Mike McGrath

C# Programming

modern coding with C# 10 and .NET 6

in easy steps

3rd Edition

Updated for Visual Studio 2022

PLAIN ENGLISH

EASY TO FOLLOW

FULLY ILLUSTRATED

IN FULL COLOR

FOR BEGINNERS AND EXPERIENCED PROGRAMMERS

EXAMPLES AND DEMONSTRATIONS

CREATE YOUR OWN CODING

OBJECT ORIENTED PROGRAMMING

C# Essentials

Common Data Types

int char float double
decimal bool string

Variable Declaration

data-type variable-name **=** *value*

Array Declaration

*data-type***[ ]** *array-name* **=**
  { *value***,** *value***,** *value* **}**

Comments

// *Single-line comment*
/*
  Multi-line comment
*/

Arithmetic Operators

+ *Add (or concatenate strings)*
- *Subtract*
***** *Multiply*
/ *Divide*
% *Modulus*
++ *Increment*
-- *Decrement*

Comparison Operators

== *Equal*
!= *Not equal*
< *Less than*
<= *Less or equal*
> *Greater than*
>= *Greater or equal*

Logical Operators

&& *Logical AND*
|| *Logical OR*
! *Logical NOT*

Assignment Operators

= *Assign*
+= *Add and assign*
-= *Subtract and assign*
***=** *Multiply and assign*
/= *Divide and assign*
%= *Modulus and assign*

Ternary Operator

(*condition* **) ?** *statement* **:** *statement*

If Else

if (*condition* **)**
{ *statements* **}**
else
{ *statements* **}**

Switch

switch (*expression* **)**
{
  case *value* **:** *statements* **; break ;**
  case *value* **:** *statements* **; break ;**
  default : *statements* **; break ;**
}

While Loop

while (*condition* **)**
{ *statements* **}**

Do While Loop

do
{ *statements* **} while (** *condition* **)**

For Loop

for (*initializer* **;** *condition* **;** *updater* **)**
{ *statements* **}**

Foreach Loop

foreach (*data-type variable* **in** *collection* **)**
{ *statements* **}**

Error Handling

try
{ *statements* **}**
catch (Exception *error* **)**
{ *statements* **}**

Method

public void *method-name* **(** *parameters* **)**
{
  statements
}

Class

access-specifier **class** *Name*
{
  member variables
  member methods
}

Mike McGrath

C# Programming

in easy steps

Third Edition

In easy steps is an imprint of In Easy Steps Limited
16 Hamilton Terrace · Holly Walk · Leamington Spa
Warwickshire · United Kingdom · CV32 4LY
www.ineasysteps.com

Third Edition

Printed and bound in the United Kingdom

ISBN 978-1-84078-973-7

Contents

The syntax highlighting of the C# code in this book's easy steps is similar to the default colors in the Visual Studio Code Editor:

Default Code Color: Black	Keywords: Blue	Class Objects: Light Blue
Class Methods: Brown	String Literals: Red	Comments: Green

The source code of all examples in this book is available for free download at www.ineasysteps.com/resource-center/downloads

The screenshots in this book illustrate the actual results of executing the listed code steps. If you don't achieve the result illustrated in any example, simply compare your code to that in the original example files you have downloaded to discover where you went wrong.

1 Getting started

Welcome to the exciting world of C# programming. This chapter introduces the Visual Studio Integrated Development Environment and shows you how to create a real Windows application.

Introducing C#

The introduction of the Microsoft .NET framework at the Professional Developers Conference in July 2000 also saw Microsoft introduce a new programming language called C# (pronounced "see-sharp"). The name was inspired by musical notation where a # sharp symbol indicates that a written note should be a semitone higher in pitch. This notion is similar to the naming of the C++ programming language where the ++ symbol indicates that a written value should be incremented by 1.

- C# is designed to be a simple, modern, general-purpose, object-oriented programming language, borrowing key concepts from several other languages – most notably the Java programming language. Consequently, everything in C# is a class "object" with "properties" and "methods" that can be employed by a program.

- C# is an elegant and "type-safe" programming language that enables developers to build a variety of secure and robust applications. You can use C# to create Windows client applications, XML web services, distributed components, client-server applications, database applications, and much, much more.

- C# is specifically designed to utilize the proven functionality built into the .NET framework "class libraries". Windows applications written in C# therefore require the Microsoft .NET framework to be installed on the computer running the application – typically, an integral component of the system.

The Microsoft .NET Framework

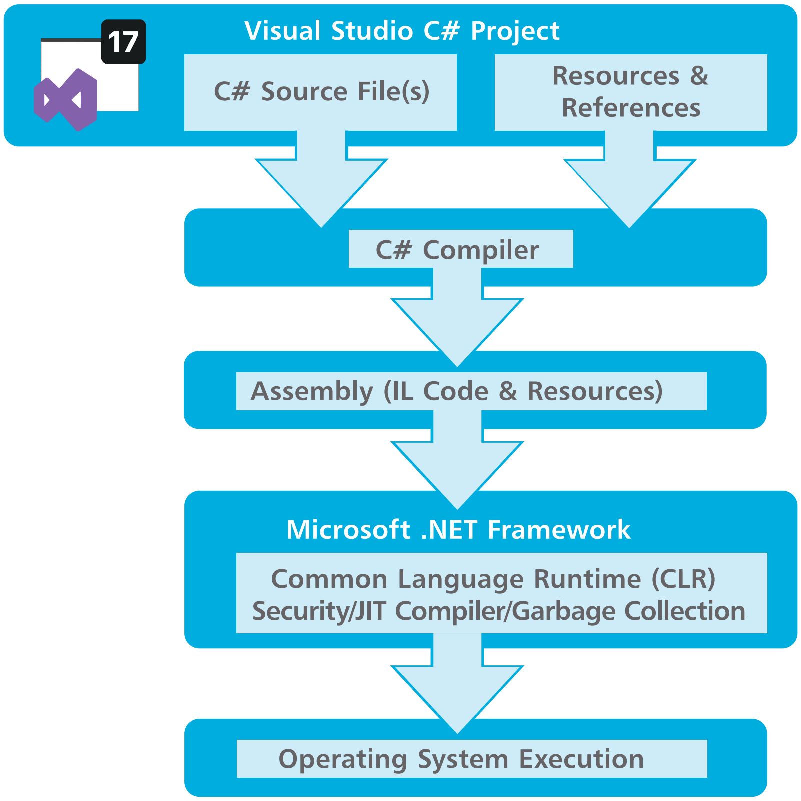
Each version of the Microsoft .NET framework includes a unified set of class libraries and a virtual execution system called the Common Language Runtime (CLR). The CLR allows the C# language and the class libraries to work together seamlessly.

To create an executable program, source code written in the C# language is compiled by the C# Compiler into Intermediate Language (IL) code. This is stored on disk, together with other program resources such as images, in an "assembly". Typically, the assembly will have a file extension of **.exe** or **.dll**. Each assembly contains a "manifest" that provides information about that program's security requirements.

When a C# program is executed, the assembly is loaded into the Common Language Runtime (CLR), and the security requirements specified in its assembly manifest are examined. When the security requirements are satisfied, the CLR performs Just-In-Time (JIT) compilation of the IL code into native machine instructions. The CLR then performs "garbage collection", exception handling, and resource management tasks before calling upon the operating system to execute the program:

Hot tip

Just-In-Time compilation is also known as "Dynamic Translation".


Visual Studio C# Project
17
C# Source File(s)
Resources & References
C# Compiler
Assembly (IL Code & Resources)
Microsoft .NET Framework
Common Language Runtime (CLR)
Security/JIT Compiler/Garbage Collection
Operating System Execution

Don't forget

Just-In-Time compilation occurs during program execution, rather than prior to its execution.

As language interoperability is a key feature of the Microsoft .NET framework, the IL code generated by the C# Compiler can interact with code generated by the .NET versions of other languages such as Visual Basic and Visual C++. The examples throughout this book demonstrate Visual C# program code.

Installing Visual Studio

In order to create Windows applications with the C# programming language, you will first need to install a Visual Studio Integrated Development Environment (IDE).

Microsoft Visual Studio is the professional development tool that provides a fully Integrated Development Environment for Visual Basic, Visual C++, Visual J#, and Visual C#. Within its IDE, code can be written in Visual Basic, C++, J#, or the C# programming language to create Windows applications.

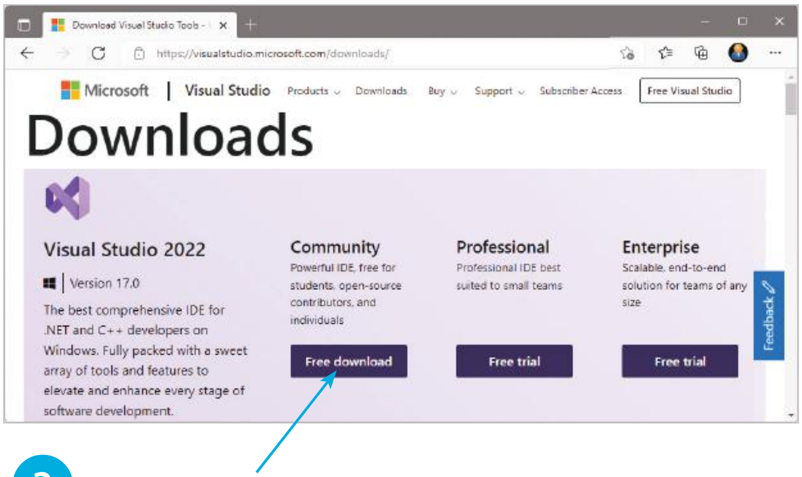
Visual Studio Community edition is a streamlined version of Visual Studio, specially created for those people learning programming. It has a simplified user interface and omits advanced features of the professional edition to avoid confusion. C# code can be written within the **Code Editor** of either version of the Visual Studio IDE to create Windows applications.

Both Visual Studio and Visual Studio Community provide an IDE for C# programming but, unlike the fully-featured Visual Studio product, the Visual Studio Community edition is completely free and can be installed on any system meeting the following minimum requirements:

Component	Requirement
Operating system	Windows 11 Windows 10 (version 1909 or higher) Windows Server 2016 or 2019 *Must be the 64-bit version of any of the above the operating systems.
CPU (processor)	1.8 GHz or faster, 64-bit processor
RAM (memory)	4 GB (16 GB recommended)
HDD (hard drive)	Up to 210 GB available space
Video Card	Minimum resolution of 1366 x 768 Optimum resolution of 1920 x 1080

The Visual Studio Community edition is used throughout this book to demonstrate programming with the C# language, but the examples can also be recreated in Visual Studio. Follow the steps opposite to install the Visual Studio Community edition.

1 Open your web browser and navigate to the Visual Studio download page – at the time of writing, this can be found at **visualstudio.microsoft.com/downloads**

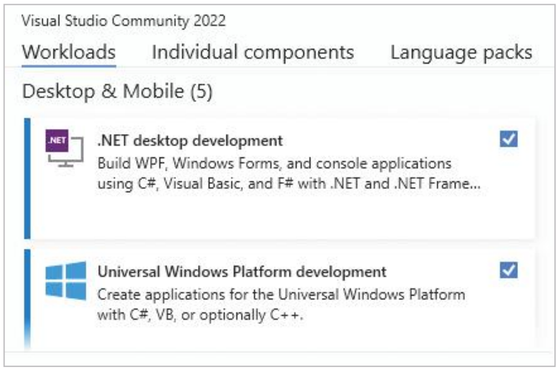

Download Visual Studio Tools -
https://visualstudio.microsoft.com/downloads/
Microsoft | Visual Studio Products Downloads Buy Support Subscriber Access Free Visual Studio
Downloads
Visual Studio 2022
Version 17.0
The best comprehensive IDE for .NET and C++ developers on Windows. Fully packed with a sweet array of tools and features to elevate and enhance every stage of software development.
Community
Powerful IDE, free for students, open-source contributors, and individuals
Free download
Professional
Professional IDE best suited to small teams
Free trial
Enterprise
Scalable, end-to-end solution for teams of any size
Free trial
Feedback

2 Click the button in the Community edition section to download a **VisualStudioSetup.exe** setup file

3 Click on the setup file icon to begin setup and to run the **Visual Studio Installer**

4 Accept the suggested installation location, then click **Next**

5 Check the two C# **Installer** options shown below


Visual Studio Community 2022
Workloads Individual components Language packs
Desktop & Mobile (5)
.NET desktop development ✓
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...
Universal Windows Platform development ✓
Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

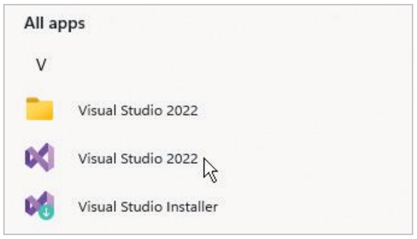
6 Click the **Install** button at the bottom-right of the installer to begin the download and installation process

Choosing a different destination folder may require other paths to be adjusted later – it's simpler to just accept the suggested default.

Both **Visual Studio** and **Visual Studio Installer** items get added to your **All Apps** menu. You can re-run the installer at a later date to add or remove features.

Exploring the IDE

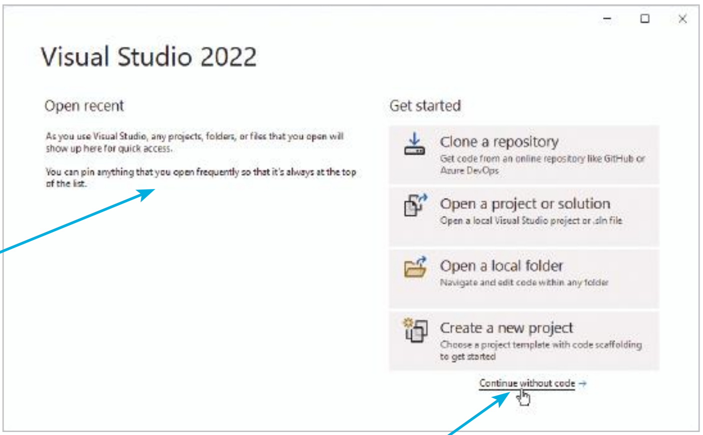
1. Go to your **All apps** menu, then select the Visual Studio 2022 menu item added there by the installer:



The first time Visual Studio starts it takes a few minutes as it performs configuration routines.

2. Sign in with your Microsoft account, or register an account then sign in, to continue

3. See a default **Start Page** appear where recent projects will be listed alongside several “Get started” options



Visual Studio 2022

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

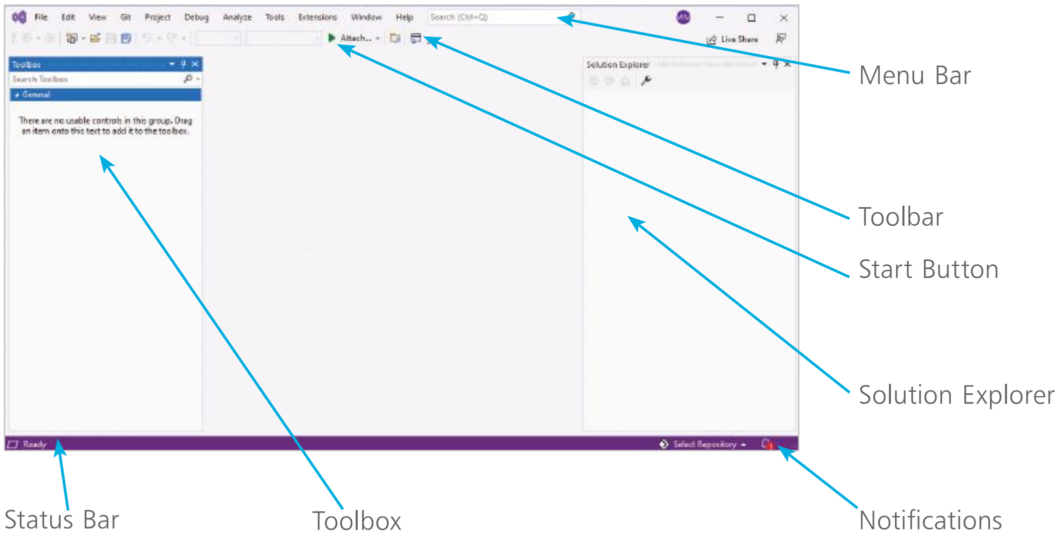
Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

In the future your recent projects will be listed here so you can easily reopen them.

4. For now, just click the **Continue without code** link to launch the Visual Studio application

The Visual Studio Integrated Development Environment (IDE) appears, from which you have instant access to everything needed to produce complete Windows applications – from here, you can create exciting visual interfaces, enter code, compile and execute applications, debug errors, and much more.



Visual Studio IDE components

The Visual Studio IDE initially provides these standard features:

- **Menu Bar** – Where you can select actions to perform on all your project files and to access Help. When a project is open, extra menus of Project and Build are shown in addition to the default menu selection of File, Edit, View, Git, Project, Debug, Analyze, Tools, Extensions, Window, and Help.
- **Toolbar** – Where you can perform the most popular menu actions with a single click on their associated shortcut icons.
- **Toolbox** – Where you can select visual elements to add to a project. Click View, Toolbox or a side bar button to see its contents. When a project is open, "controls" such as Button, Label, CheckBox, RadioButton, and TextBox are shown here.
- **Solution Explorer** – Where you can see at a glance all the files and resource components contained within an open project.
- **Status Bar** – Where you can read the state of the current activity being undertaken. When building an application, a "Build started" message is displayed here, changing to a "Build succeeded" or "Build failed" message upon completion.

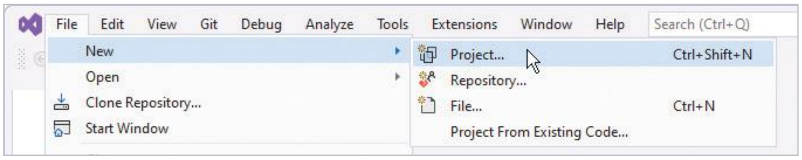
Hot tip

The IDE may have a **Light** color theme by default. To change the color theme, choose the **Tools**, **Options** menu then select **Environment**, **General**, **Color Theme** and select **Blue** or **Dark** theme, or select the **Use system setting** option.

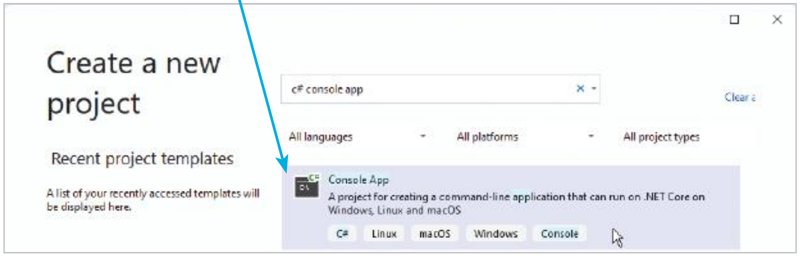
Starting a Console project

GettingStarted

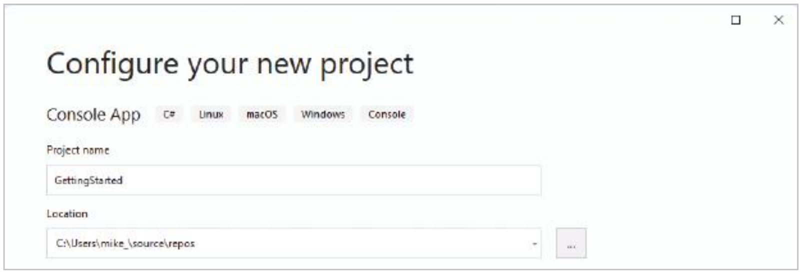
1. On the Menu Bar, click **File**, **New**, **Project...** or press **Ctrl** + **Shift** + **N**, to open the "Create a new project" dialog



2. In the "Create a new project" dialog box, select the **C# Console App** item (for .NET Core on Windows, Linux and macOS), then click **Next**


Create a new project
c# console app
Clear all
All languages
All platforms
All project types
Recent project templates
A list of your recently accessed templates will be displayed here.
Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C#
Linux
macOS
Windows
Console

3. In the next dialog, enter a project name plus location and click the **Create** button, then select the .NET 6.0 (Long-term support) framework and click **Create** again


Configure your new project

Console App C# Linux macOS Windows Console

Project name: GettingStarted

Location: C:\Users\mike_\source\repos

Visual Studio now creates your new project and loads it into the IDE. A **Code Editor** window appears, containing default skeleton project code generated by Visual Studio.

4. Drag the **Code Editor** window tab to undock the **Code Editor** window from the Visual Studio IDE frame

Hot tip

The source code of all examples in this book is available for free download at **www.ineasysteps.com/resource-center/downloads**

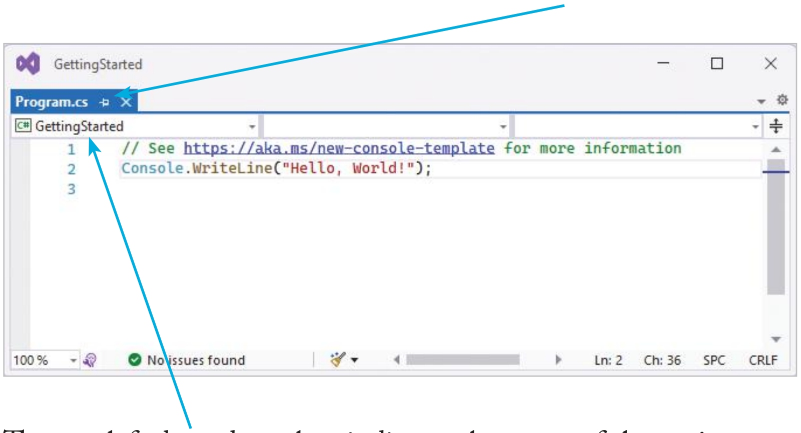
Hot tip

The default location for Visual Studio projects is a **C:\Users\*username*\source\repos** directory.

Hot tip

If the **Code Editor** window does not open automatically, click the **Program.cs** file icon in Solution Explorer to open the **Code Editor**.

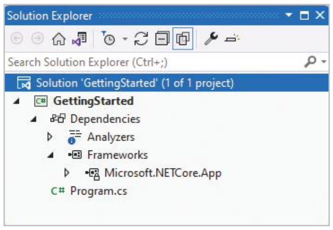
The undocked window title displays the project name, and the tab displays the file name of the code as "Program.cs".



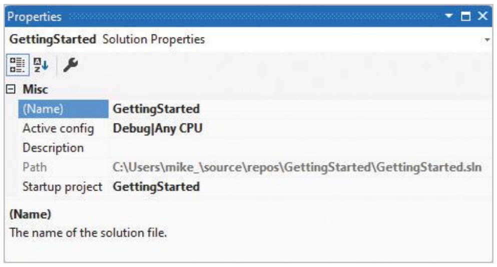
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

The top-left drop-down box indicates the name of the project to which this file belongs – in this case, it's "GettingStarted".

5. Select the **View**, **Solution Explorer** menu to open a **Solution Explorer** window, to discover all the items in your project – click the arrow buttons to expand or collapse categories



6. Select the **View**, **Properties Window** menu to open a **Properties** window, then select any item in the **Solution Explorer** window to see its properties then appear in the **Properties** window

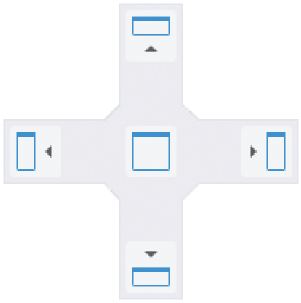


(Name)	GettingStarted
Active config	Debug|Any CPU
Description	
Path	C:\Users\mike_\source\repos\GettingStarted\GettingStarted.sln
Startup project	GettingStarted

(Name)
The name of the solution file.

The **Code Editor** window is where you write C# code to create an application. The Visual Studio IDE has now gathered all the resources needed to build a default Console application.

You can drag the title bar of any window to undock that window from the Visual Studio IDE frame. When dragging, you can drop a window on the "guide diamond" (shown below) to dock the window in your preferred position.



Running a Console project

GettingStarted
(continued)

In order to run a program, Visual Studio will first build the app then execute the app. This can be done in one of two modes:

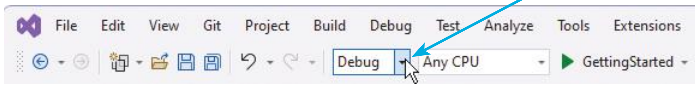
- **Debug Mode** – The program is compiled with symbolic debugging information included in the program files. This allows Visual Studio's built-in debugger to find bugs, but has optimization of Intermediate Language (IL) code disabled.
- **Release Mode** – The program is compiled without debugging information included in the program files, but has optimization of Intermediate Language (IL) code enabled.

During program development it is generally preferable to run your programs in Debug mode. There is an option to run a program in Debug mode without debugging, but it's seldom desirable as this executes the program without the possibility of stepping through the code to breakpoints.

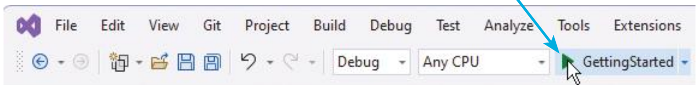
Hot tip

You will discover how to use breakpoints in Chapter 8 (Solving problems).

1. On the toolbar, set the solution configuration to **Debug**

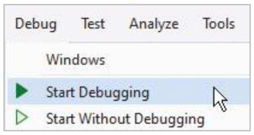


2. Now, click the green arrow "**Start**" button to build and run the program with debugging enabled

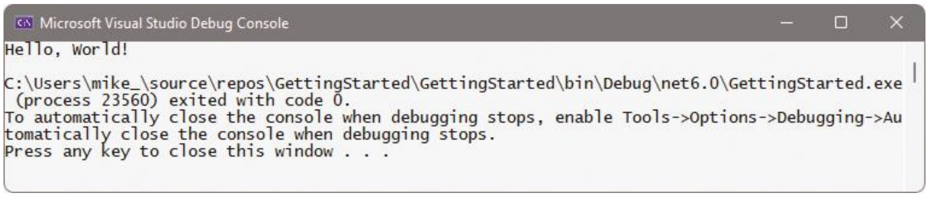


Hot tip

Alternatively, select **Debug**, **Start Debugging** to build and run a program with debugging enabled.



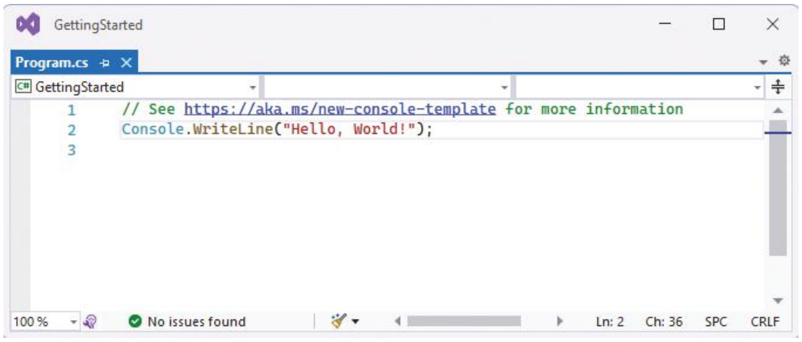
3. See the app display a traditional greeting in the Console



Microsoft Visual Studio Debug Console

```
Hello, World!

C:\Users\mike_\source\repos\GettingStarted\GettingStarted\bin\Debug\net6.0\GettingStarted.exe
 (process 23560) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Au
tomatically close the console when debugging stops.
Press any key to close this window . . .
```

4. Press any keyboard key to close the Console window, and return to the Visual Studio **Code Editor**



```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

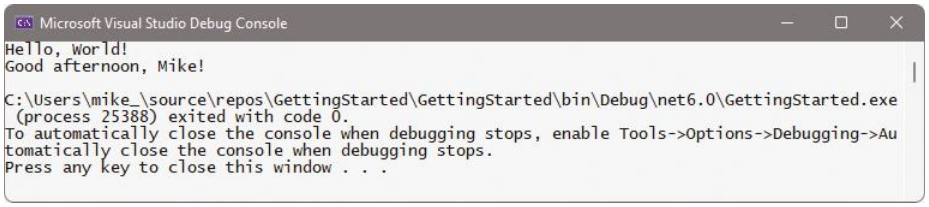
Code analysis

Examination of the code helps to understand what is happening:

- The first line is a comment. Anything on a line after // is ignored by the compiler. By default, comments are colored green in the Visual Studio **Code Editor**. In this case, the comment includes a hyperlink that will open a page in your web browser. The page explains how the .NET 6 framework uses new templates to simplify the creation of Console apps.
- **Console.WriteLine("Hello, World!") ;** This is a statement that calls upon the **WriteLine( )** method of the **Console** class to output the text string enclosed in quote marks within its parentheses. Notice that the statement is terminated by a ; semi-colon character. By default, class names are colored light blue, method names are colored brown, strings are colored red, and other code here is colored black.

5. Add another line to the code, as a statement to output a second text string containing your own name such as... **Console.WriteLine("Good afternoon, Mike!") ;**

6. Run the modified code in Debug mode to see the result



```
Hello, World!
Good afternoon, Mike!

C:\Users\mike_\source\repos\GettingStarted\GettingStarted\bin\Debug\net6.0\GettingStarted.exe
 (process 25388) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Au
tomatically close the console when debugging stops.
Press any key to close this window . . .
```

Hot tip

New in .NET 6 are code templates, which the compiler uses to automatically generate namespace, class, and method elements when building a Console app.

Hot tip

To edit the default Console window colors and font, right-click its window Titlebar and choose **Properties**. For clarity, all other Console window screenshots in this book feature Lucida Console 14-pixel **Font** in black **Screen Text** on a white **Screen Background**.

Following the rules

As with all programming languages, C# has a number of syntax rules that must be precisely followed to ensure the code is correctly formatted for the C# compiler to clearly understand:

- **Case-sensitivity** – C# is a case-sensitive language, which means that uppercase "A" and lowercase "a" are regarded as totally different items.
- **Termination** – All statements in C# language must be terminated by a **;** semicolon character, just as all sentences in English language must be terminated by a . period character – for example: **Console.WriteLine("Hello World!") ;**
- **Single-line comments** – Brief comments on a single line must begin with // two forward slash characters – for example: **// Output the traditional greeting.**
- **Block comments** – Extended comments on multiple lines must begin with /* forward slash and asterisk characters, and must end with the reverse */ asterisk and forward slash – for example:

```
/*
  C# Programming in easy steps.
  Getting started with the traditional greeting.
*/
```

- **White space** – Spaces, tabs, newline characters, and comments are ignored by the C# compiler, so can be used extensively to organize code without performance penalty.
- **Escape sequences** – The C# compiler recognizes **\n** as a newline character and **\t** as a tab character, so these can be used to format output – for example: **Console.WriteLine("Line One \n Line Two") ;**
- **Naming conventions** – A programmer-defined identifier name in C# code may begin with an _ underscore character or a letter in uppercase or lowercase. The name may also contain underscore, letter, and number characters – for example: **class MyNo1_Class**
- **Keywords** – The C# language has a number of keywords (listed opposite) that have special syntactic meaning and may not be used to name programmer-defined items in code.

Hot tip

It is recommended that you comment your code to make it readily understood by others or when revisiting your own code later.

Don't forget

The **WriteLine( )** method automatically adds a newline after its output.

…cont'd

C# Reserved Keywords

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort
using	virtual	void	volatile
while			

If you absolutely must use a keyword to name a programmer-defined element, it may be prefixed by an **@** character to distinguish it from the keyword – but this is best avoided.

C# Contextual Keywords

add	and	alias	ascending	async
await	by	descending	dynamic	equals
from	get	global	group	init
into	join	let	managed	nameof
nint	not	notnull	nuint	on
or	orderby	partial	record	remove
select	set	unmanaged	value	var
when	where	with	yield	

Contextual keywords have special significance in certain code – for example, **get** and **set** in method declarations.

Summary

- **C#** is an object-oriented programming language that utilizes the proven functionality of the Microsoft **.NET** class libraries.
- The C# compiler generates **Intermediate Language (IL)** code that is stored on disk alongside resources in an assembly.
- The **Common Language Runtime (CLR)** examines an assembly's security requirements before JIT compilation.
- **Just-In-Time** compilation translates IL code into native machine code for execution by the operating system.
- Microsoft Visual Studio provides a fully **Integrated Development Environment (IDE)** for C# programming.
- A new Visual C# **Console** application generates default skeleton project code in the Visual Studio **Code Editor**.
- The Visual Studio **Solution Explorer** shows all files in a project, and the **Properties** window shows their properties.
- The default C# skeleton code provided in the **Code Editor** can create a C# Console program that outputs a text string.
- The **Console** class has a **WriteLine( )** method that can be used to output a quoted text string specified within its parentheses.
- A C# program can be run in the Visual Studio IDE by selecting the **Debug**, **Start Debugging** menu, or by setting a **Debug** solution configuration and clicking the **Start** button.
- C# is a case-sensitive programming language in which all statements must be terminated by a **;** semicolon character.
- Single-line **//** comments and **/* */** block comments can be incorporated to explain C# program code.
- C# has keywords that have special syntactic meaning, so cannot be used to name programmer-defined code items.

2 Storing values

This chapter demonstrates how to store various types of data within a C# program.

Creating variables

A "variable" is like a container in a C# program in which a data value can be stored inside the computer's memory. The stored value can be referenced using the variable's name.

The programmer can choose any name for a variable, providing it adheres to the C# naming conventions – a chosen name may only contain letters, numbers, and the underscore character, but must begin with a letter, underscore, or @ character. Also, the C# keywords must be avoided. It's good practice to choose meaningful names to make the code more comprehensible.

To create a new variable in a program it must be "declared", specifying the type of data it may contain and its chosen name. A variable declaration has this syntax:

data-type variable-name **;**

Multiple variables of the same data type can be created in a single declaration as a comma-separated list with this syntax:

data-type variable-name1 **,** ***variable-name2*** **,** ***variable-name3*** **;**

The most common C# data types are listed in the table below, together with a brief description and example content:

Data Type	Description
int	An integer whole number, e.g. **100**
char	A single character, e.g. **'A'**
float	A floating-point number of 7-digit precision
double	A floating-point number of 15-digit precision
decimal	A floating-point number of 28-digit precision
bool	A Boolean value of **true** or **false**
string	A string of characters, e.g. **"In Easy Steps"**

Variable declarations must appear before statements that use them – so they will be available for reference within statements.

Beware

Names are case-sensitive in C# – so variables named **num**, **Num**, and **NUM** are treated as three individual variables. Traditionally, C# variable names are created in all lowercase characters.

Beware

Character values of the **char** data type must be enclosed in single quotes, but character strings of the **string** data type must be enclosed between double quotes.

Hot tip

The **decimal** data type is preferred for storage of monetary values.

When a value is assigned to a variable it is said to have been "initialized". Optionally, a variable may be initialized in its declaration. The value stored in any initialized variable can be displayed using the **WriteLine( )** method, which was used on page 16 to display the "Hello World!" greeting:

1. Open the Visual Studio IDE, then start a new **C# Console App** project and name it "Variables"

2. Remove the default skeleton code, then type this code to name the Console window

```
Console.Title = "Variables" ;
```

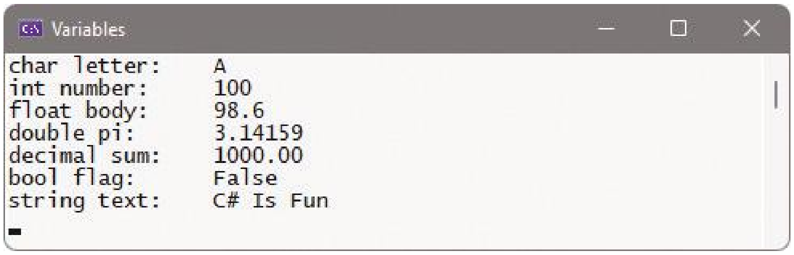
3. Next, precisely type these statements to declare and initialize variables of common C# data types

```
char letter ;       letter = 'A' ;     // Declared then initialized.
int number ;        number = 100 ;     // Declared then initialized.
float body = 98.6f ;                   // Declared and initialized.
double pi = 3.14159 ;                  // Declared and initialized.
decimal sum = 1000.00m ;               // Declared and initialized.
bool flag = false ;                    // Declared and initialized.
string text = "C# Is Fun" ;            // Declared and initialized.
```

4. Now, insert statements to display each stored value

```
Console.WriteLine( "char letter:\t" + letter ) ;
Console.WriteLine( "int number:\t" + number ) ;
Console.WriteLine( "float body:\t" + body ) ;
Console.WriteLine( "double pi:\t" + pi ) ;
Console.WriteLine( "decimal sum:\t" + sum ) ;
Console.WriteLine( "bool flag:\t" + flag ) ;
Console.WriteLine( "string text:\t" + text ) ;
Console.ReadKey( ) ;
```

5. Press **Start** to run the application in Debug mode and see the values stored inside variables



```
char letter:    A
int number:     100
float body:     98.6
double pi:      3.14159
decimal sum:    1000.00
bool flag:      False
string text:   C# Is Fun
```

Variables

Don't forget

Suffix **f** to a **float** value and **m** to a **decimal** value to distinguish them from a **double** value.

Hot tip

The **\t** in the string is an "escape sequence" that prints a tab in the output. The **+** symbol is a "concatenation" operator, which adds a stored value to the string.

Don't forget

Calling the **Readkey( )** method is a little trick to keep the Console window open when the app is published, until the user presses any key.

Reading input

In order to interact with the user, C# programs will typically require the user to input some values. The **ReadLine( )** method – a companion to the **ReadKey( )** method – can be used to read user input. User input within the Console can be assigned to a **string** data type variable by the **ReadLine( )** method when the user hits **Enter** to complete a line.

When requesting user input, it's preferable to prompt the user without adding a newline after the request. This is easily achieved using the **Console.Write( )** method:

Input

1. Open the Visual Studio IDE, then start a new **C# Console App** project and name it "Input"

2. Remove the default skeleton code, then type this code to name the Console window

```
Console.Title = "Input" ;
```

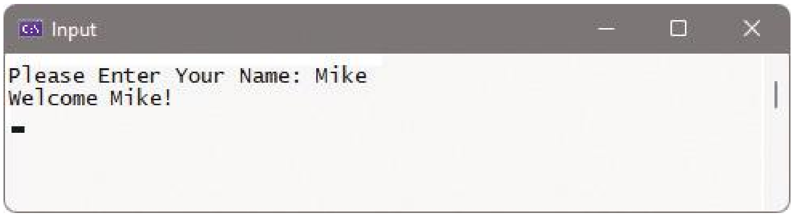
3. Next, precisely type these statements to request user input for assignment to a variable

```
Console.Write( "Please Enter Your Name: " ) ;
string name = Console.ReadLine( ) ;
```

4. Now, add statements to display a message containing the stored user input value

```
Console.WriteLine( "Welcome " + name + "!" ) ;
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application and see the stored user input value displayed in output



```
Please Enter Your Name: Mike
Welcome Mike!
_
```

Numerical values input by the user are assigned to the **string** variable as characters – so arithmetic cannot be performed on these.

Beware

It is important to recognize that the **ReadLine( )** method always reads input as a **string** value.

Hot tip

Notice how two **+** concatenation operators are used here to insert input into a string.

When you require the user to input numerical values it is necessary to convert the **string** values read by the **ReadLine( )** method into numerical data types. The **System.Convert** class provides a number of useful methods for this purpose, including:

Method	Returns
Convert.ToInt32( )	A 32-bit signed integer
Convert.ToDouble( )	A floating-point precision number
Convert.ToDecimal( )	A decimal precision number

The **string** value read by the **ReadLine( )** method needs simply to be specified within the parentheses of the appropriate method:

1. Start a new **C# Console App** and name it "Conversion"

```
Console.Title = "Conversion" ;
```

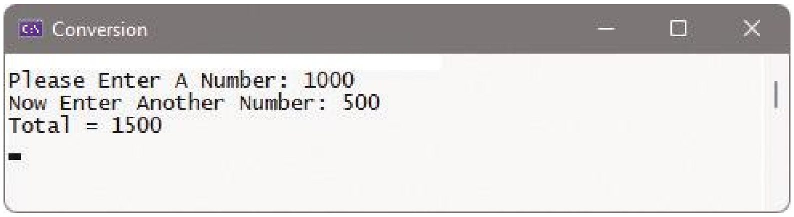
2. Type these statements to request user input for conversion and assignment to two variables

```
Console.Write( "Please Enter A Number: " ) ;
double num = Convert.ToDouble( Console.ReadLine( ) ) ;
Console.Write( "Now Enter Another Number: " ) ;
double sum=
        num + Convert.ToDouble( Console.ReadLine( ) ) ;
```

3. Now, add statements to display a message containing the sum total of the user input values

```
Console.WriteLine( "Total = " + sum ) ;
Console.ReadKey( ) ;
```

4. Press **Start** or **F5** to run the application, then enter two values to see the converted input added in output



```
Please Enter A Number: 1000
Now Enter Another Number: 500
Total = 1500
```

Hot tip

There is also a useful **Convert.ToString( )** method to translate values to **string** type.

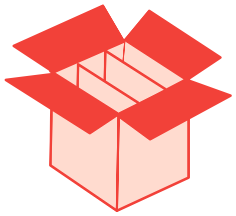
C#

Conversion

Don't forget

Here, the second input number gets converted to a numerical value before addition to the first input number.

Employing arrays



An array is a variable that can store multiple items of data – unlike a regular variable, which can only store one piece of data. The pieces of data are stored sequentially in array "elements" that are numbered, starting at zero. So, the first value is stored in element 0, the second value is stored in element 1, and so on.

An array is declared by stating the data type, suffixed by **[ ]** square brackets to denote an array. This alone does not initialize the array variable in memory until an array "instance" is created by specifying an element size using the **new** keyword, like this:

***data-type* [ ] *array-name* = new *data-type*[*size*] ;**

Values can then be individually assigned to each available element:

***array-name*[*element-number*] = *value* ;**

Alternatively, an array can be initialized and values assigned to each element when it is declared by stating values for each element in a comma-separated list, grouped within braces:

***data-type* [ ] *array-name* =**
new *data-type*[*size*] { *value* , *value* , *value* } ;

Any individual element's value can be referenced using the array name followed by square brackets containing the element number – for example, to reference the value within the first element:

***array-name*[0]**

Collectively, the elements of an array are known as an "index". Arrays can have more than one index – to represent multiple dimensions, rather than the single dimension of a regular array. Multi-dimensional arrays of three indices and more are uncommon, but two-dimensional arrays are useful to store grid-based information such as coordinates.

To create a multi-dimensional array, the **[ ]** square brackets following the data type in the declaration must contain a comma for each additional index. Similarly, the size of each index must be specified as a comma-separated list, like this:

***data-type* [,] *array-name* = new *data-type*[*size* , *size*] ;**

Values can then be individually assigned to each available element:

***array-name*[*element-number* , *element-number*] = *value* ;**

Don't forget

Array numbering starts at zero – so the final element in an array of six elements is number 5, not number 6. This numbering is known as a "zero-based index".

Beware

You can only store within array elements data of the type specified in the array declaration.

Hot tip

A special **foreach** loop construct allows you to easily iterate through all elements of an array – see page 56 for details.

Alternatively, a multi-dimensional array can be initialized when it is declared by stating values for each index in a comma-separated list, grouped within braces – in a comma-separated group enclosed within braces:

```
data-type [ , ] array-name = new data-type[ sizeX , sizeY ]
        { { value , value , value } , { value , value , value } } ;
```

Any individual element's value can be referenced using the array name followed by square brackets containing the element number for each index – for example, to reference the value within the first element of the second index:

```
array-name[ 1 , 0 ]
```

Beware

The number of specified values must exactly match the specified array size to avoid an error.

1. Start a new **C# Console App** and name it "Arrays"
```
Console.Title = "Arrays" ;
```

2. Next, type this statement to create a string array
```
string [ ] cars = new string[ 3 ] { "BMW", "Ford", "Opel" } ;
```

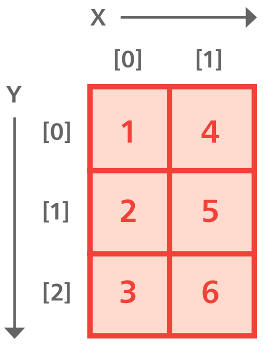
3. Now, type this statement to create a 2-dimensional array
```
int [ , ] coords = new int[ 2, 3 ] {  { 1, 2, 3 } , { 4, 5, 6 }  } ;
```

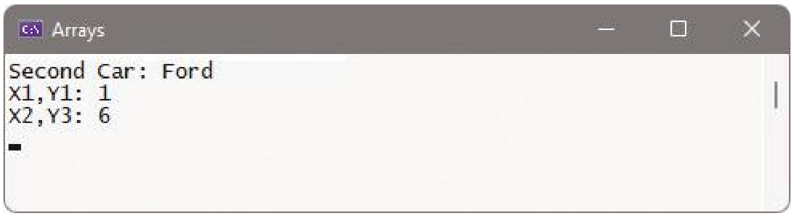
4. Add statements to display output containing the stored array element values
```
Console.WriteLine( "Second Car: " + cars[ 1 ] ) ;
Console.WriteLine( "X1,Y1: " + coords[ 0, 0 ] ) ;
Console.WriteLine( "X2,Y3: " + coords[ 1, 2 ] ) ;
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application and see the values stored within array elements

C#

Arrays





```
Second Car: Ford
X1,Y1: 1
X2,Y3: 6
```

Casting data types

A value stored in a variable can be forced (coerced) into a variable of a different data type by converting the value to a different type. This process is known as "casting", and may be implicit or explicit.

- **Implicit** casting is automatically performed by C# in a type-safe manner when converting numbers from smaller to larger precision data types. For example, when adding an **int** to a **double**, the **int** value is automatically converted to a **double**.
- **Explicit** casting, on the other hand, requires a cast operator to be included in a statement to specify the data type to which the value should be cast. The required new data type must be specified in parentheses preceding the name of the variable containing the data to be cast, so its syntax looks like this:

Hot tip

Casting is also known as "type conversion".

```
variable-name = ( data-type ) variable-name ;
```

This is the traditional form of casting that is also found in the C programming language. Casting is often necessary to accurately store the result of an arithmetic operation to prevent data loss. Division of one integer by another integer will always produce an integer result, which may be truncated. For example, the integer division **7/2** produces the truncated integer result of **3**.

To store the accurate floating-point result would require the result be cast into a suitable data type, such as a **double**, like this:

```
double sum = ( double ) 7 / 2 ;          // sum = 3.5
```

Don't forget

The result of dividing an integer by another integer is truncated, not rounded – so a result of 9.9 would become 9.

It should be noted that operator precedence casts the first operand **7** into the specified data type before implementing the arithmetic operation **/2** division, so effectively the statement is:

```
double sum = ( double ) ( 7 ) / 2 ;      // sum = 3.5
```

Bracketing the expression as **(7 / 2)** would perform the arithmetic first on integers, so the integer result would be truncated before being cast into the **float** variable – not the desired effect!

```
double sum = ( double ) ( 7 / 2 ) ;      // sum = 3.0
```

Single character values can usefully be explicitly cast to an **int** data type to reveal their Unicode code value. Similarly, the process can be reversed to reveal the character value of an integer.

1. Start a new **C# Console App** and name it "Cast"
```
Console.Title = "Cast" ;
```

2. Type these statements to create and initialize a double-precision floating-point variable and an integer variable
```
double num = 10.5 ;
int integer = 2 ;
```

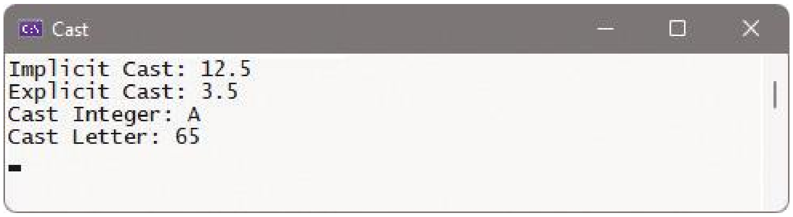
3. Next, add statements to implicitly cast the integer value into a double-precision value, and display the result
```
num = num + integer ;
Console.WriteLine( "Implicit Cast: " + num ) ;
```

4. Now, explicitly cast the result of an integer division into a double-precision value, and display that result
```
num = ( double ) 7 / integer ;
Console.WriteLine( "Explicit Cast: " + num ) ;
```

5. Cast an integer value into a **char** data type and display its Unicode equivalent character
```
char letter = ( char ) 65 ;
Console.WriteLine( "Cast Integer: " + letter ) ;
```

6. Cast a character value into an **int** data type and display its Unicode equivalent code number
```
int unicode = ( int ) 'A' ;
Console.WriteLine( "Cast Letter: " + unicode ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the values cast into other data types


```
Implicit Cast: 12.5
Explicit Cast: 3.5
Cast Integer: A
Cast Letter: 65
```

Cast

Hot tip

Remove the **(double)** cast from Step 4 to see the result become truncated.

Fixing constants

Variable declarations explicitly specify their permissible data type using keywords such as **string**, **int**, or **double** to ensure the program cannot accidentally assign inappropriate data. Where you are certain this will never occur, you may use the **var** (variant) keyword when declaring a variable, to implicitly specify its data type according to the data type of its initial value. In this case, the C# compiler will automatically determine the appropriate data type. The **var** keyword is most useful to store a value that is other than of the standard C# numerical, Boolean, or string data types. All variable declarations made using the **var** keyword must assign an initial value to immediately determine that variable's data type.

Beware

Declaring variables using the **var** keyword can make the program code less readable, as it does not indicate data types.

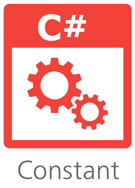
The data type of any variable can be revealed by dot-suffixing the **GetType( )** method onto the variable name. For example, where a **num** variable is a **double** data type, calling **num.GetType( )** will return a **System.Double** result.

When a stored value is never intended to be changed, a "constant" container can be created to safeguard it from change. Its declaration is similar to that of a variable, but begins with the **const** keyword. Each declaration made using the **const** keyword must assign an initial value to immediately fix its constant value.

Multiple constant values can be defined in an enumerator list using the **enum** keyword and a specified identifier name. This creates a data type that consists of a comma-separated list of named constants within **{ }** braces. The enumerator list has an underlying default **int** value that numbers the list names from zero, much like the elements in an array. Any name within the list can be referenced by dot-suffixing it to the list name, and its underlying index number revealed by casting.

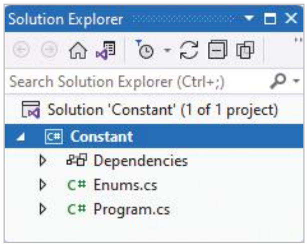
The C# **Enum** class provides several methods to work with enumerator lists. Many of these require the data type of the list as an argument, so this can usefully be assigned to a **var** variable by specifying the list name as an argument to the **typeof( )** method. The **Enum** class **GetName( )** method can reveal the name at a specified index position, and the **IsDefined( )** method can be used to test whether the list contains a specified name. An enumerator list declaration can be written in its own code file, or after all other "top-level" statements in the main "Program.cs" code file.

1. Start a new **C# Console App** and name it "Constant"
```
Console.Title = "Constant" ;
```



2. In Solution Explorer, right-click on the "Constant" project icon and choose **Add**, **New Item** from the menus

3. Select the **Code File** option, then name it "Enums.cs" and click the **Add** button – to add the file to this project

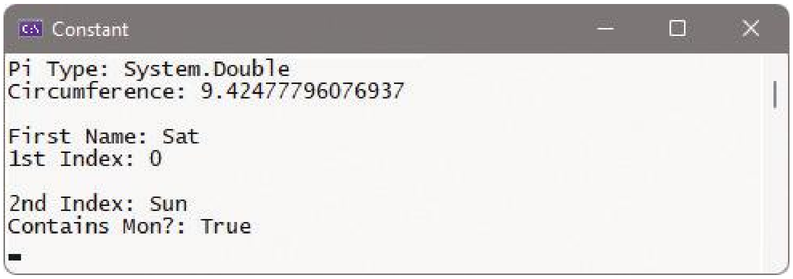

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Constant' (1 of 1 project)
Constant
Dependencies
Enums.cs
Program.cs

4. In the **Code Editor**, type this statement within the newly added file – to create an enumerator list
```
enum Days { Sat, Sun, Mon, Tue, Wed, Thu, Fri } ;
```

5. Now, turn your attention to the main "Program.cs" file, then declare and initialize a constant and three variables
```
const double pi = 3.14159265358979 ;
var daysType = typeof( Days ) ;
string name = Enum.GetName( daysType , 1 ) ;
bool flag = Enum.IsDefined( daysType , "Mon" ) ;
```

6. Next, add statements to reveal the constant's data type and details of the first two items in the enumerator list
```
Console.WriteLine( "Pi Type: " + pi.GetType( ) ) ;
Console.WriteLine( "Circumference: " + ( pi * 3 ) ) ;
Console.WriteLine( "\nFirst Name: " + Days.Sat ) ;
Console.WriteLine( "1st Index: " + ( int ) Days.Sat ) ;
Console.WriteLine( "\n2nd Index: " + name ) ;
Console.WriteLine( "Contains Mon?: " + flag ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the constant values in operation



```
Pi Type: System.Double
Circumference: 9.42477796076937

First Name: Sat
1st Index: 0

2nd Index: Sun
Contains Mon?: True
```

Don't forget

The enumerator list contains names, not strings, so they need not be enclosed in quote marks, but the name must be supplied as a string argument to the **IsDefined( )** method.

Hot tip

You may want to avoid the **var** keyword where possible to be sure of variable data types. For example, with the line **var id = getID( ) ;** the assigned value (and therefore the variable data type) could be numeric or a string.

Summary

- A variable stores a data value within the computer's memory, and the value can be referenced using that variable's name.
- A variable declaration must specify the type of data it may contain, and a valid programmer-specified name.
- Common C# data types include **int**, **char**, **float**, **double**, **decimal**, **bool**, and **string**.
- Variables must be declared before they can be referenced.
- A variable becomes initialized when it is assigned a value.
- The **Console.ReadLine( )** method can be used to assign user input to a variable.
- The **Console.WriteLine( )** method adds a newline after its output, but the **Console.Write( )** method does not.
- The **System.Convert** class provides a number of useful methods to convert string values to numerical data types.
- An array variable can store multiple items of data within sequential elements of a zero-based index.
- An array declaration must specify the type of data it may contain, followed by **[ ]** and a programmer-specified name.
- An array is not initialized until an array instance is created using the **new** keyword to specify data type and element size.
- Values can be assigned to individual array elements using the array name and the element index number.
- An array declaration can assign values to all of its array elements as a comma-separated list of values within **{ }** braces.
- Arrays can have more than one index, to represent multiple dimensions where each index is separated by a comma.
- Casting is the conversion of one data type to another.
- Implicit casting is performed automatically by C#.
- Explicit casting is performed by including a cast operator within a statement, to specify the required new data type.
- When a stored value is never intended to be changed, a "constant" container can be created using the **const** keyword.
- An enumerator list declaration is created using the **enum** keyword.

3 Performing operations

This chapter introduces C# operators and demonstrates the operations they can perform.

Doing arithmetic

The arithmetic operators commonly used in C# programs are listed in the table below, together with the operation they perform:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

Hot tip

The **%** modulus operator is also known as the "remainder" operator.

Don't forget

Values used with operators to form expressions are called "operands" – in the expression **2 + 3** the numerical values **2** and **3** are the operands.

Hot tip

The **+** operator is dual-purpose – it is also used to concatenate strings.

The operators for assignment, addition, subtraction, multiplication, and division act as you would expect, but care must be taken to group expressions where more than one operator is used – operations within innermost **( )** parentheses are performed first:

```
a = b * c - d % e / f ;                 // This is unclear.

a = ( b * c ) - ( ( d % e ) / f ) ;     // This is clearer.
```

The **%** modulus operator will divide the first given number by the second given number and return the remainder of the operation. This is useful to determine if a number has an odd or an even value.

The **++** increment operator and **--** decrement operator alter the given number by 1 and return the resulting value. These are most commonly used to count iterations in a loop – counting up on each iteration with the **++** increment operator, or counting down on each iteration with the **--** decrement operator.

The **++** increment and **--** decrement operators can be placed before or after a value to different effect – placed before the operand (prefix), its value is immediately changed; placed after the operand (postfix), its value is noted first, then the value is changed.

Arithmetic

1. Start a new **C# Console App** and name it "Arithmetic"

```
Console.Title = "Arithmetic" ;
```

2. Type these statements to create and initialize two integer variables

```
int a = 8 ;
int b = 4 ;
```

3. Next, add statements to output the result of each basic arithmetical operation

```
Console.WriteLine( "Addition:\t: " + ( a + b ) ) ;
Console.WriteLine( "Subtraction:\t: " + ( a - b ) ) ;
Console.WriteLine( "Multiplication:\t: " + ( a * b ) ) ;
Console.WriteLine( "Division:\t: " + ( a / b ) ) ;
Console.WriteLine( "Modulus:\t: " + ( a % b ) ) ;
```

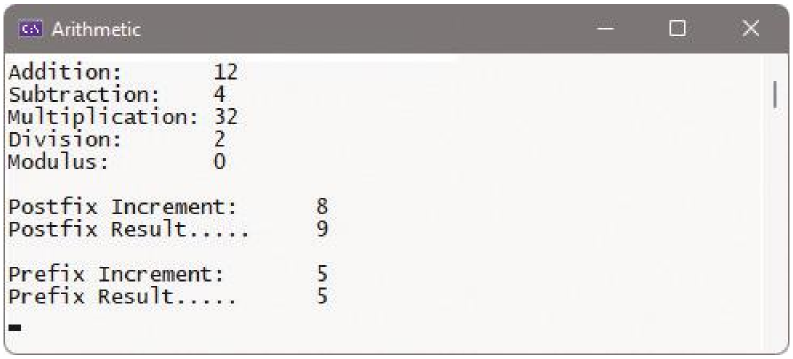
4. Now, add statements to output the result of a postfix increment operation

```
Console.WriteLine( "\nPostfix Increment:\t: " + ( a++ ) ) ;
Console.WriteLine( "Postfix Result.....:\t: " + a ) ;
```

5. Finally, add statements to output the result of a prefix increment operation – for comparison with postfix

```
Console.WriteLine( "\nPrefix Increment:\t: " + ( ++b ) ) ;
Console.WriteLine( "Prefix Result.....:\t: " + b ) ;
Console.ReadKey( ) ;
```

6. Press **Start** or **F5** to run the application and see the results produced by each arithmetic operator



```
Addition:       12
Subtraction:    4
Multiplication: 32
Division:       2
Modulus:        0

Postfix Increment:      8
Postfix Result.....     9

Prefix Increment:       5
Prefix Result.....      5
```

Don't forget

Here, the **\t** escape sequence is used to output a tab space, and the **\n** escape sequence is used to output additional newlines.

Beware

Remember that a prefix operator changes the variable value immediately but a postfix operator changes the value subsequently.

Assigning values

The operators that are used in C# programming to assign values are listed in the table below. All except the simple **=** assignment operator are a shorthand form of a longer expression, so each equivalent is given for clarity:

Operator	Example	Equivalent
=	a = b	a = b
+=	a += b	a = (a + b)
-=	a -= b	a = (a - b)
*=	a *= b	a = (a * b)
/=	a /= b	a = (a / b)
%=	a %= b	a = (a % b)

Don't forget

The **+=** combined operator is dual-purpose – it can also be used to concatenate strings.

Beware

It is important to regard the **=** operator to mean "assign" rather than "equals" to avoid confusion with the **==** equality operator that is described on page 38.

In the assignment example above, where **a = b**, the variable named "a" is assigned the value that is contained in the variable named "b" – so that is then the value stored in the **a** variable. Technically speaking, the assignment operator stores the value of the right-hand operand in the memory location denoted by the left-hand operand, then returns the value as a result.

The **+=** combined operator is useful to add a value onto an existing value that is stored in the **a** variable. In the example above, where **a += b**, the value in variable **b** is added to that in variable **a** – so the total is then the value stored in the **a** variable. The arithmetic operation is performed first with the grouped operands. The result is then stored in the memory location denoted by the first variable and returned.

All the other combined operators work in the same way by making the arithmetical operation between the two values first, then assigning the result of that operation to the first variable – to become its new stored value.

With the **%=** combined operator, the grouped left-hand operand **a** is divided by the grouped right-hand operand **b**, then the remainder of that operation is assigned to the **a** first variable.

...cont'd

1. Start a new **C# Console App** and name it "Assign"

```
Console.Title = "Assign" ;
```

2. Type these statements to declare two integer variables

```
int a ;
int b ;
```

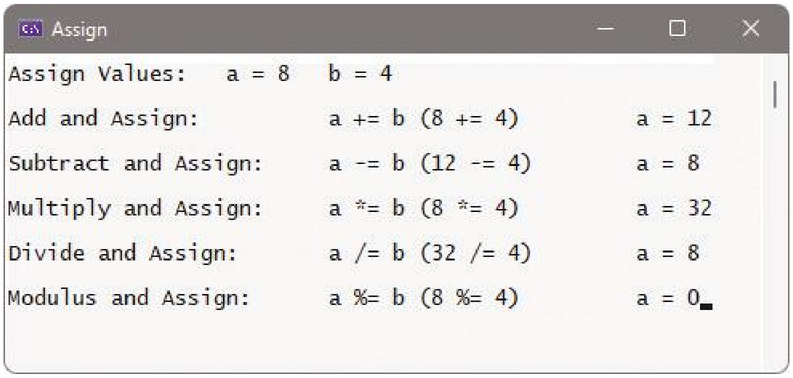
3. Next, add statements to output simple assigned values

```
Console.Write( "Assign Values: " ) ;
Console.Write( "\t a = " + ( a = 8 ) ) ;
Console.WriteLine( "\t b = " + ( b = 4 ) ) ;
```

4. Now, add statements to output combined assigned values

```
Console.Write( "\n\nAdd and Assign: " ) ;
Console.Write( "\t a += b (8 += 4)\t a = " + ( a += b ) ) ;

Console.Write( "\n\nSubtract and Assign: " ) ;
Console.Write( "\t a -= b (12 -= 4)\t a = " + ( a -= b ) ) ;

Console.Write( "\n\nMultiply and Assign: " ) ;
Console.Write( "\t a *= b (8 *= 4)\t a = " + ( a *= b ) ) ;

Console.Write( "\n\nDivide and Assign: " ) ;
Console.Write( "\t a /= b (32 /= 4)\t a = " + ( a /= b ) ) ;

Console.Write( "\n\nModulus and Assign: " ) ;
Console.Write( "\t a %= b (8 %= 4)\t a = " + ( a %= b ) ) ;
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application and see the results produced by each assignment operator



```
Assign Values:    a = 8    b = 4

Add and Assign:           a += b (8 += 4)        a = 12

Subtract and Assign:      a -= b (12 -= 4)       a = 8

Multiply and Assign:      a *= b (8 *= 4)        a = 32

Divide and Assign:        a /= b (32 /= 4)       a = 8

Modulus and Assign:       a %= b (8 %= 4)        a = 0
```

Assign

Hot tip

Notice how **\n\n** escape sequences are used here to add two newlines for spacing output.

Beware

The operands must be of the same data type, or the right-hand operand must be implicitly convertible to the type of the left-hand operand.

Comparing values

The operators that are commonly used in C# programming to compare two numerical values are listed in the table below:

Operator	Comparative test
==	Equality
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Hot tip

The comparison operators are also known as "relational operators".

Hot tip

You can specify a Boolean value to the **Convert.ToDouble()** method to discover that **true** is represented numerically as 1, and **false** is represented as 0.

The **==** equality operator compares two operands and will return **true** if both are equal in value, otherwise the **==** operator will return **false**. If both are the same number, they are equal, or if both are characters, their Unicode code values are compared numerically.

Conversely, the **!=** inequality operator returns **true** if two operands are not equal, using the same rules as the **==** equality operator, otherwise it returns **false**. Equality and inequality operators are useful in testing the state of two variables to perform conditional branching in a program.

The **>** "greater than" operator compares two operands and will return **true** if the first is greater in value than the second, or it will return **false** if it is equal or less in value. The **<** "less than" operator makes the same comparison but returns **true** if the first operand is less in value than the second, otherwise it returns **false**. Typically, a **>** "greater than" or **<** "less than" operator is used to test the value of an iteration counter in a loop structure.

Adding the **=** assignment operator after a **>** "greater than" operator or a **<** "less than" operator makes it also return **true** if the two operands are exactly equal in value.

1. Start a new **C# Console App** and name it "Comparison"
```
Console.Title = "Comparison" ;
```

Comparison

2. Type these statements to declare three integer variables and two character variables to be compared
```
int nil = 0 , num = 0 , max = 1 ;
char cap = 'A' , low = 'a' ;
```

3. Next, add statements to output the result of equality comparisons of integers and characters
```
Console.Write( "Equality:" ) ;
Console.Write( "\t (0 == 0) : " + ( nil == num ) ) ;
Console.Write( "\n\t\t (A == a) : " + ( cap == low ) ) ;
```

4. Now, add statements to output the result of other integer comparisons
```
Console.Write( "\n\nInequality:" ) ;
Console.Write( "\t (0 != 1) : " + ( nil != max ) ) ;

Console.Write( "\n\nGreater:" ) ;
Console.Write( "\t (0 > 1) : " + ( nil > max ) ) ;

Console.Write( "\nLess:" ) ;
Console.Write( "\t\t (0 < 1) : " + ( nil < max ) ) ;

Console.Write( "\n\nGreater/Equal:" ) ;
Console.Write( "\t (0 >= 0) : " + ( nil >= num ) ) ;

Console.Write( "\nLess or Equal:" ) ;
Console.Write( "\t (1 <= 0) : " + ( max <= nil ) ) ;
Console.ReadKey( ) ;
```

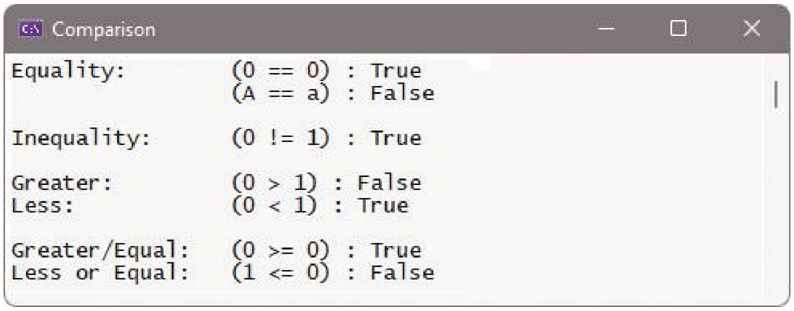
5. Press **Start** or **F5** to run the application and see the results produced by each comparison

Don't forget

The Unicode code value for uppercase **'A'** is 65 but for lowercase **'a'** it's 97 – so their comparison here returns **false**.

Beware

When comparing numbers, remember to test for equality as well as testing for higher and lower values.


Comparison
```
Equality:         (0 == 0) : True
                  (A == a) : False

Inequality:       (0 != 1) : True

Greater:          (0 > 1) : False
Less:             (0 < 1) : True

Greater/Equal:    (0 >= 0) : True
Less or Equal:    (1 <= 0) : False
```

Assessing logic

The logical operators most commonly used in C# programming are listed in the table below:

Operator	Operation
&&	Logical AND
||	Logical OR
!	Logical NOT

The logical operators are used with operands that have Boolean values of **true** or **false**, or are values that convert to **true** or **false**.

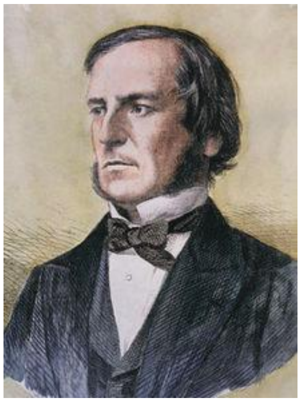
The **&&** logical AND operator will evaluate two operands and return **true** only if both operands themselves are **true**. Otherwise, the **&&** logical AND operator will return **false**. This is used in conditional branching where the direction of a program is determined by testing two conditions – if both conditions are satisfied, the program will go in a certain direction, otherwise it will take a different direction.

Unlike the **&&** logical AND operator, which needs both operands to be **true**, the **||** logical OR operator will evaluate its two operands and return **true** if either one of the operands itself returns **true**. If, however, neither operand returns **true**, then the **||** logical OR operator will return **false**. This is useful in C# programming to perform a certain action if either one of two test conditions has been met.

The **!** logical NOT operator is a unary operator that is used before a single operand. It returns the inverse value of the given operand, so if the variable **a** had a value of **true** then **!a** would have a value of **false**. The **!** logical NOT operator is useful in C# programs to toggle the value of a variable in successive loop iterations with a statement like **a = !a**. This ensures that on each pass the value is changed, like flicking a light switch on and off.

Hot tip

The term "Boolean" refers to a system of logical thought developed by the English mathematician George Boole (1815-1864).



1 Start a new **C# Console App** and name it "Logic"

```
Console.Title = "Logic" ;
```

Logic

2 Type this statement to declare two Boolean variables

```
bool yes = true , no = false ;
```

3 Add these statements to output the result of AND logic

```
Console.Write( "AND Logic:" ) ;
Console.Write( "\t (yes && yes) : " + ( yes && yes )  ) ;
Console.Write( "\n\t\t (yes && no ) : " + ( yes && no )  ) ;
Console.Write( "\n\t\t (no  && no ) : " + ( no && no )  ) ;
```

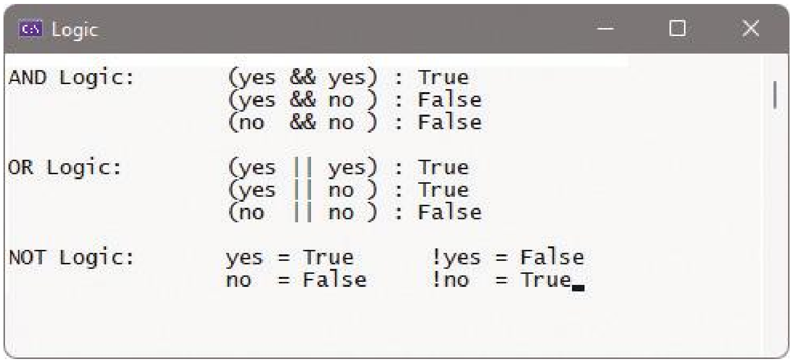
4 Now, add statements to output the result of OR logic

```
Console.Write( "\n\nOR Logic:" ) ;
Console.Write( "\t (yes || yes) : " + ( yes || yes )  ) ;
Console.Write( "\n\t\t (yes || no ) : " + ( yes || no )  ) ;
Console.Write( "\n\t\t (no  || no ) : " + ( no || no )  ) ;
```

5 Then, add statements to output the result of NOT logic

```
Console.Write( "\n\nNOT Logic:" ) ;
Console.Write( "\t yes = " + yes ) ;
Console.Write( "\t !yes = " + !yes ) ;
Console.Write( "\n\t\t no = " + no ) ;
Console.Write( "\t !no = " + !no ) ;
Console.ReadKey( ) ;
```

6 Press **Start** or **F5** to run the application and see the results produced by each logical operation



```
AND Logic:         (yes && yes) : True
                   (yes && no ) : False
                   (no  && no ) : False

OR Logic:          (yes || yes) : True
                   (yes || no ) : True
                   (no  || no ) : False

NOT Logic:         yes  = True       !yes = False
                   no   = False      !no  = True
```

Don't forget

The value returned by the **!** logical NOT operator is the inverse of the stored value – the stored value itself remains unchanged.

Hot tip

Notice that **false && false** returns **false**, not **true** – demonstrating the maxim "two wrongs don't make a right".

Examining conditions

Possibly, the C# programmer's favorite test operator is the **?:** "ternary" operator. This operator first evaluates an expression for a **true** or **false** condition, then returns one of two specified values depending on the result of the evaluation. For this reason, it is also known as the "conditional" operator.

The **?:** ternary operator has this syntax:

(*test-expression*) ? *if-true-return-this* : *if-false-return-this* ;

Although the ternary operator can initially appear a little confusing, it is well worth becoming familiar with this operator as it can execute powerful program branching with minimal code – for example, to branch when a variable is not a value of 1:

(*variable* != 1) ? *if-true-do-this* : *if-false-do-this* ;

Don't forget

The ternary operator has three operands – the one before the **?** and those before and after the **:**.

The ternary operator is commonly used in C# programming to assign the maximum or minimum value of two variables to a third variable, for example, to assign a minimum, like this:

c = (a < b) ? a : b ;

The expression in parentheses returns **true** when the value of variable **a** is less than that of variable **b** – so in this case, the lesser value of variable **a** gets assigned to variable **c**.

Similarly, replacing the **<** less than operator in the test expression with the **>** greater than operator would assign the greater value of variable **b** to variable **c**.

Hot tip

Value equality, where two variables contain an equal value, is also known as "equivalence".

Another common use of the ternary operator incorporates the **%** modulus operator in the test expression to determine whether the parity value of a variable is an odd number or an even number:

(*variable* % 2 != 0) ? *if-true(odd)-do-this* : *if-false(even)-do-this* ;

Where the result of dividing the variable value by 2 does leave a remainder, the number is odd – where there is no remainder, the number is even. The test expression **(*variable* % 2 == 1)** would have the same effect, but it is preferable to test for inequality – it's easier to spot when something is different than when it's identical.

...cont'd

1 Start a new **C# Console App** and name it "Condition"

```
Console.Title = "Condition" ;
```

2 Type these statements to declare and initialize two integer variables

```
int a = 8 ;
int b = 3 ;
```

3 Next, add these statements to output an appropriate string with correct grammar for quantity

```
string verb = ( a != 1 ) ? " are " : " is " ;
Console.WriteLine( "There" + verb + a +"\n" ) ;
```

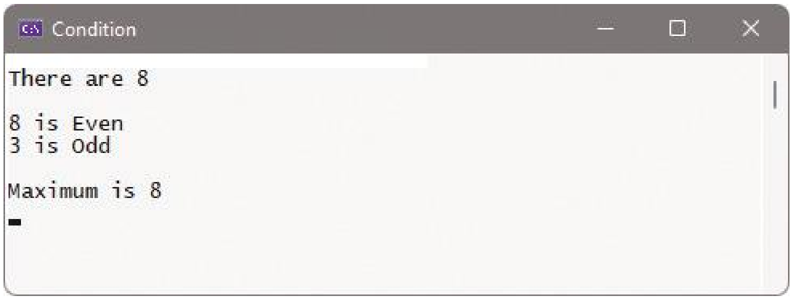
4 Now, add statements to output an appropriate string correctly describing the parity of each variable value

```
string parity = ( a % 2 != 0 ) ? "Odd" : "Even" ;
Console.WriteLine( a + " is " + parity ) ;

parity = ( b % 2 != 0 ) ? "Odd" : "Even" ;
Console.WriteLine( b + " is " + parity ) ;
```

5 Then, add statements to output a string reporting the greater of these two variable values

```
int max = ( a > b ) ? a : b ;
Console.Write( "\nMaximum is " + max ) ;
Console.ReadKey( ) ;
```

6 Press **Start** or **F5** to run the application and see the results produced by examining each variable value



```
There are 8

8 is Even
3 is Odd

Maximum is 8
```

C#

Condition

Hot tip

The **?:** ternary operator can return values of any data type – numbers, strings, Booleans, etc.

Setting precedence

Operator precedence determines the order in which C# evaluates expressions. For example, the expression **1 + 5 * 3** evaluates to 16, not 18, because the * multiplication operator has a higher precedence than the + addition operator. Parentheses can be used to specify precedence, so that **(1 + 5) * 3** evaluates to 18.

When operators have equal precedence, their "associativity" determines how expressions are grouped. For example, the - subtraction operator is left-associative, grouping left-to-right (LTR), so **8 - 4 - 2** is grouped as **(8 - 4) - 2** and thus evaluates to 2. Other operators are right-associative, grouping right-to-left (RTL).

The table below lists common operators in order of precedence, with the highest-precedence ones at the top. Operators on the same line have equal precedence, so operator associativity determines how expressions are grouped and evaluated.

Category	Operator	Associativity
Postfix	`( ) [ ] . ++ --`	LTR ☞

Unary
Sign
Prefix
	`!`
	`+ -`
	`++ --`
		☜ RTL

Multiplicative	`* / %`	LTR ☞
Additive	`+ -`	LTR ☞
Comparative	`< <= > >=`	LTR ☞
Equivalence	`== !=`	LTR ☞
Conditional	`&&`	LTR ☞
Conditional	`||`	LTR ☞
Conditional	`?:`	☜ RTL
Assignment	`= += -= *= /= %=`	☜ RTL
Comma	`,`	LTR ☞

Don't forget

The * multiply operator is on a higher row than the + addition operator – so in the expression **a=1+5*3**, multiplication is completed first, before the addition.

Hot tip

There are also a number of "bitwise" operators, which are used to perform binary arithmetic. This is outside the scope of this book but there is a section devoted to binary arithmetic in our book C Programming in easy steps. Those operators perform in just the same way in C#.

1. Start a new **C# Console App** and name it "Precedence"
```
Console.Title = "Precedence" ;
```

2. Type this statement to declare an integer variable
```
int sum ;
```

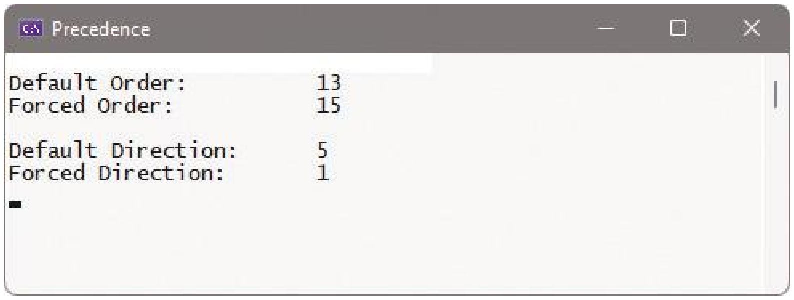
3. Add these statements to initialize the variable with the result of an ungrouped expression and display that result
```
sum = 1 + 4 * 3 ;
Console.WriteLine( "Default Order:\t\t" + sum ) ;
```

4. Next, add statements to assign the result of a grouped expression and display that result
```
sum = ( 1 + 4 ) * 3 ;
Console.WriteLine( "Forced Order:\t\t" + sum ) ;
```

5. Add statements to assign the result of a new ungrouped expression and display that result
```
sum = 7 - 4 + 2 ;
Console.WriteLine( "\nDefault Direction:\t" + sum ) ;
```

6. Now, add statements to assign the result of the new grouped expression and display that result
```
sum = 7 - ( 4 + 2 ) ;
Console.WriteLine( "\nForced Direction:\t" + sum ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the results produced by examining each variable value



```
Default Order:          13
Forced Order:           15

Default Direction:      5
Forced Direction:       1
```

Precedence

Don't forget

The * multiplication operator takes precedence over the + addition operator – so multiplication is performed first.

Don't forget

The - subtraction operator and the + addition operator have equal precedence but also have left-to-right associativity – so subtraction is performed first before addition.

Beware

It is best to clarify all expressions by adding parentheses to group operations.

Summary

- Arithmetic operators can form expressions with two operands for **+** addition, **-** subtraction, ***** multiplication, and **/** division.
- The **%** modulus arithmetic operator divides the first operand by its second operand, then returns the remainder.
- Care must be taken to group expressions within **( )** parentheses where more than one operator is used.
- The **++** increment and **--** decrement operators may be postfixed or prefixed to modify a single operand by 1.
- The **=** assignment operator can be combined with an arithmetic operator to assign the result of an operation.
- Comparison operators can form expressions with two operands for **==** equality or for **!=** inequality.
- Comparison operators can form expressions with two operands for **>** greater or for **<** lesser value comparison.
- Equality can also be recognized in comparisons with the **>=** greater-or-equal and **<=** less-or-equal operators.
- The **&&** logical-AND operator will return **true** only if both operands are themselves **true**.
- The **||** logical-OR operator will return **true** if either of the two operands are themselves **true**.
- The **!** logical-NOT operator returns the inverse Boolean value of a single given operand.
- The **?:** ternary operator evaluates a given Boolean expression, then returns one of two operands depending on the result.
- Operator precedence determines the order in which expressions are evaluated.
- When operators have equal precedence, their associativity determines how expressions are grouped for evaluation.

4 Making statements

This chapter demonstrates how statements can evaluate expressions to determine the direction in which a C# program will proceed.

Branching with if

The C# **if** keyword performs the basic conditional test that evaluates a given expression for a Boolean value of **true** or **false** – and its syntax looks like this:

if (*test-expression*) { *statements-to-execute-when-true* }

The braces following the test may contain one or more statements, each terminated by a ; semicolon, but these will only be executed when the expression is found to be **true**. When the test is found to be **false**, the program proceeds to its next task.

To allow "conditional branching", an **if** statement can offer alternative statements to execute when the test fails by appending an **else** statement block after the **if** statement block, like this:

if (*test-expression*) { *statements-to-execute-when-true* }
else { *statements-to-execute-when-false* }

To test two conditions, the test expression may use the **&&** operator – for example, **if ((num > 5) && (letter == 'A'))**.

Alternatively, an **if** statement can be "nested" within another **if** statement, so those statements in the inner statement block will only be executed when both tests succeed – but statements in the outer statement block will be executed if the outer test succeeds:

Hot tip

Shorthand can be used when testing a Boolean value – so the expression **if(flag == true)** can be written as **if(flag)**.

C#

IfElse

1. Start a new **C# Console App** and name it "IfElse"

```
Console.Title = "IfElse" ;
```

2. Type these statements to create and initialize two variables from user input

```
Console.Write( "Please Enter A Number: " ) ;
double num = Convert.ToDouble( Console.ReadLine( ) ) ;
Console.Write( "Thanks. Now Enter A Letter: " ) ;
char letter = Convert.ToChar( Console.ReadLine( ) ) ;
```

3. Next, add a statement to output a message if the user's number input exceeds a specified value

```
if ( num >= 6 )
{
  Console.WriteLine( "\nNumber Exceeds 5" ) ;
  // Nested statement to be inserted here (Step 5).
}
```

Beware

Avoid nesting more than three levels of **if** statements – to prevent confusion and errors.

4 Now, add a statement to output an alternative message if the user's number input is less than the specified value

```
else
{
  Console.WriteLine( "\nNumber Is 5 Or Less" ) ;
}
Console.ReadKey( ) ;
```

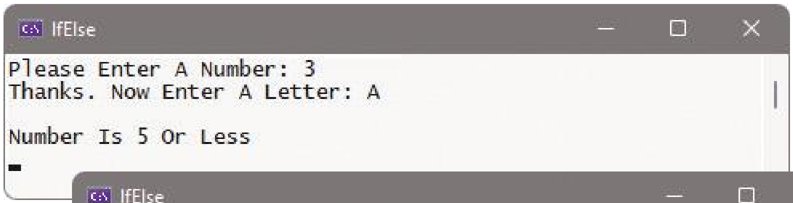
5 Insert a statement within the **if** block to output a message when the user's letter input matches a specified value

```
if ( letter == 'C' )
{
  Console.WriteLine( "Letter Is 'C'" ) ;
}
```

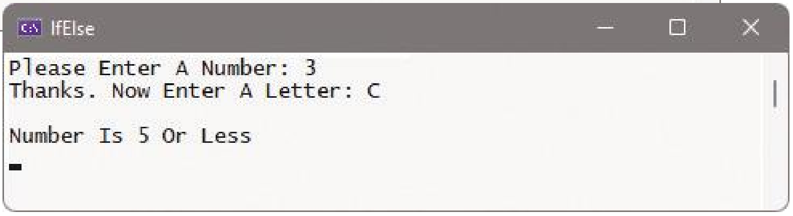
6 Press **Start** or **F5** to run the application and enter values to see the program branch according to your input

Hot tip

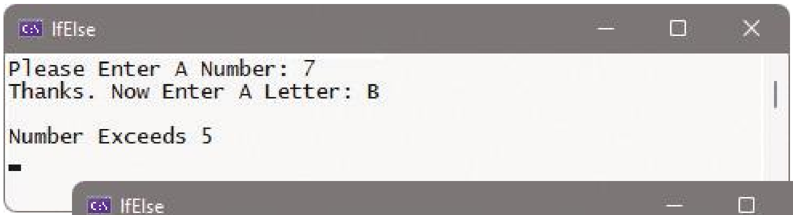
Where there is only one statement to execute when the test succeeds, the **{ }** braces may be omitted – but retaining them aids code clarity.


IfElse

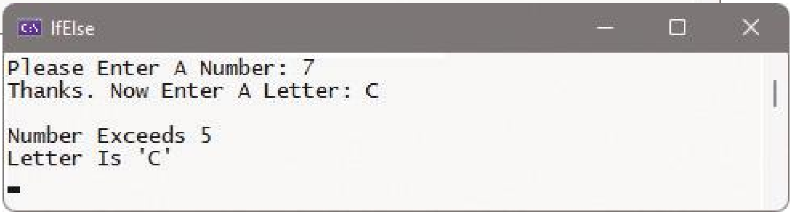
```
Please Enter A Number: 3
Thanks. Now Enter A Letter: A

Number Is 5 Or Less
```


IfElse

```
Please Enter A Number: 3
Thanks. Now Enter A Letter: C

Number Is 5 Or Less
```


IfElse

```
Please Enter A Number: 7
Thanks. Now Enter A Letter: B

Number Exceeds 5
```


IfElse

```
Please Enter A Number: 7
Thanks. Now Enter A Letter: C

Number Exceeds 5
Letter Is 'C'
```

Switching branches

The **if** and **else** keywords, introduced on pages 48-49, allow programs to branch in a particular direction according to the result of a test condition, and can be used to repeatedly test a variable to match a value – for example, testing for an integer:

```
if ( num == 1 ) { Console.Write( "Monday" ) ; }
else
if ( num == 2 ) { Console.Write( "Tuesday" ) ; }
else
if ( num == 3 ) { Console.Write( "Wednesday" ) ; }
else
if ( num == 4 ) { Console.Write( "Thursday" ) ; }
else
if ( num == 5 ) { Console.Write( "Friday" ) ; }
```

The program will branch in the direction of the match.

Hot tip

Avoid writing lengthy **if-else** statements like the one shown here – where possible, use a **switch** statement instead.

Conditional branching with long **if-else** statements can often be more efficiently performed using a **switch** statement instead, especially when the test expression evaluates just one variable.

The **switch** statement works in an unusual way. It takes a given variable value, or expression, then seeks a matching value among a number of **case** statements. One or more statements associated with the matching case statement will then be executed.

When no match is found, no **case** statements will be executed, but you may add a **default** statement after the final **case** statement to specify statements to be executed when no match is found. The syntax of a typical switch statement looks like this:

```
switch( variable-name )
{
  case value1 : statement ; break ;
  case value2 : statement ; break ;
  case value3 : statement ; break ;
  default : statement ; break ;
}
```

Beware

Missing **break** keywords in C# **case** statements are syntax errors.

It is important to follow each case statement with the **break** keyword. Unlike other programming languages, C# does not allow fall-through from one **case** statement to another – each **case** statement must allow control to be handed back in order to exit the **switch** block.

...cont'd

1 Start a new **C# Console App** and name it "Switch"

```
Console.Title = "Switch" ;
```

Switch

2 Type this statement to create and initialize an integer variable

```
int num = 3 ;
```

3 Next, add a statement to declare a **string** variable

```
string day ;
```

4 Now, add a statement to initialize the **string** variable according to the value of the integer variable

```
switch( num )
{
  case 1 : day = "Monday" ; break ;
  case 2 : day = "Tuesday" ; break ;
  case 3 : day = "Wednesday" ; break ;
  case 4 : day = "Thursday" ; break ;
  case 5 : day = "Friday" ; break ;
  // Default statement to be inserted here (Step 5).
}
```

Hot tip

A **case** statement can also try to match against a **string** value – for example: **case : "ABC"**.

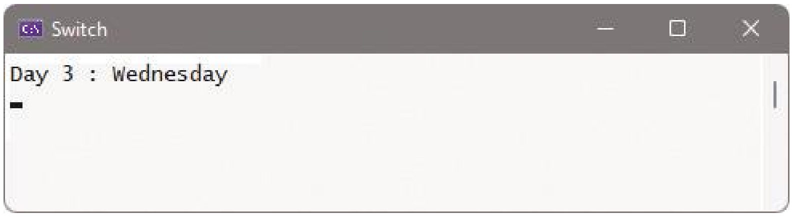
5 Then, insert a final statement into the **switch** block to initialize the **string** variable when no match is found

```
default : day = "Weekend Day" ; break ;
```

6 Finally, add statements to output the assigned value

```
Console.WriteLine( "Day " + num + " : " + day ) ;
Console.ReadKey( ) ;
```

7 Press **Start** or **F5** to run the application and see the **string** result of the **switch** block assignment


Switch

```
Day 3 : Wednesday
```

Looping for

A loop is a piece of code in a program that automatically repeats. One complete execution of all statements contained within the loop block is known as an “iteration” or “pass”.

The number of iterations made by a loop is controlled by a conditional test made within the loop. While the tested expression remains **true**, the loop will continue – until the tested expression becomes **false**, at which time the loop ends.

The three types of loop structures in C# programming are **for** loops, **while** loops, and **do-while** loops. Perhaps the most commonly used loop is the **for** loop, which has this syntax:

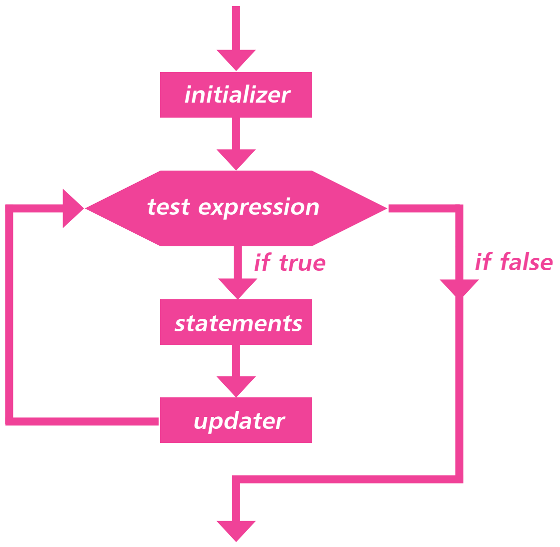
for (***initializer*** **;** ***test-expression*** **;** ***updater*** **) {** ***statements*** **}**

The initializer sets the starting value for a counter of the number of iterations to be made by the loop. An integer variable is used for this purpose and is traditionally named “i”.

Upon each iteration of the loop, the test expression is evaluated, and that iteration will only continue while this expression is **true**. When the tested expression becomes **false**, the loop ends immediately without executing the statements again. On each iteration, the counter is updated then the statements executed.

Hot tip

The updater may increment the counter using the **++** operator to count up, or decrement the counter using the **--** decrement operator to count down.



Loops may be nested within other loops – so that the inner loop will fully execute its iterations on each iteration of the outer loop.

1. Start a new **C# Console App** and name it "ForLoop"

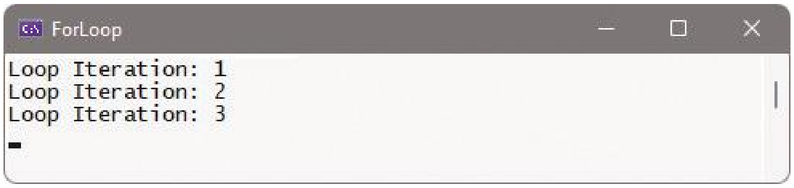
```
Console.Title = "ForLoop" ;
```

ForLoop

2. Type these statements to create a loop that will make three iterations

```
for( int i = 1 ; i < 4 ; i++ )
{
  Console.WriteLine( "Loop Iteration: " + i ) ;
  // Nested loop to be inserted here (Step 4).
}
Console.ReadKey( ) ;
```

3. Press **Start** or **F5** to run the application and see the loop counter displayed on each iteration

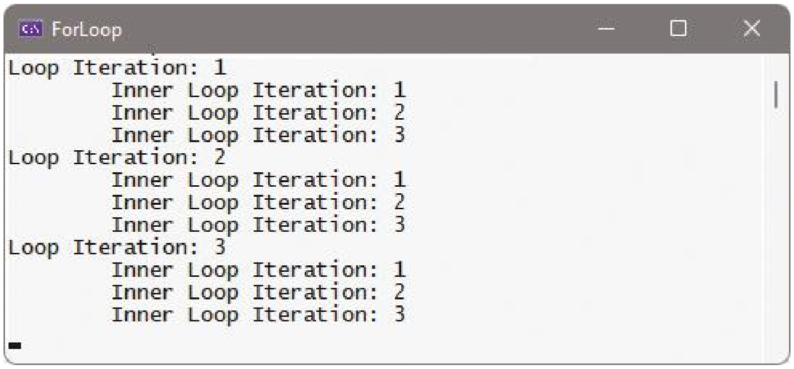


```
Loop Iteration: 1
Loop Iteration: 2
Loop Iteration: 3
▬
```

4. Inside the **for** loop block, insert another **for** loop that will also make three iterations

```
  for( int j = 1 ; j < 4 ; j++ )
  {
    Console.WriteLine( "\tInner Loop Iteration: " + j ) ;
  }
```

5. Press **Start** or **F5** to run the application once more and now see both loop counters displayed on each iteration



```
Loop Iteration: 1
        Inner Loop Iteration: 1
        Inner Loop Iteration: 2
        Inner Loop Iteration: 3
Loop Iteration: 2
        Inner Loop Iteration: 1
        Inner Loop Iteration: 2
        Inner Loop Iteration: 3
Loop Iteration: 3
        Inner Loop Iteration: 1
        Inner Loop Iteration: 2
        Inner Loop Iteration: 3
▬
```

Don't forget

On the third iteration of these loops, the updater increments the counter value to **4** – so when it is next evaluated, the test expression returns **false** and the loop ends.

Looping while

An alternative to the **for** loop, introduced on pages 52-53, uses the **while** keyword, followed by an expression to be evaluated. When the expression is **true**, statements contained within braces following the test expression will be executed. The expression will then be evaluated again, and the **while** loop will continue until the expression is found to be **false**.

while (*test-expression*) { *statements* }

The loop's statement block <u>must</u> contain code that will affect the tested expression in order to change the evaluation result to **false**, otherwise an infinite loop is created that will lock the program! If the tested expression is found to be **false** upon its first evaluation, the **while** loop's statement block will never be executed.

A subtle variation of the **while** loop places the **do** keyword before the loop's statement block and a **while** test after it, with this syntax:

do { *statements* } while (*test-expression*) ;

In a **do-while** loop, the statement block will always be executed at least once – because the expression is not evaluated until after the first iteration of the loop.

Hot tip

If you accidentally start running an infinite loop – for example:

```
while( true )
{
  Console.Write( i ) ;
  i++ ;
}
```

press the **Ctrl** + **C** keys to terminate the process.

Breaking out of loops

A **break** statement can be included in any kind of loop to immediately terminate the loop when a test condition is met. The **break** ensures no further iterations of that loop will be executed.

Similarly, a **continue** statement can be included in any kind of loop to immediately terminate that particular iteration of the loop when a test condition is met. The **continue** statement allows the loop to proceed to the next iteration:

WhileLoop

1. Start a new **C# Console App** and name it “WhileLoop”
```
Console.Title = "WhileLoop" ;
```

2. Type these statements to create and initialize an integer array variable and a regular integer counter variable
```
int [ ] nums = new int[ 10 ] ;
int i = 0 ;
```

3 Next, add a **while** loop to assign its incrementing counter value to an array element and display it on each iteration

```
while( i < nums.Length )
{
  nums[ i ] = i ;
  Console.Write( " | " + nums[ i ] ) ;
  i++ ;
}
Console.Write( "\n\n" ) ;
```

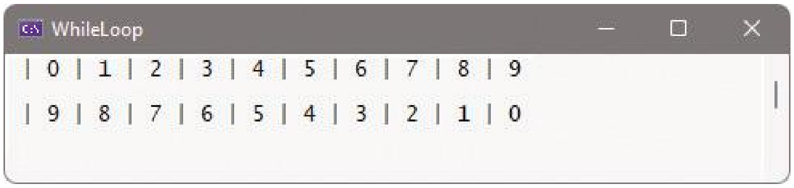
Notice how the array's **Length** property is used to determine the number of elements it contains.

4 Now, add a **do-while** loop to display its decrementing counter value on each iteration

```
do
{
  i-- ;
  // Statements to be inserted here (step 6).
  Console.Write( " | " + nums[ i ] ) ;
}
while( i > 0 ) ;
Console.ReadKey( ) ;
```

Remember that the **while** statement at the end of a **do-while** loop must be terminated with a **;** semicolon.

5 Press **Start** or **F5** to run the application and see both loop counters displayed on each iteration



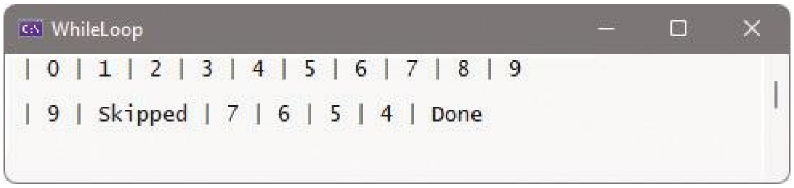
```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0
```

6 In the **do-while** loop, insert a statement to skip a single iteration, and a statement to prematurely exit the loop

```
if( i == 8 ) { Console.Write( " | Skipped" ) ; continue ; }
if( i == 3 ) { Console.Write( " | Done" ) ; break ; }
```

The position of **break** and **continue** statements is important – they must appear after the updater, to avoid creating an infinite loop, but before any other statements, to prevent their execution.

7 Press **Start** or **F5** to run the application once more, and now see the **do-while** loop iterations skipped



```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

| 9 | Skipped | 7 | 6 | 5 | 4 | Done
```

Iterating for each

C# provides a special **foreach-in** statement that allows you to easily traverse all elements in an array to access all stored values. This is an “iterator”, rather than a loop structure, so it cannot be used to assign or modify element values – it can only be used to read their values. Syntax of a **foreach-in** statement looks like this:

foreach (*data-type variable-name* in *array-name*) { *statements* }

The **foreach** keyword must specify a variable of an appropriate data type to which each element value can be assigned, one by one, and the **in** keyword must specify the name of the array:

ForEach

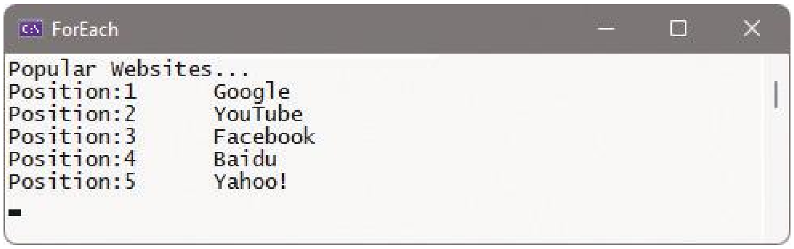
1. Start a new **C# Console App** and name it “ForEach”
```
Console.Title = "ForEach" ;
```

2. Type this statement to create and initialize a **string** array
```
string [ ] websites = new string [ 5 ]
{ "Google", "YouTube", "Facebook", "Baidu", "Yahoo!" } ;
```

3. Next, add a statement to display a list heading
```
Console.WriteLine( "Popular Websites..." ) ;
```

4. Now, add a **foreach** statement to display a counter value and list item value on each iteration
```
int rank = 1 ;
foreach( string site in websites )
{
  Console.WriteLine( "Position: " + rank + "\t" + site ) ;
  rank++ ;
}
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application and see the iterator display the counter and element values


ForEach
```
Popular Websites...
Position:1      Google
Position:2      YouTube
Position:3      Facebook
Position:4      Baidu
Position:5      Yahoo!
```

Beware

Use a **for** loop if you need to assign or modify element values.

Hot tip

The **break** and **continue keywords** can be used in a **foreach** statement to exit the iteration cycle.

A **foreach-in** statement can also be used to traverse all elements of a C# "dictionary" that contains associated key-value pairs. The data types of the key and value must be specified as a comma-separated pair within < > angled brackets in the declaration:

```
Dictionary < data-type, data-type > dictionary-name =
          new Dictionary < data-type, data-type > ( ) ;
```

The **foreach** keyword must specify a **KeyValuePair** to which each key-value pair value can be assigned, one by one, and the **in** keyword must specify the name of the dictionary:

1. Start a new **C# Console App** and name it "KeyValue"

```
Console.Title = "KeyValue" ;
```

KeyValue

2. Type these statements to create and initialize a dictionary named "BookList"

```
Dictionary < string, string > BookList =
          new Dictionary < string, string > ( ) ;
BookList.Add( "Michael Price", "Microsoft 365" ) ;
BookList.Add( "Nick Vandome","Windows 11" ) ;
BookList.Add( "Mike McGrath", "Java" ) ;
```

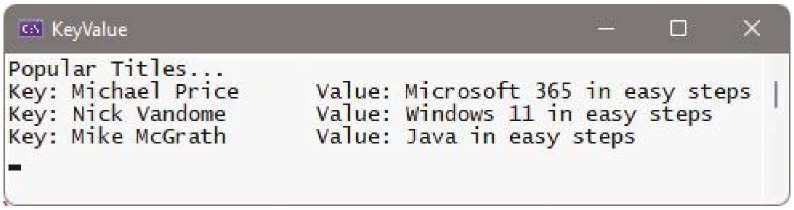
3. Now, add a statement to display a list heading

```
Console.WriteLine( "Popular Titles..." ) ;
```

4. Add a **foreach** statement to display each key-value pair

```
foreach( KeyValuePair < string, string > book in BookList )
{
  Console.WriteLine( "Key: " + book.Key +
          "\tValue: "+ book.Value + " in easy steps" ) ;
}
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application and see the iterator display the dictionary's key-value pairs



```
Popular Titles...
Key: Michael Price       Value: Microsoft 365 in easy steps
Key: Nick Vandome        Value: Windows 11 in easy steps
Key: Mike McGrath        Value: Java in easy steps
```

Hot tip

Note that a **Dictionary** object has **Add( )** and **Remove( )** methods, and has **Key** and **Value** properties.

Summary

- An **if** statement evaluates a given test expression for a Boolean value of **true** or **false**.
- Statements contained in braces after an **if** statement will only be executed when the evaluation is found to be **true**.
- Multiple **if** statements can be nested, so statements in the inner loop are only executed when both loop tests return **true**.
- The **if** and **else** keywords are used to perform conditional branching according to the result of a tested expression.
- A **switch** statement is an alternative form of conditional branching that matches a **case** statement to a given value.
- Each **case** statement must be terminated by the **break** keyword, as C# does not allow fall-through.
- The **for** loop structure has parameters declaring an initializer, a test expression, and an updater.
- A loop updater may **++** increment a counter to count up, or may **--** decrement a counter to count down.
- A **while** loop may never execute its statements, but a **do-while** loop will always execute its statements at least once.
- A **while** loop and **do-while** loop must always have an updater within their loop body to avoid creating an infinite loop.
- Any type of loop can be immediately terminated by including a **break** statement within the loop body.
- A single iteration of any type of loop can be skipped by including a **continue** statement within the loop body.
- A **foreach-in** statement is an iterator that can traverse all elements of an array to read their values.
- A **foreach-in** statement can traverse all key-value pairs of a dictionary to read their values.

5 Devising methods

This chapter demonstrates how to create methods that can be called to execute statements whenever the C# program requires them.

Creating function

A C# programmer-specified “function” encloses a section of reusable code that can be called whenever required by the program. Local functions can be created in the “Program.cs” file and can be called to execute their section of code as needed.

Functions can also be created within class structures. These are referred to as “methods” of the class. For example, the **WriteLine()** function is a method of the **Console** class.

Functions (a.k.a. methods) provide these three main benefits:

- Modular program code is easier to understand and maintain.
- Tried and tested code can be re-used by other programs.
- The workload on large programs can be more easily shared.

A function is declared in a C# program by specifying the various elements of its structure with this syntax:

```
access-specifier return-data-type function-name ( parameters )
{
  statements
}
```

The access specifier determines the visibility of the function to other classes. The keyword **public** specifies global accessibility, whereas **private** or **static** keywords specify only local accessibility.

If the function will return a value to the caller, the data type of that value must be specified. If the function will never return a value, the **void** keyword must be specified.

A name must be specified for each programmer-defined function, adhering to the same naming conventions as variable names.

Optionally, parameters may be specified within parentheses after the function name to represent “argument” values passed from the caller. Once defined, a method without parameters may be called simply by stating its name followed by parentheses.

Variables declared within a function are not, by default, visible to other parts of the program. The visibility of variables is known as “variable scope” and variables within a function have only “local” scope. This means that variables declared in different functions may be given the same name without conflict.

Hot tip

The first part of a function declaration, defining its access, return type, name, and parameters, is known as the function “signature”.

Hot tip

The use of methods by other classes is described and demonstrated in the chapter on Object Oriented Programming – see page 112.

Hot tip

Parameters and arguments are demonstrated in the example on page 62.

1. Start a new **C# Console App** and name it "Function"
```
Console.Title = "Function" ;
```

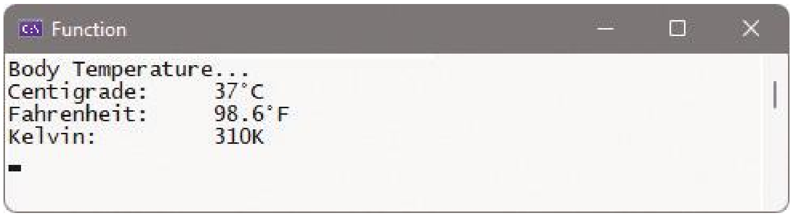
2. Type this code to create a function that will output two text strings, but return nothing to the caller
```
static void bodyTempC( )
{
  Console.WriteLine( "Body Temperature..." ) ;
  Console.WriteLine( "Centigrade:\t37ºC" ) ;
}
```

3. Next, add a function that outputs nothing, but will return a floating-point value to the caller
```
static double bodyTempF( )
{
  double temperature = 98.6 ;
  return temperature ;
}
```

4. Now, add a function that outputs nothing, but will return an integer value to the caller
```
static int bodyTempK( )
{
  int temperature = 310 ;
  return temperature ;
}
```

5. Finally, add statements to call each of the programmer-specified functions created above
```
bodyTempC( ) ;
Console.WriteLine( "Fahrenheit:\t" + bodyTempF( ) + "ºF" ) ;
Console.WriteLine( "Kelvin:\t" + bodyTempK( ) + "K" ) ;
Console.ReadKey( ) ;
```

6. Press **Start** or **F5** to run the application and see the output featuring the function calls


```
Body Temperature...
Centigrade:     37°C
Fahrenheit:     98.6°F
Kelvin:         310K
```

C#

Function

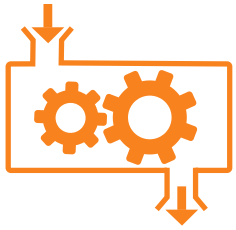
Don't forget

Each variable can have the same name as they have only local scope.

Hot tip

You can use the Windows Accessories, Character Map facility to produce the º degree symbol.

Passing arguments



Functions may optionally specify one or more parameters as a comma-separated list within the parentheses of their definition. Each specified parameter must state the data type it may contain, and a name by which it may be referenced within the function – for example, parameters for text and a number, like this:

```
static void setUser( string name , int age ) { statements }
```

When a function with parameters is called, the caller must normally pass the correct number of argument values of the required data type. These are listed within the parentheses of the calling statement – for example, calling the function above:

```
setUser( "Alison" , 18 ) ;
```

Optionally, the parameter declaration may be assigned a default value to use if no argument value is passed by the caller. In this case, the caller may pass, or omit, an argument value:

```
static void setUser( string name , int age = 21 ) { statements }
```

```
setUser( "Brenda" , 19 ) ;
setUser( "Christine" ) ;          // age = 21.
```

There are three ways to pass arguments into functions:

- **By Value** – Arguments passed to functions "by value" assign a copy of the original value to the parameter. Changes made to the parameter inside the function do not affect the original.
- **By Reference** – Arguments passed to functions "by reference" assign the address of the memory location of the original value to the parameter. Changes made to the parameter inside the function do affect the original value. Reference arguments must include the C# **ref** keyword in both the function call parentheses and the function definition parameter list.
- **For Output** – Arguments passed to functions "for output" assign the memory location address of the argument to the parameter, to which the function can assign a value. This is similar to passing by reference except that data is passed from the function, rather than to the function. Output arguments must include the C# **out** keyword in both the function call parentheses and the function definition parameter list. This is useful to return more than one value from a function.

Beware

Calling a function without passing required arguments, or arguments of the wrong data type, will cause an error.

Hot tip

Most function calls pass arguments by value rather than by reference or for output.

1. Start a new **C# Console App** and name it "Parameter"
```
Console.Title = "Parameter" ;
```

2. Type these statements to declare two variables
```
double weight ;
string num ;
```

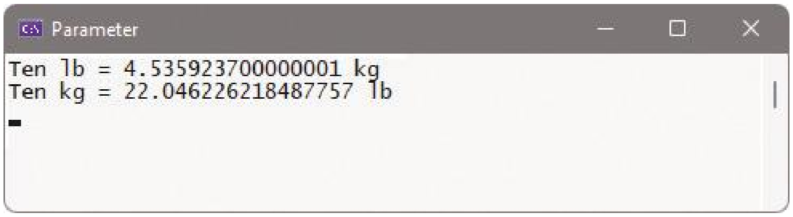
3. Add this function that will output a double value and return a string value
```
static string getWeight( out double theWeight )
{
  theWeight = 10 ;
  return "Ten" ;
}
```

4. Next, add a function that will return a multiplied value of its parameter argument
```
static double lbToKg( double pounds = 5 )
{ return ( pounds * 0.45359237 ) ; }
```

5. Now, add a function that will assign a divided value to its parameter reference argument but return nothing
```
static void kgToLb( ref double weight )
{ weight = ( weight / 0.45359237 ) ; }
```

6. Finally, add statements to pass arguments to each of the functions declared above
```
num = getWeight( out weight ) ;
Console.WriteLine( num +" lb = " + lbToKg(weight) +" kg" ) ;
kgToLb( ref weight ) ;
Console.WriteLine( num + " kg = " + weight + " lb" ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the output featuring arguments passed to method parameters



```
Ten lb = 4.535923700000001 kg
Ten kg = 22.046226218487757 lb
```

Parameter

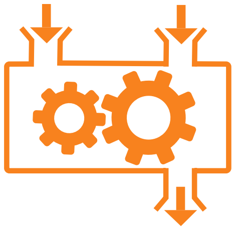
Don't forget

The default parameter value is not used here, as the caller passes in an argument value.

Hot tip

Both the **num** and **weight** variables are initialized by the **getWeight( )** function. The value of **weight** is set in the function, as it is passed by reference. The variable **num** is initialized using the value returned from the function.

Overloading methods



Method "overloading" allows methods of the same name to happily co-exist in the same program, providing their parameters differ in number, data type, or both number and data type. The compiler matches a method call to the correct version of the method by recognizing its parameter number and data types – a process known as "method resolution".

Where a parameter represents the same value as that in another method, the parameters should have the same name. Parameters with the same name should appear in the same position in each method's parameter list, but you should not use **ref** or **out** modifiers to overload methods.

It is useful to create overloaded methods in a custom class when their tasks are similar, yet subtly different. A class is created using the **class** keyword followed by a name of your choice. The methods of the class can then be added as function declarations within **{ }** curly brackets following the class name.

In order to use methods of a custom class, a new instance of the class object must first be created using this syntax:

Class-name instance-name **= new( ) ;**

The methods can then be called by dot-suffixing the method name to the instance name.

Overload

1. Start a new **C# Console App** and name it "Overload"
```
Console.Title = "Overload" ;
```

2. Declare two floating-point variables and an instance of a class named "Compute" (to be created later in Step 7)
```
double num ;
double area ;
Compute size = new( ) ;
```

3. Add statements to initialize the first variable from input
```
Console.Write( "Please Enter Dimension In Feet: " ) ;
num = Convert.ToDouble( Console.ReadLine( ) ) ;
```

4. Now, initialize the second variable by calling a method that accepts only one argument, then displays its value
```
area = size.Zone( num ) ;
Console.WriteLine( "\nCircle:\t\tArea = "+area+" sq.ft." ) ;
```

Remember that the **ReadLine( )** method returns a **string** value – so this must be converted in order to perform arithmetic.

5 Assign a new value to the second variable by calling the function that accepts two arguments, then display its value

```
area = size.Zone( num , num ) ;
Console.WriteLine( "Square:\t\tArea = "+area+" sq.ft." ) ;
```

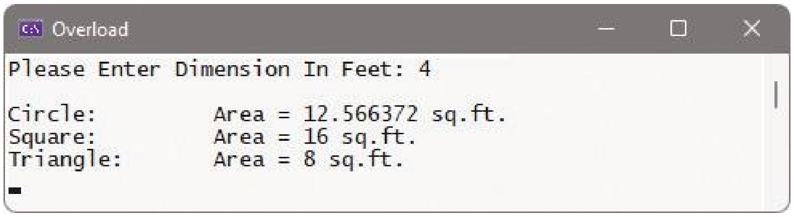
6 Assign a new value to the second variable by calling the function that accepts three arguments, then display its value

```
area = size.Zone( num , num , 'T' ) ;
Console.WriteLine( "Triangle:\tArea = "+area+" sq.ft." ) ;
Console.ReadKey( ) ;
```

7 Finally, add the custom class containing the three methods

```
class Compute
{
  public double Zone( double width )
  {
    double radius = width / 2 ;
    return ( ( radius * radius ) * 3.141593 ) ;
  }

  public double Zone( double width, double height )
  {
    return ( width * height ) ;
  }

  public double Zone( double width, double height , char c )
  {
    return ( ( width / 2 ) * height ) ;
  }
}
```

8 Press **Start** or **F5** to run the application, then enter a number and see the output from the overloaded methods



```
Please Enter Dimension In Feet: 4

Circle:         Area = 12.566372 sq.ft.
Square:         Area = 16 sq.ft.
Triangle:       Area = 8 sq.ft.
_
```

Don't forget

The value passed to the **char c** parameter is never used here – it is included merely to determine which method should be called.

Beware

The class block is a "type declaration", whereas the rest of the code are "top-level statements". Order is important, as top-level statements must precede any type declarations – so the class block must appear at the end of this code.

Refactoring code

Functions can call themselves recursively, to repeatedly execute the statements contained in their function body – much like a loop. As with loops, a recursive function must contain an updater and a conditional test to call itself again, or stop repeating when a condition is met. The syntax of a recursive function looks like this:

```
return-data-type function-name ( parameter-list )
{
        statements-to-be-executed ;
        updater ;
        conditional-test-to-recall-or-exit ;
}
```

The updater will change the value of a passed argument – so subsequent calls will pass the adjusted value back to the function:

Refactor

1. Start a new **C# Console App** and name it "Refactor"

```
Console.Title = "Refactor" ;
```

2. Add this recursive function to return the factorial value of a positive integer argument

```
static int factorial( int num )
{
  int result ;
  if( num == 1 )
  {
    result = 1 ;
  }
  else
  {
    result = ( factorial( num - 1 ) * num ) ;
  }
  return result ;
}
```

3. Next, add a method to display a range of integers and their computed factorial values

```
static void computeFactorials( int num, int max )
{
  while( num <= max )
  {
    Console.Write( "Factorial Of " + num + " : " ) ;
    Console.WriteLine( factorial( num )  ) ;
    num++ ;
  }
}
```

Hot tip

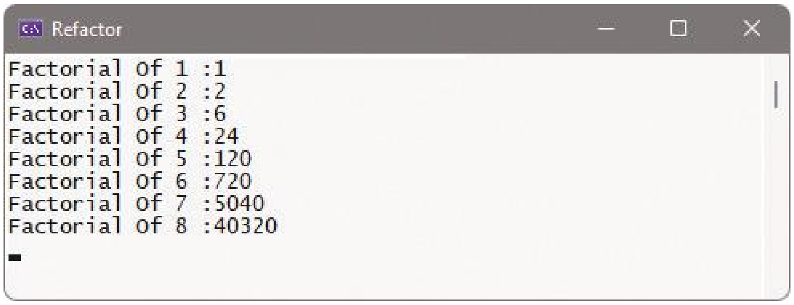
A factorial is the product of multiplying a positive integer by all lesser positive integers.

Hot tip

The two calls to write output in Step 3 could be refactored into a single **WriteLine( )** call for greater efficiency.

4. Now, add statements to pass a range of argument values to be computed for display
```
computeFactorials( 1 , 8 ) ;
Console.ReadKey( ) ;
```

5. Press **Start** or **F5** to run the application to see the output



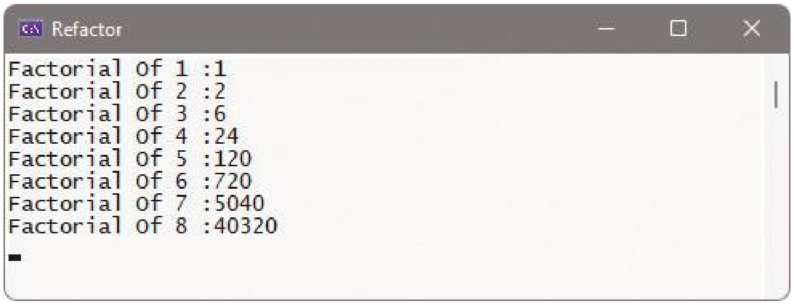
```
Factorial Of 1 :1
Factorial Of 2 :2
Factorial Of 3 :6
Factorial Of 4 :24
Factorial Of 5 :120
Factorial Of 6 :720
Factorial Of 7 :5040
Factorial Of 8 :40320
```

If you accidentally run an infinite recursive function, press the **Ctrl** + **C** keys to terminate the process.

The output lists factorial values (factorial 3 is 3x2x1=6, etc.) but the program can be optimized by refactoring the recursive **factorial( )** method. This method does not need a variable if written with the ternary operator:

6. Edit the **factorial( )** method listed in Step 2 to make it more efficient
```
static int factorial( int num )
{
  return ( num == 1 ) ? 1 : ( factorial( num - 1 ) * num ) ;
}
```

7. Press **Start** or **F5** to run the application once more and see the same output, produced more efficiently



```
Factorial Of 1 :1
Factorial Of 2 :2
Factorial Of 3 :6
Factorial Of 4 :24
Factorial Of 5 :120
Factorial Of 6 :720
Factorial Of 7 :5040
Factorial Of 8 :40320
```

A recursive method generally uses more system resources than a loop – but it can make the code more readable. Modern compilers perform optimization, so it is better to focus on writing readable code.

Summary

- A programmer-specified function is a modular section of code that can be called to execute its statements whenever required.
- It is easier to understand, maintain, re-use, and share modular program code that contains methods.
- A function declaration must at least specify a return data type and the function name.
- Methods not intended for use by other classes must be initialized in memory by the **static** keyword.
- Optionally, a function declaration may include a parameter list.
- Variables declared within a function have local scope, so are not accessible from outside that function.
- Arguments passed by value assign a copy of the original value to the function parameter, so the original will not be affected.
- Arguments passed by reference assign a memory location to the function parameter, so the original value will be affected.
- Arguments passed for output retrieve a value from the function and are useful to return more than one value.
- Method overloading allows methods of the same name to co-exist in the same program if their parameters differ.
- The compiler matches calls to overloaded methods by recognizing their parameter number and data types.
- Parameters representing the same values in overloaded methods should have the same name and the same position.
- The reference **ref** and output **out** modifiers should not be used with overloaded methods as their sole difference.
- Code can be refactored for optimum efficiency by reducing the number of variables and function calls.
- Recursive functions repeatedly execute their statements, so must contain an updater to stop repeating at some point.

6 Handling strings

This chapter demonstrates how to manipulate and format text strings within C# programs.

Discovering string features

The C# **String** class library provides properties and methods that enable a program to easily discover features of a text string. Usefully, the **String.IsNullOrWhiteSpace( )** and **String.IsNullOrEmpty( )** methods can be used to validate user input. These are **static** methods, so are written dot-suffixed to the **String** class name.

Other properties and methods operate on an “instance” of the class, so are written dot-suffixed to a **string** variable name. Instance methods can report **string** size and query its contents. The **Length** property returns an integer that is the number of characters within the string, including spaces. Additionally, the **StartsWith( )**, **EndsWith( )**, and **Contains( )** methods can each accept a string argument to match within the **string** variable value. When a match is found, they return **true**, otherwise they return **false**:

Features

1. Start a new **C# Console App** and name it “Features”
```
Console.Title = "Features" ;
```

2. Type these statements requesting user input to initialize a string variable
```
Console.Write( "Please Enter Text: " ) ;
string text = Console.ReadLine( ) ;
```

3. Next, add a conditional test to ensure the user entered input before hitting the **Enter** key
```
if ( String.IsNullOrWhiteSpace( text )  )
{
  Console.WriteLine( "\nERROR: No Text Found!" ) ;
}
else
{
  // Statements to be inserted here (Steps 4-7).
}
Console.ReadKey( ) ;
```

4. Now, insert statements to report the length of the string
```
Console.WriteLine( "\nThanks. You Entered:\n'"+text+"'" ) ;
Console.WriteLine( "\nText Length: " + text.Length ) ;
```

5. Insert another statement to test how the string begins
```
string query = text.StartsWith( "C#" ) ? "Does" : "Does Not" ;
Console.WriteLine( "Text " + query + " Start With 'C#'" ) ;
```

Hot tip

String class methods can be written using the **String.** class prefix or its **string.** alias prefix – our examples use **String** to differentiate from the **string** data type.

Beware

Note that some of these strings include ‘ single quotes within the outer “ double quote marks.

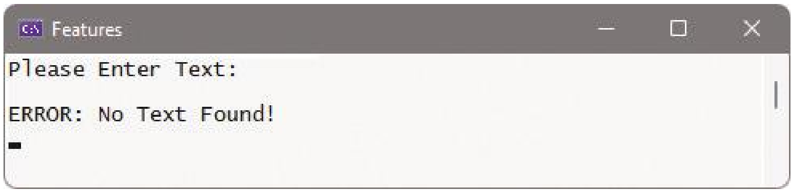
6 Next, insert a statement to test how the string ends

```
query = text.EndsWith( "steps" ) ? "Does" : "Does Not" ;
Console.WriteLine( "Text " + query + " End With 'steps'" ) ;
```

7 Now, insert a statement to test what the string contains

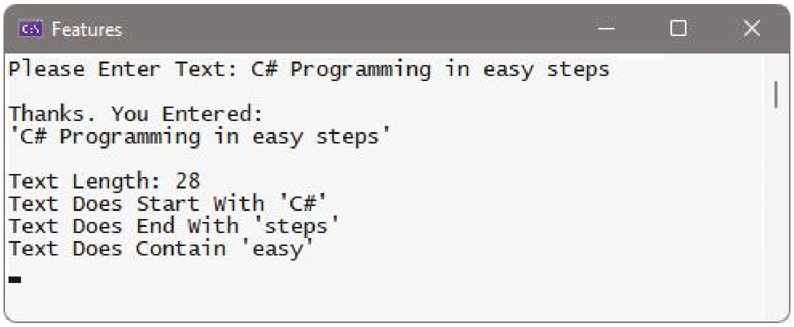
```
query = text.Contains( "easy" ) ? "Does" : "Does Not" ;
Console.WriteLine( "Text " + query + " Contain 'easy'" ) ;
```

8 Press **Start** or **F5** to run the application, and hit **Enter** without input to see the error message

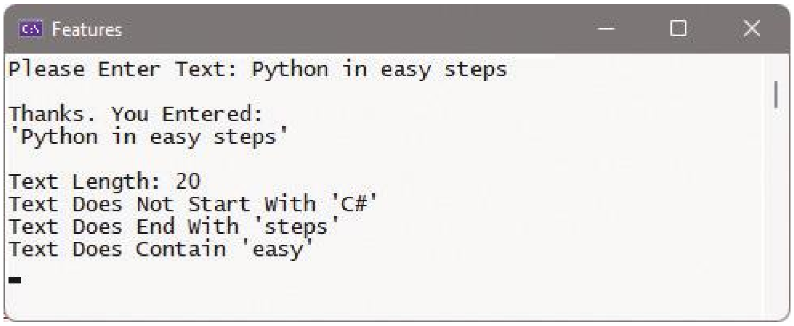


```
Please Enter Text:

ERROR: No Text Found!
```

9 Press **Start** or **F5** to run the application again, type input, then hit **Enter** to discover input string features



```
Please Enter Text: C# Programming in easy steps

Thanks. You Entered:
'C# Programming in easy steps'

Text Length: 28
Text Does Start With 'C#'
Text Does End With 'steps'
Text Does Contain 'easy'
```



```
Please Enter Text: Python in easy steps

Thanks. You Entered:
'Python in easy steps'

Text Length: 20
Text Does Not Start With 'C#'
Text Does End With 'steps'
Text Does Contain 'easy'
```

Don't forget

Notice that the **?:** ternary operator is used here to assign an appropriate **string** value.

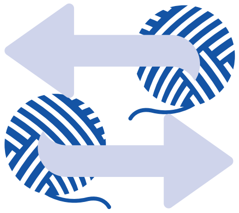
Hot tip

String values are often referred to as "literals", as they comprise characters to be read literally, as text.

Hot tip

Methods with a **String.** (or **string.**) prefix are using the **String** class itself, whereas dot-suffixed methods use an "instance" object of the class – see Chapter 9.

Manipulating strings



The C# **String** class library provides methods that enable a program to manipulate text strings. The **ToLower( )** and **ToUpper( )** methods can be dot-suffixed to a **string** variable to change all characters within the string to lowercase, or to uppercase.

Similarly, the **TrimStart( )**, **TrimEnd( )**, and **Trim( )** methods can be used to remove whitespace or other characters from a **string** at its start, end, or both start and end. By default, these methods will remove whitespace from the string, but you can alternatively specify a character to be removed as an argument to the method.

Conversely, the **PadLeft( )** and **PadRight( )** methods can be used to add whitespace or other characters onto a **string** at its start or end. Their arguments must specify the total character length of the padded string, and a padding character unless default whitespace padding is required. If you want to add padding onto both start and end, the methods can be "chained" as **PadLeft( ).PadRight( )** stating each padding length argument, and character if required:

Manipulate

1. Start a new **C# Console App** and name it "Manipulate"
```
Console.Title = "Manipulate" ;
```

2. Type these statements requesting user input to initialize a string variable
```
Console.Write( "Please Enter Text: " ) ;
string text = Console.ReadLine( ) ;
```

3. Add statements to display the user input string and report its length
```
Console.Write( "\nThanks. You Entered:\n'" + text + "'" ) ;
Console.WriteLine( "\t\tText Length: " + text.Length ) ;
```

4. Remove leading and trailing whitespace, then display the manipulated version and report its length
```
text = text.Trim( ) ;
Console.Write( "\nTrimmed:\t'" + text + "'" ) ;
Console.WriteLine( "\tText Length: " + text.Length ) ;
```

5. Next, add statements to create and display an uppercase version of the trimmed string
```
string upper = text.ToUpper( ) ;
Console.WriteLine( "\nUpperCase:\t'" + upper + "'" ) ;
```

Hot tip

It is a good idea to always use **Trim( )** to remove spaces accidentally added by the user when typing input.

6 Now, create and display a lowercase version of the trimmed string

```
string lower = text.ToLower( ) ;
Console.WriteLine( "LowerCase:\t'" + lower + "'" ) ;
```

7 Then, further manipulate all three strings to add whitespace and character padding

```
upper = upper.PadLeft( 40 ) ;
lower = lower.PadRight( 40 , '#' ) ;
text = text.PadLeft( 30 , '*' ).PadRight( 40 , '*' ) ;
```

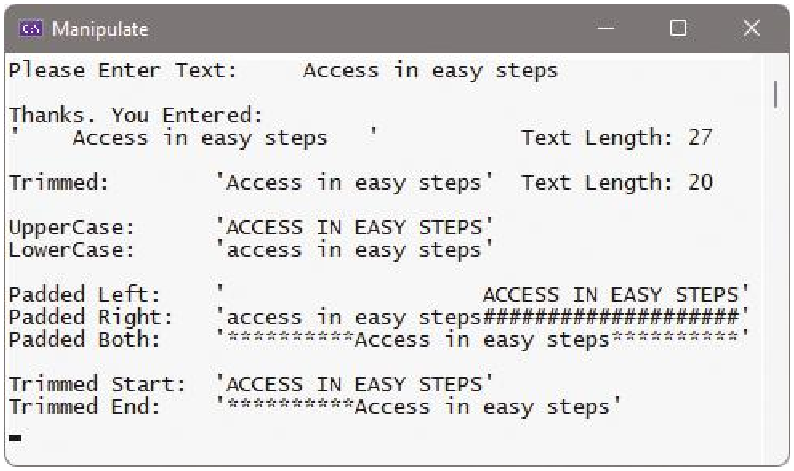
8 Add statements to display all three strings to see the padded whitespace and padded characters

```
Console.WriteLine( "\nPadded Left:\t'" + upper + "'" ) ;
Console.WriteLine( "Padded Right:\t'" + lower + "'" ) ;
Console.WriteLine( "Padded Both:\t'" + text + "'" ) ;
```

9 Finally, add statements to display trimmed versions of two padded strings

```
Console.WriteLine( "\nTrimmed Start:\t'" +
                            upper.TrimStart( ) + "'" ) ;
Console.WriteLine( "Trimmed End:\t'" +
                            text.TrimEnd( '*' ) + "'" ) ;
Console.ReadKey( ) ;
```

10 Press **Start** or **F5** to run the application and enter a string with leading and trailing space to see it manipulated



```
Please Enter Text:     Access in easy steps

Thanks. You Entered:
'     Access in easy steps   '          Text Length: 27

Trimmed:        'Access in easy steps'  Text Length: 20

UpperCase:      'ACCESS IN EASY STEPS'
LowerCase:      'access in easy steps'

Padded Left:    '                    ACCESS IN EASY STEPS'
Padded Right:   'access in easy steps####################'
Padded Both:    '**********Access in easy steps**********'

Trimmed Start:  'ACCESS IN EASY STEPS'
Trimmed End:    '**********Access in easy steps'
```

Don't forget

Here, all three strings are expanded to a total width of 40 characters.

Hot tip

You can also specify multiple characters to be trimmed as a comma-separated list of arguments to the **TrimStart()**, **TrimEnd()**, or **Trim()** methods.

Beware

When padding both left and right you must individually specify by how much to expand the string in each direction – in this case, expanding from 20 to 30 characters left, then from 30 to 40 characters right.

Joining and comparing strings

When the **+** operator is used to concatenate (join) strings in an assignment, the combined strings get stored in the **string** variable. But when used in the **Write( )** or **WriteLine( )** methods, the strings are only combined in output – the variable values are unchanged.

The C# **String** class library provides a **String.Concat( )** method that can be used to join strings as an alternative to using the **+** operator. This method accepts a comma-separated list of **string** arguments to be joined into a single **string**.

Similarly, the **String.Join( )** method can also be used to join strings as an alternative to using the **+** operator. This, too, accepts a comma-separated list of **string** arguments to be joined, but its first argument usefully specifies a separator **string**. It places this separator between each **string** argument in the joined **string**. The separator might be a single space to separate words, or perhaps an HTML tag to separate text, or any other **string** value.

Hot tip

The examples in this book use the **+** operator for concatenation rather than **String.Concat( )** as it provides better readability without any loss of performance.

String comparisons can be made for alphabetic order by specifying two string arguments to the **String.Compare( )** method. This returns an integer denoting whether the alphabetic relationship of the first string to the second is before (**-1**), after (**1**), or equal (**0**). There is also a **CompareTo( )** method that provides the same results, but this can be dot-suffixed onto the first string, and the second string specified as its argument.

As with numeric comparisons, the **==** operator can be used to test for string equality. Alternatively, the **Equals( )** method can be dot-suffixed onto the first string, and the second string specified as its argument. Either will return a **true** or **false** Boolean result.

C#

Joined

1. Start a new **C# Console App** and name it "Joined"
```
Console.Title = "Joined" ;
```

2. Type this statement to create and initialize a **string** array variable simply named "a"
```
string [ ] a = new string[ 3 ] { "Alpha", "Bravo", "Charlie" } ;
```

3. Next, assign a concatenated version of the first two array element values to a **string** variable and display its value
```
string s = String.Concat( a[0] , a[1] ) ;
Console.WriteLine( "Concatenated:\t" + s ) ;
```

4 Assign a joined version of the first two array elements and space separator to the string variable, then display its value

```
s = String.Join( " " , a[0] , a[1] ) ;
Console.WriteLine( "Joined:\t\t" + s ) ;
```

5 Assign a joined version of all three array elements and tag separator to the string variable, then display its value

```
s = String.Join( "<br>" , a ) ;
Console.WriteLine( "\nHTML:\t" + s +"\n" ) ;
```

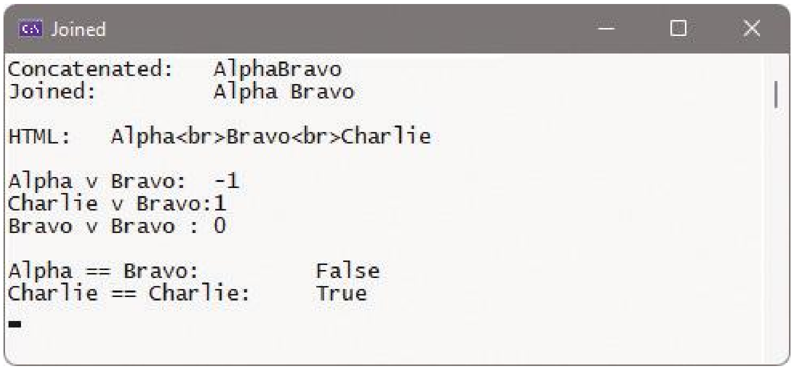
6 Next, add statements to compare, in turn, all three array elements for alphabetic order

```
int num = String.Compare( a[0] , a[1] ) ;
Console.WriteLine( a[0] + " v " + a[1] + ":\t" + num ) ;

num = String.Compare( a[2] , a[1] ) ;
Console.WriteLine( a[2] + " v " + a[1] + ":\t" + num ) ;

num = a[1].CompareTo( a[1] ) ;
Console.WriteLine( a[1]+ " v "+a[1]+":\t" + num+"\n" ) ;
```

7 Finally, test the array element values for equality

```
bool flag = ( a[0] == a[1] ) ;
Console.WriteLine( a[0]+" == "+a[1]+":\t\t" + flag ) ;
flag = a[2].Equals( a[2] ) ;
Console.WriteLine( a[2]+" == "+a[2]+":\t\t" + flag ) ;
Console.ReadKey( ) ;
```

8 Press **Start** or **F5** to run the application to see the joined strings and comparison results



```
Concatenated:   AlphaBravo
Joined:         Alpha Bravo

HTML:   Alpha<br>Bravo<br>Charlie

Alpha v Bravo:  -1
Charlie v Bravo:1
Bravo v Bravo : 0

Alpha == Bravo:         False
Charlie == Charlie:     True
```

Hot tip

Notice how only the array name needs to be specified as the argument to join all three array elements.

Hot tip

You can add or remove the **\t** escape sequences to format the output to your liking.

Don't forget

Two alternative methods are used here to compare strings and to test for equality.

Copying and swapping strings

The C# **String** class library provides a **String.Copy( )** method that can be used to copy one string to another string as an alternative to using the = assignment operator. This method accepts the string to be copied as its argument. Perhaps more usefully, the **CopyTo( )** method can be dot-suffixed onto a **string** variable to copy its contents into a **char** character array. This requires four arguments to specify the index number from which to start copying in the source **string**, the name of the **char** array, the index number at which to start copying in the destination **char** array, and the number of characters to copy.

Other methods can be dot-suffixed to a **string** variable to swap its contents. The **Remove( )** method requires an integer argument to specify the index number at which to begin removing characters from the **string**. This will remove all characters from the specified index position up to the end of the **string**, unless you add a second argument to specify the number of characters to be removed.

Conversely, the **Insert( )** method requires an integer argument to specify the index number at which to begin inserting characters into the **string**, and an argument specifying the **string** to insert. The **Replace( )** method simply requires two **string** arguments to specify a substring to seek within the **string**, and a **string** to replace that substring when a match is found:

Hot tip

Copying with an = assignment produces two **string** variables that contain the same value and reference the same memory location, whereas copying with the **String.Copy( )** method produces two **string** variables that contain the same value but reference different memory locations.

C#

Copied

1. Start a new **C# Console App** and name it "Copied"
```
Console.Title = "Copied" ;
```

2. Type these statements to create and initialize two **string** variables with the names of two Maserati car models
```
string car1 = "Ghibli" ;
string car2 = "GranTurismo" ;
```

3. Next, display the original values contained in each variable
```
Console.Write( "Original:" ) ;
Console.WriteLine( "\tCar 1: "+ car1 +" \t\tCar 2: " + car2 ) ;
```

4. Now, copy the value of the second **string** variable into the first **string** variable and display their modified values
```
car1 = String.Copy( car2 ) ;
Console.Write( "\nCopied:" ) ;
Console.WriteLine( "\tCar 1: "+ car1 + "\tCar 2: " + car2 ) ;
```

5 Initialize an integer variable with the length of the first **string** variable, then create a character array of that length

```
int num = car1.Length ;
char [ ] model = new char [ num ] ;
```

6 Next, copy the first **string** variable into the character array, then display a space-separated list of the element contents

```
car1.CopyTo( 0 , model , 0 , num ) ;
Console.Write( "\nCharacter Array:" ) ;
foreach( char c in model ) { Console.Write( c + " " ) ; }
```

7 Now, remove the end of the first **string** variable, starting from its fifth element, and display the modified value

```
car1 = car1.Remove( 4 ) ;
Console.WriteLine( "\n\nRemoved... \tCar 1: " + car1 ) ;
```

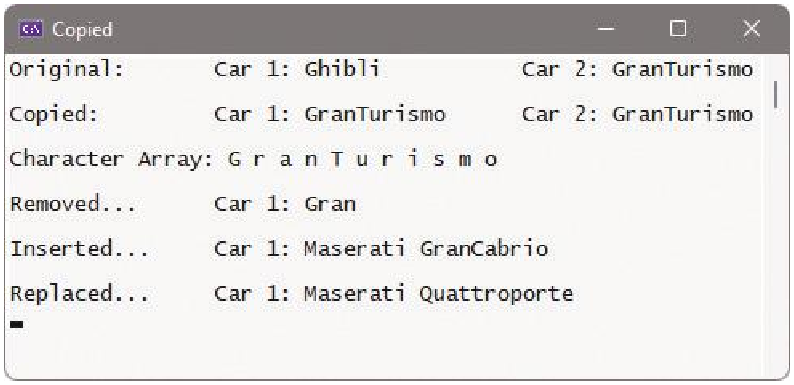
8 Insert two strings into the first **string** variable, at its beginning and end, then display the modified value

```
car1 = car1.Insert( 0 , "Maserati " ) ;
car1 = car1.Insert( 13 , "Cabrio" ) ;
Console.WriteLine( "\nInserted... \tCar 1: " + car1 ) ;
```

9 Finally, replace a substring within the first **string** variable, and once more display the modified value

```
car1 = car1.Replace( "GranCabrio" , "Quattroporte" ) ;
Console.WriteLine( "\nReplaced... \tCar 1: " + car1 ) ;
Console.ReadKey( ) ;
```

10 Press **Start** or **F5** to run the application to see the copied and swapped strings



```
Original:       Car 1: Ghibli          Car 2: GranTurismo
Copied:         Car 1: GranTurismo     Car 2: GranTurismo
Character Array: G r a n T u r i s m o
Removed...      Car 1: Gran
Inserted...     Car 1: Maserati GranCabrio
Replaced...     Car 1: Maserati Quattroporte
```

The **char** array must be of the same size as the **string** to be copied – use the string's **Length** property to specify the size of the **char** array.

The fifth element of an array is at index position 4 in a zero-based index.

Finding substrings

The C# **String** class library provides a number of methods that can be dot-suffixed to a **string** variable to seek a string within a **string** (i.e. a "substring"). The **IndexOf( )** method requires the substring as its argument and, if found, returns the index position of the substring within the searched **string**. Otherwise, it will return **-1**. The **IndexOf( )** method searches forward, from left-to-right, and returns the index position of the first occurrence of the substring. It has a companion **LastIndexOf( )** method that works in just the same way, but searches backward, from right-to-left.

Similarly, there is an **IndexOfAny( )** method and its companion **LastIndexOfAny( )** method that require a character array argument. These seek any character of the specified array within a string, and return the index position of the first occurrence, or **-1** otherwise.

You can also dot-suffix a **Substring( )** method to a **string** variable if you want to extract a copy of a substring from within that **string**. This method requires two arguments to specify the index position at which to begin copying, and the number of characters to copy:

C#

Substring

1. Start a new **C# Console App** and name it "Substring"

```
Console.Title = "Substring" ;
```

2. Add this function to report the result of a substring search

```
static void report( int pos , string sub )
{
  if( pos != -1 )
  { Console.WriteLine( "'" + sub + "' Found At " + pos ) ; }
  else
  { Console.WriteLine( "'" + sub + "' Not Found!" ) ; }
}
```

3. Now, initialize a **string** variable, then display its value and length

```
string text = "My dog is a cute dog" ;
Console.WriteLine( text + "\tLength: " + text.Length ) ;
```

4. Next, request user input to initialize another string variable with a substring to seek

```
Console.WriteLine( "\nPlease Enter A Substring To Seek: " ) ;
string sub = Console.ReadLine( ) ;
```

Don't forget

The position reported when a substring is found is the index position of the first character of that substring within the searched string – whether searching forward or backward.

5 Create a character array and copy the entire substring value into the character array

```
char [ ] arr = new char[ sub.Length ] ;
sub.CopyTo( 0 , arr , 0 , sub.Length ) ;
```

6 Then, seek the first occurrence of the substring and call the function to report the search result

```
int pos = text.IndexOf( sub ) ;
report( pos , sub ) ;
```

7 Now, seek the last occurrence of the substring and report the result

```
pos = text.LastIndexOf( sub ) ;
report( pos , sub ) ;
```

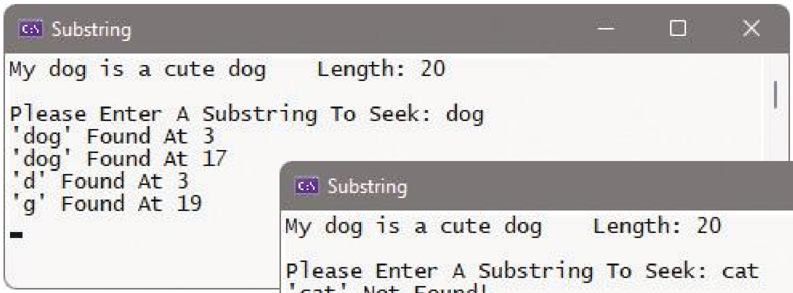
8 Next, seek the first occurrence of any character of the substring and report the result

```
pos = text.IndexOfAny( arr ) ;
report( pos , text.Substring( pos , 1 ) ) ;
```

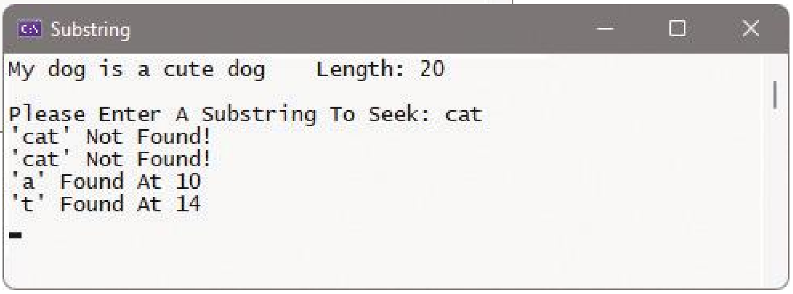
9 Finally, seek the last occurrence of any character of the substring and report the result

```
pos = text.LastIndexOfAny( arr ) ;
report( pos , text.Substring( pos , 1 ) ) ;
Console.ReadKey( ) ;
```

10 Press **Start** or **F5** to run the application, then enter a substring to seek and see the reported results


Substring

```
My dog is a cute dog    Length: 20

Please Enter A Substring To Seek: dog
'dog' Found At 3
'dog' Found At 17
'd' Found At 3
'g' Found At 19
_
```


Substring

```
My dog is a cute dog    Length: 20

Please Enter A Substring To Seek: cat
'cat' Not Found!
'cat' Not Found!
'a' Found At 10
't' Found At 14
_
```

Don't forget

The **CopyTo( )** method is described and demonstrated in the previous example on pages 76-77.

Hot tip

Notice that a copy of the located character is extracted from the original **string** using the **Substring( )** method for output in the report.

Formatting strings

In C# every object has a **ToString( )** method that returns a string representation of that object. This means that the **ToString( )** method can be dot-suffixed to any numeric variable to get its value represented in **string** format. The **ToString( )** method can, optionally, accept a string argument to specify how the **string** version should be formatted. Common specifiers are listed below:

Specifier	Returns
G	General
F	Fixed Point
N	Number (with comma-separated thousands)
C	Currency (prevailing on your system)
P	Percentage (multiplied by 100)
00.0000	Zero Padding

Hot tip

The C# specifications provide further string formatting specifier options in addition to the commonly used ones shown here.

The C# **String** class library provides a **String.Format( )** method that uses the same specifiers to also produce values represented in **string** format. This requires a **string** argument to specify the format, which may include "placeholders", and an argument list to be substituted for each placeholder in the output **string**.

Each placeholder can be numbered sequentially (starting at zero) within **{ }** braces, to match the list position of the substitution argument. For example, **{0}** matches the first argument in the substitution argument list, **{1}** matches the second, and so on. Additionally, the number may be followed by a **:** colon and one of the format specifiers in the table above – to specify how the substituted value should appear in its **string** – for example, **{0:G}**.

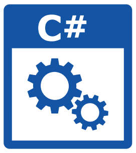
Hot tip

Multiple placeholders in one string can be numbered alike – if you want each one to be substituted by the same list argument value.

Data strings that contain separators, such as a comma-separated list of data retrieved from a database query, can be easily broken into individual items of data by dot-suffixing the **Split( )** method to a **string** variable. The individual strings can be assigned to a **string** array variable, and could then be formatted in output by the **String.Format( )** method if required.

1 Start a new **C# Console App** and name it "Format"

```
Console.Title = "Format" ;
```



Format

2 Initialize a numeric variable, then use the **ToString()** method to output its value as a currency **string** format

```
double sum = 2500 ;
Console.WriteLine( "Currency String: " + sum.ToString( "C" ) ) ;
```

3 Next, use the **String.Format()** method to output the same numeric value in various common **string** formats

```
Console.Write( String.Format( "\nGeneral:\t {0:G}" , sum )  ) ;
Console.Write( String.Format( "\nFixed Point:\t {0:F}", sum )  ) ;
Console.Write( String.Format( "\nNumber:\t\t {0:N}" , sum )  ) ;
Console.Write( String.Format( "\nCurrency:\t {0:C}" , sum )  ) ;
```

Hot tip

The specifier letters may be written as either uppercase or lowercase.

4 Now, reduce the numeric value, then output it in a percentage **string** format and with padded zeros

```
sum /= 1000 ;
Console.Write( String.Format( "\nPercentage:\t {0:P}", sum )  ) ;
Console.Write
( String.Format( "\nZero Padded:\t {0:00.0000} \n", sum )  ) ;
```

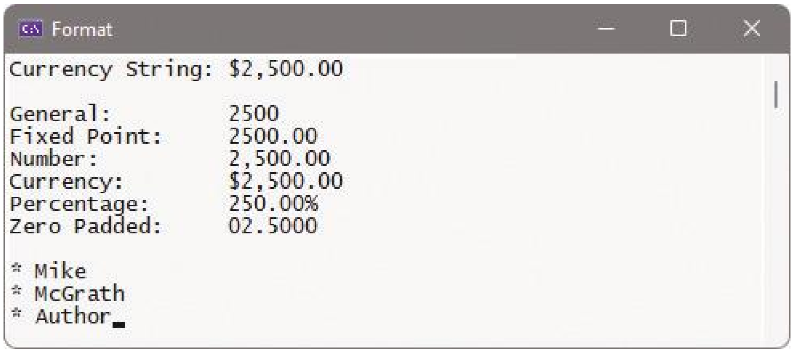
5 Then, create a comma-separated **string** list and split it into individual elements of a **string** array variable for output

```
string data = "Mike,McGrath,Author" ;
string [ ] items = data.Split( ',' ) ;
foreach ( string item in items )
{ Console.Write( String.Format( "\n* {0}" , item )  ) ; }
Console.ReadKey( ) ;
```

Beware

The argument to the **Split( )** method must be a single **char** character – enclosed in <u>single</u> quotes.

6 Press **Start** or **F5** to see the formatted string output



```
Currency String: $2,500.00

General:        2500
Fixed Point:    2500.00
Number:         2,500.00
Currency:       $2,500.00
Percentage:     250.00%
Zero Padded:    02.5000

* Mike
* McGrath
* Author_
```

Don't forget

You can use the + concatenation operator for formatting, but many prefer **String.Format( )** for easily readable code.

Formatting date strings

The C# **DateTime** class library provides methods and properties to easily work with dates and times. Its **Now** property returns a **DateTime** object of the current local date and time of your system. Alternatively, you can create a **DateTime** object using the **new** keyword, and specifying a date and time as a comma-separated list of arguments. Many methods and properties can be dot-suffixed to any **DateTime** object to specify its format, or to extract specific components of the date or time, or to modify its value. For example, the **DayOfWeek** property supplies the day name, the **ToShortString( )** method supplies the date in number form, and the **AddYears( )** method can modify the year component.

The **String.Format( )** method, introduced in the previous example on pages 80-81, also has these specifiers that can be used to determine the format of dates and times in output:

Specifier	Returns	Example
d	Short date	**7/4/2023**
D	Long date	**Tuesday, July 4, 2023**
t	Short time	**8:15 AM**
T	Long time	**8:15:30 AM**
f	Full datetime	**Tuesday, July 4, 2023 8:15 AM**
F	Full+seconds	**Tuesday, July 4, 2023 8:15:30 AM**
M	Month+day	**July 4**
Y	Month+year	**July 2023**
dd	Day number	**04**
dddd	Day name	**Tuesday**
HH	Hour 0-23	**08**
mm	Minute	**15**
ss	Second	**30**
YYYY	Year	**2023**

Don't forget

IntelliSense is an intelligent code completion tool that is built into Visual Studio. You can use IntelliSense to choose methods and properties – for example, type **DateTime.Now.** and select from the pop-up list that appears.

Hot tip

The C# specifications provide further date and time formatting specifier options in addition to the commonly used ones shown here.

1. Start a new **C# Console App** and name it "DateFormat"
```
Console.Title = "DateFormat" ;
```

DateFormat

2. Initialize a **DateTime** object and display its value
```
DateTime now = DateTime.Now ;
Console.Write( "Current Date And Time: "+ now ) ;
```

3. Next, display specific components of the **DateTime** object
```
Console.Write( "\nDay Name: "+ now.DayOfWeek ) ;
Console.Write( "\nDate Only: "+ now.ToShortDateString( ) ) ;
Console.Write( "\nTime Only: "+ now.ToShortTimeString( ) ) ;
```

4. Modify the **DateTime** object and display its new value
```
now = now.AddYears( 4 ) ;
Console.Write( "\n\nFuture Date: "+ now ) ;
```

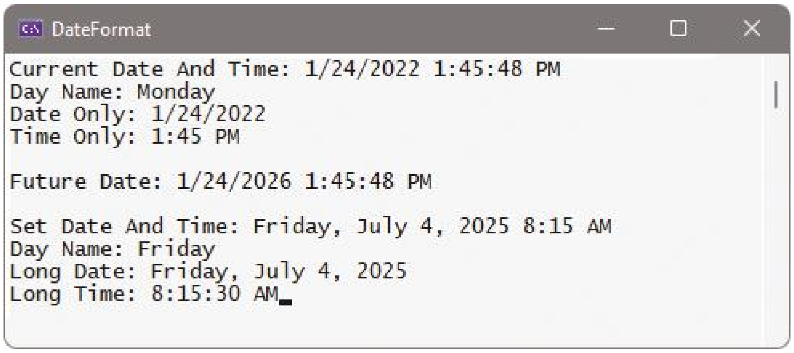
5. Now, create a new **DateTime** object and display its value
```
DateTime dt = new DateTime( 2025, 7, 4, 8, 15, 30 ) ;
Console.Write( "\n\nSet Date And Time: {0:f}", dt ) ;
```

6. Display specific components of the new **DateTime** object
```
Console.Write( "\nDay Name: {0:dddd}", dt ) ;
Console.Write( "\nLong Date: {0:D}", dt ) ;
Console.Write( "\nLong Time: {0:T}", dt ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the formatted date and time output

Hot tip

You need only specify the date components when creating a new **DateTime** object – the time will automatically be set to 12:00 AM (midnight).


DateFormat
```
Current Date And Time: 1/24/2022 1:45:48 PM
Day Name: Monday
Date Only: 1/24/2022
Time Only: 1:45 PM

Future Date: 1/24/2026 1:45:48 PM

Set Date And Time: Friday, July 4, 2025 8:15 AM
Day Name: Friday
Long Date: Friday, July 4, 2025
Long Time: 8:15:30 AM_
```

Summary

- Class library **static** methods, such as **String.IsNullOrEmpty( )**, are dot-suffixed onto the **String** class library name.
- String values can be joined using the **String.Concat( )** and **String.Join( )** methods, or by using the **+** concatenation operator.
- String comparisons can be made for alphabetic order using the **String.Compare( )** method.
- A value can be copied from one **string** variable to another using the **String.Copy( )** method, or the **=** assignment operator.
- The **String.Format( )** method can be used to format both numeric and date strings in output.
- The **DateTime** class library provides methods and properties, such as **Now**, to work with date and time strings.
- Instance methods and properties, such as **Contains( )** and **Length**, are dot-suffixed onto **string** variable names.
- The character case of a **string** value can be changed using the **ToLower( )** and **ToUpper( )** methods.
- Whitespace or other characters can be removed from a **string** value using the **TrimStart( )**, **TrimEnd( )**, and **Trim( )** methods.
- Whitespace or other characters can be added to a **string** value using the **PadLeft( )** and **PadRight( )** methods.
- String comparisons can be made using the **CompareTo( )** and **Equals( )** methods, or the **==** equality operator.
- A **string** value can be copied from a **string** variable into a **char** array using the **CopyTo( )** method.
- The contents of a string variable can be swapped using the **Remove( )**, **Insert( )**, and **Replace( )** methods.
- A substring can be sought using **IndexOf( )**, **LastIndexOf( )**, **IndexOfAny( )**, **LastIndexOfAny( )**, and **Substring( )** methods.
- Every object has a **ToString( )** method that returns a string representation, which can be formatted in output.

7 Accessing files

This chapter demonstrates how C# programs can store data in text files, and retrieve data from text files.

Writing a file

The C# **System.IO.File** class library provides methods that enable a program to easily write text into a file on your computer.

The **File.WriteAllText( )** method simply requires two arguments to specify the text file path and the string to write. Backslash characters in the path string must be escaped to avoid an error.

If you would like to ensure the specified text file does not already exist, you can first test for its existence by specifying its path as the argument to the **File.Exists( )** method. This will return **true** when the file is found, otherwise it will return **false**.

In order to ensure the text file was written successfully, it is worthwhile wrapping the **File.WriteAllText( )** call in a **try**-**catch** block. A statement confirming success can be included in the **try** part of the block, and a statement to advise of failure can be included in the **catch** part of the block:

WriteText

1. Start a new **C# Console App** and name it "WriteText"

```
Console.Title = "WriteText" ;
```

2. Add these statements to initialize two variables – inserting your own user name where indicated in the path

```
// Edit the line below to include your own user name.
string path = "C:\\Users\\username\\Desktop\\poem.txt" ;

string poem = "\r\n\tI never saw a man who looked" ;
poem += "\r\n\tWith such a wistful eye" ;
poem += "\r\n\tUpon that little tent of blue" ;
poem += "\r\n\tWhich prisoners call the sky" ;
```

3. Next, add a statement to test if a file already exists of the specified path and filename

```
if( File.Exists( path )  )
{
  Console.WriteLine( "File Already Exists: " + path ) ;
}
else
{
  // Statements to be inserted here (Step 4).
}
Console.ReadKey( ) ;
```

Don't forget

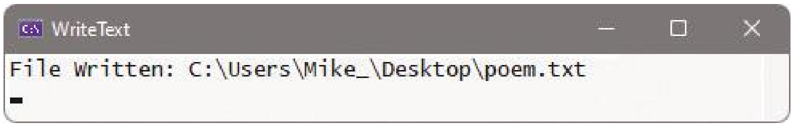
The **\r\n\t** escape sequence is a carriage return, a newline, and a tab.

4 Now, insert statements that attempt to write a text file and confirm success, or advise of failure

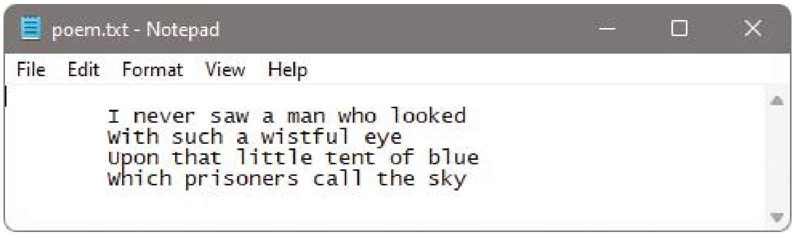
```
try
{
  File.WriteAllText( path , poem ) ;
  Console.WriteLine( "File Written: " + path ) ;
}
catch( Exception error )
{
  Console.WriteLine( error.Message ) ;
}
```

Catching exceptions with the **try-catch** block is fully described in Chapter 8, dealing with problem solving – see pages 106-107.

5 Press **Start** or **F5** to run the application and see a text file written on your Desktop

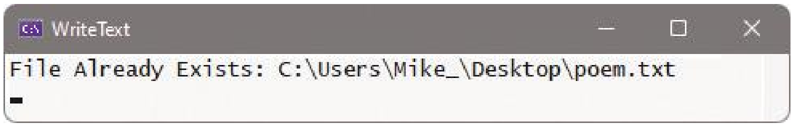


```
File Written: C:\Users\Mike_\Desktop\poem.txt
```



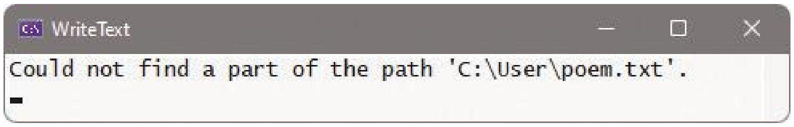
```
        I never saw a man who looked
        With such a wistful eye
        Upon that little tent of blue
        Which prisoners call the sky
```

6 Run the app again – to see that the file already exists



```
File Already Exists: C:\Users\Mike_\Desktop\poem.txt
```

7 Edit the path to a non-existing location, then run the app once more to see the error message



```
Could not find a part of the path 'C:\User\poem.txt'.
```

Appending to a file

The C# **System.IO.File** class library provides a **WriteAllLines( )** method that can be used to write the contents of a **string** array to a text file, with each element appearing on a separate line, and a **try-catch** block can ensure the text file was written successfully.

Additionally, the **System.IO.File** class has an **AppendAllText( )** method, which can be used to add text to an existing file, and the **File.Exists( )** method can ensure that the file first exists:

AppendText

The **\r\n** character return, newline escape sequences can be omitted from the **string** array, as the **WriteAllLines( )** method automatically writes each element on new lines.

1. Start a new **C# Console App** and name it "AppendText"

```
Console.Title = "AppendText" ;
```

2. Add these statements to initialize three variables – inserting your own user name where indicated in the path

```
string path = "C:\\Users\\username\\Desktop\\oscar.txt" ;

string [ ] poem = new string [ ]
{
  "\tIn Debtors' Yard the stones are hard" ,
  "\tAnd the dripping wall is high"
} ;

string attrib = "\r\n\tThe Ballad Of Reading Gaol " ;
attrib += "(Oscar Wilde 1898)" ;
```

3. Next, add a statement to test if a file already exists of the specified path and filename

```
if( File.Exists( path ) )
{
  // Statements to be inserted here (Step 4).
}
else
{
  // Statements to be inserted here (Step 5).
}
Console.ReadKey( ) ;
```

4. Now, insert statements that attempt to append text if the file already exists, and advise of success

```
  File.AppendAllText( path , attrib ) ;
  Console.WriteLine( "Appended To File: " +  path ) ;
```

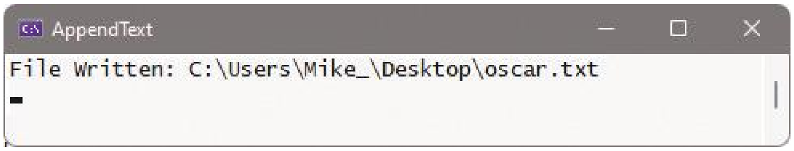
5 Then, insert statements that attempt to write a text file and confirm success, or advise of failure

```
try
{
  File.WriteAllLines( path , poem ) ;
  Console.WriteLine( "File Written: " + path ) ;
}
catch( Exception error )
{
  Console.WriteLine( error.Message ) ;
}
```

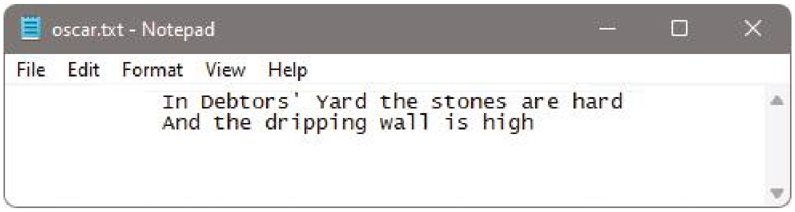
Hot tip

As with **WriteAllText( )**, the **WriteAllLines( )** method requires the text file path and **string** to write as its arguments.

6 Press **Start** or **F5** to run the application and see a text file written on your Desktop, or see an error message



```
File Written: C:\Users\Mike_\Desktop\oscar.txt
```

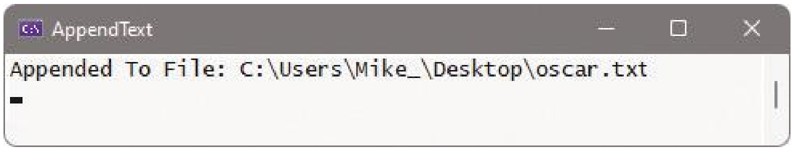


```
        In Debtors' Yard the stones are hard
        And the dripping wall is high
```

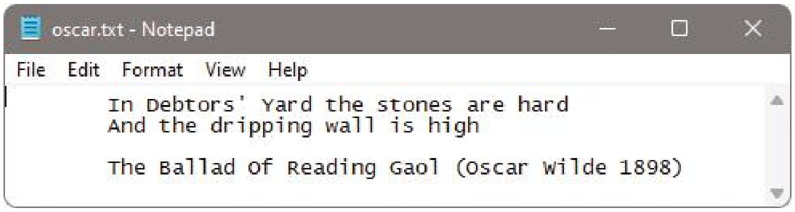
Hot tip

There is also an **AppendAllLines( )** method that can be used to add the contents of a **string** array to a file.

7 Run the application once more and see a confirmation appear and see text now get appended to the file



```
Appended To File: C:\Users\Mike_\Desktop\oscar.txt
```



```
     In Debtors' Yard the stones are hard
     And the dripping wall is high

     The Ballad Of Reading Gaol (Oscar Wilde 1898)
```

Beware

After its first run, this application will append text each time it is run.

Reading text and lines

The C# **System.IO.File** class library provides a **ReadAllText( )** method that can be used to read text from an existing file and assign its entire contents to a **string** variable. The **File.Exists( )** method can ensure that the text file first exists, and a **try-catch** block can ensure the file was read successfully.

Additionally, the **System.IO.File** class has a **ReadAllLines( )** method, which can be used to assign each line of a text file to an individual element of a **string** array:

ReadText

1. Start a new **C# Console App** and name it "ReadText"

```
Console.Title = "ReadText" ;
```

2. Add this statement to initialize a variable – inserting your own user name where indicated in the path

```
string path = "C:\\Users\\username\\Desktop\\word.txt" ;
```

3. Next, add a statement to test if a file already exists of the specified path and filename

```
if( File.Exists( path ) )
{
  // Statements to be inserted here (Step 4).
}
else
{
  Console.WriteLine( "File Not Found: " + path ) ;
}
Console.ReadKey( ) ;
```

4. Now, insert a statement to display a message if the file cannot be read successfully

```
  try
  {
    // Statements to be inserted here (Steps 5 and 6).
  }
  catch( Exception error )
  {
    Console.WriteLine( error.Message ) ;
  }
```

Beware

When a text file is read into a **string** array, the array's **Length** property, which returns the number of its elements, will represent the number of lines read – including empty lines!

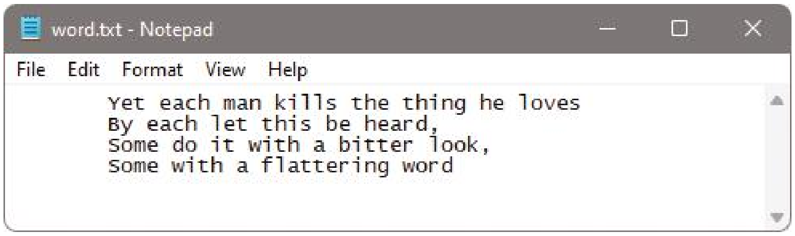
5 Then, insert statements to assign the text file contents to a variable and display its value

```
string text = File.ReadAllText( path ) ;
Console.WriteLine( "File Read: " + path + "\n" ) ;
Console.WriteLine( text + "\n" ) ;
```

6 Finally, insert statements to assign the text file contents to an array variable and display each element with a counter

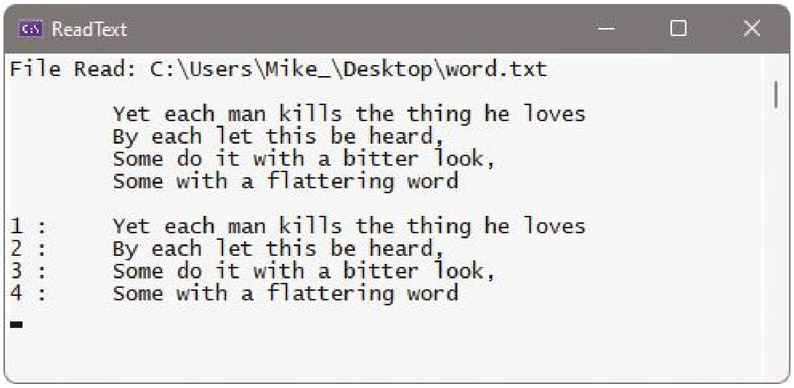
```
string [ ] lines = File.ReadAllLines( path ) ;
int num = 1 ;
foreach( string line in lines )
{
  Console.WriteLine( num + " : " + line ) ;
  num++ ;
}
```

7 Open a text editor such as Notepad, and create a multi-line text file named "word.txt" on your Desktop



```
	Yet each man kills the thing he loves
	By each let this be heard,
	Some do it with a bitter look,
	Some with a flattering word
```

8 Press **Start** or **F5** to run the application and see a text file read from your Desktop, or see an error message



```
File Read: C:\Users\Mike_\Desktop\word.txt

	Yet each man kills the thing he loves
	By each let this be heard,
	Some do it with a bitter look,
	Some with a flattering word

1 :	Yet each man kills the thing he loves
2 :	By each let this be heard,
3 :	Some do it with a bitter look,
4 :	Some with a flattering word
```

Don't forget

The text file's invisible newline and tab characters are preserved when read by the **ReadAllText( )** method, but only the invisible tab characters are preserved when it is read by the **ReadAllLines( )** method.

Hot tip

Remove the text files' Read permission on your system and run the application again to see an "Access to path denied" message.

Streaming lines

The **File.WriteAllText( )** and **File.AppendAllText( )** methods are simple to use but provide few options. The **System.IO.StreamWriter** class also provides object methods that can be used to write text to a file, and these provide more options. An instance of a **StreamWriter** object must first be created using the **new** keyword, and the text file path specified as an argument to its "constructor" method, like this:

```
StreamWriter name = new StreamWriter( path ) ;
```

Methods such as **Write( )** and **WriteLine( )** can then be dot-suffixed to the instance name. A **StreamWriter** instance is disposable, so its statements are best enclosed within a **using** construct to ensure it is removed from memory upon completion, like this:

```
using ( StreamWriter name = new StreamWriter( path ) )
{
  // Statements.
}
```

The **using** construct can also be wrapped in a **try-catch** block, to ensure the text file was written successfully.

Optionally, the **StreamWriter** constructor can accept a second argument of **true** to append to existing text:

Don't forget

The **using** keyword has different meanings according to its context.

1. Start a new **C# Console App** and name it "WriteStream"

```
Console.Title = "WriteStream" ;
```

WriteStream

2. Add these statements to initialize three variables – inserting your own user name where indicated in the path

```
string path = "C:\\Users\\username\\Desktop\\robert.txt" ;

string[ ] poem = new string [ ]
{
  "\tThis truth finds honest Tam o' Shanter" ,
  "\tAs he from Ayr one night did canter" ,
  "\tOld Ayr, which never a town surpasses" ,
  "\tFor honest men and bonny lasses."
} ;

string attrib = "\n\tTam o' Shanter (Robert Burns 1790)" ;
```

3 Next, add a statement to display a message if the file cannot be written successfully

```
try
{
 // Statements to be inserted here (Steps 4 and 5).
}
catch( Exception error )
{ Console.WriteLine( error.Message ) ; }
Console.ReadKey( ) ;
```

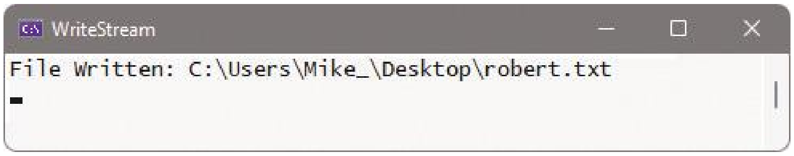
4 Now, insert statements that attempt to write the contents of the variable array into a text file

```
  using ( StreamWriter writer = new StreamWriter( path )  )
  {
    foreach( string line in poem )
    { writer.WriteLine( line ) ; }
  }
```

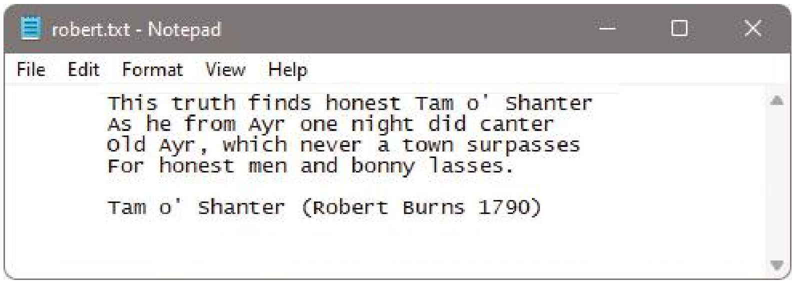
5 Finally, insert statements that attempt to append the contents of the regular variable into a text file

```
  using ( StreamWriter writer =
                                  new StreamWriter( path, true )  )
  {
    writer.WriteLine( attrib ) ;
    Console.WriteLine( "File Written: " + path ) ; }
  }
```

6 Press **Start** or **F5** to run the application and see a text file written on your Desktop, or see an error message



```
File Written: C:\Users\Mike_\Desktop\robert.txt
```



```
	This truth finds honest Tam o' Shanter
	As he from Ayr one night did canter
	Old Ayr, which never a town surpasses
	For honest men and bonny lasses.

	Tam o' Shanter (Robert Burns 1790)
```

Hot tip

The **using** construct ensures the **StreamWriter** is disposed of when its operations complete, so the same name can be used for the new object.

Hot tip

Try writing this file into a non-existent folder to see an "Access to path denied" message.

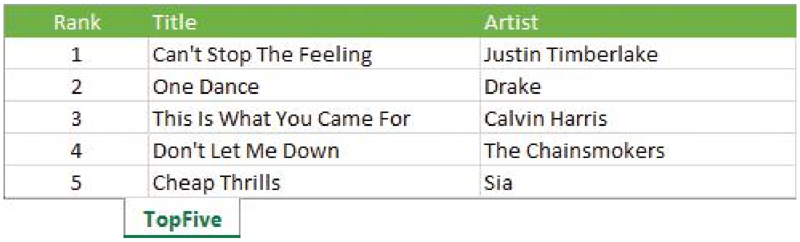
Manipulating input and output

The **System.IO.StreamReader** class provides object methods that can be used to read text from a file. An instance of a **StreamReader** object must first be created using the **new** keyword, and the text file path specified as an argument to its "constructor":

StreamReader *name* = new StreamReader(*path*) ;

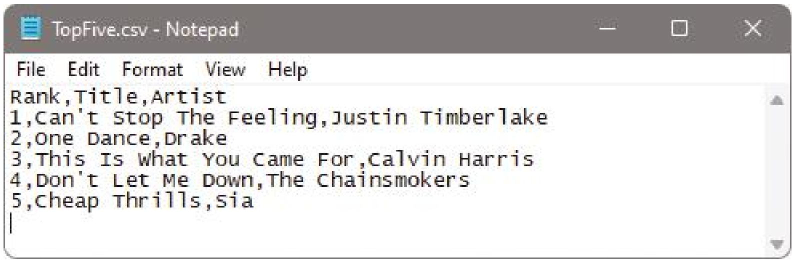
Methods such as **Read( )** and **ReadLine( )** can then be dot-suffixed to the instance name. A **StreamReader** instance is disposable, so its operation statements are best enclosed within a **using** construct to ensure it is removed from memory when the operation has completed. The **using** construct can also be wrapped in a **try-catch** block, to ensure the text file was read successfully.

Text read by a **StreamReader** object can be manipulated to change its format before output, for example, to manipulate cells exported from an Excel spreadsheet as comma-separated values:



Rank	Title	Artist
1	Can't Stop The Feeling	Justin Timberlake
2	One Dance	Drake
3	This Is What You Came For	Calvin Harris
4	Don't Let Me Down	The Chainsmokers
5	Cheap Thrills	Sia

TopFive



```
Rank,Title,Artist
1,Can't Stop The Feeling,Justin Timberlake
2,One Dance,Drake
3,This Is What You Came For,Calvin Harris
4,Don't Let Me Down,The Chainsmokers
5,Cheap Thrills,Sia
```

C#

ReadStream

1. Start a new **C# Console App** and name it "ReadStream"
   Console.Title = "ReadStream" ;

2. Add a statement to initialize a variable – inserting your own user name where indicated in the path
   string path = "C:\\Users\\*username*\\Desktop\\TopFive.csv" ;

3 Next, add a statement to display a message if the file cannot be read successfully

```
try
{
  // Statements to be inserted here (Step 4).
}
catch( Exception error )
{ Console.WriteLine( error.Message ) ; }
Console.ReadKey( ) ;
```

4 Now, insert statements that attempt to read the contents of the text file into a variable, line by line

```
  using ( StreamReader reader = new StreamReader( path ) )
  {
    string line ;
    while( ( line = reader.ReadLine( ) ) != null )
    {
      // Statements to be inserted here (Steps 5-6).
    }
  }
```

When the **ReadLine( )** method reaches the end of the file being read, it returns a **null** value.

5 Insert statements to modify the case of the column headers and amend an artist name

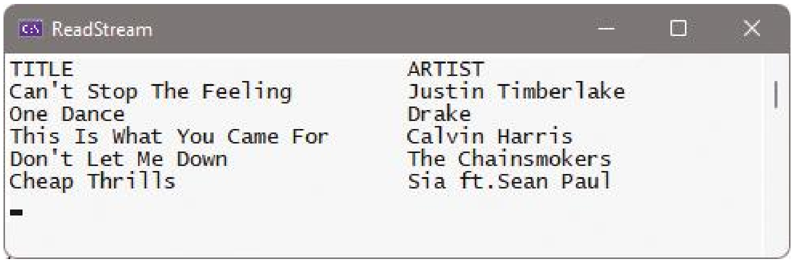
```
    if( line.IndexOf( "Rank" ) != -1 ) line = line.ToUpper( ) ;
    if( line.IndexOf( "Sia" ) != -1 ) line += " ft.Sean Paul" ;
```

6 Finally, insert statements that display the content of just two columns, formatted for alignment

```
    string [ ] sub = line.Split( ',' ) ;
    line = String.Format( "{0,-30}{1,-20}", sub[1], sub[2] ) ;
    Console.WriteLine( line ) ;
```

The **String.Format( )** placeholders specify character widths as negative numbers to align strings to the left.

7 Press **Start** or **F5** to run the application and see the manipulated output from the file content



ReadStream

```
TITLE                         ARTIST
Can't Stop The Feeling        Justin Timberlake
One Dance                     Drake
This Is What You Came For     Calvin Harris
Don't Let Me Down             The Chainsmokers
Cheap Thrills                 Sia ft.Sean Paul
```

Summary

- The **System.IO.File** class provides methods to easily read or write text files on your computer.
- The **System.IO.File.WriteAllText( )** method requires two arguments to specify a file path and content to write there.
- The **System.IO.File.Exists( )** method will determine if the file specified as its argument already exists.
- It is recommended all read or write operations be wrapped in a **try-catch** block to report when an attempted operation fails.
- The **System.IO.File.WriteAllLines( )** method can write the element content of a **string** array as separate lines of a file.
- The **System.IO.File.AppendAllText( )** method requires two arguments to specify a file path and content to append there.
- The **System.IO.File.ReadAllText( )** method can be used to assign the entire content of a text file to a **string** variable.
- The **System.IO.File.ReadAllLines( )** method can assign individual lines of a text file to elements of a **string** array.
- The **System.IO.StreamWriter** class provides object methods to write text files on your computer.
- The **System.IO.StreamReader** class provides object methods to read from text files on your computer.
- An instance of a **StreamReader** object or **StreamWriter** object is created using the **new** keyword and by specifying a file path within the parentheses of its constructor.
- A **StreamReader** object has **Read( )** and **ReadLine( )** methods that can be dot-suffixed to an instance name.
- A **StreamWriter** object has **Write( )** and **WriteLine( )** methods that can be dot-suffixed to an instance name.
- All **StreamReader** and **StreamWriter** objects are disposable, so should each be enclosed in a **using** construct.

8 Solving problems

This chapter demonstrates how to detect and manage errors in C# programs.

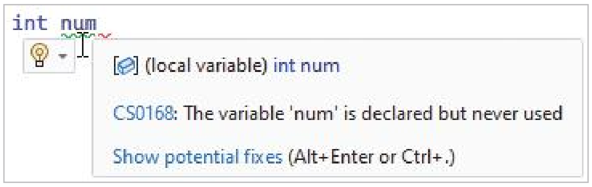
Detecting real-time errors

As you type code in the **Code Editor** window, the Visual Studio IDE is constantly monitoring your code for possible errors. It examines the code you type, and provides real-time feedback of possible errors by adding a wavy underline to questionable code.

Warnings of potential problems are indicated by a green wavy underline. These are not critical and will not prevent execution of the application. A rollover tooltip explains the warning:

1. First, type this variable declaration in the **Code Editor** `int num`

2. A green wavy line appears below the **num** variable name. Place the cursor over the green wavy underline to discover that the warning is merely indicating a potential problem as the variable has not yet been assigned a value



```
int num
```

(local variable) int num

CS0168: The variable 'num' is declared but never used

Show potential fixes (Alt+Enter or Ctrl+.)

Hot tip

Visual Studio 2022 provides live code analysis, which displays a light bulb icon when the compiler detects an issue with your code, and has a suggestion of how to fix that issue.

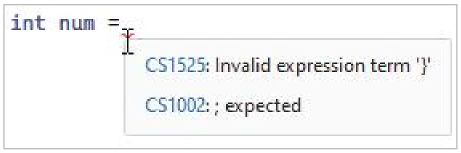
Beware

Warnings can be accepted but errors must be corrected.

Errors are indicated by a red wavy underline. Unlike warnings, these are critical and will prevent execution of the application:

1. Type this variable declaration in the **Code Editor** `int num =`

2. Place the cursor over the red wavy underline to discover that the error is due to a missing value in the expression



```
int num =
```

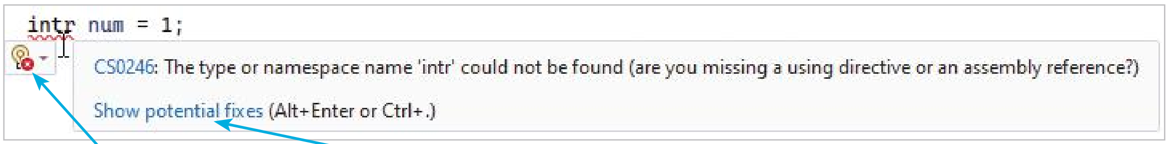
CS1525: Invalid expression term '}'

CS1002: ; expected

Real-time error detection in the Visual Studio IDE is a fantastic tool to help prevent errors when you are writing code. It not only indicates errors, but can even provide a list of correction options:

1. Now, type this variable declaration in the **Code Editor**
   intr num = 1 ;

2. A red wavy line appears below the **intr** variable type. Place the cursor over the red wavy underline to discover that the error is due to an unknown type specification

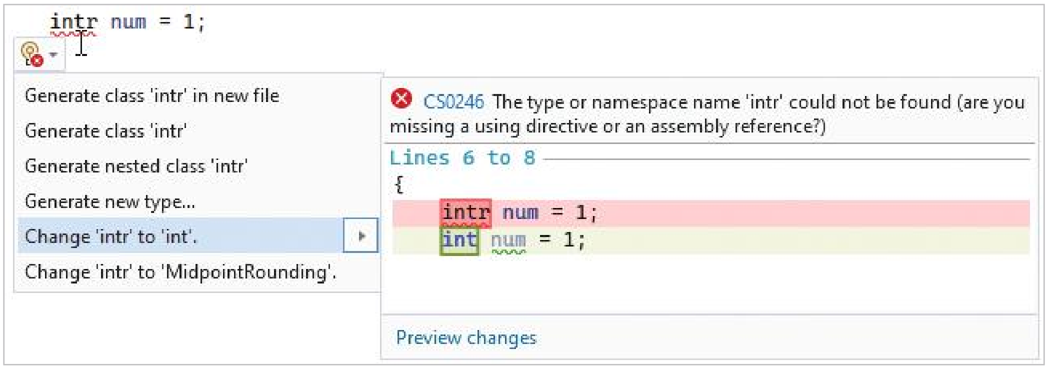


```
intr num = 1;
```

CS0246: The type or namespace name 'intr' could not be found (are you missing a using directive or an assembly reference?)

Show potential fixes (Alt+Enter or Ctrl+.)

3. Click the light bulb icon, or click the **Show potential fixes** link, to see a list of error correction options



```
intr num = 1;
```

Generate class 'intr' in new file
Generate class 'intr'
Generate nested class 'intr'
Generate new type...
Change 'intr' to 'int'.
Change 'intr' to 'MidpointRounding'.

CS0246 The type or namespace name 'intr' could not be found (are you missing a using directive or an assembly reference?)

```
Lines 6 to 8
{
    intr num = 1;
    int num = 1;
```

4. If this error is simply a spelling error for the **int** data type, select the **Change 'intr' to 'int'** option – see your code get instantly corrected accordingly. Other correction options allow you to create a new data type if that is what you require

Fixing compile-time errors

While syntax errors like those on page 99 can be detected by the **Code Editor** in real time, other errors that employ correct syntax cannot be detected until the code is compiled. Compile errors are typically errors of logic, and they cause the execution to halt when an “exception” occurs. For example, when incompatible data types appear in an expression, an **InvalidCastException** occurs and execution stops immediately:

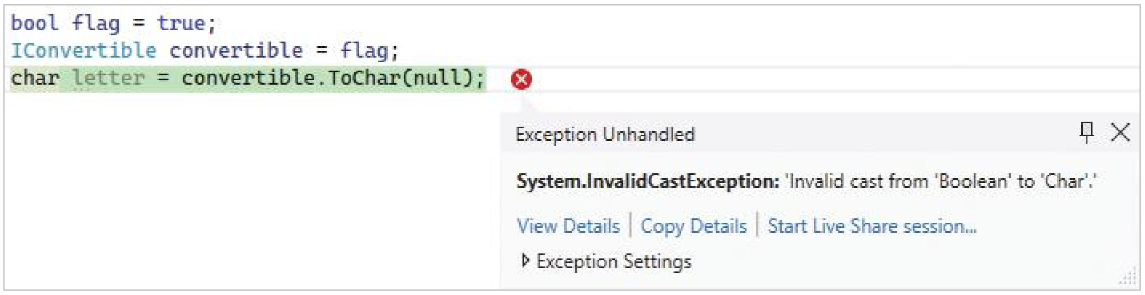
1. Type the following lines into the **Code Editor**

```
bool flag = true ;
IConvertible convertible = flag ;
char letter = convertible.ToChar( null ) ;
```

2. Press **Start** or **F5** to run the application and see execution is soon halted. The line causing the exception becomes highlighted in the **Code Editor**, and an **Exception Assistant** pop-up window appears with a list of possible solutions

Hot tip

The **IConvertible** interface provides methods that convert a value to a CLR type, but it cannot meaningfully convert a **bool** to **char**.



```
bool flag = true;
IConvertible convertible = flag;
char letter = convertible.ToChar(null);
```

Exception Unhandled

System.InvalidCastException: 'Invalid cast from 'Boolean' to 'Char'.'

View Details | Copy Details | Start Live Share session...

Exception Settings

To fix this **InvalidCastException**, the code would need amending so both values are of compatible data types.

The cause of other compile errors may be less obvious without some further investigation – for example, when a loop that is reading array elements attempts to address an element index that does not exist, causing an **IndexOutOfRangeException**.

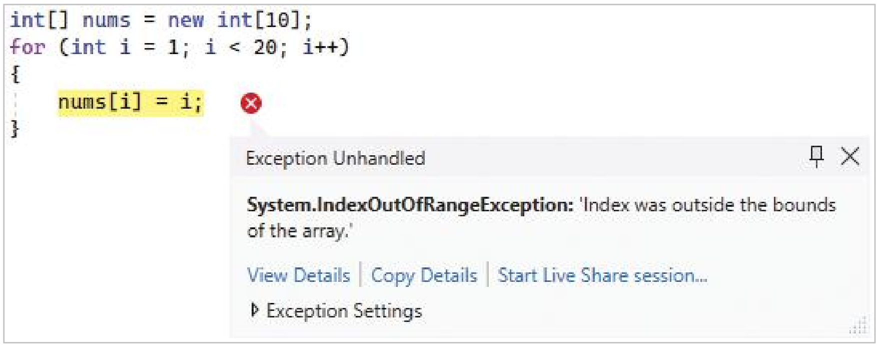
Hot tip

You can click on the **View Detail** link for more error information.

Execution halts immediately, so it is useful to examine the counter value to identify the precise iteration causing the compile error.

1. In the **Code Editor**, type the following variable array declaration of 10 elements (0-9), and a loop
```
int [ ] nums = new int [ 10 ] ;
for ( int i = 1 ; i < 20 ; i++ ) { nums [ i ] = i ; }
```

2. Press **Start** or **F5** to run the application and see execution is soon halted. The code causing the exception becomes highlighted in the **Code Editor**, and an **Exception Assistant** pop-up window appears with a list of possible solutions



```
int[] nums = new int[10];
for (int i = 1; i < 20; i++)
{
    nums[i] = i;
}
```

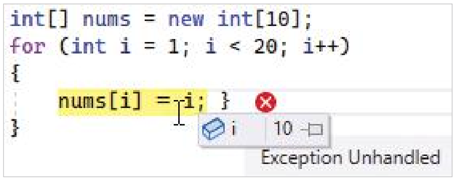
Exception Unhandled

System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'

View Details | Copy Details | Start Live Share session...

Exception Settings

3. Place the cursor over the assignment to the array variable to see a pop-up appear displaying its current value



```
int[] nums = new int[10];
for (int i = 1; i < 20; i++)
{
    nums[i] = i; }
}
```

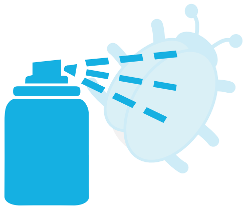
i 10

Exception Unhandled

It's now clear that execution halted when the loop attempted to address **nums[10]** – beyond the bounds of last element **nums[9]**. To fix this **IndexOutOfRangeException**, the code would need amending to end the loop after 10 iterations.

Another common error is the **FileNotFoundException** that occurs when a file is missing or its path name is incorrect.

Debugging code



C#

Debug

It is sometimes useful to closely examine the progression of a program by watching its execution line by line to locate any bugs.

Progress is controlled by clicking the **Step Into** button on the **Debug** Menu Bar to move through the program one line at a time. When you begin debugging, you can open a **Watch** window to monitor the value of particular variables as execution proceeds:

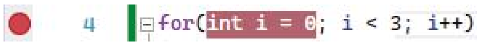
1. Type the following code in the **Code Editor**

```
int pass = 0 ;
int unit = 2 ;

for ( int i = 0 ; i < 3 ; i++ )
{
  pass = ( pass + 1 ) ;
  unit = square( unit ) ;
}
```

2. Now, add this simple arithmetic function

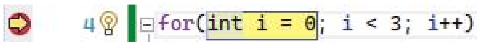
```
static int square( int num )
{
  return ( num * num ) ;
}
```

3. In the **Code Editor**, click in the gray margin against the start of the loop – to set a debug starting "breakpoint"



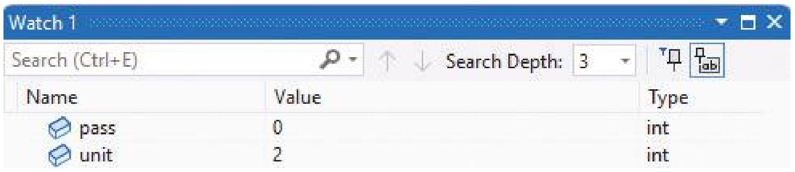
```
4   for(int i = 0; i < 3; i++)
```

4. Click the **Start** button to begin debugging and see the code run to the breakpoint then halt



```
4   for(int i = 0; i < 3; i++)
```

5. Click **Debug**, **Windows**, **Watch**, **Watch1** on the Menu Bar to launch a **Watch** window, then enter the variable names into its **Name** column



Watch 1

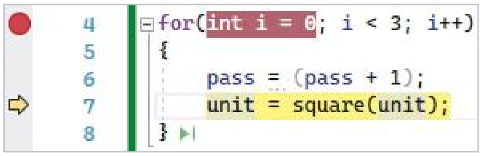
Search (Ctrl+E)   Search Depth: 3

Name	Value	Type
pass	0	int
unit	2	int

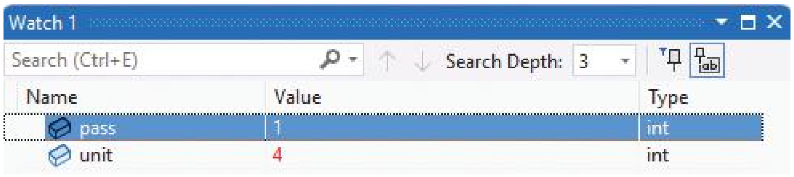
If you can't see the **Step Into** button, right-click on the Menu Bar and select **Debug** to add the debugging buttons.

You can click the **Stop Debugging** button at any time to return to **Code Editor** mode.

6 Repeatedly click the **Step Into** button to progress through each line of the loop and through the **square( )** function one time to see the variable values change in the **Watch** window

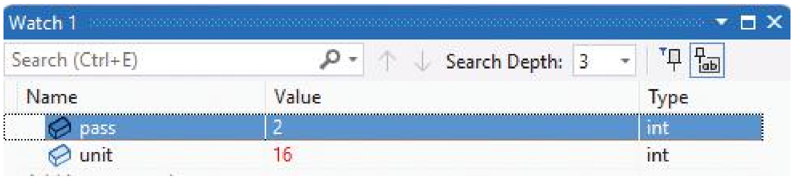


```
for(int i = 0; i < 3; i++)
{
    pass = (pass + 1);
    unit = square(unit);
}
```



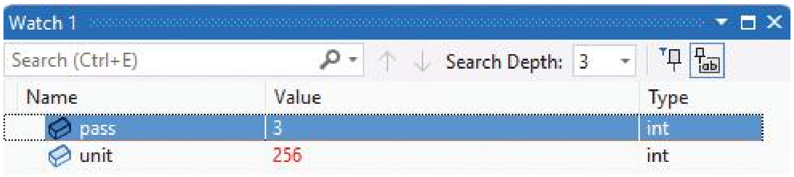
Name	Value	Type
pass	1	int
unit	4	int

7 Repeatedly click the **Step Into** button to progress through each line of the loop and through the **square( )** function a second time and see the variable values change again



Name	Value	Type
pass	2	int
unit	16	int

8 Now, repeatedly click the **Step Over** button to progress through each line of the loop and call the **square( )** function without stepping through the function



Name	Value	Type
pass	3	int
unit	256	int

9 Click the **Step Over** button to move to the end of the program and see the debugger automatically close, returning to regular **Code Editor** mode

10 Click the red dot you added in the margin to remove the breakpoint

Hot tip

The **Step Out** button is used to return to the function caller when you are stepping through lines of a called function.

Setting breakpoints

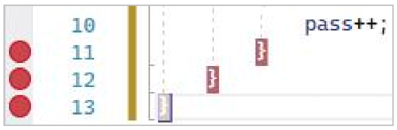
In all but the smallest of programs, stepping through each line is very tedious when debugging. Instead, you can quickly reach the part you wish to examine by setting multiple breakpoints to halt execution on particular lines. Setting one or more breakpoints is useful to help you understand how certain C# code constructs work – such as the nested loop construct shown here:

Breakpoints

1. Type this code to create three nested loops that each increment a counter variable within each loop, and a total pass iteration counter in the innermost loop

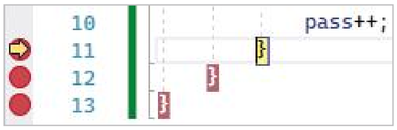
```
int i , j , k ;
int pass = 0 ;

for ( i = 1 ; i < 4 ; i++ )
{
  for ( j = 1 ; j < 4 ; j++ )
  {
    for ( k = 1 ; k < 4 ; k++ )
    {
      pass++ ;
    }
  }
}
```

2. Click in the gray margin against each line containing the closing brace of each loop to set three breakpoints – a red dot will appear in the margin and each closing brace is highlighted to indicate the breakpoints are set



```
10            pass++;
11 ●        }
12 ●      }
13 ●    }
```

3. Click the **Start** button and see the application run to the first breakpoint it meets

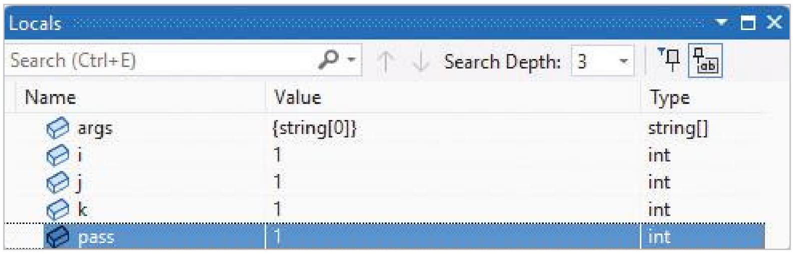


```
10            pass++;
11 ➩        }
12 ●      }
13 ●    }
```

4. Click **Debug**, **Windows**, **Locals** to launch the **Locals** window and notice the current value of each variable

Hot tip

Yellow arrows indicate the current position. Click on the red dot to cancel a breakpoint.

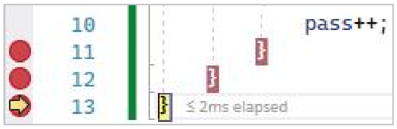


Name	Value	Type
args	{string[0]}	string[]
i	1	int
j	1	int
k	1	int
pass	1	int

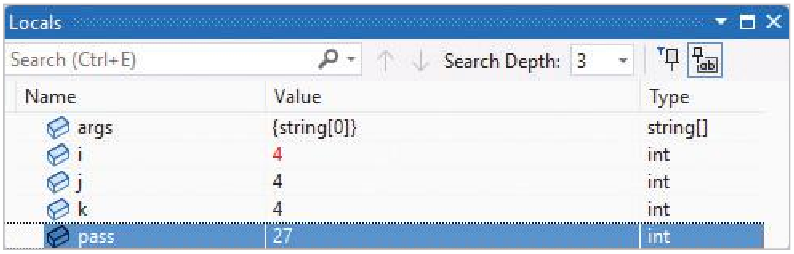
The **Locals** window shows all variables in current scope as the program proceeds.

5 Watch the variable values change as you repeatedly click the **Start** (**Continue**) button to move to each successive breakpoint until you reach the third outer loop breakpoint

6 Repeatedly click **Step Into** until you reach the last iteration of the outermost loop to see the final values



```
            pass++;
        }
    }
}  ≤ 2ms elapsed
```



Name	Value	Type
args	{string[0]}	string[]
i	4	int
j	4	int
k	4	int
pass	27	int

At the end of the program, each counter variable has been incremented beyond the upper limit set in the **for** statements, to exit each loop, and there has been a total of 27 iterations (3x3x3).

7 Click the **Start** button once more to run to the first breakpoint, then click **Debug**, **Windows**, **Immediate** to launch the **Immediate** window

8 In the **Immediate** window, type **i = 3** and hit **Enter**, then use the **Step Into** button to step through each line of just the final complete outer loop

Any code you type into the **Immediate** window is dynamically applied to the application being debugged, but does not change its code.

Catching runtime errors

When you are able to predict potential runtime errors by considering all eventualities, you can provide code to handle each **Exception** class error that may arise – by adding a **try-catch** construct. Your program can supply information to the user about the error, should you wish to do so, then proceed normally:

ErrorHandling

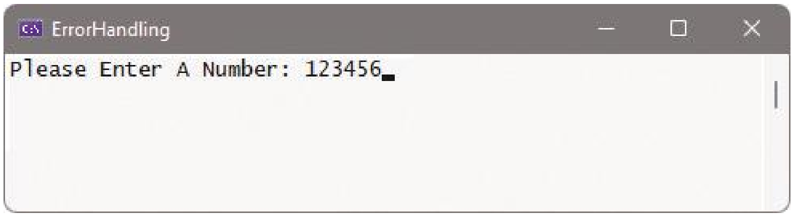
1. Add this program code to request user input of two numeric values for addition, then display their sum total

```
Console.Write( "Please Enter A Number: " ) ;
double num1 = Convert.ToInt16( Console.ReadLine( ) ) ;

Console.Write( "Now Enter Another Number: " ) ;
double num2 = Convert.ToInt16( Console.ReadLine( ) ) ;

Console.WriteLine( "Total: " + ( num1 + num2 ) ) ;
```

Beware

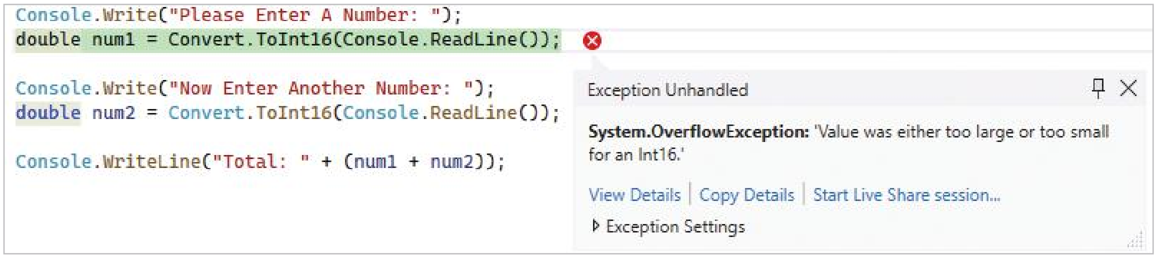
An **Int16** is a 16-bit integer within the range -32,768 to +32,767 – whereas an **Int32** is a 32-bit integer within -2,147,483,648 to +2,147,483,647.

2. Press **Start** or **F5** to run the application, then enter any six-figure integer and hit **Enter**



```
Please Enter A Number: 123456
```

3. The compiler reports an **OverflowException** error



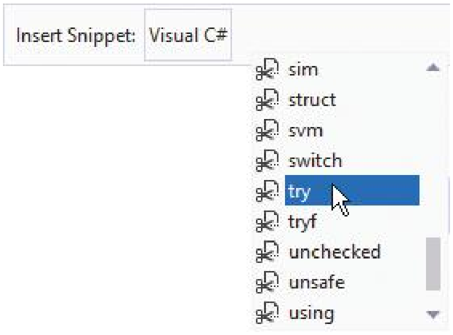
```
Console.Write("Please Enter A Number: ");
double num1 = Convert.ToInt16(Console.ReadLine());

Console.Write("Now Enter Another Number: ");
double num2 = Convert.ToInt16(Console.ReadLine());

Console.WriteLine("Total: " + (num1 + num2));
```

Exception Unhandled

System.OverflowException: 'Value was either too large or too small for an Int16.'

4. Click the **Stop Debugging** button so you can edit the code

5 Drag the mouse to highlight all statements in Step 1 on the previous page, then right-click on the highlighted area and choose **Snippet**, **Insert Snippet:**, **Visual C#**, **try**

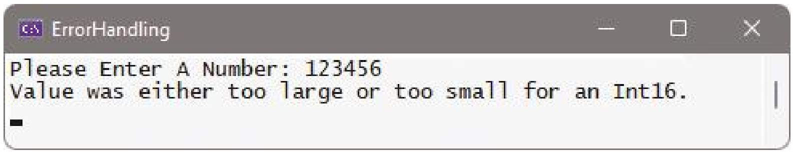


6 A **try-catch** construct is added to the code, enclosing the highlighted statements within the **try** block

7 Edit the default **catch** block to display an error message

```
catch ( Exception error )
{ Console.WriteLine( error.Message ) ; }
```

8 Run the application, then enter any six-figure integer and hit **Enter** to now see a default explanatory error message



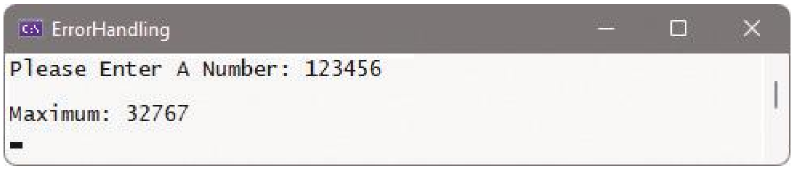
```
Please Enter A Number: 123456
Value was either too large or too small for an Int16.
```

You can provide your own error messages to handle a specific **Exception** by stating its type in the **catch( )** parentheses:

9 Edit the default **catch** block to display a custom message

```
catch ( OverflowException )
{ Console.WriteLine( "\nMaximum: " + Int16.MaxValue ) ; }
```

10 Run the application, then enter any six-figure integer and hit **Enter** to now see a custom explanatory error message



```
Please Enter A Number: 123456

Maximum: 32767
```

Hot tip

The **Insert Snippet** feature provides lots of useful pieces of code to paste into the **Code Editor** – take some time to explore its contents.

Hot tip

Each **Exception** has several methods and properties. For example, use **error.GetType()** to see the type of **Exception**.

Hot tip

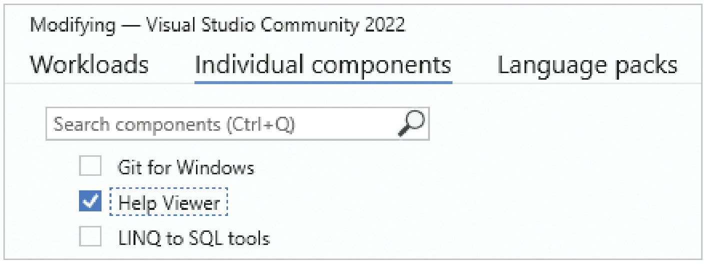
You can add multiple **catch** blocks after the **try** block, to handle different types of **Exception**.

Getting help

The Visual Studio **Help** system provides an extensive source of reference for many programming languages. You can choose to install a Help library on your computer for the C# programming language so you can easily refer to it at any time, but the Help Viewer must first be installed in Visual Studio:

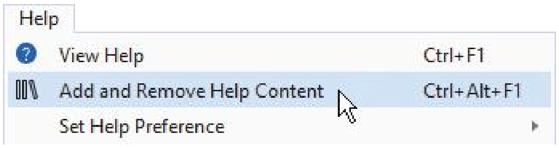
1. Click **Visual Studio Installer** on the Windows Start menu, then select the **Installed** menu item and click the **Modify** button on the Installer dialog

2. Select the **Individual components** tab, check the **Help Viewer** option, then click the **Modify** button at the bottom right of this dialog to install the Help Viewer


Modifying — Visual Studio Community 2022

- [ ] Git for Windows
- [x] Help Viewer
- [ ] LINQ to SQL tools

3. On the Visual Studio Menu Bar, click **Help**, **Add and Remove Help Content** to open the **Help Viewer**



4. On the **Manage Content** tab, expand **Recommended Documentation**, then choose the **Add** link in the **Action** column for the **Visual C#** library

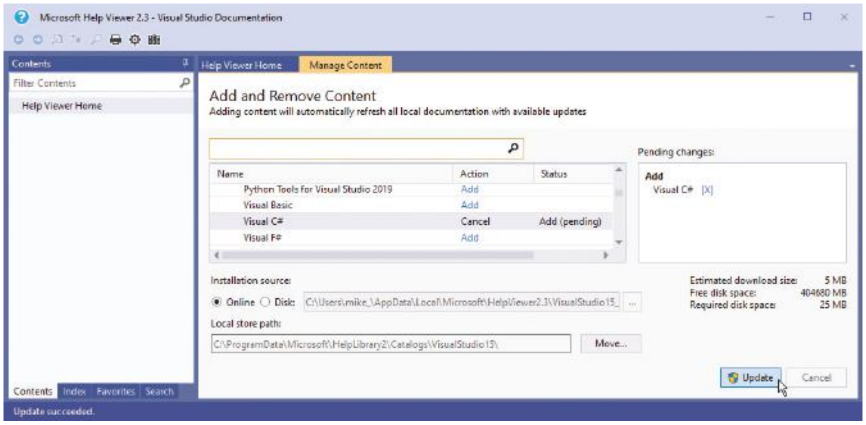
5. When your selection is added to the **Pending changes** list, click the **Update** button to download that library

Don't forget

The **Help Viewer** may not be installed by default when you install Visual Studio unless you select it from the Individual components list.

Hot tip

The **Help Viewer** allows you to download **Help** libraries for offline use, check for available updates, and seek help from installed **Help** libraries.


Microsoft Help Viewer 2.3 - Visual Studio Documentation

Add and Remove Content
Adding content will automatically refresh all local documentation with available updates

Name	Action	Status
Python Tools for Visual Studio 2019	Add	
Visual Basic	Add	
Visual C#	Cancel	Add (pending)
Visual F#	Add	

Pending changes:
Add
Visual C# [X]

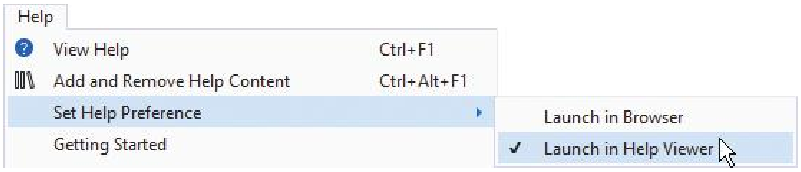
Installation source:
Online ○ Disk: C:\Users\mike_\AppData\Local\Microsoft\HelpViewer2.3\VisualStudio15_

Local store path:
C:\ProgramData\Microsoft\HelpLibrary2\Catalogs\VisualStudio15\

Estimated download size:	5 MB
Free disk space:	404680 MB
Required disk space:	25 MB

Update succeeded.

6 On the Menu Bar, click **Help**, **Set Help Preference**, **Launch in Help Viewer** to use installed library



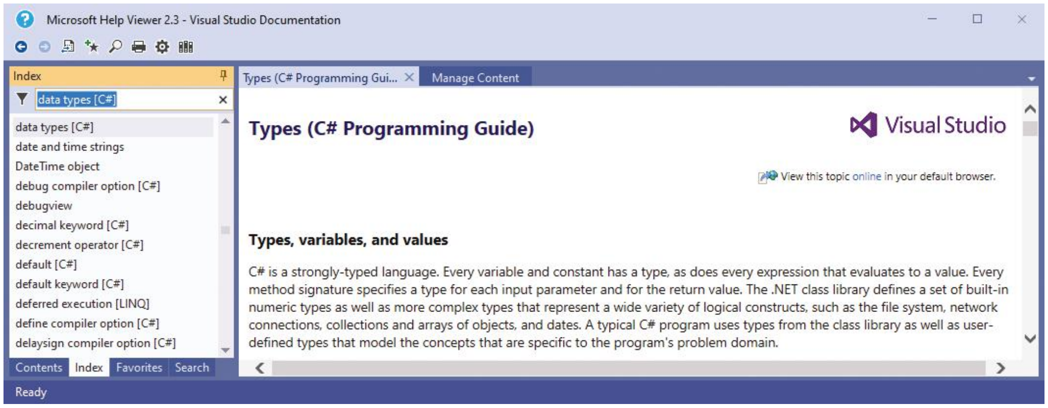
7 Next, click **Help**, **View Help** to launch **Help Viewer**

8 Now, select the **Index** tab in the left-hand pane

9 Type the item for help, such as "data types", in the **Help Viewer** Search box, then hit **Enter** to see the results

Hot tip

You can **Set Help Preference** to **Launch in Browser** if you want to search online help without installing libraries, but local help is often more convenient.


Microsoft Help Viewer 2.3 - Visual Studio Documentation

data types [C#]
date and time strings
DateTime object
debug compiler option [C#]
debugview
decimal keyword [C#]
decrement operator [C#]
default [C#]
default keyword [C#]
deferred execution [LINQ]
define compiler option [C#]
delaysign compiler option [C#]

Types (C# Programming Guide)

View this topic online in your default browser.

Types, variables, and values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

Ready

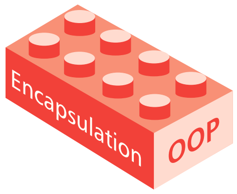
Summary

- The **Code Editor** constantly monitors your code to provide real-time error detection.
- Warnings are not critical and are indicated by a green wavy underline – whereas errors are critical and are indicated by a red wavy underline.
- A light bulb icon beside a red wavy underline indicates that a list of potential fixes is available.
- Typically, real-time errors are errors of syntax, and compile errors are errors of logic.
- When a compile error occurs in **Debug Mode**, execution stops and the **Exception Assistant** offers a list of possible fixes.
- In **Debug Mode** you can discover the current value of any variable simply by placing the cursor over the variable name.
- When debugging code, the **Step Into** button lets you walk through a program one line at a time.
- The **Step Over** button lets you bypass the lines of a called method, and the **Step Out** button lets you return to the line where that method is called.
- Variable values can be monitored as a program proceeds, using the **Watch** window or the **Locals** window.
- Breakpoints halt the execution of a program to allow examination of the part of the program where they are set.
- In **Debug Mode**, code can be dynamically applied using the **Immediate** window.
- Runtime errors occur when the user action has not been anticipated by the programmer.
- A **try-catch** block can be used to handle anticipated exceptions.
- The **Help** library system provides extensive reference sources for both offline and online assistance.

9 Creating objects

This chapter demonstrates encapsulation, inheritance, and polymorphism – the three principles of C# Object Oriented Programming.

Encapsulating data



A class is a data structure that can contain both variables and methods in a single entity. These are collectively known as its “members”, and the variables are also known as its “properties”.

If a class is not declared as **static** it can be used to create “instances” that are assigned to a variable for use by other classes.

Access to class members from outside the class is controlled by “access specifiers” in the **class** declaration. Typically, these will deny access to the variable members, but allow access to methods that can store and retrieve data from those variable members. This technique of “data hiding” ensures that stored data is safely encapsulated within the class variable members, and is the first principle of Object Oriented Programming (OOP).

Don't forget

If not specified, the default access specifier for a class declaration is **internal**, and the default access specifier for class members is **private**.

A class declaration comprises a class access specifier, the **class** keyword, and a programmer-specified name – adhering to the usual C# naming conventions, but beginning in uppercase.

The class declaration is followed by a pair of braces containing the variable and method declarations, which typically begin with their own access specifier. So, class syntax looks like this:

```
access-specifier class ClassName
{
  // Member variable property.
  access-specifier data-type variable-name ;

  // Member method.
  access-specifier return-type method-name ( parameter-list )
  {
    statements
  }
}
```

An access specifier may be any one of these keywords:

- **public** – accessible from any place where the class is visible.
- **private** – accessible only to other members of the same class.
- **protected** – accessible only to other members of the same class and members of classes derived from that class.
- **internal** – accessible only to members of the same assembly.

Hot tip

Derived classes, which use the **protected** member access specifier, are introduced later – see page 118.

Any real-world object can be defined by its attributes and by its actions. For example, a dog has attributes such as name, age, and color, plus actions it can perform such as bark. The class mechanism in C# provides a way to create a virtual dog object within a program, where the variable members of a class can represent its attributes and the methods represent its actions:

```
public class Dog
{
  // MEMBER VARIABLES...
  private string name ;
  private int age ;
  private string color ;

  // MEMBER METHODS...

  // Setter & Getter Methods:
  public void setName( string tag )
  {
    name = tag ;            // Store the argument value.
  }

  public string getName( )
  {
    return name ;           // Retrieve the stored value.
  }

  public void setAge( int yrs ) { age = yrs ; }
  public int getAge( ) { return age ; }

  public void setColor( string coat ) { color = coat ; }
  public string getColor( ) { return color ; }

  // Other Methods:
  public string bark( ) { return "\nWoof, woof!\n" ; }
}
```

It is important to recognize that a class declaration only defines a data structure – in order to create an object you must declare an "instance" of that data structure, assigned to a variable. This requires the **new** keyword and class name followed by parentheses:

```
Dog fido = new Dog( ) ;  // Create a "fido"instance of the Dog class.
```

The principle of encapsulation in C# programming describes the grouping together of data and functionality in class members – name, age, color attributes, and bark action in the **Dog** class.

Hot tip

A program class cannot perfectly emulate a real-world object, but aims to encapsulate all relevant attributes and actions.

Don't forget

The **public** "setter" and "getter" methods assign values to, and retrieve values from, the **private** variable members.

Hot tip

It is conventional to begin class names with an uppercase character, and instance names with lowercase.

Creating multiple objects

A program can easily create multiple objects simply by declaring multiple **new** instances of a **class**, and each object can have unique attributes by assigning individual values with its setter methods.

It is often convenient to combine the setter methods into a single method that accepts arguments for each **private** variable. This means that all values can be assigned with a single statement in the program, but the method will contain multiple statements.

In C# **class** declarations, the **public** "setter" methods, which assign data to **private** variable members, and **public** "getter" methods, which retrieve data from **private** variable members, are often named as the variable they address – but with the first letter made uppercase and prefixed by "set" or "get" respectively. For example, methods to access an **age** variable may be **setAge( )** and **getAge( )**.

Parameters may also be named as the variable they address. The code can differentiate between the parameter and like-named variable member by dot-prefixing **this** to the variable name:

Objects

Don't forget

In the setter method, the **this** prefixed names reference the **class** variable members, and those without prefix reference the parameters. No prefix is needed to reference the **class** variable members.

1. Start a new **C# Console App** and name it "Objects"

```
Console.Title = "Objects" ;
// Top-level statements to be inserted here (Steps 6-10).
```

2. Next, declare a **class** named "Dog" with three variable members, plus setter and getter methods for each variable

```
public class Dog
{
  private string name , color ;
  private int age ;

   public void setValues( string name, int age, string color )
  {
    this.name = name ;
    this.age = age ;
    this.color = color ;
  }

  public string getName( ) { return name ; }
  public int getAge( ) { return age ; }
  public string getColor( ) { return color ; }
}
```

3. Then, add a final miscellaneous method in the **Dog** class

```
public string bark( ) { return "\nWoof, woof!\n" ; }
```

4 Above the class declaration, begin to add top-level statements by creating an instance of the **Dog** class

```
Dog fido = new Dog( ) ;
```

5 Next, call the new instance object's setter method to initialize all its variable members

```
fido.setValues( "Fido" , 3 , "Brown" ) ;
```

6 Now, retrieve all properties of the new object

```
string tagF = String.Format( "{0} is a {1} year old {2} dog",
          fido.getName( ) ,
          fido.getAge( ) ,
          fido.getColor( )
) ;
```

7 Display all properties and call the miscellaneous method

```
Console.WriteLine( tagF + fido.bark( )  ) ;
```

8 Now, create another instance of the **Dog** class

```
Dog lucy = new Dog( ) ;
lucy.setValues( "Lucy" , 2 , "Gray" ) ;
```

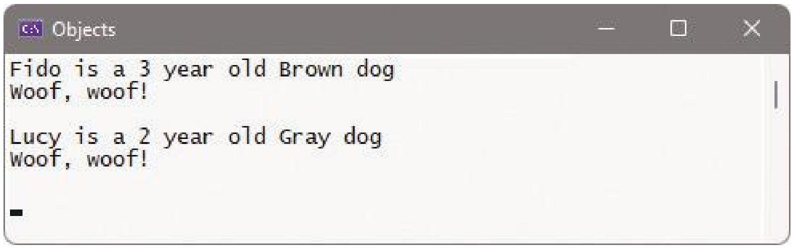
9 Next, retrieve all properties of this new object

```
string tagL = String.Format( "{0} is a {1} year old {2} dog",
          lucy.getName( ) ,
          lucy.getAge( ) ,
          lucy.getColor( )
) ;
```

10 Display all properties and call the miscellaneous method

```
Console.WriteLine( tagL + lucy.bark( )  ) ;
Console.ReadKey( ) ;
```

11 Press **Start** or **F5** to see each object's properties



```
Fido is a 3 year old Brown dog
Woof, woof!

Lucy is a 2 year old Gray dog
Woof, woof!
```

Don't forget

Top-level statements must precede any class declarations in the program code.

Hot tip

Notice here how the **String.Format( )** method call is coded using newlines to clearly build a **string** value for output.

Initializing class members

Class variable members can be initialized by a special "constructor" method that is called whenever an instance of the class is created – allowing you to specify default values for class variable members.

The constructor method is always named exactly as the class name, and can contain statements to initialize class variable members – for example, **public class Cat** has a **public Cat( )** constructor.

When all class variables have been initialized by the class constructor method, any instance object of that class will immediately have those initial property values. Individual setter methods can usefully adjust the class variable values as needed:

Constructor

1. Start a new **C# Console App** and name it "Constructor"
```
Console.Title = "Constructor" ;
// Top-level statements to be inserted here (Steps 6-11).
```

2. Next, declare a **class** named "Cat" with three variable members
```
public class Cat
{
  private string name , color ;
  private int age ;
  // Methods to be inserted here (Steps 3-5).
}
```

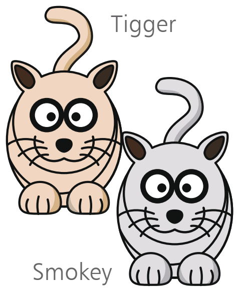
3. Now, insert a class constructor method to set default values for all its variable members
```
  public Cat( )
  {
    name = "Tigger" ; age = 3 ; color = "Brown" ;
  }
```

4. Then, insert setter and getter methods for each variable
```
public void setName( string name ) { this.name = name ; }
public void setAge( int age ) { this.age = age ; }
public void setColor( string color ) { this.color = color ; }

public string getName( ) { return name ; }
public int getAge( ) { return age ; }
public string getColor( ) { return color ; }
```

5. Add a final miscellaneous method in the **Cat** class
```
public string cry( ) { return "\nMeow, meow!\n" ; }
```

6 Above the class declaration, begin to add top-level statements by creating an instance of the **Cat** class

```
Cat tigger = new Cat( ) ;
```



7 Now, retrieve all (default) properties of the new object

```
string tagT = String.Format( "{0} is a {1} year old {2} cat",
          tigger.getName( ) ,
          tigger.getAge( ) ,
          tigger.getColor( )
) ;
```

8 Display all properties and call the miscellaneous method

```
Console.WriteLine( tagT + tigger.cry( )  ) ;
```

9 Now, create another instance of the **Cat** class and set each property with new values

```
Cat smokey = new Cat( ) ;

smokey.setName( "Smokey" ) ;
smokey.setAge( 2 ) ;
smokey.setColor( "Gray" ) ;
```

10 Next, retrieve all (adjusted) properties of this new object

```
string tagS = String.Format( "{0} is a {1} year old {2} cat",
          smokey.getName( ) ,
          smokey.getAge( ) ,
          smokey.getColor( )
) ;
```

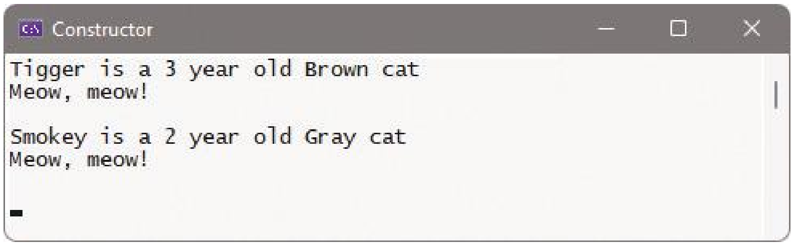
11 Display all properties and call the miscellaneous method

```
Console.WriteLine( tagS + smokey.cry( )  ) ;
Console.ReadKey( ) ;
```

Hot tip

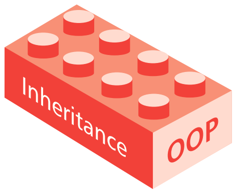
You can also specify parameters to a constructor method in order to allow argument values to be passed when a **new** instance object is created.

12 Press **Start** or **F5** to run the application and see the properties of each object instance and method called



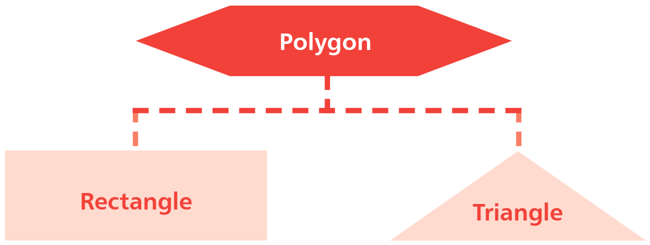
```
Tigger is a 3 year old Brown cat
Meow, meow!

Smokey is a 2 year old Gray cat
Meow, meow!
```

Inheriting class properties



A C# class can be created as a brand new class, like those in previous examples, or can be "derived" from an existing class. Importantly, a derived class inherits members of the parent (base) class from which it is derived – in addition to its own members.

The ability to inherit members from a base class allows derived classes to be created that share certain common properties, which have been defined in the base class. For example, a "Polygon" base class may define width and height properties that are common to all polygons. Classes of "Rectangle" and Triangle" could be derived from the Polygon class – inheriting width and height properties, in addition to their own members defining their unique features.



The virtue of inheritance is extremely powerful and is the second principle of Object Oriented Programming (OOP).

A derived class declaration adds a colon : after its class name, followed by the name of the class from which it derives:

Inheritance

1. Start a new **C# Console App** and name it "Inheritance"

```
Console.Title = "Inheritance" ;
// Top-level statements to be inserted here (Steps 5-7).
```

2. Next, declare a base **class** named "Polygon", containing two variable members and one setter method member

```
public class Polygon
{
  protected int width , height ;

  public void setValues( int width , int height )
  {
    this.width = width ;
    this.height = height ;
  }
}
```

3 Now, define a class that derives from the base class, inheriting members and adding a method

```
public class Rectangle : Polygon
{
  public int area( ) { return ( width * height ) ; }
}
```

4 Then, define another class that derives from the base class, inheriting members and adding a similar method to that in the previous step

```
public class Triangle : Polygon
{
  public int area( ) { return ( ( width * height ) / 2 ) ; }
}
```

5 Above the class declarations, begin to add top-level statements by creating an instance object from each derived class

```
Rectangle rect = new Rectangle( ) ;
Triangle cone = new Triangle( ) ;
```

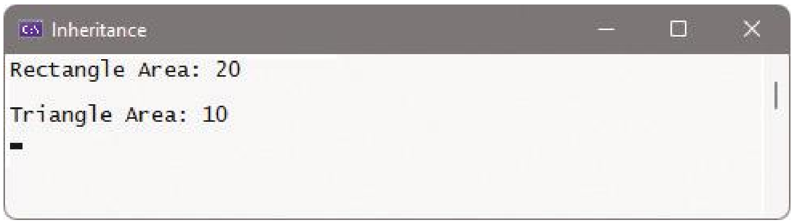
6 Call the inherited setter method of each derived class to initialize all the inherited variable members

```
rect.setValues( 4 , 5 ) ;
cone.setValues( 4 , 5 ) ;
```

7 Finally, call the added method in each derived class to display their computed values

```
Console.WriteLine( "Rectangle Area: " + rect.area( ) ) ;
Console.WriteLine( "\nTriangle Area: " + cone.area( ) ) ;
Console.ReadKey( ) ;
```

8 Press **Start** or **F5** to run the application and see the output from inherited variables



```
Rectangle Area: 20

Triangle Area: 10
```

Hot tip

The : operator is used here to create derived classes, and is equivalent to the **extends** keyword in other programming languages – such as Java.

Don't forget

The methods added to each derived class can be named alike, as they only exist within the scope of their respective class.

Don't forget

Notice that the setter method and variables are not defined in the derived classes, as they are inherited from the base class.

Calling base constructors

Although derived classes inherit the members of their parent base class, they do not inherit its constructor method. Nonetheless, a constructor method of the base class is always called when a new object of a derived class is created. The call to the base class constructor method is made in addition to the call to the constructor method of the derived class.

The default constructor method of a base class has no parameters – but the base class may also have one or more "overloaded" constructor methods that do have parameters.

If you prefer to call an overloaded constructor of the base class when a new object of a derived class is created, you can create a matching overloaded constructor in the derived class – having the same number and type of arguments. The matching derived class constructor must then explicitly call the overloaded base class constructor using the **base** keyword. This is appended to the derived class constructor declaration after a : colon character.

When a new object of a derived class is created, argument values can be passed to an overloaded derived class constructor, and also onwards to its matching overloaded base class constructor in parentheses following the **base** keyword:

Don't forget

An overloaded method is one that has the same name as another method but different parameters.

Base

1. Start a new **C# Console App** and name it "Base"

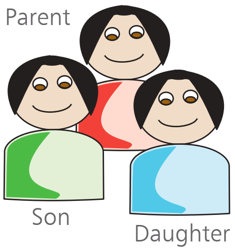
```
Console.Title = "Base" ;
// Top-level statements to be inserted here (Step 6).
```

2. Next, declare a base **class** named "Parent", containing a default constructor method and an overloaded constructor method that requires a single integer argument

```
public class Parent
{
  public Parent( )
  {
    Console.WriteLine( "Parent Called" ) ;
  }

  public Parent( int num )
  {
        Console.WriteLine( "Parent+ Called: " + num ) ;
  }
}
```

3. After the base class, add a derived class containing a default constructor method

```
public class Daughter : Parent
{
  public Daughter( )
  { Console.WriteLine( "\tDaughter Called\n" ) ; }
}
```



4. Next, add another derived class, containing a default constructor method

```
public class Son : Parent
{
  public Son( )
  { Console.WriteLine( "\tSon Called\n" ) ; }

  // Overloaded constructor to be inserted here (Step 5).
}
```

5. Now, insert an overloaded constructor method into the derived class, which requires a single integer argument

```
  public Son( int num ) : base( num )
  {
    Console.WriteLine( "\tSon+ Called: " + num ) ;
  }
```

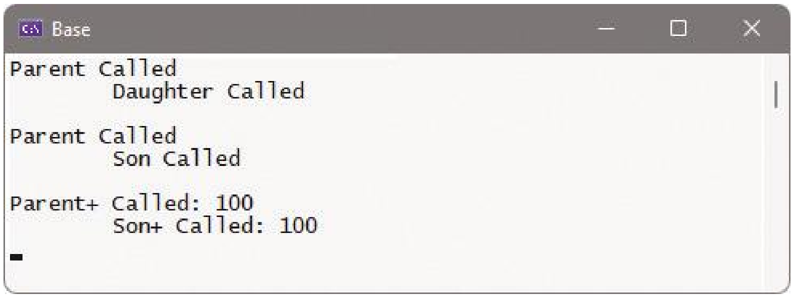
Beware

The default base constructor method will be called implicitly unless the **base** keyword is used to call explicitly.

6. Above the class declarations, add top-level statements that create instances of the derived classes

```
Daughter anna = new Daughter( ) ;
Son brad = new Son( ) ;
Son carl = new Son( 100 ) ;
Console.ReadKey( ) ;
```

7. Press **Start** or **F5** to run the application and see the output from base and derived class constructors



```
Parent Called
        Daughter Called

Parent Called
        Son Called

Parent+ Called: 100
        Son+ Called: 100
```

Don't forget

Here, the argument value is passed to the derived class and base class.

Hiding base methods

A method can be declared in a derived class to “hide” a similar method in the base class – if both method declarations have matching name, arguments, and return type.

Creation of a matching method in a derived class effectively hides the base class method, as it generally becomes inaccessible. To indicate that hiding is intentional, rather than accidental, the hiding method declaration should include the **new** keyword.

Base class methods can be called explicitly from non-static methods by prefixing their method name with the **base** keyword. Alternatively, as a derived class is a specialization of its base class, base class methods can be called explicitly using an explicit cast:

Hide

1. Start a new **C# Console App** and name it “Hide”

```
Console.Title = “Hide” ;
// Top-level statements to be inserted here (Steps 5-10).
```

2. Next, declare a base **class** named “Man”, containing a simple method without parameters plus an overloaded method with a single parameter

```
public class Man
{
  public void speak( )
  {
    Console.Write( “Hello: ” ) ;
  }

  public void speak( string message )
  {
    Console.WriteLine( message + “!\n” ) ;
  }
}
```

3. After the base class, add a derived class containing a method to intentionally hide the overloaded method of the same name and parameters in the base class

```
public class Hombre : Man
{
  public new void speak( string message )
  {
    // Statement to be inserted here (Step 4).
    Console.WriteLine( message ) ;
  }
}
```

4 Next, insert a statement in the derived class to explicitly call the simple method in the base class
```
base.speak( ) ;
```

5 Above the class declarations, begin to add top-level statements by creating an instance of the base class
```
Man henry = new Man( ) ;
```

6 Next, create an instance of the derived class
```
Hombre enrique = new Hombre( ) ;
```

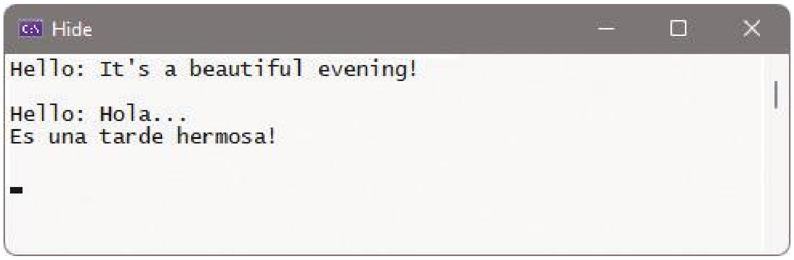
7 Add a call to the simple method inherited by the instance from the base class
```
henry.speak( ) ;
```

8 Now, add a call to the overloaded method inherited from the base class
```
henry.speak( "It's a beautiful evening" ) ;
```

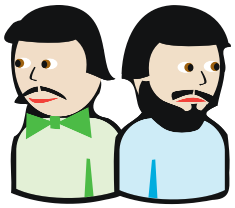
9 Then, add a call to the hiding method in the derived class – that will also explicitly call the simple method in the base class
```
enrique.speak( "Hola..." ) ;
```

10 Finally, add a statement using an explicit cast – to explicitly call the overloaded method in the base class
```
( (Man) enrique ).speak( "Es una tarde hermosa" ) ;
Console.ReadKey( ) ;
```

11 Press **Start** or **F5** to run the application and see the output from base class methods and hiding method



```
Hello: It's a beautiful evening!

Hello: Hola...
Es una tarde hermosa!

_
```



A derived class method name and parameter list must match that in its base class to override it.

Directing method calls

The three cornerstones of Object Oriented Programming (OOP) are **encapsulation**, **inheritance**, and **polymorphism**. Previous examples have demonstrated **encapsulation** of data within a class, and **inheritance** of base class members by derived classes. The term **polymorphism** (from Greek, meaning "many forms") is the ability to have a different behavior according to context. C# class methods can be made to be polymorphic. Method declarations in a base class can include the **virtual** keyword to allow that method to be overridden in derived classes. Similarly, method definitions in a derived class can include the **override** keyword to denote it will be overriding a **virtual** base class method. This lets derived class methods provide their own version of a base class method. The great advantage of polymorphism with multiple derived class objects is that calls to methods of the same name are directed to the appropriate overriding method. This can allow inconsistencies, however – this example seems to imply that chickens can fly!

Hot tip

C# overloaded operators can be described as polymorphic. For example, the + operator can add or concatenate according to its context.

C#

Override

1. Start a new **C# Console App** and name it "Override"

```
Console.Title = "Override" ;
// Top-level statements to be inserted here (Steps 5-7).
```

2. Next, declare a base **class** named "Bird", containing two methods that allow overriding

```
public class Bird
{
  public virtual void talk( )
  { Console.WriteLine( "A Bird Talks..." ) ; }

  public virtual void fly( )
  { Console.WriteLine( "A Bird Flies...\n"  ) ; }
}
```

3. After the base class, add a derived class containing two methods that will override the base class methods

```
public class Pigeon : Bird
{
  public override void talk( )
  { Console.WriteLine( "Pigeon Says: Coo! Coo!" ) ; }

  public override void fly( )
  {
    Console.WriteLine( "A Pigeon Flies Away..."  ) ;
    base.fly( ) ;
  }
}
```

Don't forget

Here, the **fly( )** method in each derived class also calls the base class **fly( )** method directly.

4 Next, add another derived class containing two methods that will also override the base class methods

```
public class Chicken : Bird
{
  public override void talk( )
  { Console.WriteLine( "Chicken Says: Cluck! Cluck!" ) ; }

  public override void fly( )
  {
    Console.WriteLine( "I'm A Chicken - I Can't Fly"  ) ;
    base.fly( ) ;
  }
}
```

5 Above the class declarations, begin to add top-level statements with a function to call both overriding methods

```
static void describe( Bird obj )
{
    obj.talk( ) ;
    obj.fly( ) ;
}
```

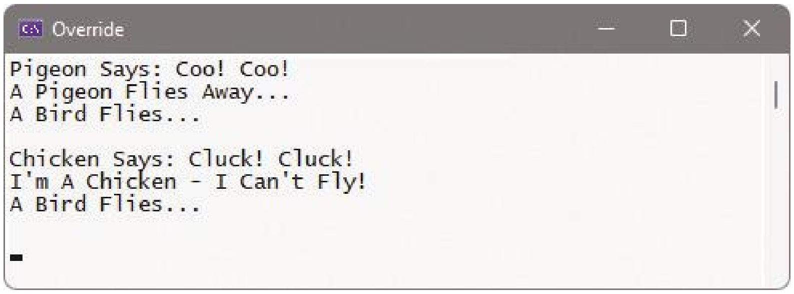
6 Then, create instances of each non-base class

```
Pigeon joey = new Pigeon( ) ;
Chicken lola = new Chicken( ) ;
```

7 Finally, add statements to call appropriate methods

```
describe( joey ) ;
describe( lola ) ;
Console.ReadKey( ) ;
```

8 Press **Start** or **F5** to run the application and see the output from overriding methods and base class method



```
Pigeon Says: Coo! Coo!
A Pigeon Flies Away...
A Bird Flies...

Chicken Says: Cluck! Cluck!
I'm A Chicken - I Can't Fly!
A Bird Flies...

_
```

Beware

You must use the **base** keyword prefix to directly call a base class method from a derived class.

Hot tip

Each instance is passed as an argument for C# to determine the appropriate overriding method to execute.

Hot tip

Polymorphism in programming is often called "one interface, multiple functions".

Providing capability classes

Classes whose sole purpose is to allow other classes to be derived from them are known as "capability classes" – they provide capabilities to the derived classes. Capability classes generally contain no data, but merely declare a number of methods that can be overridden in their derived classes.

A capability class and its methods can all be declared using the **abstract** keyword to denote that they can only be implemented in derived classes. In this case, the method signature is followed by a semicolon, rather than a method block containing statements. Method definitions in derived classes can then include the **override** keyword to implement the capability class methods.

Adding the **sealed** keyword to a class declaration is a safeguard that prevents that class being used as a base class. Understandably, an **abstract** capability class cannot be **sealed**.

The following example builds upon the previous example on pages 124-125 to demonstrate how the "Bird" class can be better written as a capability class. This now prevents the base class methods being called directly, to avoid inconsistencies:

Capability

1. Start a new **C# Console App** and name it "Capability"

```
Console.Title = "Capability" ;
// Top-level statements to be inserted here (Steps 6-7).
```

2. Next, define a base capability **class** named "Bird", containing two methods

```
public abstract class Bird
{
  public abstract void talk( ) ;

  public abstract void fly( ) ;
}
```

3. Add a derived class containing two methods that will override the base class methods

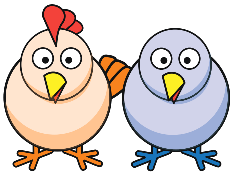
```
public sealed class Pigeon : Bird
{
  public override void talk( )
  { Console.WriteLine( "Pigeon Says: Coo! Coo!" ) ; }

  public override void fly( )
  { Console.WriteLine( "A Pigeon Flies Away..."  ) ; }
}
```

Beware

You cannot create an instance of an **abstract** class.

4 Now, add another class that also contains two methods that will override the base class methods

```
public sealed class Chicken : Bird
{
  public override void talk( )
  { Console.WriteLine( "Chicken Says: Cluck! Cluck!" ) ; }

  public override void fly( )
  { Console.WriteLine( "I'm A Chicken - I Can't Fly"  ) ; }
}
```



5 Then, add another class containing a single method that accepts an object argument

```
public sealed class Caller
{
  public void describe( Bird obj )
  {
    obj.talk( ) ;
    obj.fly( ) ;
  }
}
```

6 Above the class declarations, begin to add top-level statements by creating instances of each non-base class

```
  Pigeon joey = new Pigeon( ) ;
  Chicken lola = new Chicken( ) ;
  Caller call = new Caller( ) ;
```

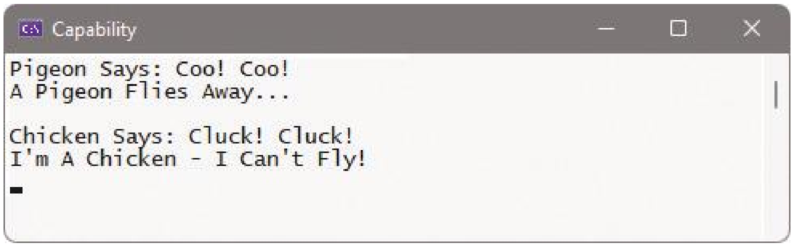
7 Finally, add statements to call appropriate methods by passing an object as an argument

```
  call.describe( joey ) ;
  call.describe( lola ) ;
  Console.ReadKey( ) ;
```

Don't forget

Each instance is passed as an argument for C# to determine the appropriate overriding method to execute.

8 Press **Start** or **F5** to run the application and see the output from the overriding methods of derived classes


Capability

```
Pigeon Says: Coo! Coo!
A Pigeon Flies Away...

Chicken Says: Cluck! Cluck!
I'm A Chicken - I Can't Fly!
```

Employing partial classes

The source code for each example listed in this book is generally contained in a single **.cs** file, as each one is just a simple program. When working on larger projects it is often desirable to spread the source code over separate files to allow multiple programmers to work on the project at the same time. Visual Studio allows you to easily add one or more code files in which to write classes.

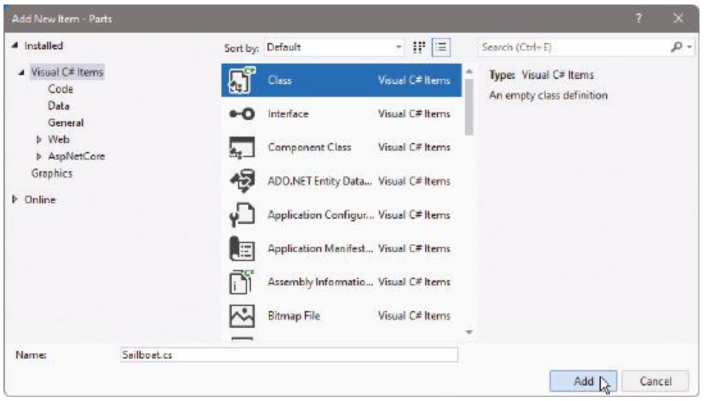
Class definitions can also be spread across several files by including the **partial** keyword in each separate part of the definition. Providing all parts have the same level of accessibility, the C# compiler will combine all parts into a single definition:

Parts

1. Start a new **C# Console App** and name it "Parts"

```
Console.Title = "Parts" ;
// Top-level statements to be inserted here (Steps 9-10).
```

2. On the Visual Studio menu, select **Project**, **Add Class...** to launch the "Add New Item" dialog box

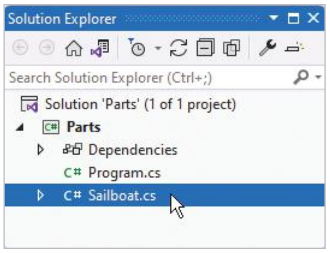

Add New Item - Parts
Installed
Visual C# Items
Code
Data
General
Web
AspNetCore
Graphics
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C# Items
Interface Visual C# Items
Component Class Visual C# Items
ADO.NET Entity Data... Visual C# Items
Application Configur... Visual C# Items
Application Manifest... Visual C# Items
Assembly Informatio... Visual C# Items
Bitmap File Visual C# Items
Type: Visual C# Items
An empty class definition
Add
Cancel

Name: Sailboat.cs

3. Edit the name field to the name of the class to be added there – in this case, it's to be a **class** named "Sailboat"

4. Click the **Add** button to add the new file to your project

5. Next, select **View**, **Solution Explorer** and double-click on **Sailboat.cs** to open that file in the **Code Editor**


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Parts' (1 of 1 project)
Parts
Dependencies
Program.cs
Sailboat.cs

Beware

If any part is declared **abstract**, the whole class will be **abstract**, and if any part is declared **sealed**, the whole class will be **sealed**.

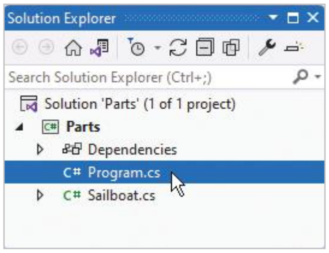
Hot tip

Alternatively, you can right-click on the project name in Solution Explorer and choose **Add**, **New Item** to launch this dialog.

6. Replace the default code with this class constructor part

```
public partial class Sailboat
{
  private string make ;
  private string model ;

  public Sailboat( string make , string model )
  {
    this.make = make ;
    this.model = model ;
  }
}
```

7. Now, in **Solution Explorer**, double-click on **Program.cs** to open that file in the **Code Editor**


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Parts' (1 of 1 project)
Parts
Dependencies
Program.cs
Sailboat.cs

8. Then, add a class method part

```
public partial class Sailboat
{
  public void describe( )
  {
    Console.WriteLine( "Sailboat: {0} {1}" , make , model ) ;
  }
}
```

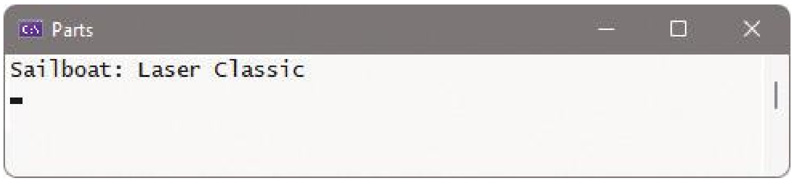
9. Above the class declaration, begin to add top-level statements by creating an instance object

```
Sailboat boat = new Sailboat( "Laser" , "Classic" ) ;
```

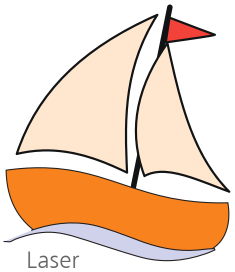
10. Finally, add a statement to call the added class method

```
boat.describe( ) ;
Console.ReadKey( ) ;
```

11. Press **Start** or **F5** to run the application and see the output from the class parts spread over two files



```
Sailboat: Laser Classic
```



Laser
Sailboat (Boat)

Hot tip

Notice how the strings are substituted for output in this example from members declared in another file.

Summary

- Encapsulation, inheritance, and polymorphism are the three cornerstones of Object Oriented Programming.
- A C# **class** is a data structure that can contain both variable members and method members.
- Access to **class** members is controlled by access specifiers.
- An instance object of a **class** is created using the **new** keyword followed by the class name and parentheses.
- Typically, **public** setter and getter methods provide access to **private** variables to ensure stored data is safely encapsulated.
- Parameters can be named as the variable they address, and the **this** keyword prefix can be used to differentiate between them.
- The constructor method of a **class** is named as the class name and is called each time an instance of that **class** is created.
- Derived classes inherit members of the base parent **class** from which they are derived.
- A derived **class** declaration adds a colon after the class name, followed by the name of the **class** from which it derives.
- An overloaded constructor of a base **class** can be called using the **base** keyword in a derived class declaration.
- A derived **class** can include a **new** method to hide a method in its parent class if name, arguments, and return type match.
- Base class methods can be called explicitly from a derived class by prefixing the method name with the **base** keyword.
- The **virtual** and **override** keywords can be used to allow base class methods to be overridden by derived class methods.
- Capability classes can include the **abstract** keyword to denote their methods must be implemented in derived classes.
- Adding the **sealed** keyword to a class declaration is a safeguard that prevents that class being used as a base class.
- Class definitions can be spread across several files by including the **partial** keyword in each part of the definition.

10 Controlling events

This chapter demonstrates how a C# program can recognize and respond to events that occur in a graphical application.

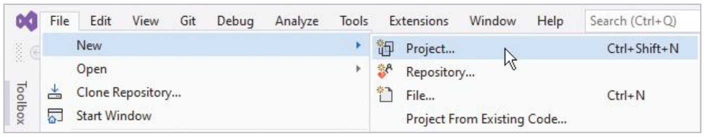
Starting a Forms project

Visual Studio provides a **Windows Forms App** template that allows you to easily create a C# program that provides a Graphical User Interface (GUI) to interact with the user:

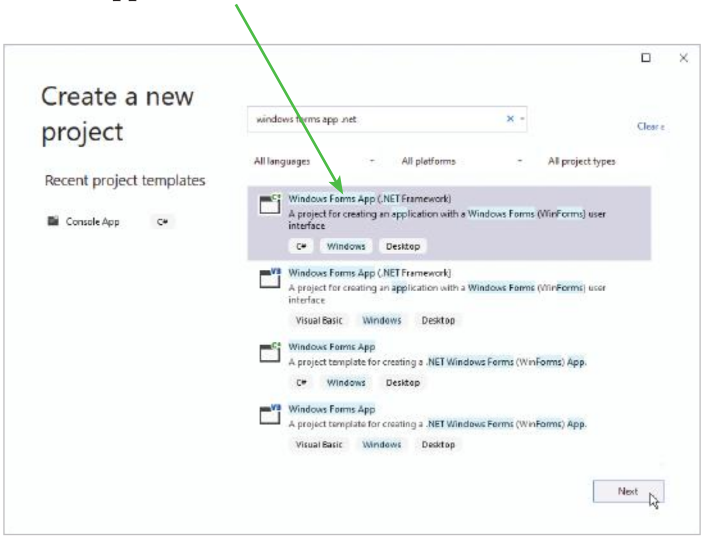
C#

FirstGUI

1. On the Menu Bar, click **File**, **New**, **Project**, to open the "Create a new project" dialog



2. In the "Create a new project" dialog, select **Windows Forms App (.NET Framework)** then click **Next**



Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop

Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
Visual Basic Windows Desktop

Windows Forms App
A project template for creating a .NET Windows Forms (WinForms) App.
C# Windows Desktop

Windows Forms App
A project template for creating a .NET Windows Forms (WinForms) App.
Visual Basic Windows Desktop

Hot tip

Windows GUI applications provide interactive controls that produce "events" in response to user actions, and your program can respond to those actions. This is known as event-driven programming.

3. Enter a project name of your choice in the **Name** field, then click on the **Create** button to create the new project – in this case, the project name will be "FirstGUI"

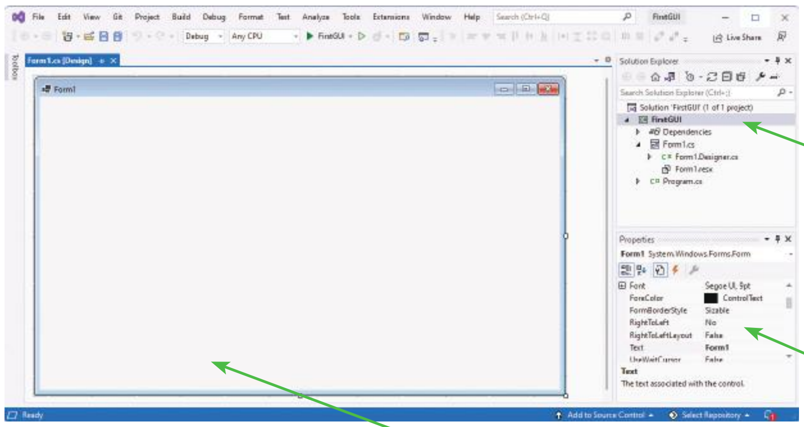
4. Select the .NET version, then wait until Visual Studio creates your new project. You should then see a **Form Designer** window appear displaying a blank Form

5. Select the **View**, **Solution Explorer** menu to open a Solution Explorer window and see all files in your project

Hot tip

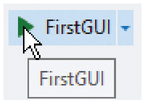
You can open the FormDesigner window from the menu by clicking **View**, **Designer** or by pressing the **Shift** + **F7** keyboard keys.

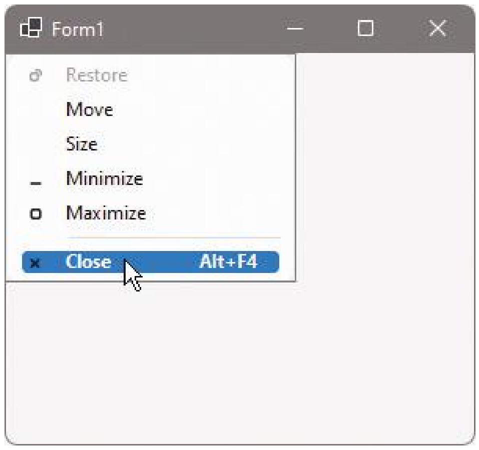
6 Now, select the **View**, **Properties** menu to open a **Properties** window to reveal all properties of your Form



The **Form Designer** is where you create visual interfaces for your applications, and the **Properties** window contains details of the item that is currently selected in the Form Designer window.

7 The Visual Studio IDE has now gathered all the resources needed to build a default Windows application – click the **Start** button on the toolbar to launch this application





The application simply creates a basic window – you can move it, minimize it, maximize it, resize it, and quit the application by closing it. It may not do much, but you have already created a real Windows GUI app!

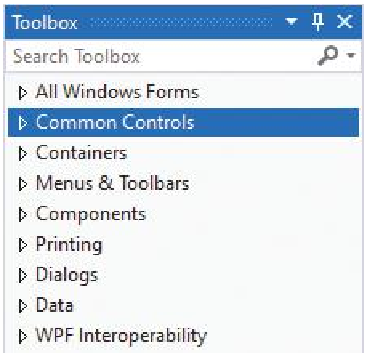
Alternatively, you can run applications using the **Debug**, **Start Debugging** menu options or by pressing the **F5** keyboard key.

Adding visual controls

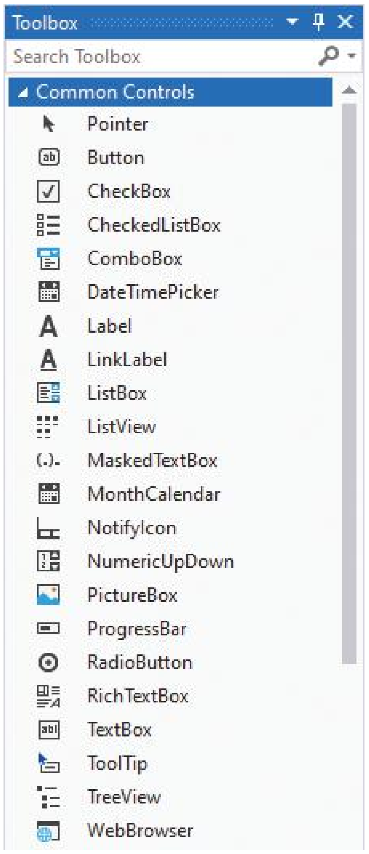
The **Toolbox** in the Visual Studio IDE contains a wide range of visual controls, which are the building blocks of your applications. Using the project created on pages 132-133, follow these steps to start using the Toolbox now:

FirstGUI (continued)

1 Click the vertical **Toolbox** tab at the left edge of the IDE window, or click **View**, **Toolbox** on the Menu Bar, to display the Toolbox contents. The visual controls are contained under various category headings beside an ▷ expansion arrow



2 Click on the expansion arrow beside the **Common Controls** category heading to expand the list of the most commonly used visual controls. Usefully, each control name appears beside an icon depicting that control as a reminder. You can click on the category heading again to collapse the list, then expand the other categories to explore the range of controls available to build your application interfaces



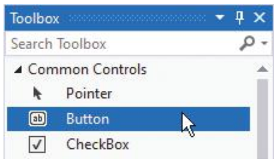
Don't forget

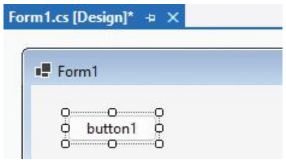
Any pinned Window in the IDE can be dragged from its usual location to any position you prefer. Drag it back to the initial location to re-dock it.

Hot tip

The **Toolbox** will automatically hide when you click on another part of the IDE, but it can be fixed in place so it will never hide, using the **Pin** button on the Toolbox bar.

3 Find the **Button** item in the **Common Controls** category of the Toolbox, then drag it onto the Form in the Designer window. Alternatively, double-click the **Button** item to add it to the Form

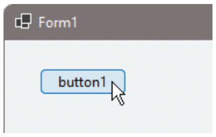




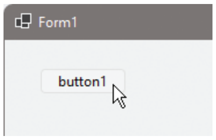
The **Button** control appears on the Form surrounded by "handles" that can be dragged to resize the button's width and height.

4 Click the ▶ **Start** button to run the application and see the **Button** control appear in its initial default state

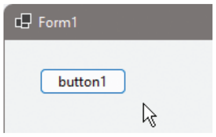
5 Next, move the pointer over the **Button** control to see its appearance change to its "MouseHover" state



6 Now, click and hold down the **Button** control to see its appearance change to its "MouseDown" state



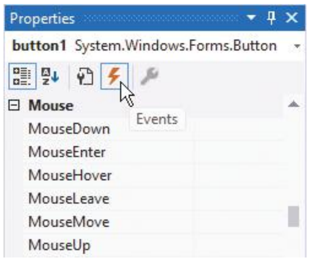
7 Finally, release the **Button** control to see its appearance change to its "MouseUp" state

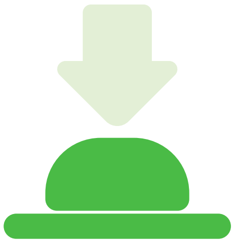


Each change of appearance is Windows' default response to an event that occurs on the **Button** control, but your C# program can provide its own response to these, and many more, control events:

8 Select the **Button** control then open its **Properties** window

9 Click the **Events** button and scroll down to see all possible **Button** events





A **Button** is one of the most useful interface controls – your program determines what happens when the user clicks it.

This **Button** control performs no function when it's clicked – until you add some code.

Each control can be given a more meaningful name in the **Properties** window, but for simplicity, the default names are used by the examples in this book.

Writing functional code

The Visual Studio IDE automatically generates code in the background, to incorporate the visual controls you add to your program interface. Additional code can be added manually to the "code-behind" page using the IDE's integral **Code Editor**, to determine how your program should respond to interface events:

Hot tip

Use the tabs to switch between the **Code Editor** and the **Form Designer**.

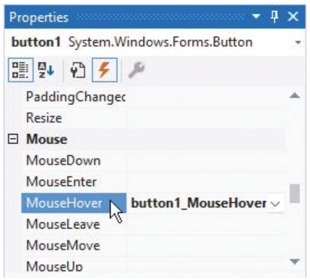
C#

FirstGUI (continued)

Using the project created on pages 132-135, follow these steps to add your own responses to user actions that produce events:

1. Select the **Button** control named **button1**, then open its **Properties** window and click the **Events** button

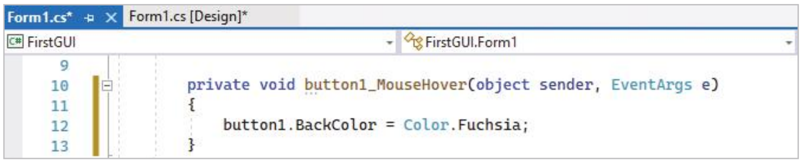
2. Double-click on the **MouseHover** item to open a **Form1.cs** code-behind page in the **Code Editor** at a generated event-handler method



3. Next, insert this statement within the generated method – to change the **Button**'s background color in response to a **MouseHover** event
button1.BackColor = Color.Fuchsia ;

Hot tip

The **Color** class provides lots of standard color properties – type **Color** then a period and use IntelliSense to see them.



```
private void button1_MouseHover(object sender, EventArgs e)
{
    button1.BackColor = Color.Fuchsia;
}
```

4. Similarly, double-click on the **MouseDown** item and insert this statement within another generated method – to change the **Button**'s background color in response to a **MouseDown** event
button1.BackColor = Color.Lime ;

5. Likewise, double-click on the **MouseUp** item and insert this statement within a further generated method – to change the **Button**'s background color in response to a **MouseUp** event
button1.BackColor = Color.Aqua ;

Beware

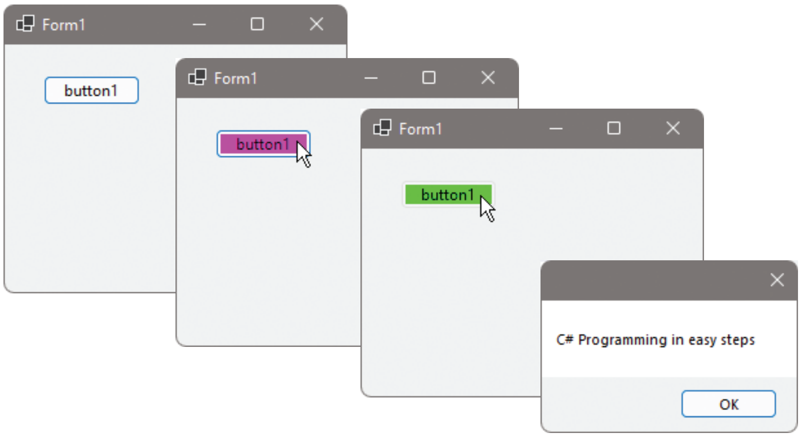
The lines of code in this example are to be inserted into each individual event-handler method that is automatically generated by Visual Studio.

6. Now, click the **Form1.cs [Design]** tab to return to the **Form Designer** window

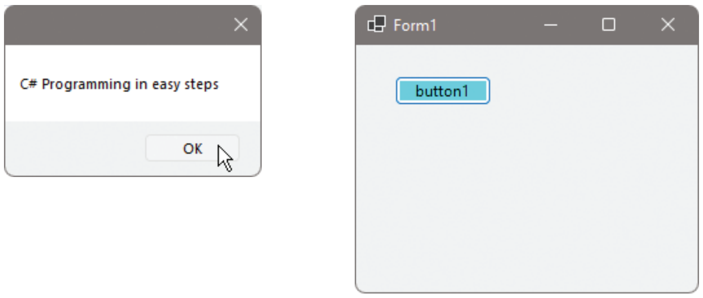
7. Double-click on the **Button** control on the Form to open the **Form1.cs** code-behind page in the **Code Editor** at yet another generated event-handler method

8. Finally, insert this statement within the generated method – to open a message box in response to a **Click** event
```
MessageBox.Show( "C# Programming in easy steps" ) ;
```

9. Run the application, then click the button to see the **MouseHover**, **MouseDown**, and **Click** event responses



10. Push the **OK** button to close the message box and see the **MouseUp** event response



Hot tip

You can use the **View** menu on the Menu Bar to open the **Code Editor**, the **Form Designer**, or any other window you require at any time.

Hot tip

You can see many more uses for the **MessageBox** class on page 142.

Don't forget

Most Windows software works by responding to events. For example, when you press a key, a **KeyPress** event can call its event-handler to respond to that event.

Gathering text entries

A **TextBox** control can be added to a Form in a **Windows Forms Application** to allow the user to enter input. The current value within a **TextBox** can be assigned to a variable, typically in response to a **Button Click** event, for use inside the program code.

A new value can be assigned to a **TextBox** as output to the user, but the user can modify this value. Alternatively, output can be assigned to a **Label** control, which the user cannot modify:

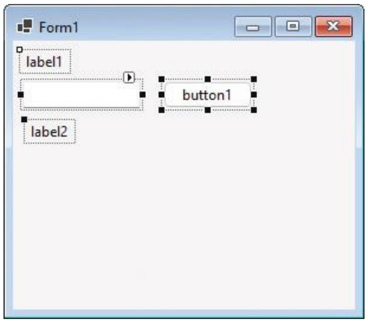
Hot tip

A **Label** control is generally used to display static text.

C#

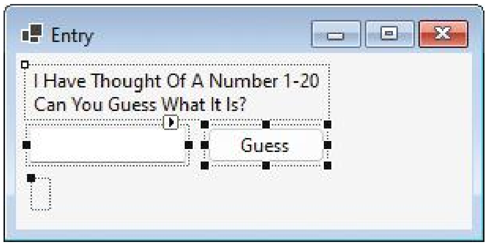
Entry

1. Start a new **Windows Forms App**, then add a **TextBox**, a **Button**, and two **Label** controls



2. Select the Form itself, then in the **Properties** window modify its **Appearance, Text** and **Layout, Height** values

3. Next, modify the **Text** property values of the **Button** control and **Label** controls so they look like this:



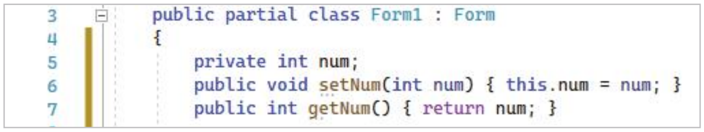
Beware

In the **Layout** property category, you must set **AutoSize** to **False** before you can adjust the **Width** and **Height** values in the **Size** property category. This is needed to allow multiple lines of text on a **Label** control.

4. Now, double-click the **Button** control to open the **Code Editor** at a generated **Click** event-handler

5. At the very beginning of the **Form1** class block, add a variable with setter and getter methods

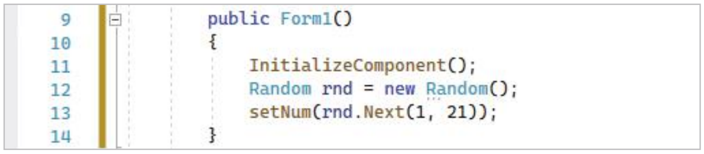
```
private int num ;
public void setNum( int num ) { this.num = num ; }
public int getNum( ) { return num ; }
```



```
public partial class Form1 : Form
{
    private int num;
    public void setNum(int num) { this.num = num; }
    public int getNum() { return num; }
```

6 In the **Form1( )** constructor method block, insert statements to initialize the variable with a random value

```
Random rnd = new Random( ) ;
setNum( rnd.Next( 1, 21 )  ) ;
```



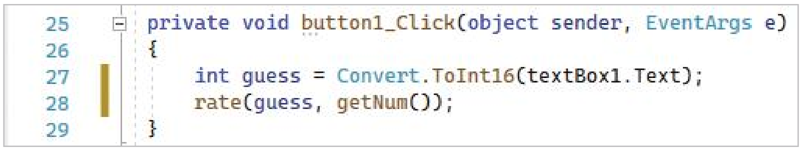
```
public Form1()
{
    InitializeComponent();
    Random rnd = new Random();
    setNum(rnd.Next(1, 21));
}
```

7 Then, add a method to compare two arguments and set a Label control's Text property with an appropriate message

```
public void rate( int guess, int num )
{
  if ( guess < num ) label2.text = "Too Low!" ;
  else
  if ( guess > num ) label2.text = "Too High!" ;
  else
  label2.text = "*** Correct ***" ;
}
```

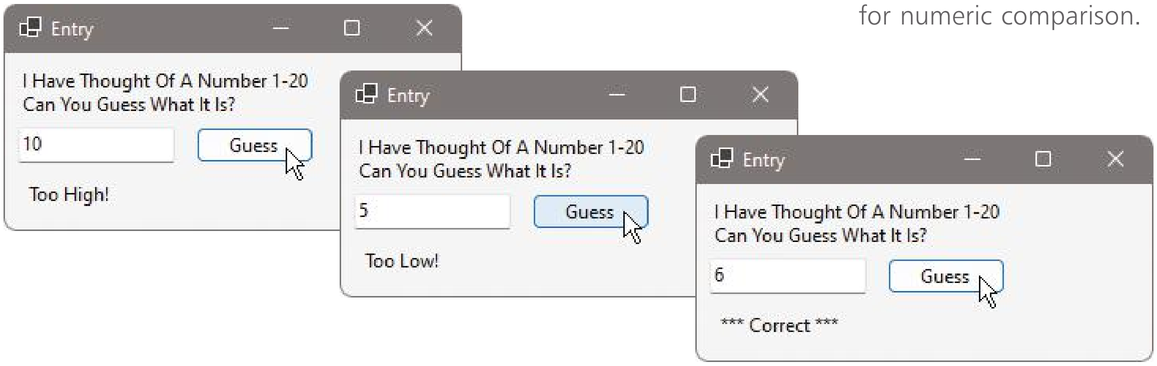
8 Finally, insert statements within the **Button Click** event-handler to compare user input against the random value

```
int guess = Convert.ToInt16( textBox1.text ) ;
rate( guess, getNum( )  ) ;
```



```
private void button1_Click(object sender, EventArgs e)
{
    int guess = Convert.ToInt16(textBox1.Text);
    rate(guess, getNum());
}
```

9 Press **Start** or **F5** to run the application and enter input to see the appropriate output messages



```
Entry
I Have Thought Of A Number 1-20
Can You Guess What It Is?
10        [Guess]
Too High!
```

```
Entry
I Have Thought Of A Number 1-20
Can You Guess What It Is?
5         [Guess]
Too Low!
```

```
Entry
I Have Thought Of A Number 1-20
Can You Guess What It Is?
6         [Guess]
*** Correct ***
```

Beware

If you specify minimum and maximum arguments to the **Next( )** method, it will return a random integer between the specified minimum and maximum-1 – so specifying 21 will allow a maximum of 20.

Don't forget

The content of a **TextBox** control is a string value – so it must be converted to a numeric data type for numeric comparison.

Ticking option boxes

A **CheckBox** control is a small box with a caption. It lets the user select the caption choice by clicking on the box, and a check mark appears in the box to indicate it has been chosen. Clicking the box once more deselects the choice and unchecks the box. **CheckBox** controls are ideal to present a set of choices from which the user can select none, one, or more than one choice.

A **RadioButton** control is like a **CheckBox**, but with one crucial difference – the user can check only one choice in the group. Checking a **RadioButton** automatically unchecks any others. **RadioButton** controls are ideal to present a set of choices from which the user can select only one choice.

Selected **CheckBox** and **RadioButton** items can usefully be added to the collection displayed in a **ListBox** control:

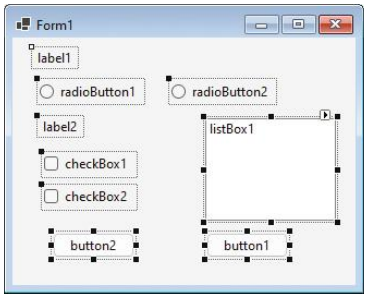
Beware

You must set one RadioButton control to Checked to specify a default option.

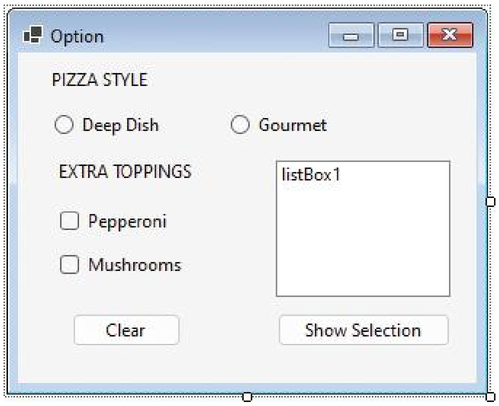
C#

Option

1 Start a new **Windows Forms App**, then add **RadioButton**, **CheckBox**, **ListBox**, **Button**, and **Label** controls to the Form



2 Modify the **Text** property values of the controls to look like this:



Don't forget

The name of the **ListBox** control that appears in the Form Designer will not be displayed when the application runs.

3 Now, double-click the "Show Selection" **Button** control to open the **Code Editor** at its generated **Click** event-handler

4 In the **Form1( )** constructor method block, insert a statement to specify a default option

```
radioButton1.Checked = true ;
```

5 Next, insert a statement within the "Show Selection" **Button Click** event-handler to empty any listed items

```
listBox1.Items.Clear( ) ;
```

6 Insert statements within the "Show Selection" **Button Click** event-handler to add selected options to the list

```
if ( radioButton1.Checked )
  listBox1.Items.Add( radioButton1.Text ) ;
if ( radioButton2.Checked )
  listBox1.Items.Add( radioButton2.Text ) ;
if ( checkBox1.Checked )
  listBox1.Items.Add( checkBox1.Text ) ;
if ( checkBox2.Checked )
  listBox1.Items.Add( checkBox2.Text ) ;
```

7 Now, double-click the "Clear" **Button** control to open the **Code Editor** at its generated **Click** event-handler

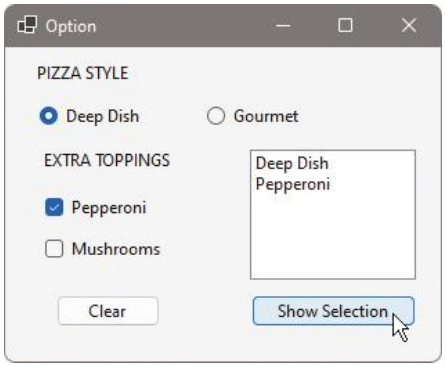
8 Insert statements within the "Clear" **Button Click** event-handler to reset all the options

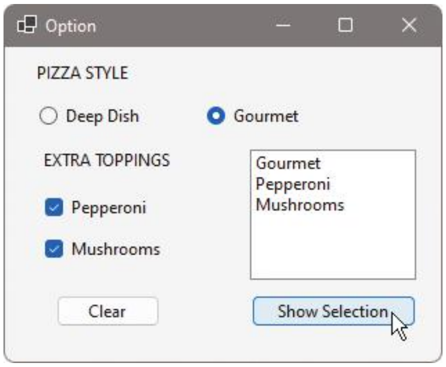
```
listBox1.Items.Clear( ) ;
radioButton1.Checked = true ;
checkBox1.Checked = false ;
checkBox2.Checked = false ;
```

9 Press **Start** or **F5** to run the application and select options to see them appear in the list – then hit **Clear**

Hot tip

The **ListBox** control has an **Items** property that provides methods to add or remove list items from a collection.





Showing user messages

The features of a **MessageBox** dialog can be determined by adding three further arguments after the message string within its **Show( )** method. These can specify a caption, which buttons the dialog will display, and which graphic icon will appear on the dialog.

MessageBoxButtons constants
AbortRetryIgnore
OK
OKCancel
RetryCancel
YesNo
YesNoCancel

The dialog button combinations can be specified using the **MessageBoxButtons** constant values listed in this table. For example, to have the dialog display **Yes**, **No**, and **Cancel** buttons, specify the **MessageBoxButtons.YesNoCancel** constant.

MessageBoxIcon constants	
Error Hand Stop	⛔
Question	?
Exclamation Warning	⚠
Asterisk Information	ℹ
None	

The dialog icon can be specified using the **MessageBoxIcon** constant values listed in this table. For example, to have the dialog display the question mark icon, specify the **MessageBoxIcon.Question** constant.

When the user presses a **MessageBox** dialog button it returns an appropriate **DialogResult** constant value to the program. These are named exactly as the individual button label they represent. For example, any **MessageBox** dialog **OK** button returns the **DialogResult.OK** constant. The program can therefore examine the returned value to determine how to proceed.

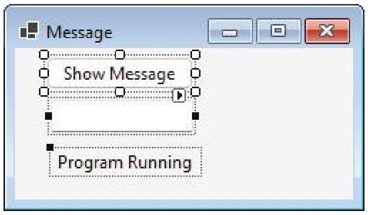
Don't forget

A **MessageBox** is "modal" – the user must deal with its dialog message before the program can proceed.

Hot tip

Always specify a graphic icon when calling a **MessageBox** dialog to help the user easily understand the nature of the message.

1 Start a new **Windows Forms App**, then add a **Button**, a **TextBox**, and a **Label** control to the Form



C#

Message

2 Double-click the "Show Message" **Button** control to open the **Code Editor** at its generated **Click** event-handler

3 Insert a block within the "Show Message" **Button** **Click** event-handler to deliberately throw an exception

```
try
{
  throw new NotImplementedException( ) ;
}
catch ( NotImplementedException notImp )
{
  // Statements to be inserted here (Steps 4-5 ).
}
```

Hot tip

This technique of throwing a **NotImplementedException** is favored by some programmers as a reminder to complete an unfinished part of the program code.

4 Add a statement to assign the user's decision to a variable

```
DialogResult result =
MessageBox.Show( "Proceed?", notImp.Message ,
  MessageBoxButtons.YesNo , MessageBoxIcon.Error ) ;
```

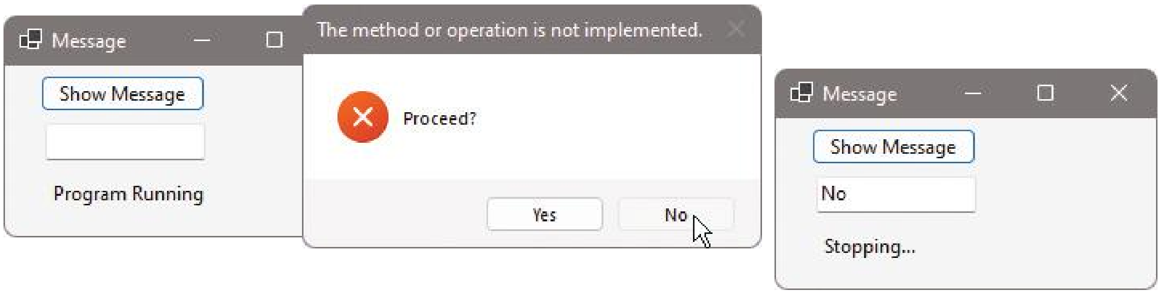
5 Now, add statements to respond to the user's decision

```
textBox1.Text = result.ToString( ) ;
if( result == DialogResult.Yes ) label1.Text = "Proceeding..." ;
if( result == DialogResult.No ) label1.Text = "Stopping..." ;
```

Hot tip

Alternatively, this program could stop by calling the form's **Close( )** method in response to **DialogResult.No**.

6 Press **Start** or **F5** to run the application and use the **MessageBox** buttons to determine how to proceed

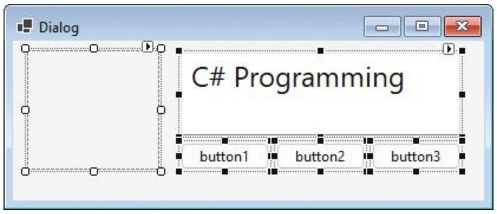


Calling system dialogs

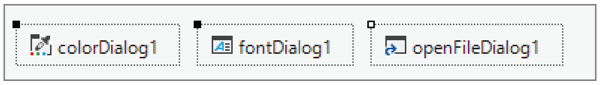
Dialog

Visual Studio makes it simple to add the ability to call upon the standard Windows selection dialogs so the user can choose options within your applications – for example, calling the Color dialog to pick a color, calling the Font dialog to choose font options, or calling the Open dialog to select an image:

1. Start a new **Windows Forms App** and add a **PictureBox**, a **TextBox**, and three **Button** controls to the Form



2. From the **Dialogs** section of the **Toolbox**, add a **ColorDialog**, a **FontDialog**, and an **OpenFileDialog** component to the Form – see them appear in the **Component Tray** at the bottom of the Form Designer



3. Double-click the first **Button** to open the **Code Editor** at its generated **Click** event-handler, then insert this code to change the background color of the **Form**

```
if ( colorDialog1.ShowDialog( ) == DialogResult.OK )
  this.BackColor = colorDialog1.Color ;
```

4. Double-click the second **Button** to open the **Code Editor** at its generated **Click** event-handler, then insert this code to change the font of the **TextBox** control

```
if ( fontDialog1.ShowDialog( ) == DialogResult.OK )
  textBox1.Font = fontDialog1.Font ;
```

5. Double-click the third **Button** to open the **Code Editor** at its generated **Click** event-handler, then insert this code to choose an image for the **PictureBox** control

```
if ( openFileDialog1.ShowDialog( ) == DialogResult.OK )
{
  // Statements to be inserted here (Step 6).
}
```

Don't forget

The **DialogResult.OK** value indicates the user pressed the **OK** button – just as it does in the **MessageBox** example on page 143.

Hot tip

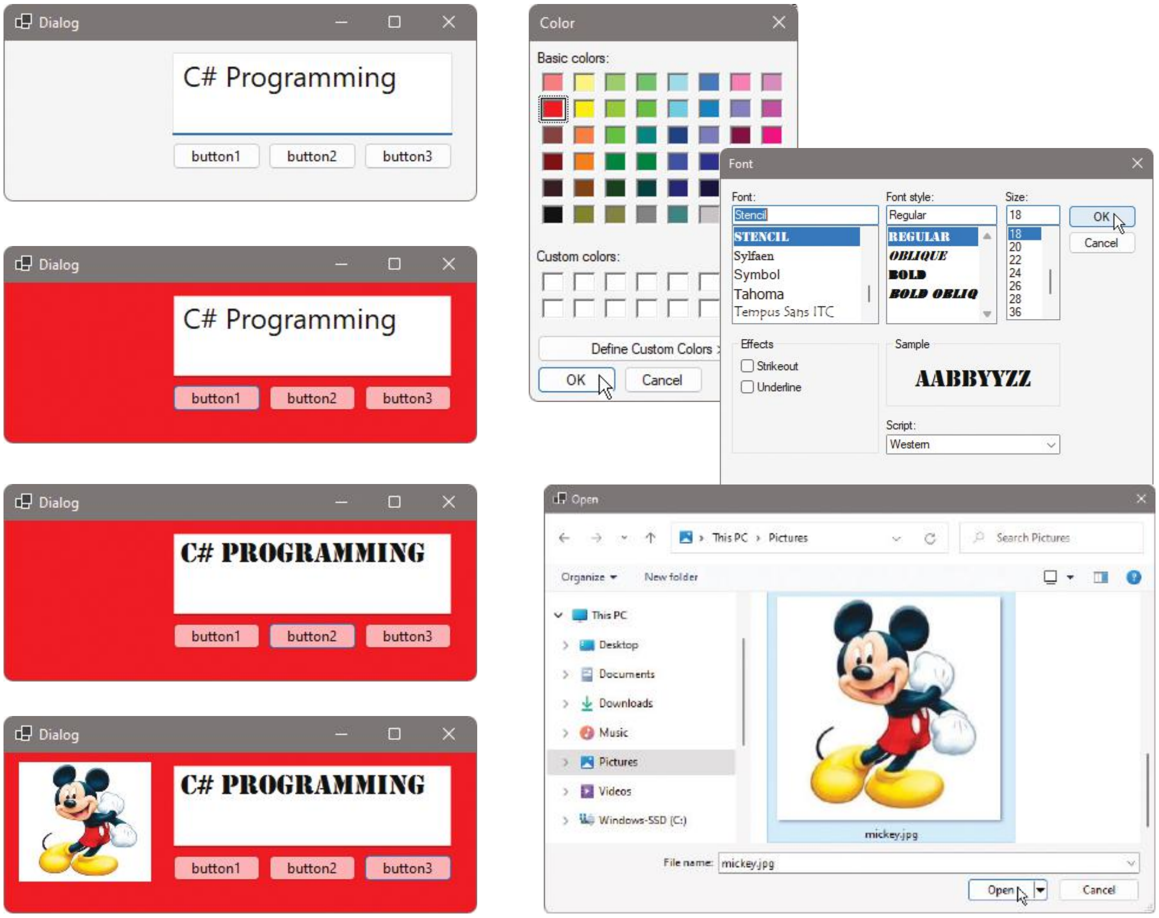
The **SaveFileDialog** is demonstrated in the example on page 149.

6 Insert a block to assign a selected image or display an error message

```
try
{
  pictureBox1.SizeMode =
          PictureBoxSizeMode.StretchImage ;
  pictureBox1.Image =
          new Bitmap( openFileDialog1.FileName ) ;
}
catch ( Exception )
{
  MessageBox.Show( "Error: Select An Image File!" , " ",
          MessageBoxButtons.OK, MessageBoxIcon.Error ) ;
}
```

The **PictureBoxSizeMode. StretchImage** property will stretch, or shrink, a selected image to fit the **PictureBox** dimensions as a new **Bitmap** image.

7 Press **Start** or **F5** to run the application and call the system dialogs to choose program options



Creating application menus

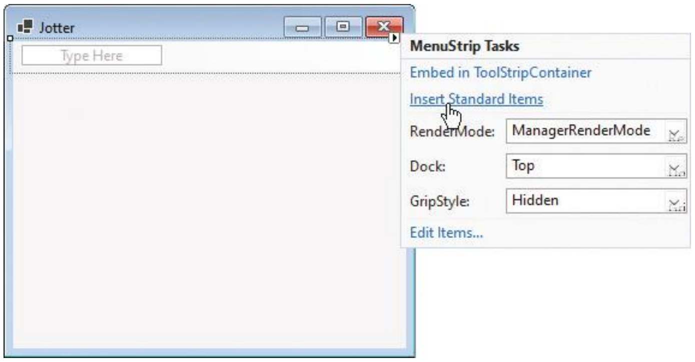
Drop-down menus, toolbars, and status bars, like those found in most Windows applications, can easily be added to your own C# GUI applications from the Toolbox:

C#

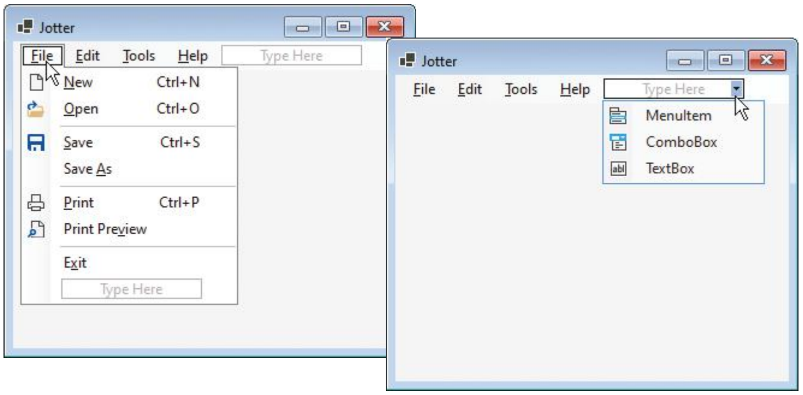
Jotter

1. Start a new **Windows Forms App** and find the **Menus & Toolbars** section of the Toolbox, then double-click the **MenuStrip** item to add it to the Form

2. Click the MenuStrip control's ▶ arrow button to open its **Smart Tag**, then select **Insert Standard Items**



3. Right-click on any of familiar items and use the context menu to edit that item. Also, type new custom items into the **Type Here** box as required

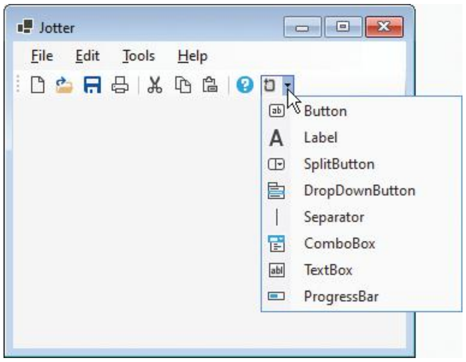


Hot tip

Alternatively, you can create your own custom menus using the **Type Here** box instead of **Insert Standard Items**.

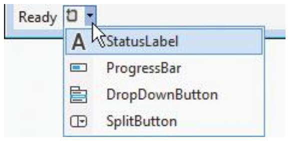
4. In the Toolbox, double-click on the **ToolStrip** item to add it to the Form, then open its **Smart Tag** and once more select **Insert Standard Items**

5 When the familiar icon buttons have been added to the **ToolStrip**, right-click on any item and use the context menu to edit that item. Also add further custom items from the drop-down list as required

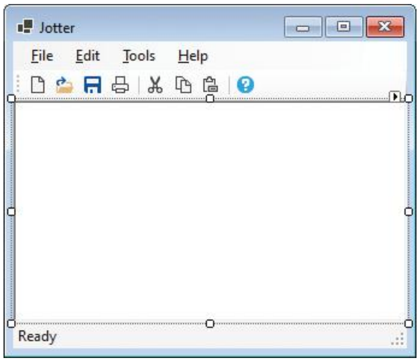


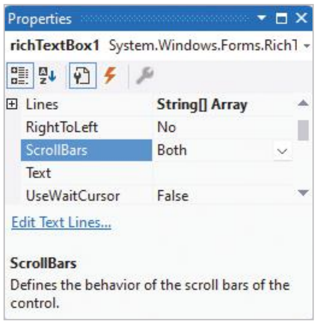
6 In the Toolbox, double-click on the **StatusStrip** item to add it to the Form

7 Select the **StatusLabel** item on the StatusStrip drop-down list, and set its **Text** property to "Ready"



8 Add a **RichTextBox** control to the center of the Form, open its **Smart Tag** and select the option to **Dock in parent container**, then ensure that its **ScrollBars** property is set to **Both**




Properties

richTextBox1 System.Windows.Forms.RichT

Lines	String[] Array
RightToLeft	No
ScrollBars	Both
Text	
UseWaitCursor	False

Edit Text Lines...

ScrollBars
Defines the behavior of the scroll bars of the control.

Hot tip

Use **StatusBar** messages to provide feedback to the user.

Don't forget

The menus are not truly functional until you add some program code – as described on pages 148-149.

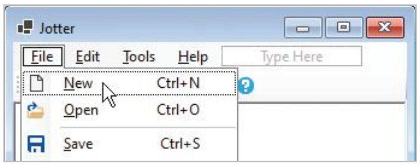
Making menus work

The menu items and toolbar buttons created on pages 146-147 will not truly function until you add code to make them work. For actions that appear both in a menu and on a button, you can create a method that can be called from the **Click** event-handler of the menu item and that of the button – to avoid duplication:

C#

Jotter (continued)

1. In Form Designer, click the app's **File** menu item to open its drop-down menu



2. Double-click on the **New** menu item to open the **Code Editor** in its **Click** event-handler, and add this method call

```
newFile( ) ;
```

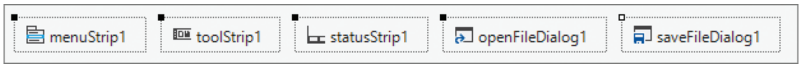
3. After the **Click** event-handler block, add this method to clear any existing text and display a status message

```
private void newFile( )
{
  richTextBox1.Text = null ;
  toolStripStatusLabel1.Text = "Ready" ;
}
```

4. Return to the **Form Designer**, then double-click on the **New** toolbar button to open the **Code Editor** in that event-handler, and add a call to the method above

```
newFile( ) ;
```

5. Add an **OpenFileDialog** and **SaveFileDialog** component from the **Dialogs** section of the **Toolbox**



6. In the **Click** event-handlers of both the **Open** menu item and the **Open** toolbar button, add this method call

```
openFile( ) ;
```

When you enable New, Open, and Save dialogs, keyboard shortcuts are already configured – try **Ctrl** + **N**, **Ctrl** + **S**, and **Ctrl** + **O** to test them.

7 After the **Click** event-handler blocks, add this method to load a plain text file

```
private void openFile( )
{
  openFileDialog1.FileName = "" ;
  openFileDialog1.Filter = "Text Files | *.txt" ;
  if ( openFileDialog1.ShowDialog( ) == DialogResult.OK )
  {
    richTextBox1.LoadFile( openFileDialog1.FileName ,
                            RichTextBoxStreamType.PlainText ) ;
  }
}
```

8 In the **Click** event-handlers of both the **Save** menu item and the **Save** toolbar button, add this method call

```
saveFile( ) ;
```

9 After the **Click** event-handler blocks, add this method to save a plain text file

```
private void saveFile( )
{
  saveFileDialog1.Filter = "Text Files | *.txt" ;
  if ( saveFileDialog1.ShowDialog( ) == DialogResult.OK )
  {
    richTextBox1.SaveFile( saveFileDialog1.FileName ,
                            RichTextBoxStreamType.PlainText ) ;
  }
}
```

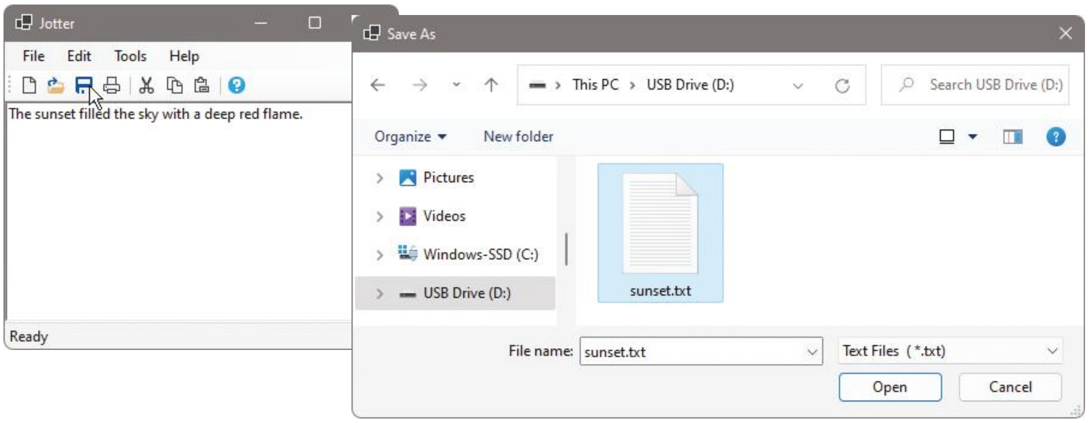
10 Press **Start** or **F5** to run the application and test the functionality of the **New**, **Open**, and **Save** file menu items and toolbar buttons

Hot tip

You can change the Filter to **"RichText | *.rtf"** and the **RichTextBoxStreamType** property to **.RichText** for files with embedded objects such as images.

Hot tip

To make the **File**, **Exit** menu item functional, simply add the call **Application.Exit( ) ;** to its **Click** event-handler.



The sunset filled the sky with a deep red flame.

File name: sunset.txt

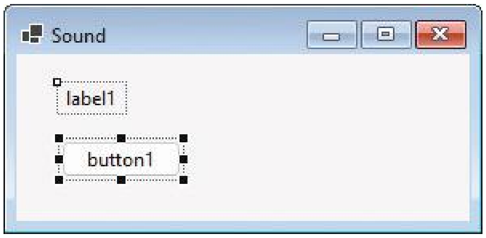
Text Files (*.txt)

Importing audio resources

Sound files can be imported into an application as a resource, in much the same way that image files can be imported as a resource, to enhance the application. These can then be played as required:

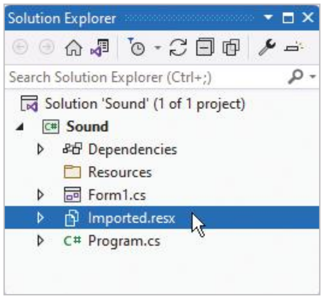
Sound

1. Start a new **Windows Forms App**, then add a **Label** and a **Button** control to the Form



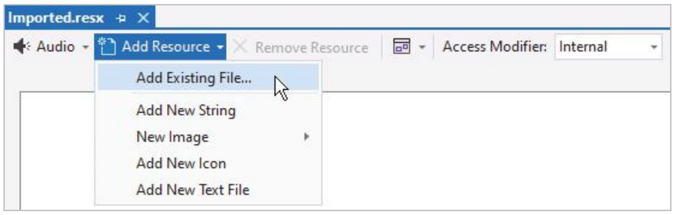
2. In Solution Explorer, right-click on the project item and select **Add**, **New Item**

3. Choose the **Resources File** option and name the file **Imported.resx**, then click the **Add** button



4. In Solution Explorer, double-click the newly added **Imported.resx** item to open its resources window

5. Select **Audio** and **Add Resource**, **Add Existing File...** from the drop-down menus to open a dialog



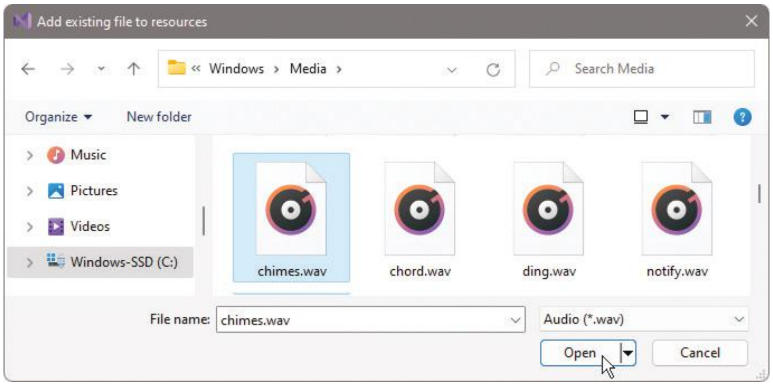
Beware

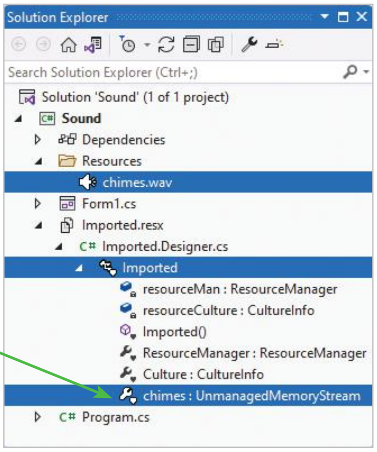
Only sound files in the WAV format (.wav) can be imported as audio resources.

Hot tip

You can typically find the Windows sound files in the folder on your computer at **C:\Windows\Media**.

6 Browse to select an audio file, then click the **Open** button to add it to the project's Resources – here, it's **chimes.wav**




Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Sound' (1 of 1 project)
Sound
Dependencies
Resources
chimes.wav
Form1.cs
Imported.resx
Imported.Designer.cs
Imported
resourceMan : ResourceManager
resourceCulture : CultureInfo
Imported()
ResourceManager : ResourceManager
Culture : CultureInfo
chimes : UnmanagedMemoryStream
Program.cs

7 On the Menu Bar, click **Build**, **Build Solution** to generate an **Imported.chimes** class that calls a **ResourceManager.GetStream()** method

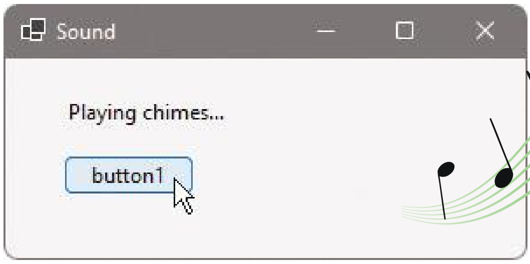
8 Double-click the **Button** control, then add this statement above its **Click** event-handler block to create an object

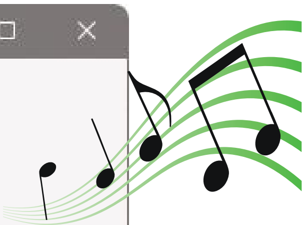
```
System.Media.SoundPlayer chimes =
  new System.Media.SoundPlayer( Imported.chimes ) ;
```

9 Now, add these statements inside the **Button** control's **Click** event-handler to play the sound and show a message

```
chimes.Play( ) ;
label1.Text = "Playing chimes..." ;
```

10 Press **Start** or **F5** to run the application and push the **Button** to hear the sound play





Notice that the sound file name only is dot-suffixed to **Imported** – without its file extension but the case must match that of the imported file name. A sound can be played repeatedly using the **PlayLooping( )** method – and the loop can be ended with the **Stop( )** method.

Summary

- The **Windows Forms App** template in the "Create a new project" dialog is used to begin a Windows GUI project.
- The **Form Designer** window of the Visual Studio IDE is where you create the application's visual interface.
- The **Properties** window contains details of the item that is currently selected in the **Form Designer** window.
- The **Common Controls** section of the **Toolbox** contains a wide range of visual controls to build GUI applications.
- The **Code Editor** is used to create the code-behind page that determines how the application responds to interface events.
- When the user pushes a **Button** control, it creates a **Click** event to which its event-handler can respond.
- User input into a **TextBox** control can be assigned to a variable for use within the application.
- The user cannot directly enter text on a **Label** control.
- Only one option can be checked in a **RadioButton** group, but any number of options can be checked in a **CheckBox** group.
- A **ListBox** control has an **Items** property that provides methods to add or remove items from the list it contains.
- The **Show( )** method of the **MessageBox** class accepts arguments to specify a message, caption, buttons, and icon.
- The **Dialogs** section of the **Toolbox** contains components that allow an application to use the Windows system dialogs.
- The **Menus & Toolbars** section of the **Toolbox** contains components that allow an application to include the familiar Windows system menu items.
- The **Menus & Toolbars** components will not truly function until code is added to their **Click** event-handlers.
- The **Solution Explorer** window can be used to import resources into an application project.
- A **SoundPlayer** object can be created using the **System.Media** class, and provides methods to handle audio resources.

11 Building an application

This chapter demonstrates how to create, test, and publish a simple C# program application.

Planning the program

When creating a new application it is useful to spend some time planning its design. Clearly define the program's precise purpose, decide what application functionality will be required, then decide what interface components will be needed.

A plan for a simple application to pick numbers for a lottery game entry might look like this:

Program purpose

- The program will generate a series of six different random numbers in the range 1–59, and have the ability to be reset.

Functionality required

- A random number generator.
- A method to display six different random numbers.
- A method to clear the last series from display.

Components needed

- Six **Label** controls to display the series of numbers – one number per Label.
- One **Button** control to generate and display the numbers in the Label controls when this Button is clicked. This Button will not be enabled when numbers are on display.
- One **Button** control to clear the numbers on display in the Label controls when this Button is clicked. This Button will not be enabled when no numbers are on display.
- One **PictureBox** control to display a static image – just to enhance the appearance of the interface.

Beware

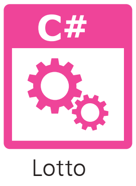
Omission of the planning stage can require time-consuming changes to be made later. It's better to "plan your work, then work your plan".

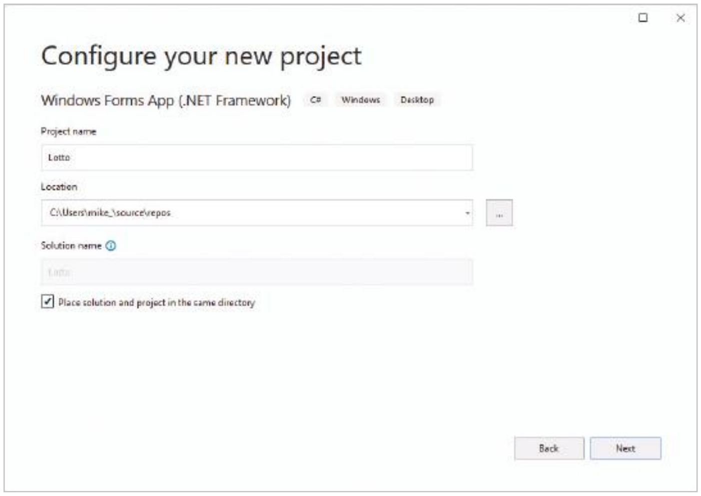
Hot tip

Toggle the value of a Button's **Enabled** property to steer the user. In this case, the application must be reset before a further series of numbers can be generated.

Having established a program plan means you can now create the application basics by adding the components needed to a Form:

1. Open the Visual Studio IDE and create a new **Windows Forms App (.NET Framework)**, then name it "Lotto"





Configure your new project

Windows Forms App (.NET Framework) C# Windows Desktop

Project name: Lotto

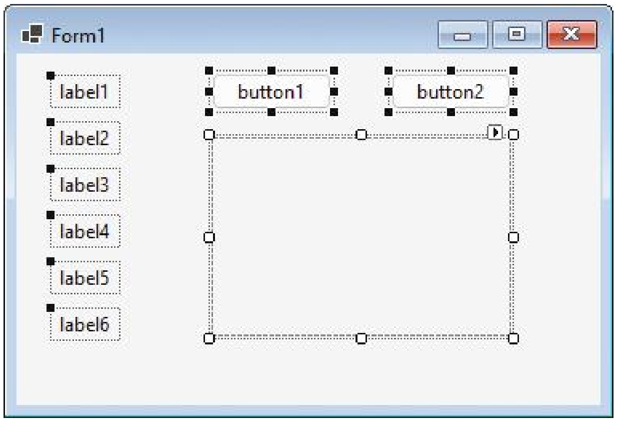
Location: C:\Users\mike_\source\repos

Solution name: Lotto

☑ Place solution and project in the same directory

2. In the **Form Designer**, add six **Label** controls to the Form from the **Toolbox**

3. Now, add two **Button** controls and a **PictureBox** control to the Form



You can drag and drop items from the **Toolbox**, or double-click them to add them to the Form.

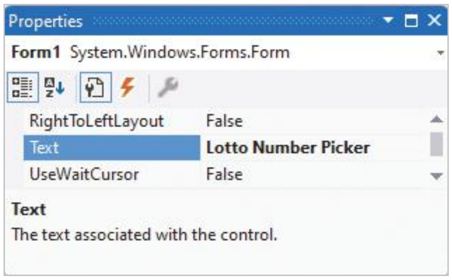
Assigning fixed properties

Having created the application basics on page 155, you can now assign static values using the **Properties** window:

C#

Lotto (continued)

1. Click on the Form to select it, then in the **Properties** window, set the Form's **Appearance, Text** property to "Lotto Number Picker"



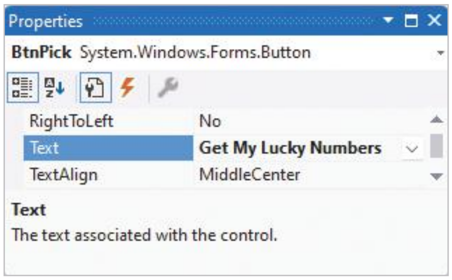
Properties

Form1 System.Windows.Forms.Form

RightToLeftLayout	False
Text	**Lotto Number Picker**
UseWaitCursor	False

Text
The text associated with the control.

2. Select the first **Button** control, then change its **Design, (Name)** to **BtnPick**, and its **Appearance, Text** to "Get My Lucky Numbers"



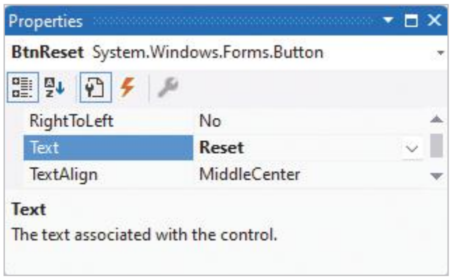
Properties

BtnPick System.Windows.Forms.Button

RightToLeft	No
Text	**Get My Lucky Numbers**
TextAlign	MiddleCenter

Text
The text associated with the control.

3. Select the second **Button** control, then change its **Design, (Name)** to **BtnReset**, and its **Appearance, Text** to "Reset"



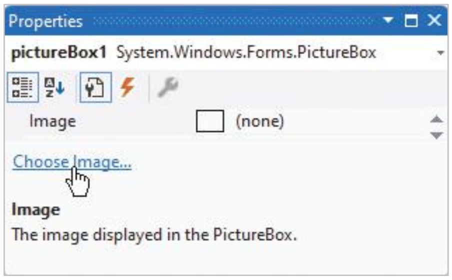
Properties

BtnReset System.Windows.Forms.Button

RightToLeft	No
Text	**Reset**
TextAlign	MiddleCenter

Text
The text associated with the control.

4. Select the **PictureBox** control, then click the **Choose Image...** link to launch the **Select Resource** dialog



Properties

pictureBox1 System.Windows.Forms.PictureBox

Image	(none)

Choose Image...

Image
The image displayed in the PictureBox.

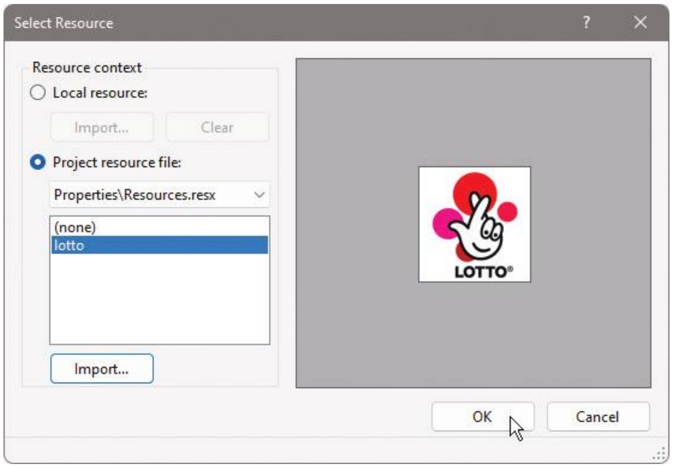
Don't forget

You can open the **Properties** window using the **F4** key, or by clicking **View**, **Properties Window** on the Menu Bar.

Hot tip

The **Label** controls in this program will have their Text property values assigned dynamically at runtime – no initial properties are required.

5 Select the **Project resource file** option, then click the **Import...** button and browse to the image location



6 Click the **OK** button to import the image resource and see it get added to the project

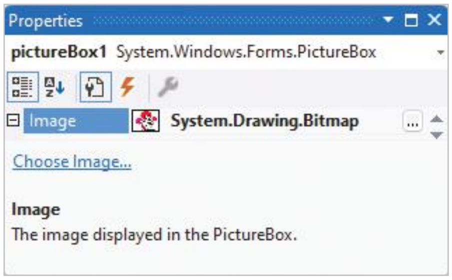
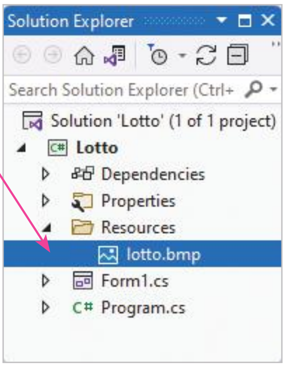


Image
The image displayed in the PictureBox.



7 Click **File, Save All** or press the **Ctrl + Shift + S** keys to save the changes made to the project

Hot tip

You can use the drop-down list at the top of the **Properties** window to select any control on the form.

Beware

Save your project periodically as you build it using **File**, **Save All** on the Menu Bar or **Ctrl** + **Shift** + **S** keys.

Designing the layout

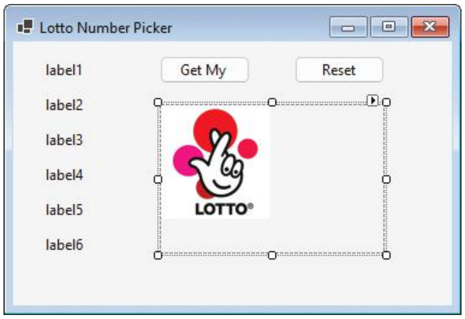
Having assigned fixed property values on pages 156-157, you can now design the interface layout.

The size of both the **PictureBox** control and the **BtnPick** control first needs to be adjusted to accommodate their content. This can easily be achieved by specifying an **AutoSize** value so that Visual Studio will automatically fit the control neatly around its content:

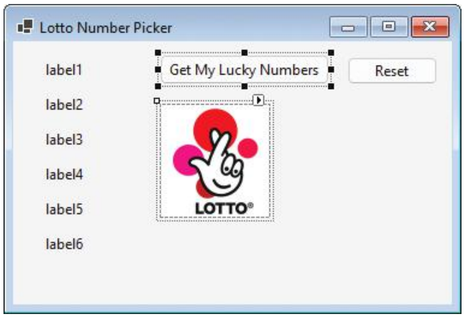
C#

Lotto (continued)

1 Select the **PictureBox** control, then in the **Properties** window, change its **Behavior, SizeMode** to **AutoSize**



2 Select the **BtnPick** control, then in the **Properties** window, set its **Layout, AutoSize** property to **True** – see that the **PictureBox** control now fits snugly around the image, and the **BtnPick** control has expanded to fit its text



Hot tip

Alternatively, you can use the **Smart Tag** arrow button on a **PictureBox** control to set its **SizeMode** property.

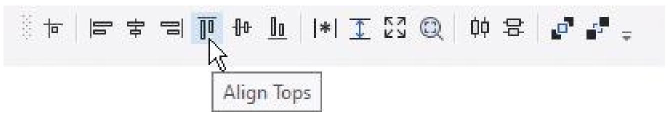
Beware

Ensure that all PictureBox **Margin** properties are set to zero if you do not require margins around the image.

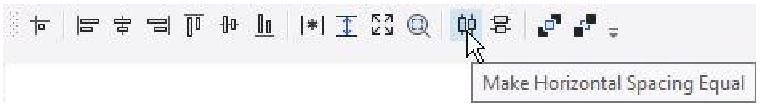
3 Hold down the left mouse button and drag around the Labels to select all **Label** controls

4 Now, click **View, Toolbars, Layout** to ensure that the Layout toolbar is enabled

5 Click the **Align Tops** button to stack the Labels in a pile

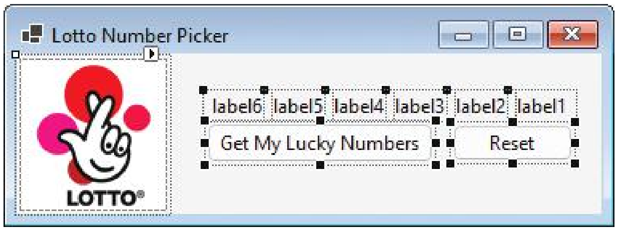


6 Click the **Make Horizontal Spacing Equal** toolbar button to arrange the pile of Labels into a row

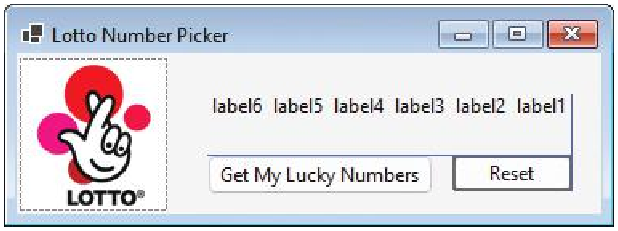


7 Use the Form's right grab handle to extend its width to accommodate the row of Labels and **PictureBox**, then drag the row and both Buttons to the top-right of the Form

8 Drag the **PictureBox** control to the top-left of the Form, then use the Form's bottom grab handle to adjust its height to match that of the image



9 Use the **Snap Lines** that appear when you drag controls around the Form to position the row of Labels and the Buttons to make the interface look like the layout below



10 Set the Form's **Window Style**, **MaximizeBox** and **MinimizeBox** properties to **False**, as **Maximize** and **Minimize** buttons are not required on this interface

Don't forget

In this case, it does not matter in what order the Labels appear in the row.

Hot tip

Avoid the temptation to change the default styles of Form controls so they remain familiar to users.

Setting dynamic properties

Having designed the interface on pages 158-159, you can now add some functionality to dynamically set the initial **Text** properties of the **Label** controls and the initial **Button** states:

C#

Lotto (continued)

Don't forget

The technique described here demonstrates how to use **IntelliSense** – but you can, of course, just type the code directly.

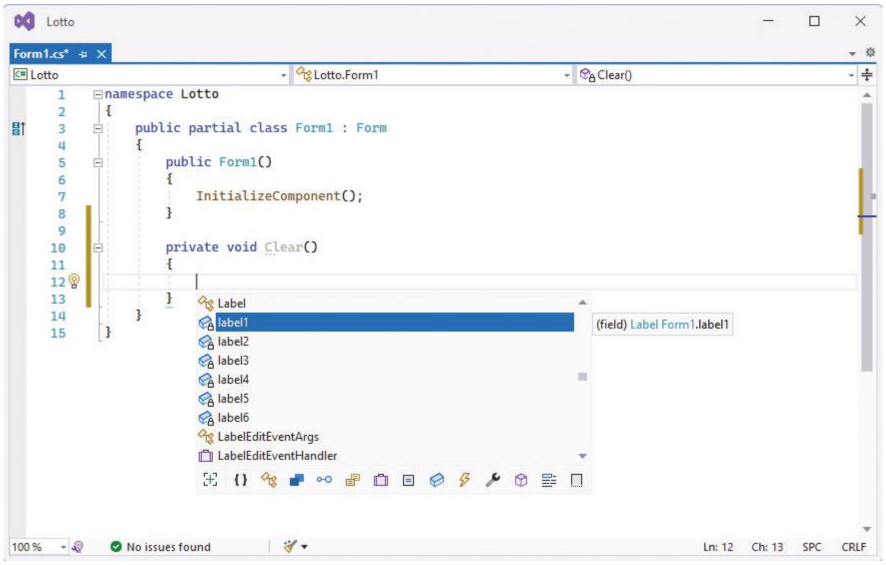
1. Click **View**, **Code** on the Menu Bar, or press **F7**, to open the **Code Editor** window

2. After the **Form1( )** constructor method block, add a new method block

```
private void Clear( )
{
  // Statements to be inserted here (Steps 4-8).
}
```

3. With the cursor inside the new method block, press **Ctrl** + **J**, to open the **IntelliSense** pop-up window

4. Scroll down the list of items in the **IntelliSense** window and double-click on the **label1** item to add it into the **Clear( )** method block



```
namespace Lotto
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Clear()
        {
            |
        }
    }
}
```

5. Type a period, then double-click the **Text** item when the **IntelliSense** window reappears, to add that code

6. Now, type = "..." ; to complete the line so it reads like this:
`label1.Text = "..." ;`

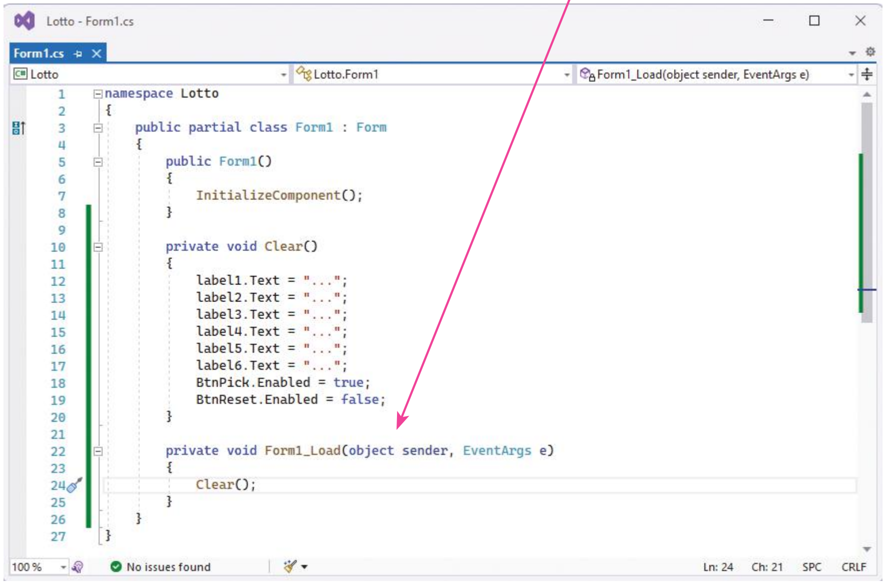
7. Repeat this procedure for the other **Label** controls – so that the **Clear( )** method assigns each an ellipsis string

8. With the cursor inside the **Clear( )** method block, use **IntelliSense** in the same way, to add these two lines:
```
BtnPick.Enabled = true ;
BtnReset.Enabled = false ;
```

This completes the **Clear( )** method functionality by setting the **Button** states. All that remains is to add a call to the **Clear( )** method to execute all of its instructions when the program starts.

9. In the **Form Designer**, double-click on the Form to open the **Code Editor** in its **Load** event-handler, then press **Ctrl** + **J** to open the **IntelliSense** window

10. Scroll down the list in the **IntelliSense** window and double-click on the **Clear** item you have just created, to add a call statement in the **Load** event-handler



```
namespace Lotto
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Clear()
        {
            label1.Text = "...";
            label2.Text = "...";
            label3.Text = "...";
            label4.Text = "...";
            label5.Text = "...";
            label6.Text = "...";
            BtnPick.Enabled = true;
            BtnReset.Enabled = false;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            Clear();
        }
    }
}
```

Hot tip

Type "Intellisense" into the Quick Launch box at the top of the Visual Studio window to discover the IntelliSense Menus and Options.

Adding runtime function

Having created code to initialize dynamic properties on pages 160-161, you can now add runtime functionality to respond to clicks on the **Button** controls:

Lotto (continued)

1. In the **Form Designer**, double-click on the **BtnReset Button** control to open the **Code Editor** in its **Click** event-handler, then add this method call

```
Clear( ) ;
```

This is all that is needed to provide dynamic functionality for the **BtnReset** control. The main dynamic functionality of this application is provided by the **BtnPick** control, which requires an instance of a **Random** class, random number generator object:

2. In the **Form Designer**, double-click on the **BtnPick Button** control to open the **Code Editor** in its **Click** event-handler

3. In the **Click** event-handler block, add this statement to create a random number generator object

```
Random rnd = new Random( ) ;
```

4. Next, add a statement to create an array of 59 elements to store a sequence of numbers

```
int [ ] seq = new int[ 59 ] ;
```

5. Now, add a loop to fill the array elements (index 0 to 58) with integer values 1 to 59

```
for ( int i = 1 ; i < 60 ; i++ )
{
  seq[ i - 1 ] = i ;
}
```

6. Add a second loop to shuffle the values within all the array elements – an algorithm to randomize their order

```
for ( int i = 0 ; i < 59 ; i++ )
{
  int j = ( rnd.Next( ) % 59 ) ;
  int k = seq[ i ] ; seq[ i ] = seq[ j ] ; seq[ j ] = k ;
}
```

Hot tip

The random number generator is used here to generate random values between 0 and 58 to shuffle array elements.

Don't forget

You don't need to understand in detail the algorithm that is used to shuffle the values.

7 Next, add the following lines to display the numbers contained in array elements 1-6 in the **Label** controls

```
label1.Text = seq[ 1 ].ToString( ) ;
label2.Text = seq[ 2 ].ToString( ) ;
label3.Text = seq[ 3 ].ToString( ) ;
label4.Text = seq[ 4 ].ToString( ) ;
label5.Text = seq[ 5 ].ToString( ) ;
label6.Text = seq[ 6 ].ToString( ) ;
```

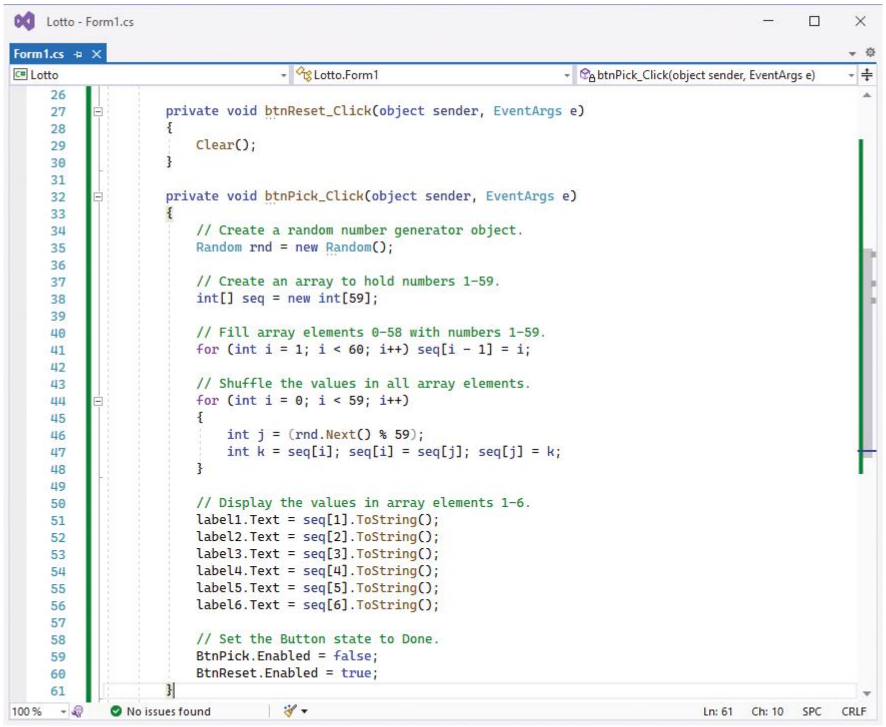
8 Now, add these two lines to set the **Button** states ready to reset the application

```
BtnPick.Enabled = false ;
BtnReset.Enabled = true ;
```

9 Add comments for others examining the code, and for yourself when revisiting the code later – for example, maybe to explain the algorithm that shuffles the numbers

Beware

Label controls can only contain **string** values, so the **int** numeric values must be converted.



```
        private void btnReset_Click(object sender, EventArgs e)
        {
            Clear();
        }

        private void btnPick_Click(object sender, EventArgs e)
        {
            // Create a random number generator object.
            Random rnd = new Random();

            // Create an array to hold numbers 1-59.
            int[] seq = new int[59];

            // Fill array elements 0-58 with numbers 1-59.
            for (int i = 1; i < 60; i++) seq[i - 1] = i;

            // Shuffle the values in all array elements.
            for (int i = 0; i < 59; i++)
            {
                int j = (rnd.Next() % 59);
                int k = seq[i]; seq[i] = seq[j]; seq[j] = k;
            }

            // Display the values in array elements 1-6.
            label1.Text = seq[1].ToString();
            label2.Text = seq[2].ToString();
            label3.Text = seq[3].ToString();
            label4.Text = seq[4].ToString();
            label5.Text = seq[5].ToString();
            label6.Text = seq[6].ToString();

            // Set the Button state to Done.
            BtnPick.Enabled = false;
            BtnReset.Enabled = true;
        }
```

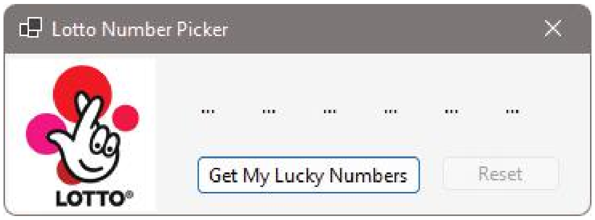
10 Click **File, Save All** or press the **Ctrl** + **Shift** + **S** keys to save the changes made to the project code-behind page

Testing the program

Having worked through the program plan on the previous pages, the components needed and functionality required have now been added to the application – so it's ready to be tested:

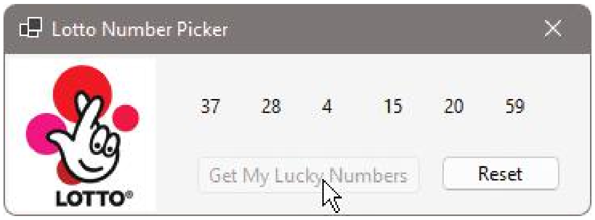
Lotto
(continued)

1. Click the **Start** button, or press **F5**, to run the application then examine its initial start-up appearance



The Form's **Load** event-handler has set the initial dynamic values of each **Label** control and disabled the reset button as required.

2. Click the **BtnPick Button** control to execute the instructions within its **Click** event-handler



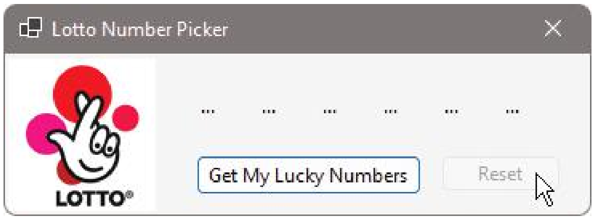
A series of numbers within the desired range is displayed, and the **Button** states have changed as required – a further series of numbers cannot be generated until the application has been reset.

3. Make a note of the numbers generated in this first series for comparison later

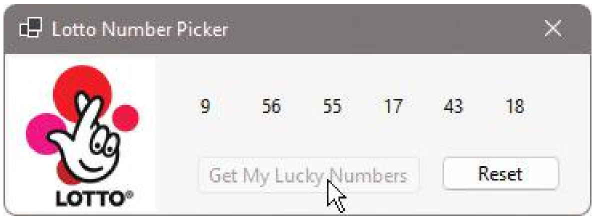
4. Click the **BtnReset** control to execute the instructions within that **Click** event-handler and see the application return to its initial start-up appearance as required

Hot tip

Notice that no number is repeated in any series.

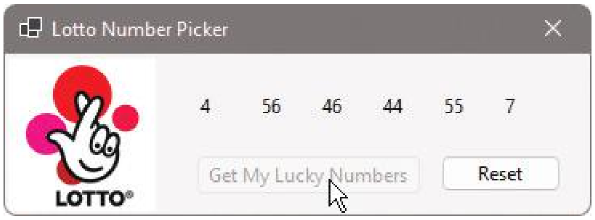


5. Click the **BtnPick Button** control again to execute its **Click** event-handler code a second time



Another series of numbers within the desired range is displayed and are different to those in the first series when compared – good, the numbers are being randomized as required.

6. Click the **Stop Debugging** button, then click the **Start** button to restart the application and click the **BtnPick Button** control once more



The generated numbers in this first series of numbers are different to those noted in the first series the last time the application ran – great, the random number generator is not repeating the same sequence of number series each time the application runs.

Don't forget

Remember that this application interface has no **Minimize** or **Maximize** buttons because their properties were set to false – see page 159, Step 11.

Hot tip

Alternatively, you can click the app's **X** button to close the application and stop debugging.

Publishing the application

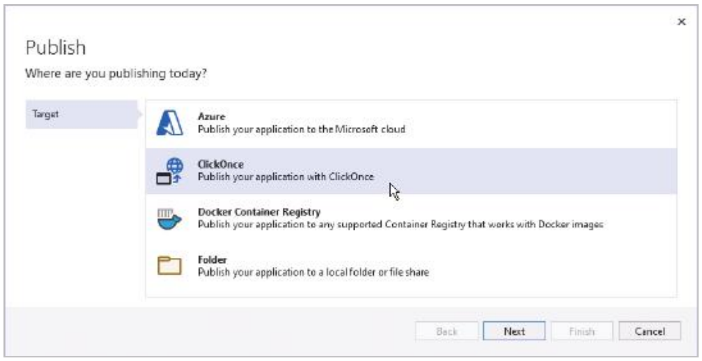
Having tested the application on pages 164-165, you can now create a standalone version for installation outside the Visual Studio IDE, and for deployment elsewhere.

Lotto
(continued)

Hot tip

Applications signing is optional, but does verify authentication. Find more details online at **docs.microsoft.com/en-us/visualstudio/ide/how-to-sign-application-and-deployment-manifests**

1. On the Menu Bar, click **Build**, **Publish Selection**, then select the **ClickOnce** option and click **Next**

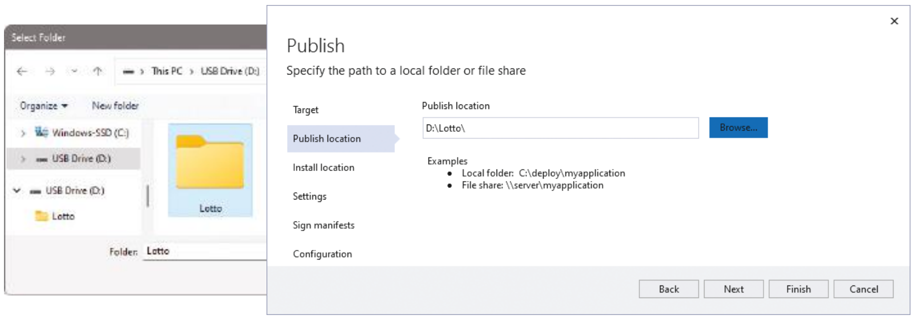


Publish
Where are you publishing today?

Target

- **Azure** – Publish your application to the Microsoft cloud
- **ClickOnce** – Publish your application with ClickOnce
- **Docker Container Registry** – Publish your application to any supported Container Registry that works with Docker images
- **Folder** – Publish your application to a local folder or file share

2. Now, use the wizard's **Browse** button to select a location where you wish to publish the application – for example, into a folder created on a USB drive, then click **Next**



Select Folder
This PC › USB Drive (D:)
Organize ▾ New folder
Windows-SSD (C:)
USB Drive (D:)
USB Drive (D:)
Lotto
Folder: Lotto

Publish
Specify the path to a local folder or file share

Publish location: D:\Lotto\

Examples
- Local folder: C:\deploy\myapplication
- File share: \\server\myapplication

3. Then, choose how the app will be installed – for example, from a USB drive, then click **Next**

4. Choose a version number for the app, then click **Next**

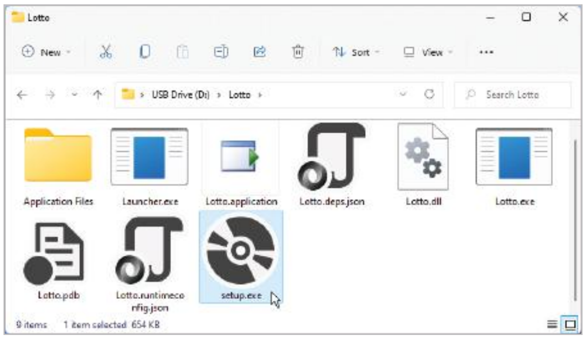
5. Check the **Sign the ClickOnce manifests** box, then click **Select from store** and choose a certificate, or click **Create Test Certificate** for temporary testing only

Don't forget

Each time you publish an application, its version number is automatically incremented – 1.0.0.0, 1.0.0.1, 1.0.0.2, etc.

6 Click the **Next** button to move to the final dialog page, confirm the listed choices, then click the **Finish** button

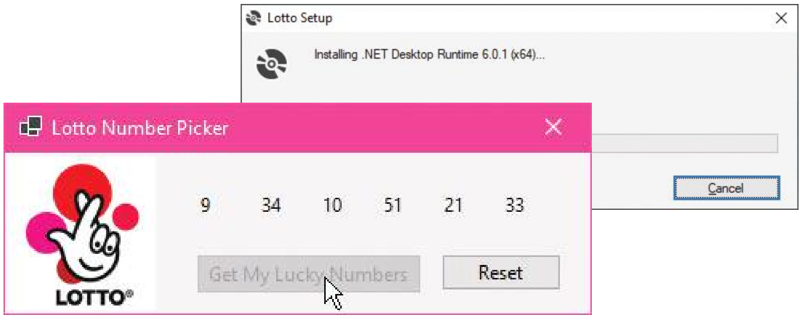
7 Finally, click the **Publish** button to publish the application at the specified location



The **Publish Wizard** generates a number of files, including a familiar "setup.exe" executable installer.

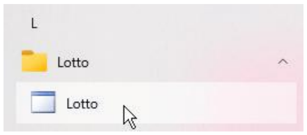
8 Move the portable media to the system where it is to be deployed, then run **setup.exe** to install the application

The installer will install the targeted .NET runtime if required and add a shortcut to the **All apps** menu that can be used to launch the application. The user can then run the application just as it performed during testing in the Visual Studio IDE.



Installing .NET Desktop Runtime 6.0.1 (x64)...

9 34 10 51 21 33



The installer also adds an item to the client system's **Add/Remove Programs** list, which can be used to uninstall the application – just like any other Windows program.

Summary

- Always make an initial program plan, to avoid the need for time-consuming changes later.
- A program plan should clearly define the program purpose, functionality required, and components needed.
- Fixed properties that will not change when the application is running can be set at design time in the **Properties** Window.
- The **Import** button in the **Select Resources** dialog can automatically assign an image to a **PictureBox** control.
- An **AutoSize** property value makes Visual Studio automatically fit a control neatly around its content.
- The Form Designer's **Layout** toolbar contains useful features to quickly align and space multiple interface controls.
- **Snap Lines** help you to easily align a selected control to others in the interface at design time.
- Setting a Form's **Window Style**, **MaximizeBox**, and **MinimizeBox** properties to **False** removes those buttons.
- Dynamic properties that will change when the application is running can be initialized with the Form's **Load** event-handler.
- The pop-up **IntelliSense** window lets you easily add program code when using the **Code Editor**.
- Runtime functionality responds to user actions by changing dynamic properties.
- A **Debug** version of an application allows its functionality to be tested as the application is being created in text format.
- The **Build** process compiles a **Release** version of an application in binary format.
- The **Publish** process creates a final **Release** version with an installer so the application can be deployed elsewhere.
- Applications created with the Visual Studio IDE can be installed and uninstalled just like other Windows applications.

12 Targeting devices

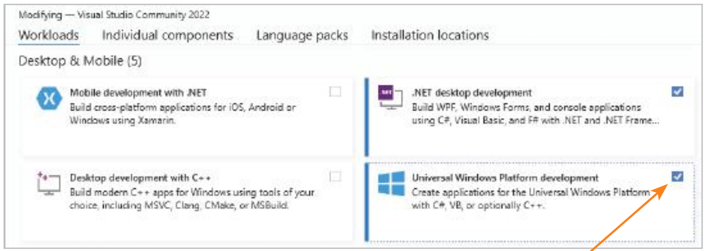
This chapter demonstrates how to create a Universal Windows Platform application using C# programming.

Starting a Universal project

Windows 10 introduced the **Universal Windows Platform** (UWP) that enables you to create a single **Universal Windows Application** (UWA) that will run on any modern Windows-based device. The interface layout of a UWA uses the **eXtensible Application Markup Language** (XAML) to specify components. In order to develop apps for the UWP, you should be running Windows 10 or Windows 11, and your Visual Studio IDE must include the **Universal Windows App Development Tools**:

Universal

1. Click **Visual Studio Installer** on the **All apps** menu, then click the **Modify** button on the Installer dialog



Modifying — Visual Studio Community 2022

Workloads | Individual components | Language packs | Installation locations

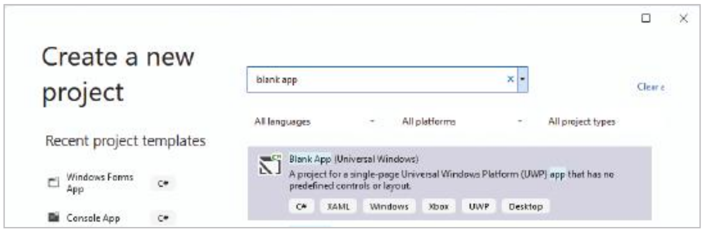
Desktop & Mobile (5)

- Mobile development with .NET — Build cross-platform applications for iOS, Android or Windows using Xamarin. ☐
- .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame... ☑
- Desktop development with C++ — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild. ☐
- Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, or optionally C++. ☑

2. See if the **Universal Windows Platform development** option box is checked in the **Desktop & Mobile** category on the **Workloads** tab – indicating this item is installed

3. If the box is not checked, check it now then click the **Modify** button at the bottom-right of the Installer dialog to download and install the **Universal Windows Platform development** tools

4. Select **File**, **New**, **Project** and create a new **C# Blank App (Universal Windows)** and name it "Universal" – for Windows, Desktop, Xbox, etc.



Create a new project

blank app

Clear

All languages | All platforms | All project types

Recent project templates

- Windows Forms App — C#
- Console App — C#

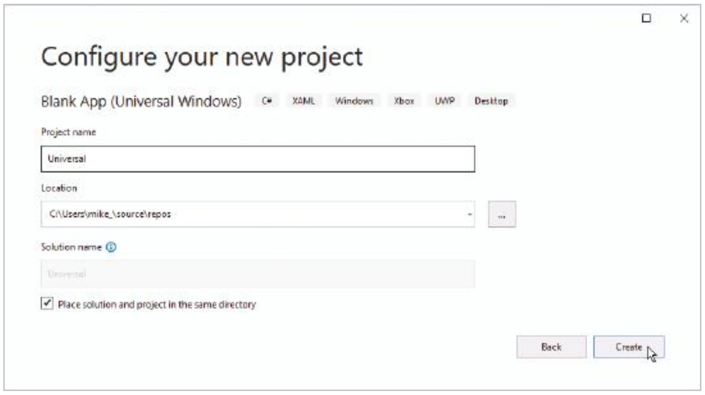
Blank App (Universal Windows)
A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.
C# | XAML | Windows | Xbox | UWP | Desktop

Beware

The example in this chapter is for Visual Studio 2022 on Windows 10 or 11 – it won't work in earlier Windows editions.

Don't forget

Depending upon your choices when you installed Visual Studio, you may see the options checked when the installer launches, to indicate you already have the **Universal Windows App development** tools.



Configure your new project

Blank App (Universal Windows)

Project name: Universal

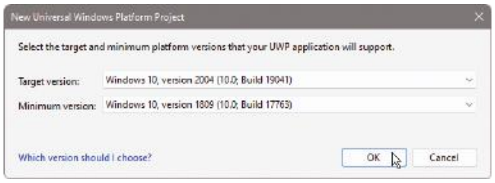
Location: C:\Users\mike_\source\repos

Solution name: Universal

[✓] Place solution and project in the same directory

In Windows you should ensure that the **Developer Mode** option is enabled in **Settings**, **Privacy & Security**, **For developers**.

5 Click **OK** to accept the suggested target versions



New Universal Windows Platform Project

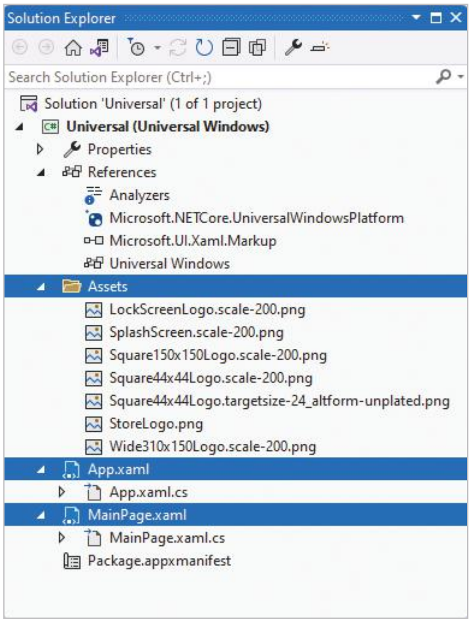
Select the target and minimum platform versions that your UWP application will support.

Target version: Windows 10, version 2004 (10.0; Build 19041)

Minimum version: Windows 10, version 1809 (10.0; Build 17763)

6 After Visual Studio creates the new project, select **View**, **Solution Explorer** to examine the generated files:

- A set of logo images in an **Assets** folder.
- Internal XAML and C# files for the **App**.
- XAML and C# files for the **MainPage** – here is where you will create interface components and functional code.
- Other miscellaneous **Package** files.



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'Universal' (1 of 1 project)
  Universal (Universal Windows)
    Properties
    References
      Analyzers
      Microsoft.NETCore.UniversalWindowsPlatform
      Microsoft.UI.Xaml.Markup
      Universal Windows
    Assets
      LockScreenLogo.scale-200.png
      SplashScreen.scale-200.png
      Square150x150Logo.scale-200.png
      Square44x44Logo.scale-200.png
      Square44x44Logo.targetsize-24_altform-unplated.png
      StoreLogo.png
      Wide310x150Logo.scale-200.png
    App.xaml
      App.xaml.cs
    MainPage.xaml
      MainPage.xaml.cs
    Package.appxmanifest
```

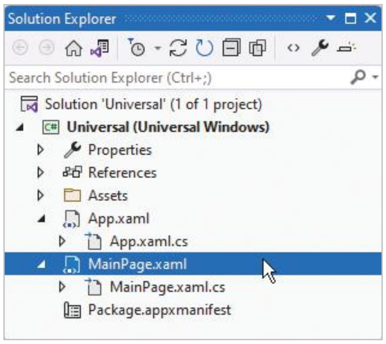
These files are essential to all UWP apps using C#, and exist in every project Visual Studio creates to target the Universal Windows Platform with C#.

Inserting page components

Universal (continued)

Visual Studio provides a two-part window to insert interface components into a UWP app. This comprises a **Design** view of the components, and a **XAML** view for the XAML code:

1. Open **Solution Explorer** then double-click on **MainPage.xaml** – to launch a two-part window

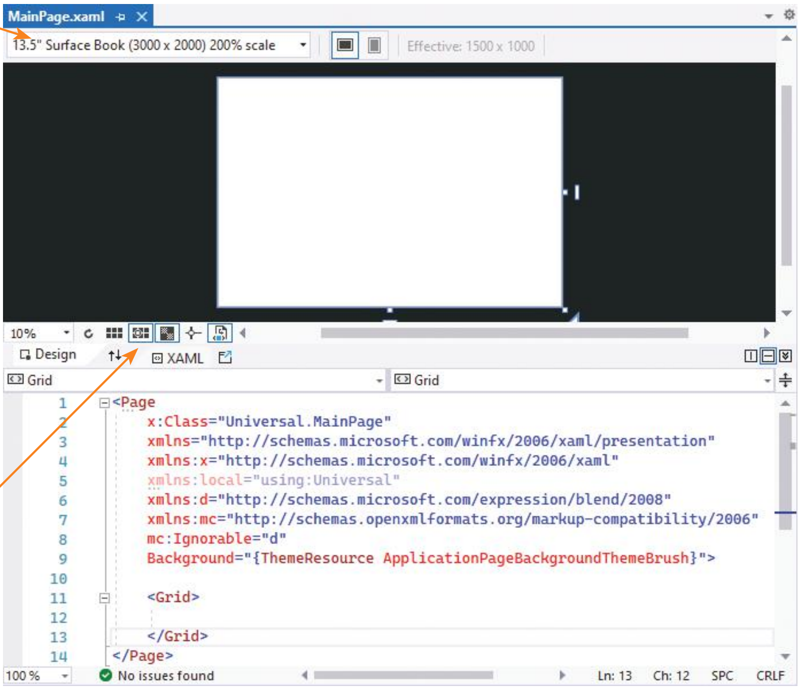


2. See that, by default, the **Design** view displays a blank canvas in **Landscape** mode, and the **Code** view contains some default code

You can select from a range of device emulators in this menu.

XAML is pronounced "zammel". You can write XAML code in the **Code** view to add components to the canvas previewed in **Design** view.

Explore these buttons to change the magnification and grid characteristics.



```
<Page
    x:Class="Universal.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Universal"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>

    </Grid>
</Page>
```

There is a pop-out **Toolbox** feature that lets you add components onto the canvas, but you will need to edit them in the XAML code later. In this example, the components are created in XAML code from the very start.

3 Now, see that by default, the **XAML** view reveals there are **<Grid> </Grid>** tags – this is the root element of the canvas in which you can add component elements

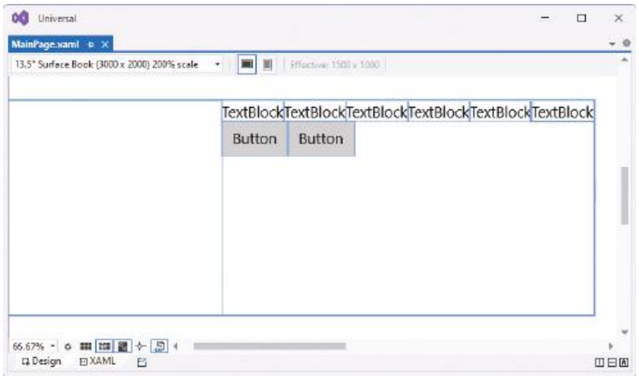
Component elements are best nested within a **<StackPanel>** element, as this can be given an **x:Name** for reference in functional code, and an **Orientation** attribute to specify the direction in which the nested elements should appear. Common component elements include **<Image>**, **<TextBox>**, **<TextBlock>** (label), and **<Button>**. Several **<StackPanel>** elements can be nested within each other to determine the **Horizontal** and **Vertical** layout of components:

4 Insert elements between the root **<Grid> </Grid>** tags, so the **XAML** code looks precisely like this:

```
<Grid>
 <StackPanel x:Name="MainStack" Orientation="Horizontal">
  <Image x:Name="Image" Width="200" Height="200" />
  <StackPanel x:Name="Controls" Orientation="Vertical">
   <StackPanel x:Name="Labels" Orientation="Horizontal">
    <TextBlock x:Name="textBlock1" Text="TextBlock" />
    <TextBlock x:Name="textBlock2" Text="TextBlock" />
    <TextBlock x:Name="textBlock3" Text="TextBlock" />
    <TextBlock x:Name="textBlock4" Text="TextBlock" />
    <TextBlock x:Name="textBlock5" Text="TextBlock" />
    <TextBlock x:Name="textBlock6" Text="TextBlock" />
   </StackPanel>
   <StackPanel x:Name="Buttons" Orientation="Horizontal">
    <Button x:Name="BtnPick" Content="Button" />
    <Button x:Name="BtnReset" Content="Button" />
   </StackPanel>
  </StackPanel>
 </StackPanel>
</Grid>
```

The outer horizontal **<StackPanel>** contains an **<Image>** element, and a nested vertical **<Stackpanel>** containing two further horizontal **<StackPanel>** elements. The **x:** prefix before the **Name** attribute refers to the XAML schema used by UWP apps. Each **<TextBlock>** element has a **Text** attribute that can be referenced in functional code, such as **textBlock1->Text**.

5 As you add the component elements in **XAML** view, they appear in the **Design** view until it looks like this:


Universal
MainPage.xaml
13.5" Surface Book (3000 x 2000) 200% scale
Effective: 1500 x 1000
TextBlock TextBlock TextBlock TextBlock TextBlock TextBlock
Button Button
66.67%
Design
XAML

Importing program assets

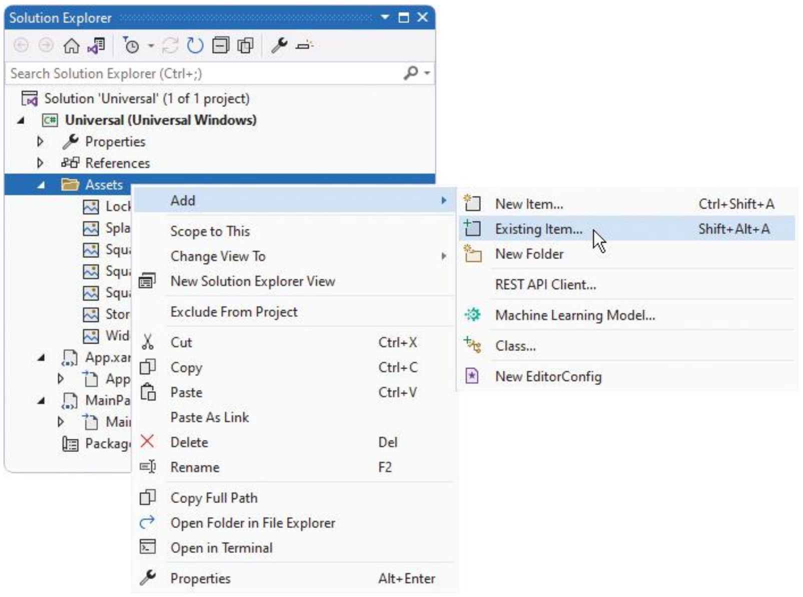
In order to have a XAML **<Image>** component display a graphic, an image file first needs to be added to the projects **Assets** folder. It can then be assigned to a **Source** attribute of the **<Image>** tag:

C#

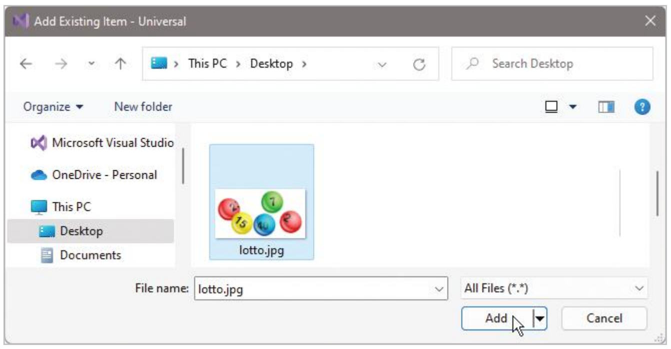
Universal (continued)

1. Open **Solution Explorer**, then right-click on the **Assets** folder and choose **Add** from the context menu

2. Now, choose **Existing Item...** from the next context menu – to open an **Add Existing Item** dialog box



3. In the **Add Existing Item** dialog, browse to the location of an image, then select the file and click the **Add** button

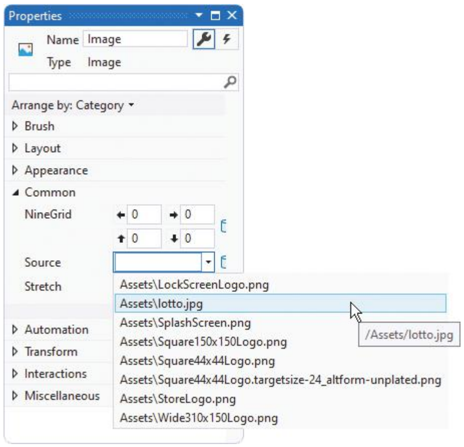


Hot tip

An image for display may be in any popular file format – such as **.bmp**, **.gif**, **.jpg**, **.png**, or **.tif**.

4 In **Solution Explorer**, the selected image file now appears in the project's **Asset** folder

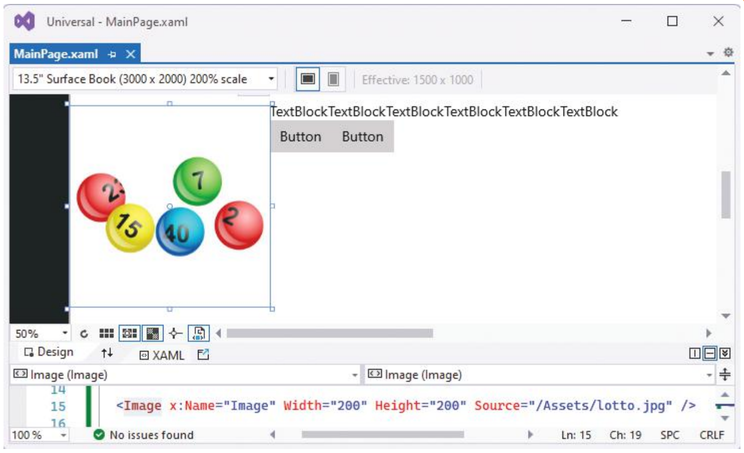
5 Select the Image component in **Designer** view, then click **View**, **Properties Window** to reveal its properties


Properties
Name Image
Type Image
Arrange by: Category
Brush
Layout
Appearance
Common
NineGrid 0 0 0 0
Source
Stretch
Automation
Transform
Interactions
Miscellaneous

Assets\LockScreenLogo.png
Assets\lotto.jpg
Assets\SplashScreen.png
Assets\Square150x150Logo.png
Assets\Square44x44Logo.png
Assets\Square44x44Logo.targetsize-24_altform-unplated.png
Assets\StoreLogo.png
Assets\Wide310x150Logo.png
/Assets/lotto.jpg

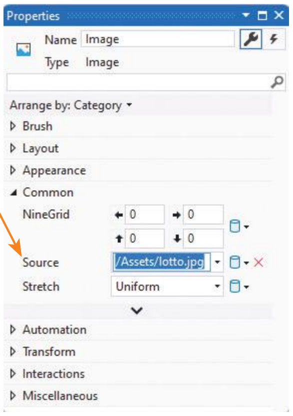
6 In the **Properties** window, expand the **Common** category, then click the **Source** item's arrow button and select the added image from the drop-down list

7 The image now appears in the **Design** view, and its path gets added to the **XAML** code and **Source** property


Universal - MainPage.xaml
MainPage.xaml
13.5" Surface Book (3000 x 2000) 200% scale
Effective: 1500 x 1000
TextBlockTextBlockTextBlockTextBlockTextBlockTextBlock
Button Button
50%
Design XAML
Image (Image) Image (Image)

```
<Image x:Name="Image" Width="200" Height="200" Source="/Assets/lotto.jpg" />
```

100 % No issues found Ln: 15 Ch: 19 SPC CRLF


Properties
Name Image
Type Image
Arrange by: Category
Brush
Layout
Appearance
Common
NineGrid 0 0 0 0
Source /Assets/lotto.jpg
Stretch Uniform
Automation
Transform
Interactions
Miscellaneous

Explore the **Appearance** and **Transform** options in an image's **Properties** window, to discover how you can modify how it will be displayed.

Designing the layout

To complete the app's layout, design attributes can be added to the XAML element tags to specify what they will display and precisely where in the interface they will appear.

A single **Margin** value sets all four margins around a component. You can specify two values to set left and right, and top and bottom margins – for example, **Margin = "10,30"**. Alternatively, you can specify four values to set left, top, right, bottom margins individually – e.g. **Margin = "10,30,10,50"**.

C#

Universal (continued)

1. Open **MainPage.xaml**, then add two attributes to the outer **MainStack <StackPanel>** element to fix its position
   HorizontalAlignment = "Left" VerticalAlignment = "Top"

2. Next, edit the **<Image>** element by modifying the initial assigned value of 200 – to increase its width
   Width = "300"

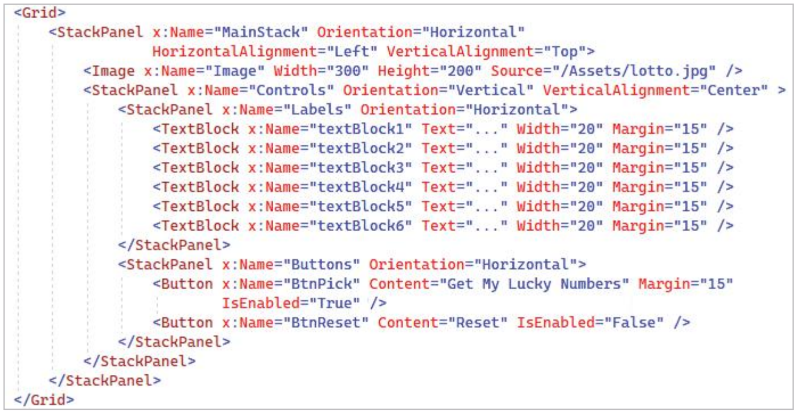
3. Now, add an attribute to the nested **Controls <StackPanel>** element to fix its position
   VerticalAlignment = "Center"

4. Then, edit all six **<TextBlock>** elements alike, to specify their initial content, width, and margin on all four sides
   Text = "..." Width = "20" Margin = "15"
   Text = "..." Width = "20" Margin = "15"
   Text = "..." Width = "20" Margin = "15"
   Text = "..." Width = "20" Margin = "15"
   Text = "..." Width = "20" Margin = "15"
   Text = "..." Width = "20" Margin = "15"

5. Edit the **BtnPick <Button>** element to specify its button label content, margin on all four sides, and initial state
   Content = "Get My Lucky Numbers" Margin = "15"
   IsEnabled = "True"

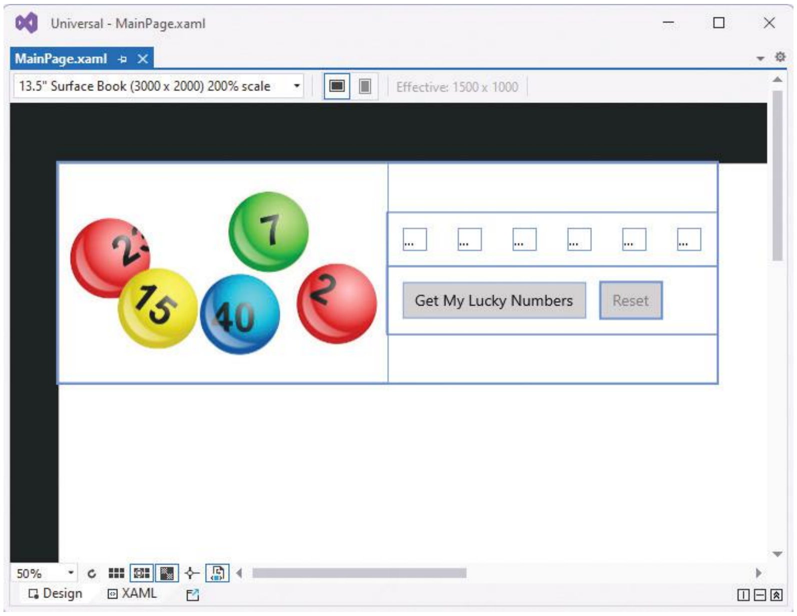
6. Edit the **BtnReset <Button>** element to specify its button label content, and initial state
   Content = "Reset" IsEnabled = "False"

The order in which the attributes appear in each element is unimportant, but the elements within the **MainPage.xaml** file should now look similar to the screenshot below:



```
<Grid>
    <StackPanel x:Name="MainStack" Orientation="Horizontal"
                HorizontalAlignment="Left" VerticalAlignment="Top">
        <Image x:Name="Image" Width="300" Height="200" Source="/Assets/lotto.jpg" />
        <StackPanel x:Name="Controls" Orientation="Vertical" VerticalAlignment="Center" >
            <StackPanel x:Name="Labels" Orientation="Horizontal">
                <TextBlock x:Name="textBlock1" Text="..." Width="20" Margin="15" />
                <TextBlock x:Name="textBlock2" Text="..." Width="20" Margin="15" />
                <TextBlock x:Name="textBlock3" Text="..." Width="20" Margin="15" />
                <TextBlock x:Name="textBlock4" Text="..." Width="20" Margin="15" />
                <TextBlock x:Name="textBlock5" Text="..." Width="20" Margin="15" />
                <TextBlock x:Name="textBlock6" Text="..." Width="20" Margin="15" />
            </StackPanel>
            <StackPanel x:Name="Buttons" Orientation="Horizontal">
                <Button x:Name="BtnPick" Content="Get My Lucky Numbers" Margin="15"
                        IsEnabled="True" />
                <Button x:Name="BtnReset" Content="Reset" IsEnabled="False" />
            </StackPanel>
        </StackPanel>
    </StackPanel>
</Grid>
```

As you make changes to the **XAML** code, the component layout gets changed accordingly in the **Design** view, and should now look like this:



Hot tip

You can optionally add **Margin = "0"** attributes to explicitly require elements to have no margin width.

Adding runtime function

Having completed the application component layout with XAML elements on pages 176-177, you are now ready to add functionality with C# programming code:

C#

Universal (continued)

1. In **Design** view, double-click on the **BtnPick** button

2. The **MainPage.xaml.cs** code-behind page opens in the **Code Editor** at a generated **BtnPick_Click** event-handler

3. In the **BtnPick_Click** event-handler block, insert these statements to create a randomized array of integers between 1 and 59

```
Random rnd = new Random( ) ;
int [ ] seq = new int[ 59 ] ;
for ( int i = 1 ; i < 60 ; i++ ) seq[ i - 1 ] = i ;
for ( int i = 0 ; i < 59 ; i++ )
{
  int j = ( rnd.Next( ) % 59 ) ;
  int k = seq[ i ] ; seq[ i ] = seq[ j ] ; seq[ j ] = k ;
}
// Statements to be inserted here (Steps 4-5).
```

4. Next, insert statements to assign six array element values to the **<TextBlock>** components

```
textBlock1.Text = seq[ 1 ].ToString( ) ;
textBlock2.Text = seq[ 2 ].ToString( ) ;
textBlock3.Text = seq[ 3 ].ToString( ) ;
textBlock4.Text = seq[ 4 ].ToString( ) ;
textBlock5.Text = seq[ 5 ].ToString( ) ;
textBlock6.Text = seq[ 6 ].ToString( ) ;
```

5. Next, insert statements to set the **<Button>** states

```
BtnPick.IsEnabled = false ;
BtnReset.IsEnabled = true ;
```

6. Return to **MainPage.xaml**, then in **Design** view, double-click on the **BtnReset** button

7. The **MainPage.xaml.cs** code-behind page opens in the **Code Editor** at a generated **BtnReset_Click** event-handler

Hot tip

The randomizer routine uses exactly the same logic as that of the **Windows Forms Application** example – see page 162.

Don't forget

There is no **Label** component in UWP apps; it is called a **TextBlock** instead.

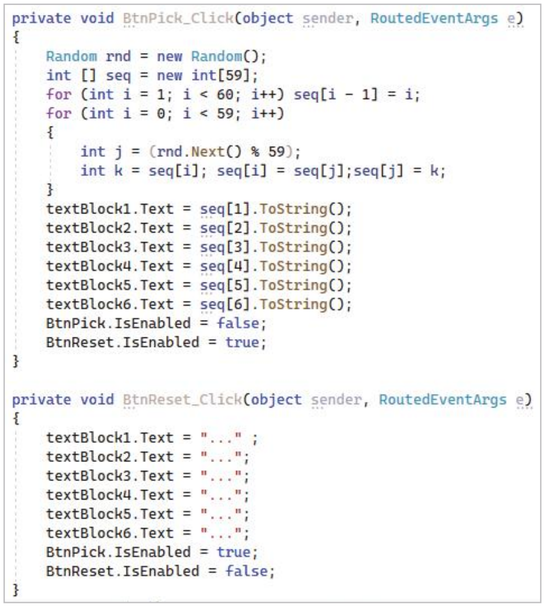
Beware

There is no **Enabled** property in UWP apps; it is called **IsEnabled** instead.

8. In the **BtnReset_Click** event-handler block, insert statements to assign strings to the **<TextBlock>** components
   textBlock1.Text = "..." ;
   textBlock2.Text = "..." ;
   textBlock3.Text = "..." ;
   textBlock4.Text = "..." ;
   textBlock5.Text = "..." ;
   textBlock6.Text = "..." ;
   // Statements to be inserted here (Step 9).

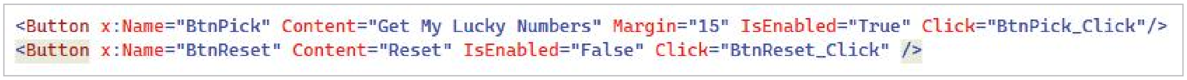
9. Finally, insert statements to set the **<Button>** states
   BtnPick.IsEnabled = true ;
   BtnReset.IsEnabled = false ;

The functions in the **MainPage.xaml.cs** code-behind page should now look like the screenshot below:



```
private void BtnPick_Click(object sender, RoutedEventArgs e)
{
    Random rnd = new Random();
    int [] seq = new int[59];
    for (int i = 1; i < 60; i++) seq[i - 1] = i;
    for (int i = 0; i < 59; i++)
    {
        int j = (rnd.Next() % 59);
        int k = seq[i]; seq[i] = seq[j];seq[j] = k;
    }
    textBlock1.Text = seq[1].ToString();
    textBlock2.Text = seq[2].ToString();
    textBlock3.Text = seq[3].ToString();
    textBlock4.Text = seq[4].ToString();
    textBlock5.Text = seq[5].ToString();
    textBlock6.Text = seq[6].ToString();
    BtnPick.IsEnabled = false;
    BtnReset.IsEnabled = true;
}

private void BtnReset_Click(object sender, RoutedEventArgs e)
{
    textBlock1.Text = "..." ;
    textBlock2.Text = "...";
    textBlock3.Text = "...";
    textBlock4.Text = "...";
    textBlock5.Text = "...";
    textBlock6.Text = "...";
    BtnPick.IsEnabled = true;
    BtnReset.IsEnabled = false;
}
```

10. Return to the **MainPage.xaml** file, then in **XAML** view, see that attributes have been automatically added to the **<Button>** elements to call the event-handler code



```
<Button x:Name="BtnPick" Content="Get My Lucky Numbers" Margin="15" IsEnabled="True" Click="BtnPick_Click"/>
<Button x:Name="BtnReset" Content="Reset" IsEnabled="False" Click="BtnReset_Click" />
```

Beware

The **BtnReset** button simply returns the **<TextBox>** and **<Button>** components to their original states.

Hot tip

Notice that the first **for** loop contains only one statement to be executed on each iteration, so braces are not required.

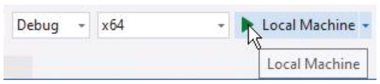
Testing the program

Having added functionality with C# code on pages 178-179, you are now ready to test the program for two devices:

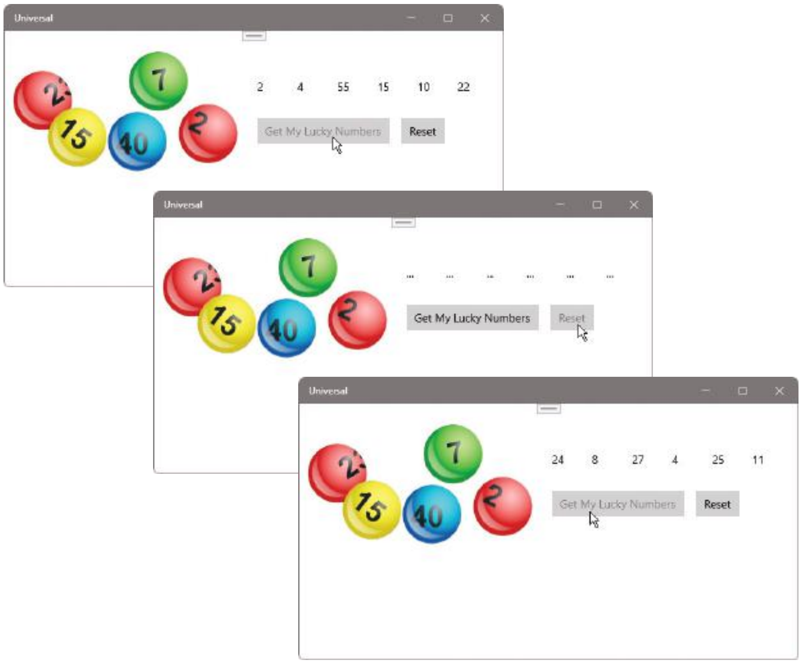
C#

Universal (continued)

1. On the Visual Studio standard toolbar, select **Debug** for **x64** architecture and **Local Machine** options, then click the **Start** button to run the app with debugging enabled

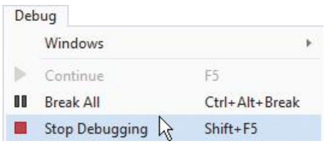


2. Wait while the application gets built and loaded, then click the buttons to try out their functionality



The app looks good – numbers are being randomized and the button states are changing as required.

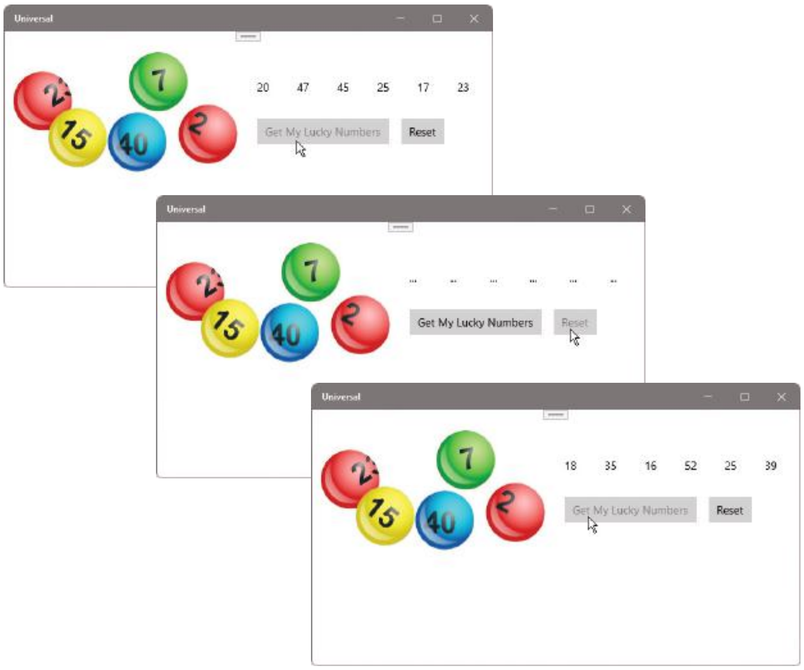
3. Now, on the Visual Studio standard toolbar, select **Debug**, **Stop Debugging** to exit the running program



Don't forget

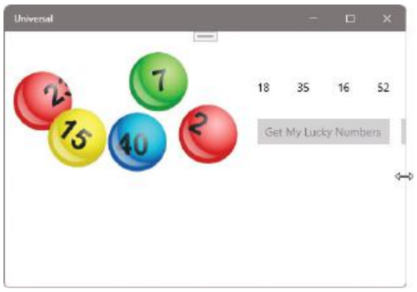
You must have your PC set to **Developer Mode** in **Settings**, **Privacy & Security**, **For developers**.

4 Click the **Start** button to restart the app, then click the buttons and compare the sequences of randomized numbers to those generated in the previous test



Great – the random number generator is seeded with the current time to ensure a different sequence each time the app gets run.

5 Drag the window edge to reduce its width, to simulate how the app would look on a device with a narrower screen



This is unsatisfactory – adjustments will be needed to the interface layout so it will adapt to suit narrower display areas.

Hot tip

Test on the emulator with lowest memory and smallest screen, and it should be fine running on devices with better features.

Adjusting the interface

The app test for mobile devices on pages 180-181 failed to satisfactorily present the controls, as the interface is too wide for small screen devices. Happily, the interface can be made to adapt to different screen sizes so it can also look good on mobile devices. The adaptation relies upon recognizing the screen size and changing the orientation of a **<StackPanel>** element in **XAML** for narrow screen devices:

C#

Universal (continued)

Hot tip

XAML code recognizes the same **<!-- -->** comment tags that are used in HTML code.

1. Open **MainPage.xaml**, then in **XAML** view add these elements immediately below the opening **<Grid>** element and before the component elements

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>

  <!-- Elements to be inserted here (Steps 2-3) -->

  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

2. Next, insert elements to recognize wide screens

```
<VisualState x:Name = "wideState" >

  <VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth = "641" />
  </VisualState.StateTriggers>

</VisualState>
```

Don't forget

Remember that the outer **<StackPanel>** in this app contains an **<Image>** and a nested **<StackPanel>** displayed horizontally, side by side. If displayed vertically, they should appear one above the other.

3. Now, insert elements to recognize narrow screens, and to change the **Orientation** of the outer **<StackPanel>**

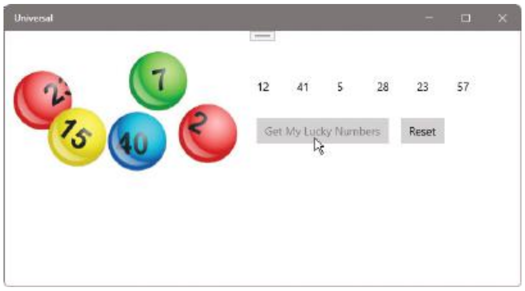
```
<VisualState x:Name = "narrowState" >

  <VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth = "0" />
  </VisualState.StateTriggers>

  <VisualState.Setters>
    <Setter
     Target = "MainStack.Orientation" Value = "Vertical" />
  </VisualState.Setters>

</VisualState>
```

The beginning of the **MainPage.xaml** file should now look similar to the screenshot at the top of the opposite page:

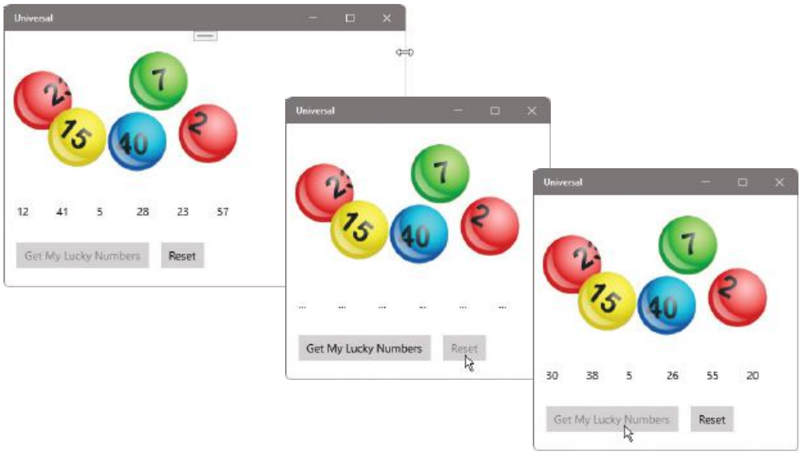
...cont'd

```
<VisualStateManager.VisualStateGroups >
    <VisualStateGroup>
        <VisualState x:Name="wideState">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="641" />
            </VisualState.StateTriggers>
        </VisualState>
        <VisualState x:Name="narrowState">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="0" />
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="MainStack.Orientation" Value="Vertical" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

4 Select **x64** and **Local Machine** to run the app once more – it still looks and functions well



5 Now, drag the window edge to reduce its width, and see the interface adapt to suit a narrower screen



Hot tip

You can have Visual Studio nicely format the XAML code by pressing **Ctrl** + **K**, **Ctrl** + **D**.

Beware

Although an app may work well on an emulator, it is recommended you always test on actual devices before deployment.

Deploying the application

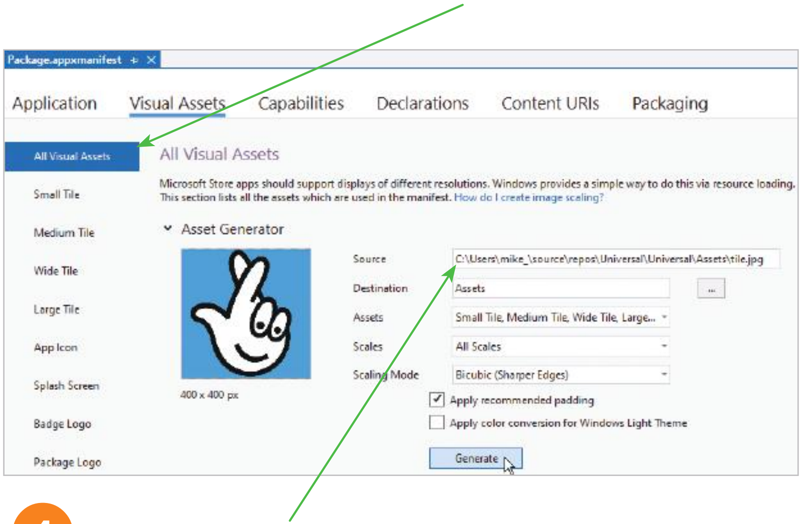
Universal (continued)

Having tested the app on page 183, it can now be deployed onto your PC. This will register the app on your PC and add it to the **All apps** menu, so an icon image can usefully be added to the program files to identify the app:

1. Add a 400 x 400 pixel icon image into the project's **Assets** folder on your PC

2. In **Solution Explorer**, right-click on the **Assets** folder and choose **Add, Existing Item** from the context menu and add the icon image

3. Next, double-click **Package.appxmanifest** and select the **Visual Assets tab** and the **All Visual Assets** item



Package.appxmanifest

Application | Visual Assets | Capabilities | Declarations | Content URIs | Packaging

All Visual Assets
Small Tile
Medium Tile
Wide Tile
Large Tile
App Icon
Splash Screen
Badge Logo
Package Logo

All Visual Assets

Microsoft Store apps should support displays of different resolutions. Windows provides a simple way to do this via resource loading. This section lists all the assets which are used in the manifest. How do I create image scaling?

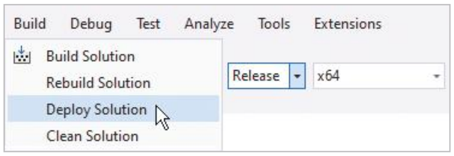
Asset Generator

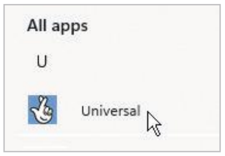
400 x 400 px

Source: C:\Users\mike_\source\repos\Universal\Universal\Assets\tile.jpg
Destination: Assets
Assets: Small Tile, Medium Tile, Wide Tile, Large...
Scales: All Scales
Scaling Mode: Bicubic (Sharper Edges)
[✓] Apply recommended padding
[ ] Apply color conversion for Windows Light Theme
Generate

4. Add the icon image as the **Source** and click the **Generate** button to produce variously-sized tiles and logo images

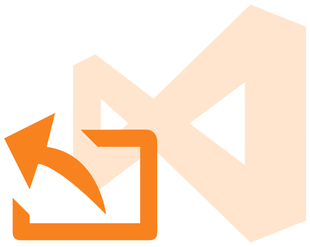
5. Change the build type to **Release**, then click **Build, Deploy Solution** to register the app on your PC and see it get added to your PC's **All apps** menu



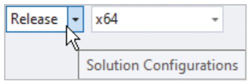


You can choose the **Remote Machine** option to test via a network connection.

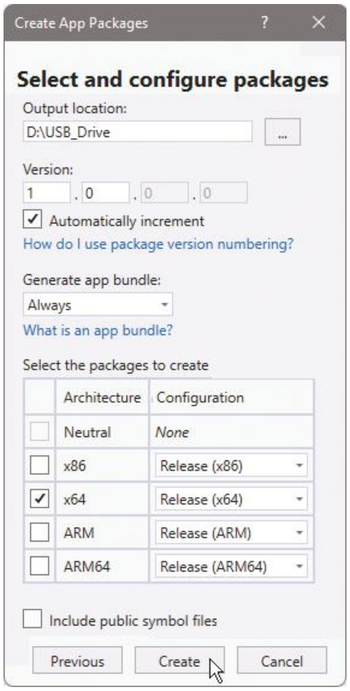
An **App Package** can be created for deployment on other devices running Windows version 1809 or later:



1. Select **Release** on the toolbar



2. In **Solution Explorer**, right-click on the top-level **Universal** project folder, then choose **Publish, Create App Packages** from the context menu to launch a wizard
3. In the wizard, choose the **Sideloading** and **Enable automatic updates** options, then click **Next**
4. Select a package signing certificate, then click **Next**
5. Choose a destination (e.g. a USB drive), version numbering, and architecture – then click **Create**



Create App Packages

Select and configure packages

Output location: D:\USB_Drive

Version: 1 . 0 . 0 . 0

[✓] Automatically increment

How do I use package version numbering?

Generate app bundle: Always

What is an app bundle?

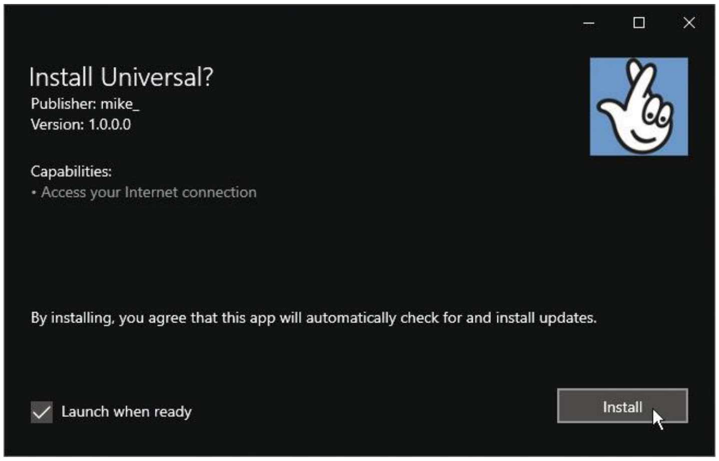
Select the packages to create

	Architecture	Configuration
☐	Neutral	None
☐	x86	Release (x86)
✓	x64	Release (x64)
☐	ARM	Release (ARM)
☐	ARM64	Release (ARM64)

☐ Include public symbol files

The build creates a folder of several items, including an Application Package file (**.msixbundle**) and a Security Certificate (**.cer**) file.

6. Move the folder to another device, then right-click on the Security Certificate file. Select **Install** then **Local Machine** and **Trusted People** to import the certificate
7. Finally, double-click the Application Package file to **Install** and **Launch** the app on that device



Install Universal?

Publisher: mike_

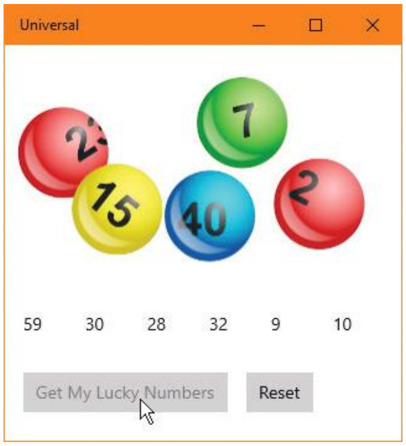
Version: 1.0.0.0

Capabilities:

• Access your Internet connection

By installing, you agree that this app will automatically check for and install updates.

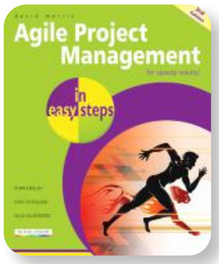
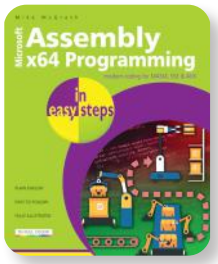
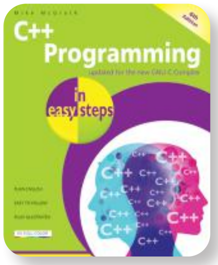
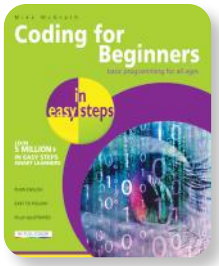
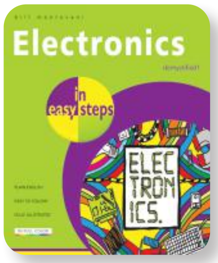
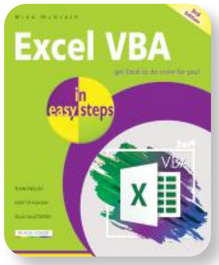
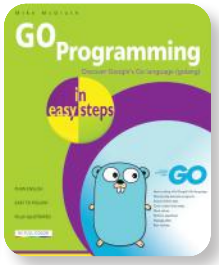
[✓] Launch when ready



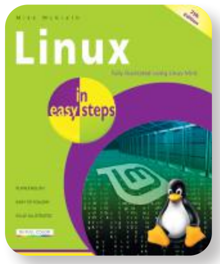
Summary

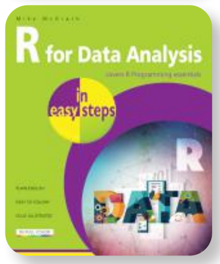
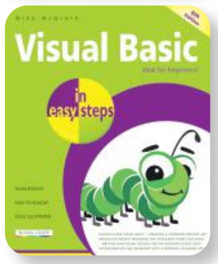
- The **Universal Windows Platform** (UWP) enables a single app to run on any modern Windows-based device.
- The **Universal Windows App Development Tools** are needed in order to develop UWP apps.
- The **C# Blank App (Universal Windows)** template can be used to create a new UWP project.
- The **Developer Mode** setting must be enabled in the Windows options in order to develop and test UWP apps.
- The **eXtensible Application Markup Language** (XAML) is used to specify components and layout on UWP apps.
- Visual Studio provides a graphical **Design** view and a text code **XAML** view for the **MainPage.xaml** file.
- Component elements can be placed within XAML **<StackPanel>** elements to control how they are laid out.
- Image files can be added to the **Assets** folder and assigned to XAML **<Image>** elements for display on the interface.
- Space can be added around a component by adding a **Margin** attribute and assigned value within its element tag.
- Functional C# programming code can be added to the **MainPage.xaml.cs** code-behind page.
- A UWP app can be tested in **Debug** mode on the **Local Machine**, an **Emulator**, and a connected **Device**.
- The interface of a UWP app can adapt to different screen sizes by changing the orientation of **<StackPanel>** elements.
- Image files can be added to the **Assets** folder for assignment as logos in the **Package.appmanifest** window.
- The **Release** version can be deployed by selecting the target configuration, then using the **Build**, **Deploy Solution** menu.

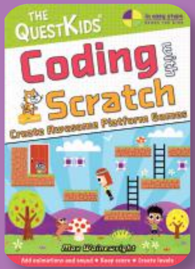
Index

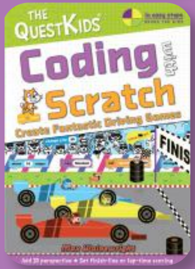

Agile Project Management
in easy steps

Microsoft Assembly x64 Programming
in easy steps

C++ Programming
in easy steps

Coding for Beginners
in easy steps

Electronics
in easy steps

Excel VBA
in easy steps

GO Programming
in easy steps

HTML, CSS & JavaScript
in easy steps

Java
in easy steps

Linux
in easy steps

PHP
in easy steps

R for Data Analysis
in easy steps

Visual Basic
in easy steps


The QuestKids
Coding with Scratch
Create Awesome Platform Games

The QuestKids
Coding with HTML & JavaScript
Create Epic Computer Games

The QuestKids
Coding with Scratch
Create Fantastic Driving Games

Why choose this book?

It's written in plain English
Put off by the jargon? Don't worry, we keep things straightforward.

It's easy to follow
Clear, step-by-step instructions make learning simple.

It's fully illustrated
We don't just tell you how to do it, we also **show** you how.

It's in full color
This book's printed in color to make it simpler to use and easier on the eye.

And it's fantastic value
Do we need to say any more?

C# Programming in easy steps will teach you every aspect of the C# language you'll need to produce professional programming results. Assuming no prior knowledge, it shows how to install the free edition of Visual Studio and create an environment to start executing your own programs. By using clear syntax-highlighted code, which is free to download, this primer:

- Demonstrates all the C# language basics including variables, arrays, logic, looping, methods and classes.
- Covers Object Oriented Programming principles.
- Shows how to use this acquired knowledge to create graphic programs for traditional PC Desktop apps, and also as Universal apps for multiple devices.
- Includes a chapter on detecting and debugging errors in C# programs.

A trusted source for anyone who wants to learn C# programming efficiently and effectively!

Let these icons make it even easier

Hot tip — to spice up your learning

Don't forget — highlights something worth remembering

Beware — wards you off potential danger!

£11.99 UK / $17.99 US

ISBN 978-1-84078-973-7
US$17.99
51799>
9 781840 789737

Categories: Computers/Programming

www.ineasysteps.com