Module 1: Review of Visual C# Syntax

Contents:

Module overview

The Microsoft® .NET Framework version 4.7 provides a comprehensive development platform that you can use to build, deploy, and manage applications and services. By using the .NET Framework, you can create visually compelling applications, enable seamless communication across technology boundaries, and provide support for a wide range of business processes.

In this module, you will learn about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You will also learn about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

Objectives

After completing this module, you will be able to:

- Describe the architecture of .NET Framework applications and the features that

Visual Studio 2017 and Visual C# provide.

- Use basic Visual C# data types, operators, and expressions.
- Use standard Visual C# constructs.

Lesson 1: Overview of Writing Application by Using Visual C#

The .NET Framework 4.7 and Visual Studio provide many features that you can use when developing your applications.

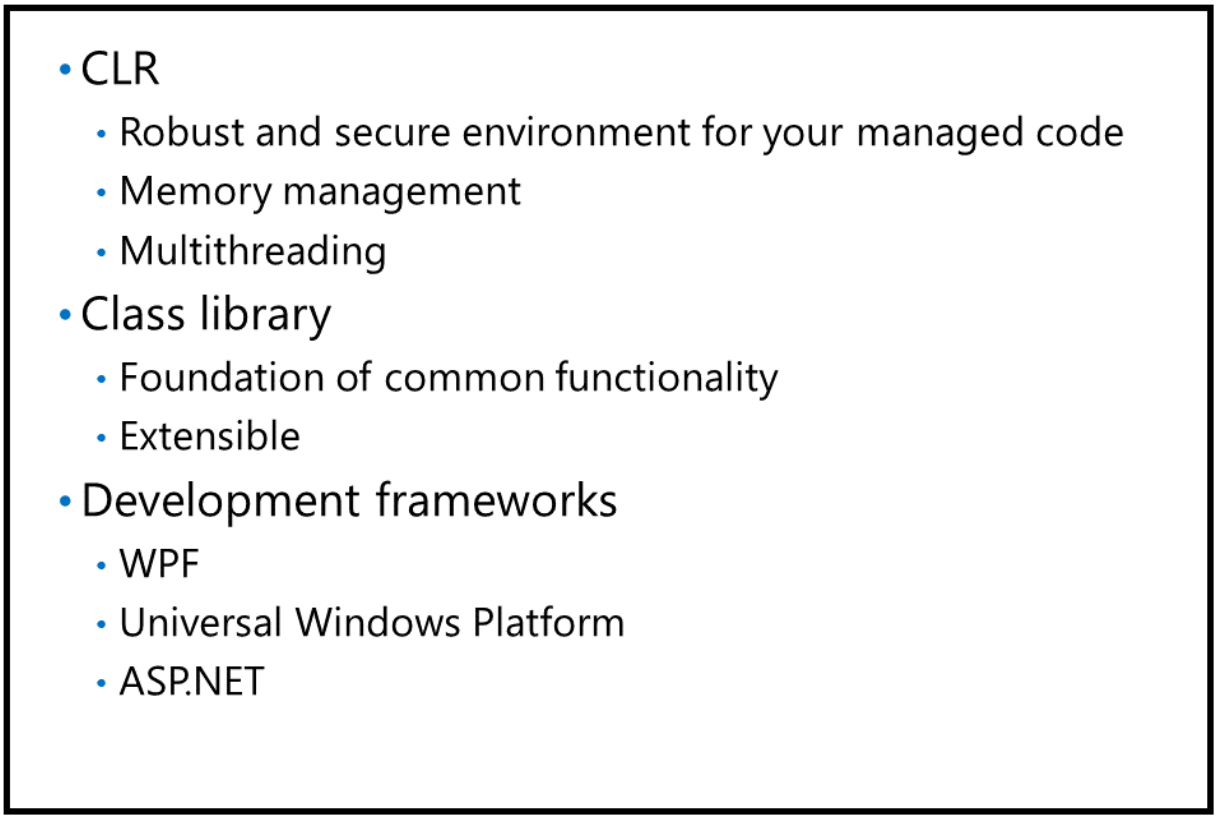
In this lesson, you will learn about the features that Visual Studio 2017 and the .NET Framework 4.7 provide that enable you to create your own applications.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the purpose of the .NET Framework.
- Describe the key features of Visual Studio 2017.
- Describe the project templates provided in Visual Studio 2017.
- Create a .NET Framework application.
- Describe XAML.

What Is the .NET Framework?



The .NET Framework 4.7 provides a comprehensive development platform that offers a fast and efficient way to build applications and services. By using Visual Studio 2017, you can use the .NET Framework 4.7 to create a wide range of solutions that operate across a broad range of computing devices.

The .NET Framework 4.7 provides three principal elements:

- The Common Language Runtime (CLR).
- The .NET Framework class library.
- A collection of development frameworks.

The Common Language Runtime

The .NET Framework provides an environment called the CLR. The CLR manages the execution of code and simplifies the development process by providing a robust and highly secure execution environment that includes:

- Memory management.

- Transactions.
- Multithreading.

The .NET Framework Class Library

The .NET Framework provides a library of reusable classes that you can use to build applications. The classes provide a foundation of common functionality and constructs that help to simplify application development by, in part, eliminating the need to constantly reinvent logic. For example, the **System.IO.File** class contains functionality that enables you to manipulate files on the Windows file system. In addition to using the classes in the .NET Framework class library, you can extend these classes by creating your own libraries of classes.

Development Frameworks

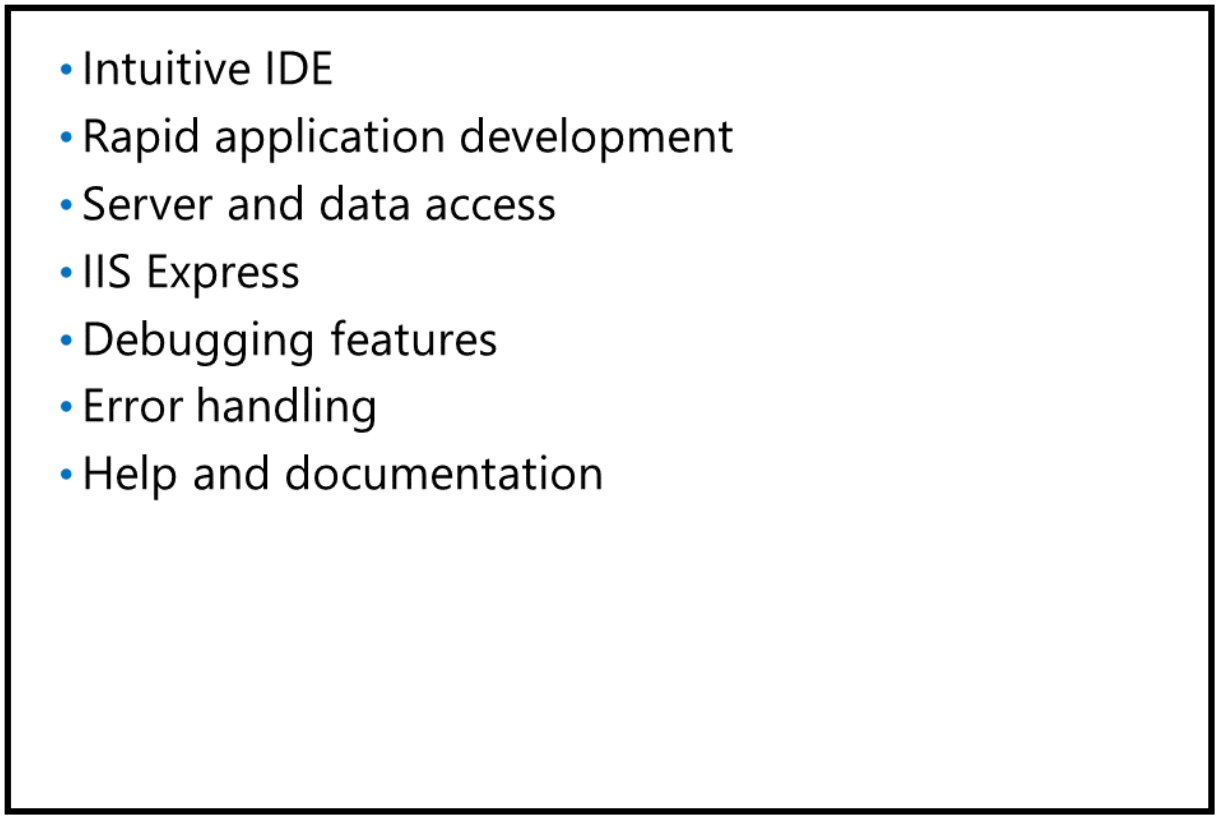
The .NET Framework provides several development frameworks that you can use to build common application types, including:

- Desktop client applications, by using Windows Presentation Foundation (WPF).
- Universal Windows Platform (UWP) applications, by using XAML.
- Server-side web applications, by using Microsoft ASP.NET Web Applications or ASP.NET MVC.
- Service-oriented web applications, by using ASP.NET Web API.
- Long-running applications, by using Windows services.

Each framework provides the necessary components and infrastructure to get you started.

> **Additional Reading:** For more information about the .NET Framework, see the **Overview of the .NET Framework** page at **http://go.microsoft.com/fwlink/?LinkID=267639**.

Key Features of Visual Studio 2017



Visual Studio 2017 provides a single development environment that enables you to rapidly design, implement, build, test, and deploy various types of applications and components by using a range of programming languages.

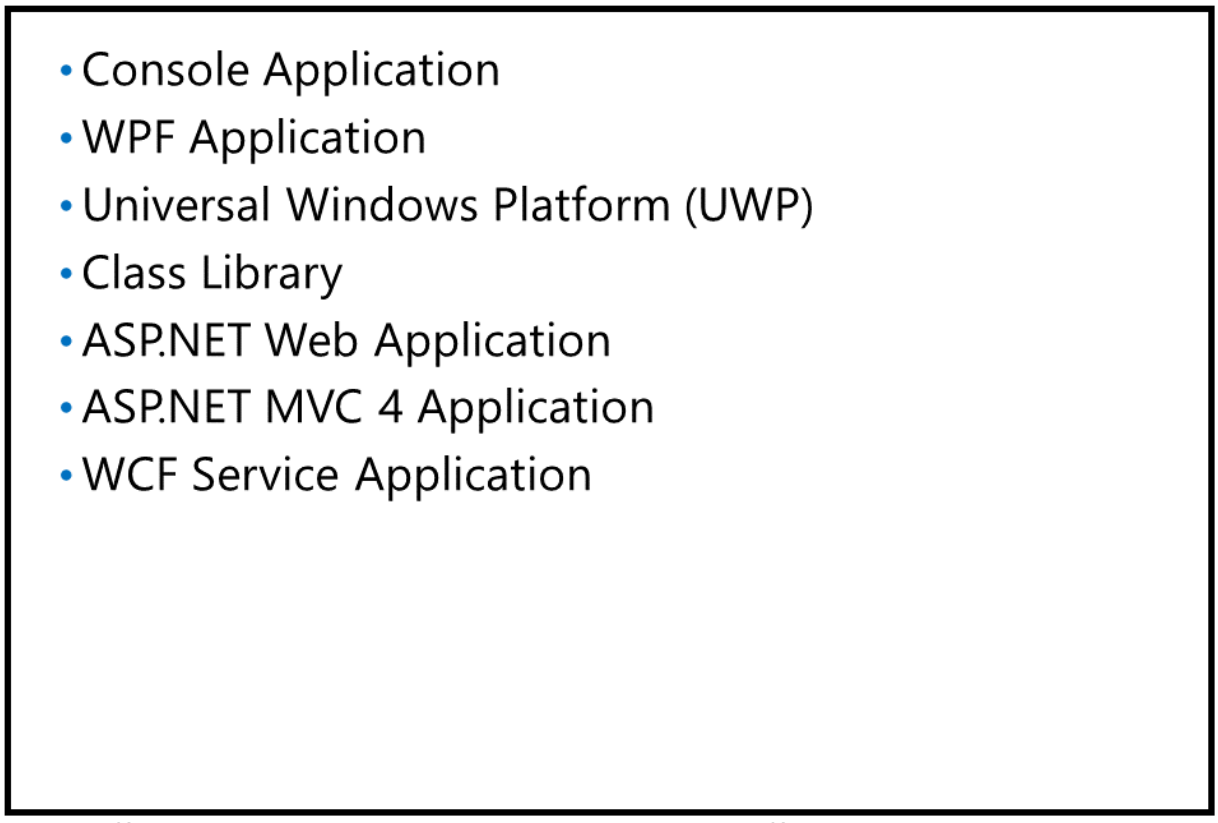
Some of the key features of Visual Studio 2017 are:

- Intuitive integrated development environment (IDE). The Visual Studio 2017 IDE provides all of the features and tools that are necessary to design, implement, build, test, and deploy applications and components.
- Rapid application development. Visual Studio 2017 provides design views for graphical components that enable you to easily build complex user interfaces. Alternatively, you can use the Code Editor views, which provide more control but are not as easy to use. Visual Studio 2017 also provides wizards that help speed up the development of particular components.
- Server and data access. Visual Studio 2017 provides the Server Explorer, which enables you to log on to servers and explore their databases and system services. It also provides a familiar way to create, access, and modify databases that your application uses by using the new table designer.

- Internet Information Services (IIS) Express. Visual Studio 2017 provides a lightweight version of IIS as the default web server for debugging your web applications.
- Debugging features. Visual Studio 2017 provides a debugger that enables you to step through local or remote code, pause at breakpoints, and follow execution paths.
- Error handling. Visual Studio 2017 provides the **Error List** window, which displays any errors, warnings, or messages that are produced as you edit and build your code.
- Help and documentation. Visual Studio 2017 provides help and guidance through Microsoft IntelliSense®, code snippets, and the integrated help system, which contains documentation and samples.

> **Additional Reading:** For more information about what is new in Visual Studio 2017, see the **What's New in Visual Studio 2017** page at **https://aka.ms/moc-20483c-m1-pg1**

Templates in Visual Studio 2017



Visual Studio 2017 supports the development of different types of applications such as Windows-based client applications, web-based applications, services, and libraries. To help you get started, Visual Studio 2017 provides application templates that provide a structure for the different types of applications. These templates:

- Provide starter code that you can build on to quickly create functioning applications.
- Include supporting components and controls that are relevant to the project type.
- Configure the Visual Studio 2017 IDE to the type of application that you are developing.
- Add references to any initial assemblies that this type of application usually requires.

Types of Templates

The following table describes some of the common application templates that you might use when you develop .NET Framework applications by using Visual Studio 2017.

Template	Description
Console Application	Provides the environment settings, tools, project references, and starter code to develop an application that runs in a command-line interface. This type of application is considered lightweight because there is no graphical user interface.
WPF Application	Provides the environment settings, tools, project references, and starter code to build a rich graphical Windows application. A WPF application enables you to create the next generation of Windows applications, with much more control over user interface design.
UWP	Provides the environment settings, tools, project references, and starter code to build a rich graphical application targeted at the Windows 10 operating systems. UWP applications enable you to reuse skills obtained from WPF development by using XAML and Visual C#, but also from web development by using HTML 5, CSS 3.0, and JavaScript.

Template	Description
Class Library	Provides the environment settings, tools, and starter code to build a .dll assembly. You can use this type of file to store functionality that you might want to invoke from many other applications.
ASP.NET Web Application	Provides the environment settings, tools, project references, and starter code to create a server-side, compiled ASP.NET web application.
ASP.NET MVC 4 Application	Provides the environment settings, tools, project references, and starter code to create a Model-View-Controller (MVC) Web application. An ASP.NET MVC web application differs from the standard ASP.NET web application in that the application architecture helps you separate the presentation layer, business logic layer, and data access layer.
WCF Service Application	Provides the environment settings, tools, project references, and starter code to build Service Orientated Architecture (SOA) services.

Creating a .NET Framework Application

1. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project dialog** box, choose a template, location, name, and then click **OK**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
  class Program
  {
     static void Main(string[] args) { }
  }
}
```

The application templates provided in Visual Studio 2017 enable you to start creating an application with minimal effort. You can then add your code and customize the project to meet your own requirements.

The following steps describe how to create a console application:

1. Open **Visual Studio 2017**.
2. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
3. In the **New Project** dialog box, do the following:
   a. Expand **Templates**, **Visual C#**, and then click **Windows**.
   b. Click the **Console Application** template.
   c. In the **Name** box, specify a name for the project.
   d. In the **Location** box, specify the path where you want to save the project.
4. Click **OK**.
5. The **Code Editor** window now shows the default **Program** class, which contains the entry point method for the application.

The following code example shows the default **Program** class that Visual Studio provides when you use the **Console Application** template.

Program Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
   class Program
   {
       static void Main(string[] args)
       {
       }
   }
}
```

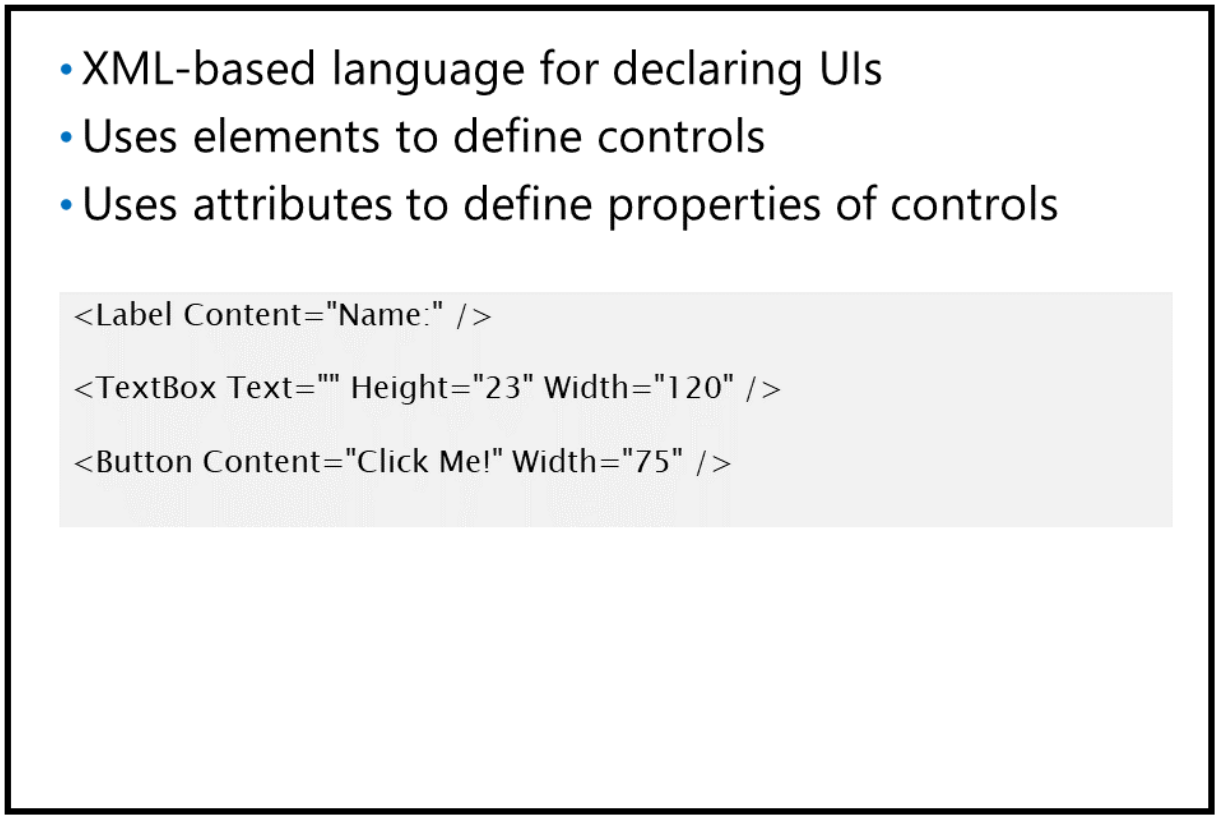
After you create a project, you can then use the features that Visual Studio provides to create your application.

Programmer Productivity Features

Visual Studio 2017 provides a host of features that can help you to write code. When writing code, you need to recall information about many program elements. Instead of manually looking up information by searching help files or other source code, the IntelliSense feature in Visual Studio provides the information that you need directly from the editor. IntelliSense provides the following features:

- The Quick Info option displays the complete declaration for any identifier in your code. Move the mouse so that the pointer rests on an identifier to display Quick Info for that identifier, which appears in a yellow pop-up box.
- The Complete Word option enters the rest of a variable, command, or function name after you have typed enough characters to disambiguate the term. Type the first few letters of the name and then press Alt+Right Arrow or Ctrl+Spacebar to complete the word.

Overview of XAML



Extensible Application Markup Language (XAML) is an XML-based language that you can use to define your .NET application UIs. By declaring your UI in XAML as opposed to writing it in code makes your UI more portable and separates your UI from your application logic.

XAML uses elements and attributes to define controls and their properties in XML syntax. When you design a UI, you can use the toolbox and properties pane in Visual Studio to visually create the UI, you can use the XAML pane to declaratively create the UI, you can use Microsoft Expression Blend, or you can use other third-party tools. Using the XAML pane gives you finer grained control than dragging controls from the toolbox to the window.

The following example shows the XAML declaration for a label, textbox, and button:

Defining Controls in XAML

```
<Label Content="Name:"/>
<TextBox Text="" Height="23" Width="120" />
<Button Content="Click Me!" Width="75" />
```

You can use XAML syntax to produce simple UIs as shown in the previous example or to create much more complex interfaces. The markup syntax provides the functionality to bind data to controls, to use gradients and textures, to use templates for application-wide formatting, and to bind events to controls in the window. The toolbox in Visual Studio also includes container controls that you can use to position and size your controls appropriately regardless of how your users resize their application window.

> **Additional Reading:** For more information about XAML, see Module 9 of this course.

Lesson 2: Data Types, Operators, and Expressions

All applications use data. This data might be supplied by the user through a user interface, from a database, from a network service, or from some other source. To store and use data in your applications, you must familiarize yourself with how to define and use variables and how to create and use expressions with the variety of operators that Visual C# provides.

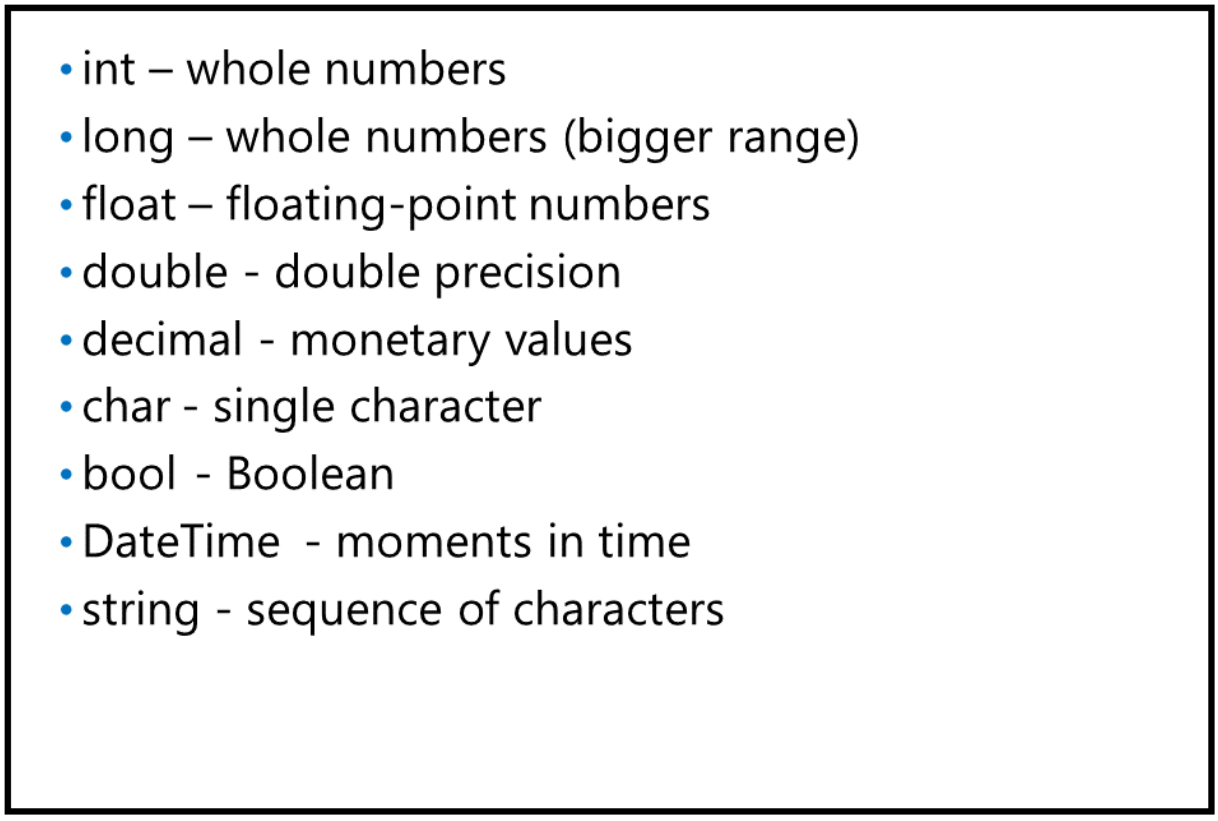
In this lesson, you will learn how to use some of the fundamental constructs in Visual C#, such as variables, type members, casting, and string manipulation.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the data types provided by Visual C#.
- Create and use expressions.
- Declare and assign variables.
- Access type members.
- Cast data from one type to another.
- Concatenate and validate strings.

What are Data Types?



A variable holds data of a specific type. When you declare a variable to store data in an application, you need to choose an appropriate data type for that data. Visual C# is a type-safe language, which means that the compiler guarantees that values stored in variables are always of the appropriate type.

Commonly Used Data Types

The following table shows the commonly used data types in Visual C#, and their characteristics.

Type	Description	Size (bytes)	Range
int	Whole numbers	4	–2,147,483,648 to 2,147,483,647
long	Whole numbers (bigger range)	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Floating-point numbers	4	+/–3.4 × 10^38

Type	Description	Size (bytes)	Range
double	Double precision (more accurate) floating-point numbers	8	+/–1.7 × 10^308
decimal	Monetary values	16	28 significant figures
char	Single character	2	N/A
bool	Boolean	1	True or false
DateTime	Moments in time	8	0:00:00 on 01/01/2001 to 23:59:59 on 12/31/9999
string	Sequence of characters	2 per character	N/A

Additional Reading: For more information about data types, see the **Reference Tables for Types (C# Reference)** page at **http://go.microsoft.com/fwlink/?LinkID=267770**.

Expressions and Operators in Visual C#

Example expressions:

- + operator

```
a + 1
```

- / operator

```
5 / 2
```

- + and – operators

```
a + b - 2
```

- + operator (string concatenation)

```
"ApplicationName: " + appName.ToString()
```

Expressions are a central component of practically every Visual C# application, because expressions are the fundamental constructs that you use to evaluate and

manipulate data. Expressions are collections of operands and operators, which you can define as follows:

- Operands are values, for example, numbers and strings. They can be constant (literal) values, variables, properties, or return values from method calls.
- Operators define operations to perform on operands, for example, addition or multiplication. Operators exist for all of the basic mathematical operations, as well as for more advanced operations such as logical comparison or the manipulation of the bits of data that constitute a value.

All expressions are evaluated to a single value when your application runs. The type of value that an expression produces depends on the types of the operands that you use and the operators that you use. There is no limit to the length of expressions in Visual C# applications, although in practice, you are limited by the memory of your computer and your patience when typing. However, it is usually advisable to use shorter expressions and assemble the results of expression-processing piecemeal. This makes it easier for you to see what your code is doing, as well as making it easier to debug your code.

Operators in Visual C#

Operators combine operands together into expressions. Visual C# provides a wide range of operators that you can use to perform most fundamental mathematical and logical operations. Operators fall into the following three categories:

- Unary. This type of operator operates on a single operand. For example, you can use the **-** operator as a unary operator. To do this, you place it immediately before a numeric operand, and it converts the value of the operand to its current value multiplied by –1.
- Binary. This type of operand operates on two values. This is the most common type of operator, for example, *****, which multiplies the value of two operands.
- Ternary. There is only one ternary operator in Visual C#. This is the **? :** operator that is used in conditional expressions.

The following table shows the operators that you can use in Visual C#, grouped by type.

Type	Operators
Order of operations	( )
Arithmetic	+, -, *, /, %
Increment, decrement	++, --
Comparison	==, !=, <, >, <=, >=, is, ??
String concatenation	+
Logical/bitwise operations	&, |, ^, !, ~, &&, ||
Indexing (counting starts from element 0)	[ ]
Casting	( ), as
Assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Bit shift	<<, >>
Type information	sizeof, typeof
Delegate concatenation and removal	+, -
Overflow exception control	checked, unchecked
Indirection and Address (unsafe code only)	*, ->, [ ], &
Conditional (ternary operator)	?:

Expression Examples

You can combine the basic building blocks of operators and operands to make expressions as simple or as complex as you like.

The following code example shows how to use the + operator.

+ Operator

```
a + 1
```

The **+** operator can operate on different data types, and the result of this expression depends on the data types of the operands. For example, if **a** is an integer, the result of the expression is an integer with the value 1 greater than **a**. If **a** is a double, the result is a double with the value 1 greater than **a**. The difference is subtle but important. In the second case (**a** is a double), the Visual C# compiler has to generate code to convert the constant integer value **1** into the constant double value **1** before the expression can be evaluated. The rule is that the type of the expression is the same as the type of the operands, although one or more of the operands might need to be converted to ensure that they are all compatible.

The following code example shows how to use the **/** operator to divide two **int** values.

/ Operator

```
5 / 2
```

The value of the result is the integer value 2 (not 2.5). If you convert one of the operands to a double, the Visual C# compiler will convert the other operand to a double, and the result will be a double.

The following code example shows how to use the **/** operator to divide a **double** value by an **int** value.

/ Operator

```
5.0 / 2
```

The value of the result now is the double value 2.5. You can continue building up expressions with additional values and operators.

The following code example shows how use the **+** and **–** operators in an expression.

+ and – Operators

```
a + b - 2
```

This expression evaluates to the sum of variables **a** and **b** with the value 2 subtracted from the result.

Some operators, such as **+**, can be used to evaluate expressions that have a range of types.

The following code example shows how to use the **+** operator to concatenate two **string** values.

+ Operator

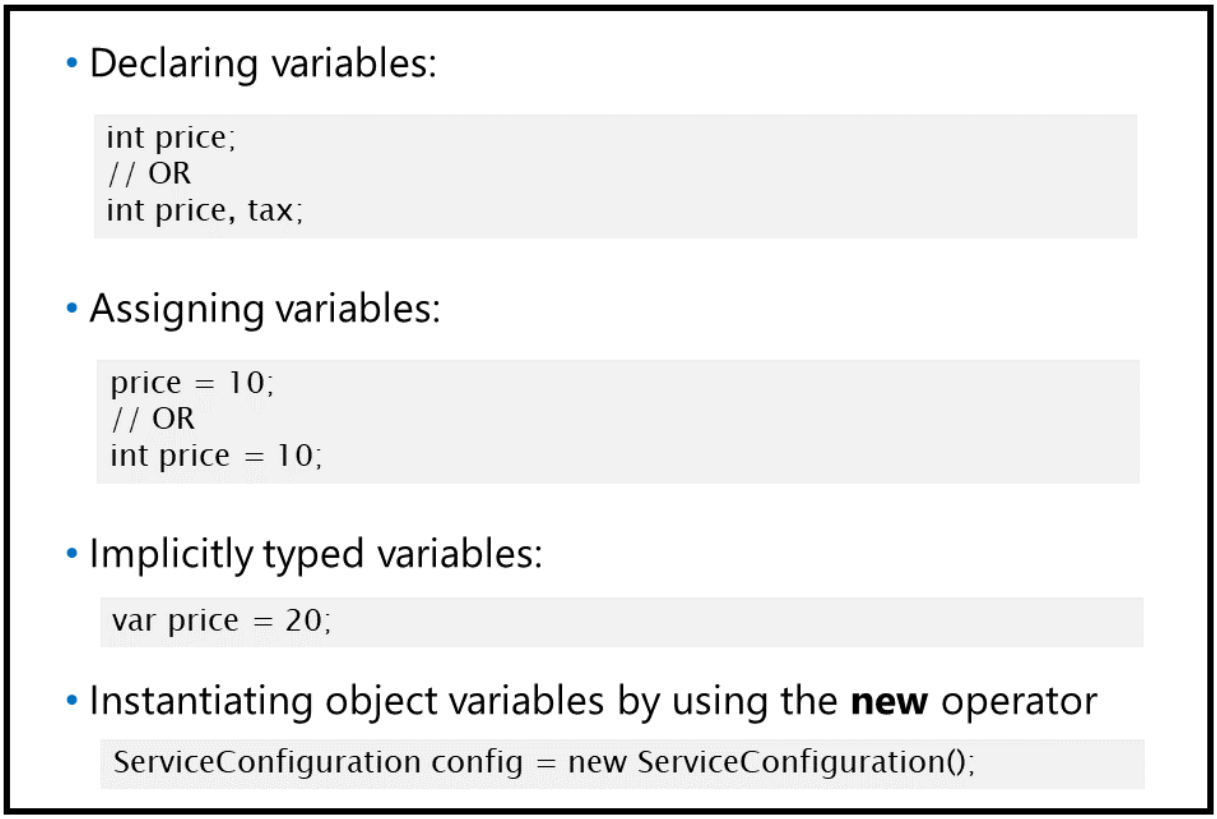
```
"ApplicationName: " + appName.ToString()
```

The **+** operator uses an operand that is a result of a method call, **ToString()**. The **ToString()** method converts the value of a variable into a string, whatever type it is.

The .NET Framework class library contains many additional methods that you can use to perform mathematical and string operations on data, such as the **System.Math** class.

> **Additional Reading:** For more information about operators, see the **C# Operators** page at **https://aka.ms/moc-20483c-m1-pg2**.

Declaring and Assigning Variables



Before you can use a variable, you must declare it so that you can specify its name and characteristics. The name of a variable is referred to as an identifier. Visual C# has specific rules concerning the identifiers that you can use:

- An identifier can only contain letters, digits, and underscore characters.
- An identifier must start with a letter or an underscore.
- An identifier for a variable should not be one of the keywords that Visual C# reserves for its own use.

Visual C# is case sensitive. If you use the name **MyData** as the identifier of a variable, this is not the same as **myData**. You can declare two variables at the same time called **MyData** and **myData** and Visual C# will not confuse them, although this is not good coding practice.

When declaring variables you should use meaningful names for your variables, because this can make your code easier to understand. You should also adopt a naming convention and use it!

Declaring and Assigning Variable

When you declare a variable, you reserve some storage space for that variable in memory and the type of data that it will hold. You can declare multiple variables in a single declaration by using the comma separator; all variables declared in this way have the same type.

The following example shows how to declare a new variable.

Declaring a Variable

```
// DataType variableName;
int price;
// OR
// DataType variableName1, variableName2;
int price, tax;
```

After you declare a variable, you can assign a value to it by using an assignment statement. You can change the value in a variable as many times as you want during the running of the application. The assignment operator **=** assigns a value to a variable.

The following code example shows how to use the **=** operator to assign a value to a variable.

Assigning a Variable

```
// variableName = value;
price = 10;
```

The value on the right side of the expression is assigned to the variable on the left side of the expression.

You can declare a variable and assign a value to it at the same time.

The following code example declares an **int** named **price** and assigns the value **10**.

Declaring and Assigning Variables

```
int price = 10;
```

When you declare a variable, it contains a random value until you assign a value to it. This behavior was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value. Visual C# does not allow you to use an unassigned variable. You must assign a value to a variable before you can use it; otherwise, your program might not compile.

Implicitly Typed Variables

When you declare variables, you can also use the **var** keyword instead of specifying an explicit data type such as **int** or **string**. When the compiler sees the **var** keyword, it uses the value that is assigned to the variable to determine the type.

In the following example shows how to use the **var** keyword to declare a variable.

Declaring a Variable by Using the var Keyword

```
var price = 20;
```

In this example, the **price** variable is an implicitly typed variable. However, the **var** keyword does not mean that you can later assign a value of a different type to **price**. The type of **price** is fixed, in much the same way as if you had explicitly declared it to be an **integer** variable.

Implicitly typed variables are useful when you do not know, or it is difficult to establish explicitly, the type of an expression that you want to assign to a variable.

Object Variables

When you declare an object variable, it is initially unassigned. To use an object variable, you must create an instance of the corresponding class, by using the **new** operator, and assign it to the object variable.

The **new** operator does two things: it causes the CLR to allocate memory for your object, and it then invokes a constructor to initialize the fields in that object. The version of the constructor that runs depends on the parameters that you specify for the **new** operator.

The following code example shows how to create an instance of a class by using the **new** operator.

The new Operator

```
ServiceConfiguration config = new ServiceConfiguration();
```

Additional Reading: For more information about declaring and assigning variables, see the **Implicitly Typed Local Variables (C# Programming Guide)** page at **http://go.microsoft.com/fwlink/?LinkID=267772**.

Accessing Type Members

- Invoke instance members

```
<instanceName>.<memberName>
```

- Example:

```
var config = new ServiceConfiguration();

// Invoke the LoadConfiguration method.
config.LoadConfiguration();

// Get the value from the ApplicationName property.
var applicationName = config.ApplicationName;

// Set the .DatabaseServerName property.
config.DatabaseServerName = "78.45.81.23";

// Invoke the SaveConfiguration method.
config.SaveConfiguration();
```

To access a member of an instance of a type, use the name of the instance, followed by a period, followed by the name of the member. This is known as dot notation. Consider the following rules and guidelines when you access a member of an instance:

- To access a method, use parentheses after the name of the method. In the parentheses, pass the values for any parameters that the method requires. If the method does not take any parameters, the parentheses are still required.

- To access a public property, use the property name. You can then get the value of that property or set the value of that property.

The following code example shows how to invoke the members that the **ServiceConfiguration** class exposes.

Invoking Members

```
var config = new ServiceConfiguration();
// Invoke the LoadConfiguration method.
var loadSuccessful = config.LoadConfiguration();
// Get the value from the ApplicationName property.
var applicationName = config.ApplicationName;
// Set the .DatabaseServerName property.
config.DatabaseServerName = "78.45.81.23";
// Invoke the SaveConfiguration method.
var saveSuccessful = config.SaveConfiguration();
```

Additional Reading: For more information about using properties, see the **Properties (C# Programming Guide)** page at **http://go.microsoft.com/fwlink/?LinkID=267773**.

Additional Reading: For more information about using methods, see the **Methods (C# Programming Guide)** page at **http://go.microsoft.com/fwlink/?LinkID=267774**.

Casting Between Data Types

- Implicit conversion:

```
int a = 4;
long b = 5;
b = a;
```

- Explicit conversion:

```
int a = (int) b;
```

- **System.Convert** conversion:

```
string possibleInt = "1234";
int count = Convert.ToInt32(possibleInt);
```

When you are developing an application, you will often need to convert data from one type to another type, for example, when a value of one type is assigned to a variable of a different type. Consider the scenario where a user enters a number into a text box. To use this number in a numerical calculation, you will need to convert the string value 99 that you have read from the text box into the integer value 99 so that you can store it in an integer variable. The process of converting a value of one data type to another type is called type conversion or casting.

There are two types of conversions in the .NET Framework:

- Implicit conversion, which is automatically performed by the CLR on operations that are guaranteed to succeed without losing information.
- Explicit conversion, which requires you to write code to perform a conversion that otherwise could lose information or produce an error.

Explicit conversion reduces the possibility of bugs in your code and makes your code more efficient. Visual C# prohibits implicit conversions that lose precision. However, be aware that some explicit conversions can yield unexpected results.

Implicit Conversions

An implicit conversion occurs when a value is converted automatically from one data type to another. The conversion does not require any special syntax in the source code. Visual C# only allows safe implicit conversions, such as the widening of an integer.

The following code example shows how data is converted implicitly from an integer to a long, which is termed *widening*.

Implicit Conversion

```
int a = 4;
long b;
b = a;        // Implicit conversion of int to long.
```

This conversion always succeeds and never results in a loss of information. However, you cannot implicitly convert a **long** value to an **int**, because this conversion risks losing information (the **long** value might be outside the range supported by the **int** type). The following table shows the implicit type conversions that are supported in Visual C#.

From	To
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Explicit Conversions

In Visual C#, you can use a cast operator to perform explicit conversions. A cast specifies the type to convert to, in round brackets before the variable name.

The following code example shows how to perform an explicit conversion.

Explicit Conversion

```
int a;
long b = 5;
a = (int) b;    // Explicit conversion of long to int.
```

You can only perform meaningful conversions in this way, such as converting a **long** to an **int**. You cannot use a cast if the format of the data has to physically change, such as if you are converting a **string** to an **integer**. To perform these types of conversions, you can use the methods of the **System.Convert** class.

Using the System.Convert Class

The **System.Convert** class provides methods that can convert a base data type to another base data type. These methods have names such as **ToDouble**, **ToInt32**, **ToString**, and so on. All languages that target the CLR can use this class. You might find this class easier to use for conversions than implicit or explicit conversions because IntelliSense helps you to locate the conversion method that you need.

The following code example converts a string to an int.

Conversions by Using the ToInt32 Method

```
string possibleInt = "1234";
int count = Convert.ToInt32(possibleInt);
```

Some of the built-in data types in Visual C# also provide a **TryParse** method, which enables you to determine whether the conversion will succeed before you perform the conversion.

The following code example shows how to convert a **string** to an **int** by using the **int.TryParse()** method.

TryParse Conversion

```
int number = 0;
string numberString = "1234";
if (int.TryParse(numberString, out number))
{
    // Conversion succeeded, number now equals 1234.
```

```
}
else
{
   // Conversion failed, number now equals 0.
}
```

Additional Reading: For more information about casting variables, see the **Casting and Type Conversions (C# Programming Guide)** page at **http://go.microsoft.com/fwlink/?LinkID=267775**.

Manipulating Strings

- Concatenating strings

```
StringBuilder address = new StringBuilder();
address.Append("23");
address.Append(", Main Street");
address.Append(", Buffalo");
string concatenatedAddress = address.ToString();
```

- Validating strings

```
var textToTest = "hell0 w0rld";
var regularExpression = "\\d";

var result = Regex.IsMatch(textToTest, regularExpression,
RegexOptions.None);

if (result)
{
   // Text matched expression.
}
```

Strings are a very useful data type that enable you to capture and store alphanumeric data.

Concatenating Strings

Concatenating multiple strings in Visual C# is simple to achieve by using the + operator. However, this is considered bad coding practice because strings are

immutable. This means that every time you concatenate a string, you create a new string in memory and the old string is discarded.

> **Note:** In Visual C#, data types are either mutable or immutable. This refers to their ability to change their intrinsic values. Immutable data types cannot be changed, and any change made to them results in the creation of a new copy of the modified value alongside the old value. Mutable data types can be changed and are not copied when changed. Therefore, when you concatenate a string, a new value with the complete string is created in memory, alongside the two original strings.

The following code example creates five string values as it runs.

Concatenation by Using the + Operator

```
string address = "23";
address = address + ", Main Street";
address = address + ", Buffalo";
```

An alternative approach is to use the **StringBuilder** class, which enables you to build a string dynamically and much more efficiently.

The following code example shows how to use the **StringBuilder** class.

Concatenation by Using the StringBuilder Class

```
StringBuilder address = new StringBuilder();
address.Append("23");
address.Append(", Main Street");
address.Append(", Buffalo");
string concatenatedAddress = address.ToString();
```

Validating Strings

When acquiring input from the user interface of an application, data is often provided as strings that you need to validate and then convert into a format that your application logic expects. For example, a text box control in a WPF application will return its contents as a **string**, even if a user specified an **integer** value. It is important that you validate such input so that you minimize the risk of errors, such as **InvalidCastExceptions**.

Regular expressions provide a mechanism that enables you to validate input. The .NET Framework provides the **System.Text.RegularExpressions** namespace that includes the **Regex** class. You can use the **Regex** class in your applications to test a **string** to ensure that it conforms to the constraints of a regular expression.

The following code example shows how to use the **Regex.IsMatch** method to see if a **string** value contains any numerical digits.

Regex.IsMatch Method

```
var textToTest = "hell0 w0rld";
var regularExpression = "\\d";
var result = Regex.IsMatch(textToTest, regularExpression,
RegexOptions.None);
if (result)
{
   // Text matched expression.
}
```

Regular expressions provide a selection of expressions that you can use to match to a variety of data types. For example, the **\d** expression will match any numeric characters.

> **Additional Reading:** For more information about using regular expressions, see the **Regex Class** page at **https://aka.ms/moc-20483c-m1-pg3**.

Lesson 3: Visual C# Programming Language Constructs

When developing an application, you will often need to execute logic based on a condition, or to repeatedly execute a section of logic until a condition is met. You may also want to store a collection of related data in a single variable. Visual C# provides a number of constructs than enable you model complex behavior in your applications.

In this lesson, you will learn how to implement decision and iteration statements and how to store collections of related data. You will also learn how to structure the API of your application by using namespaces, and how to use some of the debugging features that Visual Studio provides.

Lesson objectives

After completing this lesson, you will be able to:

- Use conditional statements.
- Use iteration statements.
- Create and use arrays.
- Describe the purpose of namespaces.
- Use breakpoints in Visual Studio.

Implementing Conditional Logic

- **if** statements

```
if (response == "connection_failed") {. . .}
else if (response == "connection_error") {. . .}
else { }
```

- **select** statements

```
switch (response)
{
   case "connection_failed":
      . . .
      break;
   case "connection_success":
      . . .
      break;
   default:
      . . .
      break;
}
```

Application logic often needs to run different sections of code depending on the state of data in the application. For example, if a user requests to close a file, they may be asked whether they wish to save any changes. If they do, the application must execute code to save the file. If they don't, the application logic can simply close the file. Visual C# uses conditional statements to determine which code section to run.

The primary conditional statement in Visual C# is the **if** statement. There is also a **switch** statement that you can use for more complex decisions.

Conditional Statements

You use **if** statements to test the truth of a statement. If the statement is **true**, the block of code associated with the **if** statement is executed, if the statement is **false**, control passes over the block.

The following code shows how to use an **if** statement to determine if a **string** contains the value **connection_failed**.

if Statement

```
string response = "….";
if (response == "connection_failed")
{
    // Block of code to execute if the value of the response
variable is "connection_failed".
}
```

if statements can have associated **else** clauses. The **else** block executes when the **if** statement is **false**.

The following code example shows how to use an **if else** statement to execute code when a condition is **false**.

if else Statements

```
string response = "….";
if (response == "connection_failed")
{
    // Block of code executes if the value of the response variable
is "connection_failed".
}
else
{
    // Block of code executes if the value of the response variable
is not "connection_failed".
}
```

if statements can also have associated **else if** clauses. The clauses are tested in the order that they appear in the code after the **if** statement. If any of the clauses returns **true**, the block of code associated with that statement is executed and control leaves the block of code associated with the entire **if** construct.

The following code example shows how to use an **if** statement with an **else if** clause.

else if Statements

```
string response = "….";
if (response == "connection_failed")
{
    // Block of code executes if the value of the response variable
is "connection_failed".
}
else if (response == "connection_error")
{
    // Block of code executes if the value of the response variable
is "connection_error".
}
else
{
    // Block of code executes if the value of the response variable
is not "connection_failed" or "connection_error".
}
```


Selection Statements

If there are too many **if**/**else** statements, code can become messy and difficult to follow. In this scenario, a better solution is to use a **switch** statement. The **switch** statement simply replaces multiple **if**/**else** statements.

The following sample shows how you can use a **switch** statement to replace a collection of **elseif** clauses.

switch Statement

```
string response = "….";
switch (response)
{
   case "connection_failed":
```

```
        // Block of code executes if the value of response is
"connection_failed".
        break;
    case "connection_success":
        // Block of code executes if the value of response is
"connection_success".
        break;
    case "connection_error":
        // Block of code executes if the value of response is
"connection_error".
        break;
    default:
        // Block executes if none of the above conditions are met.
        break;
}
```

In each **case** statement, notice the **break** keyword. This causes control to jump to the end of the switch after processing the block of code. A **switch case** must end with a jump statement, such as **break, return** or **goto**. If you omit the jump statement, your code will not compile.

Notice that there is a block labeled **default:**. This block of code will run when none of the other blocks match.

> **Additional Reading:** For more information about selection statements, see the **Selection Statements (C# Reference)** page at **https://aka.ms/moc-20483c-m1-pg4**.

Implementing Iteration Logic

- **for** loop

```
for (int i = 0 ; i < 10; i++) { ... }
```

- **foreach** loop

```
string[] names = new string[10];
foreach (string name in names) { ... }
```

- **while** loop

```
bool dataToEnter = CheckIfUserWantsToEnterData();
while (dataToEnter)
{
    ...
    dataToEnter = CheckIfUserHasMoreData();
}
```

- **do** loop

```
do
{
    ...
    moreDataToEnter = CheckIfUserHasMoreData();
} while (moreDataToEnter);
```

Iteration provides a convenient way to execute a block of code multiple times. For example, iterating over a collection of items in an array or just executing a function multiple times. Visual C# provides a number of standard constructs known as loops that you can use to implement iteration logic.

For Loops

The **for** loop executes a block of code repeatedly until the specified expression evaluates to false. You can define a **for** loop as follows.

for ([initializers]; [expression]; [iterators])

{

[body]

}

When using a **for** loop, you first initialize a value as a counter. On each loop, you check that the value of the counter is within the range to execute the **for** loop, and if so, execute the body of the loop.

The following code example shows how to use a **for** loop to execute a code block 10 times.

for Loop

```
for (int i = 0 ; i < 10; i++)
{
    // Code to execute.
}
```

In this example, **i = 0;** is the initializer, **i < 10;** is the expression, and **i++;** is the iterator.

For Each Loops

While a **for** loop is easy to use, it can be tricky to get right. For example, when iterating over a collection or an array, you have to know how many elements the collection or array contains. In many cases this is straightforward, but sometimes it can be easy to get wrong. Therefore, it is sometimes better to use a **foreach** loop.

The following code example shows how to use a **foreach** loop to iterate a **string** array.

foreach Loop

```
string[] names = new string[10];
// Process each name in the array.
foreach (string name in names)
{
    // Code to execute.
}
```

While Loops

A **while** loop enables you to execute a block of code while a given condition is **true**. For example, you can use a **while** loop to process user input until the user indicates that they have no more data to enter.

The following code example shows how to use a **while** loop.

while Loop

```
bool dataToEnter = CheckIfUserWantsToEnterData();
while (dataToEnter)
{
    // Process the data.
    dataToEnter = CheckIfUserHasMoreData();
}
```

Do Loops

A **do** loop is very similar to a **while** loop, with the exception that a **do** loop will always execute at least once. Whereas if the condition is not initially met, a **while** loop will never execute. For example, you can use a **do** loop if you know that this code will only execute in response to a user request to enter data. In this scenario, you know that the application will need to process at least one piece of data, and can therefore use a **do** loop.

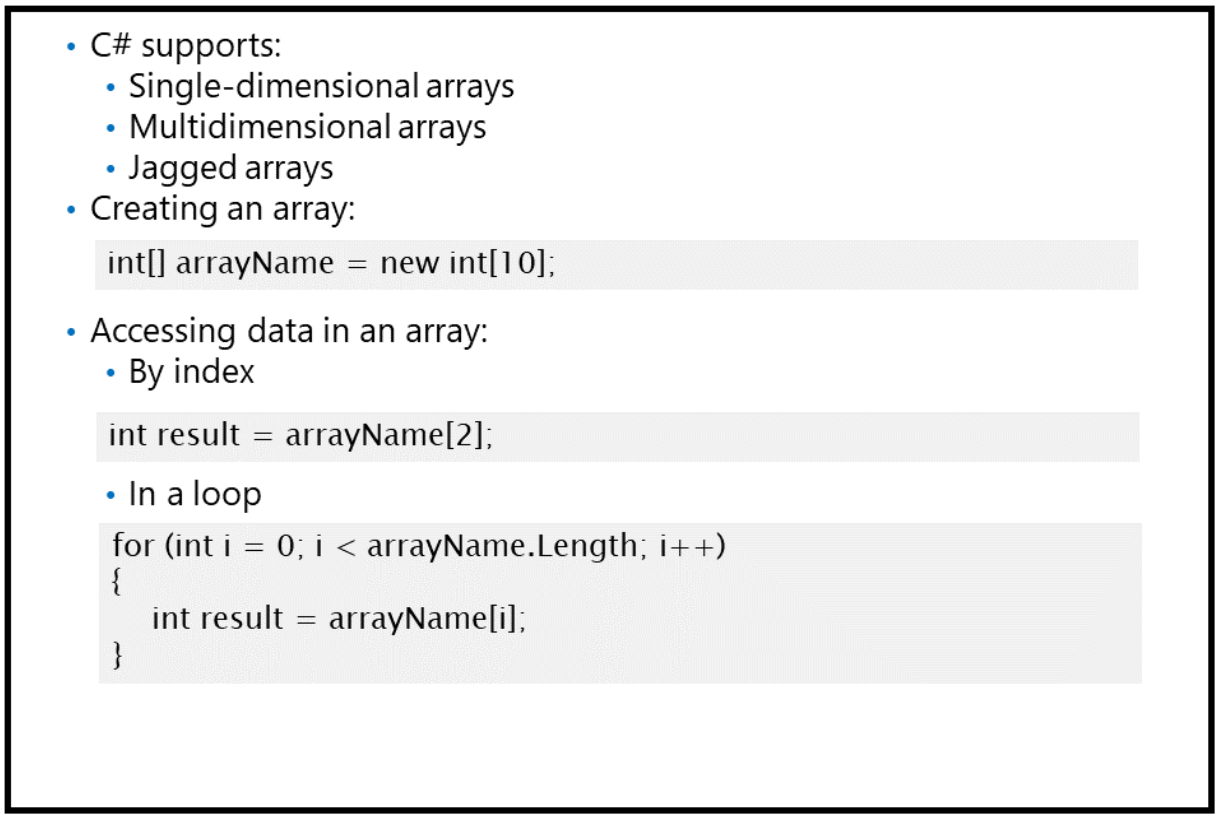
The following code example shows how to use a **do** loop.

do Loop

```
do
{
    // Process the data.
    moreDataToEnter = CheckIfUserHasMoreData();
} while (moreDataToEnter);
```

Additional Reading: For more information about loops, see the **Iteration Statements (C# Reference)** page at **http://go.microsoft.com/fwlink/?LinkID=267778**.

Creating and Using Arrays



An array is a set of objects that are grouped together and managed as a unit. You can think of an array as a sequence of elements, all of which are the same type. You can build simple arrays that have one dimension (a list), two dimensions (a table), three dimensions (a cube), and so on. Arrays in Visual C# have the following features:

- Every element in the array contains a value.
- Arrays are zero-indexed, that is, the first item in the array is element 0.
- The size of an array is the total number of elements that it can contain.
- Arrays can be single-dimensional, multidimensional, or jagged.
- The rank of an array is the number of dimensions in the array.

Arrays of a particular type can only hold elements of that type. If you need to manipulate a set of unlike objects or value types, consider using one of the collection types that are defined in the **System.Collections** namespace.

Creating Arrays

When you declare an array, you specify the type of data that it contains and a name for the array. Declaring an array brings the array into scope, but does not actually allocate any memory for it. The CLR physically creates the array when you use the **new** keyword. At this point, you should specify the size of the array.

The following list describes how to create single-dimensional, multidimensional, and jagged arrays:

- Single-dimensional arrays. To declare a single-dimensional array, you specify the type of elements in the array and use brackets, [] to indicate that a variable is an array. Later, you specify the size of the array when you allocate memory for the array by using the **new** keyword. The size of an array can be any integer expression. The following code example shows how to create a single-dimensional array of integers with elements zero through nine.

```
int[] arrayName = new int[10];
```

- Multidimensional arrays. An array can have more than one dimension. The number of dimensions corresponds to the number of indexes that are used to identify an individual element in the array. You can specify up to 32 dimensions, but you will rarely need more than three. You declare a multidimensional array variable just as you declare a single-dimensional array, but you separate the dimensions by using commas. The following code example shows how to create an array of integers with three dimensions.

```
int[ , , ] arrayName = new int[10,10,10];
```

- Jagged arrays. A jagged array is simply an array of arrays, and the size of each

array can vary. Jagged arrays are useful for modeling sparse data structures where you might not always want to allocate memory for every item if it is not going to be used. The following code example shows how to declare and initialize a jagged array. Note that you must specify the size of the first array, but you must not specify the size of the arrays that are contained within this array. You allocate memory to each array within a jagged array separately, by using the **new** keyword.


```
int[][] jaggedArray = new int[10][];
jaggedArray[0] = new Type[5]; // Can specify different sizes.
jaggedArray[1] = new Type[7];
...
jaggedArray[9] = new Type[21];
```

Accessing Data in an Array

You can access data in an array in several ways, such as by specifying the index of a specific element that you require or by iterating through the entire collection and returning each element in sequence.

The following code example uses an index to access the element at index two.

Accessing Data by Index

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int number = oldNumbers[2];
```

> **Note:** Arrays are zero-indexed, so the first element in any dimension in an array is at index zero. The last element in a dimension is at index N-1, where N is the size of the dimension. If you attempt to access an element outside this range, the CLR throws an **IndexOutOfRangeException** exception.

You can iterate through an array by using a **for** loop. You can use the **Length** property of the array to determine when to stop the loop**.**

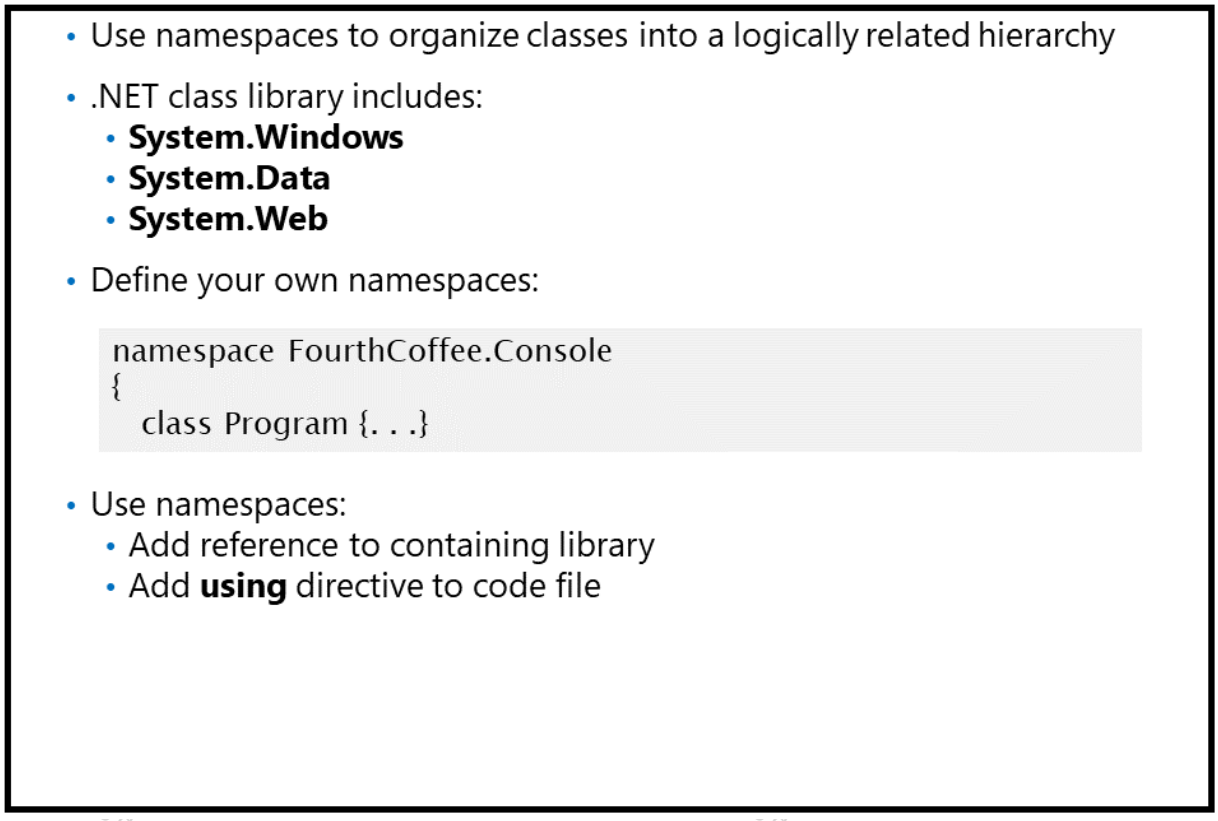
The following code example shows how to use a **for** loop to iterate through an array.

Iterating Over an Array

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
for (int i = 0; i < oldNumbers.Length; i++)
{
    int number = oldNumbers[i];
    ...
}
```

Additional Reading: For more information about arrays, see the **Arrays (C# Programming Guide)** page at **http://go.microsoft.com/fwlink/?LinkID=267779**.

Referencing Namespaces



The Microsoft .NET Framework consists of many namespaces that organize its classes into logically related hierarchies. You can use namespaces in your own applications to similarly organize your classes into hierarchies.

Namespaces function as both an internal system for organizing your application and as an external way to avoid name clashes between your code and other applications. Each namespace contains types that you can use in your program, such as classes, structures, enumerations, delegates, and interfaces. Because different classes can have the same name, you use namespaces to differentiate the same named class into two different hierarchies to avoid interoperability issues.

.NET Framework Class Library Namespaces

The most important namespace in the .NET Framework is the **System** namespace, which contains the classes that most applications use to interact with the operating system. A few of the namespaces provided by the .NET Framework through the **System** namespace are listed in the following table:

Namespace	Definition
System.Windows	Provides the classes that are useful for building WPF applications.
System.IO	Provides classes for reading and writing data to files.
System.Data	Provides classes for data access.
System.Web	Provides classes that are useful for building web applications.

User-Defined Namespaces

User-defined namespaces are namespaces defined in your code. It is good practice to define all your classes in namespaces. The Visual Studio environment follows this recommendation by using the name of your project as the top-level namespace in a project.

The following code example shows how to define a namespace with the name **FourthCoffee.Console**, which contains the **Program** class.

Defining a Namespace

```
namespace FourthCoffee.Console
{
   class Program
   {
       static void Main(string[] args)
       {
       }
   }
}
```

Using Namespaces

When you create a Visual C# project in Visual Studio, the most common base class assemblies are already referenced. However, if you need to use a type that is in an assembly that is not already referenced in your project, you will need to add a reference to the assembly by using the **Add Reference** dialog box. Then at the top of your code file, you list the namespaces that you use in that file, prefixed with the **using** directive. The **using** directive instructs your application to import the types in the namespace to allow them to be called directly, the same as if they existed in the current namespace.

The following code example shows how to import the **System** namespace and use the **Console** class.

Importing a Namespace

```
using System;
…
Console.WriteLine("Hello, World");
```

You can call any type in a referenced assembly provided you specify that type's entire name, including its namespace. This is referred to as the type's fully qualified name.

The following code example shows how to use the **Console** class without importing the **system** namespace.

Calling Console by its fully qualified name

```
System.Console.WriteLine("Hello, World");
```

> **Additional Reading:** For more information about namespaces, see the **namespace (C# Reference)** page at **https://aka.ms/moc-20483c-m1-pg2**.

Using Breakpoints in Visual Studio 2017

- Breakpoints enable you to view and modify the contents of variables:
  - Immediate Window
  - Autos, Locals, and Watch panes
- Debug menu and toolbar functions enable you to:
  - Start and stop debugging
  - Enter break mode
  - Restart the application
  - Step through code

Debugging is an essential part of application development. You may notice errors as you write code, but some errors, especially logic errors, may only occur in circumstances that you do not predict. Users may report these errors to you and you will have to correct them.

Visual Studio 2017 provides several tools to help you debug code. You might use these while you develop code, during a test phase, or after the application has been

released. You will use the tools in the same way regardless of the circumstances. You can run an application with or without debugging enabled. When debugging is enabled, your application is said to be in **Debug** mode.

Using Breakpoints

If you know the approximate location of the issue in your code, you can use a breakpoint to make the Visual Studio debugger enter break mode before executing a specific line of code. This enables you to use the debugging tools to review or modify the status of your application to help you rectify the bug. To add a breakpoint to a line of code, on the **Debug** menu, click **Toggle Breakpoint**.

When you are in break mode, you can hover over variable names to view their current value. You can also use the **Immediate Window** and the **Autos**, **Locals**, and **Watch** panes to view and modify the contents of variables.

Using Debug Controls

After viewing or modifying variables in break mode, you will likely want to move through the subsequent lines of code in your application. You might want to simply run the remainder of the application or you might want to run one line of code at a time. Visual Studio provides a variety of commands on the **Debug** menu that enable you to do this and more. The following table lists the key items on the **Debug** menu and the **Debug** toolbar, and the corresponding keyboard shortcuts for navigating through your code.

Menu item	Toolbar button	Keyboard shortcut	Description
Start Debugging	Start/continue	F5	This button is available when your application is not running and when you are in break mode. It will start your application in Debug mode or resume the application if you are in break mode.

Menu item	Toolbar button	Keyboard shortcut	Description
Break All	Break all	Ctrl+Alt+Break	This button causes application processing to pause and break mode to be entered. The button is available when an application is running.
Stop Debugging	Stop	Shift+F5	This button stops debugging. It is available when an application is running or is in break mode.
Restart	Restart	Ctrl+Shift+F5	This button is equivalent to stop followed by start. It will cause your application to be restarted from the beginning. It is available when an application is running or is in break mode.
Step Into	Step into	F11	This button is used for stepping into method calls.
Step Over	Step over	F10	This button is used for stepping over method calls.
Step Out	Step out	Shift+F11	This button is used for executing the remaining code in the method and returning to the next statement in the calling method.

Additional Reading: For more information about debugging, see the **Debugging in Visual Studio** page at **https://aka.ms/moc-20483c-m1-pg6**.

Demonstration: Developing the Class Enrollment Application Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration steps

You will find the steps in the **Demonstration: Developing the Class Enrollment Application Lab** section on the following page:

https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_DEMO.md.

Lab: Developing the Class Enrollment Application

Scenario

You are a Visual C# developer working for a software development company that is writing applications for The School of Fine Arts, an elementary school for gifted children.

The school administrators require an application that they can use to enroll students in a class. The application must enable an administrator to add and remove students from classes, as well as to update the details of students.

You have been asked to write the code that implements the business logic for the application.

> **Note:** During the labs for the first two modules in this course, you will write code for this class enrollment application.
> When The School of Fine Arts ask you to extend the application functionality, you realize that you will need to test proof of concept and obtain client feedback before writing the final application, so in the lab for Module 3, you will begin developing a prototype application and continue with this until then end of Module 8.
> In the lab for Module 9, after gaining signoff for the final application, you will develop the user interface for the production version of the application, which you will work on for the remainder of the course.

Objectives

After completing this lab, you will be able to:

- Write Visual C# code that implements the logic necessary to edit the details of a student.
- Write Visual C# code that implements the logic necessary to add new students.

- Write Visual C# code that implements the logic necessary to remove students from a class.
- Perform simple data transformations for displaying information.

Lab setup

Estimated Time: 105 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_LAB_MANUAL.md**.

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_LAK.md**.

Exercise 1: Implementing Edit Functionality for the Students List

Scenario

In this exercise, you will write the code that enables an administrator using the application to edit a student's details.

A list of students is displayed in the user interface of the application. When the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Enter. If they did press Enter, you will write code to display the student's details in a separate form, which the user can use to modify the details. When the user closes the form, you will copy the updated details back to the list box displaying the list of students. Finally, you will run the application to verify that your code functions as expected, and then use the debugging tools to examine code as it runs.

> **Result**: After completing this exercise, users will be able to edit the details of a student.

Exercise 2: Implementing Insert Functionality for the Students List

Scenario

In this exercise, you will write code that enables an administrator using the application to add a new student to the students list.

A list of students is displayed in the user interface of the application. When the user presses a key on the keyboard, you will check whether the key they pressed was Insert. If they did press Insert, you will write code to display a form in which the user can enter the details of a new student, including their first name, last name, and date of birth. When the user closes the form, you will add the new student to the list of students and display the details in the list box. Finally, you will run the application to verify that your code functions as expected.

Result: After completing this exercise, users will be able to add new students to a class.

Exercise 3: Implementing Delete Functionality for the Students List

Scenario

In this exercise, you will write code that enables an administrator to remove a student from the students list.

A list of students is displayed in the user interface of the application. If the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Delete. If they did press Delete, you will write code to prompt the user to confirm that they want to remove the selected student from the class. If they do, the student will be deleted from the students list for the appropriate class, otherwise nothing changes. Finally, you will run the application to verify that your code functions as expected.

Result: After completing this exercise, users will be able to remove students from classes.

Exercise 4: Displaying a Student's Age

Scenario

In this exercise, you will update the application to display a student's age instead of their date of birth.

You will write code in the **AgeConverter** class that is linked to the grid column displaying student ages. In this class, you will write code to work out the difference between the current date and the date of birth of the student, and then convert this value into years. Then you will run the application to verify that the Age column now displays age in years instead of the date of birth.

> **Result**: After completing this exercise, the application will display a student's age in years.

Module review and takeaways

In this module, you learned about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You also learned about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

Review Question(s)

Check Your Knowledge

Select the best answer

What Visual Studio template would you use to create a .dll?

- Console application
- Windows Forms application
- WPF application
- Class library

WCF Service application

Check answer Show solution Reset

Check Your Knowledge

Select all that apply

Given the following for loop statement, what is the value of the count variable once the loop has finished executing?

```
var count = 0;


for (int i = 5; i < 12; i++)


{

 count++;


}
```

3

5

7

9

11

Check answer Show solution Reset