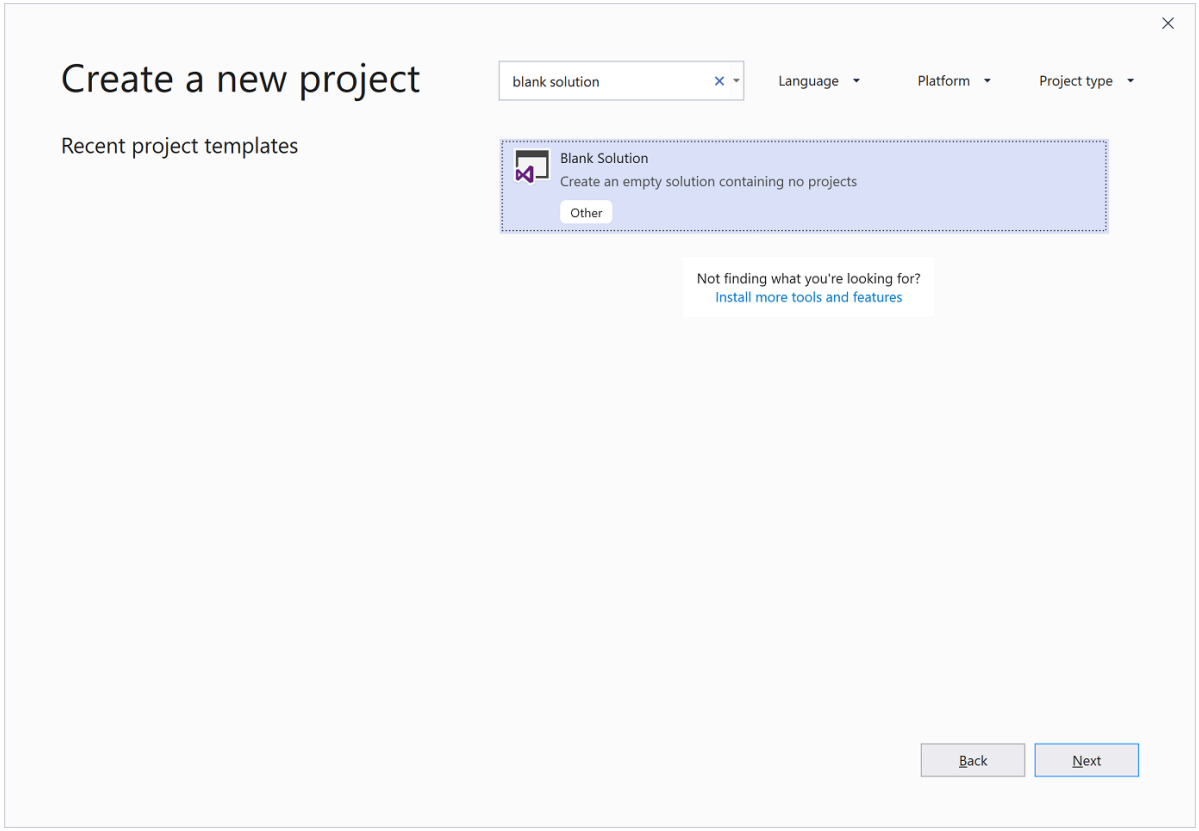
1. Open Visual Studio.
2. On the menu bar, choose **File** > **New** > **Project**.

   The **New Project** dialog box opens.
3. In the left pane, expand **Other Project Types**, then choose **Visual Studio Solutions**. In the center pane, choose the **Blank Solution** template. Name your solution **QuickSolution**, and then choose **OK**.



The **Start Page** closes, and a solution appears in **Solution Explorer** on the right-hand side of the Visual Studio window. You'll probably use **Solution Explorer** often, to browse the contents of your projects.

1. Open Visual Studio.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** page, enter **blank solution** into the search box, select the **Blank Solution** template, and then choose **Next**.



4. Name the solution **QuickSolution**, and then choose **Create**.

   A solution appears in **Solution Explorer** on the right-hand side of the Visual Studio window. You'll probably use **Solution Explorer** often, to browse the contents of your projects.

Add a project

Now let's add our first project to the solution. We'll start with an empty project and add the items we need to the project.

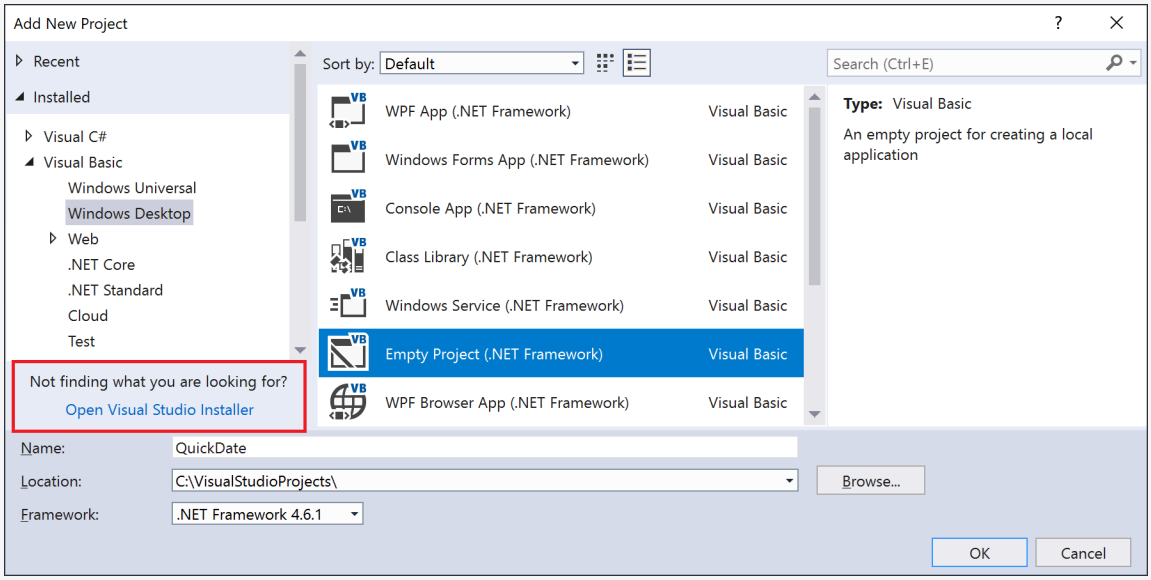
1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, choose **Add** > **New Project**.

   The **Add New Project** dialog box opens.

2. In the left pane, expand **Visual Basic** and choose **Windows Desktop**. Then, in the middle pane, choose the **Empty Project (.NET Framework)** template. Name the project **QuickDate**, then choose the **OK** button.

   A project named QuickDate appears beneath the solution in **Solution Explorer**. Currently it contains a single file called *App.config*.

> **NOTE**
>
> If you don't see **Visual Basic** in the left pane of the dialog box, you need to install the **.NET desktop development** Visual Studio *workload*. Visual Studio uses workload-based installation to only install the components you need for the type of development you do. An easy way to install a new workload is to choose the **Open Visual Studio Installer** link in the bottom left corner of the **Add New Project** dialog box. After Visual Studio Installer launches, choose the **.NET desktop development** workload and then the **Modify** button.



1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, choose **Add** > **New Project**.

   A dialog box opens that says **Add a new project**.

2. Enter the text **empty** into the search box at the top, and then select **Visual Basic** under **Language**.

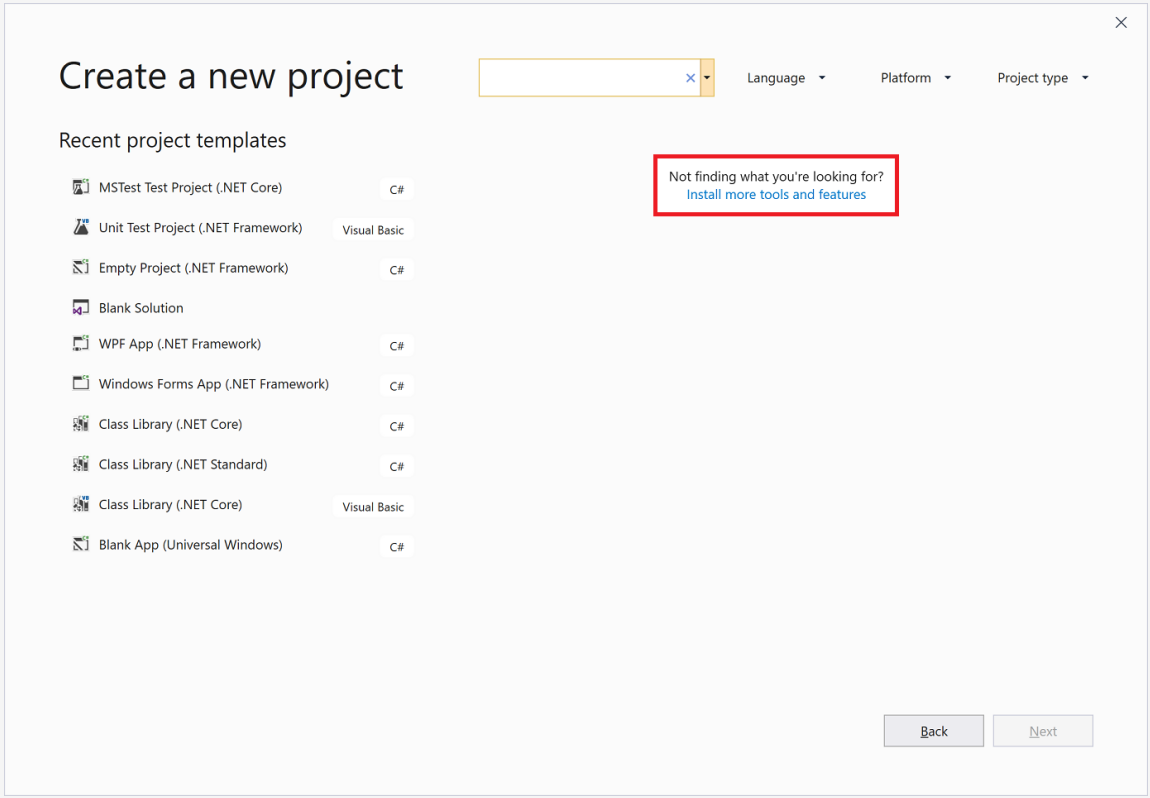
3. Select the **Empty Project (.NET Framework)** template, and then choose **Next**.

4. Name the project **QuickDate**, then choose **Create**.

   A project named QuickDate appears beneath the solution in **Solution Explorer**. Currently it contains a single file called *App.config*.

NOTE

If you don't see the **Empty Project (.NET Framework)** template, you need to install the **.NET desktop development** Visual Studio *workload*. Visual Studio uses workload-based installation to only install the components you need for the type of development you do. An easy way to install a new workload when you're creating a new project is to choose the **Install more tools and features** link under the text that says **Not finding what you're looking for?**. After Visual Studio Installer launches, choose the **.NET desktop development** workload and then the **Modify** button.



Add an item to the project

We have an empty project. Let's add a code file.

1. From the right-click or context menu of the **QuickDate** project in **Solution Explorer**, choose **Add** > **New Item**.

   The **Add New Item** dialog box opens.

2. Expand **Common Items**, then choose **Code**. In the middle pane choose the **Class** item template. Name the class **Calendar**, and then choose the **Add** button.

   A file named *Calendar.vb* is added to the project. The *.vb* on the end is the file extension that's given to Visual Basic code files. The file appears in the visual project hierarchy in **Solution Explorer**, and its contents open in the editor.

3. Replace the contents of the *Calendar.vb* file with the following code:

```
Class Calendar
    Public Shared Function GetCurrentDate() As Date
        Return DateTime.Now.Date
    End Function
End Class
```

The `Calendar` class contains a single function, `GetCurrentDate`, that returns the current date.

4. Open the project properties by double-clicking **My Project** in **Solution Explorer**. On the **Application** tab, change **Application type** to **Class Library**. This step is necessary to build the project successfully.

5. Build the project by right-clicking on **QuickDate** in **Solution Explorer** and choosing **Build**. You should see a successful build message in the **Output** window.

Add a second project

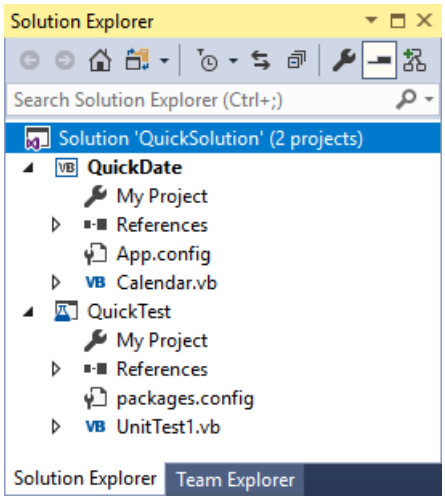
It's common for solutions to contain more than one project, and often these projects reference each other. Some projects in a solution might be class libraries, some executable applications, and some might be unit test projects or websites.

Let's add a unit test project to our solution. This time we'll start from a project template so we don't have to add an additional code file to the project.

1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, choose **Add** > **New Project**.

2. In the left pane, expand **Visual Basic** and choose the **Test** category. In the middle pane, choose the **Unit Test Project (.NET Framework)** project template. Name the project **QuickTest**, and then choose **OK**.

   A second project is added to **Solution Explorer**, and a file named *UnitTest1.vb* opens in the editor.



2. In the **Add a new project** dialog box, enter the text **unit test** into the search box at the top, and then select **Visual Basic** under **Language**.

3. Choose the **Unit Test Project (.NET Framework)** project template, and then choose **Next**.

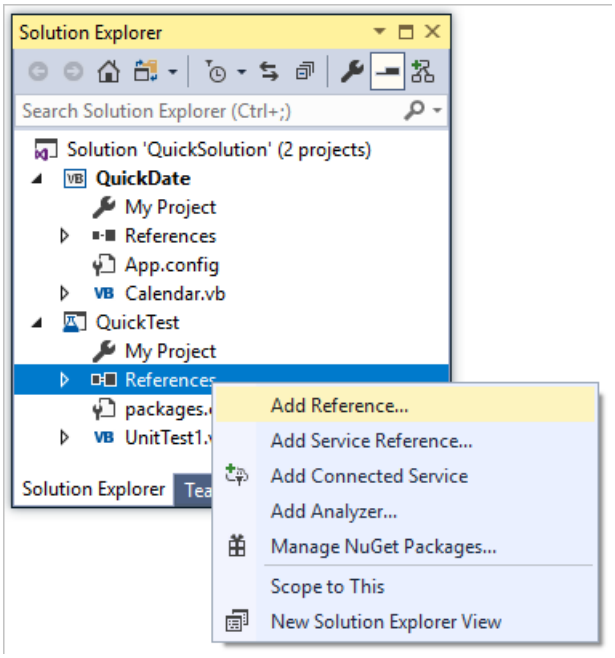
4. Name the project **QuickTest**, and then choose **Create**.

   A second project is added to **Solution Explorer**, and a file named *UnitTest1.vb* opens in the editor.

Add a project reference

We're going to use the new unit test project to test our method in the **QuickDate** project, so we need to add a reference to that project. This creates a *build dependency* between the two projects, meaning that when you build the solution, **QuickDate** is built before **QuickTest**.

1. Choose the **References** node in the **QuickTest** project, and from the right-click or context menu, choose **Add Reference**.



The **Reference Manager** dialog box opens.

2. In the left pane, expand **Projects** and choose **Solution**. In the middle pane, choose the checkbox next to **QuickDate**, and then choose the **OK** button.

   A reference to the **QuickDate** project is added.

Add test code

1. Now we'll add test code to the Visual Basic code file. Replace the contents of *UnitTest1.vb* with the following code.

```
<TestClass()> Public Class UnitTest1

    <TestMethod()> Public Sub TestGetCurrentDate()
        Assert.AreEqual(Date.Now.Date, QuickDate.Calendar.GetCurrentDate())
    End Sub

End Class
```

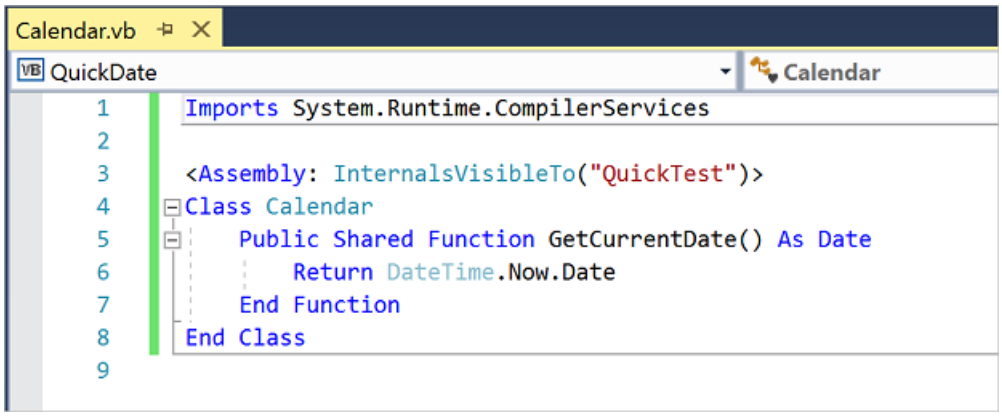
   You'll see a red squiggle under some of the code. We'll fix this error by making the test project a friend assembly to the **QuickDate** project.

2. Back in the **QuickDate** project, open the *Calendar.vb* file if it's not already open, and add the following Imports statement and InternalsVisibleToAttribute attribute, to resolve the error in the test project.

```
Imports System.Runtime.CompilerServices

<Assembly: InternalsVisibleTo("QuickTest")>
```

   The code file should look like this:

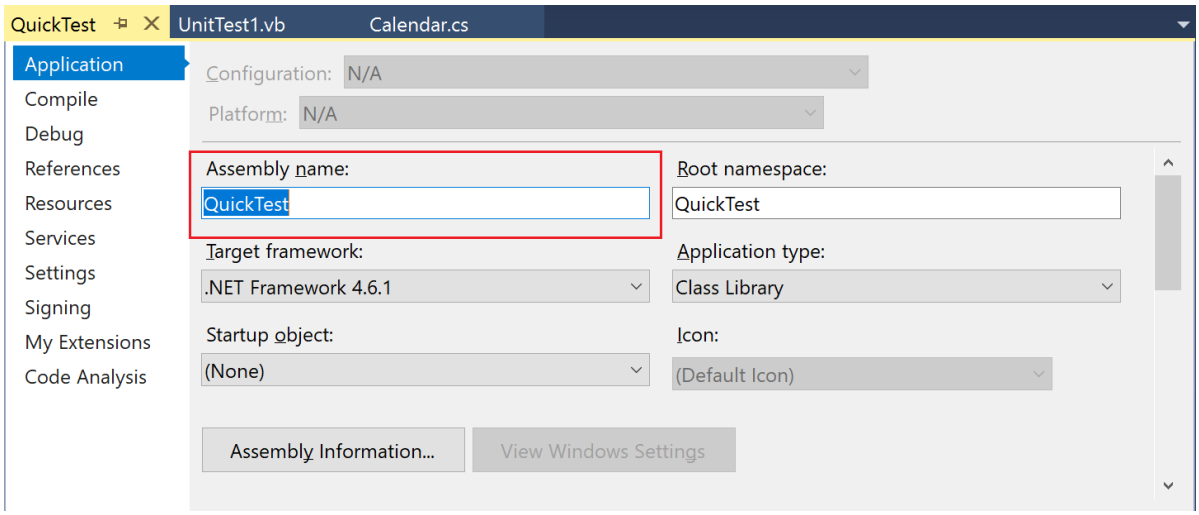


Project properties

The line in the *Calendar.vb* file that contains the InternalsVisibleToAttribute attribute references the assembly name (file name) of the **QuickTest** project. The assembly name might not always be the same as the project name. To find the assembly name of a project, open the project properties.

1. In **Solution Explorer**, select the **QuickTest** project. From the right-click or context menu, select **Properties**, or just press **Alt**+**Enter**. (You can also double-click **My Project** in **Solution Explorer**.)

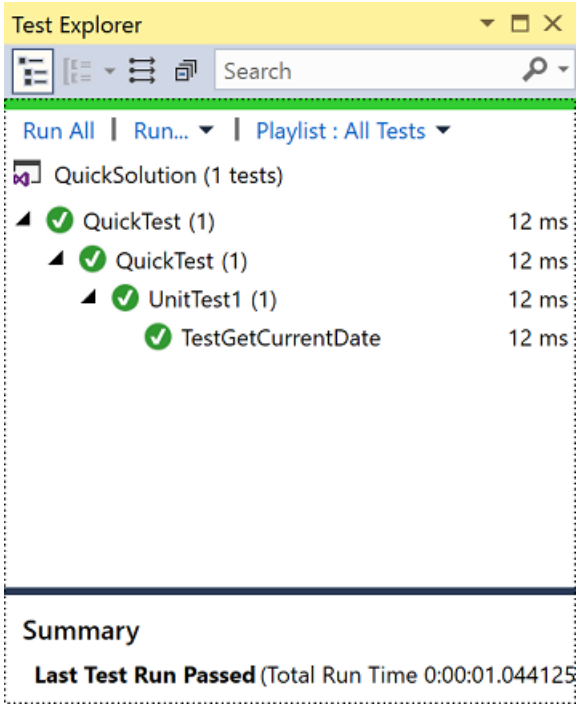
   The *property pages* for the project open on the **Application** tab. The property pages contain various settings for the project. Notice that the assembly name of the **QuickTest** project is indeed "QuickTest". If you wanted to change it, this is where you'd do that. Then, when you build the test project, the name of the resulting binary file would change from *QuickTest.dll* to whatever you chose.



2. Explore some of the other tabs of the project's property pages, such as **Compile** and **Settings**. These tabs are different for different types of projects.

(Optional) Run the test

If you want to check that your unit test is working, choose **Test** > **Run** > **All Tests** from the menu bar. A window called **Test Explorer** opens, and you should see that the **TestGetCurrentDate** test passes.



TIP

If **Test Explorer** doesn't open automatically, open it by choosing **Test** > **Windows** > **Test Explorer** from the menu bar.

Next steps

If you want to further explore Visual Studio, consider creating an app by following one of the Visual Basic tutorials.

See also

- Create projects and solutions
- Manage project and solution properties
- Manage references in a project
- Develop code in Visual Studio without projects or solutions
- Visual Studio IDE overview

Features of Visual Studio

4/16/2019 • 7 minutes to read • Edit Online

The Visual Studio IDE overview article gives a basic introduction to Visual Studio. This article describes features that might be more appropriate for experienced developers, or those developers who are already familiar with Visual Studio.

Modular installation

Visual Studio's modular installer enables you to choose and install *workloads*. Workloads are groups of features needed for the programming language or platform you prefer. This strategy helps to keep the footprint of the Visual Studio installation smaller, which means it installs and updates faster too.

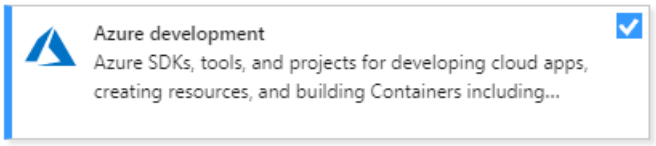
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

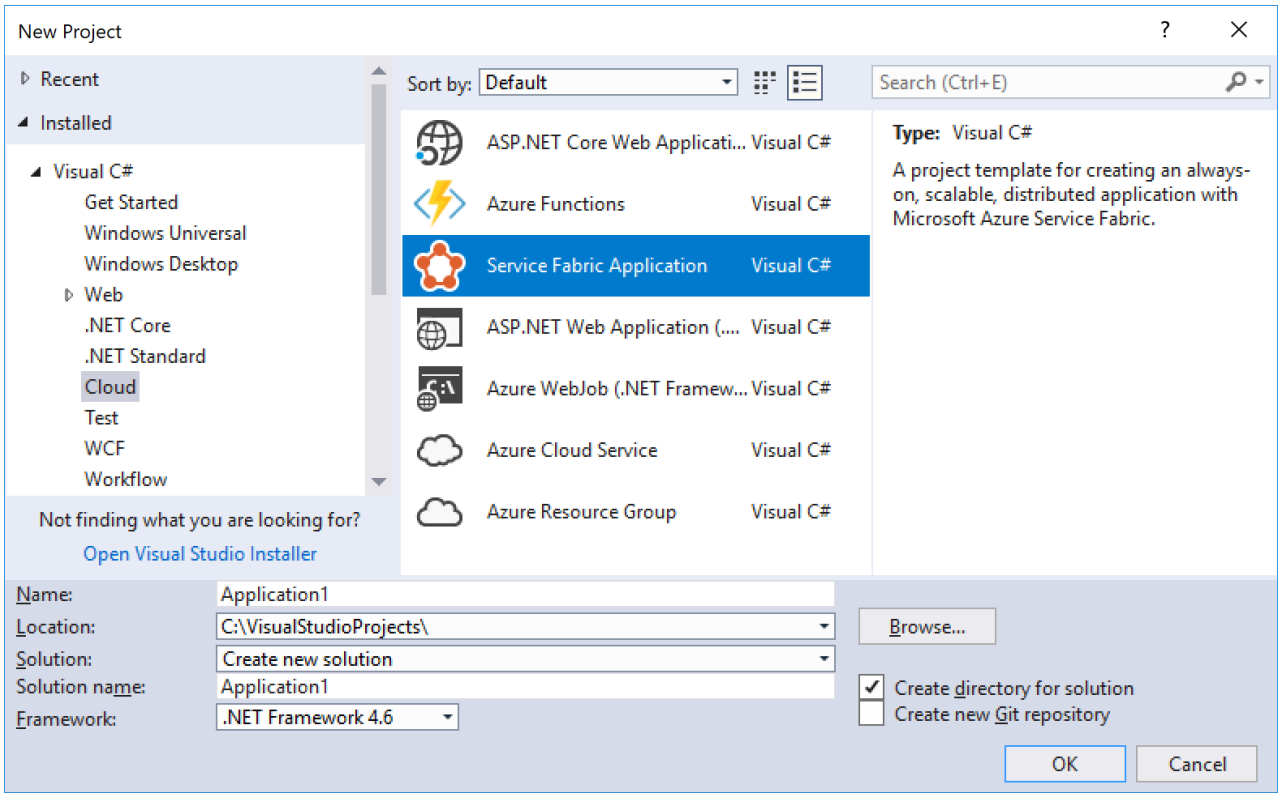
To learn more about setting up Visual Studio on your system, see Install Visual Studio.

Create cloud-enabled apps for Azure

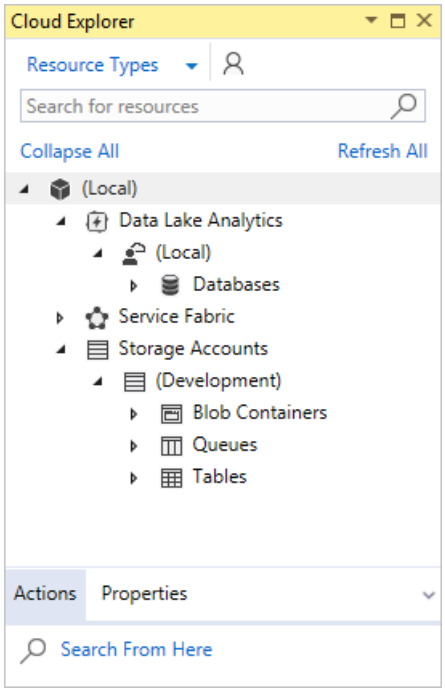
Visual Studio offers a suite of tools that enable you to easily create cloud-enabled applications powered by Microsoft Azure. You can configure, build, debug, package, and deploy applications and services on Microsoft Azure directly from the IDE. To get the Azure tools and project templates, select the **Azure development** workload when you install Visual Studio.



After you install the **Azure development** workload, the following **Cloud** templates for C# are available in the **New Project** dialog:



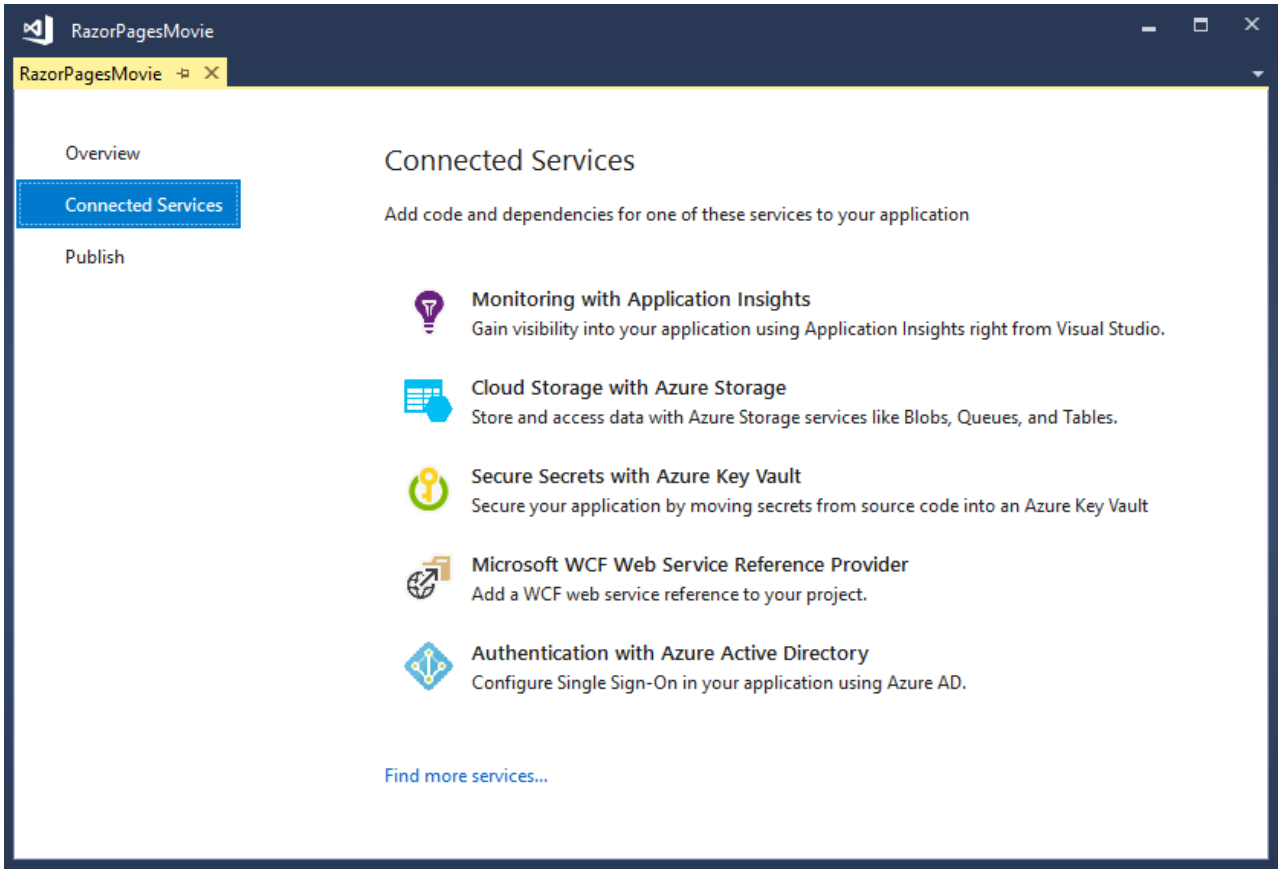
Visual Studio's Cloud Explorer lets you view and manage your Azure-based cloud resources within Visual Studio. These resources may include virtual machines, tables, SQL databases, and more. **Cloud Explorer** shows the Azure resources in all the accounts managed under the Azure subscription you're logged into. And if a particular operation requires the Azure portal, **Cloud Explorer** provides links that take you to the place in the portal where you need to go.



You can leverage Azure services for your apps using **Connected Services** such as:

- Active Directory connected service so users can use their accounts from Azure Active Directory to connect to web apps
- Azure Storage connected service for blob storage, queues, and tables
- Key Vault connected service to manage secrets for web apps

The available **Connected Services** depend on your project type. Add a service by right-clicking on the project in **Solution Explorer** and choosing **Add** > **Connected Service**.



For more information, see Move to the cloud With Visual Studio and Azure.

Create apps for the web

The web drives our modern world, and Visual Studio can help you write apps for it. You can create web apps using ASP.NET, Node.js, Python, JavaScript, and TypeScript. Visual Studio understands web frameworks like Angular, jQuery, Express, and more. ASP.NET Core and .NET Core run on Windows, Mac, and Linux operating systems. ASP.NET Core is a major update to MVC, WebAPI and SignalR, and runs on Windows, Mac, and Linux. ASP.NET Core has been designed from the ground up to provide you with a lean and composable .NET stack for building modern cloud-based web apps and services.

For more information, see Modern web tooling.

Build cross-platform apps and games

You can use Visual Studio to build apps and games for macOS, Linux, and Windows, as well as for Android, iOS, and other mobile devices.

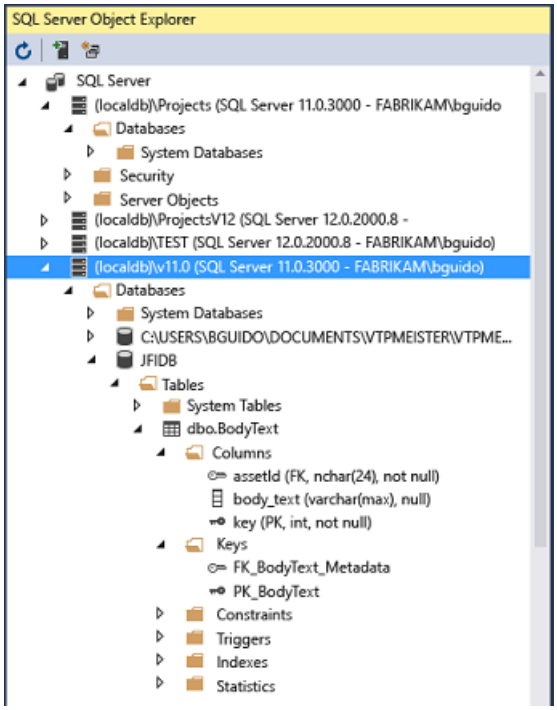
- Build .NET Core apps that run on Windows, macOS, and Linux.
- Build mobile apps for iOS, Android, and Windows in C# and F# by using Xamarin.
- Use standard web technologies—HTML, CSS, and JavaScript—to build mobile apps for iOS, Android, and Windows by using Apache Cordova.
- Build 2D and 3D games in C# by using Visual Studio Tools for Unity.
- Build native C++ apps for iOS, Android, and Windows devices. Share common code in libraries built for iOS, Android, and Windows, by using C++ for cross-platform development.
- Deploy, test, and debug Android apps with the Android emulator.

Connect to databases

Server Explorer helps you browse and manage SQL Server instances and assets locally, remotely, and on Azure, Salesforce.com, Office 365, and websites. To open **Server Explorer**, on the main menu, choose **View** > **Server Explorer**. For more information on using Server Explorer, see Add new connections.

SQL Server Data Tools (SSDT) is a powerful development environment for SQL Server, Azure SQL Database, and Azure SQL Data Warehouse. It enables you to build, debug, maintain, and refactor databases. You can work with a database project, or directly with a connected database instance on- or off-premises.

SQL Server Object Explorer in Visual Studio provides a view of your database objects similar to SQL Server Management Studio. SQL Server Object Explorer enables you to do light-duty database administration and design work. Work examples include editing table data, comparing schemas, executing queries by using contextual menus right from SQL Server Object Explorer, and more.



Debug, test, and improve your code

When you write code, you need to run it and test it for bugs and performance. Visual Studio's cutting-edge debugging system enables you to debug code running in your local project, on a remote device, or on a device emulator. You can step through code one statement at a time and inspect variables as you go. You can set breakpoints that are only hit when a specified condition is true. Debug options can be managed in the code editor itself, so that you don't have to leave your code. To get more details about debugging in Visual Studio, see First look at the debugger.

To learn more about improving the performance of your apps, checkout out Visual Studio's profiling feature.

For testing, Visual Studio offers unit testing, Live Unit Testing, IntelliTest, load and performance testing, and more. Visual Studio also has advanced code analysis capabilities to catch design, security, and other types of flaws.

Deploy your finished application

When your application is ready to deploy to users or customers, Visual Studio provides the tools to do that. Deployment options include to Microsoft Store, to a SharePoint site, or with InstallShield or Windows Installer technologies. It's all accessible through the IDE. For more information, see Deploy applications, services, and components.

Manage your source code and collaborate with others

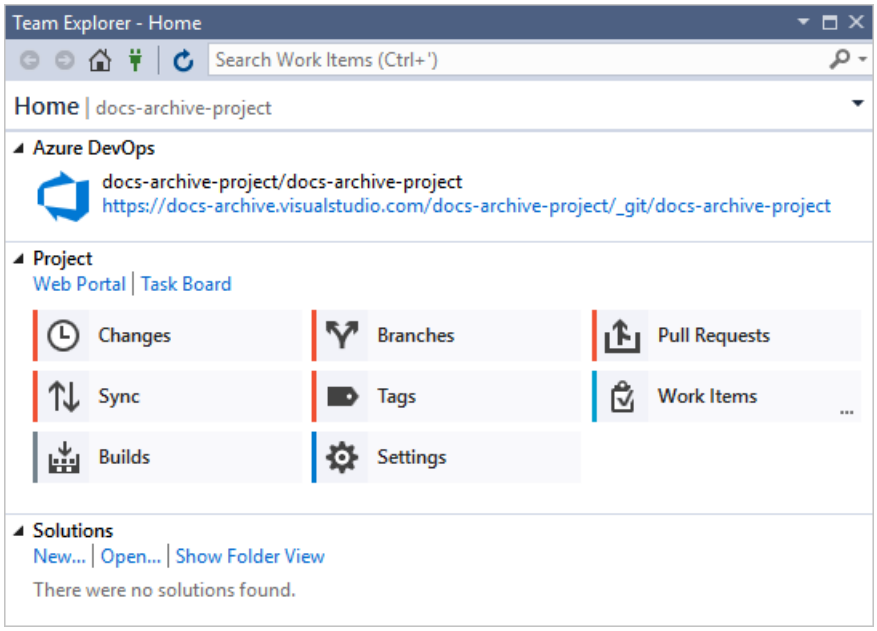
You can manage your source code in Git repos hosted by any provider, including GitHub. Or use Azure DevOps Services to manage code alongside bugs and work items for your whole project. See Get started with Git and Azure Repos to learn more about managing Git repos in Visual Studio using Team Explorer. Visual Studio also has other built-in source control features. To learn more about them, see New Git features in Visual Studio (blog).

Azure DevOps Services are cloud-based services to plan, host, automate, and deploy software and enable collaboration in teams. Azure DevOps Services support both Git repos (distributed version control) and Team Foundation Version Control (centralized version control). They support pipelines for continuous build and release (CI/CD) of code stored in version control systems. Azure DevOps Services also support Scrum, CMMI and Agile development methodologies.

Team Foundation Server (TFS) is the application lifecycle management hub for Visual Studio. It enables everyone involved with the development process to participate using a single solution. TFS is useful for managing heterogeneous teams and projects, too.

If you have an Azure DevOps organization or a Team Foundation Server on your network, you connect to it through the **Team Explorer** window in Visual Studio. From this window you can check code into or out of source control, manage work items, start builds, and access team rooms and workspaces. You can open **Team Explorer** from the search box, or on the main menu from **View** > **Team Explorer** or from **Team** > **Manage Connections**.

The following image shows the **Team Explorer** window for a solution that is hosted in Azure DevOps Services.



You can also automate your build process to build the code that the devs on your team have checked into version control. For example, you can build one or more projects nightly or every time that code is checked in. For more information, see Azure Pipelines.

Extend Visual Studio

If Visual Studio doesn't have the exact functionality you need, you can add it! You can personalize the IDE based on your workflow and style, add support for external tools not yet integrated with Visual Studio, and modify existing functionality to increase your productivity. To find the latest version of the Visual Studio Extensibility Tools (VS SDK), see Visual Studio SDK.

You can use the .NET Compiler Platform ("Roslyn") to write your own code analyzers and code generators. Find everything you need at Roslyn.

Find existing extensions for Visual Studio created by Microsoft developers as well as our development community.

To learn more about extending Visual Studio, see Extend Visual Studio IDE.

See also

- Visual Studio IDE overview
- What's new in Visual Studio 2017
- What's new in Visual Studio 2019

Tutorial: Get started with Visual Basic in Visual Studio

4/17/2019 • 6 minutes to read • Edit Online

In this tutorial for Visual Basic (VB), you'll use Visual Studio to create and run a few different console apps and explore some features of the Visual Studio integrated development environment (IDE) while you do so.

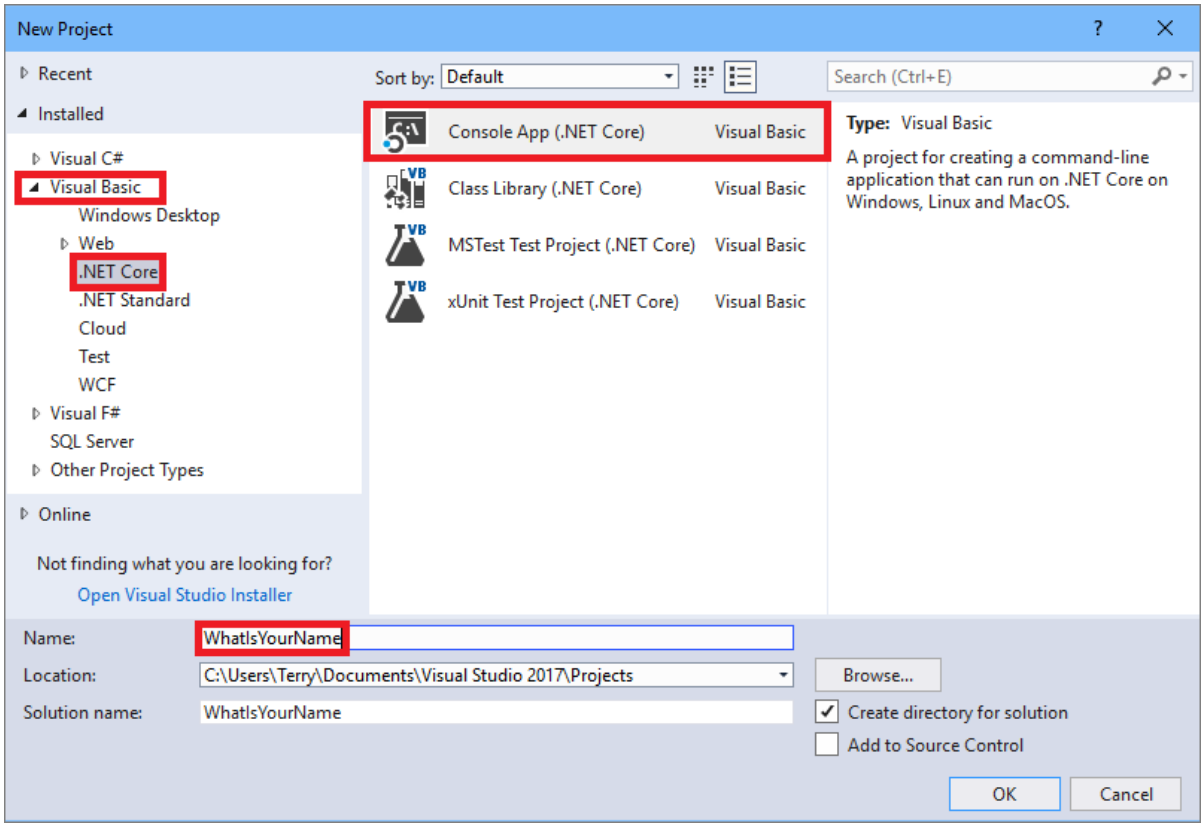
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

Create a project

First, we'll create a Visual Basic application project. The project type comes with all the template files you'll need, before you've even added anything!

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **New** > **Project**.
3. In the **New Project** dialog box in the left pane, expand **Visual Basic**, and then choose **.NET Core**. In the middle pane, choose **Console App (.NET Core)**. Then name the file *HelloWorld*.

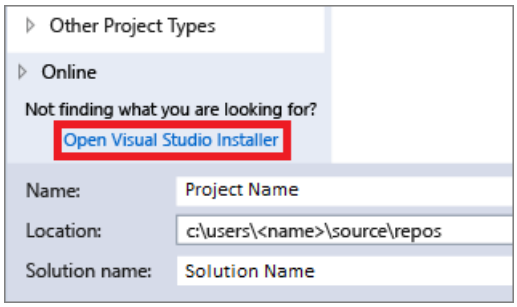


Add a workload (optional)

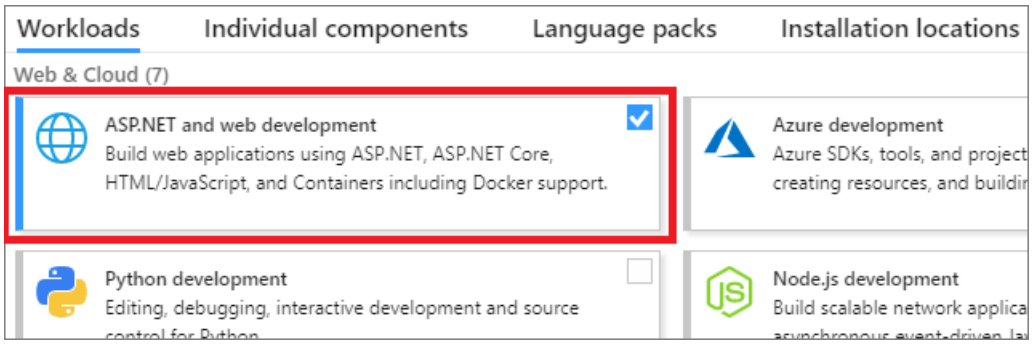
If you don't see the **Console App (.NET Core)** project template, you can get it by adding the **.NET Core cross-platform development** workload. You can add this workload in one of the two following ways, depending on which Visual Studio 2017 updates are installed on your machine.

Option 1: Use the New Project dialog box

1. Click the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box.



2. The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, and then choose **Modify**.



Option 2: Use the Tools menu bar

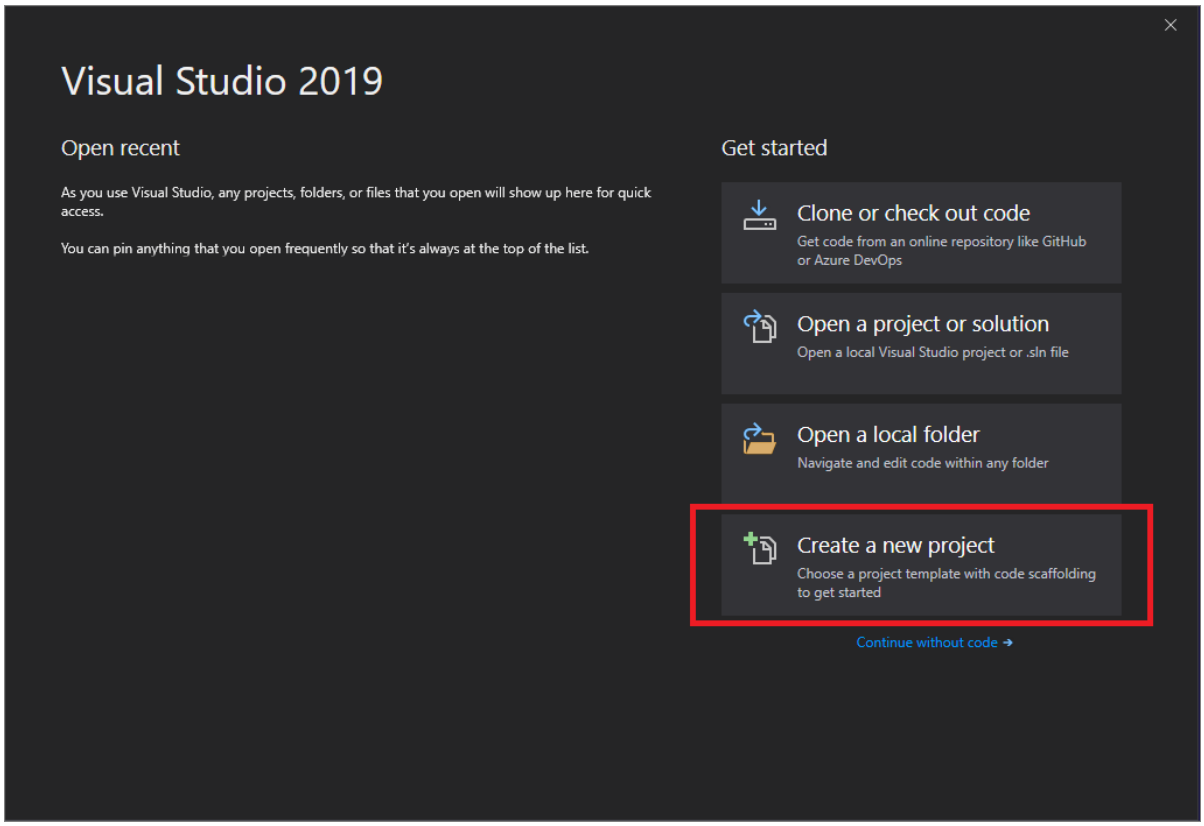
1. Cancel out of the **New Project** dialog box and from the top menu bar, choose **Tools** > **Get Tools and Features**.

2. The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, and then choose **Modify**.

> **NOTE**
> Some of the screenshots in this tutorial use the dark theme. If you aren't using the dark theme but would like to, see the Personalize the Visual Studio IDE and Editor page to learn how.

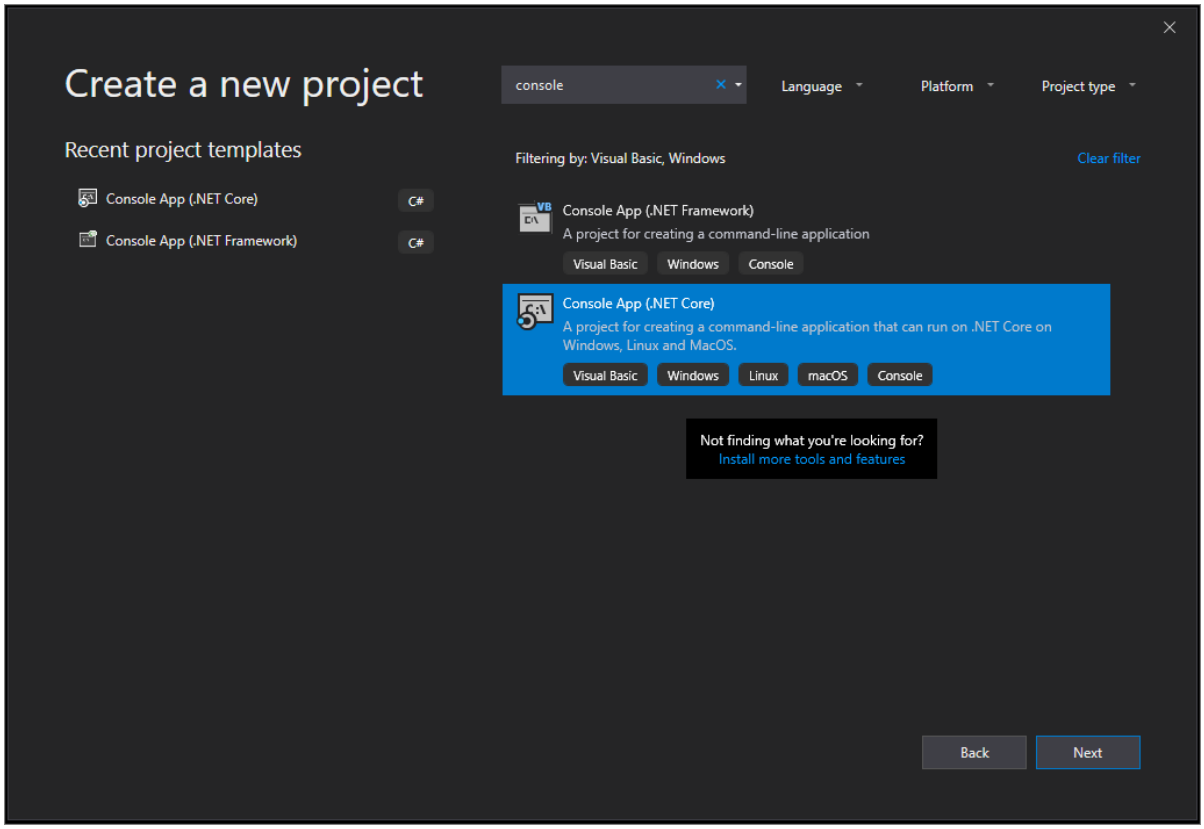
1. Open Visual Studio 2019.

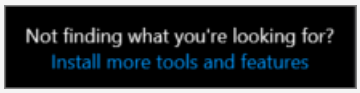
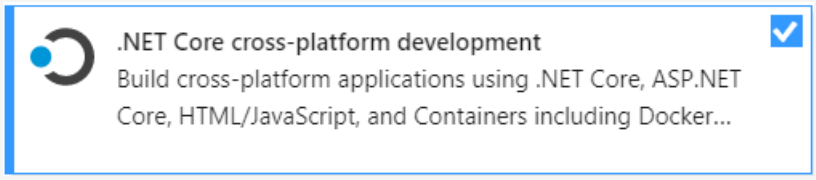
2. On the start window, choose **Create a new project**.



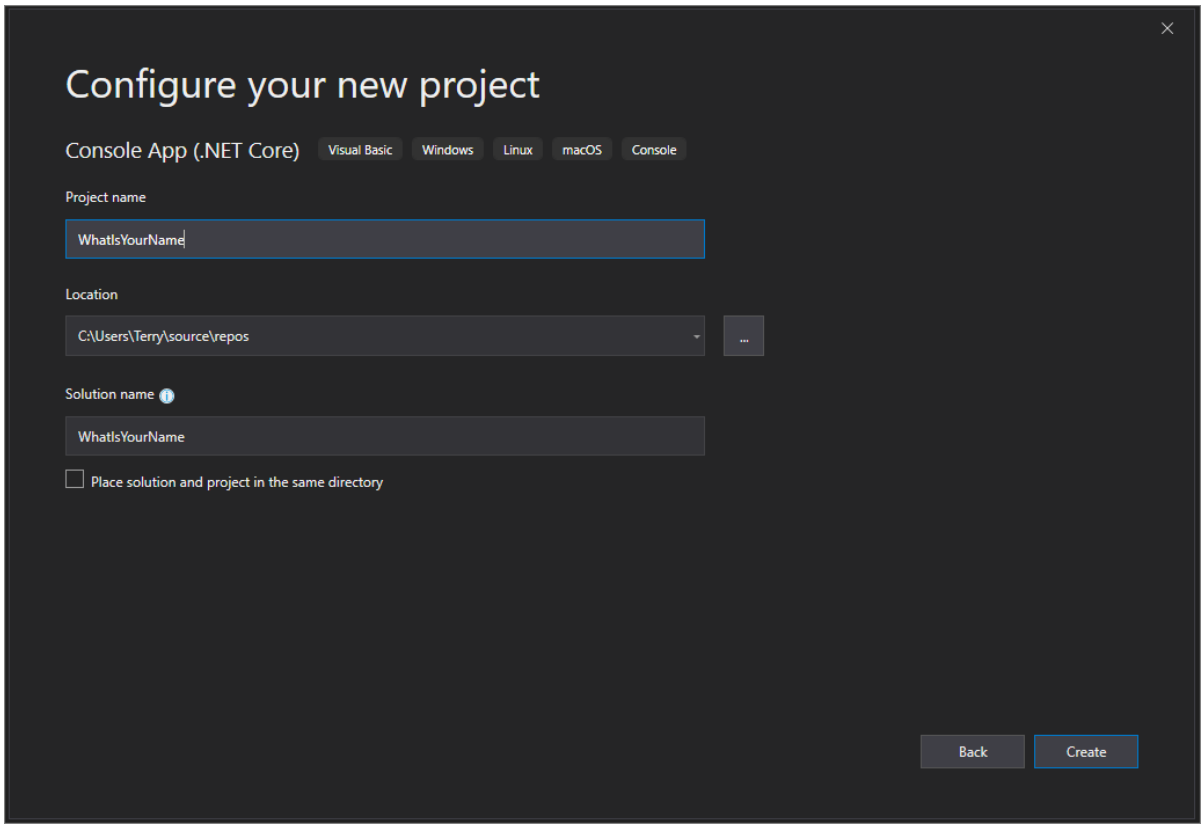
3. On the **Create a new project** window, enter or type *console* in the search box. Next, choose **Visual Basic** from the Language list, and then choose **Windows** from the Platform list.

   After you apply the language and platform filters, choose the **Console App (.NET Core)** template, and then choose **Next**.



> **NOTE**
>
> If you do not see the **Console App (.NET Core)** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.
>
> 
>
>
> Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.
>
> 
>
>
> After that, choose the **Modify** button in the Visual Studio Installer. You might be prompted to save your work; if so, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

4. In the **Configure your new project** window, type or enter *WhatIsYourName* in the **Project name** box. Then, choose **Create**.



Visual Studio opens your new project.

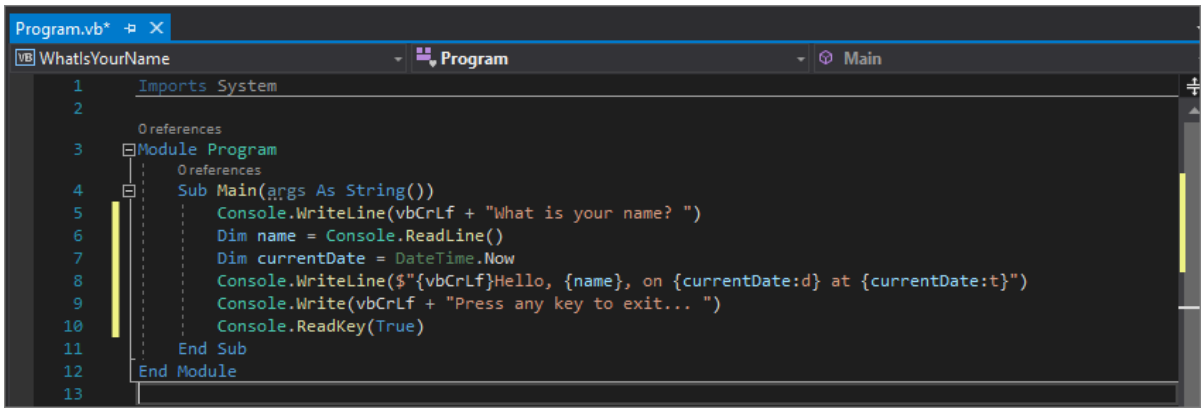
Create a "What Is Your Name" application

Let's create an app that prompts you for your name and then displays it along with the date and time. Here's how:

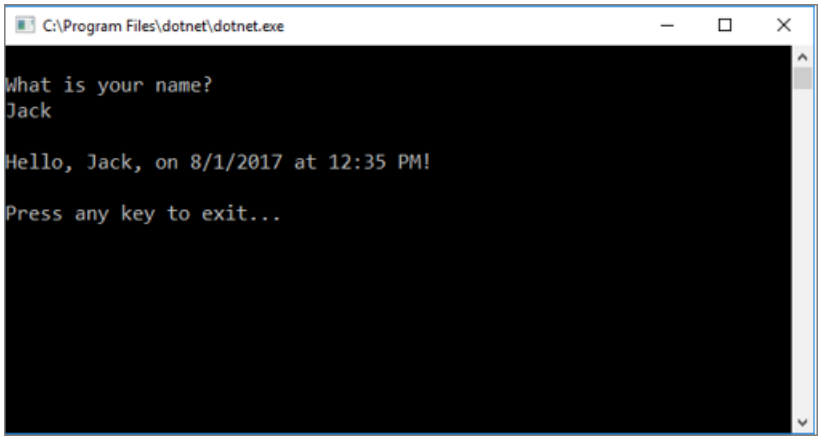
1. If it is not already open, then open your *WhatIsYourName* project.
2. Enter the following Visual Basic code immediately after the opening bracket that follows the `Sub Main(args As String())` line and before the `End Sub` line:

```
Console.WriteLine(vbCrLf + "What is your name? ")
Dim name = Console.ReadLine()
Dim currentDate = DateTime.Now
Console.WriteLine($"{vbCrLf}Hello, {name}, on {currentDate:d} at {currentDate:t}")
Console.Write(vbCrLf + "Press any key to exit... ")
Console.ReadKey(True)
```

This code replaces the existing WriteLine, Write, and ReadKey statements.



3. When the console window opens, enter your name. Your console window should look similar to the following screenshot:

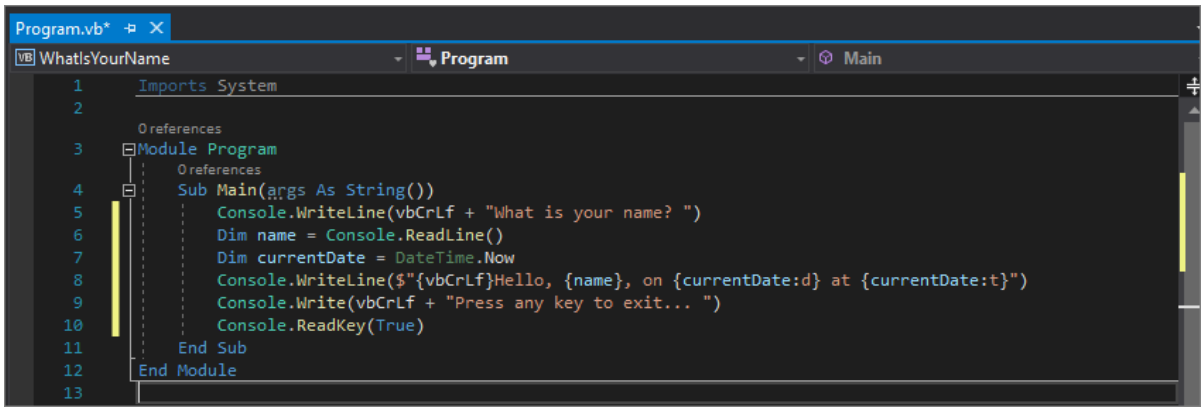


4. Press any key to close the console window.

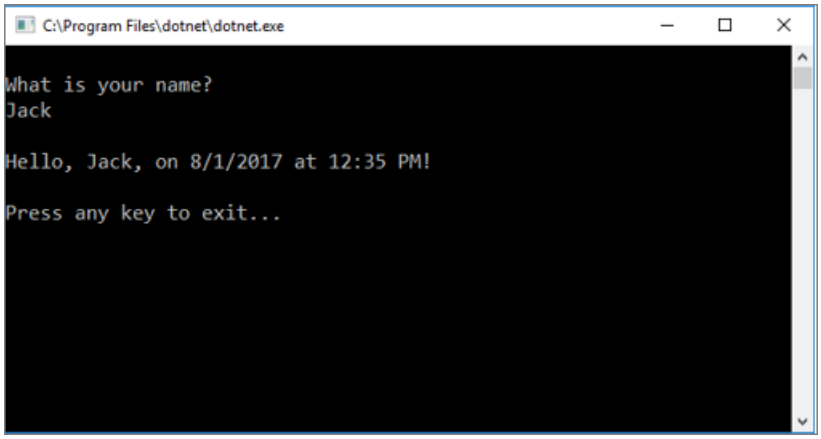
1. In the *WhatIsYourName* project, enter the following Visual Basic code immediately after the opening bracket that follows the `Sub Main(args As String())` line and before the `End Sub` line:

```
Console.WriteLine(vbCrLf + "What is your name? ")
Dim name = Console.ReadLine()
Dim currentDate = DateTime.Now
Console.WriteLine($"{vbCrLf}Hello, {name}, on {currentDate:d} at {currentDate:t}")
Console.Write(vbCrLf + "Press any key to exit... ")
Console.ReadKey(True)
```

This code replaces the existing WriteLine, Write, and ReadKey statements.



2. When the console window opens, enter your name. Your console window should look similar to the following screenshot:



3. Press any key to close the console window.

Create a "Calculate This" application

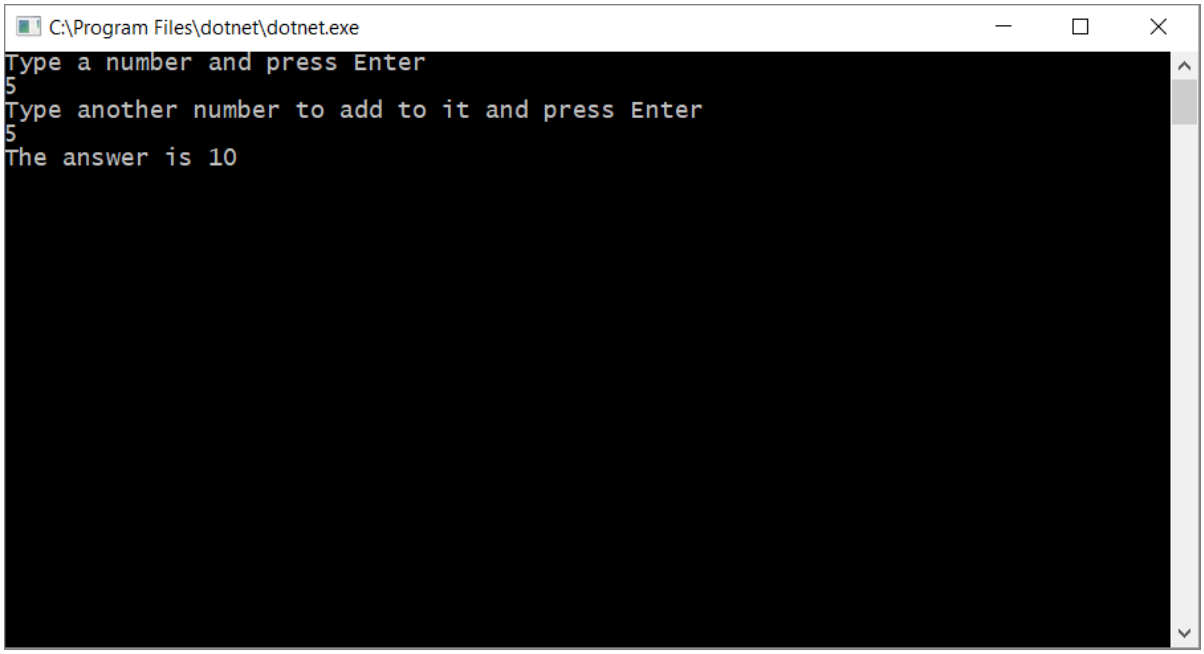
1. Open Visual Studio 2017, and then from the top menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog box in the left pane, expand **Visual Basic**, and then choose **.NET Core**. In the middle pane, choose **Console App (.NET Core)**. Then name the file *CalculateThis*.
3. Enter the following code between the `Module Program` line and `End Module` line:

```
Public num1 As Integer
Public num2 As Integer
Public answer As Integer
Sub Main()
    Console.WriteLine("Type a number and press Enter")
    num1 = Console.ReadLine()
    Console.WriteLine("Type another number to add to it and press Enter")
    num2 = Console.ReadLine()
    answer = num1 + num2
    Console.WriteLine("The answer is " & answer)
    Console.ReadLine()
End Sub
```

Your code window should look like the following screenshot:

```
Program.vb*
CalculateThis | Program | num2
1   Imports System
2
3   Module Program
4       Public num1 As Integer
5       Public num2 As Integer
6       Public answer As Integer
7       Sub Main()
8           Console.WriteLine("Type a number and press Enter")
9           num1 = Console.ReadLine()
10          Console.WriteLine("Type another number to add to it and press Enter")
11          num2 = Console.ReadLine()
12          answer = num1 + num2
13          Console.WriteLine("The answer is " & answer)
14          Console.ReadLine()
15      End Sub
16  End Module
17
```

4. Click **CalculateThis** to run your program. Your console window should look similar to the following screenshot:

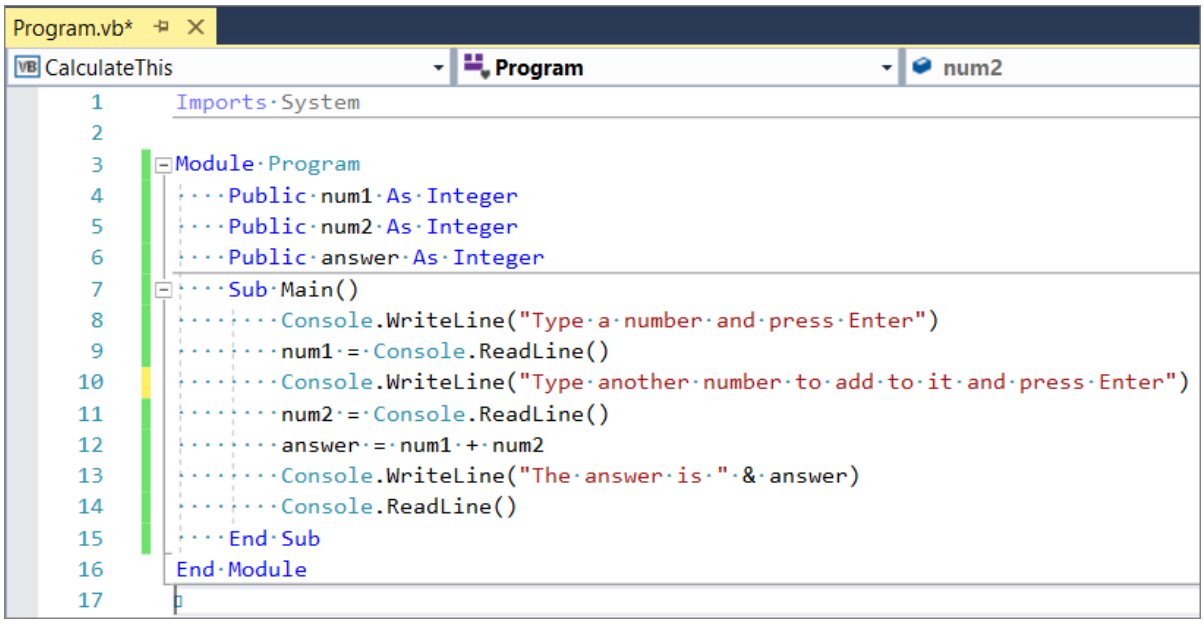


1. On the start window, choose **Create a new project**.
2. On the **Create a new project** window, enter or type *console* in the search box. Next, choose **Visual Basic** from the Language list, and then choose **Windows** from the Platform list.
3. After you apply the language and platform filters, choose the **Console App (.NET Core)** template, and then choose **Next**.

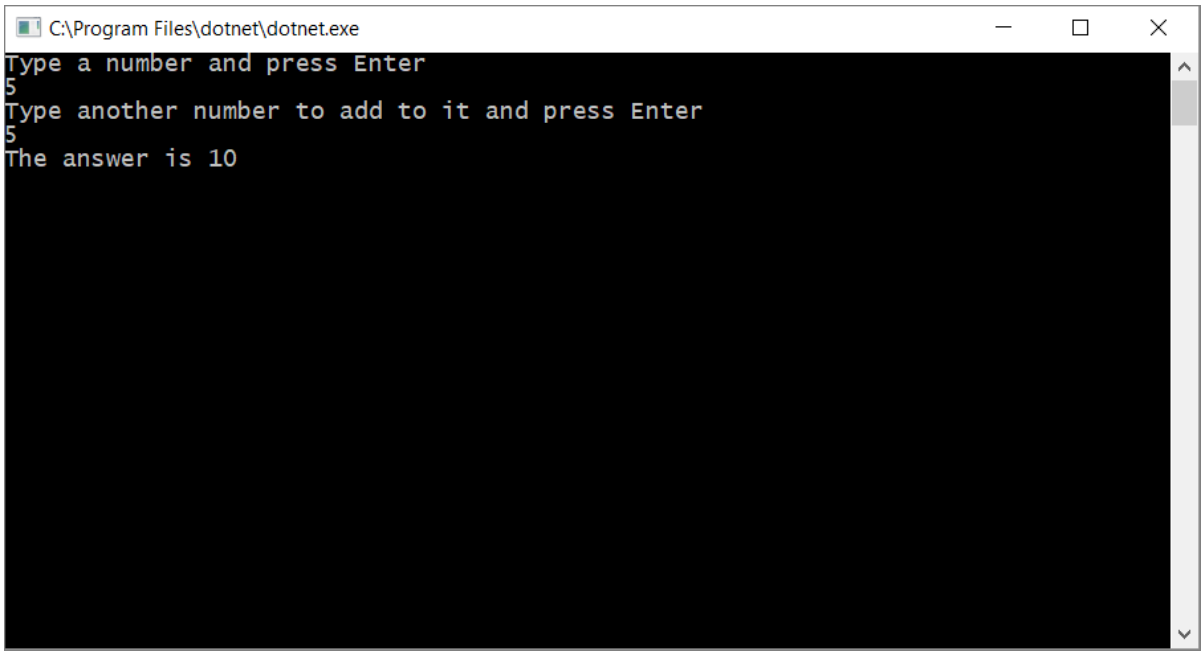
   Then, in the **Configure your new project** window, type or enter *WhatIsYourName* in the **Project name** box. Next, choose **Create**.
4. Enter the following code between the `Module Program` line and `End Module` line:

```
Public num1 As Integer
Public num2 As Integer
Public answer As Integer
Sub Main()
    Console.WriteLine("Type a number and press Enter")
    num1 = Console.ReadLine()
    Console.WriteLine("Type another number to add to it and press Enter")
    num2 = Console.ReadLine()
    answer = num1 + num2
    Console.WriteLine("The answer is " & answer)
    Console.ReadLine()
End Sub
```

Your code window should look like the following screenshot:



5. Click **CalculateThis** to run your program. Your console window should look similar to the following screenshot:



Quick answers FAQ

Here's a quick FAQ to highlight some key concepts.

What is Visual Basic?

Visual Basic is a type-safe programming language that's designed to be easy to learn. It is derived from BASIC, which means "Beginner's All-purpose Symbolic Instruction Code".

What is Visual Studio?

Visual Studio is an integrated development suite of productivity tools for developers. Think of it as a program you can use to create programs and applications.

What is a console app?

A console app takes input and displays output in a command-line window, a.k.a. a console.

What is .NET Core?

.NET Core is the evolutionary next step of the .NET Framework. Where the .NET Framework allowed you to share code across programming languages, .NET Core adds the ability to share code across platforms. Even better, it's open source. (Both the .NET Framework and .NET Core include libraries of prebuilt functionality as well as a common language runtime (CLR), which acts as a virtual machine in which to run your code.)

Next steps

Congratulations on completing this tutorial! To learn even more, see the following tutorial.

Build a library with Visual Basic and the .NET Core SDK in Visual Studio

See also

- Visual Basic language walkthroughs
- Visual Basic language reference
- IntelliSense for Visual Basic code files

Tutorial: Create a simple application with Visual Basic

6/13/2019 • 9 minutes to read • Edit Online

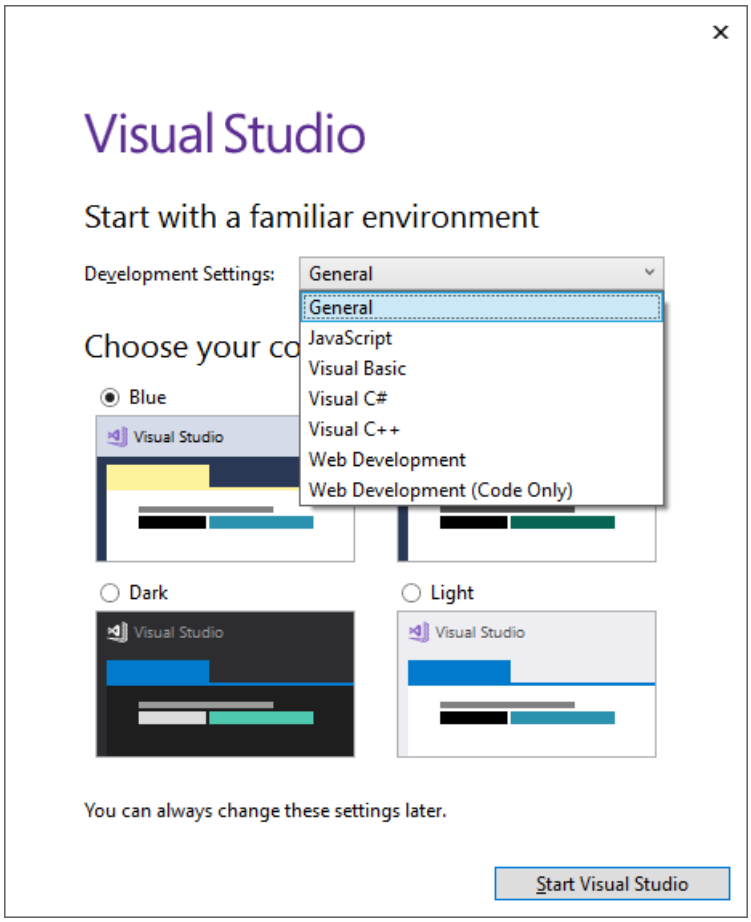
By completing this tutorial, you'll become familiar with many of the tools, dialog boxes, and designers that you can use when you develop applications with Visual Studio. You'll create a "Hello, World" application, design the UI, add code, and debug errors, while you learn about working in the integrated development environment (IDE).

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

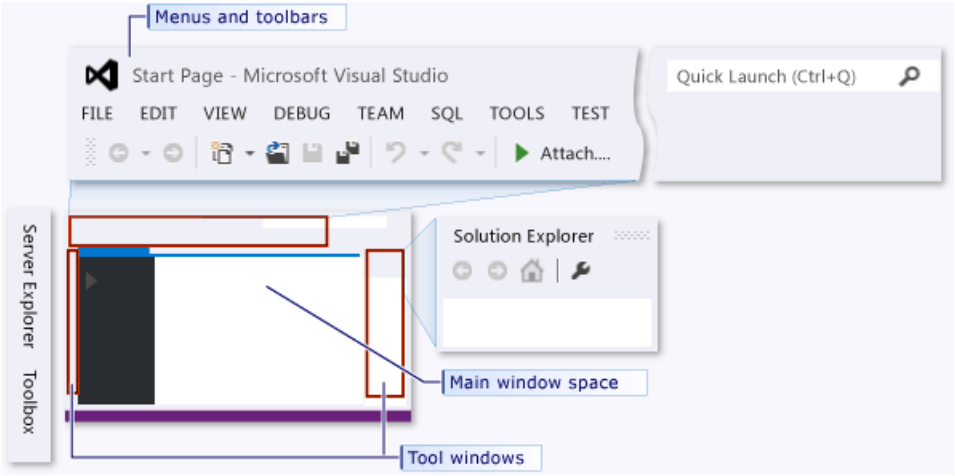
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

Configure the IDE

When you open Visual Studio for the first time, you'll be prompted to sign in. This step is optional for this tutorial. Next you may be shown a dialog box that asks you to choose your development settings and color theme. Keep the defaults and choose **Start Visual Studio**.



After Visual Studio launches, you'll see tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window, with **Quick Launch**, the menu bar, and the standard toolbar at the top. In the center of the application window is the **Start Page**. When you load a solution or project, editors and designers appear in the space where the **Start Page** is. When you develop an application, you'll spend most of your time in this central area.

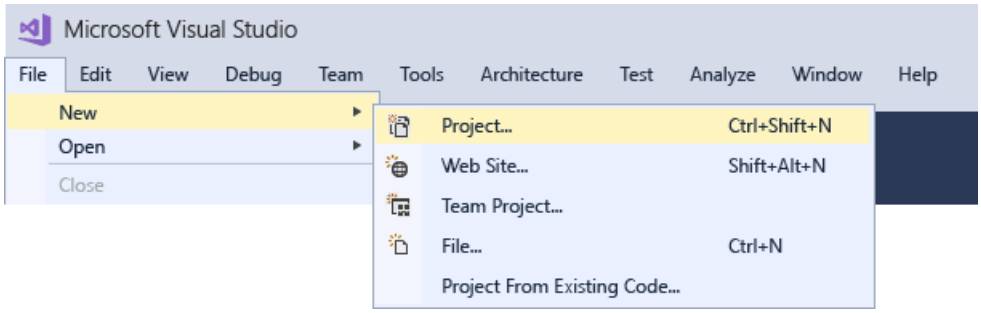


When you launch Visual Studio, the start window opens first. Select **Continue without code** to open the development environment. You'll see tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window, with a search box, the menu bar, and the standard toolbar at the top. When you load a solution or project, editors and designers appear in the central space of the application window. When you develop an application, you'll spend most of your time in this central area.

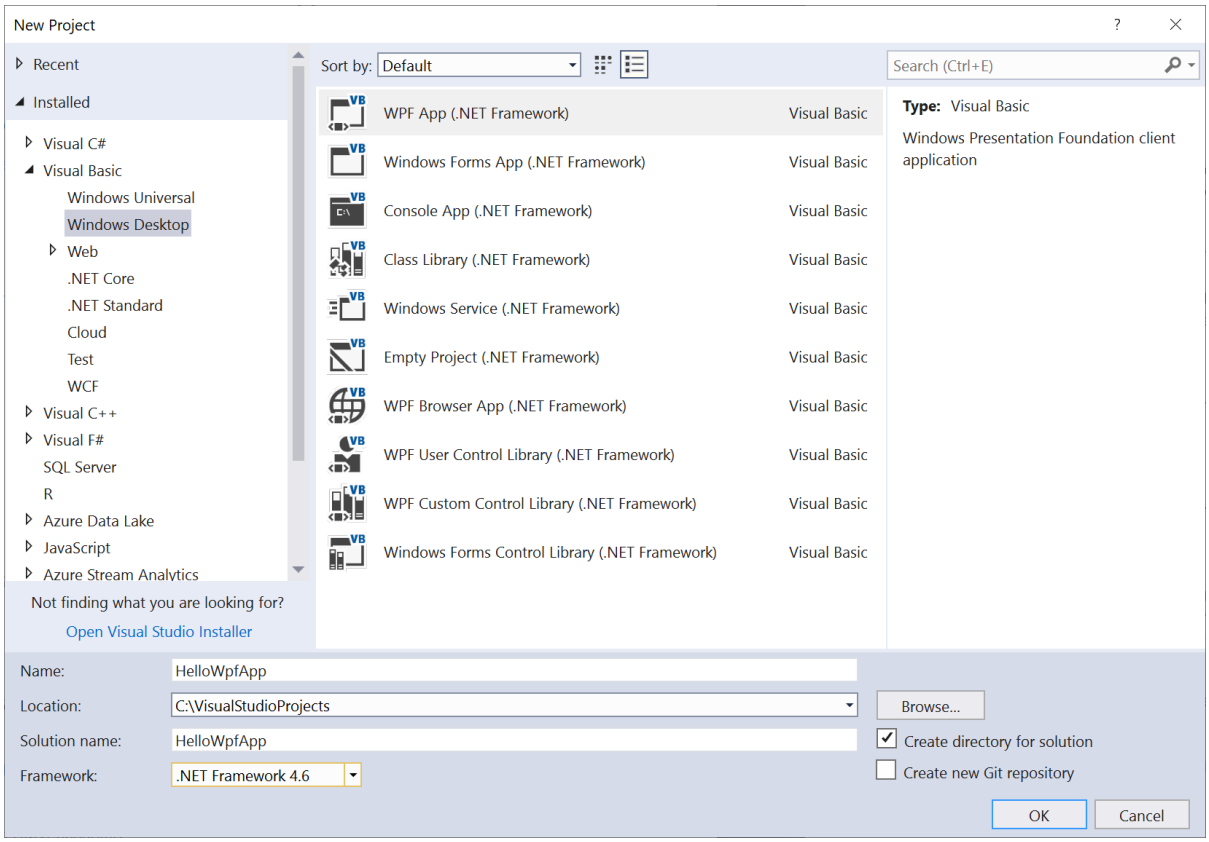
Create the project

When you create an application in Visual Studio, you first create a project and a solution. For this example, you'll create a Windows Presentation Foundation (WPF) project.

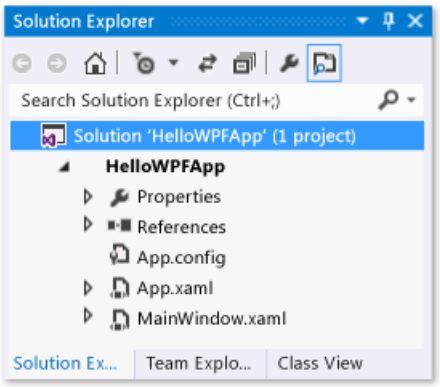
1. Create a new project. On the menu bar, select **File** > **New** > **Project**.



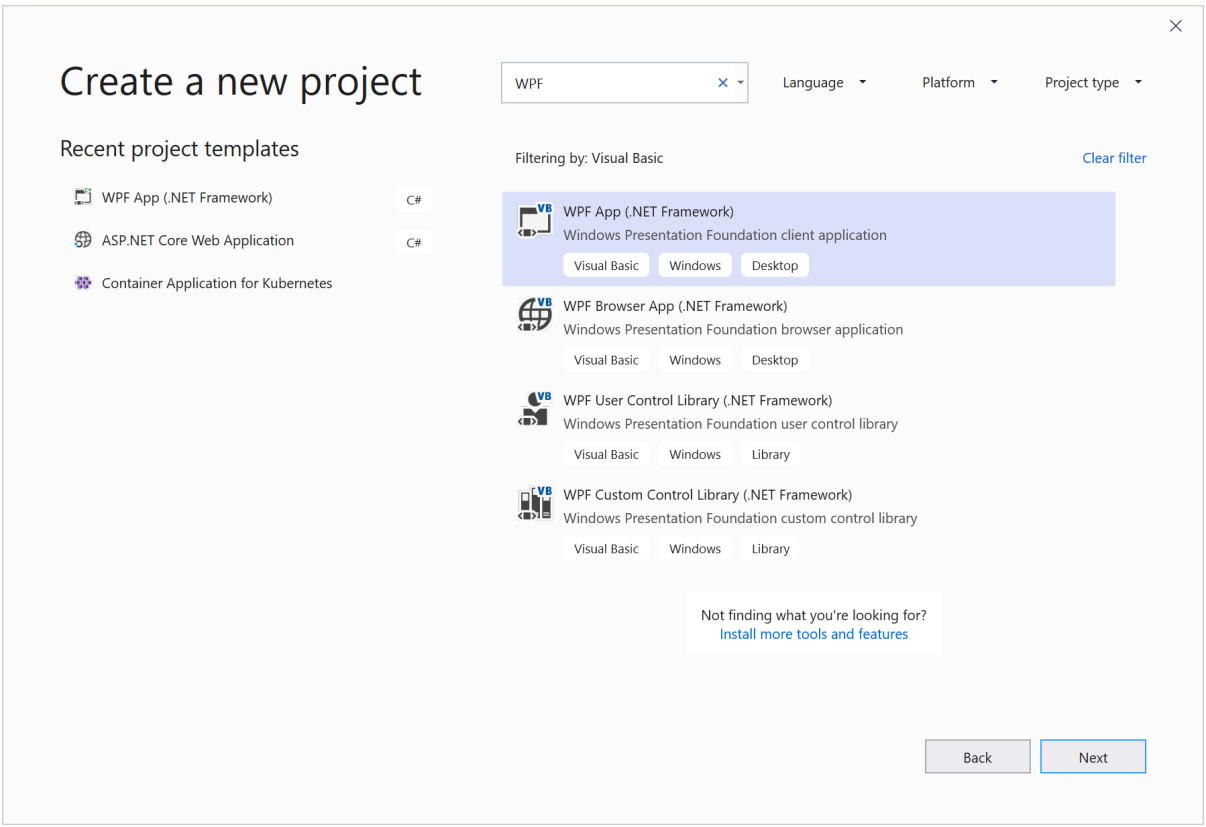
2. In the **New Project** dialog, select the **Installed** > **Visual Basic** > **Windows Desktop** category, and then select the **WPF App (.NET Framework)** template. Name the project **HelloWPFApp**, and select **OK**.



Visual Studio creates the HelloWPFApp project and solution, and **Solution Explorer** shows the various files. The **WPF Designer** shows a design view and a XAML view of *MainWindow.xaml* in a split view. You can slide the splitter to show more or less of either view. You can choose to see only the visual view or only the XAML view. The following items appear in **Solution Explorer**:

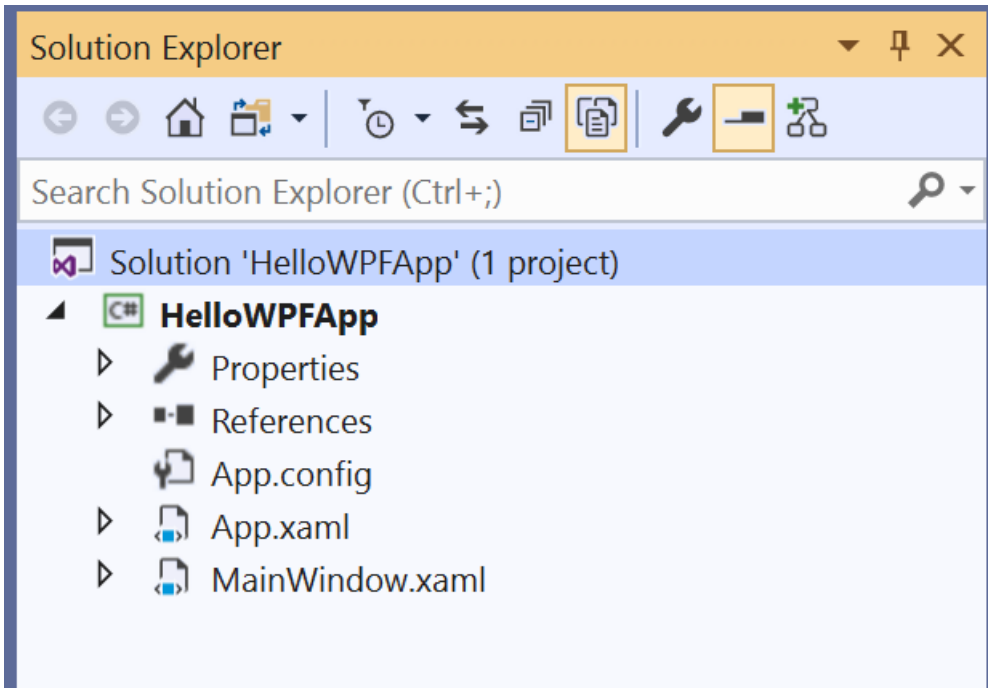


1. Open Visual Studio 2019.
2. On the **Create a new project** screen, search for "WPF", and choose **WPF App (.NET Framework)**, and then choose **Next**.



3. At the next screen, give the project a name, **HelloWPFApp**, and choose **Create**.

Visual Studio creates the HelloWPFApp project and solution, and **Solution Explorer** shows the various files. The **WPF Designer** shows a design view and a XAML view of *MainWindow.xaml* in a split view. You can slide the splitter to show more or less of either view. You can choose to see only the visual view or only the XAML view. The following items appear in **Solution Explorer**:



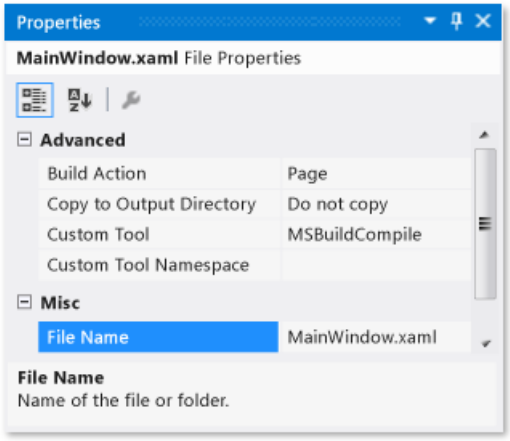
> **NOTE**
>
> For more information about XAML (eXtensible Application Markup Language), see the XAML overview for WPF page.

After you create the project, you can customize it. By using the **Properties** window (found on the **View** menu), you can display and change options for project items, controls, and other items in an application.

Change the name of MainWindow.xaml

Let's give MainWindow a more specific name.

1. In **Solution Explorer**, select *MainWindow.xaml*. You should see the **Properties** window, but if you don't, choose the **View** menu and then the **Properties Window** item.
2. Change the **File Name** property to `Greetings.xaml`.



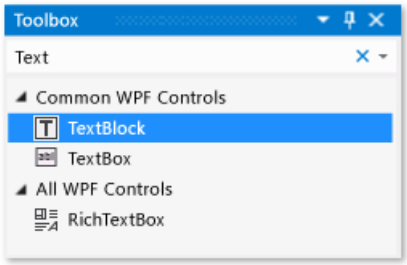
Solution Explorer shows that the name of the file is now *Greetings.xaml*, and the nested code file is now named *Greetings.xaml.vb*. This code file is nested under the *.xaml* file node to show they are closely related to each other.

Design the user interface (UI)

We will add three types of controls to this application: a TextBlock control, two RadioButton controls, and a Button control.

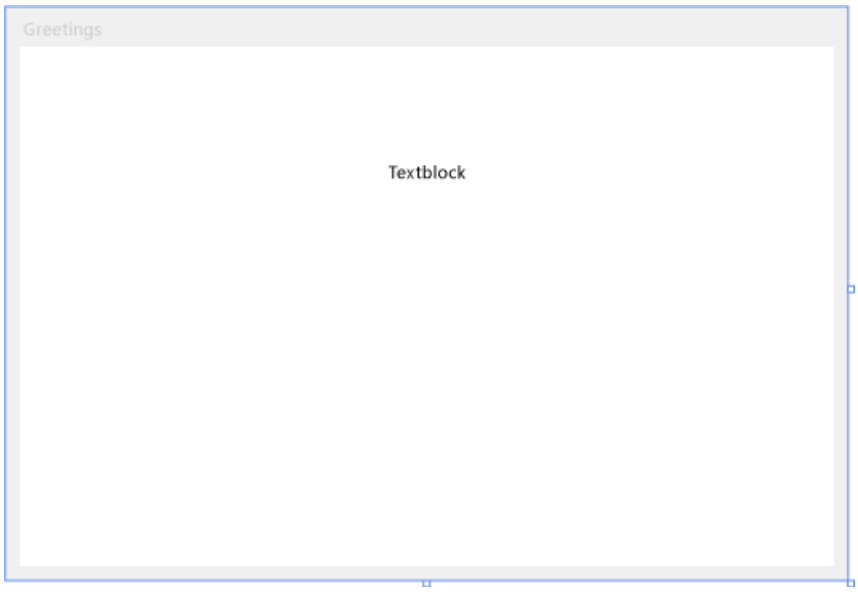
Add a TextBlock control

1. Enter **Ctrl**+**Q** to activate the search box and type **Toolbox**. Choose **View > Toolbox** from the results list.
2. In the **Toolbox**, expand the **Common WPF Controls** node to see the TextBlock control.



3. Add a TextBlock control to the design surface by choosing the **TextBlock** item and dragging it to the window on the design surface. Center the control near the top of the window.

Your window should resemble the following illustration:



The XAML markup should look something like the following example:

```
<TextBlock HorizontalAlignment="Left" Margin="381,100,0,0" TextWrapping="Wrap" Text="TextBlock"
VerticalAlignment="Top"/>
```

Customize the text in the text block

1. In the XAML view, locate the markup for TextBlock and change the Text attribute:

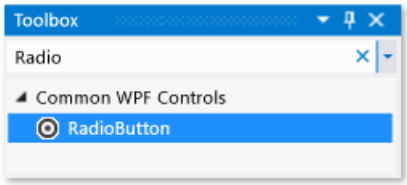
   ```
   Text="Select a message option and then choose the Display button."
   ```

2. Center the TextBlock again if necessary, and save your changes by pressing Ctrl+S or using the **File** menu item.

Next, you'll add two RadioButton controls to the form.

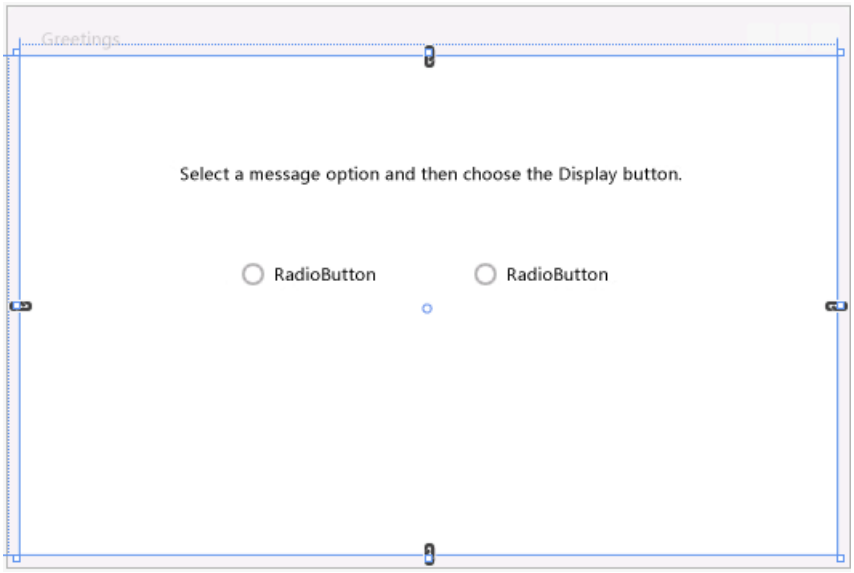
Add radio buttons

1. In the **Toolbox**, find the **RadioButton** control.

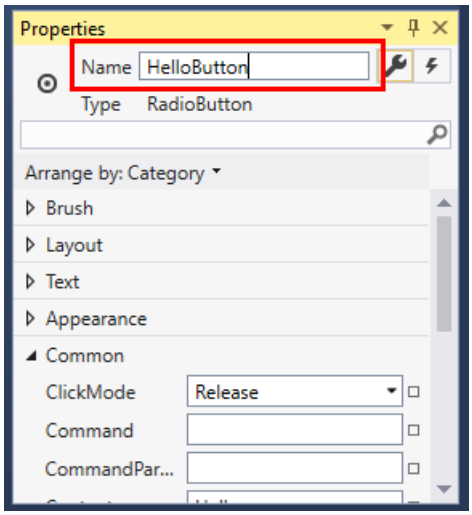
   

2. Add two RadioButton controls to the design surface by choosing the **RadioButton** item and dragging it to the window on the design surface. Move the buttons (by selecting them and using the arrow keys) so that the buttons appear side by side under the TextBlock control.

   Your window should look like this:



3. In the **Properties** window for the left RadioButton control, change the **Name** property (the property at the top of the **Properties** window) to `HelloButton`.



4. In the **Properties** window for the right RadioButton control, change the **Name** property to `GoodbyeButton`, and then save your changes.

You can now add display text for each RadioButton control. The following procedure updates the **Content** property for a RadioButton control.

Add display text for each radio button

1. On the design surface, open the shortcut menu for HelloButton by pressing the right mouse button on HelloButton, choose **Edit Text**, and then enter `Hello`.

2. Open the shortcut menu for GoodbyeButton by pressing the right mouse button on GoodbyeButton, choose **Edit Text**, and then enter `Goodbye`.

Set a radio button to be checked by default

In this step, we'll set HelloButton to be checked by default so that one of the two radio buttons is always selected.

In the XAML view, locate the markup for HelloButton and add an **IsChecked** attribute:

```
IsChecked="True"
```

The final UI element that you'll add is a Button control.

Add the button control

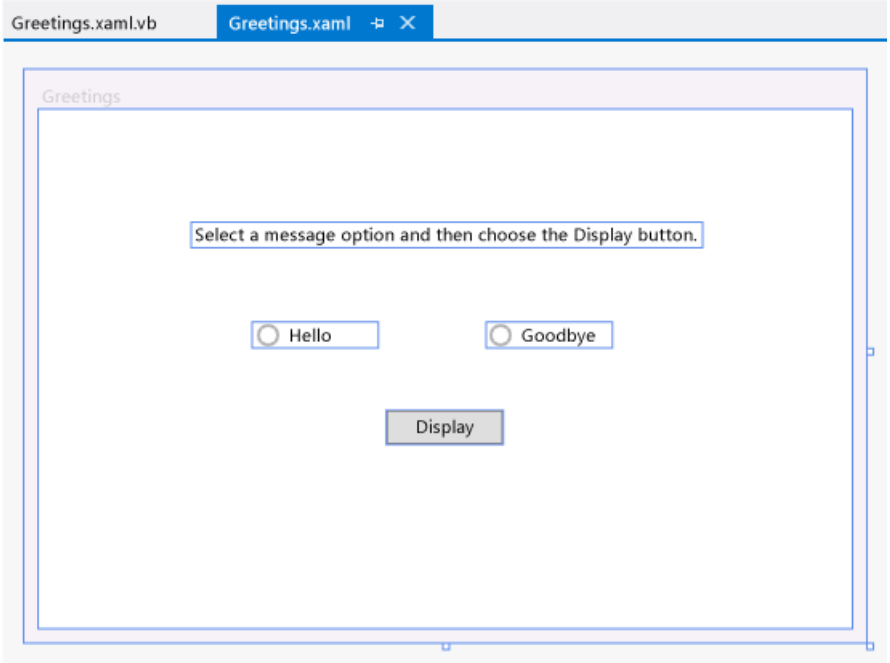
1. In the **Toolbox**, find the **Button** control, and then add it to the design surface under the RadioButton controls by dragging it to the form in the design view.

2. In the XAML view, change the value of **Content** for the Button control from `Content="Button"` to `Content="Display"`, and then save the changes.

   The markup should resemble the following example:

   ```
   <Button Content="Display" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
   Margin="215,204,0,0"/>
   ```

   Your window should resemble the following illustration.



Add code to the display button

When this application runs, a message box appears after a user chooses a radio button and then chooses the **Display** button. One message box will appear for Hello, and another will appear for Goodbye. To create this behavior, you'll add code to the `Button_Click` event in *Greetings.xaml.vb* or *Greetings.xaml.cs*.

1. On the design surface, double-click the **Display** button.

   Greetings.xaml.vb opens, with the cursor in the `Button_Click` event.

   ```
   Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)

   End Sub
   ```

2. Enter the following code:

   ```
   If HelloButton.IsChecked = True Then
       MessageBox.Show("Hello.")
   ElseIf GoodbyeButton.IsChecked = True Then
       MessageBox.Show("Goodbye.")
   End If
   ```

3. Save the application.

Debug and test the application

Next, you'll debug the application to look for errors and test that both message boxes appear correctly. The following instructions tell you how to build and launch the debugger, but later you might read Build a WPF application (WPF) and Debug WPF for more information.

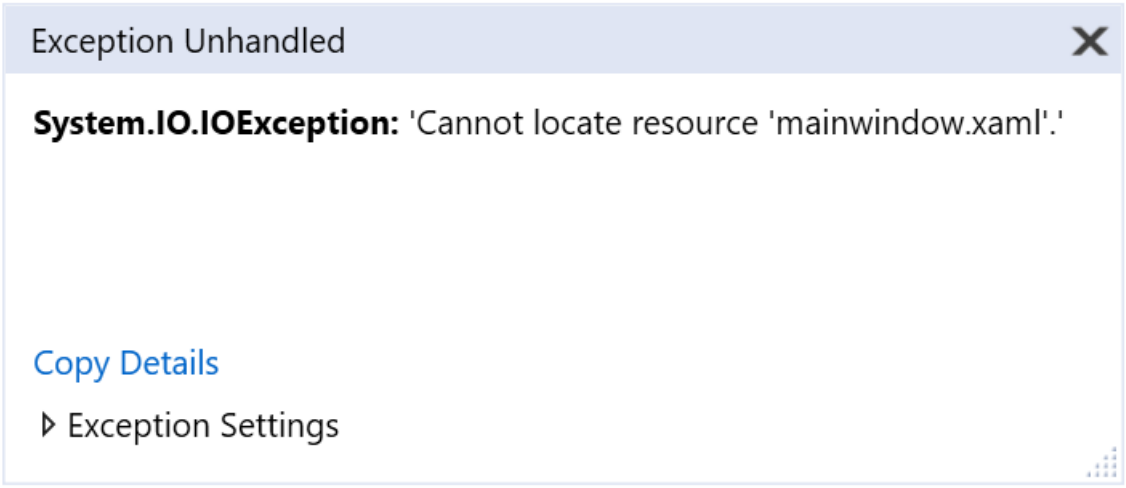
Find and fix errors

In this step, you'll find the error that we caused earlier by changing the name of the *MainWindow.xaml* file.

Start debugging and find the error

1. Start the debugger by pressing **F5** or selecting **Debug**, then **Start Debugging**.

   A **Break Mode** window appears, and the **Output** window indicates that an IOException has occurred: Cannot locate resource 'mainwindow.xaml'.



2. Stop the debugger by choosing **Debug** > **Stop Debugging**.

We renamed *MainWindow.xaml* to *Greetings.xaml* at the start of this tutorial, but the code still refers to *MainWindow.xaml* as the startup URI for the application, so the project can't start.

Specify Greetings.xaml as the startup URI

1. In **Solution Explorer**, open the *Application.xaml* file.
2. Change `StartupUri="MainWindow.xaml"` to `StartupUri="Greetings.xaml"`, and then save the changes.

Start the debugger again (press **F5**). You should see the **Greetings** window of the application. Now close the application window to stop debugging.

Debug with breakpoints

You can test the code during debugging by adding some breakpoints. You can add breakpoints by choosing **Debug** > **Toggle Breakpoint**, by clicking in the left margin of the editor next to the line of code where you want the break to occur, or by pressing **F9**.

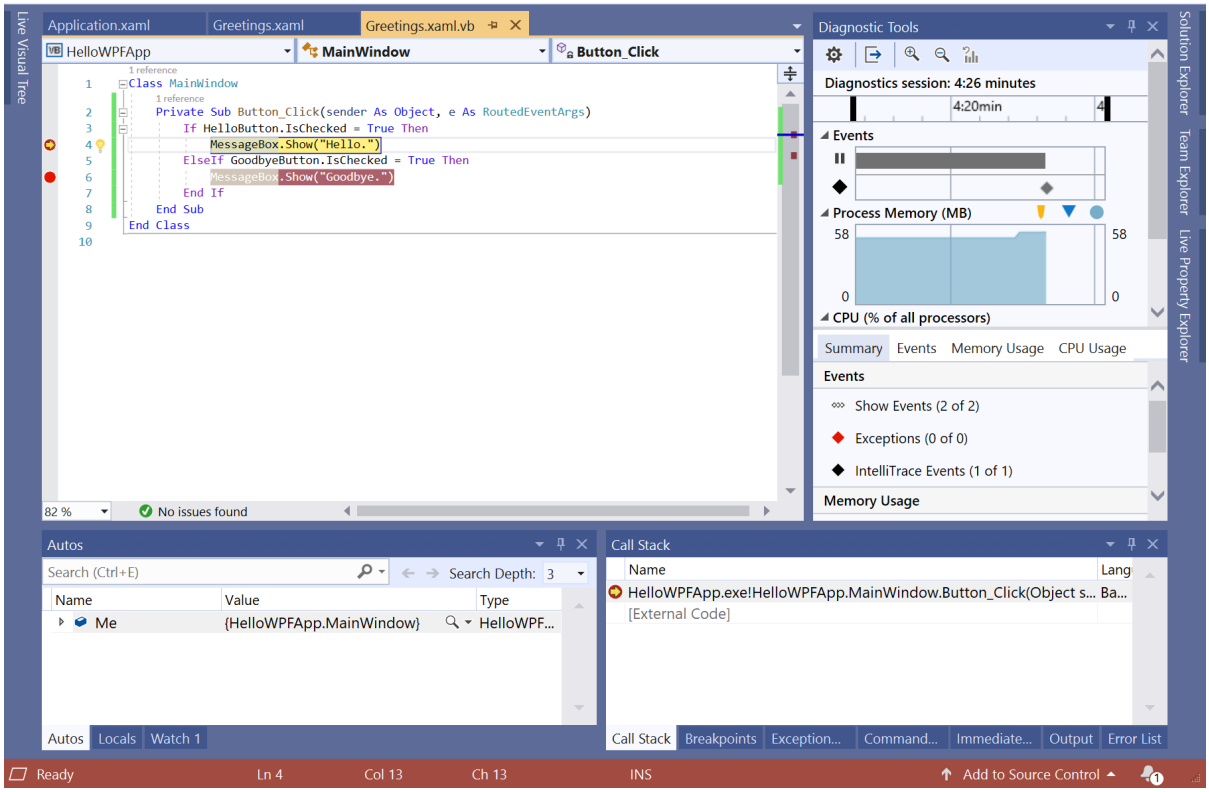
Add breakpoints

1. Open *Greetings.xaml.vb*, and select the following line: `MessageBox.Show("Hello.")`
2. Add a breakpoint by pressing **F9** or from the menu by selecting **Debug**, then **Toggle Breakpoint**.

   A red circle appears next to the line of code in the far left margin of the editor window.
3. Select the following line: `MessageBox.Show("Goodbye.")`.
4. Press the **F9** key to add a breakpoint, and then press **F5** to start debugging.
5. In the **Greetings** window, choose the **Hello** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Hello.")` is highlighted in yellow. At the bottom of the IDE, the Autos, Locals, and

Watch windows are docked together on the left side, and the Call Stack, Breakpoints, Exception Settings, Command, Immediate, and Output windows are docked together on the right side.



6. On the menu bar, choose **Debug** > **Step Out**.

   The application resumes execution, and a message box with the word "Hello" appears.

7. Choose the **OK** button on the message box to close it.

8. In the **Greetings** window, choose the **Goodbye** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Goodbye.")` is highlighted in yellow.

9. Choose the **F5** key to continue debugging. When the message box appears, choose the **OK** button on the message box to close it.

10. Close the application window to stop debugging.

11. On the menu bar, choose **Debug** > **Disable All Breakpoints**.

Build a release version of the application

Now that you've verified that everything works, you can prepare a release build of the application.

1. On the main menu, select **Build** > **Clean solution** to delete intermediate files and output files that were created during previous builds. This is not necessary, but it cleans up the debug build outputs.

2. Change the build configuration for HelloWPFApp from **Debug** to **Release** by using the dropdown control on the toolbar (it says "Debug" currently).

3. Build the solution by choosing **Build** > **Build Solution**.

Congratulations on completing this tutorial! You can find the *.exe* you built under your solution and project directory (...*HelloWPFApp**HelloWPFApp**bin**Release*).

See also

- What's new in Visual Studio 2017

- Productivity tips

- What's new in Visual Studio 2019
- Productivity tips

Create a Windows Forms app in Visual Studio with Visual Basic

4/16/2019 • 4 minutes to read • Edit Online

In this short introduction to the Visual Studio integrated development environment (IDE), you'll create a simple Visual Basic application that has a Windows-based user interface (UI).

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

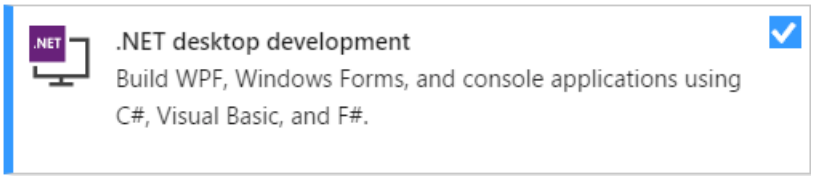
> **NOTE**
> Some of the screenshots in this tutorial use the dark theme. If you aren't using the dark theme but would like to, see the Personalize the Visual Studio IDE and Editor page to learn how.

Create a project

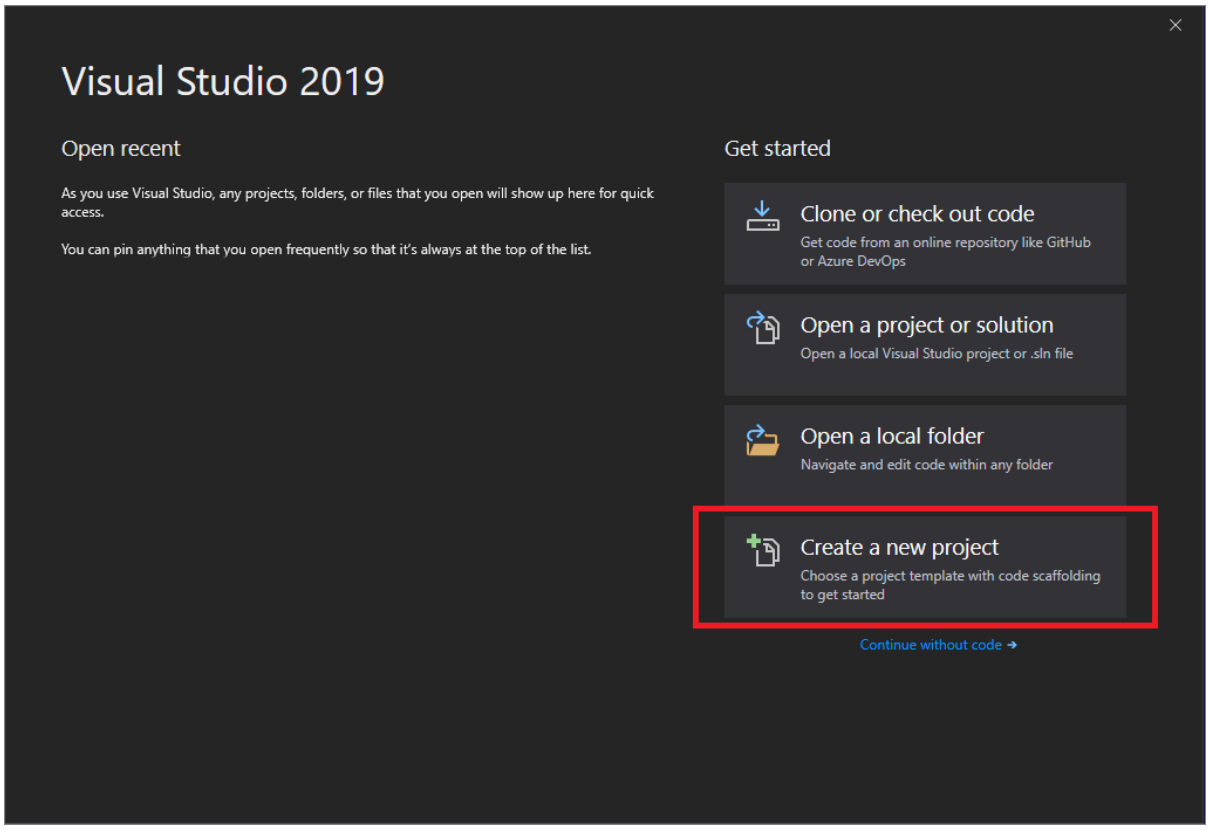
First, you'll create a Visual Basic application project. The project type comes with all the template files you'll need, before you've even added anything.

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **New** > **Project**.
3. In the **New Project** dialog box in the left pane, expand **Visual Basic**, and then choose **Windows Desktop**. In the middle pane, choose **Windows Forms App (.NET Framework)**. Then name the file `HelloWorld`.

   If you don't see the **Windows Forms App (.NET Framework)** project template, cancel out of the **New Project** dialog box and from the top menu bar, choose **Tools** > **Get Tools and Features**. The Visual Studio Installer launches. Choose the **.NET desktop development** workload, then choose **Modify**.

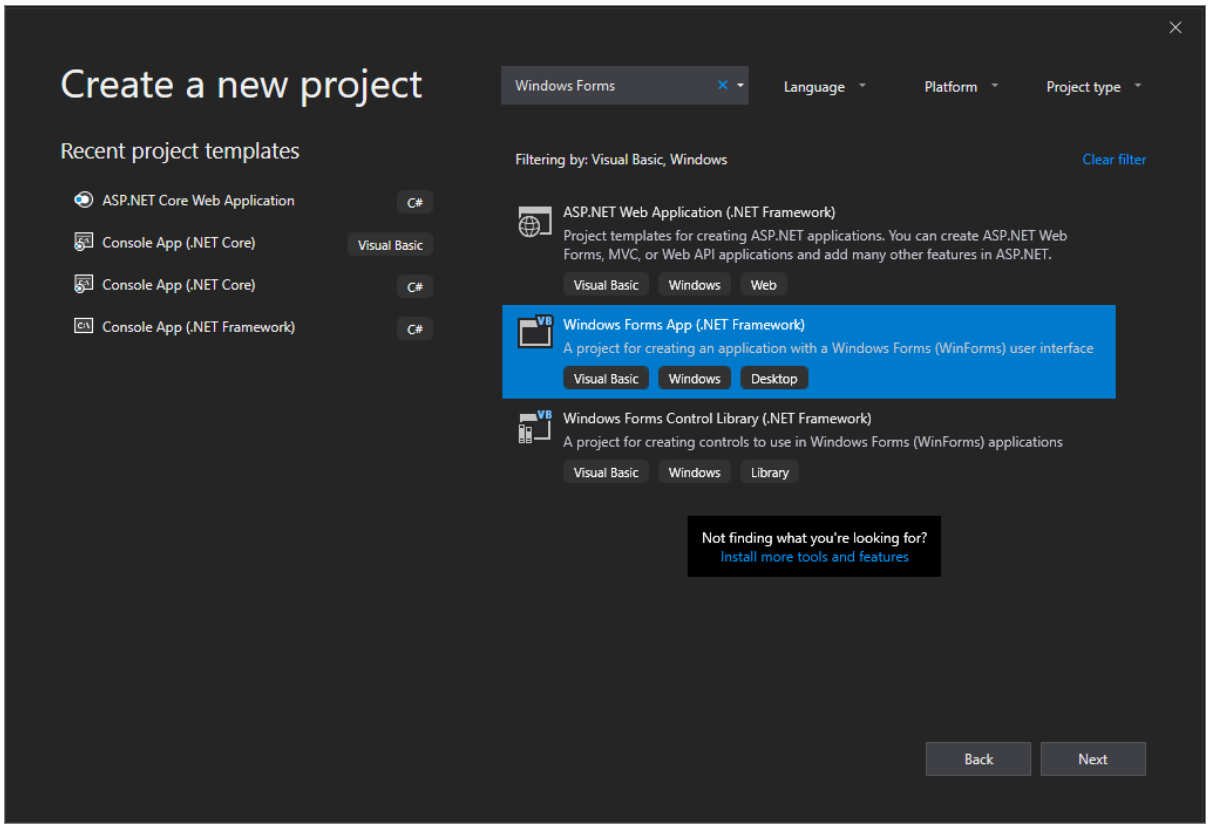


1. Open Visual Studio 2019.
2. On the start window, choose **Create a new project**.



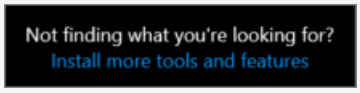
3. On the **Create a new project** window, enter or type *Windows Forms* in the search box. Next, choose **Visual Basic** from the Language list, and then choose **Windows** from the Platform list.

   After you apply the language and platform filters, choose the **Windows Forms App (.NET Framework)** template, and then choose **Next**.

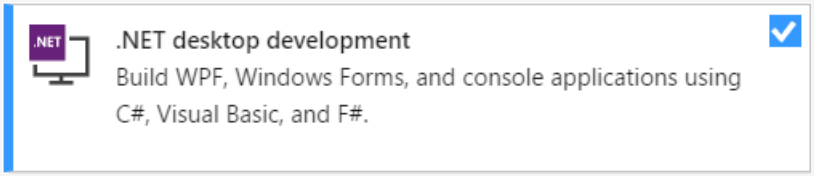


NOTE

If you do not see the **Windows Forms App (.NET Framework)** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.

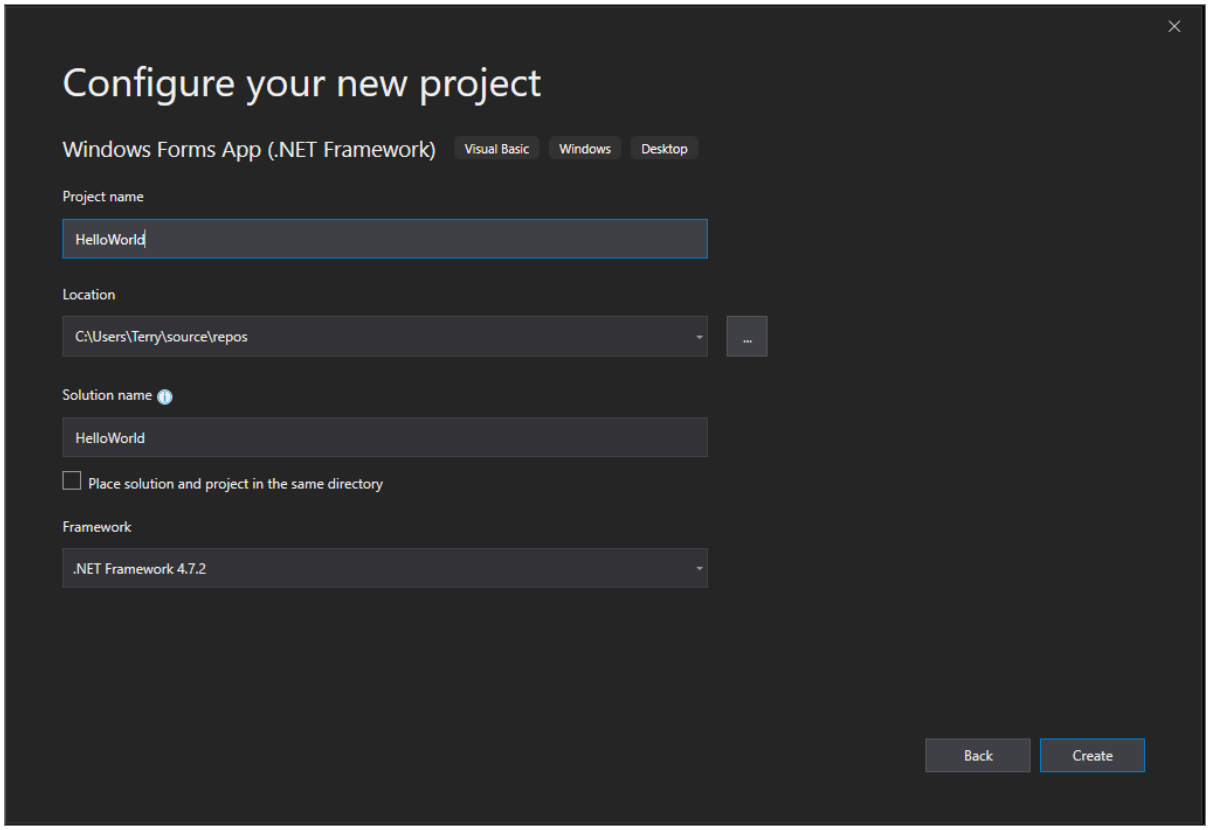


Next, in the Visual Studio Installer, choose the Choose the **.NET desktop development** workload.



After that, choose the **Modify** button in the Visual Studio Installer. You might be prompted to save your work; if so, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

4. In the **Configure your new project** window, type or enter *HelloWorld* in the **Project name** box. Then, choose **Create**.



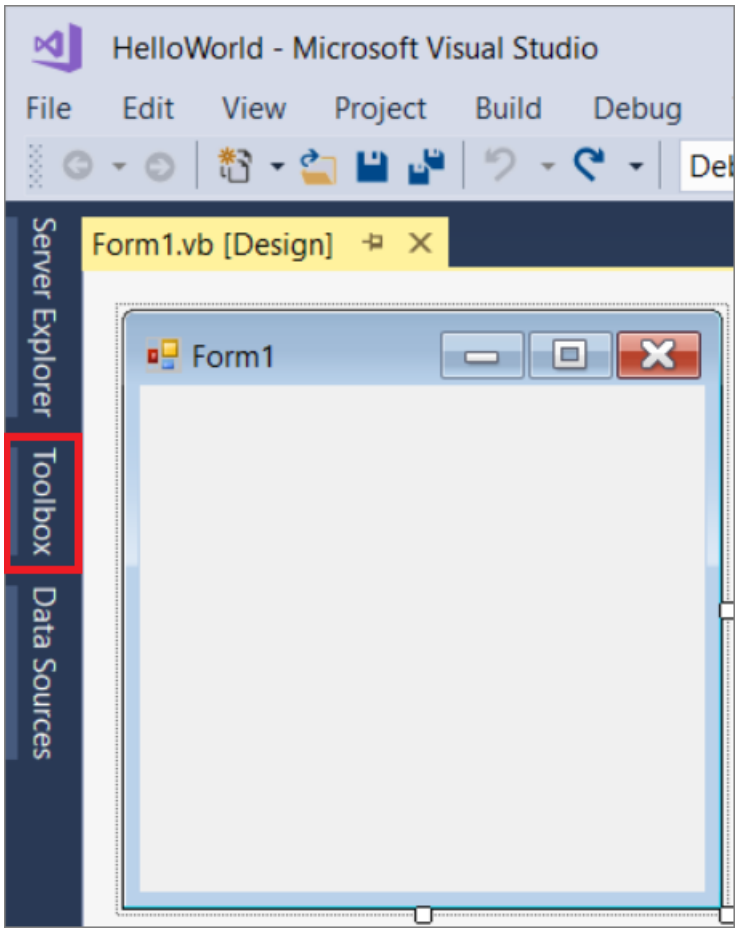
Visual Studio opens your new project.

Create the application

After you select your Visual Basic project template and name your file, Visual Studio opens a form for you. A form is a Windows user interface. We'll create a "Hello World" application by adding controls to the form, and then we'll run the application.

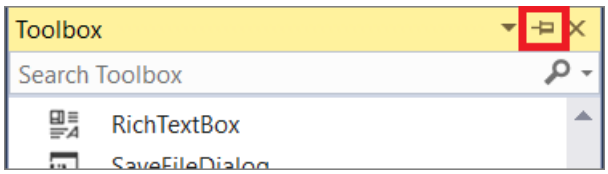
Add a button to the form

1. Click **Toolbox** to open the Toolbox fly-out window.

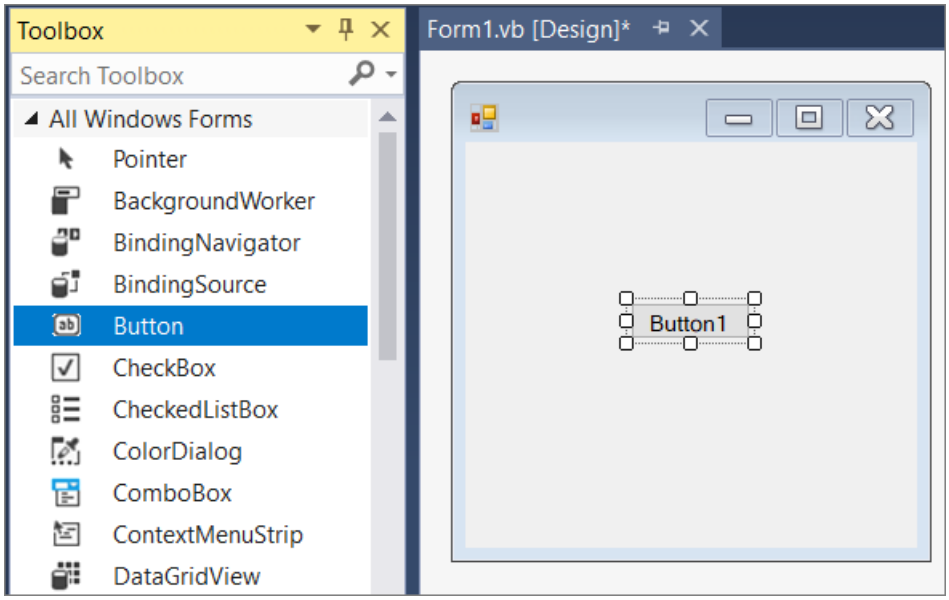


(If you don't see the **Toolbox** fly-out option, you can open by pressing **Ctrl**+**Alt**+**X**.)

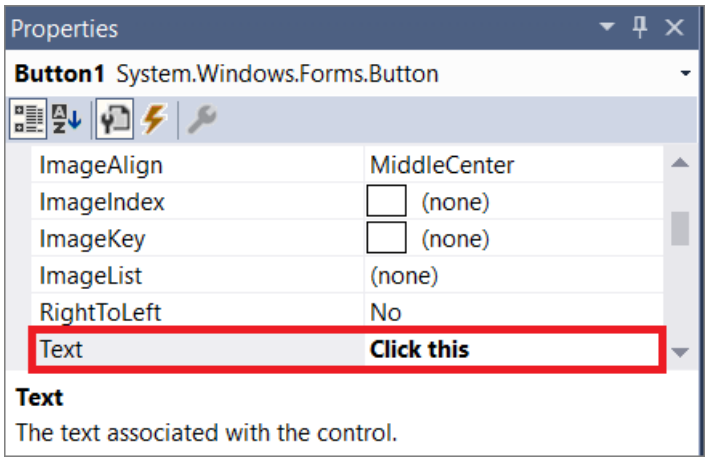
2. Click the **Pin** icon to dock the **Toolbox** window.



3. Click the **Button** control and then drag it onto the form.

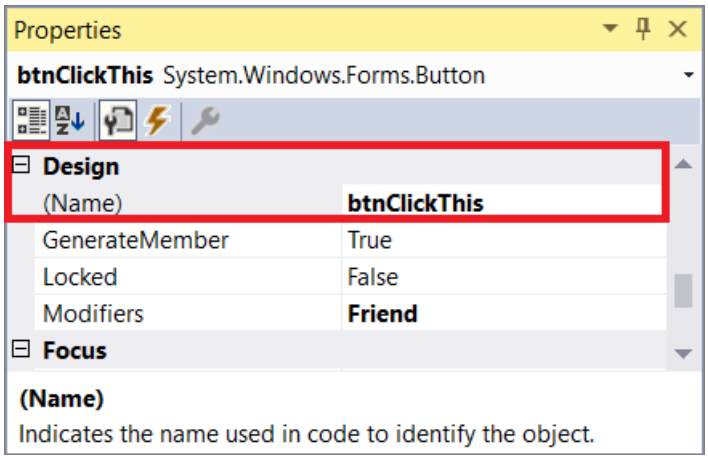


4. In the **Appearance** section (or the **Fonts** section) of the **Properties** window, type `Click this`, and then press **Enter**.



(If you don't see the **Properties** window, you can open it from the menu bar. To do so, click **View** > **Properties Window**. Or, press **F4**.)

5. In the **Design** section of the **Properties** window, change the name from **Button1** to `btnClickThis`, and then press **Enter**.



Add a label to the form

Now that we've added a button control to create an action, let's add a label control to send text to.

1. Select the **Label** control from the **Toolbox** window, and then drag it onto the form and drop it beneath the **Click this** button.

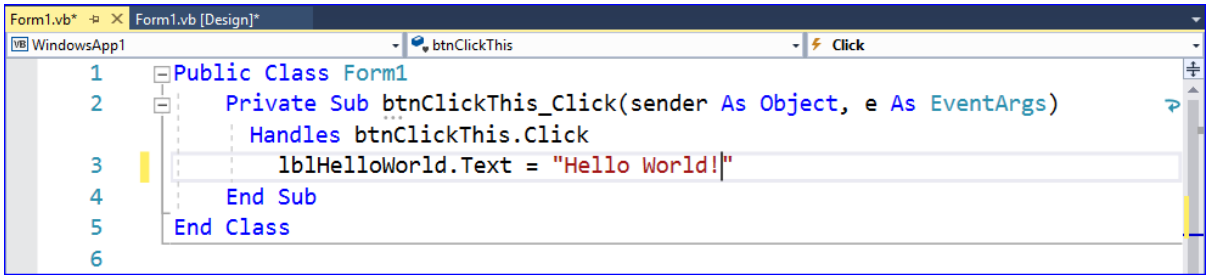
2. In the **Design** section of the **Properties** window, change the name from **Label1** to `lblHelloWorld`, and then press **Enter**.

Add code to the form

1. In the **Form1.vb [Design]** window, double-click the **Click this** button to open the **Form1.vb** window.

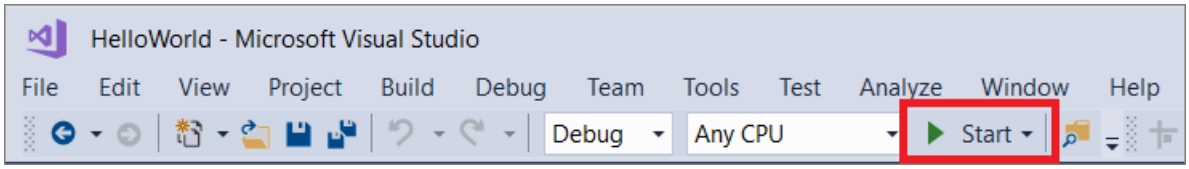
   (Alternatively, you can expand **Form1.vb** in **Solution Explorer**, and then click **Form1**.)

2. In the **Form1.vb** window, between the **Private Sub** line and the **End Sub** line (or between the **Public Class Form1** line and the **End Class** line), type the following code.



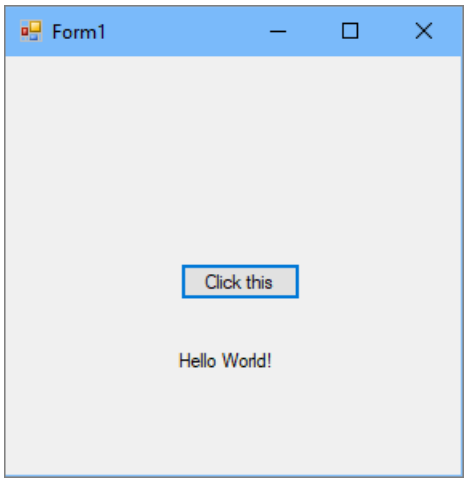
Run the application

1. Click the **Start** button to run the application.



   Several things will happen. In the Visual Studio IDE, the **Diagnostics Tools** window will open, and an **Output** window will open, too. But outside of the IDE, a **Form1** dialog box appears. It will include your **Click this** button and text that says **Label1**.

2. Click the **Click this** button in the **Form1** dialog box. Notice that the **Label1** text changes to **Hello World!**.



Congratulations on completing this quickstart! We hope you learned a little bit about Visual Basic and the Visual Studio IDE. If you'd like to delve deeper, please continue with a tutorial in the **Tutorials** section of the table of contents.

See also

- Quickstart: Create a console app in Visual Studio with Visual Basic
- Learn more about Visual Basic IntelliSense

Tutorial: Open a project from a repo

5/30/2019 • 3 minutes to read • Edit Online

In this tutorial, you'll use Visual Studio to connect to a repository for the first time and then open a project from it.

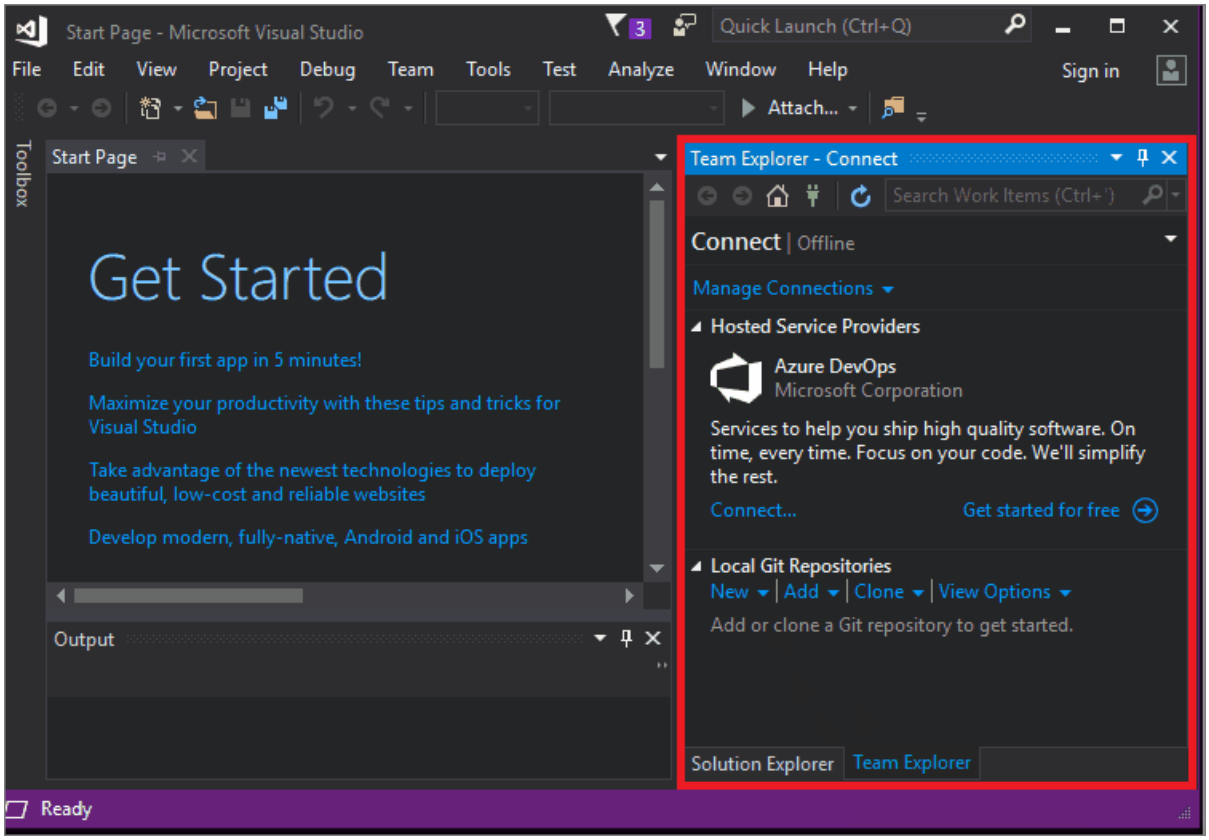
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

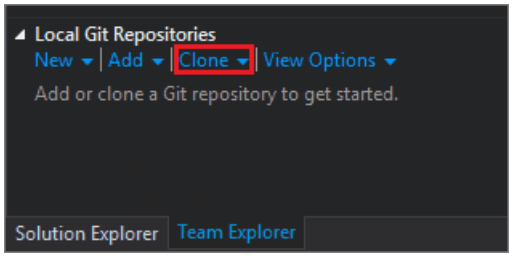
Open a project from a GitHub repo

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.



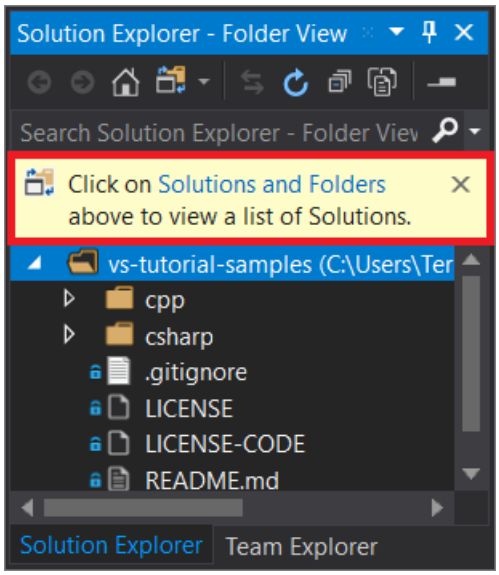
3. In the **Local Git Repositories** section, choose **Clone**.



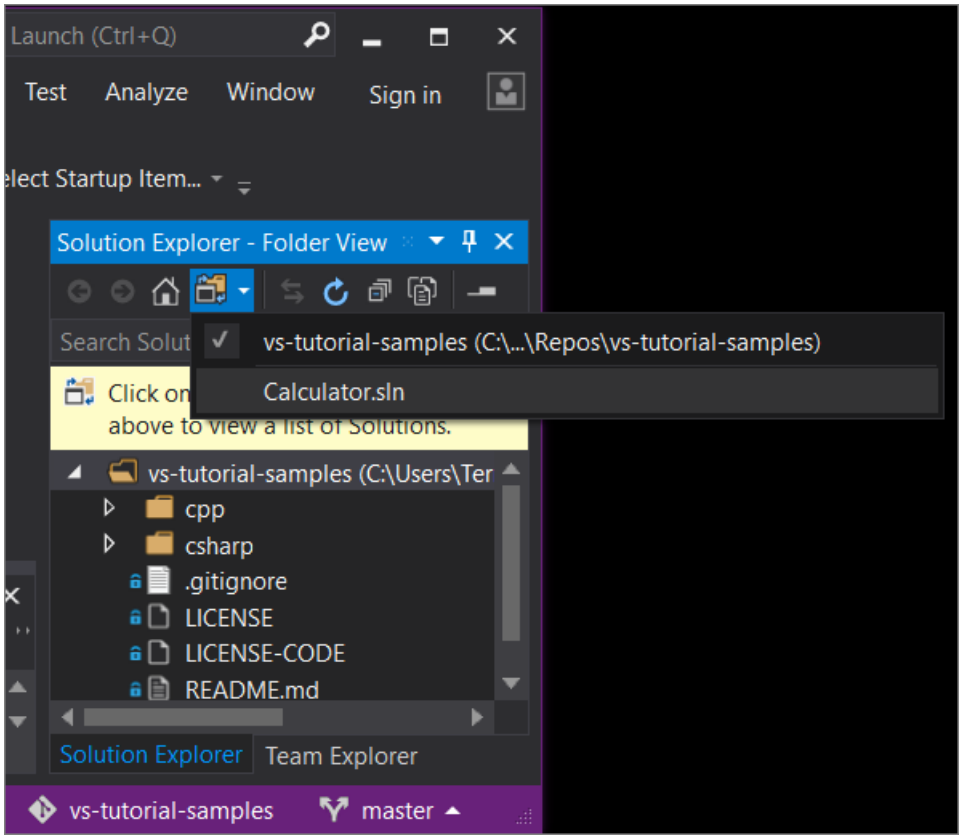
4. In the box that says ***Enter the URL of a Git repo to clone***, type or paste the URL for your repo, and then press **Enter**. (You might receive a prompt to sign in to GitHub; if so, do so.)

   After Visual Studio clones your repo, Team Explorer closes and Solution Explorer opens. A message appears

that says *Click on Solutions and Folders above to view a list of Solutions*. Choose **Solutions and Folders**.



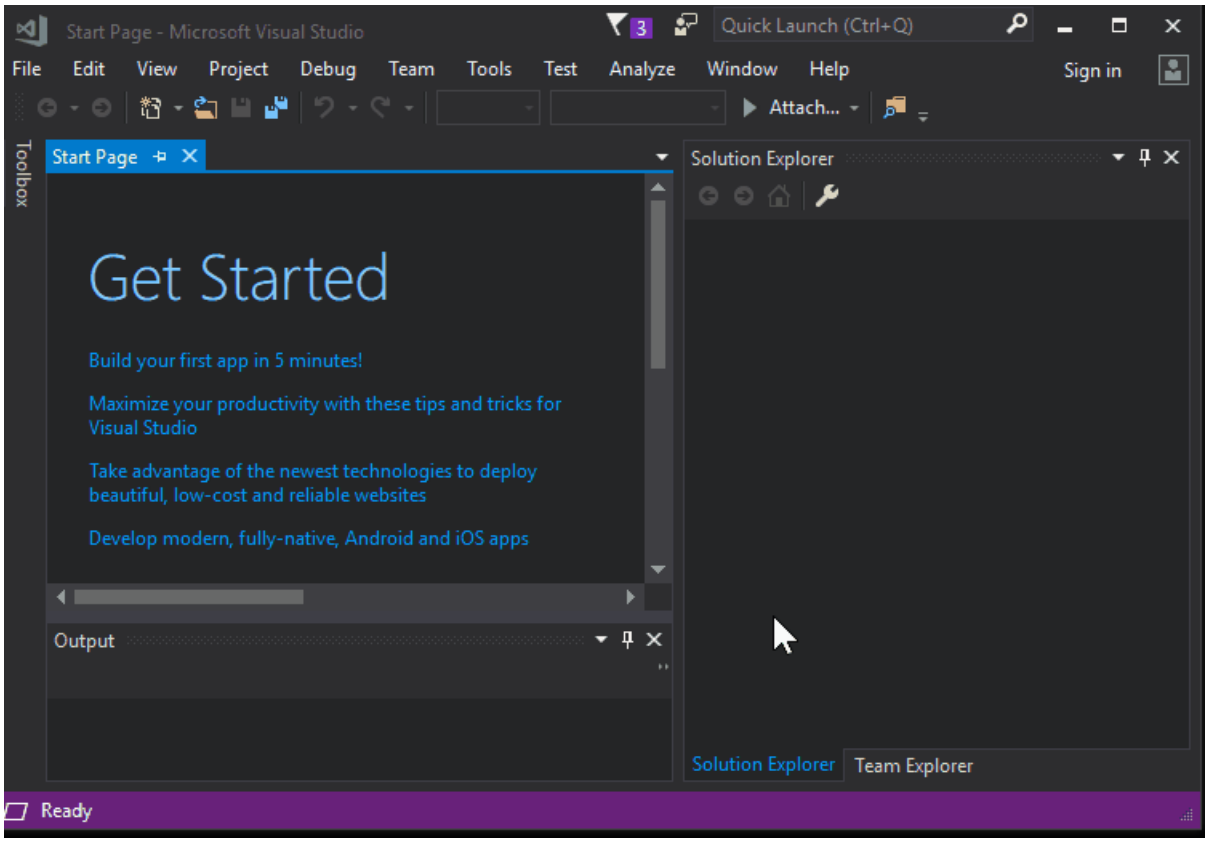
5. If you have a solution file available, it will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.



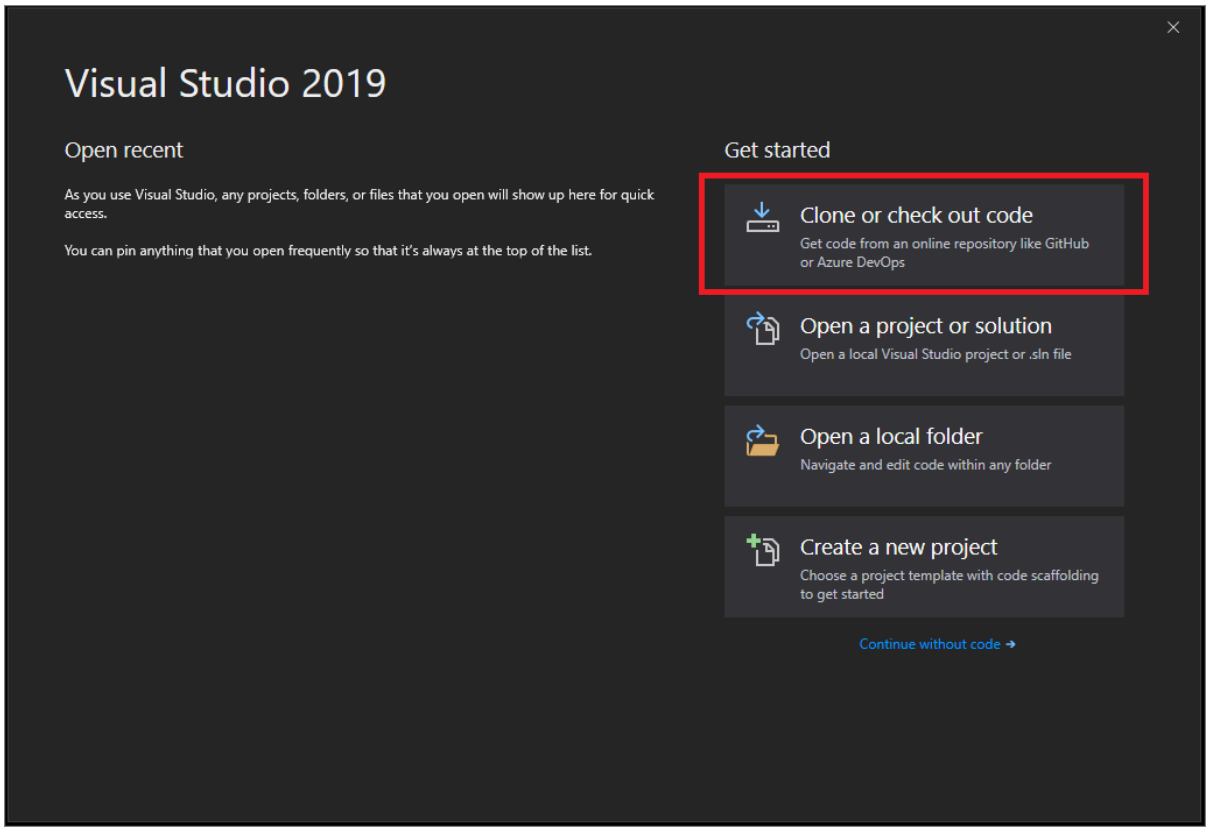
   If you do not have a solution file (specifically, a .sln file) in your repo, the fly-out menu will say "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Review your work

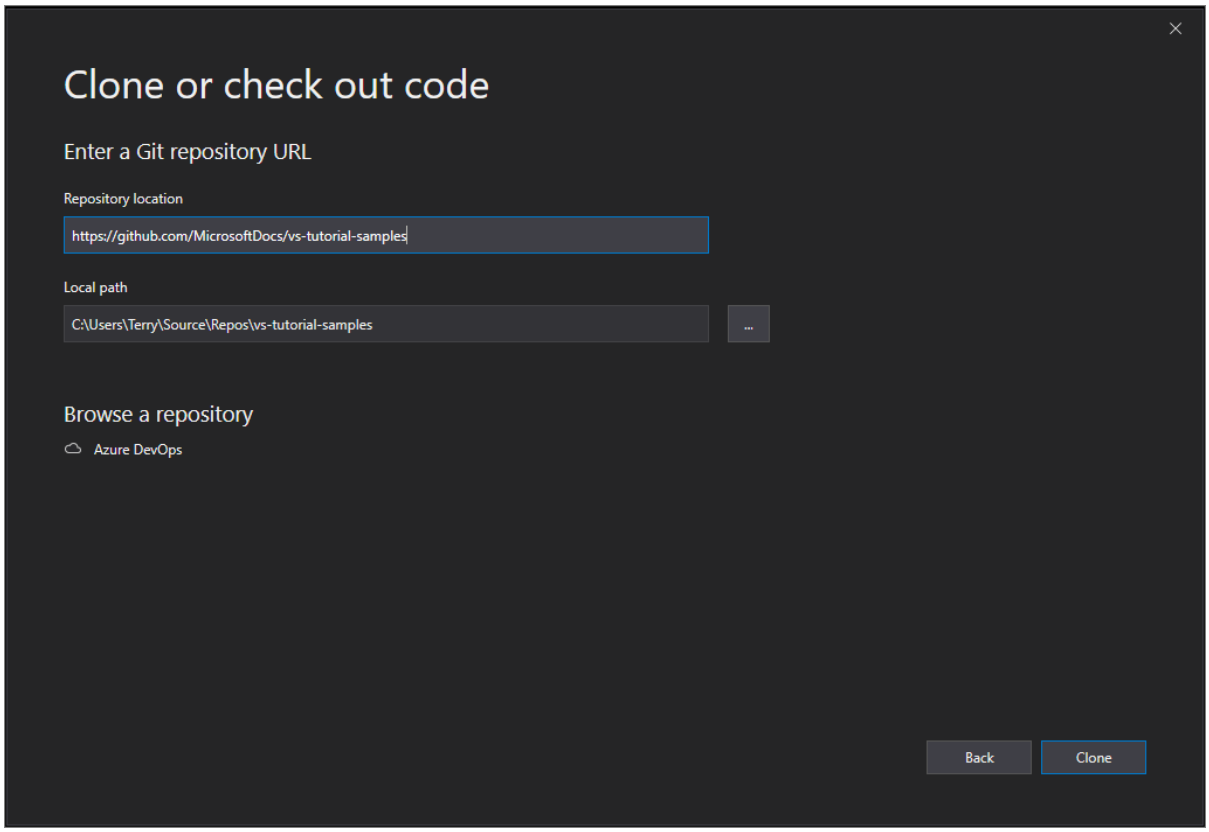
View the following animation to check the work that you completed in the previous section.



1. Open Visual Studio 2019.
2. On the start window, choose **Clone or check out code**.

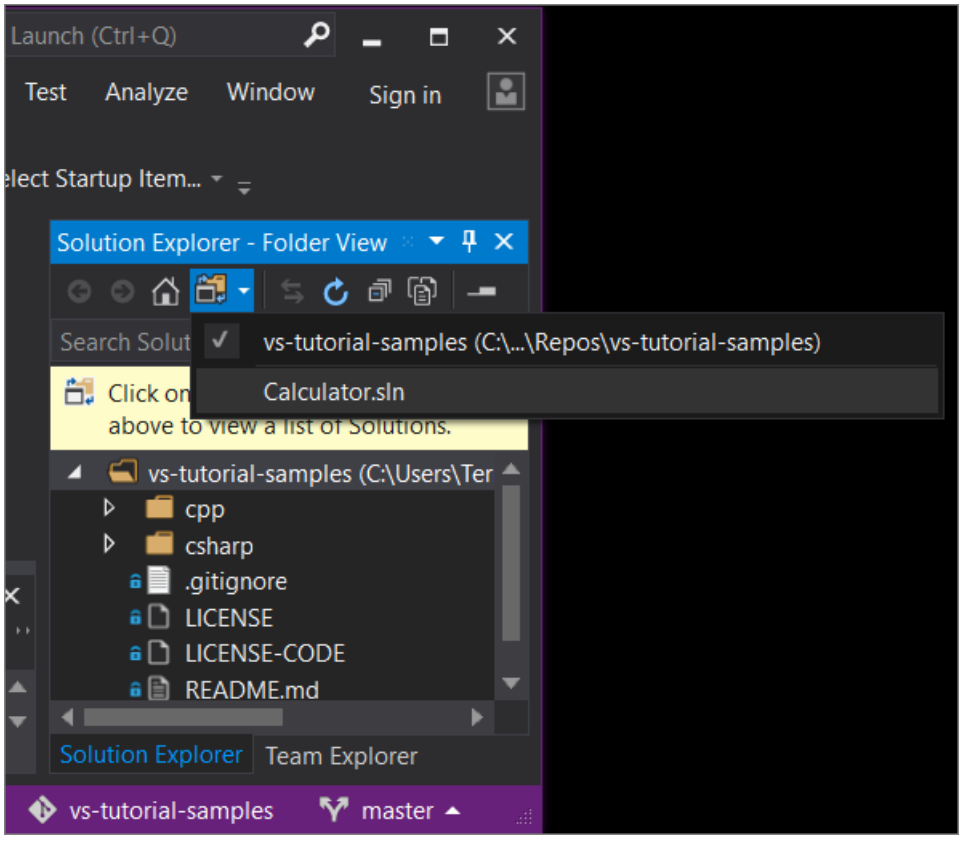


3. Enter or type the repository location, and then choose **Clone**.



Visual Studio opens the project from the repo.

4. If you have a solution file available, it will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.

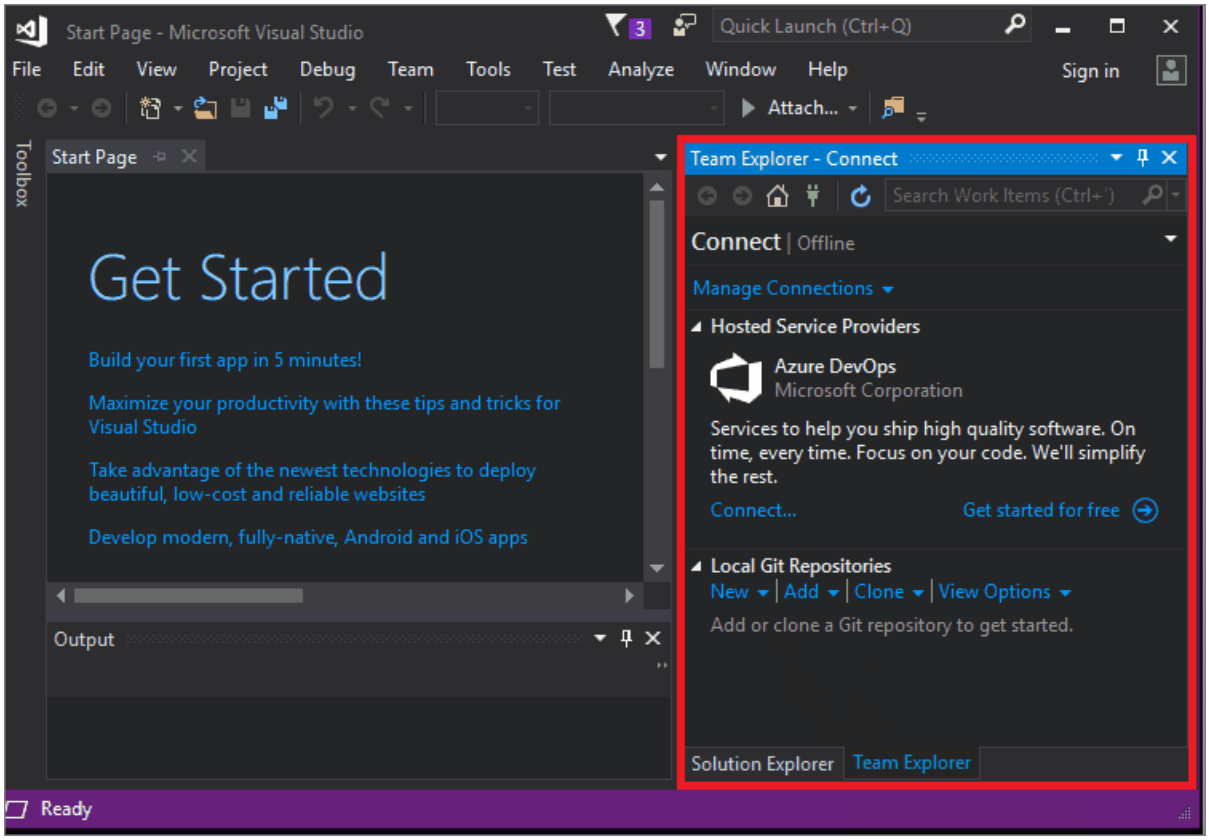


If you do not have a solution file (specifically, a .sln file) in your repo, the fly-out menu will say "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Open a project from an Azure DevOps repo

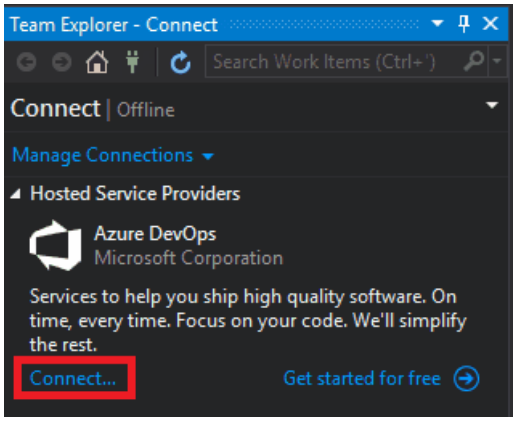
1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.

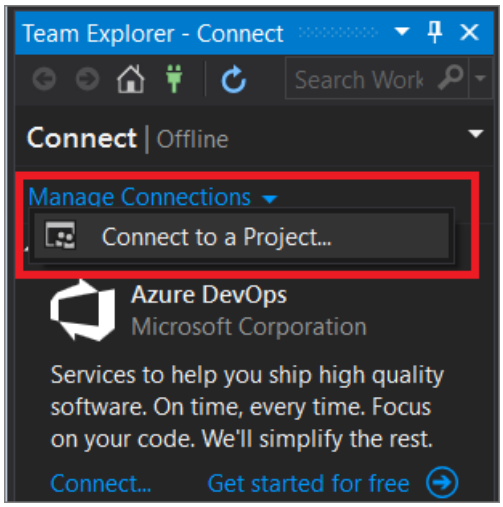


3. Here are two ways to connect to your Azure DevOps repo:

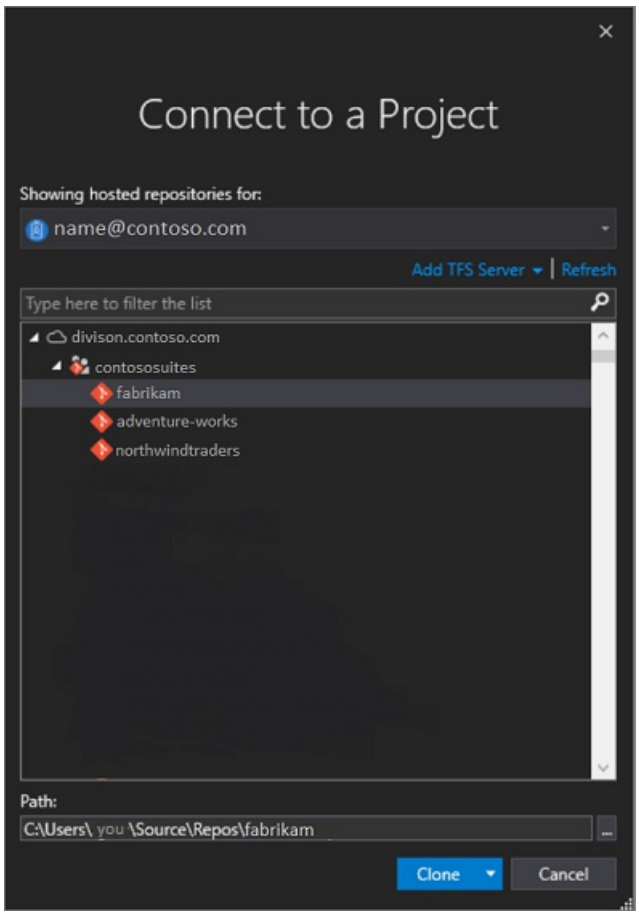
   - In the **Hosted Service Providers** section, choose **Connect...**.



   - In the **Manage Connections** drop-down list, choose **Connect to a Project...**.

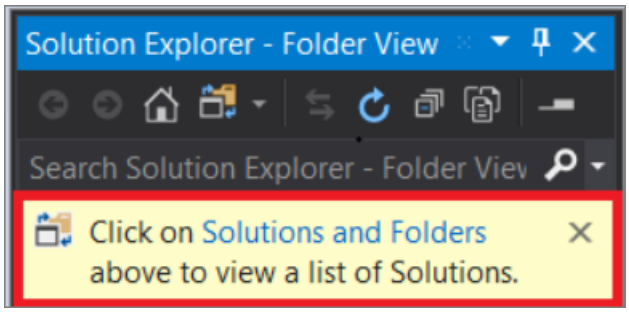


4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then choose **Clone**.



> **NOTE**
>
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

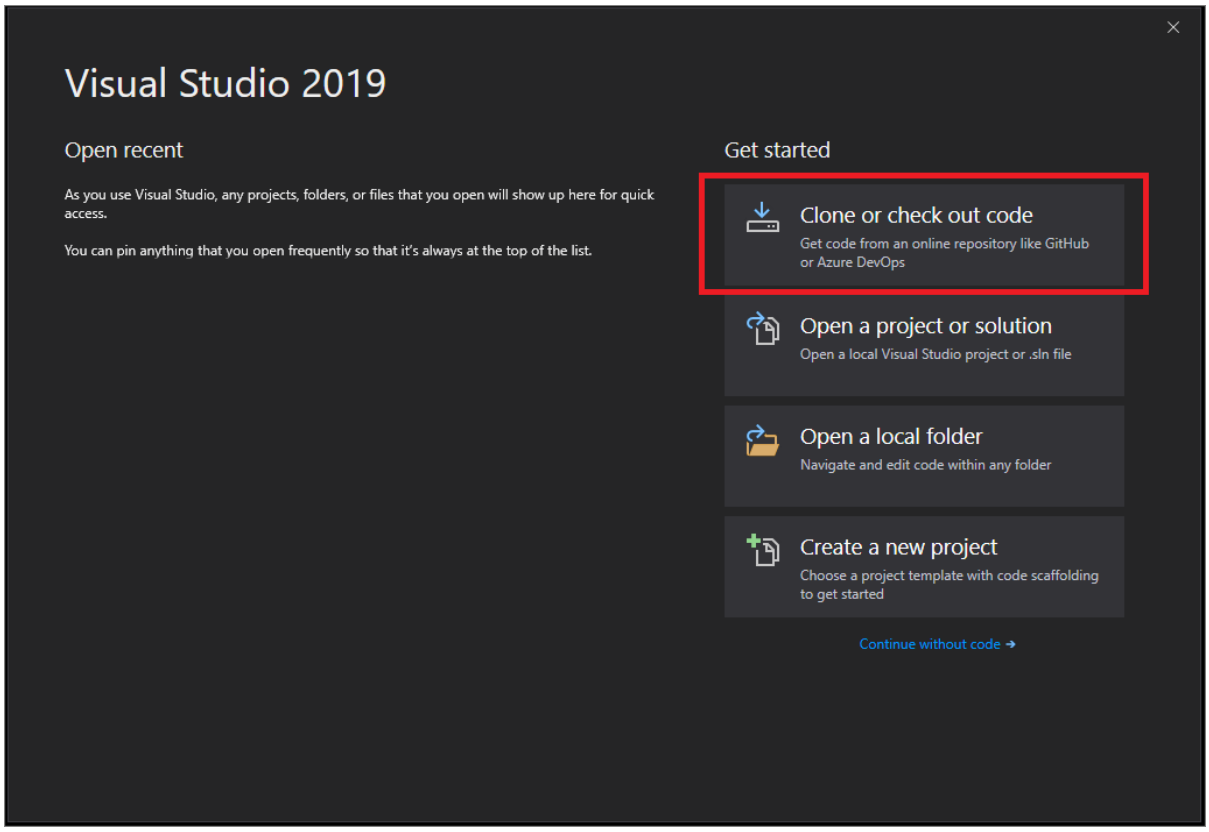
5. After Visual Studio clones your repo, Team Explorer closes and Solution Explorer opens. A message appears that says *Click on Solutions and Folders above to view a list of Solutions*. Choose **Solutions and Folders**.



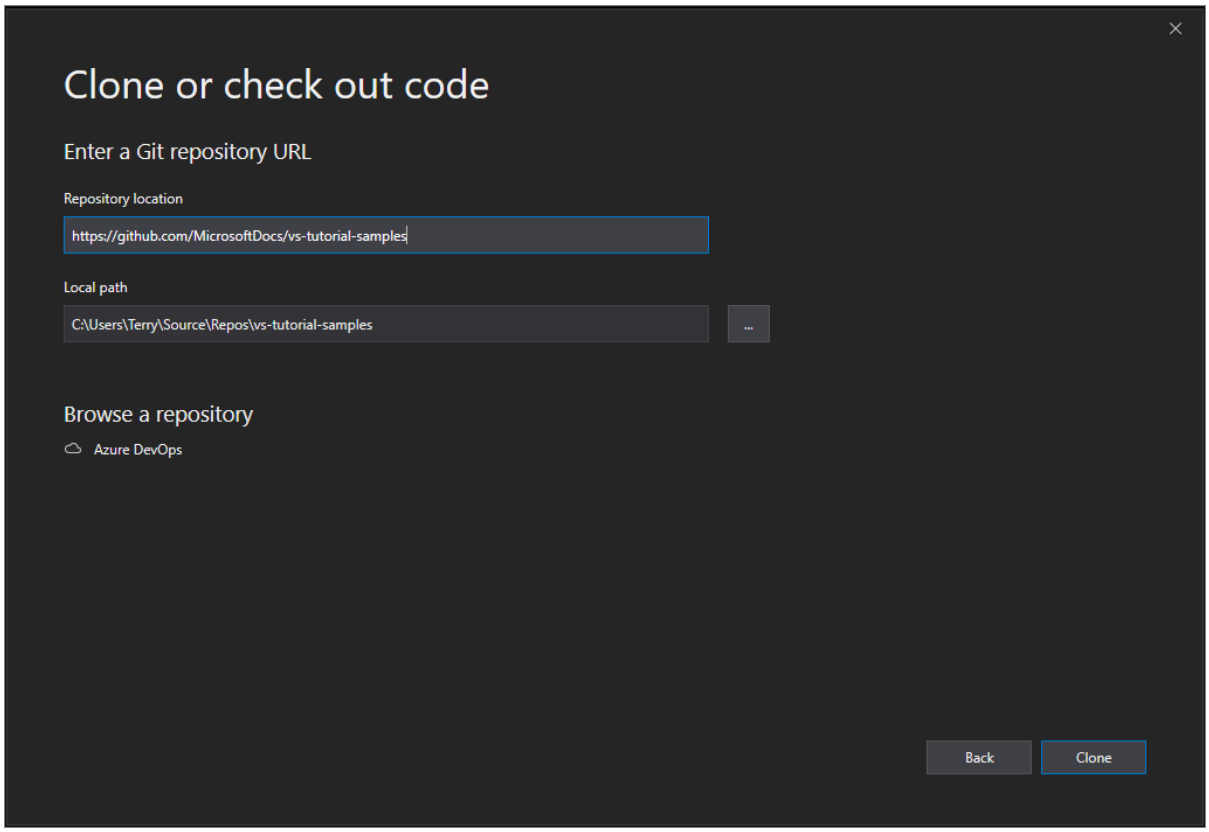
A solution file (specifically, a .sln file), will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.

If you do not have a solution file in your repo, the fly-out menu will say "No Solutions Found". However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

1. Open Visual Studio 2019.
2. On the start window, choose **Clone or check out code**.

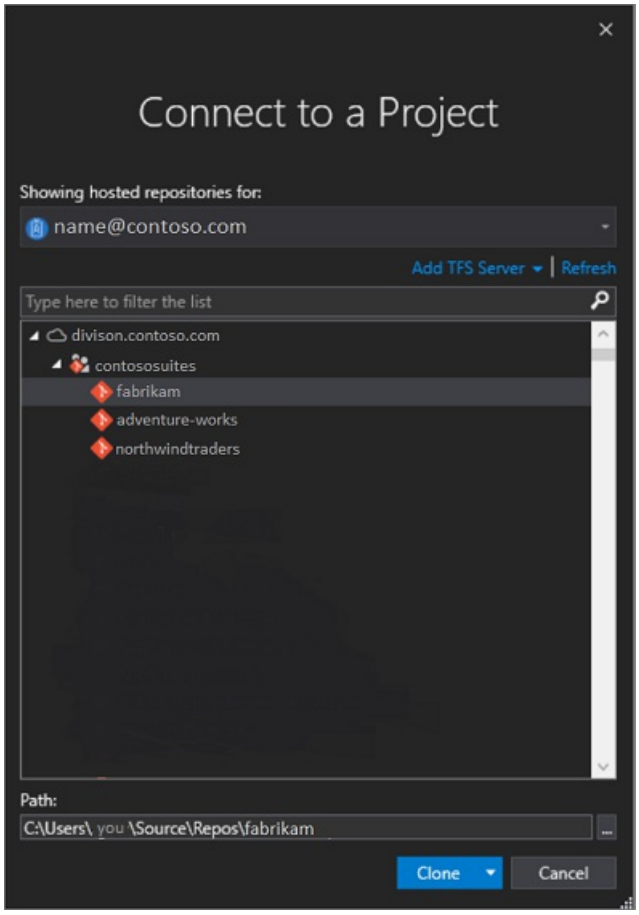


3. In the **Browse a repository** section, choose **Azure DevOps**.



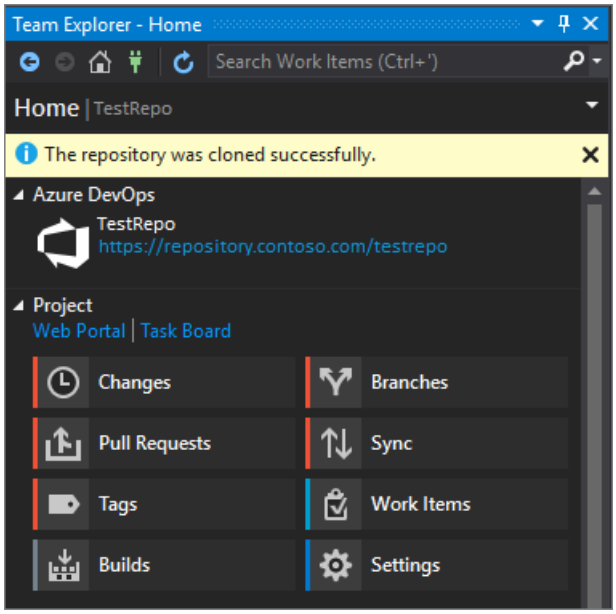
If you see a sign-in window, sign in to your account.

4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then choose **Clone**.

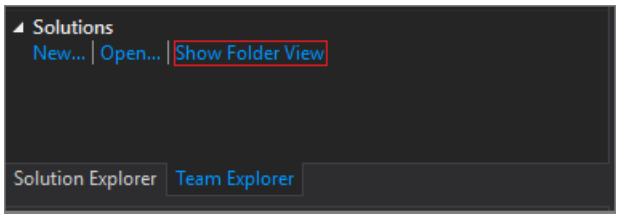


> **NOTE**
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

Visual Studio opens **Team Explorer** and a notification appears when the clone is complete.

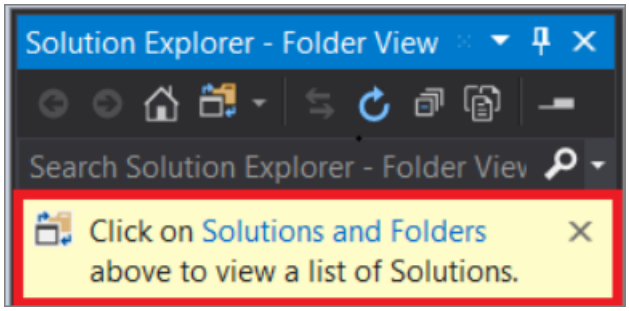


5. To view your folders and files, choose the **Show Folder View** link.



Visual Studio opens **Solution Explorer**.

6. Choose the **Solutions and Folders** link to search for a solution file (specifically, a .sln file) to open.



If you do not have a solution file in your repo, a "No Solutions Found" message appears. However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Next steps

If you're ready to code with Visual Studio, dive into any of the following language-specific tutorials:

- Visual Studio tutorials | **C#**
- Visual Studio tutorials | **Visual Basic**
- Visual Studio tutorials | **C++**
- Visual Studio tutorials | **Python**

- Visual Studio tutorials | **JavaScript**, **TypeScript**, and **Node.js**

See also

- Azure DevOps Services: Get started with Azure Repos and Visual Studio
- Microsoft Learn: Get started with Azure DevOps

Learn to use the code editor

4/17/2019 • 6 minutes to read • Edit Online

In this 10-minute introduction to the code editor in Visual Studio, we'll add code to a file to look at some of the ways that Visual Studio makes writing, navigating, and understanding code easier.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

This article assumes you're already familiar with Visual Basic. If you aren't, we suggest you look at a tutorial such as Get started with Visual Basic in Visual Studio first.

> **TIP**
> To follow along with this article, make sure you have the Visual Basic settings selected for Visual Studio. For information about selecting settings for the integrated development environment (IDE), see Select environment settings.

Create a new code file

Start by creating a new file and adding some code to it.

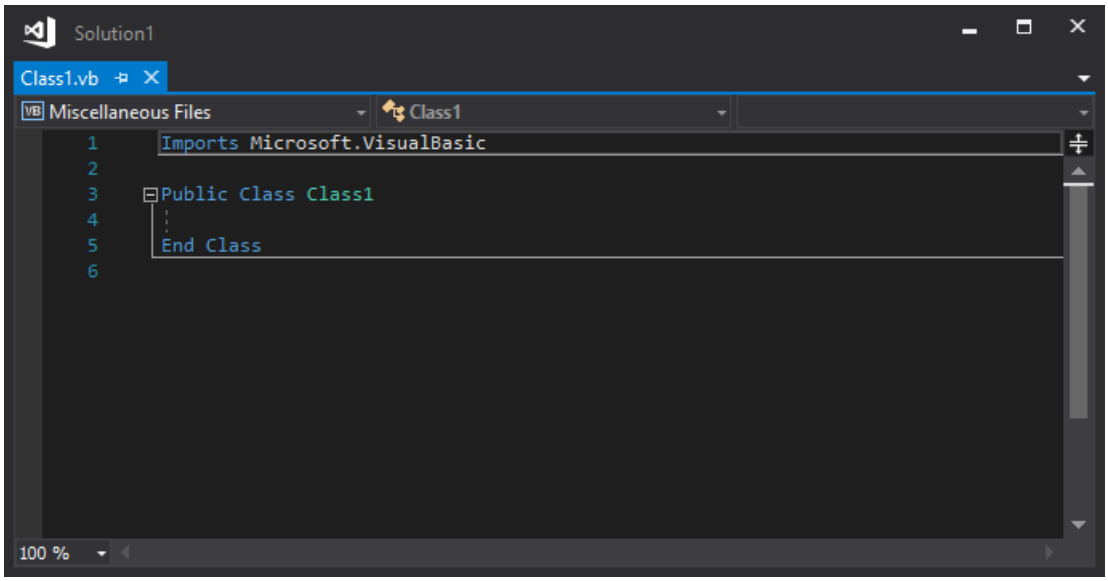
1. Open Visual Studio.

1. Open Visual Studio. Press **Esc** or click **Continue without code** on the start window to open the development environment.

2. From the **File** menu on the menu bar, choose **New File**.

3. In the **New File** dialog box, under the **General** category, choose **Visual Basic Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a Visual Basic class. (You can already notice that you don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers, such as syntax highlighting. All you need is a code file!)

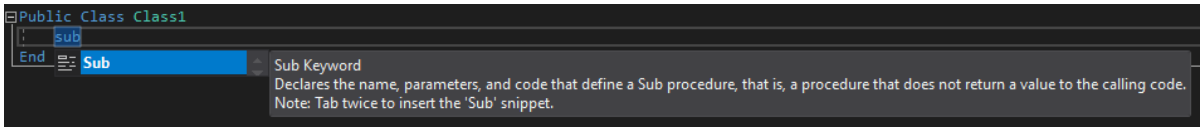


Use code snippets

Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including Visual Basic, C#, and C++. Let's add the Visual Basic **Sub** snippet to our file.

1. Place your cursor above the line that says `End Class`, and type **sub**.

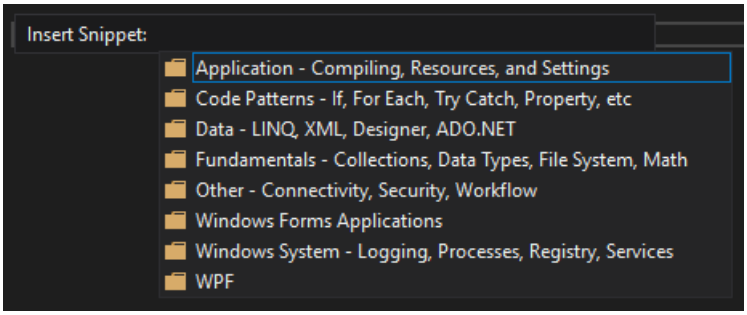
   A pop-up dialog box appears with information about the `Sub` keyword and how to insert the **Sub** code snippet.



2. Press **Tab** twice to insert the code snippet.

   The outline for the Sub procedure `MySub()` is added to the file.

The available code snippets vary for different programming languages. You can look at the available code snippets for Visual Basic by choosing **Edit** > **IntelliSense** > **Insert Snippet** (or press **Ctrl**+**K**, **Ctrl**+**X**). For Visual Basic, code snippets are available for the following categories:



There are snippets for determining if a file exists on the computer, writing to a text file, reading a registry value, executing a SQL query, creating a For Each...Next statement, and many more.

Comment out code

The toolbar, which is the row of buttons under the menu bar in Visual Studio, can help make you more productive as you code. For example, you can toggle IntelliSense completion mode, increase or decrease a line indent, or comment out code that you don't want to compile. (IntelliSense is a coding aid that displays a list of matching

methods, amongst other things.) In this section, we'll comment out some code.

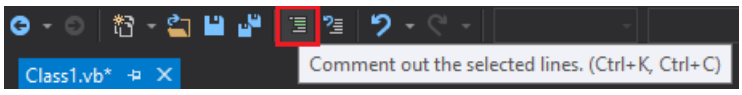
1. Paste the following code into the `MySub()` procedure body.

```
' _words is a string array that we'll sort alphabetically
Dim _words = New String() {
"the",
"quick",
"brown",
"fox",
"jumps"
}

Dim morewords = New String() {
"over",
"the",
"lazy",
"dog"
}

Dim query = From word In _words
            Order By word.Length
            Select word
```

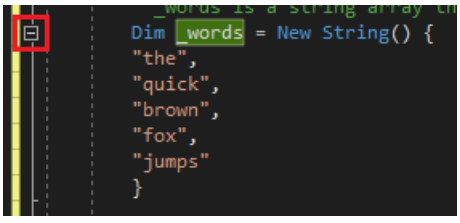
2. We're not using the `morewords` array, but we may use it later so we don't want to completely delete it. Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing curly brace, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**K**, **Ctrl**+**C**.



   The Visual Basic comment character `'` is added to the beginning of each selected line to comment out the code.

Collapse code blocks

You can collapse sections of code to focus just on the parts that are of interest to you. To practice, let's collapse the `_words` array to one line of code. Choose the small gray box with the minus sign inside it in the margin of the line that says `Dim _words = New String() {`. Or, if you're a keyboard user, place the cursor anywhere in the array definition and press **Ctrl**+**M**, **Ctrl**+**M**.



The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

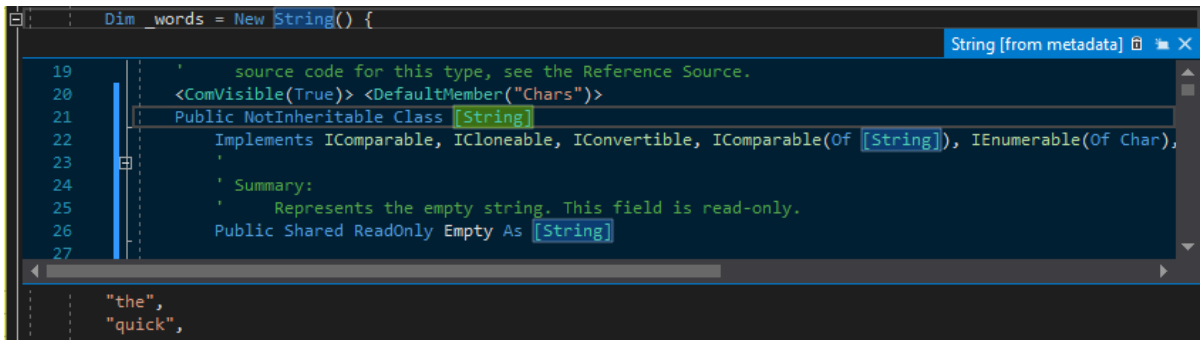
View symbol definitions

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing **Go to Definition** anywhere the symbol is referenced. An

even quicker way that doesn't move your focus away from the file you're working in is to use Peek Definition. Let's peek at the definition of the `String` type.

1. Right-click on the word `String` and choose **Peek Definition** from the content menu. Or, press **Alt**+**F12**.

   A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

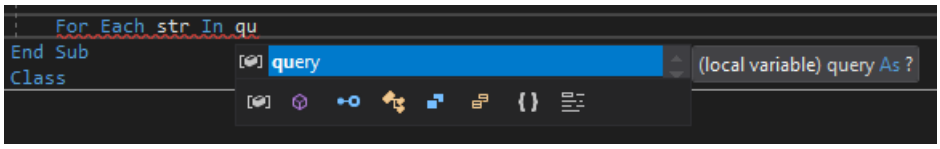
Use IntelliSense to complete words

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it. Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
For Each str In qu
```

   You see IntelliSense show you **Quick Info** about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense's word completion functionality, press **Tab**.

3. Finish off the code block to look like the following code.

```
For Each str In query
    Console.WriteLine(str)
Next
```

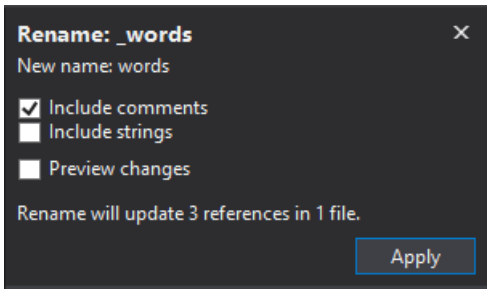
Refactor a name

Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `_words` variable to `words`.

1. Place your cursor over the definition of the `_words` variable and choose **Rename** from the right-click or context menu.

   A pop-up **Rename** dialog box appears at the top right of the editor.

2. With the variable `_words` still selected, type in the desired name of **words**. Notice that the reference to `words` in the query is also automatically renamed. Before you press **Enter** or click **Apply**, select the **Include comments** checkbox in the **Rename** pop-up box.



3. Press **Enter** or click **Apply**.

   Both occurrences of `words` are renamed, as well as the reference to `words` in the code comment.

Next steps

Learn about projects and solutions

See also

- Code snippets
- Navigate code
- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

Compile and build in Visual Studio

7/23/2019 • 2 minutes to read • Edit Online

When you build source code, the build engine creates assemblies and executable applications. In general, the build process is very similar across many different project types such as Windows, ASP.NET, mobile apps, and others. The build process is also similar across programming languages such as C#, Visual Basic, C++, and F#.

By building your code often, you can quickly identify compile-time errors, such as incorrect syntax, misspelled keywords, and type mismatches. You can also detect and correct run-time errors, such as logic errors and semantic errors, by building and running debug versions of the code.

A successful build validates that the application's source code contains correct syntax and that all static references to libraries, assemblies, and other components can resolve. An application executable is produced that can be tested for proper functioning in both a debugging environment and through a variety of manual and automated tests to validate code quality. Once the application has been fully tested, you can compile a release version to deploy to your customers. For an introduction to this process, see Walkthrough: Building an application.

You can use any of the following methods to build an application: the Visual Studio IDE, the MSBuild command-line tools, and Azure Pipelines:

BUILD METHOD	BENEFITS
IDE	- Create builds immediately and test them in a debugger. - Run multi-processor builds for C++ and C# projects. - Customize different aspects of the build system.
MSBuild command line	- Build projects without installing Visual Studio. - Run multi-processor builds for all project types. - Customize most areas of the build system.
Azure Pipelines	- Automate your build process as part of a continuous integration/continuous delivery pipeline. - Apply automated tests with every build. - Employ virtually unlimited cloud-based resources for build processes. - Modify the build workflow and create build activities to perform deeply customized tasks.

The documentation in this section goes into further details of the IDE-based build process. For more information on the other methods, see MSBuild and Azure Pipelines, respectively.

> **NOTE**
>
> This topic applies to Visual Studio on Windows. For Visual Studio for Mac, see Compile and build in Visual Studio for Mac.

Overview of building from the IDE

When you create a project, Visual Studio created default build configurations for the project and the solution that contains the project. These configurations define how the solutions and projects are built and deployed. Project configurations in particular are unique for a target platform (such as Windows or Linux) and build type (such as debug or release). You can edit these configurations however you like, and can also create your own configurations as needed.

For a first introduction to building within the IDE, see Walkthrough: Building an application.

Next, see Building and cleaning projects and solutions in Visual Studio to learn about the different aspects customizations you can make to the process. Customizations include changing output directories, specifying custom build events, managing project dependencies, managing build log files, and suppressing compiler warnings.

From there, you can explore a variety of other tasks:

- Understand build configurations
- Understand build platforms
- Manage project and solution properties.
- Specify build events in C# and Visual Basic.
- Set build options
- Build multiple projects in parallel.

See also

- Building (compiling) website projects
- Compile and build (Visual Studio for Mac)

Tutorial: Learn to debug Visual Basic code using Visual Studio

5/15/2019 • 11 minutes to read • Edit Online

This article introduces the features of the Visual Studio debugger in a step-by-step walkthrough. If you want a higher-level view of the debugger features, see First look at the debugger. When you *debug your app*, it usually means that you are running your application with the debugger attached. When you do this, the debugger provides many ways to see what your code is doing while it runs. You can step through your code and look at the values stored in variables, you can set watches on variables to see when values change, you can examine the execution path of your code, see whether a branch of code is running, and so on. If this is the first time that you've tried to debug code, you may want to read Debugging for absolute beginners before going through this article.

In this tutorial, you will:

- Start the debugger and hit breakpoints.
- Learn commands to step through code in the debugger
- Inspect variables in data tips and debugger windows
- Examine the call stack

Prerequisites

You must have Visual Studio 2019 installed and the **.NET desktop development** workload.

You must have Visual Studio 2017 installed and the **.NET desktop development** workload.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you need to install the workload but already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. The Visual Studio Installer launches. Choose the **.NET desktop development** workload, then choose **Modify**.

Create a project

1. Open Visual Studio.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **visual basic**, choose **Templates**, then choose **Create new Console App (.NET Framework) project**. In the dialog box that appears, type a name like **get-started-debugging**, and then choose **Create**.

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New project** dialog box, under **Visual Basic**, choose **Windows Desktop**, and then in the middle pane choose **Console App (.NET Framework)**. Then, type a name like **get-started-debugging** and click **OK**.

   If you don't see the **Console App (.NET Framework)** project template, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **.NET desktop development** workload, then choose **Modify**.

   Visual Studio creates the project.

2. In *Module1.vb*, replace the following code

```
Module Module1

    Sub Main()
    End Sub

End Module
```

with this code:

```
Imports System
Imports System.Collections.Generic

Public Class Shape

  ' A few example members
    Public Property X As Integer
        Get
            Return X
        End Get
        Set
        End Set
    End Property

    Public Property Y As Integer
        Get
            Return Y
        End Get
        Set
        End Set
    End Property

    Public Property Height As Integer
        Get
            Return Height
        End Get
        Set
        End Set
    End Property

    Public Property Width As Integer
        Get
            Return Width
        End Get
        Set
        End Set
    End Property

    ' Virtual method
    Public Overridable Sub Draw()
        Console.WriteLine("Performing base class drawing tasks")
    End Sub
End Class

Public Class Circle
    Inherits Shape

    Public Overrides Sub Draw()
        ' Code to draw a circle...
        Console.WriteLine("Drawing a circle")
        MyBase.Draw()
    End Sub
End Class

Public Class Rectangle
    Inherits Shape
```

```
    Public Overrides Sub Draw()
        ' Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle")
        MyBase.Draw()
    End Sub
End Class

Public Class Triangle
    Inherits Shape

    Public Overrides Sub Draw()
        ' Code to draw a triangle...
        Console.WriteLine("Drawing a trangle")
        MyBase.Draw()
    End Sub
End Class

Module Module1

    Sub Main(ByVal args() As String)

        Dim shapes = New List(Of Shape) From
        {
            New Rectangle,
            New Triangle,
            New Circle
        }

        For Each shape In shapes
            shape.Draw()
        Next
        ' Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()

    End Sub

End Module

' Output:
'    Drawing a rectangle
'    Performing base class drawing tasks
'    Drawing a triangle
'    Performing base class drawing tasks
'    Drawing a circle
'    Performing base class drawing tasks
```

Start the debugger!

1. Press **F5** (**Debug > Start Debugging**) or the **Start Debugging** button in the Debug Toolbar.

   F5 starts the app with the debugger attached to the app process, but right now we haven't done anything special to examine the code. So the app just loads and you see the console output.

```
Drawing a rectangle
Performing base class drawing tasks
Drawing a triangle
Performing base class drawing tasks
Drawing a circle
Performing base class drawing tasks
```

   In this tutorial, we'll take a closer look at this app using the debugger and get a look at the debugger features.

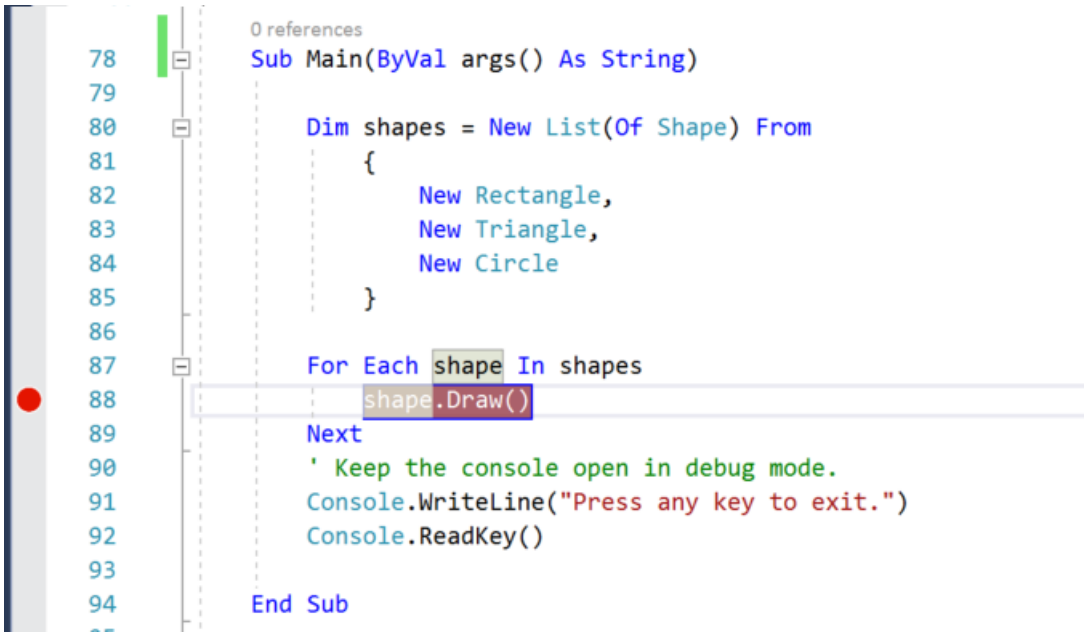
2. Stop the debugger by pressing the red stop ■ button.

Set a breakpoint and start the debugger

1. In the `For Each` loop of the `Main` function, set a breakpoint by clicking the left margin of the following line of code:

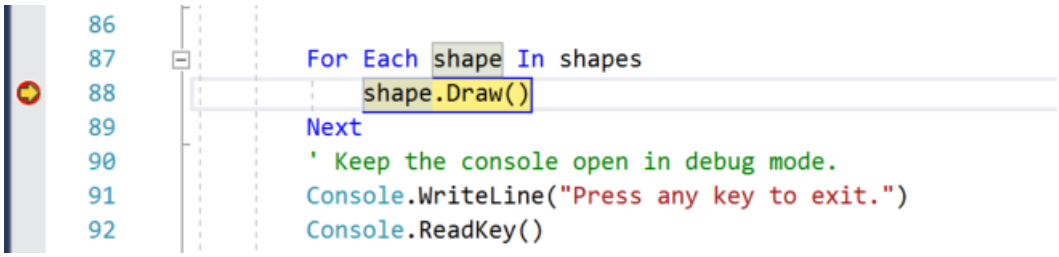
   ```
   shape.Draw()
   ```

   A red circle appears where you set the breakpoint.



   Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code so you can take a look at the values of variables, or the behavior of memory, or whether or not a branch of code is getting run.

2. Press **F5** or the **Start Debugging** button ▶, the app starts, and the debugger runs to the line of code where you set the breakpoint.



   The yellow arrow represents the statement on which the debugger paused, which also suspends app execution at the same point (this statement has not yet executed).

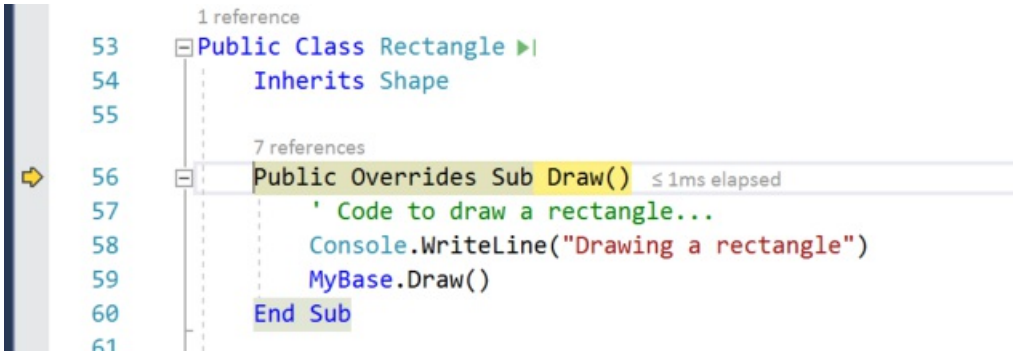
   If the app is not yet running, **F5** starts the debugger and stops at the first breakpoint. Otherwise, **F5** continues running the app to the next breakpoint.

   Breakpoints are a useful feature when you know the line of code or the section of code that you want to examine in detail.

Navigate code in the debugger using step commands

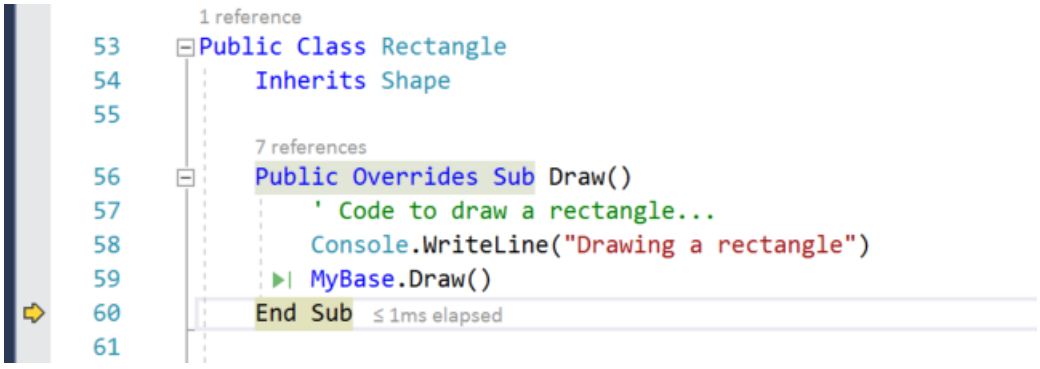
Mostly, we use the keyboard shortcuts here, because it's a good way to get fast at executing your app in the debugger (equivalent commands such as menu commands are shown in parentheses).

1. While paused in the `shape.Draw` method call in the `Main` function, press **F11** (or choose **Debug > Step Into**) to advance into code for the `Rectangle` class.



F11 is the **Step Into** command and advances the app execution one statement at a time. F11 is a good way to examine the execution flow in the most detail. (To move faster through code, we show you some other options also.) By default, the debugger skips over non-user code (if you want more details, see Just My Code).

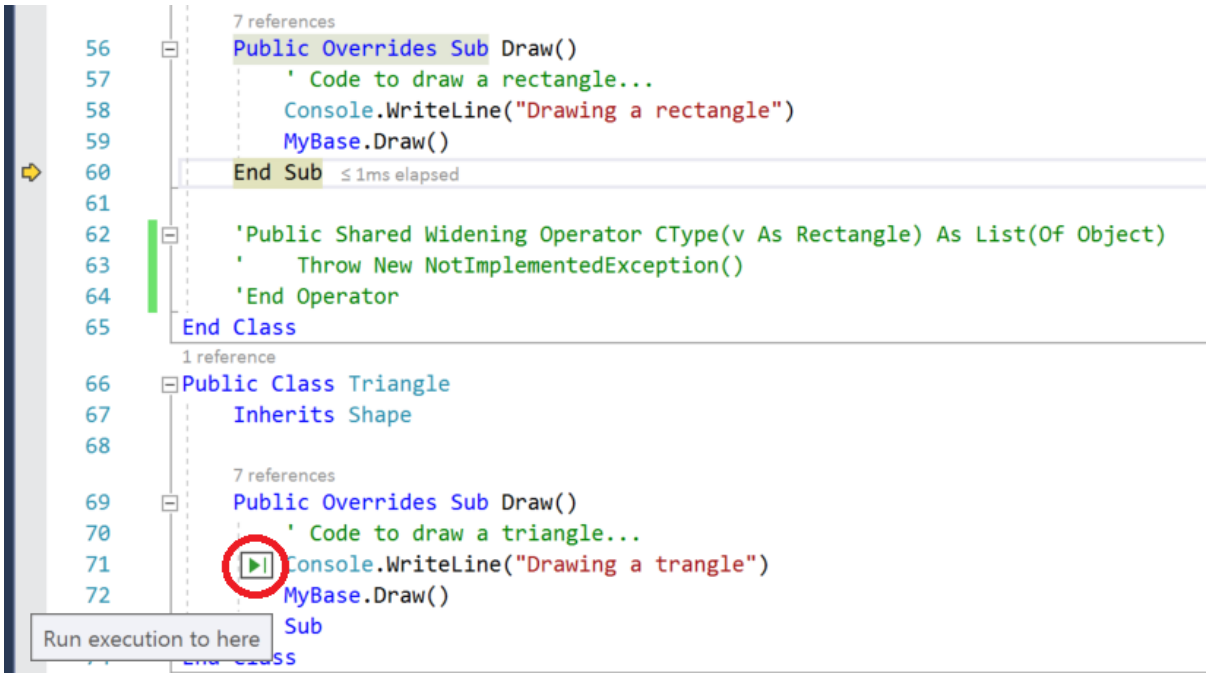
2. Press **F10** (or choose **Debug > Step Over**) a few times until the debugger stops on the `MyBase.Draw` method call, and then press **F10** one more time.



Notice this time that the debugger does not step into the `Draw` method of the base class (`Shape`). **F10** advances the debugger without stepping into functions or methods in your app code (the code still executes). By pressing **F10** on the `MyBase.Draw` method call (instead of **F11**), we skipped over the implementation code for `MyBase.Draw` (which maybe we're not interested in right now).

Navigate code using Run to Click

1. In the code editor, scroll down and hover over the `Console.WriteLine` method in the `Triangle` class until the green **Run to Click** button appears on the left. The tooltip for the button shows "Run execution to here".



> **NOTE**
>
> The **Run to Click** button is new in Visual Studio 2017. If you don't see the green arrow button, use **F11** in this example instead to advance the debugger to the right place.

2. Click the **Run to Click** button .

   Using this button is similar to setting a temporary breakpoint. **Run to Click** is handy for getting around quickly within a visible region of app code (you can click in any open file).

   The debugger advances to the `Console.WriteLine` method implementation for the `Triangle` class.

   While paused, you notice a typo! The output "Drawing a trangle" is misspelled. We can fix it right here while running the app in the debugger.

Edit code and continue debugging

1. Click into "Drawing a trangle" and type a correction, changing "trangle" to "triangle".
2. Press **F11** once and you see that the debugger advances again.

> **NOTE**
>
> Depending on what type of code you edit in the debugger, you may see a warning message. In some scenarios, the code will need to recompile before you can continue.

Step out

Let's say that you are done examining the `Draw` method in the `Triangle` class, and you want to get out of the function but stay in the debugger. You can do this using the **Step Out** command.

1. Press **Shift** + **F11** (or **Debug** > **Step Out**).

   This command resumes app execution (and advances the debugger) until the current function returns.

   You should be back in the `For Each` loop in the `Main` method.

Restart your app quickly

Click the **Restart** button in the Debug Toolbar (**Ctrl** + **Shift** + **F5**).

When you press **Restart**, it saves time versus stopping the app and restarting the debugger. The debugger pauses at the first breakpoint that is hit by executing code.

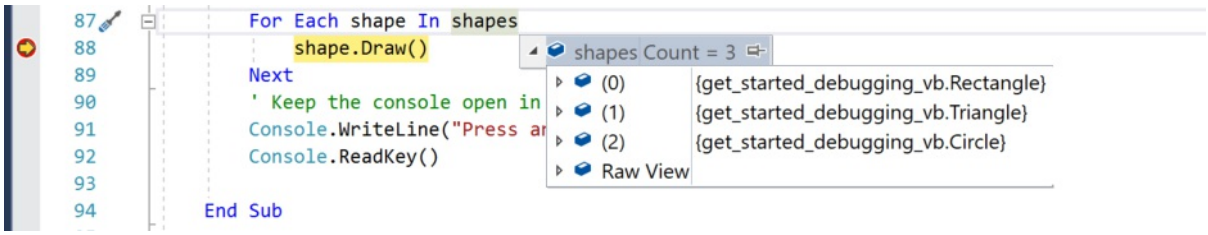
The debugger stops again at the breakpoint you set, on the `shape.Draw()` method.

Inspect variables with data tips

Features that allow you to inspect variables are one of the most useful features of the debugger, and there are different ways to do it. Often, when you try to debug an issue, you are attempting to find out whether variables are storing the values that you expect them to have at a particular time.

1. While paused on the `shape.Draw()` method, hover over the `shapes` object and you see its default property value, the `Count` property.

2. Expand the `shapes` object to see all its properties, such as the first index of the array `[0]`, which has a value of `Rectangle`.

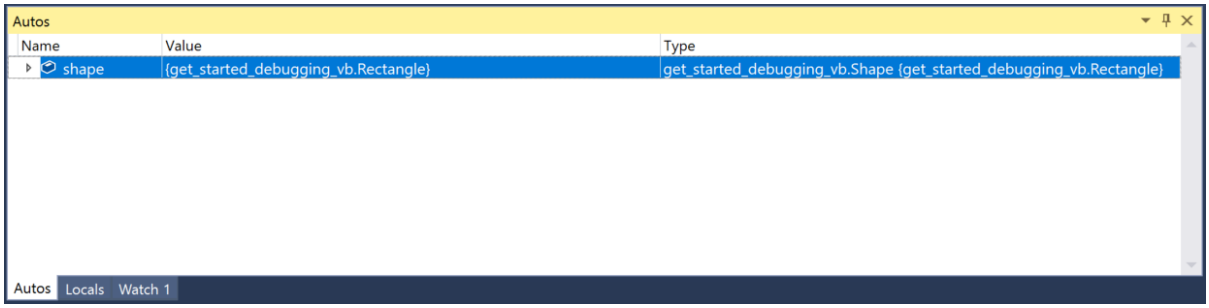


   You can further expand objects to view their properties, such as the `Height` property of the rectangle.

   Often, when debugging, you want a quick way to check property values on objects, and the data tips are a good way to do it.

Inspect variables with the Autos and Locals windows

1. Look at the **Autos** window at the bottom of the code editor.



   In the **Autos** window, you see variables and their current value. The **Autos** window shows all variables used on the current line or the preceding line (Check documentation for language-specific behavior).

2. Next, look at the **Locals** window, in a tab next to the **Autos** window.

   The **Locals** window shows you the variables that are in the current scope, that is, the current execution context.

Set a watch

1. In the main code editor window, right-click the `shapes` object and choose **Add Watch**.

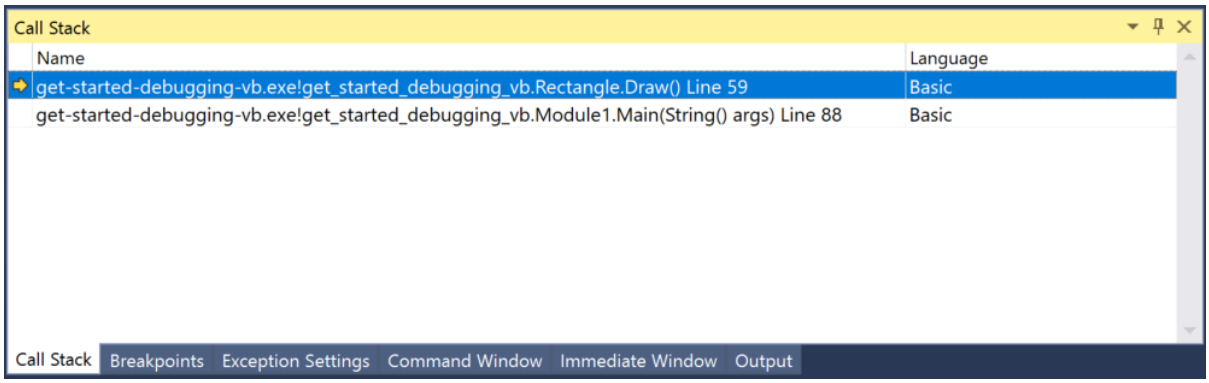
The **Watch** window opens at the bottom of the code editor. You can use a **Watch** window to specify a variable (or an expression) that you want to keep an eye on.

Now, you have a watch set on the `shapes` object, and you can see its value change as you move through the debugger. Unlike the other variable windows, the **Watch** window always shows the variables that you are watching (they're grayed out when out of scope).

Examine the call stack

1. While paused in the `For Each` loop, click the **Call Stack** window, which is by default open in the lower right pane.

2. Click **F11** a few times until you see the debugger pause in the `MyBase.Draw` method of the `Rectangle` class in the code editor. Look at the **Call Stack** window.



The **Call Stack** window shows the order in which methods and functions are getting called. The top line shows the current function (the `Rectangle.Draw` method in this app). The second line shows that `Rectangle.Draw` was called from the `Main` function, and so on.

> **NOTE**
> The **Call Stack** window is similar to the Debug perspective in some IDEs like Eclipse.

The call stack is a good way to examine and understand the execution flow of an app.

You can double-click a line of code to go look at that source code and that also changes the current scope being inspected by the debugger. This action does not advance the debugger.

You can also use right-click menus from the **Call Stack** window to do other things. For example, you can insert breakpoints into specified functions, advance the debugger using **Run to Cursor**, and go examine source code. For more information, see How to: Examine the Call Stack.

Change the execution flow

1. With the debugger paused in the `MyBase.Draw` method call of the `Rectangle` class, use the mouse to grab the yellow arrow (the execution pointer) on the left and move the yellow arrow up one line to the `Console.WriteLine` method call.

2. Press **F11**.

   The debugger reruns the `Console.WriteLine` method (you see duplicate output in the console window output).

   By changing the execution flow, you can do things like test different code execution paths or rerun code without restarting the debugger.

> **WARNING**
>
> Often you need to be careful with this feature, and you see a warning in the tooltip. You may see other warnings, too. Moving the pointer cannot revert your application to an earlier app state.

3. Press **F5** to continue running the app.

   Congratulations on completing this tutorial!

Next steps

In this tutorial, you've learned how to start the debugger, step through code, and inspect variables. You may want to get a high-level look at debugger features along with links to more information.

First look at the debugger

Get started with unit testing

4/18/2019 • 3 minutes to read • Edit Online

Use Visual Studio to define and run unit tests to maintain code health, ensure code coverage, and find errors and faults before your customers do. Run your unit tests frequently to make sure your code is working properly.

Create unit tests

This section describes at a high level how to create a unit test project.

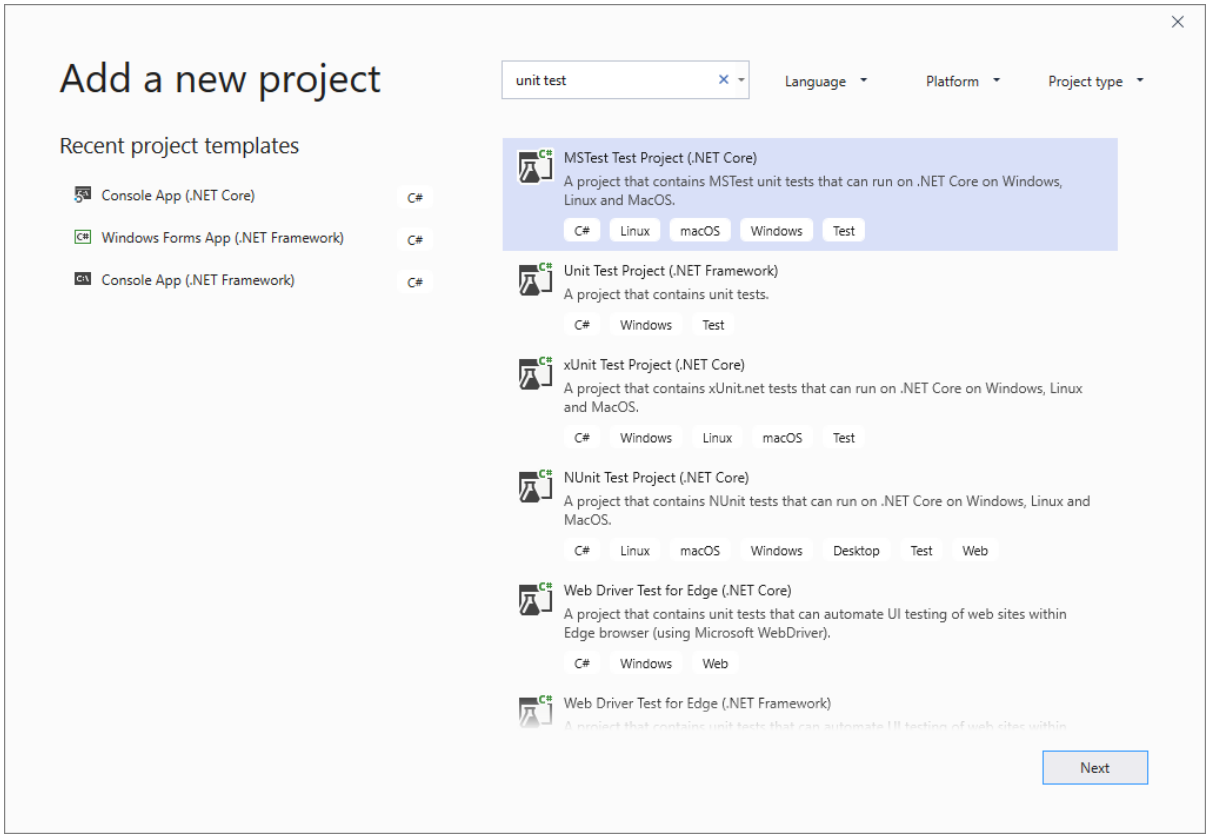
1. Open the project that you want to test in Visual Studio.

   For the purposes of demonstrating an example unit test, this article tests a simple "Hello World" project. The sample code for such a project is as follows:

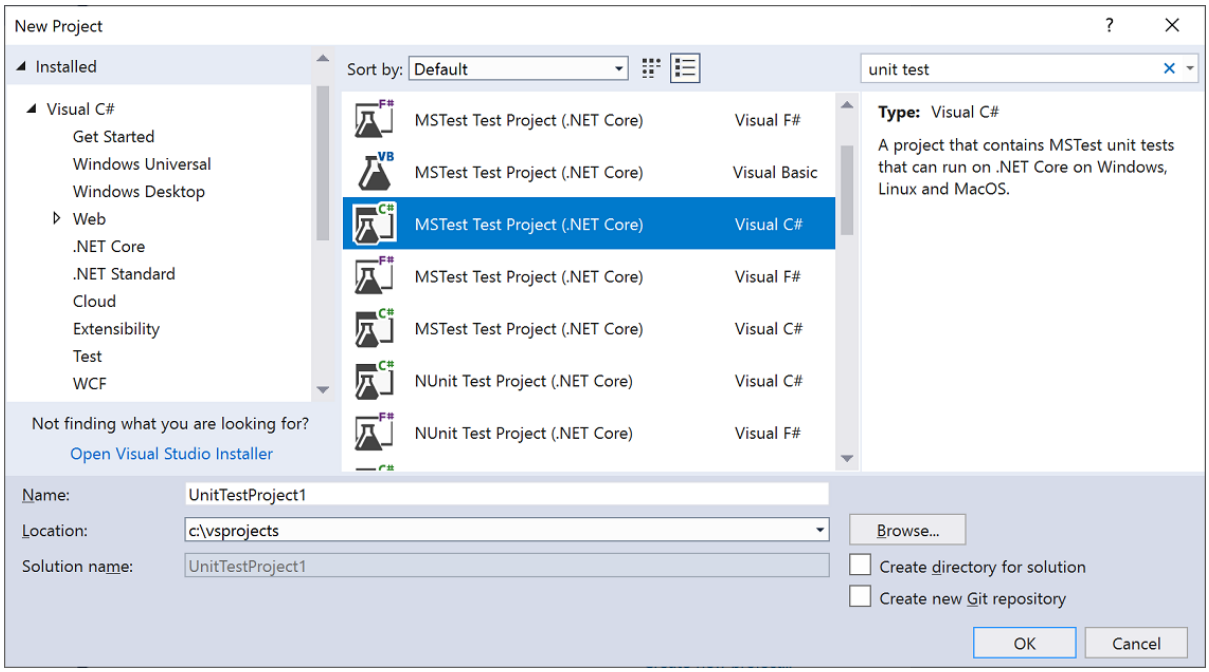
   ```
   public class Program
   {
       public static void Main()
       {
           Console.WriteLine("Hello World!");
       }
   }
   ```

2. In **Solution Explorer**, select the solution node. Then, from the top menu bar, select **File** > **Add** > **New Project**.

3. In the new project dialog box, find a unit test project template for the test framework you want to use and select it.

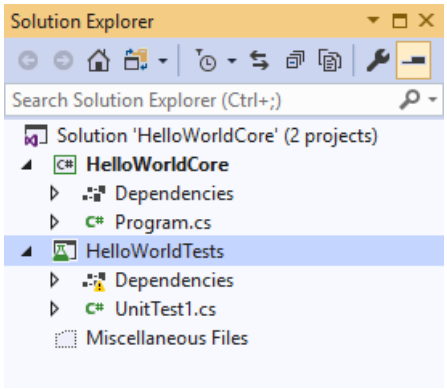


Click **Next**, choose a name for the test project, and then click **Create**.



Choose a name for the test project, and then click **OK**.

The project is added to your solution.



4. In the unit test project, add a reference to the project you want to test by right-clicking on **References** or **Dependencies** and then choosing **Add Reference**.

5. Select the project that contains the code you'll test and click **OK**.

6. Add code to the unit test method.

UnitTest1.cs | Program.cs

HelloWorldTests | HelloWorldTests.UnitTest1

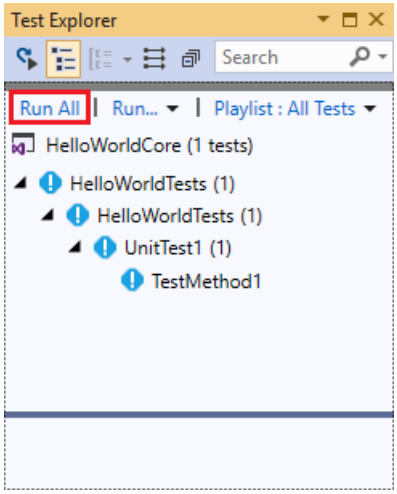
```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  using System;
3  using System.IO;
4
5  namespace HelloWorldTests
6  {
7      [TestClass]
8      public class UnitTest1
9      {
10         private const string Expected = "Hello World!";
11
12         [TestMethod]
13         public void TestMethod1()
14         {
15             using (var sw = new StringWriter())
16             {
17                 Console.SetOut(sw);
18
19                 HelloWorldCore.Program.Main();
20
21                 var result = sw.ToString().Trim();
22                 Assert.AreEqual(Expected, result);
23             }
24         }
25     }
26 }
```

TIP

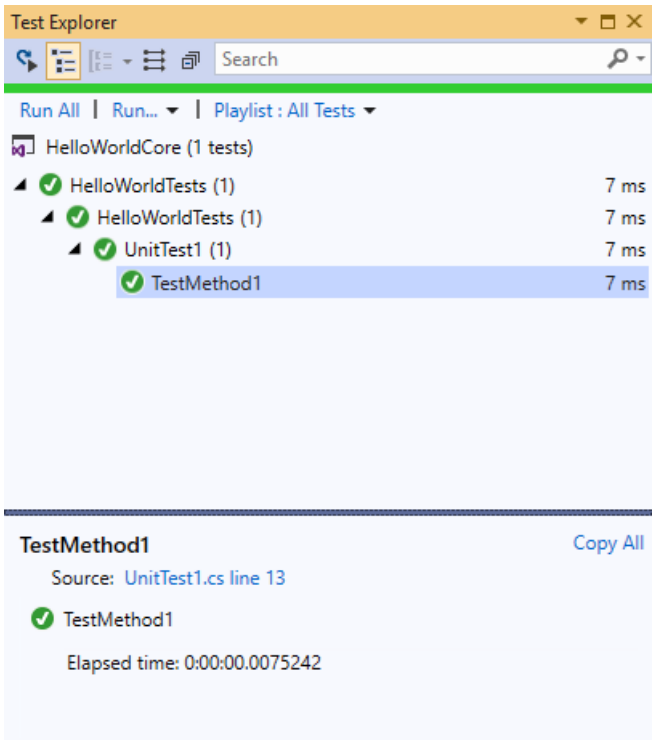
For a more detailed walkthrough of creating unit tests, see Create and run unit tests for managed code.

Run unit tests

1. Open Test Explorer by choosing **Test** > **Windows** > **Test Explorer** from the top menu bar.
2. Run your unit tests by clicking **Run All**.



After the tests have completed, a green check mark indicates that a test passed. A red "x" icon indicates that a test failed.



TIP

You can use Test Explorer to run unit tests from the built-in test framework (MSTest) or from third-party test frameworks. You can group tests into categories, filter the test list, and create, save, and run playlists of tests. You can also debug tests and analyze test performance and code coverage.

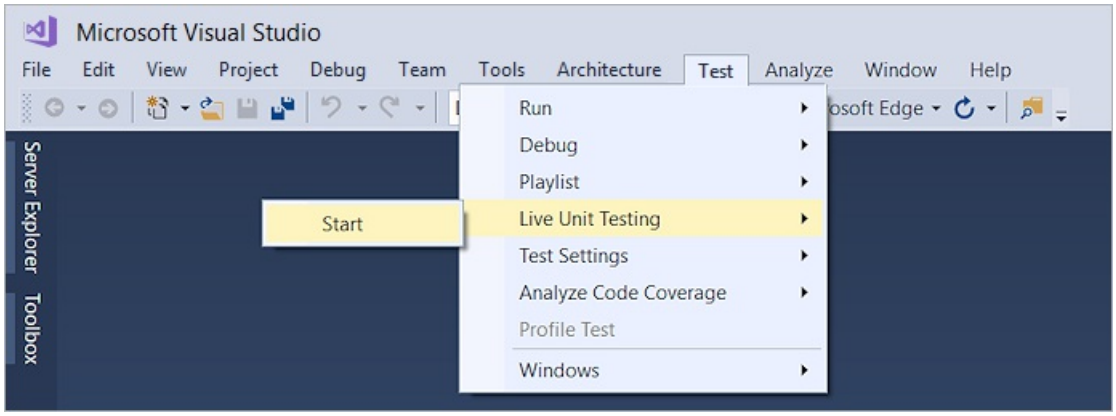
View live unit test results

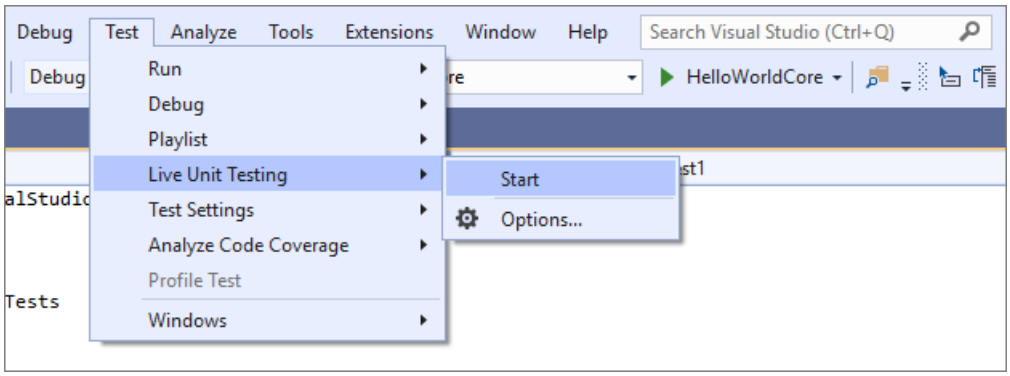
If you are using the MSTest, xUnit, or NUnit testing framework in Visual Studio 2017 or later, you can see live results of your unit tests.

NOTE

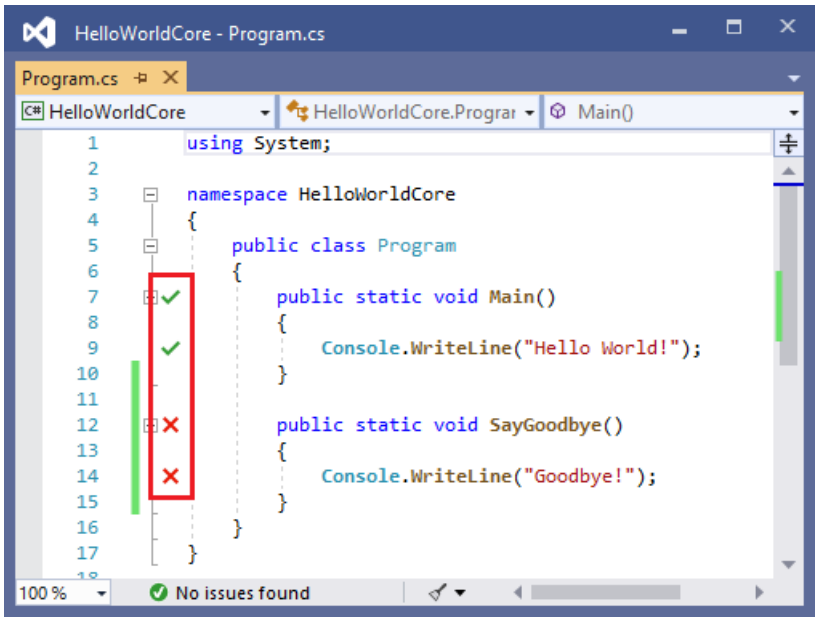
Live unit testing is available in Enterprise edition only.

1. Turn live unit testing from the **Test** menu by choosing **Test** > **Live Unit Testing** > **Start**.





2. View the results of the tests within the code editor window as you write and edit code.



3. Click a test result indicator to see more information, such as the names of the tests that cover that method.

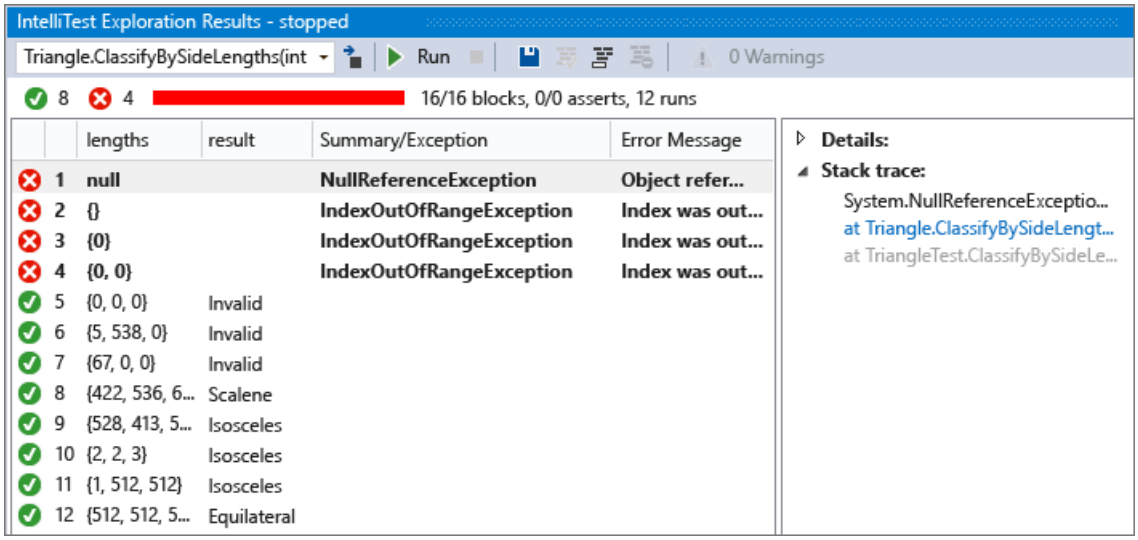
For more information about live unit testing, see Live unit testing.

Generate unit tests with IntelliTest

When you run IntelliTest, you can see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes. To learn how, see Generate unit tests for your code with IntelliTest.

TIP

IntelliTest is only available for managed code that targets the .NET Framework.



Analyze code coverage

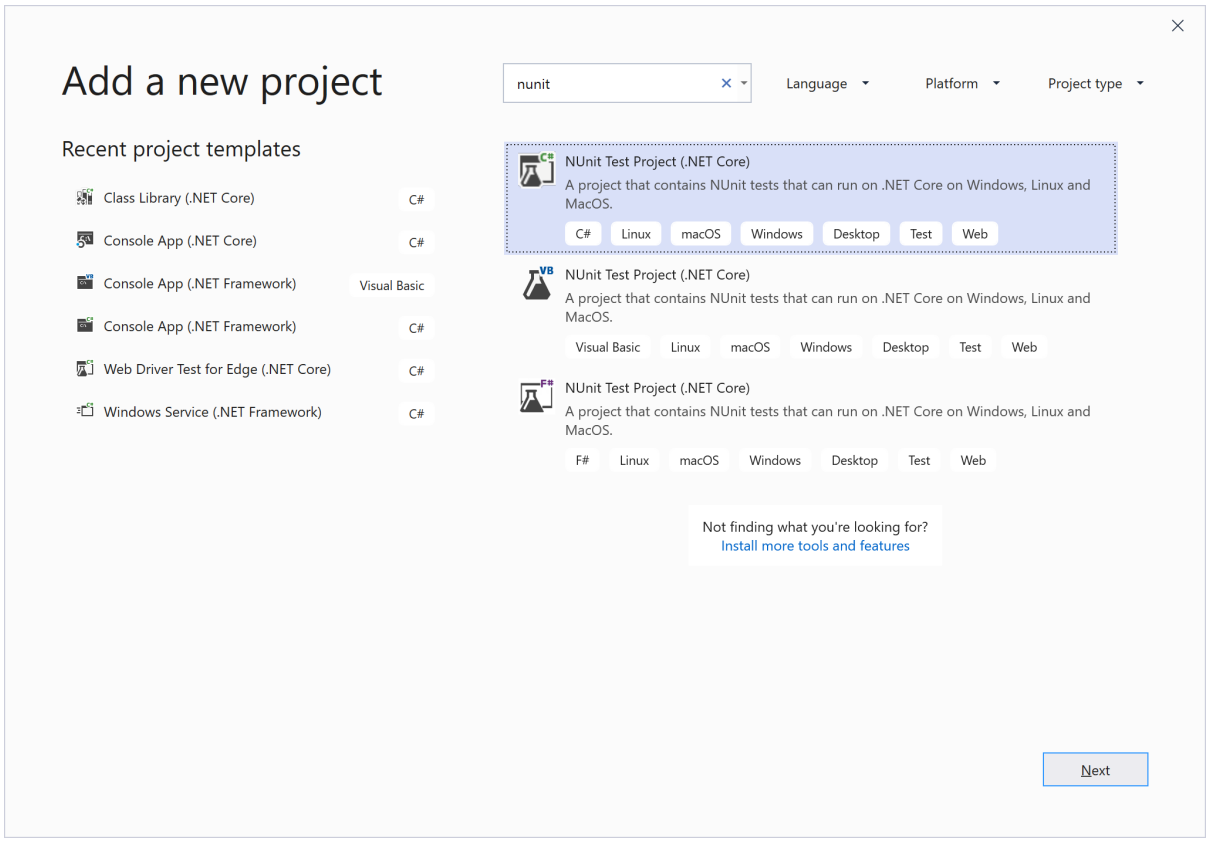
To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise a large proportion of your code. To learn how, see Use code coverage to determine how much code is being tested.

Use a third-party test framework

You can run unit tests in Visual Studio by using third-party test frameworks such as Boost, Google, and NUnit. Use the **NuGet Package Manager** to install the NuGet package for the framework of your choice. Or, for the NUnit and xUnit test frameworks, Visual Studio includes preconfigured test project templates that include the necessary NuGet packages.

To create unit tests that use NUnit:

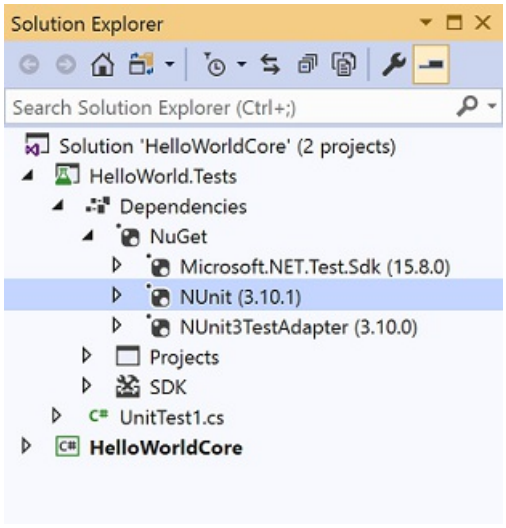
1. Open the solution that contains the code you want to test.
2. Right-click on the solution in **Solution Explorer** and choose **Add** > **New Project**.
3. Select the **NUnit Test Project** project template.



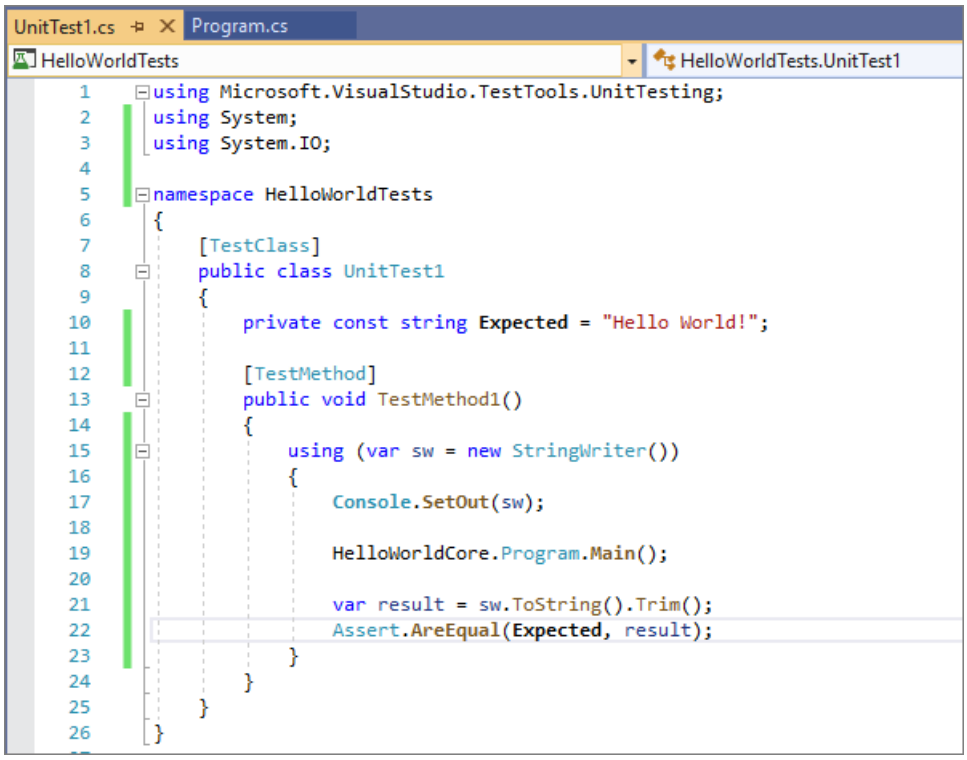
Click **Next**, name the project, and then click **Create**.

Name the project, and then click **OK** to create it.

The project template includes NuGet references to NUnit and NUnit3TestAdapter.



4. Add a reference from the test project to the project that contains the code you want to test.
5. Add code to your test method.



6. Run the test from **Test Explorer** or by right-clicking on the test code and choosing **Run Test(s)**.

See also

- Walkthrough: Create and run unit tests for managed code
- Create Unit Tests command
- Generate tests with IntelliTest
- Run tests with Test Explorer
- Analyze code coverage

Visual Studio provides several different tools to help you deploy your apps.

Experience Visual Studio deployment with 5-minute Quickstarts

First look at deployment options

Deploy to a local folder

Deploy to a website or network share

Deploy to Azure App Service

Deploy to App Service on Linux

Go deeper with tutorials

Deploy ASP.NET to Azure

Import publish settings and deploy ASP.NET to IIS

Deploy a .NET Core app

Package a UWP app for Microsoft Store

Import publish settings and deploy to Azure App Service

Package a desktop app for Microsoft Store (C#, C++)

Deploy Python to Azure

Deploy a desktop app using ClickOnce (C#)

Deploy a C/C++ app

Deploy a C++/CLR app

Create a database and add tables in Visual Studio

6/4/2019 • 4 minutes to read • Edit Online

You can use Visual Studio to create and update a local database file in SQL Server Express LocalDB. You can also create a database by executing Transact-SQL statements in the **SQL Server Object Explorer** tool window in Visual Studio. In this topic, we'll create an *.mdf* file and add tables and keys by using the Table Designer.

Prerequisites

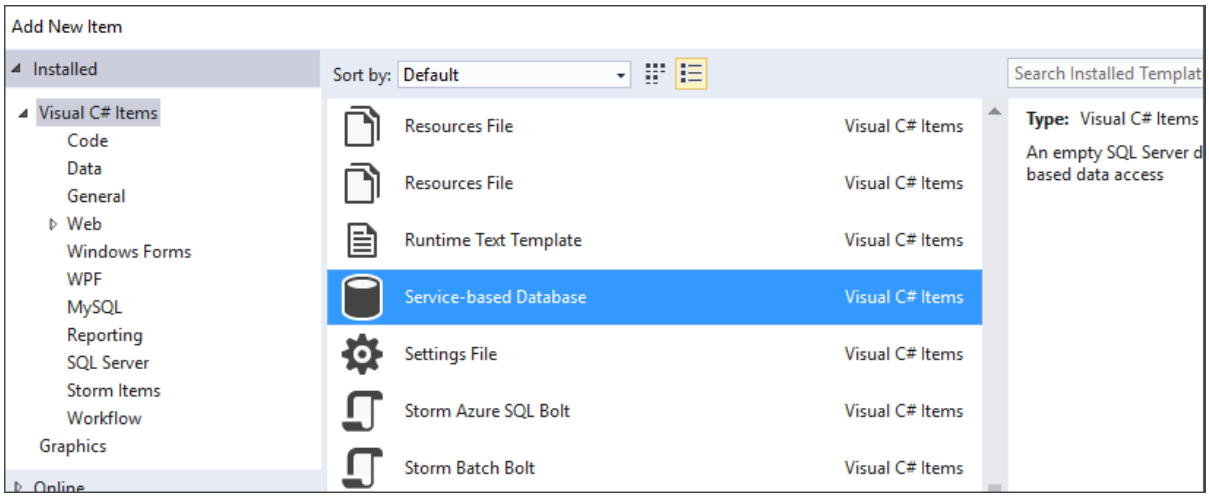
To complete this walkthrough, you must have the optional **Data storage and processing** workload installed in Visual Studio. To install it, open **Visual Studio Installer** and choose **Modify** or **More** > **Modify** next to the version of Visual Studio you want to modify.

On the **Workloads** tab, under **Other Toolsets**, choose **Data storage and processing**, and then click **Modify** to add the workload to Visual Studio.

On the **Workloads** tab, under **Web & Cloud**, choose **Data storage and processing**, and then click **Modify** to add the workload to Visual Studio.

Create a project and a local database file

1. Create a new **Windows Forms App** project and name it **SampleDatabaseWalkthrough**.
2. On the menu bar, select **Project** > **Add New Item**.
3. In the list of item templates, scroll down and select **Service-based Database**.



4. Name the database **SampleDatabase**, and then click **Add**.

Add a data source

1. If the **Data Sources** window isn't open, open it by pressing **Shift**+**Alt**+**D** or selecting **View** > **Other Windows** > **Data Sources** on the menu bar.
2. In the **Data Sources** window, select the **Add New Data Source** link.

   The **Data Source Configuration Wizard** opens.
3. On the **Choose a Data Source Type** page, choose **Database** and then choose **Next**.
4. On the **Choose a Database Model** page, choose **Next** to accept the default (Dataset).

5. On the **Choose Your Data Connection** page, select the **SampleDatabase.mdf** file in the drop-down list, and then choose **Next**.
6. On the **Save the Connection String to the Application Configuration File** page, choose **Next**.
7. One the **Choose your Database Objects** page, you'll see a message that says the database doesn't contain any objects. Choose **Finish**.

View properties of the data connection

You can view the connection string for the *SampleDatabase.mdf* file by opening the properties window of the data connection:

- Select **View** > **SQL Server Object Explorer** to open the **SQL Server Object Explorer** window. Expand **(localdb)\MSSQLLocalDB** > **Databases**, and then right-click on *SampleDatabase.mdf* and select **Properties**.
- Alternatively, you can select **View** > **Server Explorer**, if that window isn't already open. Open the properties window by expanding the **Data Connections** node, opening the shortcut menu for *SampleDatabase.mdf*, and then selecting **Properties**.

Create tables and keys by using Table Designer

In this section, you'll create two tables, a primary key in each table, and a few rows of sample data. You'll also create a foreign key to specify how records in one table correspond to records in the other table.

Create the Customers table

1. In **Server Explorer**, expand the **Data Connections** node, and then expand the **SampleDatabase.mdf** node.
2. Open the shortcut menu for **Tables**, and then select **Add New Table**.

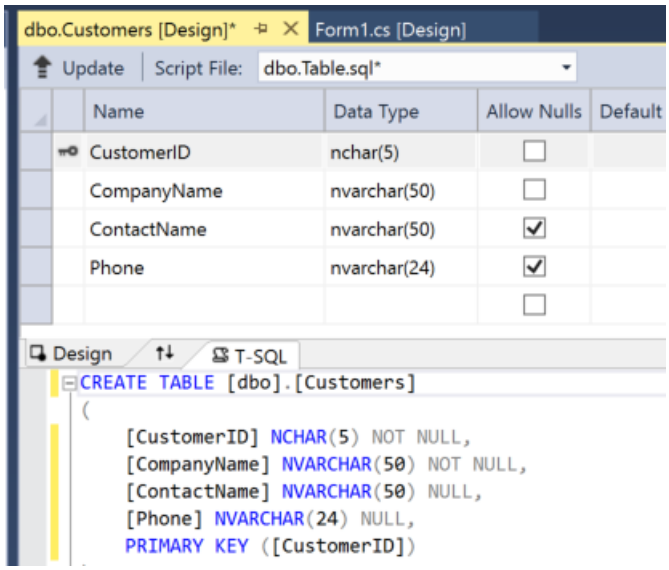
   The **Table Designer** opens and shows a grid with one default row, which represents a single column in the table that you're creating. By adding rows to the grid, you'll add columns in the table.
3. In the grid, add a row for each of the following entries:

COLUMN NAME	DATA TYPE	ALLOW NULLS
`CustomerID`	`nchar(5)`	False (cleared)
`CompanyName`	`nvarchar(50)`	False (cleared)
`ContactName`	`nvarchar (50)`	True (selected)
`Phone`	`nvarchar (24)`	True (selected)

4. Open the shortcut menu for the `CustomerID` row, and then select **Set Primary Key**.
5. Open the shortcut menu for the default row, and then select **Delete**.
6. Name the Customers table by updating the first line in the script pane to match the following sample:

```
CREATE TABLE [dbo].[Customers]
```

   You should see something like this:



7. In the upper-left corner of **Table Designer**, select **Update**.
8. In the **Preview Database Updates** dialog box, select **Update Database**.

   Your changes are saved to the local database file.

Create the Orders table

1. Add another table, and then add a row for each entry in the following table:

COLUMN NAME	DATA TYPE	ALLOW NULLS
`OrderID`	`int`	False (cleared)
`CustomerID`	`nchar(5)`	False (cleared)
`OrderDate`	`datetime`	True (selected)
`OrderQuantity`	`int`	True (selected)

2. Set **OrderID** as the primary key, and then delete the default row.
3. Name the Orders table by updating the first line in the script pane to match the following sample:

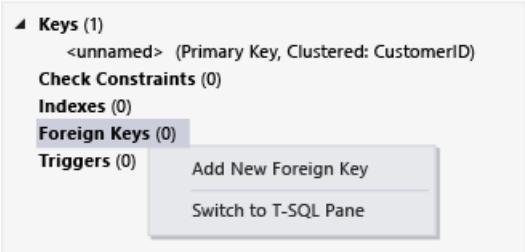
```
CREATE TABLE [dbo].[Orders]
```

4. In the upper-left corner of the **Table Designer**, select the **Update** button.
5. In the **Preview Database Updates** dialog box, select the **Update Database** button.

   Your changes are saved to the local database file.

Create a foreign key

1. In the context pane on the right side of the grid, open the shortcut menu for **Foreign Keys**, and then select **Add New Foreign Key**, as the following illustration shows.



2. In the text box that appears, replace **ToTable** with **Customers**.
3. In the T-SQL pane, update the last line to match the following sample:

```
CONSTRAINT [FK_Orders_Customers] FOREIGN KEY ([CustomerID]) REFERENCES [Customers]([CustomerID])
```

4. In the upper-left corner of the **Table Designer**, select the **Update** button.
5. In the **Preview Database Updates** dialog box, select the **Update Database** button.

   Your changes are saved to the local database file.

Populate the tables with data

1. In **Server Explorer** or **SQL Server Object Explorer**, expand the node for the sample database.
2. Open the shortcut menu for the **Tables** node, select **Refresh**, and then expand the **Tables** node.
3. Open the shortcut menu for the Customers table, and then select **Show Table Data**.
4. Add whatever data you want for some customers.

   You can specify any five characters you want as the customer IDs, but choose at least one that you can remember for use later in this procedure.
5. Open the shortcut menu for the Orders table, and then select **Show Table Data**.
6. Add data for some orders.

   > **IMPORTANT**
   > Make sure that all order IDs and order quantities are integers and that each customer ID matches a value that you specified in the **CustomerID** column of the Customers table.

7. On the menu bar, select **File** > **Save All**.

See also

- Accessing data in Visual Studio