

Module 1 – C# Syntax

Module Overview

Microsoft's .NET framework is a complete development platform that allows you to build applications and services. It allows you to build native applications, web applications and services, and much more. You can do this across a range of technologies to support enterprise scale software development. As a platform, .NET has evolved from a proprietary solution for running applications on the Windows operating system, to a cross-platform open source framework, but with enterprise-grade tooling and support if needed.

In this first module we're going to look at some of the core features of .NET and C#. This should be largely a review of your existing knowledge, or revision of what you learned in previous courses.

Objectives

After completing this module, you'll be able to:

- Give an overview of .NET and C#.
- Understand how to use the Visual Studio development environment.
- Use C# types, operators and expressions.
- Understand C# language constructs.

Lesson 1 – Writing Applications In C# and .NET

In this lesson we're going to look at the features of Visual Studio And review how .NET can enable us to build applications.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the role of the .NET framework.
- Identify some important features of Visual Studio.

- Enumerate some of the .NET project templates.
- Create a starter .NET framework app.

What Is the .NET Framework?

What is .NET?

- CLR
 - Robust and secure environment for your managed code
 - Memory management
 - Multithreading
- Class library
 - Foundation of common functionality
 - Extensible
- Development frameworks
 - WPF
 - Universal Windows Platform
 - ASP.NET

.NET gives us a complete development platform for building both apps and services. But why would you use .NET rather than some other technology? .NET gives us a very fast and efficient way to build solutions, and by saving time, it saves costs. Part of this time saving is because of the powerful range of development tooling available for .NET, including the full integrated development environment (IDE) of Visual Studio, light-weight code editors like Visual Studio Code, and a powerful set of command line utilities.

To enable us to run applications, .NET gives us a runtime environment called the common language runtime (CLR). It's the job of the Clr to manage code execution and provide a secure execution environment that includes memory management, transactions and multithreading. The platform also comes with a powerful class library Which enables us to use a vast range of functionality without having to write our own code. This greatly simplifies the development process and eliminates a huge amount of reinvention of algorithms that would otherwise be necessary. These include powerful collection classes, file handling, network communications, and database access.

In addition, there are a large number of development frameworks for particular application types, such as user interface libraries for desktop client applications, server-side web applications, web service applications, client side-web development and cross-platform solutions. In each case there are starter solutions that can help you quickly get up to speed.

Read more: For more on the .NET framework see <https://aka.ms/39GYhwX>.

Key Features of Visual Studio

Key Features of Visual Studio

- Intuitive IDE
- Rapid application development
- Server and data access
- IIS Express
- Debugging features
- Error handling
- Help and documentation

The Visual Studio suite of products provides a single integrated development environment that allows you to very quickly design, build, deploy and test a variety of different types of application and components using multiple programming languages (although in this course we will concentrate on C#). The IDE of Visual Studio gives us a range of features and tools that combine to provide a rapid application development environment. these include powerful graphical design services for building complex user interfaces, code editors, and numerous Wizards that help speed up component development.

When it comes to working with servers, and getting access to data, Visual Studio provides a component called server explorer this enables you to log on to remote servers and databases and interactively explore their services and databases full stop you can also use it to create new databases or modify existing ones using a range of database design tools.

When doing web development, Visual Studio comes with an integrated lightweight version of IIS (IIS Express), Which serves as the default web server technology for testing and debugging your web applications. When it's time to do debugging work, Visual Studio includes a debugger that supports stepping through both locally running and remote running code, setting breakpoints, and examining the values of variables.

Visual Studio also provides numerous output windows including the display of errors, warnings and other messages as you work on your code. and in addition to built-in help and documentation, Visual Studio has a powerful mechanism called Intellisense® that is able to predict code, provide integrated help and code snippets.

To find out what's new in the latest version of Visual Studio, see: <https://aka.gd/3Pv2a80>.

Key Features of Visual Studio Code

Key Features of Visual Studio Code

- Powerful code editor
- Lightweight – installs in minutes
- Can be used online
- Wide range of extensions
- Wide support for languages and platforms
- Debugging features
- Help and documentation

One of the criticisms of Visual Studio is that it's a very large traditional desktop application. This means it takes quite a long time to install (up two hours depending on what features you require) and has a large application footprint. It's also primarily a windows application (although there is a version of Visual Studio for Mac OS), and it is by new means a cross platform application.

Microsoft have produced another editor called Visual Studio Code (sometimes abbreviated to VS Code). This is a lightweight code editor built on web technology, and yet it is surprisingly powerful. It doesn't require anything like the investment of time and resource is to install compared to Visual Studio. It has become very popular in the web development community, and can also be used as an online application without requiring any install whatsoever.

Like Visual Studio, VS Code includes Intellisense, debugging tools, source code integration and other built-in features. In addition, its powerful extensibility has resulted in a rich aftermarket in extensions, allowing you to do many of the things that would otherwise require Visual Studio.

You can install Visual Studio Code by navigating to: <https://aka.gd/3NtONGO>.

Visual Studio Code is regularly updated. To see the latest features, go to:
<https://aka.ms/3wvLma5>.

Templates

Templates in Visual Studio

- Console Application
- WPF Application
- Universal Windows Platform (UWP)
- Class Library
- ASP.NET Web Application
- ASP.NET MVC Core Application

Visual Studio supports various different types of application including Windows desktop applications, web applications, web services, and development libraries. There are templates for all these different types of application built into Visual Studio, and invoking one of these using the file new project menu will scaffold out a solution for each of these different types of applications as required. The scaffolding provides you with starter code ready to build an application; the starter code being a functioning application but with minimal useful functionality so that you can use it as a starting point. A lot of these types of application require supporting components and controls, and these get included in the starter project. In addition, Visual Studio will be preconfigured according to the type of application you are building, along with any references to assemblies that are prerequisite for that type of application.

Some of the different types of application are listed in the following table:

Template	Description
Console Application	An app designed to run at the command line interface, without any graphical user interface.
WPF Application	A template for building a Windows Presentation Foundation application, in which the graphical UI design is defined using an XML file format.

UWP Application	A Universal Windows Platform application template designed to run on different platforms using WinRT APIs for advanced asynchronous UI features.
Class Library	A project to build a reusable .NET assembly as a .dll file, that can be used within multiple applications.
Blazor App	An application that uses the Blazor framework to run .NET applications in the browser by using webassembly.
ASP.NET MVC Core Web App	A server-side web application for building web sites implemented using the latest version of ASP.NET built on top of the cross-platform version of .NET (formerly .NET Core).
ASP.NET MVC Core Web API	A server-side web application for building web service endpoints implemented using the latest version of ASP.NET built on top of the cross-platform version of .NET (formerly .NET Core).

Creating a .NET Application in Visual Studio

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) { }
    }
}

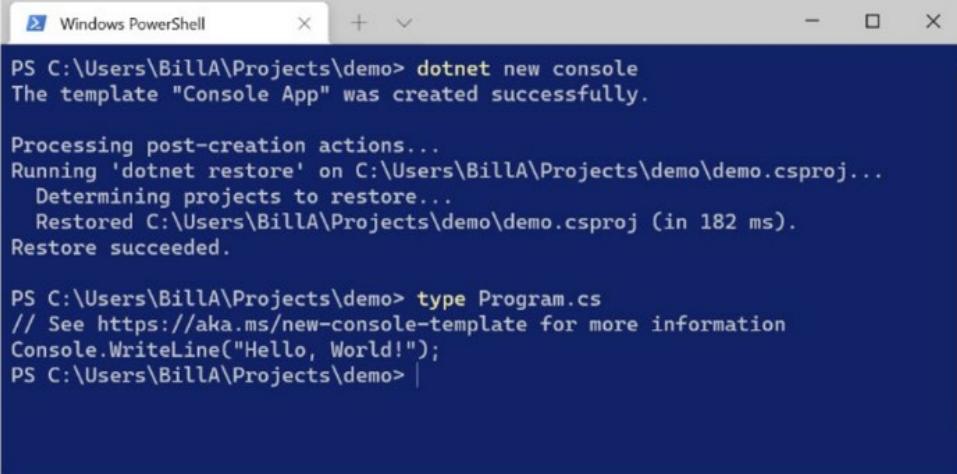
```

There are a couple of ways of ‘scaffolding’ a starter solution based on the .NET app templates. In Visual Studio you can use the File->New->Project and this presents you with a number of project templates, depending on what features are installed in Visual Studio. Let’s see how to build a console solution:

Notice that the code in Program.cs is very minimalistic. You may have seen template solutions in the past where there is more content in the initial template Program.cs file, including using statements, a namespace, a Program class and a static void Main function. Since .NET 6 these are no longer required to get started; they’re generated automatically by the compiler.

Creating a .NET Application from the Command Line

Creating a .NET Application from the Command Line



```
PS C:\Users\BillA\Projects\demo> dotnet new console
The template "Console App" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\BillA\Projects\demo\demo.csproj...
  Determining projects to restore...
    Restored C:\Users\BillA\Projects\demo\demo.csproj (in 182 ms).
Restore succeeded.

PS C:\Users\BillA\Projects\demo> type Program.cs
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
PS C:\Users\BillA\Projects\demo>
```

In addition to using Visual Studio, you can also use .NET from the command line. The **dotnet** command will scaffold out of project in the same way as Visual Studio. You can then use any code editor, including the one built into Visual Studio, to start building your application based on the starter template. For example to create a new console app we would use the following command:

```
dotnet new console
```

You can get a list of all the different templates that are available by typing the following at the command line:

```
dotnet new --list
```

If you want to see all the different options for the net command you can type the following:

```
dotnet --help
```

Overview of XAML

- XML-based language for declaring UIs
- Uses elements to define controls
- Uses attributes to define properties of controls

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

In the most general terms XAML (Extensible Application Markup Language) is an XML language for serialising .NET objects. The most common application of XAML is to define the user interface components which make up an application's UI. These components are created as instances of .NET UI classes as defined in the XAML document. It's also possible to instantiate these components directly from code.

Visual Studio includes designers that allow you to interactively create a user interface. XAML is a declarative language that you can build up using the designer, or if you prefer you can edit the XAML source code directly in a text editor.

As a markup language, XAML bears some similarities to HTML. Here is an example that creates a label, a textbox and a button:

```
<Label Content="First Name:"/>  
<TextBox Text="" Height="25" width="100" />  
<Button Content="Click Here!" width="80" />
```

With XAML you can create a simple user interface as in the previous example, or something much more complicated. The available controls include text and numeric inputs, media controls, images, buttons, and sliders. As well as providing controls, the markup includes ways to bind data to controls, to use textures and color gradients, to bind events to code, manage styles. There's also a variety of containers for the controls; e.g. grids, canvases and other layout controls. all these elements can be dragged onto the design surface from the Visual Studio toolbox.

Note: For more information about XAML, see Module 9 of this course.

Question: What markup language would you use to design a web page?

Answer: HTML5

Lesson 2 - Types of Data and Expressions

In this lesson, we'll focus on the core data types that are available in C#. We'll also look at the limitations on the various types of numeric, logical and string data types. We'll look at the differences between integers of various sizes, floating point numbers and decimal types.

Lesson Objectives

After completing this lesson, you'll be able to:

- Describe numeric data types.
- Describe character and string types.
- Describe variables and assignments

Numeric Data Types

Numeric Data Types

- Data Types
 - int
 - double
 - Byte
 - long

In the first module we talked about the binary (base 2) number system and how it differs from the decimal (base 10) numbers in everyday use. The numeric data types that you can use in a computer program are represented internally as binary numbers, but typically we work with decimal numbers, and these must be represented and operated on by the computer's arithmetic processing unit. Let's first look at the different categories of numbers from the perspective of a mathematician:

Category	Range
Natural Numbers	One to infinity. No decimal places and no negative values.
Whole Numbers	Zero to infinity. Same as the Natural Numbers, but includes zero as a valid number.
Integers	Negative infinity to positive infinity. Same as Whole Numbers, but including negative values.
Rational Numbers	All the values found in Integers plus fractional values (i.e. any number that can be expressed as a ratio of two integers) including decimal fractions.

These different number types need to be borne in mind when you want to decide on the number types and ranges to choose for a computer program. Computers have finite resources in terms of memory and the ability to work with numbers and characters. In early computing it was very important to conserve these resources and a lot of effort was put into using the minimum amount of storage.

An example of this was the representation of dates in computer applications from the last century, where the year would be represented as two digits. The year 1967 would be stored as the number 67, saving a byte (remember that a single byte can only store a number ranging from 0 to 255). But a problem arose in the year 2000, when the higher order digits of the year changed, and a lot of applications started to show erroneous dates. For example, an application might be computing a mortgage payment based on subtracting dates, but the current date was interpreted as 1900 instead of 2000, resulting in miscalculations or programs crashing with overflow errors.

Fortunately, memory is nowadays relatively inexpensive, and we don't usually have to worry about saving every last byte of memory. But nor do we want to waste memory unnecessarily. And there are still cases, for example programming microcontrollers, where memory is still a precious resource. So you still need to understand the maximum values you can assign to the data types available in modern programming languages.

Integer Data Types

Integer Data Types

Integer Type	Value Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Earlier we looked at natural numbers, whole numbers and integers. In computer programming we refer to all of these as integers. Many languages will use the keyword **int** to declare these, as an abbreviation for integer. The Microsoft .NET Framework also represents these values as **int**, but also allows you to specify how many bytes to assign to them, and whether they are signed (i.e. whether they are whole numbers or signed integers as described in the previous section).

If we declare an **int**, we will get a four-byte integer (32 bits) that can store a value from -2,147,483,648 to 2,147,483,647 inclusive (integers are signed unless we choose an unsigned integer by declaring a **uint**). On the other hand, if we choose to declare a two-byte

integer (**short**), it will hold 16-bits, and will have a range from -32,768 to 32,767. You might wonder what would happen if you had, say, a 32-bit integer variable with a value of 2,147,483,647 and added 1 to it. In this case it would follow the rules of fixed length binary addition and wrap around to the next number, which is -2,147,483,648. We can demonstrate this as follows:

1. Start Visual Studio 2022.
2. Create a new C# Console application.
3. Declare an **int** called i, and then assign it the value 2147483647 (note: no commas).
4. Add a line to write the value: `Console.WriteLine("i = " + i);`.
5. Using an assignment statement, increment the value of i, e.g. `i = i + 1;`.
6. Run the app and note the value changes from 2147483647 to -2147483648.

Resulting code in Program.cs:

```
int i = 2147483647;  
Console.WriteLine("i = " + i);  
i = i + 1;  
Console.WriteLine("i = " + i);
```

If we choose 32 bits to represent a value, it means that you can store a value only in 32 available places using either a 1 or 0. If it's a signed integer then we need the first bit to store the sign (0 = positive, 1 = negative) so we are down to 31 bits. And 2 raised to the 31st power is 2,147,483,648, but we don't get to use the last value so the highest number we can represent is $2,147,483,648 - 1 = 2,147,483,647$. You can check this using the Calculator application in Windows. To find the maximum value you can store in a given number of bits, raise 2 to the power of the number of bits, and then subtract one. You can also convert the number to binary, and you'll get, in the case of 32 bits, 1111111111111111111111111111, that's 31 1's. You can also express that more clearly by grouping the digits into 8-bit bytes:

```
01111111 11111111 11111111 11111111
```

Note that we had to add a leading 0 to the first byte. That's because the first bit is used by the sign bit. If you want to see that in hexadecimal using the calculator, you'll get 7FFFFFFF.

Sometimes the number you're representing is always positive, in other words it's a whole number. For example, the number of items in a shopping basket can never be negative, so a

signed integer isn't necessary. In those cases you can choose one of the unsigned integer types. That buys you an extra binary digit of precision, but also means that you don't have to worry about adding logic to make sure the value is always positive. In the case of unsigned integers, the effect of overflowing the maximum value is to wrap around to zero again.

In the following table you can see the different integer types and their corresponding value ranges. In practice, most programmers will use an int most of the time, unless there is a need for the additional numerical range of a long. The other types are used mainly in specialized applications, for example the byte type when doing binary shift operations in embedded devices.

Integer Type	Value Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

If you need even greater precision than the long integer type, you need to use a special large integer library with its own integer types. Libraries are a way of sharing re-usable code between programs. These integer libraries are particularly important in cryptography.

Rational Data Types

Rational Data Types

Type	Value Range	Size
decimal	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / (10^0 \text{ to } 10^{28})$	28-29 significant digits with fixed decimal places
double	$\pm 5.0 \times 10^{-324} \text{ to } \pm 1.7 \times 10^{308}$	Double precision floating point with 15 - 16 decimal digits of precision
float	$-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}$	Single precision floating point with 7 decimal digits of precision

Integers are great for counting things, but when it comes to measuring things like temperature and distance then it's a case of keeping it real. Real numbers contain fractional components, which in practical terms means decimal fractions. Fractions like $1/5$ are easily represented as 0.2. But if you want to represent a number like $1/3$ as a single number, then you have to accept that it will be an approximation in the decimal (base 10) number system. It will be 0.3333333, but with an infinitely recurring number of decimals. We can imagine an infinite number of decimals, but computers can't, so the value will be stored as a floating-point decimal number up to a certain precision.

Another example of such an approximation is the mathematical value of Pi, the ratio between the circumference of a circle and its diameter. Pi is not a rational number; it can't be expressed as a ratio. You might have learned at school that it is 3.14, or 3.1415926, or even $22/7$. These are all approximations, but Pi can be defined to an arbitrary number of decimal places. You could even write a program to compute it (using an infinite series).

These kinds of numbers are stored as floating-point values because the decimal point can be considered as floating; it can move around to change the value, or more concisely, the scale of the number. Internally, the computer stores the number as an integer fractional part (or mantissa) and a separate exponent that contains a multiplier that is a power of 10. This means that floating-point numbers retain the same degree of precision whether they are very small, e.g. 0.00000007475487, or very large, e.g. 747548700000000000.0. Numbers that lie within a certain range can be conveniently expressed in a floating-point format, e.g. 747.5487, or 0.07475487. But very large or very small numbers are usually expressed using scientific notation by normalizing the fractional part and adjusting the exponent, so instead of 747548700000000000.0, this number would be expressed as 7.475487e17, meaning 7.475487×10^{17} .

In C# there are two types of floating-point number. The single-precision number is called **float**. Because 7 or 8 digits of precision are insufficient for many scientific and engineering calculations, there is also a double-precision floating-point number called **double**. Most modern CPUs include a floating-point unit or FPU in addition to the regular integer CPU functions. The FPU can carry out multiplication, division, and exponentiation very efficiently.

For financial calculations, and a few other applications, a slightly different type of accuracy is required. It is vital that the number of pennies, or other currency subdivisions, is accurate.

This means that a fixed two decimal places is required, along with a large number of digits for the integer part. To support this, a special kind of integer with fixed decimals is provided called **decimal**.

The following table outlines the data types for numeric values that are available in C#, and several other languages.

Type	Value Range	Size
float	-3.4×10^{38} to $+3.4 \times 10^{38}$	Single precision floating point with 7 decimal digits of precision
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	Double precision floating point with 15-16 decimal digits of precision
decimal	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$	28-29 significant digits with fixed decimal places

The value ranges in the above table are a consequence of the 64-bit CPU computer architecture, on which most modern computers are based. As mentioned before, modern computers have large amounts of memory and disk storage, so program efficiency is much less of an issue than it was a couple of decades ago. But you should still make efficient choices when selecting your data types in the programs that you write. It will make your programs execute faster and consume less memory. For example, if you want to represent a value that will represent the ambient temperature, then clearly a single-precision floating-point number is sufficient. If, on the other hand, you are trying to build a computer simulation of the birth of the universe, even a double-precision number might fall short without careful programming.

Another consideration is that some compilers will ‘upgrade’ a byte, short, or int data type in code and convert it to a long data type. The designer of the compiler might decide to do this based on the target CPU. This is because some CPUs work more efficiently with data types that are a certain size, or to avoid the overhead of converting between data types.

Character Data Types

Character Data Types

- String Representations
 - String - “A string of characters”
 - Character - ‘A’
 - Unicode – ‘\U0041’

Numbers are not the only values that we need to represent in our applications. Character data types are used for text strings consisting of letters, numerals, punctuation, and other symbols. Programming languages provide data types for storing these character types, and collections of characters called character strings (or just strings).

Like most programming languages, C# uses two data types known as **char** and **string**. The **char** data type is the only true character value type or intrinsic type. It’s effectively two bytes that contain a Unicode character. In this sense the **char** data type is similar to a numeric data type and is stored in memory in the same way. The **char** data type contains one character, such as an alphabetical letter or a punctuation character.

The **char** type uses UTF-16 encoding.

The **char** data type can be assigned a character literal in a few different forms:

```
char ch1 = 'A'; // Character literal
char ch2 = '\x0041'; // Hexadecimal
char ch3 = '\u0041'; // Unicode character
```

Usually, we want more than a single character, so we use a **string** which is a series of characters, such as a sentence or a single word. Strings are considered to be an array of characters (we’ll talk about arrays later in the course). But a **string** is not stored in the same way as the other variables we’ve been looking at. It’s a little more complicated because it’s actually a reference type, even though it behaves like a value type (we’ll discuss reference and value types later as well). And because it’s an immutable object, every time we change a **string**, we need to make a new copy. The good news is that unless you’re doing a large amount of **string** manipulation, these subtleties aren’t very important, and **strings** mostly behave the way you would expect them to.

When numbers appear in a string, they're considered as character literals and not numeric values. This means that you cannot perform mathematical functions on numbers embedded in a string. You need to first convert the string to the numeric type you require:

```
string myNumberString = "123"; // String literal  
int number = int.Parse(myNumberString); // Parse string to integer  
float value = float.Parse(myNumberString); // Parse string to float
```

Strings in C# have a large number of built-in functions to help you extract individual elements of the string, find matches, count the number of characters or parse the string in various ways. These are all features of the string library that's built into the language. Here are a few examples of string literals in C#.

```
string s1 = "This is a string";  
string s2 = "This string uses numbers such as 123 and 3.4509 .";  
string s3 = "This string contains punctuation symbols such as $.";  
string s4 = "This string uses quotes that must be \"escaped\" otherwise it will end the string .";
```

You could store a single character as a string with length 1: e.g. "A" instead of 'A'. Because the char data type consumes significantly fewer resources in a program than the string data type, it's generally a good idea to use the char type for single character values.

Other Data Types

Other Data Types

AND	1	1	True
	0	1	False
	1	0	False
	0	0	False
OR	1	1	True
	0	1	True
	1	0	True
	0	0	False

```
enum DaysOfWeek {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

```
enum Months {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

Numbers and characters are the most common values that are represented by using data types in a program, but there are a few other built-in data types, such as the Boolean data type, and enumerations.

The term Boolean is derived from the name of the mathematician George Boole. He also gives his name to Boolean logic. There are two possible values for a Boolean: true and false. You can combine two Boolean values by means of various Boolean operators such as AND and OR. The tables in the slide are called “truth tables” and show the results of these operators.

The data type that is used to represent Boolean values in C# and other programming languages is known as **bool**. You can use the bool data type to store the results of comparisons in your programs. They are used in decision structures or anywhere you might need to compare the result of some operation. The logical AND and OR operators in C# use the symbols “&&” and “||” respectively. Here are some examples of Boolean assignments:

```
bool b1 = 2 > 1; // True  
bool b2 = 2 < 1; // False  
bool b3 = true; // True  
bool b4 = b2 && b3; // False  
bool b5 = b2 || b3; // True
```

Sometimes two isn't enough. You might have true, false, and ‘don’t know’. Or you might have a choice of red, green, or blue. An enumeration is a way of having a limited number of

choices in a sort of static list. In C#, an enumeration data type is called **enum**, and consists of named constants which are known as the enumerator list.

An example is an enumeration for the days of the week. You can represent these days of the week by using numbers, such as Sunday is day 1, Monday is day 2, and so on. When you define the enumeration you set up a list of named constants that will be represented as numeric values, but in source code you can use the meaningful names. By default, enums start at 0 unless you say otherwise. This means Sunday would be day 0, Monday day 1, and so on. If that's not what you wanted, you can define an initializer when you create the enum. The following are some examples of enumerations:

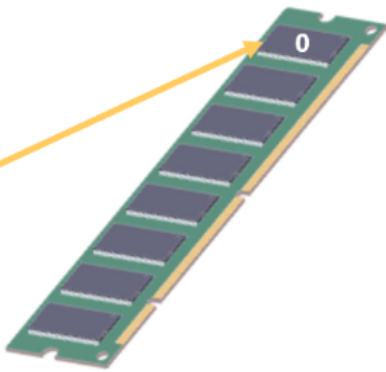
```
enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday,  
Friday, Saturday };  
  
enum Months { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,  
Oct, Nov, Dec };
```

Note that we've chosen to make the name of the enum start with an uppercase letter. This is because of a convention you'll see in the next lesson.

Variables

Introduction to Variables

```
int loopCounter ;  
loopCounter = 0;
```



The computer processes data that uses various data types by, for example, performing mathematical calculations on numeric values. These values might come from user input, from sensors, or be read from a data file. The computer needs to be able to store and update values during this processing. To be able to do this in code, we declare a variable; a named

storage location that points to a location in memory where the actual data is stored. We declare a variable by stating the data type, the variable name, and optionally an initial assignment or initialization. The compiler will generate code to allocate memory according to the data type.

Variable names must begin with a letter or an underscore. They are case-sensitive for most programming languages, including C#. Variable names cannot be the same as the keywords in the programming language you are using, and they must be unique within a given scope. These are the rules of the language, and otherwise you have free rein, but a good practice is to choose names that make the code easy to read and understand. We'll cover the topic of good naming practices towards the end of the course. Some examples of valid variable names are currentAccountName, _studentFirstName, foo, BAR, student1, i. But the following are not valid:

- 1stStudent – starts with a number
- int – reserved word or keyword
- \$amount – starts with non-alphanumeric character

Note that foo, FOO and Foo could all be used as different variables, but would make for confusing code with a high risk of errors.

We've seen a few variable declarations in the preceding sections. Here's an example that shows how to declare a variable called loopCounter that will hold an integer data type.

```
int loopCounter;
```

In the declaration above we didn't include an initializer. In C# that's not really a problem because the compiler will warn us if we attempt to use an uninitialized variable. It's worth noting that some computer languages don't have such a check, and it's possible that a memory location might contain random data from a previous program, which can create all kinds of weird bugs that are very hard to track down. If you ever find a program that produces different results each time you run it (and there's no external data or random number function) then something very bad has happened. This can sometimes indicate a hardware error, or some obscure bug like the type just described. This is why some people recommended always initializing variables. Let's do that for loopCounter:

```
int loopCounter;  
loopCounter = 0;
```

Or we could write this more succinctly as follows:

```
int loopCounter = 0;
```

Here are some more examples of variable declarations and assignments:

```
float tax = 0.09F;  
byte aByte = 245;  
long bigNumber = 9223372036854775807;  
char letter = 'C';  
string phrase = "This is a string";
```

Note that the assignment for the float type has the letter 'F' appended to the value 0.09. In C# the floating-point literal value 0.09 is interpreted as a double-precision number. The 'F' is required to instruct the compiler to assign a single-precision value. A double value will not automatically be converted because you are taking a larger data type and attempting to assign it to a smaller one, which could result in a loss of precision. Automatic conversion in the opposite direction is allowed, though. You'll get a compiler error if you omit the 'F', and Visual Studio won't allow you to run the compiler until you fix it. There are also other decorators for literal values, such as D for double, U for unsigned, and M for decimal.

The values of variables normally change during program execution. In the following example a variable is initialized as part of the declaration:

```
int uninitializedvariable;  
int initializedvariable = 3;  
initializedvariable = uninitializedvariable; // won't compile  
uninitializedvariable = 5;  
initializedvariable = initializedvariable +  
uninitializedvariable;
```

Note that we can choose to declare a variable and assign it a value later in the program, but we can't use an unassigned variable. The compiler (and the IDE) won't allow it.

```
.0;
```

Expressions

Expressions in C#

Example expressions:

- + operator

```
a + 1
```

- / operator

```
5 / 2
```

- + and - operators

```
a + b - 2
```

- + operator (string concatenation)

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

Expressions are the most fundamental programme element to evaluate and work with data. They are composed out of the operands and operators according to the syntax of the C# language. The operands are the variables, constants, and literal values call mom and function calls. As basic arithmetic operations, there are symbols for logical comparison, bit manipulation of binary numbers, and string concatenation. When the expression is evaluated it is usually assigned to some variable Although it might form a parameter to a function call or be part of a comparison operator. You can build up very complex expressions, although it usually makes sense to break them up into smaller expressions which can be assigned to variables and combined in a final step.

Operators can be divided into two main types. Unary operators take a single operand, for example the binary ‘not’ operator (!). Unary operators can also be further categorized as prefix and postfix operators, depending on whether they come before or after their operand. The unary negation operator is a prefix operator, whereas the increment and decrement operators (++ and –) have both prefix and postfix forms.

Binary operators take two operands. The +, -, *, /, and % operators perform addition, subtraction, multiplication, division, and modulus respectively. The ‘-‘ symbol can also be used as both the binary subtraction operator, and as the unary negation operator. With operators, and especially arithmetic operators, the order of evaluation becomes important. For example the expression:

```
int a = 2 + 3 * 4
```

will evaluate to 14 and not 20. This is because the multiplication operator takes **precedence** over the addition operator. If you wanted to multiply the sum of 2 and 3 by the number 4, you would have to use brackets, and write:

```
int a = (2 + 3) * 4
```

C# also includes a conditional operator ‘?:’, which is strictly a ternary operator because it takes three arguments (a condition and two values or expressions).

The table below lists the various operators available in the C# language:

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)	?:

For more information on C# operators, see the documentation at: <https://aka.ms/3u6MgJ3>.

Working with Strings

Working with 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.
}
```

C# includes a built-in string library that makes it easy to work with text strings. You can concatenate two strings by using the string concatenation operator ‘+’. This binary operator will return the result of joining the two strings together as in the following example:

```
string s1 = "Hello ";
string s2 = "world!";
string result = s1 + s2; // result contains "Hello world!"
```

Strings in C# are immutable, which means that once created they can't be changed. This means that in the example, **result** is a new string that is then initialized to the concatenated string. The values in **s1** and **s2** will be thrown away when they eventually go out of scope. In this simple example there isn't a problem, but if you are doing a lot of string concatenation this becomes quite inefficient. For this reason, the string library includes a **StringBuilder** class for occasions where you're doing a lot of string concatenation. The same code written using the **StringBuilder** class would look like this:

```
string s1 = "Hello ";
string s2 = "world!";
```

```
StringBuilder sb = new StringBuilder();
string sb = s1 + s2;
string result = sb.ToString();
```

That might seem a lot of effort for this simple example, and probably isn't justified. It's really only necessary when you are concatenating a lot of strings, for example in a loop.

There's a wide range of functions included in the string classes for finding substrings, extracting sections of a string, splitting strings, and so on. If you're doing a lot of string manipulation it is sometimes more efficient to use another class called **Regex** which can be used to work with **regular expressions**. This is a special compact language for matching and replacing in strings, and is very powerful.

To learn more about the string libraries in C#, see <https://aka.gd/38vxflw>, and for more about regular expressions, see <https://aka.gd/3DLBT0r>.

Question: You're building a program to record historic football results, and need to store the year. What's the smallest integer type you could safely use?

Answer: A **short** integer would be large enough to store dates, at least until the year 32,767. If you needed a few more millenia you could use a **ushort** instead.

Lesson 3 - C# Language Constructs

As you work with C# you will make numerous conditional branches in your code. In this lesson we'll review the structures in C# that support decisions and iterations, and also at some of the ways of storing collections of data rather than individual variables.

Lesson Objectives

After completing this lesson, you'll be able to:

- Branch on conditions.
- Create looping code structures.
- Work with arrays
- Work with namespaces

Conditional Logic

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;
}
```

Early high-level programming languages used GOTO instructions to control program flow. This tended to result in ‘spaghetti’ code that was difficult to follow and understand. For that reason, more modern languages like C# use structured programming constructs like if/then/else and for loops. These are much easier to follow and make it easier to express the intent of the programmer.

You can use a number of different constructs to follow different paths. Programmers sometimes talk about ‘branching’ when the code follows different paths depending on some expression evaluation. The simplest kind of decision is the if statement, which takes a Boolean value, or an expression that evaluates to a Boolean. The block of code following the **if** only executes if the Boolean evaluates to true. For example:

```
if (value > 1.0)
{
    Console.WriteLine("Alert! value is too high!");
}
```

In this statement, the keyword **if** is followed by a condition, enclosed by parentheses, which is followed by a block of code to execute. If you have a single line of code like this you also have the option of putting it in-line and omitting the braces. Whether you do that is a matter of programming style, but if you put it all on one line you have to be alert to the risk of accidentally adding more code that was intended to be part of the code block. The indentation of the code block is a convention, and most code editors will be able to automatically do this for you.

In Visual Studio you can find the automatic document formatting on the Edit -> Advanced -> Format Document menu.

Usually with an **if** statement you want to have two paths, depending on the condition. For example we might want another more reassuring message if all is well, so we could write the following:

```
if (value > 1.0)
{
    Console.WriteLine("Alert! value is too high!");
}
else
{
    Console.WriteLine("Value is totally fine, don't worry.");
}
```

You might want to test for more than two possibilities. In that case you can keep adding more **ifs**:

```
if (value > 1.0)
{
    Console.WriteLine("Alert! value is too high!");
}
else if(value > 0.9)
{
    Console.WriteLine("Warning - value is close to the danger
area.");
}
else
{
    Console.WriteLine("Value is totally fine, don't worry.");
}
```

```
}
```

If you find yourself adding more and more if/else statements you can end up with ugly code that is difficult to follow. That probably indicates that you need to use a **switch** statement instead. In effect a switch statement is a more concise way of writing a list of if/else statements. It selects a single case section to execute from a list of candidates based on a match with the switch expression. The switch statement contains one or more case specifications, each followed by one or more statements. The switch statement can also contain a default label that will get used if no other case is matched. The default is not always necessary, but it's recommended. Here's an example of a switch statement:

```
switch (value)
{
    case 0: Console.WriteLine("All is well"); break;
    case 1: Console.WriteLine("Warning!"); break;
    case 2: Console.WriteLine("Minor Alert!"); break;
    case 3: Console.WriteLine("Major Alert!"); break;
    case 4: Console.WriteLine("Panic!"); break;
    default: Console.WriteLine("Unknown condition"); break;
}
```

In the sample above, the selector is an integer, although other types can be used, for example a string. The code block for each case is followed by a break statement to exit the switch statement. If you forget the break, it will carry on to the next case and execute that as well. You can wrap the statements for each case with braces to make it clear that they form a code block, but it's not necessary. Because the switch is matching the switch variable to choose a case, it's not quite the same as an **if** statement, where we can use any Boolean expression such as a comparison operator.

A switch statement is a good choice when you want to compare a single value against many possible values, and when you can express the code very succinctly for each case. Ideally, the case statements should be sufficiently short that you can take in the whole of the switch statement at a glance. As a general coding rule it's recommended to have each code block enclosed in braces, but for the switch we might instead err on the side of using very concise formatting. In the example above we managed to fit every case onto a single line, which makes it a little easier to read.

If you find you need more than a couple of lines for each case, then a good strategy is to extract that code into its own function and then just put the function call in the case code block. This works especially well if several or all of the case statements can be expressed using the same function call, but with different arguments.

You can get more information about if/else and switch statements at:
<https://aka.gd/3DZgzVn>.

Creating and Using Arrays

Creating and Using Arrays

- C# supports:
 - Single-dimensional arrays
 - Multidimensional arrays
 - Jagged arrays
- Creating an array:

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

- Accessing data in an array:
 - By index

```
int result = arrayName[2];
```

- In a loop

```
for (int i = 0; i < arrayName.Length; i++)  
{  
    int result = arrayName[i];  
}
```

Frequently in programming tasks we have to deal with a set of objects of the same type that we can manage together as a group. An array is a sequence of elements along one or more dimensions (referred to as the array's rank). Arrays in C#, like most similar languages, are zero-indexed, meaning that the first element in an array has an index of 0. To create an array, you specify the number of elements in each dimension. For example one-dimensional array that holds ten elements would have its first element at index 0 and it's last element would have an index of 9. If we tried to reference an element with an index of 10 or more, we'd get an index out-of-range exception.

To declare an array, you would need to define the type that it contains, and then make use the **new** keyword to create the array. The size of the array is defined when the array is created, and can be an integer literal or an integer expression. In the following example we'll create an array of 10 integers:

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

We can also initialize an array with values at the time of creation:

```
int [] array = new int [] { 7, 2, 3, 9, 4, 9, 7 };
```

To create a 4-dimensional array of 10000 total floating point numbers, the dimensions need to be present in the declaration as shown:

```
float[,,,] array = new float[10, 10, 10, 10];
```

The first element will be `arrayName[0,0,0,0]` and the last will be `arrayName[9,9,9,9]`. The elements within the array can be any type, including arrays. We therefore have another way of creating multi-dimensional arrays:

```
int[][] jaggedArray = new int[3][];  
jaggedArray[0] = new int[4];  
jaggedArray[1] = new int[7];  
jaggedArray[2] = new int[9];
```

We have created a 2-dimensional array by creating an array of arrays. Since we are creating the inner arrays separately, we have the option of making them all different sizes, resulting in a jagged array rather than a rectangular one. The first element would be addressed as `jaggedArray[0][0]`, with a maximum for the first index of 2, but for the second index it would depend on the value of the first index. We could also use the initialization syntax to make this a one-liner:

```
int[][] jaggedArray = new int[3][] { new int[4], new int[7], new  
int[9] };
```

You could even embed initializers for the inner arrays, but it starts to get cumbersome.

Iteration Constructs

Iteration Constructs

- 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 means executing a set of steps repeatedly. That might be looping over all the items in an array or collection, or until some condition is satisfied. The most common type of iteration is where you need to repeat something a known number of times. You create a counter variable, increment its value each time you go round the loop, and stop when it reaches some value. The following very simple example shows a for loop.

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine(i + ", ");
}
```

This outputs the string “1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ”. A **for** statement consists of an initializer, a condition and an action, separated by semi-colons. Usually the initializer is used to initialize, and possibly declare, an iterator variable, in our case **i**. When the condition is no longer satisfied the loop will end. If the condition is false at the outset, then the loop won’t execute at all. The action is usually an increment or decrement operator that acts on the counter. Although the majority of **for** loops are some variation of the code above, you can have more complex expressions and conditions, or nothing at all.

You can use a **for** loop to iterate over the elements of an array by asking for a length. Here’s some code to initialize an array to zero:

```
for (int i=0; i < array.Length; i++) array[i] = 0.0;
```

Notice that, like the **if** statement, if you've just got a single line of code in your code block, you can do the whole thing inline. It's not always obvious how to get the length of a collection, so another option is to use the **foreach** iterator:

```
foreach (var a in array) Console.WriteLine(a);
```

This makes life simpler in most cases, but be aware that the iteration variable is immutable, so the following won't work:

```
foreach (int a in array) a = 0; // compiler error
```

In that case you'll have to find another way, such as a **for** loop with an index into the array, as we did before.

You could have a situation where you just needed to iterate until a condition was satisfied. You could use a **for** loop and leave the initializer and action empty (`for (;i<=10;)`), but this is a bit ugly, and in we have a much better option because this would be exactly the same as a **while** loop.

```
int i = 0;  
while (i < array.Length)  
{  
    array[i] = 0;  
    i++;  
}
```

This is a contrived example because it is the same as the **for** loop above, but in general if you have some condition to test for it's usually more suited to a **while** loop. You sometimes see code like the following:

```
while (true)  
{  
    // some code that will break or return  
}
```

On the face of it, this loop will continue forever. In practise we would normally have something in the code block that would either break out of the loop or return from the function when some condition is satisfied. There are occasions though, e.g. in embedded systems, where you intentionally want the loop to run forever.

A **while** loop evaluates the condition first, so the loop might not execute at all. You can also do the test at the end of the loop, and this is called a **do/while** loop.

```
bool finished = false;  
do  
{  
    // some code that changes the value of finished  
} while (!finished);
```

In this case the only difference to the **while** loop is that it will execute at least once, even if the condition was false at the outset (i.e. `finished` was initialized to true in the example above).

To learn more about iteration constructs, see: <https://aka.gd/3LP7zoA>.

Namespaces

- Use namespaces to organize classes into a logically related hierarchy
- .NET class library includes:
 - System.Windows
 - System.Data
 - System.Web
- Define your own namespaces:

```
namespace FourthCoffee.Console  
{  
    class Program { . . . }
```

- Use namespaces:
 - Add reference to containing library
 - Add using directive to code file

With any large application, sooner or later, there's the problem of name collisions. For example, if we created an array and called it simply 'array', there's a very high probability that the same name would be used somewhere else in the application. Another issue is the difficulty of organising and finding classes and other types within a single flat naming system. As one of the ways of avoiding these problems, .NET has the concept of hierarchical namespaces that is used to disambiguate any objects that use the same name in different

parts of the code, and to organise and group related classes. Namespaces are used for the many classes and types that are provided by the .NET framework, and you can also define your own namespaces for your code.

In .NET, **System** is the most important namespace as it contains classes for interacting with the operating system. System is further subdivided into many more namespaces such as System.Windows (for Windows UI development), System.IO (for File I/O), System.Data (for database access) and System.Web (for web development).

When you create a Visual Studio project, you automatically get the most frequently used .NET framework assemblies referenced for you. If you need to use a type from an assembly that hasn't been referenced in your project then you'll need to add the reference. You can do this in the **Add Reference** dialog in Visual Studio.

You'll also want to add a **using** directive so that you don't have to spell out the fully qualified name every time you reference a type. For example, you'd have to write System.IO.File every time you used the File class, but you can avoid this by adding using System.IO at the top of your source file. We have already made extensive use of System.Console, and we referenced it as just 'Console'. Normally we'd need to add using System to avoid having to specifically say System.Console every time. In this particular case we didn't need to add a using statement to Program.cs because it is automatically included for us.

Learn more about namespaces in .NET at: <https://aka.gd/3LNhIC0>.

Debugging

Debugging

- 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

An essential development skill is to be able to debug code. Some coding errors can be caught by the compiler, and are usually very easy to rectify, but the real problems occur when you get logic errors at runtime. These are often not very obvious and so you need to step through your code and look at the values of variables, parameters, the callstack, and other runtime properties.

Fortunately, Visual Studio includes a number of powerful debugging tools. Most of these rely on you performing a debug build of the application, which will include additional information that the debugger uses to reconcile the executing code with the source code in the editor. Normally when you are ready to release your application you will make a release build in which most of this information is omitted.

In order to step through code you first need to stop programme execution. You do this by setting breakpoints in the code editor. You can do this by selecting a line of code and then choosing Debug->Toggle Breakpoint from the menu. Alternatively you can click the grey area to the left of the code line numbers. Once your application has stopped at the breakpoint you can start to use the many tools in Visual Studio such as the **Immediate Window** or the **Autos**, **Locals**, and **Watch** windows to view and possibly modify the values of variables and parameters.

Once you have done this you probably want to continue execution. You can choose to allow the application to run until the breakpoint is hit again, or the application exits. You can also step through your programme one instruction at a time, or step into or out of methods and functions.

For more information about Visual Studio debugging see: <https://aka.gd/3Jwad0q>.

Lab: Developing the Class Enrollment Application

Lab: Developing the Class Enrollment Application

Lab 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.

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 the 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.

Lab: Developing the Class Enrollment Application

Objectives

Exercise 1:

Implementing Edit Functionality for the Students List

Exercise 2:

Implementing Insert Functionality for the Students List

Exercise 3:

Implementing Delete Functionality for the Students List

Exercise 4:

Displaying a Student's Age

Module Review and Takeaways

Review Questions

Question: You need to create a .dll file. What Visual Studio template should you select?

- A. Windows Forms application
- B. Console application
- C. Class library
- D. WCF Service application
- E. ASP.NET MVC application

Answer: C – you use a class library to generate a .dll file that can be imported into other projects.

Question: You have the following code:

```
var count = 3;  
for(int i=3; i < 10; i++) count++;  
Console.WriteLine(count);
```

When the code executes, what will be the number output to the command line?

- A. 3
- B. 7
- C. 10
- D. 13

Answer: C – the loop will execute 7 times, adding 7 to the starting value of count, so count will have a final value of 10.

Module 2 – C# Language Concepts

Module Overview

In this module we'll introduce methods, how to handle exceptions, and how to implement logging and tracing.

Objectives

After completing this module, you'll be able to:

- Create and invoke methods.
- Use method overloading and optional parameters.
- Handle exceptions.
- Monitor running applications

Lesson 1 – Methods

When programmes go beyond the simple examples we've looked at so far, it becomes increasingly important to breakdown the functionality by using functions or other means of separating different parts of the code. C# is an object oriented language that uses classes as a means of achieving this separation. the functions that we build that are associated with classes are called **methods**.

In principle, a method can be as simple or complex as you wish. but in practise it's very important to try to avoid excessively long methods because they usually end up doing too much. in order to keep code readable, a good practises to keep methods as short as possible. In this module we're going to look at creating and invoking methods.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the purpose of methods.
- Create methods in code.

- Call methods from code.
- Use debugging techniques.

What Is a Method?

What Is a Method?

- Methods encapsulate operations that protect data
- .NET Framework applications contain a **Main** entry point method
- The .NET Framework provides many methods in the base class library

Being able to create and call methods lies at the heart of object oriented program. methods enable us to encapsulate functionality and protect the data that is stored inside a type, such as a class.

In a real application, as opposed to the demonstration code we have been looking at so far, you will have many classes and many methods each with a particular purpose. Some of these methods are fundamental to an application, such as the **Main** method that acts as an entry point for the application. You might be wondering, if this is so important, where is the **Main** method and all the sample code that we have been looking at so far. In applications using .NET 6 and later these are automatically created by the compiler behind the scenes, if they are not there in the source code. Here's what our Hello World program would look like if this wasn't the case:

```
using System;
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

```
}
```

```
}
```

The using statement, the namespace that's the name of the application, the Program class, and the static Main method, are all generated in the background. If you build a C# console app with an older version of .NET then the template will scaffold the code shown above. The Main method gets called when the application starts up. You can change the namespace name, or the class name, or the return type, and it will still work. But if you change the method name, even to "main", or it isn't static, then the compiler will give an error that there is no entry point for the program.

When you create a method you can also set its visibility, for example public or private. Public methods are intended to be called from elsewhere in the code, whereas private methods are only visible inside the type itself. It's usually a good idea to design a type with a few public methods for external use, and then private methods to do most of the implementation, so that you can change your mind later about how exactly those private methods work without affecting code that calls your methods. That external code might be written by different people, or different teams.

The .NET framework classes that we use extensively use this approach. This means that we can rely on any public methods to remain the same most of the time, and only the private methods change as the framework code continues to be developed. We don't see those changes to private methods, and in many cases they are concealed from us. That's a good thing, because we don't want our code to break when implementation details change.

To learn more about the Main method and its parameters, see: <https://aka.gd/3603iiL>.

Creating Methods

Creating Methods

- Methods comprise two elements:
 - Method specification (return type, name, parameters)
 - Method body
- Use the `ref` keyword to pass parameter references

```
void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

- Use the `return` keyword to return a value from the method

```
string GetServiceName()
{
    return "FourthCoffee.SalesService";
}
```

A method consists of two components: a specification and a body. In the method specification, the name of the method is defined, as well as its parameters and return type. It is the combination of a method name and a list of parameters that is referred to as a method signature. Return values are not considered part of a method's signature. In a class, each method must have a unique signature.

Naming Methods

Method names are subject to the same syntactic restrictions as variable names. The method must start with a letter or underscore, and it can only contain letters, underscores, and numbers. As Visual C# is case sensitive, a class can contain two methods that have the same name and differ only in the case of one or more letters—although this is not a good coding practice.

A recommended best practice when choosing a method's name is to name methods using verbs or verb phrases. It makes it easier for other developers to understand your code. You can use UpperCamelCase in order to use multiple words to create informative method names.

Implementing a Method Body

Method bodies are blocks of code containing a variety of Visual C# programming constructs, enclosed in braces.

Defining variables within a method body will create them only while the method is executing; when the method returns, they go out of scope.

In the following example, a variable named isServiceRunning appears in the body of the StopService method. The variable isServiceRunning can only be accessed inside the StopService block. If you attempt to use the isServiceRunning variable outside the scope of a method, the compiler will produce a compile error with the message that 'isServiceRunning' doesn't exist in the current context.

```
void StopService()
{
var isServiceRunning = FourthCoffeeServices.Status;
...
}
```

Specifying Parameters

The parameter list of a method is a set of local variables that are populated by the caller when the method runs. You define the parameters in parentheses. Sometimes methods don't have any parameters at all, in which case the parameter list is a pair of empty parentheses. Each parameter is defined by the parameter type and its name. In a similar way to method names, parameter names should be chosen to clearly communicate their intent. There is no cost associated with the length of a name for a variable, method or parameter. A good approach is to use multiple words or a phrase, and camel case can be used to visually separate the words as in method names. A common convention for variable and parameter names is to use lowerCamelCase.

In the following example code we have a method that takes two parameters, an **int** and a **Boolean**:

```
void StartService(int upTime, bool shutdownAutomatically)
{
// Perform some processing here.
}
```

Normally parameters are passed into the method as values, but you can also use the **ref** keyword in order to pass the parameter by reference. This means that any changes to the parameter that are made inside the method will affect the variable used in the calling method. In the following code sample you can see a parameter being passed in this way:

```
void StopAllServices(ref int serviceCount)
{
```

```
serviceCount = 10;  
}  
  
int count = 0;  
  
StopAllServices(count);
```

When the call to **StopAllServices** returns, **count** will have the value 10.

You can learn more about using the **ref** keyword at: <https://aka.gd/37AMtv>.

Specifying a return type

You always have to define a return type for a method. If the method doesn't return a value then you can set the return type to **void**, in which case the use of a return statement is optional. If your method has a return type then you must include a return statement with a value of the same type. An example of how a method can return a string is shown below:

```
string GetServiceName()  
{  
    return "FourthCoffee.SalesService";  
}
```

The return statement does not have to be the last in the method, and there can even be more than one. If you have some decisione logic you need to ensure that a return statement is eventually executed. If there are any code paths that do not return a value then the compiler will produce a syntax error. For example the following code will not compile because if language were some value other than "en" it would not return a value for the string.

```
string GetServiceName(string lang) // won't compile  
{  
    if(lang == "en")  
        return "FourthCoffee.SalesService";  
}
```

Invoking Methods

Invoking Methods

To call a method specify:

- Method name
- Any arguments to satisfy parameters

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.
void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

You invoke a method to execute the code contained within it from within your application. You don't need to know how a method's code works to use it. If the code is in a class that's inside an assembly for which you don't have the source code, e.g. a .NET framework class library, you might not even have access to it.

To call a method, type the method name followed by any arguments in brackets that correspond to the method parameters. The code sample below explains how to call the StartService method, passing int and Boolean variables to satisfy the method's parameter requirements:

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.

void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

If the method returns a value, you typically assign it to a variable of the same type in the calling code. If you don't assign it or use it in some other way, for example as an argument to another method call, then the return value is discarded. The following code sample shows how to save the GetServiceName method's return value in a variable called serviceName.

```
string GetServiceName()
{
    return "FourthCoffee.SalesService";
}
var serviceName = GetServiceName();
```

To learn more about methods, see: <https://aka.gd/3LSE45I>.

Debugging Methods

Debugging Methods

- Visual Studio provides debug tools that enable you to step through code
- When debugging methods you can:
 - Step into the method
 - Step over the method
 - Step out of the method

Visual Studio's debugging features allow you to go through code one statement at a time while debugging your programme. This is a very important feature since it allows you to test the logic used by your application one statement at a time. You can, for example, step through each line in each method that is performed, or you can skip statements inside a method that you know are accurate. You may even entirely step over code without executing the statements.

When debugging methods, you can make use of the three debug features listed below to control whether you step over, into, or out of a method:

- **Step Into** performs the statement at the current execution point. If the statement is a method call, the current execution location will be moved to the method's code. After you've stepped inside a method, you may continue executing statements within it one line at a time. The Step Into button may also be used to launch a programme in debug mode. If you do this, the programme will enter break mode the moment it starts.
- **Step Over** executes the statement at the current execution point. This functionality, however, does not step into code within a method. Instead, the method's code executes, and the execution position shifts to the statement following the method call. The only exception is if the method or property's code

contains a breakpoint. In this situation, execution will continue until the breakpoint is reached. When you use Step Over while the application is closed, it will start in debug mode and break on the first line.

- **Step Out** allows you to run the method's remaining code. Execution will proceed until the method returns and then pause at the statement that called the method.

To learn more about the debugging tools in Visual Studio, go to: <https://aka.gd/3Kyq7sS>.

Question: True or False: a method must always return a value?

Answer: False. Use **void** as the type for methods that do not return a value.

Lesson 2 – Method Overloading

You've already seen how to create a method that takes a set number of parameters. Depending on the situation, you may construct a single generic method that requires a varied set of parameters. To meet this requirement, you can build overloaded methods with the same method name but each with distinct signatures. In other cases, you may want to define a method with a single signature and fixed number of parameters, but that allows an application to specify arguments for only the parameters that it requires. This can be accomplished by creating a method that accepts optional parameters. You can also use named parameters that allow the caller to specify arguments to satisfy the parameters by name.

This lesson will show you how to write overloaded methods, as well as how to define and use optional parameters, named arguments, and output parameters.

Lesson Objectives

After completing this lesson, you'll be able to:

- Create overloaded methods.
- Use optional parameters.
- Use named arguments
- Use output parameters

Creating Overloaded Methods

Creating Overloaded Methods

- Overloaded methods share the same method name
- Overloaded methods have a unique signature

```
void StopService()
{
    ...
}

void StopService(string serviceName)
{
    ...
}

void StopService(int serviceId)
{
    ...
}
```

When you create a method, you may discover that it needs different sets of parameters in different situations. Overloaded methods allow you to define many methods that have the same capability but that take different parameters based on the context the way they are called. To distinguish itself from the other overloads of the method, each must have a unique signature. A method's signature comprises the name of the method as well as the list of parameters. The signature excludes the return type, so you can't design overloaded methods that only differ in their return type.

Three overloads of **StopService** are shown in the following code example:

```
void StopService()
{
    ...
}

void StopService(string serviceName)
{
    ...
}

void StopService(int serviceId)
{
    ...}
```

```
...  
}
```

When you call the **StopService** method, you can select which overloaded version to use. You simply give the required arguments to satisfy a certain overload, and the compiler determines which version to call based on the arguments you passed.

Creating Methods that Use Optional Parameters

Creating Methods that Use Optional Parameters

- Define all mandatory parameters first

```
void StopService(  
    bool forceStop,  
    string serviceName = null,  
    int serviceId = 1)  
{  
    ...  
}
```

- Satisfy parameters in sequence

```
var forceStop = true;  
StopService(forceStop);  
  
// OR  
  
var forceStop = true;  
var serviceName = "FourthCoffee.SalesService";  
StopService(forceStop, serviceName);
```

You can implement several implementations of the same method that accept different parameters by defining overloaded methods. When you create an application with overloaded methods, it's the compiler then selects the exact version of each method to execute to fulfill each method call.

Other programming languages and technologies can be used by developers to create applications and assemblies that do not adhere to these criteria. The capability to interoperate with programs and components created in different technologies is a major feature of Visual C#. The Component Object Model is one of the most important technologies used by Windows (COM). COM doesn't really support overloaded methods and instead employs methods with optional parameters. Visual C# also allows optional parameters, which make it possible to add COM libraries to a Visual C# project.

Optional parameters can be beneficial in a variety of other scenarios. They give a concise and straightforward solution when overloading is not possible because the parameter types are the same so that the compiler is unable to differentiate between implementations.

This code sample demonstrates how to define a method with one mandatory parameter and a couple of optional parameters:

```
void StopService(bool forceStop, string serviceName = null, int  
serviceId =1)  
{  
    ...  
}
```

You must specify all the mandatory parameters prior to adding any optional parameters when you define a method that takes optional parameters. The code example below displays a method declaration with optional parameters that causes a compile error:

```
void StopService(string serviceName = null, bool forceStop, int  
serviceId = 1)  
{  
    // will not compile  
    ...  
}
```

A method that accepts optional parameters is called in the same way you would any other method. You supply the method name as well as any required parameters. The distinction between methods taking optional parameters and those that do not is that you have the option of omitting the corresponding arguments. Where an optional parameter isn't present, the method will use the default value when it executes. The following code shows how to call the **StopService** function, as defined in the correct version above, with only one (literal) argument for the obligatory forceStop parameter:

```
StopService(true);
```

In this case the value of the parameter **serviceName** will be a null value. Depending on the implementation of StopService this may be perfectly acceptable and the algorithm might have some way of dealing with this. That's really what you're saying when you make a parameter optional. Maybe there is some default fall-back service, in which case that could be defined as the default value.

If you want to specify the value for **serviceName**, then you have the option of supplying an argument for it:

```
StopService(true, "MyServiceName");
```

Calling a Method by Using Named Arguments

Calling a Method by Using Named Arguments

- Specify parameters by name
- Supply arguments in a sequence that differs from the method's signature
- Supply the parameter name and corresponding value separated by a colon

```
StopService(true, serviceID: 1);
```

When you call a regular method with positional parameters, the order of the arguments in the call to the method must match the order and position of the arguments in the method signature. If the parameters are not properly aligned and the types don't match, you'll get a compiler error.

The ability to specify parameters by name in Visual C# allows you to supply arguments in a different order than that defined in the method signature, resulting in a more flexible approach. When using named arguments, you must specify the parameter by name along with the associated value, separated by a colon.

The code sample that follows demonstrates how to use the StopService function by passing the serviceID parameter through named arguments:

```
StopService(true, serviceID: 1);
```

You can simply omit parameters when using named and optional parameters together. Any missing optional parameters are set to their default values. However, omitting any mandatory parameters will result in your code failing to build.

You might use a combination of named and positional parameters. However, all positional arguments must still be specified ahead of any named arguments.

To learn more about named arguments, see: <https://aka.gd/3xmKEwN>.

Creating Methods that Use Output Parameters

Creating Methods that Use Output Parameters

- Use the **out** keyword to define an output parameter

```
bool IsServiceOnline(string serviceName, out string statusMessage )  
{  
    ...  
}
```

- Provide a variable for the corresponding argument when you call the method

```
var statusMessage = string.Empty;  
var isServiceOnline = IsServiceOnline(  
    "FourthCoffee.SalesService",  
    out statusMessage );
```

Using a return statement, a method can return a single value to the code that called it. What if you want to return more than one value? One option would be to contrive some complex type to embed the return values, usually called a ‘tuple’, which is probably the best way to do this. But there's also an easier way. If the calling code requires more than one value to be returned, you can make use of output parameters to pass additional data out of the method. When an output parameter is added to a method, the code within the method body is supposed to assign the parameter a value. When the method returns, the value of the variable supplied as the method call's corresponding argument will have been updated to the new value.

To specify an output parameter, the parameter is prefixed with the **out** keyword in the method signature. The following sample code demonstrates how to construct a method that accepts output parameters:

```
bool IsServiceOnline(string serviceName, out string  
statusMessage)  
{  
    var isOnline = FourthCoffeeServices.GetStatus(serviceName);  
    if (isOnline)  
    {  
        statusMessage = "Services is currently running.";  
    }  
    else  
    {
```

```
statusMessage = "Services is currently stopped.";  
}  
  
return isOnline;  
}
```

A method may have an unlimited number of output parameters. When declaring an output parameter, the parameter must be assigned a value before the method returns; otherwise, the code will fail to compile.

When you call a method with an output parameter, you have to specify a variable for the matching argument and prefix it with the `out` keyword when calling the method. Your code will fail to compile if you attempt to define an argument that isn't a variable or if the `out` keyword is omitted.

The following code sample demonstrates how to call a method with an output parameter:

```
var statusMessage = string.Empty;  
  
var isServiceOnline =  
IsServiceOnline("FourthCoffee.SalesService", out statusMessage);
```

Inline declaration of variables for use with the `out` keyword is supported in newer versions of C#. This avoids the need to create a separate variable declaration. The following sample code demonstrates how to call a method with an output argument and how to define the variable in line:

```
var isServiceOnline =  
IsServiceOnline("FourthCoffee.SalesService", out var  
statusMessage);
```

Find out more about output parameters and the `out` modifier at: <https://aka.gd/3xmKEwN>.

Question: True or False: a method's signature includes the return type?

Answer: False: the return type does not form part of the signature for the purpose of defining method overloads.

Lesson 3 – Exception Handling

Good exception handling is critical notion for providing a positive user experience and limiting data loss. In this lesson, you'll learn how to implement effective exception handling in your applications and how to use exceptions in your methods to gracefully indicate an error situation in applications that use your types and methods.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the role of exceptions.
- Write exception handling code.
- Use a **finally** block.
- Throw exceptions in code.

What Is an Exception?

What Is an Exception?

- An exception is an indication of an error or exceptional condition
- The .NET Framework provides many exception classes:
 - `Exception`
 - `SystemException`
 - `ApplicationException`
 - `NullReferenceException`
 - `FileNotFoundException`
 - `SerializationException`

Countless things can go wrong when an application is running. Certain errors may occur as a result of defects in the application's logic, while others may be outside your application's control. For instance, your program cannot ensure the existence of a file or the availability of a required database. When designing an app, you need to consider keep in mind how to make sure that it can gracefully recover from such difficulties. While it's a good idea to verify the return values of methods to confirm that they have performed correctly, this approach is not always good enough to handle all possible problems because many methods don't return a value. You also need to understand the cause of the failure. And there's a whole class of unexpected errors, such as memory exhaustion, that are simply too numerous and complex to handle using a return value.

Historically, applications took advantage of some kind of global object to keep track of errors. When code generated an error, it set the contents of this object to identify the error's cause and then returned to the caller. The calling code was responsible for inspecting the error object and determining how to handle it. This strategy, however, is not very robust. It's far too easy for a developer to overlook proper error handling.

How Exceptions Propagate

The .NET framework makes use of exceptions to assist in resolving these situations. An exception is a signal that an error or something outside the normal logic flow has occurred. When a method detects something unexpected has occurred, for example, when the application attempts to open a non-existent file, the method can throw an exception.

An exception is an event in your application that's created in response to something going wrong. It's a departure from the regular flow of execution in your code, not a mechanism for controlling normal program flow. If you expect something to happen as part of your code flow, then you should use an if/then/else construct. For example, if it's highly likely that a file doesn't exist, test for this before executing code to use it. But if the file should normally always be there, then that's a good use case for structured exception handling.

When an exception is thrown inside a method, the caller code should be prepared to handle it appropriately. If the calling code does not catch the exception, the function is interrupted (in effect it returns immediately) and the exception is forwarded to the code that called the method. This procedure is repeated up the call stack, so that the exception "bubbles up", until an exception handler is reached. Once the exception-handling code has finished, execution continues in this piece of code. If no exception handler is found, the process or application will end and a message will be displayed to the user. This default user message is rarely very user-friendly, so it's best to avoid this if possible.

The Exception Type

It's a good idea to include details of the original problem when an exception occurs, so that the method handling the exception can take the necessary corrective action. Exceptions are all derived from the `Exception` base class in the .NET framework, which contains details about the exception. The act of throwing an exception produces an `Exception` object which can add information about the error's cause. After that, the `Exception` object is forwarded to the code responsible for handling the exception.

The following table summarizes several of the .NET framework's exception classes:

Exception Class	Namespace	Description
<code>Exception</code>	<code>System</code>	raised during the execution of an application.
<code>SystemException</code>	<code>System</code>	raised by the CLR. The <code>SystemException</code> class is

		the base class for all the exception classes in the System namespace.
ApplicationException	System	raised by applications and not the CLR.
NullReferenceException	System	caused when trying to use an object that is null.
FileNotFoundException	System.IO	raised when a file does not exist.
SerializationException	System.Runtime.Serialization	occurs during the serialization or deserialization process.
ArithmaticException	System	thrown for errors in an arithmetic, casting, or conversion operation.

For more on exception handling, see: <https://aka.ms/375FESN>.

Handling Exception by Using a Try/Catch Block

Handling Exception by Using a Try/Catch Block

- Use try/catch blocks to handle exceptions
- Use one or more catch blocks to catch different types of exceptions

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
```

Exception handling is implemented using the **try/catch** block. This is the fundamental programming component that allows you to use Structured Exception Handling (SEH). You

use a **try** block to wrap any code that may fail and throw an exception. One or more **catch** blocks are then used to catch any exceptions that may arise.

The following code sample demonstrates the syntax for defining a try/catch block to protect a call to the function calculateReciprocal(), which takes a parameter x and returns the value of 1/x. Even if we didn't know what calculateReciprocal did, we might decide that we wanted to ensure that it didn't produce any unexpected crashes due to exceptions being thrown. We could also put exception handling in the calculateReciprocal() function itself, if we were responsible for writing it. But we can also do it within the caller as follows:

```
decimal calculateReciprocal(decimal x)
{
    return 1 / x;
}

try
{
    var result = calculateReciprocal(0);
    Console.WriteLine("Reciprocal = " + result);
}

catch (ArithmetricException e)
{
    Console.WriteLine("Arithmetric error: " + e.Message);
}

Catch (Exception e)
{
    Console.WriteLine("Error: " + e.Message);
}
```

Running this will result in the following output:

Error: Attempted to divide by zero.

In the example above the catch statement refers to an **Exception**, which is the most generic type of exception. A good practice is to be specific about the exceptions you catch. This means you can anticipate and trap precise errors, and then write code to handle the

exception situation. Depending on the situation and the type of the exception, you may be able to do something to get around the problem, or to help the user, rather than just give them an error message, or at the very least write something to a log file. You might also just tell the user that whatever they are doing hasn't worked and give them the option to try again. For example, if you catch a **FileNotFoundException**, you might ask the user to confirm the file name – perhaps they made a mistake in entering it.

Perhaps we can anticipate the divide-by-zero error in the code above and provide a more specific error message. We can still retain the generic catch block for anything else, or we can add as many other exception types as we like. Here's an improved version of the code:

```
decimal calculateReciprocal(decimal x)

{
    return 1 / x;
}

try
{
    var result = calculateReciprocal(0);

    Console.WriteLine("Reciprocal = " + result);
}

catch(DivideByZeroException)
{
    Console.WriteLine("You tried to divide by zero!");
}

catch(ArithmeticException e)
{
    Console.WriteLine("Arithmetric Error: " + e.Message);
}

catch (Exception e)
{
    Console.WriteLine("Error: " + e.Message);
```

```
}
```

This results in the following output:

```
You tried to divide by zero!
```

Notice that we didn't use the `DivideByZeroException` exception object to form the message, so we can just put the exception type on its own in the catch brackets. We can also catch any other arithmetic errors as well as any other exception type. When you use multiple catch blocks you need to make sure that you catch the most specific exceptions first and the most generic last.

To find out more about try/catch blocks, see: <https://aka.ms/3vjcCqS>.

Using a Finally Block

Using a Finally Block

- Use a finally block to run code whether or not an exception has occurred

```
try
{
}
catch (NullReferenceException ex)
{
    // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
    // Catch all other exceptions.
}
finally
{
    // Code that always runs.
}
```

Some common problem areas such as accessing the file system or network resources involve getting things like file handles and network client objects that need to be released after they are used. This is critical code that needs to run, whether or not an exception is thrown, otherwise your application will be likely to leak resources, or leave resources tied up by the operating system after it exits.

You could do this in the `catch` block, but as we saw in the previous example, we may have several catch blocks, and we it run even if there is no exception. That's why we can have an optional `finally` code block, which can be added at the end of the catch blocks. The contents of the `finally` block get executed whether or not an exception was thrown in the code inside the try block. Here's what this looks like:

```
try
{
    AllocateResources();
    doSomethingThatMightThrowException();
}

catch(ArithmeticException e)
{
    Console.WriteLine("Arithmetic Error: " + e.Message);
}

catch (Exception e)
{
    Console.WriteLine("Error: " + e.Message);
}

finally
{
    // clean up code, e.g. release file handles, etc.
}
```

We'll look at the details of the type of resources you might want to release in a later module. For now, note that the finally block is a good place to do any clean-up because it always gets executed.

To learn more about about try, catch and finally blocks, see: <https://aka.ms/37GzxUA>.

Throwing Exceptions

Throwing Exceptions

- Use the `throw` keyword to throw a new exception

```
var ex =  
    new NullReferenceException("The 'Name' parameter is null.");  
throw ex;
```

- Use the `throw` keyword to rethrow an existing exception

```
try  
{  
}  
catch (NullReferenceException ex)  
{  
}  
catch (Exception ex)  
{  
    ...  
    throw;  
}
```

When one of the runtime libraries encounters an error, we talk about it ‘raising’ or ‘throwing’ an exception. You can also throw an exception in your own code. You might do this because you want to use the exception handling mechanism to manage an error situation. Another reason is if you catch an exception but then you decide (based on some logic) that you want the exception to be handled at a higher level in the code. In that case you can either ‘re-throw’ the existing exception:

```
try  
{  
    doSomethingThatMightThrowException();  
}  
catch (Exception e)  
{  
    // Try to handle the exception  
    ...  
    if(couldntHandle)  
    {  
        throw;
```

```
}
```

```
}
```

...or throw a new exception:

```
throw new Exception("Some helpful error message");
```

The exception you throw can be one of the many exception classes that the .NET runtime provides, or you can also create your own exception classes by deriving a class from the **System.Exception** base class. These custom exception classes can be very specific to your application domain.

```
throw new MyCustomException();
```

Lesson 4 - Monitoring

Writing code is only one element of the process of creating real-world apps. You will most likely spend a large amount of time resolving bugs, diagnosing issues, and optimizing your code's performance. Visual Studio and the .NET framework have a number of tools that can assist you in performing these activities more efficiently. In this lesson, you will learn how to monitor and troubleshoot your applications using a variety of tools and strategies.

Lesson Objectives

After completing this lesson, you'll be able to:

- Use logging and tracing.
- Perform application profiling.
- Use performance counters.

Using Logging and Tracing

Using Logging and Tracing

- *Logging* provides information to users and administrators
 - Windows event log
 - Text files
 - Custom logging destinations
- *Tracing* provides information to developers
 - Visual Studio Output window
 - Custom tracing destinations

Logging and tracing are two similar functions but they perform separate functions. When you add logging to your programme, you add code that publishes data to a destination log, such as a text file or the Windows event log. Logging allows you to deliver more information to users and administrators about what your code is doing. For example, if your programme handles an exception, you might log the details to the Windows event log so that the user or a system administrator can troubleshoot any underlying issues.

Tracing, on the other hand, is used by developers to observe the execution of a program. When you add tracing, you add code that sends signals to a trace listener, which then routes your output to a specific target. By default, your trace messages are displayed in Visual Studio's Output window. Tracing is often used to provide information about variable values or condition results to assist you in determining why your program behaves in a certain way. Tracing techniques can also be used to interrupt an application's execution in response to conditions that you designate.

Writing to the Windows Event Log

One of the more typical logging requirements is to write to the Windows event log. Diagnosis of the `System.Diagnostics.EventLog` class contains a number of static methods for writing to the Windows event log. Specifically, the `EventLog.WriteEntry` method has multiple overloads that can be used to log various combinations of data. To write to the Windows event log, you must supply at least three pieces of information:

- The **event log**: This is the name of the Windows event log to which you wish to write. Most of the time, you will write to the Application log.
- The **event source**: This identifies the source of the event and is often the name of your application. You associate the event source with an event log.
- The **message**: The text you want to output to the log. If necessary, you can additionally use the `WriteEntry` method to specify a category, an event ID, and an event severity.

Note: Writing to the Windows event log necessitates a high level of privilege. When you try to add an event source or write to the event log, and your application does not have necessary rights, it will produce a **SecurityException**.

In the following sample we write a message to the event log:

```
string eventLog = "Application";
string eventSource = "Logging Demo";
string eventMessage = "Hello from the Logging Demo application";
// Create the event source if it does not already exist.
If (!EventLog.SourceExists(eventSource))
EventLog.CreateEventSource(eventSource, eventLog);
// Log the message.
EventLog.WriteEntry(eventSource, eventMessage);
```

To learn more about the Windows event log, see: <https://aka.ms/3KxAgGm>.

Debugging and Tracing

The **System.Diagnostics** namespace has two classes, **Debug** and **Trace**, that you can use to monitor your application's execution. These two classes operate in a similar manner and employ many of the same methods. Debug statements are only active if your solution is built in Debug mode, whereas Trace statements are active in both Debug and Release mode builds.

The Debug and Trace classes offer methods for writing formatted strings to Visual Studio's Output window as well as other listeners you configure. You can also specify whether or not to write to the Output window when specific criteria are met. You can also modify the indentation of your trace messages. So, if you are sending the details of each object in an enumeration to the Output window, you may wish to indent them to distinguish them from other output.

There is also an Assert method in the Debug and Trace classes. You can use the Assert method to supply a condition (an logical expression that must evaluate to either true or false) as well as a format string. If the condition is false, the Assert method terminates the program and displays a dialogue box with the message you provide. This method is handy when you need to find the moment in a long-running program where an unexpected event occurs.

In the following sample you can see how the Debug class is used to log messages to the Output window of Visual Studio. It will also break execution in the event of a failure to parse the string:

```
int number;

Console.WriteLine("Please type a number between 1 and 10, and
then press Enter");

string userInput = Console.ReadLine();

Debug.Assert(int.TryParse(userInput, out number),
string.Format("Unable to parse {0} as integer", userInput);

Debug.WriteLine(The current value of userInput is: {0},
userInput);

Debug.WriteLine(The current value of number is: {0}, number);

Console.WriteLine("Press Enter to finish");

Console.ReadLine();
```

To find out more about tracing in Visual C#, see: <https://aka.ms/3E1Holw>.

Using Application Profiling

Using Application Profiling

- Create and run a *performance session*
- Analyze the *profiling report*
- Revise your code and repeat

Making your code work without errors is only part of your job as a developer. You must also verify that your code is efficient. You should examine how long your code takes to complete tasks and whether it consumes an excessive amount of CPU, disc, memory, or network resources. Visual Studio includes various profiling tools that can assist you in analysing the performance of your applications. Running a Visual Studio performance analysis is comprised of three high-level steps:

1. Create and launch a **performance session**. All of the analysis will occur during this performance session. From the Analyze menu in Visual Studio, launch the Performance Wizard. You can then design and run a performance session. Then you execute your application as normal while the performance session is active. While your application is running, you should try to use any functionality that you suspect is creating performance problems.
2. Study the **profiling report**. Once the execution of your program has finished, Visual Studio will display the profiling report. This comprises a variety of data that can provide insights into your application's performance. This will allow you to determine which functions used the most CPU time, view an execution timeline to see what your application was doing at any given time, and see automatically generated warnings and recommendations for improving your code.
3. Finally you can revise your code to address any concerns that you discovered, and then run the analysis again to start a new performance session and generate a fresh profile report. The Visual Studio Profiling Tools allow you to compare two reports in order to detect and quantify how your code's performance has improved.

In order to gather performance data the performance measurement tools use sampling. When you construct a performance session, you choose whether to sample CPU usage, .NET memory allocation, concurrency statistics for multi-threaded programs, or use instrumentation to capture detailed timing information for each function call. Most of the time, you'll want to start with CPU sampling (the default). CPU sampling uses statistical measurements to identify which functions consume the most amount of CPU time. This allows you to measure performance without requiring a lot of resources or slowing down your application.

For more detailed information about application profiling, see <https://aka.ms/3uvyrE7>.

Using Performance Counters

Using Performance Counters

- Create performance counters and categories in code or in Server Explorer
- Specify:
 - A name
 - Some help text
 - The base performance counter type
- Update custom performance counters in code
- View performance counters in Performance Monitor (perfmon.exe)

Performance counters are system tools that collect data on resource utilization. Viewing performance counters can provide information about what your program is doing, which parts of the code it is spending most time executing, and thus can help you troubleshoot performance issues. There are three types of performance counters::

- **Operating System Counters or Hardware counters.** This category contains counters that can be used to track processor usage, physical memory usage, disc usage, and network usage. The specifics of the counters available will vary depending on the hardware of the machine.
- **.NET framework counters.** Counters in the .NET framework and runtime can be used to measure a variety of application attributes. You can, for example, analyse the number of exceptions thrown, lock and thread use data, and garbage collector activity.
- **Custom counters.** You can add your own performance counters in code to investigate specific areas of your application's behaviour. You can, for example, establish a performance counter to count the number of calls to a specific method or the number of times a specific exception is thrown.

Performance counters are classified into groups. This makes it easier to discover the counters you want when gathering and reviewing performance statistics. The PhysicalDisk category, for example, often includes counters for the percentage of time spent reading and writing to disc, the amount of data read from and written to disc, and the queue lengths to read and write data to disc.

Note: From Visual Studio, you are able to view the performance counters on your PC. Expand **Servers**, then the name of your computer, and finally **PerformanceCounters** in Server Explorer.

Normally the data from performance counters is captured and viewed in Performance Monitor (perfmon.exe). The Windows operating system includes Performance Monitor, which allows you to capture real-time data from performance counters. You can browse the performance counter categories or add performance counters to a graphical representation within Performance Monitor. You can also make use of **data collector sets** to collect information for reporting or analysis.

Creating Custom Performance Counters in Code

You can manage performance counters in a various ways using the **PerformanceCounter** and **PerformanceCounterCategory** classes. These allow you to enumerate the performance counter categories for a particular machine, or over the performance counters within each category. In addition, you can check to see whether particular performance counter categories or performance counters are provisioned on the local computer. If needed, you can create your own performance counter categories or performance counters.

Normally, custom performance counter categories and counters are created as part of the installation, rather than during application execution. When you establish a custom performance counter on a specific computer, it stays there. It isn't necessary to generate it each time you start your application. To create a custom performance counter, use the **PerformanceCounterType** enumeration to specify a base counter type.

The following example explains how to establish a custom performance counter category programmatically. We create a new performance counter category called **FourthCoffeeOrders**. There are two performance counters in this category. The first performance counter keeps track of the overall number of coffee orders placed, while the second keeps track of the number of orders placed every second.

```
if (!PerformanceCounterCategory.Exists("FourthCoffeeOrders"))
{
    CounterCreationDataCollection counters = new
    CounterCreationDataCollection();

    CounterCreationData totalOrders = new CounterCreationData();
    totalOrders.CounterName = "# Orders";
    totalOrders.CounterHelp = "Total number of orders placed";
    totalOrders.CounterType = PerformanceCounterType.NumberOfItems32;
    counters.Add(totalOrders);

    CounterCreationData ordersPerSecond = new CounterCreationData();
    ordersPerSecond.CounterName = "# Orders/Sec";
```

```
ordersPerSecond.CounterHelp = "Number of orders placed per second";  
ordersPerSecond.CounterType = PerformanceCounterType.RateOfCountsPerSecond32;  
counters.Add(ordersPerSecond);  
PerformanceCounterCategory.Create("FourthCoffeeOrders", "A custom category for demonstration",  
PerformanceCounterCategoryType.SingleInstance, counters);  
}
```

In Visual Studio, you can also build performance counter categories and performance counters via the Server Explorer.

When you construct custom performance counters, your application must also initialize the counters. Performance counters provide methods for updating the counter value, such as the **Increment** and **Decrement** methods. The counter's processing of the value is determined by the base type you choose when you built the counter.

In the following example we will update some custom performance counters.

```
// Get a reference to the custom performance counters.  
  
PerformanceCounter counterOrders = new  
PerformanceCounter("FourthCoffeeOrders", "# Orders", false);  
  
PerformanceCounter counterOrdersPerSec = new  
PerformanceCounter("FourthCoffeeOrders", "# Orders/Sec", false);  
  
// Update the performance counter values at appropriate points in  
// your code.  
  
public void OrderCoffee()  
{  
    counterOrders.Increment();  
    counterOrdersPerSec.Increment();  
    // Coffee ordering logic goes here.  
}
```

Once you've built a custom performance counter category, you can navigate to it in Performance Monitor and pick specific performance counters. When you launch your

application, you can use Performance Monitor to observe real-time statistics from your custom performance counters.

Lab: Extending the Class Enrollment Application

Lab: Extending the Class Enrollment Application Functionality

Lab scenario

You have been asked to refactor the code that you wrote in the lab exercises for module 1 into separate methods to avoid the duplication of code in the Class Enrollment Application.

Also, you have been asked to write code that validates the student information that the user enters and to enable the updated student information to be written back to the database, handling any errors that may occur.

Objectives

Exercise 1:
Refactoring the
Enrollment Code

Exercise 2:
Validating Student
Information

Exercise 3:
Saving Changes to
the Class List

In the lab you will refactor the code that you wrote in the lab exercises for module 1. We want to break the Class Enrollment Application code into separate methods to avoid duplication.

In addition you will write code that validates the student information entered by the user, and enable the updated data to be saved to the database.

Module Review and Takeaways

Review Questions

Question: True or False: The code in a finally block will execute if an exception is thrown in the try block?

Answer: True, the code in the finally block will always execute.

Question: True or False: Trace statements will produce output in both Release builds?

Answer: True, **Trace** statements are present in both Debug and Release builds.

Module 3 – C# Structures, Collections and Events

Module Overview

In this module we are going to look at some of the fundamental constructs in the C# language. firstly we look at very simple structures that can be used to represent complex data items in your code where we need to handle collections of these structures will see how we can use the many collection classes built into .NET, and how we can add and retrieve items and iterate over these collections. We'll also look at events in C#, that might be triggered by user interaction or externally, and how you can subscribe to them to support interactivity in your applications.

Objectives

After completing this module, you'll be able to:

- Use **structs** to manage complex data items.
- Use **enums** to represent data items that can have multiple values.
- Create and use **collections** of data items.
- Subscribe to, and handle, **events**.

Lesson 1 – Structs

The .NET Framework comes with a number of built-in data types, including integers, floating point, Decimal, String, and Boolean. However, imagine you wanted to make an object that represented a point on a 2D canvas, or a person. Which one would you choose? You could employ built-in types to capture a point's coordinates, or a person's attributes, such as their name and address (strings) and age (integer).

But you still need a way of handling these complex types as a self-contained entity in order to conduct activities like drawing points on a graph or sorting lists of individuals. The purpose of complex data structures is to give us an object that encapsulates something in the real world in a way that's easy to manage in our code, and that we can re-use. In this lesson we're going

to start by looking at the simpler of two types of complex data structure in C#, which is the struct. Structs are passed by value, which means that if you have a large struct in your code and if you pass that between methods, the entire struct is passed on the stack. The stack is a limited resource, and this can generate ‘stack overflow’ errors if the stack becomes full. For this reason, the struct is only suitable for very small data objects. Some people advise against instance sizes for structs above 16 bytes. That only lets you store eight integers, and short ones at that!

Lesson Objectives

After completing this lesson, you’ll be able to:

- Create and initialize **structs**.
- Define constructors.
- Create field getters and setters.
- Use **structs** to manage complex data items.

Creating and Using Structs

Creating and Using Structs

- Use structs to create simple custom types:
 - Represent related data items as a single logical entity
 - Add fields, properties, methods, and events

```
public struct Coffee { ... }
```

- Use the **struct** keyword to create a struct

```
Coffee coffee1 = new Coffee();
```

- Use the **new** keyword to instantiate a struct

The main purpose of a struct is to give us a lightweight data object. It’s a concept that predates object-oriented programming, and is really just a way of bundling up a small number of different data types that can be stored together in memory. It’s a lightweight aggregate data type in the sense of the amount of processing and memory necessary to deal with structs, especially when compared to classes. If you have a simple data object with just a very few elements, then a struct is a good choice. In almost all other cases you’ll want to use a class, which we discuss in a later module.

You can declare a struct in C# by using the struct keyword. Structs contain member variables to store the data associated with the struct. They can also contain methods that provide functionality, such as constructors. Let's look at an example of declaring a struct that represents a point in an x-y display coordinate system. Since displays tend to be no more than a few thousand pixels in each direction we can safely use regular integers:

```
public struct Point
{
    public int x;
    public int y;
}
```

We've added the public keyword (or *access modifier*) to the member variables here because, by default, they'd be private, and you wouldn't be able to do anything with them. The access modifiers available are:

- **public:** available to any code
- **internal:** only accessible by code in the same assembly (the default value if you don't specify an access modifier)
- **private:** only accessible by code in the same struct

So far, the struct is not that complex because you're only storing two integer values, and you might feel that you could instead have used a two-dimensional array or maybe even a collection class. The problem with using an array is that accessing these members would not be intuitive as you'd need to know the array index for each value (0 and 1), as opposed to being able to refer to them by name (x and y). Also, a struct allows us to add some functionality by adding methods.

So far, we've just defined the Point structure. To use it we need to create an instance. You can use the **new** keyword when creating an instance of a struct:

```
Point point = new Point();
Console.WriteLine(point.x);
Console.WriteLine(point.y);
```

We've used point as our instance variable name, which is different from the name of the struct which is Point with an upper-case P. Notice that we can get at the members using the dot notation (because they're public).

Initializing Structs

Initializing Structs

- Use constructors to initialize a struct

```
public struct Coffee
{
    public Coffee(int strength, string bean, string origin)
    { ... }
}
```

- Provide arguments when you instantiate the struct

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

- Add multiple constructors with different combinations of parameters

If you run the code in the previous section you'll see that `point.x` and `point.y` both have the value zero. This is because we have implicitly called a *default constructor* that initializes the values to zero. We could assign values to them explicitly, e.g.

```
point.x = 56;
point.y = 75;
```

But a more convenient way is to add an explicit **constructor** that will be used for initializing the variables:

```
public struct Point
{
    public int x;
    public int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

```
}
```

```
}
```

A constructor is a special method that has the same name as the struct (or class) and that doesn't define a return value. We provide two parameters for the x and y values to be used to initialize the struct. Because we used the same names as the member variables, we have to specifically tell the compiler that we are assigning the values in the parameters to the member variables by use of the **this** keyword.

Caution: You'll generate an error if you try to create your own default constructor with no parameters in a struct. Also, you cannot initialize a member variable in the body of the struct.

We can now use the constructor to initialize the Point instance:

```
Point point = new Point(56, 75);  
Console.WriteLine(point.x);  
Console.WriteLine(point.y);
```

You can have multiple constructors if you need them, as long as they all have a unique signature, in other words they have different combinations of parameters.

Apart from constructors, you can add functionality by means of methods. For example, in the Point struct we could add a 'print' method that returns a nicely formatted string to show the coordinates. The following code includes the variables, constructor, and the print method:

```
public struct Point  
{  
    public int x;  
    public int y;  
    public Point(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
    public string print()  
    {
```

```
    return "(" + x + ", " + y + ")";
}
}
```

Now we can use the print method of the Point struct to make it display its own coordinates:

```
Point point = new Point(56, 75);
Console.WriteLine("Coordinates: " + point.print());
```

Creating Properties

Creating Properties

- Properties use get and set accessors to control access to private fields

```
private int strength;
public int Strength
{
    get { return strength; }
    set { strength = value; }
}
```

- Properties enable you to:
 - Control access to private fields
 - Change accessor implementations without affecting clients
 - Data-bind controls to property values

A **property** in C# allows you to get or set the value of an underlying private field. Although these are actually methods, the property operates like a public field to consumers of the struct or class. The property is implemented within your struct or class by employing accessors, which are a type of method. A property can have a get accessor, a set accessor, or both.

Here's an example of a struct that implements a property with both get and set methods that wrap an underlying private variable:

```
public struct Person
{
```

```
private string name;  
  
public string Name  
{  
    get { return name; }  
    set { name = value; }  
}  
}
```

Because we implemented very simple get and set methods, the effect to an external user of this struct is exactly the same as if we simply had a member variable **Name** and made it public. But by doing it this way we have the option of adding some behaviour later if needed. For example, we might want to ensure that whenever Name was updated, that the string length was greater than zero. We could implement that validation logic in the **set** method. We could also choose to omit the **set** method entirely, in which case the property would be read-only. A less common situation is to omit the **get** method if we needed a property we could write but not read.

Notice that the way the **get** method works is to return the underlying value, or whatever you want the property to represent. The corresponding **set** method assigns whatever is in the special local variable **value**, and again, you can add more logic here if you need it. Here's an example of a property that doesn't correspond to a single underlying value but instead computes a value, and has no **set** method:

```
public struct Person  
{  
    private string firstName;  
    private string secondName;  
  
    public string Name  
    {  
        get { return firstName + " " + secondName; }  
    }  
}
```

Using a property instead of a public variable gives you a lot of flexibility in case you later decide to change the internal implementation. In the example above, had we initially just had a public field value **Name** that stored first and second names together, we wouldn't have had

the option to change to storing them separately without changing all the code that uses it. A property also supports data-binding.

Since the first example of a simple getter and setter is so common, you might think that it's a lot of extra work to create a property 'just in case'. So there's a simplified syntax that's almost as simple as creating a public variable. We can simply write our **Name** property as:

```
public struct Person
{
    public string Name { get; set; }
}
```

With this syntax there is no underlying private variable we can use inside the struct, but we can just use the **Name** property itself in our code. If we want to specify a private field or do more complicated things then we're back to using the full syntax, but we can do all that later without changing any of the code that uses our struct. If we want our property to be read-only we can omit the **set**, and if want it to be writeable internally but read-only externally, we can use an access modifier:

```
public struct Person
{
    public string Name { get; private set; }
}
```

Note: you can also use other access modifiers depending on what you are trying to achieve. To find out more, see: <https://aka.gd/3xTq1Zp>.

Question: True or False: properties are always read/write?

Answer: False, properties can be made read only by not implementing a setter method.

Creating Indexers

Creating Indexers

- Use the `this` keyword to declare an indexer
- Use `get` and `set` accessors to provide access to the collection

```
public int this[int index]
{
    get { return this.beverages[index]; }
    set { this.beverages[index] = value; }
}
```

- Use the instance name to interact with the indexer

```
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
```

Sometimes you might have an array or other collection and put that inside a struct or, more likely, a class.

```
public struct Team
{
    public string[] departments = new string[] {
        "IT", "Accounts", "Sales", "Management" };
}
```

To get at the first department you could write:

```
Team team = new Team();
string firstDepartment = team.departments[0];
```

But you might not want the user of your `Team` struct to have to be concerned with your department array and how you implemented it. You could simply add an indexer like this:

```
public struct Team
{
    private string[] departments = new string[] {
        "IT", "Accounts", "Sales", "Management" };
```

```
public string this[int index]
{
    get { return this.departments[index]; }
    set { this.departments[index] = value; }
}
```

Now the departments array is hidden, and we can change the implementation later if needed. The consuming code would now be rewritten as:

```
Team team = new Team();
string firstDepartment = team[0];
```

In the above code the index is applied to the containing instance itself.

To learn more about structs in C#, see <https://aka.gd/3Lb1nYc>, and specifically for indexers, see: <https://aka.gd/3EGSrXT>.

Question: True or false: structs are an example of a reference type?

Answer: False, structs are value types, whereas classes are reference types.

Lesson 2 - Enums

The simplest data type is the Boolean, which can have only two values: *true* or *false*. But sometimes two isn't enough. You might have true, false, and 'don't know'. Or you might have a choice of red, green, or blue. An enumeration is a way of having a limited number of choices in a kind of static list. In C#, an enumeration data type is called **enum**, and consists of named constants which are known as the enumerator list.

Lesson Objectives

After completing this lesson, you'll be able to:

- Declare and setup **enums**.
- Make use of **enums**.

Creating and Using Enums

Creating and Using Enums

- Create variables with a fixed set of possible values

```
enum Day { Sunday, Monday, Tuesday, Wednesday, ... };
```

- Set instance to the member you want to use

```
Day favoriteDay = Day.Friday;
```

- Set enum variables by name or by value

```
Day day1 = Day.Friday;  
// is equivalent to  
Day day1 = (Day)4;
```

A good example use case for an **enum** is the days of the week, or months of the year. You can represent the days of the week by using numbers, such as Sunday is day 0, Monday is day 1, and so on. When you define the enumeration you set up a list of named constants that will be represented as numeric values, but in source code you can use the meaningful names. By default, **enums** start at 0 unless you say otherwise. This means Sunday would be day 0, Monday day 1, and so on.

```
enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday,  
Friday, Saturday };
```

You can also define an initializer when you create the **enum**. In the following example we use an enum for the abbreviated months of the year, but we don't really think of January as being "month zero". Instead, we can make Jan have the value 1, and then the rest will follow sequentially, so Feb will be 2, Mar will be 3 and so on:

```
enum Months { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,  
Oct, Nov, Dec };
```

Note that we've chosen to make the name of the **enum** start with an uppercase letter, which is a convention. If you execute the following code:

```
Console.WriteLine(Months.Mar);
```

...the value of Months.Mar will be converted to a string, and you'll get the output "Mar". If you want the underlying value you can cast it to an integer:

```
Console.WriteLine((int)Months.Mar);
```

...which will output "3". We can give as many values as we want, for example:

```
enum Response { Yes = 1, No=7, Unknown, DontCare=17 };
```

Any values that we don't supply explicitly will just follow the sequence, so in the example, Unknown will be given the value 8. Often we just use the **enum** in comparisons and assignments and we don't really care what the actual values are, as long as they're unique. So it's often easier to just let the compiler assign all the values. If we supply our own values, the compiler won't stop us giving two or more of the enumeration constants the same value, which might result in logic errors.

Suppose you wanted to have a lookup table for the number of days in a month (ignoring leap years for the sake of argument). You could use an **enum**:

```
enum DaysInMonth { Jan = 31, Feb=28, Mar=31, Apr=30, // etc. Not recommended!
```

Although this would work, it's not really what **enums** are designed for. A better solution would be to create a collection as described in the next lesson.

To learn more about structs in C#, see <https://aka.gd/3LbkMbx>.

Question: True or false: enumerations are always sequentially numbered?

Answer: False, you can assign specific values to them. If they don't have a value specified they will be automatically assigned sequential values.

Lesson 3 – Built-in Collections

In the first module we looked at simple variables for storing an integer, a string, and so on. We also looked at arrays of values which had a fixed length and were referenced by index. But sometimes you need to store a collection of related values where the number of items is not known in advance, or they need to be referenced by means of some kind of key. Maybe we have a look-up table of key-value pairs, for example the DaysInMonth lookup table we mentioned in the previous lesson. For these situations, where we are storing collections of

the same data type, we can use one of the many collection classes that are provided by the .NET libraries. These include lists, hash tables, stacks, queues and dictionaries.

Lesson Objectives

After completing this lesson, you'll be able to:

- Use stacks, lists and queues.
- Understand and choose from the range of collection classes.
- Understand LINQ (Language Integrated Query) syntax.

Choosing Collections

Choosing Collections

- *List* classes store linear collections of items
- *Dictionary* classes store collections of key/value pairs
- *Queue* classes store items in a first in, first out collection
- *Stack* classes store items in a last in, first out collection

Several traits are shared by all collection classes. You should be able to add items to, and remove items from, a collection in order to manage it. You need to be able to get specific items from the collection and count how many items there are in total. And you need to be able to iterate through the collection's items one at a time.

In C#, every collection class has methods and attributes that facilitate these fundamental functions. Beyond these actions, though, you may wish to handle collections in various ways depending on your application's specific requirements.

Collection classes in C# can be divided into the following high-level categories:

- List: used to hold linear collections of objects. Consider a list class to be a 1-D array that dynamically extends as items are added. For example, you could use a list to keep track of the beverages that are available in your coffee business.

- Dictionary: keeps a list of key/value pairs. Each object in the collection is made up of two parts: the key and the value. The value is the object you wish to store, and the key is used to index and locate the value. Most dictionary classes require that the key be unique, but duplicate values are not a problem. A dictionary could keep track of cooking recipes, in which case the key would be the recipe's unique name, and the value would provide the ingredients and directions for preparing the dish.
- Queue; a collection of things that are arranged in a first-in, first-out order. Items in the collection are retrieved in the order in which they were added. For example, at a coffee shop, you would use a queue class to ensure that customers get their drinks in order.
- Stack: a group of things that are ordered by last in, first out. The item you last added to the collection is the first one you retrieve. For instance, you could use a stack class to get the ten most recent comments saved in a blogging system.

When selecting a built-in collection class for a certain application, consider the following design points:

- Are you looking for a list, a dictionary, a stack, or a queue?
- Is there a requirement to sort the collection in order?
- How big do you think the collection will grow?
- Will you need to fetch items by index as well as key (if you're using a dictionary)?
- Is your collection entirely made up of strings or more complex objects?

Standard Collection Classes

Standard Collection Classes

Class	Description
ArrayList	<ul style="list-style-type: none">General-purpose list collectionLinear collection of objects
Hashtable	<ul style="list-style-type: none">General-purpose dictionary collectionStores key/value object pairs
Queue	<ul style="list-style-type: none">First in, first out collection
SortedList	<ul style="list-style-type: none">Dictionary collection sorted by keyRetrieve items by index as well as by key
Stack	<ul style="list-style-type: none">Last in, first out collection

The namespace **System.Collections** contains a variety of general-purpose collections such as lists, dictionaries, queues, and stacks. The most important of these are the **ArrayList**, **Hashtable**, **Queue**, **SortedList**, and **Stack**. All of these are collections of objects. Collections like **Hashtable** and **SortedList** also have a key that's stored as a string.

As an example, you can use a **Hashtable** to store the days in each month, which we've initialized with just January to March:

```
var DaysInMonth = new System.Collections.Hashtable() { { "Jan", 31 }, { "Feb", 28 }, { "Mar", 31 } };
Console.WriteLine(DaysInMonth["Feb"]);
```

The **System.Collections** namespace isn't included automatically, so you have to spell out the fully-qualified name or we could add using **System.Collections** at the beginning of the file. Notice how we can index on the key, which is a string representing the month. We've stored integers as the value, and we could add more in code:

```
DaysInMonth.Add("Apr", 30);
```

A major drawback of storing collections of .NET objects is that there's no inherent type-safety. We could just as easily have added some other type of variable:

```
DaysInMonth.Add("May", "thirty-one");
DaysInMonth.Add("June", 30.0);
```

That means that to make your code robust you need to be constantly checking the type of each object in your code, to verify that it's what you expect it to be. This makes it easy to have coding errors, and for this reason we don't use these 'standard' collection classes in new code. But where we are working with legacy code we need to be aware of these standard non-generic collection classes. In the next section we'll see the new, improved version of the .NET collection classes.

Generic Collection Classes

Generic Collection Classes

Legacy Class	Generic Class
ArrayList	List<T>
Hashtable	Dictionary< TKey, TValue >
Queue	Queue<T>
SortedList	SortedList< TKey, TValue >
Stack	Stack<T>

Because generic data types did not exist when .NET was first created, the collection types in the `System.Collections` are untyped. However, since then, generic data types have been added to the C# language, resulting in the addition of a new set of collections. These are in the `System.Collections.Generic` namespace. For any new development, it's strongly recommended to use these strongly typed generic collections.

When we create a strongly-typed collection using the generic collection classes, we specify the type we intend to store using the generic syntax. The definition of `List<T>` means that when we create a `List` we need to supply the type we want for the type placeholder `T`. For example a simple list of integers would be defined as follows:

```
var numbers = new List<int>() { 1, 2, 3, 4, 5, 6 };
```

The type is specified between angle brackets. We can now add more integers to the list, but not other types:

```
days.Add(7);
```

We can append another List of <int>:

```
days.AddRange(new List<int>{ 8, 9, 10, 11, 12 });
```

But we can't add another type...

```
days.Add(13.0); // won't compile
```

... unless we do a risky cast:

```
days.Add((int)13.0);
```

We can do the same thing for a Queue or a Stack, but the Dictionary and SortedList generic collections require us to provide two types – one for the key and one for the value. Here's how we might implement our DaysInMonth lookup table using a generic Dictionary class:

```
var DaysInMonth = new Dictionary<string, int>() { { "Jan", 31 },  
{ "Feb", 28 }, { "Mar", 31 } };
```

The **System.Collections.Generic** namespace is included by default. It's quite common to use a string as the key (the first type used in creating it) but the generic Dictionary class can be defined with other types for both the key and value, including complex types you create yourself. If we wanted to use the Point structure we defined earlier, there'd be nothing stopping us from creating a Dictionary<Point, Point>, if it made sense for our application.

Using List Collections

Using List Collections

- Add objects of any type

```
var p1 = new Point(2, 5);
var points = new List<Point>();
points.Add(p1);
```

- Retrieve items by index

```
Console.WriteLine(points[0].Y);
```

- Use a foreach loop to iterate over the collection

```
foreach (Point p in points)
{
    Console.WriteLine(p.X + ", " + p.Y);
}
```

Let's make a collection of Point objects. First, here's a quick definition of Point that we used earlier in the course when we were talking about structs:

```
public struct Point
{
    public int X;
    public int Y;
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

We can also create a few instances of Point. Suppose we were writing some kind of drawing application. We'd want to be able to store these points. Perhaps they make up a shape or a line, so we can create a generic **List** of type **Point** to store them:

```
var p1 = new Point(2, 5);
```

```
var p2 = new Point(3, 7);  
var points = new List<Point>();  
points.Add(p1);  
points.Add(p2);
```

Most collections support a count of the number of items:

```
Console.WriteLine(points.Count);
```

That's going to write the number 2 to the console. If we want to list the points themselves we need to do a bit more work. Collections implement the **Iterator** interface which means we can use them with the **foreach** statement. This works in a similar way to a **for** loop, except we don't have to know the length of the collection or use an indexer to get at the items. It just processes each of them in turn:

```
foreach (Point p in points)  
{  
    Console.WriteLine(p.X + ", " + p.Y);  
}
```

This will output the coordinates of each of our points. We could also implement a method in the Point struct to return a string, and that would be a better design.

Using Dictionary Collections

Using Dictionary Collections

- Specify both a key and a value when you add an item

```
var DaysInMonth = new Dictionary<string, int>()
DaysInMonth.Add("Apr", 30);
```

- Retrieve items by key

```
Console.WriteLine(DaysInMonth["Mar"]);
```

- Iterate over key collection or value collection

```
foreach (var key in DaysInMonth.Keys)
{
    Console.WriteLine(key + ": " + DaysInMonth[key]);
}
```

Earlier, we looked at how we might implement our DaysInMonth lookup table using a generic Dictionary class:

```
var DaysInMonth = new Dictionary<string, int>() { { "Jan", 31 },
{ "Feb", 28 }, { "Mar", 31 } };

DaysInMonth.Add("Apr", 30);
```

We can use the Count method to see how many items there are as with a List:

```
Console.WriteLine(DaysInMonth.Count);
```

And we retrieve values in the same way as for the non-generic Hashtable:

```
Console.WriteLine(DaysInMonth["Mar"]);
```

You might think you could iterate over the contents of the Dictionary in the same way as for the List but it's a little more complicated. Instead we need to iterate over the keys, and then use the key to retrieve each item:

```
foreach (var key in DaysInMonth.Keys)
{
    Console.WriteLine(key + ": " + DaysInMonth[key]);
}
```

That will produce a neat list of the months and the days in each month.

Querying a Collection

Querying a Collection

- Use LINQ expressions to query collections

```
var monthsByLength =
    from string key in DaysInMonth.Keys
    orderby DaysInMonth [key] ascending
    select key;
```

- Use extensions methods to retrieve specific items from results

```
var first = monthsByName.FirstOrDefault ();
var last = monthsByName.Last ();
```

LINQ is an abbreviation for Language Integrated Query, and is a set of functions and a special SQL-like query syntax included in .NET languages like C#. LINQ allows you to query data from a variety of data sources, including collections, databases, and XML documents, using a declarative query syntax. The following is the syntax for a basic LINQ query:

```
from <variable> in <source>
where <selector>
orderby <ordering>
select <variable>
```

LINQ is really some language syntax that makes a chain of method calls look more like SQL, or at least make it easier to code for somebody who is familiar with SQL. Using LINQ, we can take a collection and do things like filter and sort in the same way we would with a database table. To demonstrate, let's populate a full list of months of the year:

```
var DaysInMonth = new Dictionary<string, int>() {
    { "Jan", 31 },
    { "Feb", 28 },
    { "Mar", 31 },
```

```
{ "Apr", 30 },  
 { "May", 31 },  
 { "Jun", 30 },  
 { "Jul", 31 },  
 { "Aug", 31 },  
 { "Sep", 30 },  
 { "Oct", 31 },  
 { "Nov", 30 },  
 { "Dec", 31 }  
};
```

Now we can create a “new collection” that has the names of the months (the keys from the original collection) ordered by number of days and then iterate over it:

```
var monthsByLength =  
    from string key in DaysInMonth.Keys  
    orderby DaysInMonth[key] ascending  
    select key;  
  
Console.WriteLine("Months ordered by number of days:");  
foreach (var month in monthsByLength) Console.WriteLine(month);
```

In reality, **monthsByLength** is query that only gets evaluated when we use it, for example in the **foreach** statement above. The output will start with:

```
Months ordered by number of days:  
Feb  
Apr  
Jun  
...
```

The first item is Feb, which is the shortest month. Then the 30 day months, followed by the 31 day months. We can also use other methods like First(), Last(), Max(), or Min() to find the first and last values or the maximum and minimum of a collection of numeric values:

```
Console.WriteLine("First month: " + monthsByName.First());
```

This will output the value **Feb**. If we used DaysInMonth.First().Key we would of course get **Jan**. We can also order alphabetically by name:

```
var monthsByName =  
    from string key in DaysInMonth.Keys  
    orderby key ascending  
    select key;
```

...or just make a filtered collection of 30 day months by using a **where** clause:

```
var months30 =  
    from string key in DaysInMonth.Keys  
    where DaysInMonth[key] == 30  
    select key;
```

The collection **months30** will contain Apr, Jun, Sep, and Nov. In this case, rather than iterate over the keys, it might be easier to iterate over the whole collection, since we need both the key and the value:

```
var months30 =  
    from month in DaysInMonth  
    where month.value == 30  
    select month.Key;
```

The compiler converts the LINQ syntax into something equivalent to the following code. This is another way of querying a data source but using the LINQ query operators:

```
var months30 = DaysInMonth.Where(month => month.value ==  
30).Select(month => month.Key);
```

If you've used similar tools in other languages you might be more familiar with the **map** and **reduce** operators. In LINQ queries the **Select** performs a similar function to **map**, and you can think of **Where** as being equivalent to **filter**, and the LINQ equivalent of **reduce** is the **Aggregate** operator.

If you don't have a background in database development, and your data source is a collection rather than an actual database, the query operator syntax might be preferable. In the example above, we've used a lambda expression (`=>`), which is a functional programming idea that facilitates LINQ. Lambdas can be very expressive where the alternative would be to write an explicit anonymous function. Lambda expressions look a little strange when you first learn about them. But if we didn't have lambdas, we'd have to use a delegate and write this as something like:

```
// long-winded syntax avoiding Lambda expressions...

var months30 = DaysInMonth.Where(
    delegate(KeyValuePair<string, int> month)
{
    return month.value == 30;
}).Select(
    delegate(KeyValuePair<string, int> month)
{
    return month.Key;
});
```

That code's pretty ugly, and most people would think that learning the lambda syntax was a small price to pay for such a concise coding style.

Note: A LINQ expression's return type is `IEnumerable<T>`, where `IEnumerable` is a generic interface and `T` is the type of the collection's objects. This course will go into generic types in a later module. In the sample code above, we've used the `var` keyword to let the compiler figure out the correct type.

To learn more about LINQ and query expressions, see: <https://aka.gd/396RrAB>.

Lesson 4 - Events

Events enable objects to alert other objects when something unusual or of interest occurs. Controls on an application's user interface, for example, generate events when a user interacts with them, such as by clicking a button. You can write code that listens for these events and performs some action in response to them. Without events, your code would have to constantly poll these controls to detect any changes in state that necessitated action, which is very inefficient.

When something happens we say that the object **raises** an event, and we say that object is a **publisher**. The consumer of the event is called the **subscriber**, and when the subscriber object does something as a result of the event being raised, we say that it **handles** the event. In this module, we'll see how to raise events, as well as how to subscribe to events.

Lesson Objectives

After completing this lesson, you'll be able to:

- Define events and their delegates.
- Subscribe to events on another object.
- Raise events.
- Unsubscribe from events.

Delegates

Delegates

- Create a delegate
- Use a delegate in place of a method

```
delegate double Operator(double x, double y);  
  
double Add(double a, double b)  
{  
    return a + b;  
}  
Operator op = Add;  
var v = op(2.0, 3.0); // v = 5.0
```

When we call a method, we supply a list of zero or more arguments that correspond to the parameters of the method signature. These may be the built-in data types such as integers and floating-point numbers and strings, or they may be collections of objects or types that we have defined. But sometimes we need to pass in some code to be executed, rather than data, so that we can have a general method that has part of the logic defined later, when it gets called. In other words we need to be able to pass a function into the method. This is commonly used to have a ‘plug-in’ model for bits of functionality, or for callback methods to support event handling.

Different computer languages have different ways of doing this. In a functional programming language this is usually central to the way code is constructed. In a language like JavaScript a function is a “first-class object” that can be passed around in the same way as data. In C# we need to provide a wrapper that can be used to reference a method. These wrappers are called **delegates**. Once you’ve defined a delegate, you can pass it around like any other reference type. But unlike other types, a delegate can be invoked in the same way as a method, taking parameters and giving back a return value.

When you define a delegate, you specify a parameter list and return type just as you would a method. When you create an instance of a delegate you associate it with a particular method that matches the delegate signature.

Note: a delegate signature includes the return value as well as the parameter types. But remember that in the context of method overloading the method signature only includes the list of parameter types.

For example, suppose we are designing a collection class that needs a Sort method that sorts a list of objects alphabetically by a Name field. What if we wanted to sort in reverse

order? We could have another method called SortReverse. But what if we wanted to sort numerically on ID, or on some other property? The best solution is to be able to pass a Compare function so that the details of the comparison operator used for sorting can be defined externally, prior to calling the method. Similarly, if we were building a calculator, we might need a method that performed an operation on a collection of numbers, but we wanted to specify that without needing to know what the exact operation would be. First we need to declare the delegate:

```
delegate double operator(double x, double y);
```

We can use the delegate as a placeholder in the parameter list of an Operate method:

```
void Operate(Operator op, double[] array, double val)
{
    for(var i=0; i<array.Length; i++)
    {
        array[i] = op(array[i], val);
        Console.WriteLine("array[" + i + "] = " + array[i]);
    }
}
```

Next, we need to implement one or more methods that will be the implementations of the delegate, having the delegate signature:

```
double Noop(double a, double b)
{
    return a;
}

double Add(double a, double b)
{
    return a + b;
}

double Subtract(double a, double b)
{
```

```
return a - b;  
}
```

Now we can use the Operate method to perform different functions on the values in an array by passing different methods for the delegate:

```
var array = new double[4] { 0,13,6,24 };  
Operate(NoOp, array, 7.0);  
Console.WriteLine();  
Operate(Add, array, 7.0);  
Console.WriteLine();  
Operate(Subtract, array, 2.0);
```

It's as if we assigned the method to the delegate. You can also write something like:

```
Operator op = Add;
```

And then use **op** as if it was **Add**:

```
var v = op(2.0, 3.0); // v = 5.0
```

For more information about delegates in C#, see: <https://aka.ms/3wxfQJ5>.

Defining Events

Defining Events

- Create a delegate for the event

```
public delegate void OperationHandler(EventArgs args);
```

- Create the event and specify the delegate

```
public event OperationHandler OperationEvent;
```

Before event models, if you wanted to build any kind of asynchronous application code, such as a graphical user interface, you would have to create an endless loop checking to see if anything had happened. This was called an ‘event loop’, and it wasn’t a very efficient way of programming and quite prone to errors. Nowadays, we can use event-driven programming models instead.

When something of relevance occurs, an object can use events to alert other classes or objects. For example, when a user presses a button in a user interface, the object responsible for rendering the button would raise an event. You would then have some code that subscribes to the event, that would have the job of performing the intended action. This is referred to as ‘handling’ the event, and the code that does this is called an event handler.

Event handlers are simply methods that implement the **delegate** that corresponds to the event. You can therefore think of an **event** in C# as a special type of delegate, and that anything that implements such a delegate is an event handler.

To define an event, you first need to define the delegate:

```
delegate double OperationHandler(EventArgs args);
```

We could choose our own arguments, but a convention is to use an EventArgs object, along with any other arguments as required. Then you can define the event itself using the **event** keyword and declaring it as the delegate we just defined:

```
event OperationHandler OperationEvent;
```

Raising Events

Raising Events

- Check whether the event is null
- Raise the event by using method syntax

```
if(OperationEvent!= null)
{
    EventArgs args = new EventArgs();
    OperationEvent(args);
}
```

Once you've declared the delegate and defined the event, you can raise the event in code:

```
public struct CalculatorUI
{
    public delegate void OperationHandler(EventArgs args);
    public event OperationHandler OperationEvent;
    public void SomethingHappened()
    {
        // raise the event
        if(OperationEvent != null)
        {
            EventArgs args = new EventArgs();
            OperationEvent(args);
        }
    }
}
```

Raising the event is akin to calling the handler(s), if any. The null check prevents the event being raised if there are no subscribers, so currently, raising the event will have no effect. In order for something to happen when the event is raised, something needs to subscribe to it.

Subscribing to Events

Subscribing to Events

- Create a method that matches the delegate signature

```
public void MyHandler(EventArgs args)
{
    Console.WriteLine("Handled op!");
}
```

- Subscribe to the event

```
ui.OperationEvent = MyHandler
```

- Unsubscribe from the event

```
ui.OperationEvent = MyHandler;
```

Before you can subscribe to an event, you first need to create a method with a signature that corresponds to the event. This method is your event handler. You then need to subscribe to the event by using a special overloaded addition operator. Here's an example to subscribe to the **OperationEvent**:

```
public void MyHandler(EventArgs args)
{
    Console.WriteLine("Handled op!");
}

public void Subscribe(calculatorUI ui)
{
    ui.OperationEvent += MyHandler;
}
```

For this to work, the **MyHandler** method signature has to be an exact match for the handler delegate, including the return value.

You can also unsubscribe from the event using an overloaded subtraction operator:

```
public void UnSubscribe(CalculatorUI ui)
{
    ui.OperationEvent -= MyHandler;
}
```

You can have multiple handlers subscribed to the same event, and they will all get called when the event is raised.

The advantage of this, compared to simply calling the methods directly, is that there's a decoupling between the code that raises the event and the subscriber. This isn't obvious in the example code where we are writing both, but the most common use of event handlers is when you are subscribing to events that have been written by someone else, or more specifically, that are part of the framework you're working with. If you do any kind of UI programming then you'll be subscribing to events like button presses and other user interactions. You'll also use event handlers for asynchronous processes where there is an unpredictable amount of time between initiating an action and receiving a result, for example when making calls over a network to an external system.

To learn more about event handling, see: <https://aka.gd/3Kaoy3m>.

Lab: Building the Grades Application Prototype

Lab: Building the Grades Application Prototype

Lab scenario

The School of Fine Arts has decided that they want to extend their basic class enrollment application to enable teachers to record the grades that students in their class have achieved for each subject, and to allow students to view their own grades. This functionality necessitates implementing application log on functionality to authenticate the user and to determine whether the user is a teacher or a student.

You decide to start by developing parts of a prototype application to test proof of concept and to obtain client feedback before embarking on the final application. The prototype application will use basic WPF views rather than separate forms for the user interface. These views have already been designed and you must add the code to navigate among them.

You also decide to begin by storing the user and grade information in basic structs, and to use a dummy data source in the application to test your log on functionality.

Objectives

Exercise 1:

Adding Navigation Logic to the Grades Prototype Application

Exercise 2:

Creating Data Types to Store User and Grade Information

Exercise 3:

Displaying User and Grade Information

Module Review and Takeaways

Review Questions

Question: You create a Temperature property on a struct. You need the property to be readable from anywhere in code, but only writeable from within the struct. How should you define the property?

- A: public double Temperature { get; set; }
- B: public double Temperature { get; }
- C: private double Temperature { set; }
- D: public double Temperature { get; private set; }

Answer: D

Question: You create a glossary of terms, that contains a headword string and a longer definition string. You need to be able to efficiently look up a word in the glossary to retrieve its definition. Which is the best collection class to use?

- A: List<Dictionary>
- B: Hashtable
- C: SortedList<string>
- D: Dictionary<string, string>

Answer: D

Module 4 – C# Classes

Module Overview

Object-oriented programming (OOP) helps reduce complexity by offering a development paradigm that is modeled around real-world objects and how you manipulate them. These objects might be physical; a dog, a person, or a building; or more abstract concepts, like a message, or an index. OOP has simplified the way real-world problems are modelled in an application, making it easier to break the problem set into smaller chunks. Additionally, OOP makes it easier to save development time by reusing code, for example by allowing you to create classes that can be inherited and reused. But probably the biggest advantage is that it makes it easier to separate different ‘concerns’ in code, which avoids the situation where changing one thing in one part of an application affects everything else. As the scale of a software application increases, these advantages become more important.

With object-oriented programming, the first task is to understand the problem domain so that you can define the objects and how they relate to each other. Objects have properties (data) and behavior (methods). The exact way these objects are expressed depends on the programming language. There are a number of languages object-oriented programming languages, and most of them use the concept of **classes**. For example, C#, C++, and Java all use this “classical” model. JavaScript is an example of an object-oriented programming model that uses a fundamentally different model called prototypes, although classes have been added to the language more recently.

The purpose of a class is to act as a template for an object, and an object is an instance of a class. In this module we’ll see how to create classes and use them in code.

Objectives

After completing this module, you’ll be able to:

- Define classes.
- Create instances of classes.
- Use generics.

Lesson 1 – Creating Classes

We saw in an earlier module how structs are lightweight aggregate data structures that are one of several user-defined types in the C programming language that have been around since the 1970s. They're one of the characteristics that distinguish C as a structured programming language, but not as an object-oriented language. C#'s structs are very similar to the C struct. Classes, on the other hand, are a truly object-oriented programming feature. Classes allow for the creation of more complex objects, and facilitate code reuse and extension. In this lesson we'll look at classes and how to use them to create complex data structures with behaviours.

A class defines the data and behaviour needed to describe an object, which usually represents something in the real world. To use a class in your code, you must first construct an instance of that class, which is also known as an object (hence the name object-oriented). You can generate as many instances of a class as you need, and these instances or objects contain information that is unique to that instance. For example, if you develop a Person class, you can construct one instance for Bob and another for Tom, and Tom and Bob will each have their own data.

Lesson Objectives

After completing this lesson, you'll be able to:

- Create classes in code.
- Create instances from classes.
- Decide when to use structs or classes.
- An introduction to testing.

Creating Classes and Members

Creating Classes and Members

- Use the `class` keyword

```
public class Person
{
    // Methods, fields, properties, and events.
}
```

- Specify an access modifier:
 - `public`
 - `internal`
 - `private`
- Add methods, fields, properties, and events

A class is a programming concept that allows you to create your own custom types. When you create a class, you create a new type, which is effectively creating a template for the instance. The class defines the class members, the data and methods, that are present in all instances of the class. You do this by defining methods, fields, properties, and events within your class.

Let's create a `Person` class. We'll give the person some characteristics: name, height, and weight. We'll also add some behavior, such as walk, run, eat, and speak. We'll start with the name and numbers:

```
public class Person
{
    public float height;
    public float weight;
    public string name;
}
```

This is much the same as the struct example earlier, except that we used the keyword `class`. You're not going to see a lot of difference at the coding level until we get into the more advanced features of classes. You'll also notice that we prefixed the member names with an underscore (usually this is done just for private member variables but, for now, we made everything public). This is just a convention that some people use, and can avoid the need to use the `this` keyword when you use height, weight, and name as parameters, but it's really a matter of preference.

We can also start introducing some behavior by adding methods:

```
public class Person
```

```
{
```

```
    public string name;
```

```
    public float height;
```

```
    public float weight;
```

```
    public void Walk()
```

```
{
```

```
}
```

```
    public void Run()
```

```
{
```

```
}
```

```
    public void Eat()
```

```
{
```

```
}
```

```
    public void Speak()
```

```
{
```

```
}
```

```
}
```

This is just a simple example with everything using the **public** access modifier. In later modules we'll design classes a little more carefully. For example, the use of public variables to represent the person's characteristics is not especially good design, but it makes it easier to demonstrate. Instead we could have made these **internal** or **private** which would restrict access according to the table:

Access modifier	Effect
public	Accessible to code executing in any assembly that references the assembly containing the class.

private	Accessible to code within the same class. The private access modifier only makes sense for nested classes in a hierarchy.
internal	Accessible to code within the same assembly only. If no access modifier is specified, this is the default value.

Instantiating Classes

Instantiating Classes

- To instantiate a class, use the `new` keyword

```
Person aStudent = new Person();
```

- To infer the type of the new object, use the `var` keyword

```
var p2 = new Person();
```

- To call members on the instance, use the dot notation

```
p2.name = "Freddy";
p2.height = 2.0F;
p2.Walk();
```

We can now use the `new` keyword to create an instance of this class. This will create an object in code to represent a person, and we can then assign some values to the instance variables, and call some methods:

```
Person aStudent = new Person();
```

```
aStudent.height = 2.0F;
```

```
aStudent.weight = 140;
```

```
aStudent.Walk();
```

```
aStudent.Speak();
```

Annoyingly, we can't just assign 1.8 to `height`, because we declared `height` as type `float`. Remember that 1.8 is a literal constant, and it defaults to type `double`. The compiler can't

automatically convert a double to a float, so we must explicitly tell it that we want a float value by writing it as 1.8F. The code above instantiates the class and assigns values to some of the instance variables and calls a couple of the methods (that currently don't do anything). We didn't initialize everything, so any other member variables will get default values. For example the string **name** will be null. That might produce unexpected results or throw an exception depending on how your application works, so we probably want to make sure to initialize everything somehow.

Although simple, you might think it was rather redundant to have to define the type of **aStudent** as a **Person**. Surely the compiler can figure that out from the "new Person" that follows? To avoid this repetition, you can use the **var** keyword:

```
var aStudent = new Person();
```

This simply tells the compiler to work out the type for itself, and it's not the same as the weak typing in some languages like JavaScript, nor is it a dynamic type. In the example above, the variable **aStudent** is most definitely of type **Person**, and you'll get a compiler error if you later try to assign it some other type.

Once you have an instance, you can call its public methods by using a period:

```
aStudent.Walk();
```

You can also access any public variables in the same way:

```
aStudent.name = "Arthur";
```

```
Console.WriteLine(aStudent.name);
```

Using Constructors

Using Constructors

- Constructors are a type of method:
 - Share the name of the class
 - Called when you instantiate a class
- A default constructor accepts no arguments

```
public class Person
{
    public void Person()
    {
        // This is a default constructor.
    }
}
```

- Classes can include multiple constructors
- Use constructors to initialize member variables

When you create an instance of a class you use the **new** keyword, followed by the name of the class. After the class name there are parentheses in the same way as calling a method. This is because you're actually calling a method on the class called a "constructor". Specifically you're calling the default constructor, which means a constructor with no parameters. Explicitly initializing a class becomes quite messy, so we can use the default constructor to instantiate the object initialize some member variables in one go. If you have a class that doesn't require any logic in the default constructor you can omit it, and the compiler will generate one for you.

The following code shows an improved Person class with an explicit default constructor:

```
public class Person
{
    public string name;
    public float height;
    public float weight;
    public Person()
    {
        name = "anonymous";
        height = 1.8F;
```

```
weight = 140.0F;  
}  
  
public void Walk()  
{  
}  
  
public void Run()  
{  
}  
  
public void Eat()  
{  
}  
  
public void Speak()  
{  
}  
}
```

You can also create a constructor with one or more arguments. You can then use those arguments in the code within the constructor. This code usually involves initializing some variables in the class, so a common pattern is to define a constructor that takes a series of initial values as parameters.

Here's an example:

```
public Person(string name, float height, float weight)  
{  
    name = name;  
    height = height;  
    weight = weight;  
}
```

We can now reduce the code to instantiate a student person to a one-liner:

```
var aStudent = new Person("Bill", 1.8F, 140);
```

We can create as many constructors as we like, as long as the parameter list, or signature, of each constructor is unique in terms of parameter types.

Question: What happens if you have class with two constructors with the following signatures?

```
public Person(string name, float height, float weight)  
public Person(string fullname, float age, float shoesize)
```

Answer: You will get a compiler error, because the constructors signatures are the same.

Data Storage In .NET

Data Storage in .NET

- Value types
 - Stored on the stack
 - Passed by value (a copy)
- Reference types
 - Allocated on the heap
 - Passed by reference (memory address)



While modern high-level languages have largely freed us from having to worry about where and how application data is stored in the computer's memory, there's still one critical detail that we need to be aware of. The computer memory available to us is divided logically into areas known as the **stack** and the **heap**. A stack has a last-in-first-out (LIFO) structure and is used for program data, such as function parameters, local data and instruction pointers. The stack is a fairly valuable resource, because if we run out of stack space we get the dreaded **StackOverflowException**. The heap, on the other hand, is a block of memory that's used for application data only. It isn't really structured, and is a better choice for large amounts of data. The diagram in the slide shows the layout of the application's memory, where the stack expands from the top of the available address space, and the heap expands from the beginning of free memory after code has been loaded. This is greatly simplified, and in reality the memory available to the stack is more constrained than the diagram suggests.

The .NET runtime has two ways of saving variables in memory – the **stack** and the **heap**. These might sound like the same thing but the way they allocate memory is very different. The stack is used by the CPU to maintain the current execution context. The stack is used to store local in-scope variables by **value**, including those that you explicitly declare and those passed as parameters. These variables are ‘pushed’ onto the stack, which is implemented as an area in memory with a special CPU register called the stack pointer. When the code leaves its current **scope**, all associated data is ‘popped’ off the stack, and the stack pointer returns to its previous position, so that when a method returns to its caller the context is restored to its previous state. This also happens if you leave a code block (enclosed in braces) in which a local variable was declared.

The stack is a limited resource and should be used carefully. If we run out of stack space we’ll get a **StackOverflowException**. This typically happens if you call a method recursively, and for some reason the recursion continues indefinitely, but can also happen if too many objects are allocated to the stack. For this reason, we choose to store larger objects on the **heap**, which is another memory storage area which is less restrictive.

Memory is dynamically allocated from the heap as needed, and a **reference** is returned, which is an address in memory which can be used in code. When a reference variable goes out of scope, the memory is not automatically released from the heap, as it is with the stack. Instead, a process called the **garbage collector** is run periodically by the .NET runtime. This normally happens automatically and uses an algorithm to make decisions about when objects on the heap can be disposed of, and the memory made available for re-use. It does this by scanning the heap and disposing of any objects that no longer have references. The garbage collector can also defragment the memory space to free up contiguous memory in the same way you might defragment a disk drive. The heap doesn’t have a defined size limit and essentially depends on the amount of memory available, which may include virtual memory. This, along with the way memory is managed, means that the heap can store much larger amounts of data than the stack. But it is not an infinite resource and if it is exhausted then you’ll get an **OutOfMemoryException**.

Reference Types and Value Types

Reference Types and Value Types

- Value types
 - Contain data directly

```
int First = 100;  
int Second = First;
```

- In this case, **First** and **Second** are two distinct items in memory
- Reference types
 - Point to an object in memory

```
object First = new Object();  
object Second = First;
```

- In this case, **First** and **Second** point to the same item in memory

Classes and structs are both complex data types, and they both encapsulate data and behaviors to form an aggregate data type. On the face of it, it might seem that they are interchangeable. But there are a number of differences. The table shows some of them, the most important being the way that the data is stored. A struct, like intrinsic types, is a **value** type which means that it's stored on the stack and typically passed as a copy or by value. Classes, on the other hand, are **reference** types that are created on the heap, and are passed by reference.

Passing by reference means that a memory address is passed between the calling and called methods instead of the actual object and its data. This can have important consequences, particularly when you're passing data through function parameters. When you create an instance of a class, the variable you use for that instance holds a reference to the memory address of the object. However, when you create an instance of a struct, the variable you use for that instance holds the actual data contained in the struct itself on the stack.

The following table provides some comparisons between structs and classes:

	Struct/Intrinsic	Class
Data Type	Value	Reference
Memory Allocation	Stack	Heap
Inheritance	No	Yes

Instance size	Tens of bytes max	Any size
---------------	-------------------	----------

To know when to choose a struct or a class, remember that you should use a class if you want to model complex behaviors in your objects and if you think you might need to modify the data after the object is created. You should also use a class if the instance size is going to be more than a few tens of bytes (some would say anything over 16 bytes, which is equivalent to a couple of long integers). In practice, the vast majority of the time you will be using classes.

You should only use a struct when you know that the data contained in the struct is not going to be modified after the struct is created (immutability). Typically these will be very small short-lived objects. It's also important to note that you cannot use structs where you need to implement inheritance, which you'll learn more about in the next module. In practice you'll mainly be using classes rather than structs.

We can illustrate the consequences of using value types by the following example:

```
int value1 = 5;  
  
printValuePlusOne(value1);  
  
Console.WriteLine("Value in caller = " + value1);  
  
void printValuePlusOne(int val)  
{  
    val = val + 1;  
  
    Console.WriteLine("Value in function = " + val);  
}
```

Here's the result of running the code:

Value in function = 6

Value in caller = 5

You might think that **value1** would get incremented, but it doesn't because **val** is just a local copy of **value1**. Value types have advantages and disadvantages; using the stack is efficient from a memory perspective, but has a short lifespan. Generally, the intrinsic data types are very efficient, but complex structures can be problematic if they contain large amounts of data, because the entire structure is copied to the stack when it gets passed between methods. This is why structs are not advised for anything other than very small objects.

When you pass a reference type to a method as a parameter, you aren't passing the entire data structure like a value type; just a reference to the object's location in the memory. You

need to be aware of this, because it will behave a bit differently to passing a value, and it might not have the effect you expect. Consider this code, which is a bit like the earlier example except that this time we have created a class called `IntBox` that makes a ‘box’ for an integer value. We’re going to pass an instance into our revised `printValuePlusOne` method:

```
IntBox intbox = new IntBox();

intbox.value = 5;

printValuePlusOne(intbox);

Console.WriteLine("Value in caller = " + intbox.value);

void printValuePlusOne(IntBox ib)

{

    ib.value = ib.value + 1;

    Console.WriteLine("Value in function = " + ib.value);

}

class IntBox

{

    public int value;

}
```

Here’s the result of running the code:

Value in function = 6

Value in caller = 6

Unlike the previous example, changing the value in the function also changed the value in the caller. This is because `intbox` is passed by reference. Reference types are often large in terms of memory allocation, and they don’t go out of scope when a method ends. They remain in memory until all the references to that object are deleted, which can itself cause problems. It’s the job of the .NET garbage collector to free up that memory.

To learn more about the differences between classes and structs in C#,
see <https://aka.gd/3tMwj9G>.

Boxing and Unboxing

Boxing and Unboxing

- Converting a value type to a reference type:
 - Is referred to as 'boxing'
 - Is an implicit conversion
- Converting a reference type to a value type:
 - Is referred to as 'unboxing'

Sometimes you need to convert a value type to a reference type, or vice versa. You might store data as a particular value type, e.g. int or char, but later in the code you might assign it to a collection that expects reference types. This is quite a common requirement if you use the older style of weakly-type built-in collection classes that store objects, but the newer generic collection classes have made this unnecessary. But you will still occasionally need to do these conversions, and in .NET we use the terms **boxing** and **unboxing** when referring to type conversion between a value type and a reference type.

When you declare a variable in C#, you give it a data type. Even if you use the **var** keyword, which means we don't have to include the type in the declaration, it just means that the compiler deduces the type from the rest of the code. For example, we can write:

```
var intbox = new IntBox();
```

It's not asking a lot of the compiler to figure out what type we want **intbox** to be. If that data type is a value type, then you can assign values only of that type to the variable. But there are some automatic conversions, for example you can assign an int to a float, but not the other way round:

```
int i = 5;  
float f = i;  
  
// won't compile:  
  
int j = f;
```

The compiler won't let you assign a float to an int because there would be a loss of precision. Going in the other direction is okay, and this is called an implicit type conversion.

You can force the compiler to do a conversion that would potentially cause a loss of precision by using a cast. You might decide to do this if you happen to know that you won't lose precision, or if you don't care. The following shows how to cast a float to an int:

```
int j = (int)f;
```

If you declare **i** as an integer type and try to assign a string to it you'll get a compiler error. But you can do things like convert the string "3" to an integer by using the `int.Parse()` method:

```
string three = "3";
```

```
int i = int.Parse(three);
```

In addition to converting one value type to another, we can convert a value type to a reference by **boxing**, which means the compiler is performing an implicit conversion of a value type to `System.Object`, or to any interface type that is implemented by the value type. The .NET runtime allocates an instance of the object automatically and stores that instance on the heap. It will then copy the value from the value type into the newly created object.

There are a few situations when you might want to do this. For example, you might create an array of type `Object` because you don't know ahead of time what types will be stored. You might store `Person` objects, or ints, or even character strings. By declaring an array of `Objects`, every value type that you store in the array will be implicitly boxed into an object type and then allocated on the heap. The following code sample demonstrates this:

```
Point newPoint = new Point();
```

```
Object[] objArray = new Object[] {"one", 2, newPoint};
```

A new `Point` object is created and then an array of type `Object` is also created. In the array initializer, we store three values: a string, an integer, and an object. Because strings are already objects in .NET, the value will be stored as a `System.String` object and the `newPoint` will be stored as a generic object called `newPoint`. The integer value however must be boxed to store it in the array as an object. Whether `newPoint` is boxed depends on whether we defined `Point` as a class or a struct.

The performance penalty for this boxing isn't too bad, but there is a cost to perform the extra work to box the value. The more boxing that is required, the more your application performance will be affected.

Creating Static Classes and Members

Creating Static Classes and Members

- Use the static keyword to create a static class

```
public static class Conversions
{
    // Static members go here.
}
```

- Call members directly on the class name

```
double weightInKilos = 80;
double weightInPounds =
    Conversions.KilosToPounds(weightInKilos);
```

- Add static members to non-static classes

Not all classes are representations of objects in the problem domain. Sometimes you might want a class to act as a container for a set of methods that are related to each other. For example you might have some functionality for converting units of measure. In this case it doesn't really make sense to create an instance of that class, because there isn't any data that's related to a specific instance in this example; the conversions don't change. In this case we can create a **static** class, which means a class that will not be instantiate; in fact the compiler will not let you create an instance of a static class. The class is declared with the **static** keyword, and any members inside a static class must also be declared with the **static** keyword. The following example shows how to create a static class:

```
public static class Conversions
{
    public static double PoundsToKilos(double pounds)
    {
        // Convert argument from pounds to kilograms
        double kilos = pounds * 0.4536;
        return kilos;
    }
    public static double KilosToPounds(double kilos)
    {
```

```
// Convert argument from kilograms to pounds  
double pounds = kilos * 2.205;  
return pounds;  
}  
}
```

Since a static class has no instance, you use the name of the class itself, as follows:

```
Console.WriteLine("1lb is equivalent to " +  
Conversions.PoundsToKilos(1.0) + "kg");  
  
Console.WriteLine("1kg is equivalent to " +  
Conversions.KilosToPounds(1.0) + "lb");
```

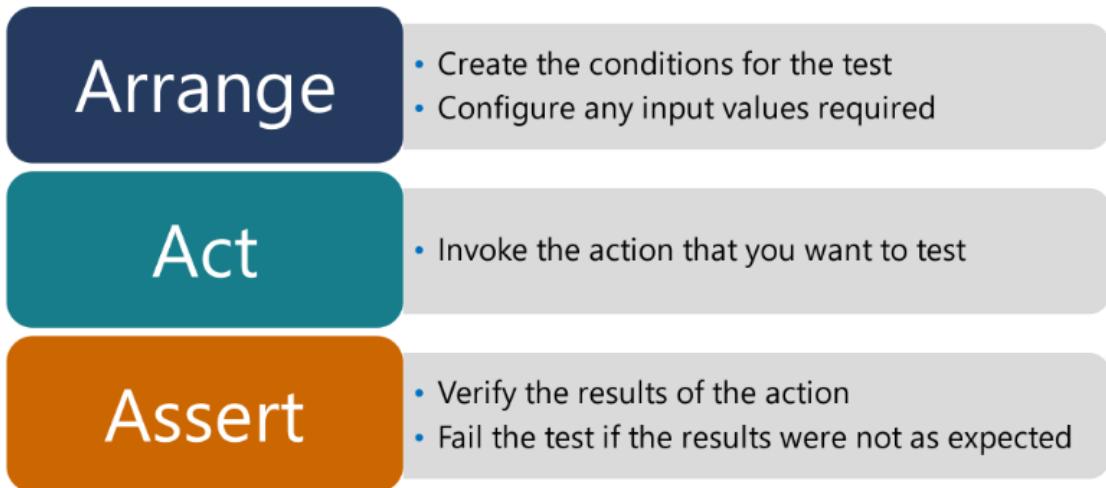
The above code produces the following output:

```
1lb is equivalent to 0.4536kg  
1kg is equivalent to 2.205lb
```

Static classes can have member variables, but because there's no instance, these must also be declared as static. You can also declare static members variables on non-static classes, and these will have the same value across all instances. If you declare a static method on a non-static class then you can only use static member variables inside it, otherwise you'll get a compiler error. Static methods are called the same way, using the class name, whether the class is static or not.

Testing Classes

Testing Classes



A good practice in modern development is to have comprehensive unit tests for every class you write. A unit test is one that tests a unit of functionality, which in object-oriented programming usually corresponds to a class or a particular method on a class. A unit test provides known inputs to the code under test, performs an action on the code under test (for example, invoking a method), and then verifies that the operation's outputs are as expected. The unit test, in this sense, becomes a specification of what your code should do. When you modify the implementation of a class or function, the unit test assures that your code always delivers specific outputs in response to specific inputs. This is important in software development because it guards against regression errors. The risk of regression errors can be a significant deterrent to improving and refactoring code, and unit tests are an effective tool to mitigate this risk.

Lets consider an example of the class discussed previously. We can create a unit test for the PoundsToKilos method of the Conversions class. In Visual Studio you can create a test project. The test methods in this project will be decorated with the **TestMethod** attribute (attributes will be explained later in the course) as follows:

```
[TestMethod]  
public void Test_PoundsToKilos()  
{  
    // Arrange.  
    var weight = 1.0;  
    var expected = 0.4536;
```

```
// Act.  
  
var result = Conversions.PoundsToKilos(weight);  
  
// Assert.  
  
// Fail the test if the actual age and the expected age are  
// different.  
  
Assert.IsTrue(result == expected), "weight not calculated  
correctly");  
  
}
```

Note that the unit test method consists of three stages, and this is known as the Arrange-Act-Assert pattern. This can be a useful way of organizing a test method. **Arrange** is where you set up the conditions for the test, **Act** is where you perform the action, usually by calling the method under test, and **Assert** is where you verify that the results of the method call are as expected. Our example class under test is static, but if it were non-static then we would create an instance in the Arrange stage.

If the Assert fails, then an exception is thrown and the test running engine will display the result as a failure. This usually is represented by red output, and if all the tests pass then there is usually some confirmation displayed in green.

Lesson 2 - Interfaces

You can think of an **interface** as a definition for a class without an actual implementation, or as a template for a class. An **interface** simply defines the signatures for methods, properties, events, and indexers, but it doesn't say how any of these members are implemented. When a class **implements** an interface, it provides the actual methods. By defining a class as an implementation of a particular interface, the class guarantees that it will be able to do what the interface says it can do. This means that elsewhere you can write code against that interface without knowing which particular class it will be at runtime, only that whatever class is used will implement those methods.

You will learn how to define and use interfaces during this lesson.

Lesson Objectives

After completing this lesson, you'll be able to:

- Define interfaces.
- Implement and use interfaces.

- Define multiple classes that implement an interface.
- Define classes that implement more than one interface.
- Look at the **IComparable** and **IComparer** interfaces.

Introducing Interfaces

Introducing Interfaces

- Interfaces define a set of characteristics and behaviors
 - Member signatures only
 - No implementation details
 - Cannot be instantiated
- Interfaces are implemented by classes or structs
 - Implementing class or struct must implement every member
 - Implementation details do not matter to consumers
 - Member signatures must match definitions in interface
- By implementing an interface, a class or struct guarantees that it will provide certain functionality

An **interface** acts as a definition for a class by defining methods, properties, events, and indexers, without an actual implementation. Classes can "implement" the interface and provide their own versions of the interface's members. So an interface can be thought of as a contract. By implementing a certain interface, a class guarantees to consumers that it will provide certain functionality through certain members, even though the actual implementation is left as part of the details of the class.

Suppose you were working on the **Conversions** class we looked at earlier. You have the **PoundsToKilos** and **KilosToPounds** methods, but then you realize you also need **FeetToMetres** and a **MetresToFeet** methods. Should they be part of the same class? Or should you have two classes? One possible design would be to create two classes called **WeightConverter** and **LengthConverter** and give each class an **ImperialToSI** method and an **SIToImperial** method:

```
public class WeightConverter
{
    public double ImperialToSI(double val)
    {
```

```
    return val * 0.4536;
}

public double SIToImperial(double val)
{
    return val * 2.205;
}

}

public class LengthConverter
{
    public double ImperialToSI(double val)
    {
        // Convert argument from feet to metres
        return val * 0.3048;
    }

    public double SIToImperial(double val)
    {
        return val * 3.28084;
    }
}
```

It would be nice to be able to write code that was passed some converter object without having to worry about what type of property was being converted. We could also use the compiler to make sure that any other converter classes were implemented in a consistent way.

Defining Interfaces

Defining Interfaces

- Use the `interface` keyword

```
public interface IConverter
{
    // Methods, properties, events, and indexers.
}
```

- Specify an access modifier:
 - `public`
 - `internal`
- Add interface members:
 - Methods, properties, events, and indexers
 - Signatures only, no implementation details

We can declare an `IConverter` interface:

```
public interface IConverter
{
    double ImperialToSI(double val);
    double SIToImperial(double val);
}
```

Interfaces can have the `public` or `internal` access modifiers.

Implementing Interfaces

Implementing Interfaces

- Add the name of the interface to the class declaration

```
public class WeightConverter: IConverter
```

- Implement all interface members

- Use the interface type and the derived class type interchangeably

```
WeightConverter conv1 = new WeightConverter();
IConverter conv2 = new WeightConverter();
```

The **conv2** variable will only expose members defined by the **IConverter** interface

We can now derive the WeightConverter from the IConverter interface:

```
public class WeightConverter : IConverter
{
    public double ImperialToSI(double val)
    {
        return val * 0.4536;
    }

    public double SIToImperial(double val)
    {
        return val * 2.205;
    }
}
```

We can do the same with the LengthConverter class:

```
public class LengthConverter: IConverter
```

```

{
public double ImperialToSI(double val)
{
// Convert argument from feet to metres
return val * 0.3048;
}

public double SIToImperial(double val)
{
return val * 3.28084;
}
}

```

Now we can create two objects of type `IConverter` and use them in code, passing them around as `IConverter` objects:

```

void OutputToSI(IConverter conv, double value, string imp, string si)
{
Console.WriteLine(value + imp + " is equivalent to " +
conv.ImperialToSI(1.0) + si + " in SI units");
}

IConverter convWeight = new WeightConverter();
IConverter convLength = new LengthConverter();
OutputToSI(convWeight, 1.0, "lb", "kg");
OutputToSI(convLength, 1.0, "ft", "m");

```

This is called polymorphism. The fact that we have an interface for the converters might not seem to be much of a benefit when writing simple code like this, but in a large project with multiple teams of developers and tens of thousands of lines of code, the extra rigour becomes increasingly important. For example, if someone needed to write a new converter for pressure, they might decide they only needed to convert in one direction:

```
public class PressureConverter : IConverter
```

```
{  
public double ImperialToSI(double val)  
{  
    // Convert argument from psi to Newtons  
    return val * 0.0068947572932;  
}  
}
```

This will result in a compiler error:

```
Error CS0535 'PressureConverter' does not implement interface  
member 'IConverter.SIToImperial(double)'
```

This is because the IConverter interface wasn't implemented properly. In practice, if that implementation was allowed, it would only be a question of time before somebody tried to use the non-existent SIToImperial method. The interface ensures that any class implementing it will have the expected methods.

Note: By convention, the names of interfaces always start with a capital I.

Exercise: how would you modify the WeightConverter and LengthConverter classes to include the strings for the SI and Imperial units names?

Implementing Multiple Interfaces

Implementing Multiple Interfaces

- Add the names of each interface to the class declaration

```
public class WeightConverter: IConverter, IUnitNamer
```

- Implement every member of every interface
- Use explicit implementation if two interfaces have a member with the same name

```
// This is an implicit implementation.  
public bool NameSI{ get; set; }
```

```
//These are explicit implementations.  
public bool IConverter.NameSI{ get; }  
public bool IUnitNamer.NameSI{ get; set; }
```

On occasion, you need to define a class that implements more than one interface. For example you might want to implement **IDisposable** so that the .NET runtime will manage disposal of your class and associated resources correctly. Or you might want to implement **IComparable** so that instances of your class can be sorted correctly when they are part of a collection.

Let's suppose that in addition to the **IConverter** interface, we define another interface to deal with naming. In our previous example we provided the names of the SI and Imperial units, but really this should be the responsibility of the converter class. But maybe we have other classes that also need to implement the same functionality (admittedly this is a bit of a contrived example). We might decide to implement another, separate, interface called **IUnitNamer**. Here's what the **IUnitNamer** might look like:

```
public interface IUnitNamer
{
    string NameSI { get; }
    string NameImp { get; }
}
```

We can now add the second interface to our **WeightConverter** class, the two interfaces separated by a comma:

```
public class WeightConverter : IConverter, IUnitNamer
{
    public string NameSI { get { return "kg"; } }
    public string NameImp { get { return "lb"; } }
    public double ImperialToSI(double val)
    {
        return val * 0.4536;
    }
    public double SIToImperial(double val)
    {
        return val * 2.205;
    }
}
```

```
}
```

If we add the `IUnitNamer` interface to our `LengthConverter` class we'll have to implement similar member properties for that, too. Then we can call them:

```
void OutputToSI(IConverter conv, double value, string imp, string si)
{
    Console.WriteLine(value + imp + " is equivalent to " +
        conv.ImperialToSI(1.0) + si + " in SI units");
}

var weight = new WeightConverter();
var length = new LengthConverter();

OutputToSI(weight, 1.0, weight.NameImp, weight.NameSI);
OutputToSI(length, 1.0, length.NameImp, length.NameSI);
```

This produces the following output:

```
1lb is equivalent to 0.4536kg in SI units
1ft is equivalent to 0.3048m in SI units
```

We don't need to use the `IUnitNamer` interface to get this functionality. It's more about managing the implementation of multiple classes in a large project.

It's actually possible to have the `NameSI` and `NameImp` properties, or other methods, on *both* interfaces. Suppose the `NameSI` property exists on both the `IUnitNamer` and the `IConverter` interfaces. In this case you can either choose to implement them implicitly:

```
public string NameSI { get { return "kg"; } }
```

or explicitly implement both, perhaps with different implementations, by disambiguating the name of the method or property:

```
public string IConverter.NameSI { get { return "kg"; } }
public string IUnitNamer.NameSI { get { return "kilogrammes"; } }
```

Which method gets called will depend on how the object is referenced. When an interface is implemented implicitly, its members can be used like any other public class member and

referred to without any special notation. But when an interface is implemented explicitly you can only access the members by casting the instance to the interface. If you don't, they'll stay hidden and the code that calls them won't compile.

Question: True or false: if you add an interface to a class, you can optionally provide an implementation for any of the methods defined?

Answer: False, your code won't compile if you don't implement all the interface's methods.

Lesson 3 – Understanding Generics in C#

In an earlier module we looked at the type-safe collections that are in the `System.Collections.Generic` namespace. These collections use **generics** to define general-purpose collection classes that can be instantiated in a type-safe way. This was an improvement on the earlier collection classes in the `System.Collections` namespace (which are still available if needed), which are collections of Objects, where the onus is on the developer to make sure the objects were of the right type. Generics is a feature of .NET that allows the use of parameters that represent types. This makes it possible to write generic code in which the specification of the types is deferred until the class is actually instantiated.

In this module we'll look in more detail into how **generics** work in C#.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand generics in C#.
- Use general purpose classes that support multiple types.
- Constrain parameter types.
- Create generic collections that implement `IEnumerable`.

Introducing Generics

Introducing Generics

- Create classes and interfaces that include a type parameter

```
public class CustomList<T>
{
    public T this[int index] { get; set; }
    public void Add(T item) { ... }
    public void Remove(T item) { ... }
}
```

- Specify the type argument when you instantiate the class

```
CustomList<Person> people =
    new CustomList<Person>();
```

Let's suppose you design a integer calculator class that has a Print method and an Operate method that takes a delegate to do the actual operation:

```
public class calculatorInt
{
    public delegate int Operator(int x, int y);
    int accum;
    public calculatorInt(int init) // constructor
    {
        accum = init;
    }
    public int operate(Operator op, int y)
    {
        return accum = op(accum, y);
    }
    public void Print()
    {
```

```
Console.WriteLine(accum);
}
}
```

This seems reasonable. We'll define a couple of operator functions Add and Sub. Now we can use it to perform a simple calculation and print the result:

```
int Add(int x, int y) => x + y;
int Sub(int x, int y) => x - y;
var calc = new CalculatorInt(0);
calc.Operate(Add, 4);
calc.Operate(Add, 7);
calc.Operate(Sub, 2);
calc.Print();
```

This produces the expected output: 9. But what if we then wanted to have the Calculator work with **double** values. We'd have to write another class called CalculatorDouble or something. CalculatorDouble would look and behave exactly the same as CalculatorInt, except with **doubles** wherever we had **ints** before:

```
public class calculatorDouble
{
    public delegate double Operator(double x, double y);
    double accum;
    public calculatorDouble(double init)
    {
        accum = init;
    }
    public double Operate(Operator op, double y)
    {
        return accum = op(accum, y);
    }
}
```

```
public void Print()
{
    Console.WriteLine(accum);
}
```

The same would be true of CalculatorFloat and CalculatorDecimal, and so on. But we'd have a lot of near identical code, and if we wanted to add a new method to our calculators, like a UnaryOperate() function for example, or we found a bug, we'd have to be careful to make the same changes multiple times.

With generics, we can just write one class:

```
public class calculator<T>
{
    public delegate T Operator(T x, T y);

    T accum;

    public calculator(T init)
    {
        accum = init;
    }

    public T Operate(Operator op, T y)
    {
        return accum = op(accum, y);
    }

    public void Print()
    {
        Console.WriteLine(accum);
    }
}
```

We create a **generic** type declaration by putting one or more type parameters inside angle brackets after the name of the type. The type parameter is a placeholder for the specific type that will be specified when an instance is created. A generic type acts as a template for multiple different types, which differ by the choice of type arguments. The type placeholders can be replaced by built-in types, types defined in libraries we've imported, or our own custom types. The correct word for a type that's generated in this way is a “constructed type”, and you just use them like any other type.

Let's use our new generic Calculator class to do the calculation with integers:

```
int Add(int x, int y) => x + y;  
int Sub(int x, int y) => x - y;  
var calc = new Calculator<int>(0);  
calc.Operate(Add, 4);  
calc.Operate(Add, 7);  
calc.Operate(Sub, 2);  
calc.Print();
```

If we want to create a double-precision floating point version, we can just replace the type parameter, T, with the desired type, remembering to also update the Add and Sub functions:

```
double Add(double x, double y) => x + y;  
double Sub(double x, double y) => x - y;  
var calc = new Calculator<double>(0);  
calc.Operate(Add, 4);  
calc.Operate(Add, 7);  
calc.Operate(Sub, 2);  
calc.Print();
```

You might wonder why we didn't implement the Add and Sub functions inside our generic class. The problem is that the arithmetic operators '+' and '-' only work with certain numeric types, and the compiler doesn't know what types we might choose, so it won't compile.

We're not restricted to built-in types like int, float, and double. As long as we can devise an Operator function, we can create a calculator for anything. Let's use the complex number class in System.Numerics, which already defines some arithmetic functions:

```
using System.Numerics;

var calc = new Calculator<Complex>(new Complex(0,1));
calc.Operate(Complex.Add, 4);
calc.Operate(Complex.Add, 7);
calc.Operate(Complex.Subtract, 2);
calc.Print();
```

An implicit conversion takes the integer operands and converts them to the real part of the complex numbers. This results in the output: (9, 1) which is a complex number with 9 for the real part and 1 for the imaginary part. We could also create our own 3D vector type:

```
public struct Vector3D
{
    public double x;
    public double y;
    public double z;
    public Vector3D(double x, double y, double z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public override string ToString()
    {
        return "(" + x + ", " + y + ", " + z + ")";
    }
}
```

We call it in the usual way:

```

vector3D Add(Vector3D a, Vector3D b) => new Vector3D(a.x + b.x,
a.y + b.y, a.z + b.z);

Vector3D Sub(Vector3D a, Vector3D b) => new Vector3D(a.x - b.x,
a.y - b.y, a.z - b.z);

var calc = new Calculator<Vector3D>(new Vector3D(0,4,2));
calc.Operate(Add, new Vector3D(4, 4, 2));
calc.Operate(Add, new Vector3D(7, 1, 1));
calc.Operate(Sub, new Vector3D(2, 8, 6));
calc.Print();

```

This gives the result of the operations on the vectors: (9, 1, -1). We can even go one further and make the Vector3D struct itself generic:

```

public struct Vector3D<T>
{
    public T x;
    public T y;
    public T z;

    public Vector3D(T x, T y, T z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public override string ToString()
    {
        return "(" + x + ", " + y + ", " + z + ")";
    }
}

```

Now we can use any numeric type to define the vectors, e.g. float:

```
Vector3D<float> Add(Vector3D<float> a, Vector3D<float> b) => new  
Vector3D<float>(a.x + b.x, a.y + b.y, a.z + b.z);  
  
Vector3D<float> Sub(Vector3D<float> a, Vector3D<float> b) => new  
Vector3D<float>(a.x - b.x, a.y - b.y, a.z - b.z);  
  
var calc = new Calculator<Vector3D<float>>(new  
Vector3D<float>(0,4,2));  
  
calc.Operate(Add, new Vector3D<float>(4, 4, 2));  
calc.Operate(Add, new Vector3D<float>(7, 1, 1));  
calc.Operate(Sub, new Vector3D<float>(2, 8, 6));  
calc.Print();
```

Notice that we now create a new Calculator object from a constructed type, where the type we pass it is itself a constructed type.

Advantages of Generics

Advantages of Generics

Generic types offer three advantages over non-generic types:

- Type safety
- No casting
- No boxing and unboxing

Generic classes give you the advantages of type safety in classes that contain other classes. The most common situation where you need this is in collection classes. A class like a list, or a dictionary, or a sorted list, could be implemented without using generics.

The only practical alternative to generics is to implement a collection class that contains objects. This is the way the collections in the System.Collections namespace work that were the original .NET collection classes, and still available. If you use one of these classes you store your items as objects, and then convert them back to the correct type when you retrieve them. There's nothing the compiler can do to stop you storing one type of item and

then casting it as something else when you retrieve it, resulting in a runtime error. There's also a lot of boxing and unboxing if you're using value types.

Generics allows us to use a strongly typed collection class, and for that reason the generic collections are now recommended in preference to the older non-generic collection classes. For example, we can have a simple list of integers by using the `List<T>` collection in the `System.Collections.Generic` namespace:

```
var intlist = new List<int>();
```

You can also pass it an initializer:

```
var intlist = new List<int>() { 1,2,3,4,5 };
```

We can also have a list of 3D vectors using the class we defined earlier:

```
var veclist = new List<Vector3D>();
```

If you're doing a lot of inserting of values and don't like the performance profile of `List<T>` you can try switching to a doubly-linked list:

```
var veclist = new LinkedList<Vector3D>();
```

If you have a collection of key/value pairs you can use one of the dictionary classes, the most commonly used being `Dictionary< TKey, TValue >`. Note that a Dictionary has two type parameters, one for the key and one for the item's value. We could store a collection of named 3D vectors as follows:

```
var vecset = new Dictionary<string, Vector3D>();
vecset.Add("ship", new Vector3D(4, 4, 2));
vecset.Add("sub", new Vector3D(7, 1, 1));
vecset.Add("torpedo", new Vector3D(2, 8, 6));
Console.WriteLine(vecset["torpedo"]);
```

In this example, each vector is stored with a text name or key. The object can then be retrieved by the key using a syntax similar to an array index. The Dictionary class is designed to retrieve items efficiently by key value by making use of hash tables and other techniques.

Occasionally you might want a heterogeneous list where the items in the collection can be of different types. In that case you could use one of the `System.Collections` classes, or use a generic class and use the `object` type:

```
var veclist = new List<object>();
veclist.Add(2);
veclist.Add(3.4);
veclist.Add(new Vector3D(2, 8, 6));
Console.WriteLine(veclist[2]);
```

Note: in this sample the Add() method is a member of the List<T> class that appends an item to the end of the list.

In the example we were able to add a number of items of different types, without getting any errors. The downside is that we have sacrificed all the advantages of generic collections such as type-safety.

Implementing the **IComparable** Interface

Implementing the **IComparable** Interface

- If you want instances of your class to be sortable in collections, implement the **IComparable** interface

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

- The **List.Sort** method calls the **IComparable.CompareTo** method on collection members to sort items in a collection

There are many different collection classes in the .NET Framework that let you sort the items in a collection, such as the **List** class. There is a method called **Sort** in these classes. When this method is called on a **List**, the collection's items are put in order. How the ordering is carried out depends on the type that the **List** contains. For example, if we have a **List<int>**, the items will be sorted numerically in ascending order, and if it's a **List<string>** then they will be sorted alphabetically.

We can also define our own sort order. For this we need to implement the **IComparable** interface and then define the **CompareTo** method. This method gets called by the sort algorithm and is passed another object, as an argument, to be compared. The

`CompareTo` method returns an `int` value. If the current value should be positioned, in the sort order before the value passed as an argument, then the `CompareTo` method should return a value of 1 (or some positive value). If it should be placed after, the return value is -1, and if the two values should be regarded as equal it can return 0.

Let's create a class called **Score** that will hold an integer number (called `score`). Scores should be ranked in order except that we have a special "magic value" of 42 that trumps all the others. So when we rank the scores in ascending order, any score of 42 should come last, regardless of the other numerical values. To do this, we define a class **Score** that implements the `IComparable<T>` interface:

```
public class Score : IComparable<Score>
{
    const int magic_value = 42;
    public int score;
    public int CompareTo(Score other)
    {
        if (score == magic_value) return 1;
        if (other.score == magic_value) return -1;
        return score.CompareTo(other.score);
    }
    public Score(int i)
    {
        score = i;
    }
    public override string ToString()
    {
        return score.ToString();
    }
}
```

Because **Score** is implementing **IComparable**, we must define a **CompareTo** method that takes an argument of type `Score` that we will be comparing with. We then implement some

logic in CompareTo that any value of score of 42 takes precedence in the sort, and similarly if the ‘other’ value is 42 then it takes precedence.

Note: If both values in the comparison are 42, in this particular instance, it doesn’t really make any difference.

Let’s create a List<Score> and populate it with some values. Then we’ll do a Sort and see the results:

```
var scores = new List<Score>();  
scores.Add(new Score(37));  
scores.Add(new Score(21));  
scores.Add(new Score(42));  
scores.Add(new Score(11));  
scores.Add(new Score(77));  
scores.Add(new Score(23));  
scores.Add(new Score(56));  
scores.Sort();  
Console.WriteLine(string.Join(", ", scores));
```

Running this code results in the following output:

```
11, 21, 23, 37, 56, 77, 42
```

The Score values are all placed in order, except that the Score of 42 is placed at the end of the sort, above the numerically higher values of 56 and 77.

Implementing the IComparer Interface

Implementing the **IComparer** Interface

- To sort collections by custom criteria, implement the **Compare** interface

```
public interface IComparer<T>
{
    int Compare(T x, T y);
}
```

- To use an **IComparer** implementation to sort a **List**, pass an **IComparer** instance to the **List.Sort** method

```
var list = new List<int>();
// Add some items to the collection.
list.Sort(new CustomComparer());
```

Sometimes you have an array of some type that you don't control, or that is one of the built-in types, but you still need to do some kind of custom sort. A very common example is if you have a collection of string values. A simple alphanumeric sort might not suffice. In these situations you can supply a custom comparer function by implementing the **IComparer** interface.

To illustrate, we'll use the same logic as in the previous example, but this time with a simple collection of **int** values. We can start with a much simpler piece of code, because we just have a list of integers:

```
var scores = new List<int>() { 37, 21, 42, 11, 77, 23, 56 };
scores.Sort();
Console.WriteLine(string.Join(", ", scores));
```

We can run this as it stands, and of course the output is in numerical order:

```
11, 21, 23, 37, 42, 56, 77
```

To make 42 the 'magic number' we can simply define our own **IComparer** as follows:

```
public class MagicComparer : IComparer<int>
{
    int magic_value;
```

```
public int Compare(int x, int y)
{
    if (x == magic_value) return 1;
    if (y == magic_value) return -1;
    return x.CompareTo(y);
}

public MagicComparer(int m)
{
    magic_value = m;
}
```

We have implemented a Compare function that compares two values. We could have had a hard-coded magic number, but instead we have made our MagicComparer a bit more general by adding a constructor that takes an integer value that is, of course, the desired magic number.

Now we can create an instance of our MagicComparer, and pass it to an overload of the Sort method that will use it to replace the default sort logic:

```
var scores = new List<int>() { 37, 21, 42, 11, 77, 23, 56 };
scores.Sort(new MagicComparer(42));
Console.WriteLine(string.Join(", ", scores));
```

This results in the desired output as before:

```
11, 21, 23, 37, 56, 77, 42
```

Constraining Generic Types

Constraining Generics

You can constrain type parameters in six ways:

- where T : <name of interface>
- where T : <name of base class>
- where T : U
- where T : new()
- where T : struct
- where T : class

Sometimes, when designing a generic type, you need to limit the types that can be supplied as parameters. For example, you might make use of the methods of a particular interface in your class. In that case it would only make sense to allow the use of types that implemented that interface. You can also limit the types to ones that are derived from a particular base class.

You apply these constraints by using the **where** keyword, followed by one or more constraints. Let's suppose we have implemented a sorted collection class. If you try to call the Sort() method, and the items in the collection don't support the CompareTo() method, then you'll get a System.InvalidOperationException at runtime. You might decide that you'll only allow types that implement IComparable, as follows:

```
public class CustomList<T> where T : IComparable<T>
{
    // implementation...
}
```

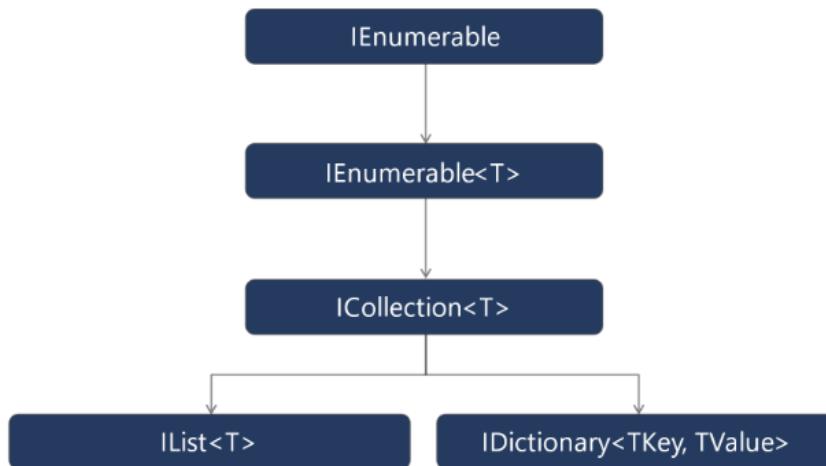
The types of constraint you can apply are shown in the following table:

Syntax	Constraint on T
where T : IMyInterface	Must implement the IMyInterface interface.
where T : MyBaseClass	Must derive from, or be, the MyBaseClass class.
where T : U	Must derive from, or be, the supplied type given in the type argument U .

where T : new()	Must have a public default constructor.
where T : struct	Must be a value type.
where T : class	Must be a reference type

Using Collection Interfaces?

Using Collection Interfaces



Most of the time you can use the pre-built collection classes in the `System.Collections.Generic` namespace. If you need to create your own custom collection class, then for consistency you'll want to follow the same pattern and implement certain interfaces.

If you want to be able to use the **foreach** loop construct, you'll need to implement the `IEnumerable<T>` interface. This implements a method `GetEnumerator()` that needs to return an `IEnumerator<T>` object. The compiler will look for this method when it encounters a `foreach` loop, and will produce a compiler error if it's not there.

All the collection classes implement the `ICollection<T>` interface. In fact `ICollection` inherits from `IEnumerable` – you can think of an `ICollection` as a specialized type of `IEnumerable`. `ICollection` defines the `Add`, `Clear`, `Contains`, `CopyTo` and `Remove` methods that provide the base functionality of collection classes. It also defines the `Count` and `IsReadOnly` properties.

There is also an `IList<T>` interface that the list classes use, which is a more specialized type of `ICollection`. `IList` inherits from `ICollection`. `IList` adds the `Insert` and `RemoveAt` methods and the `IndexOf` property.

`IDictionary< TKey, TValue >` is an interface that itself inherits from `ICollection`, and includes the `ContainsKey` and `TryGetValue` methods and the `Keys` and `Values` properties and the `Item` indexer.

For more information about the generic collections namespace, see:
<https://aka.ms/3Nm2ggY>.

Creating Enumerable Collections

Creating Enumerable Collections

- Implement `IEnumerable< T >` to support enumeration (`foreach`)
- Implement the `GetEnumerator` method by either:
 - Creating an `IEnumerator< T >` implementation
 - Using an iterator
- Use the `yield return` statement to implement an iterator

A very common programming task is to perform some operation over every item in a collection. The easiest way of doing this is to use a `foreach` loop. This returns each item in the collection without you needing to maintain some kind of counter or indexer. This is possible because all collections implement `IEnumerable< T >` which means that they have to implement a `GetEnumerator` method. This returns a special enumerator object that implements the `IEnumerator< T >` interface that includes the `MoveNext` and `Reset` methods, which as the names imply, advance the enumerator to the next object or set it back to the first item.

You can think of the enumerator as a cursor or pointer that is used to work its way through the collection. The order might be arbitrary, or if the collection has been sorted it will be in the order as defined by the sort function. When you reach the end of the collection then the `MoveNext` method returns false, and the `foreach` loop will exit. The `IEnumerator` interface also defines a `Current` property which is the current item as the loop is executed.

These interfaces are a little tedious to implement, but in practice you rarely need to implement these methods because you can use the existing ones provided in the framework.

The main reason for specifying these interfaces is so that you can take advantage of the `foreach` keyword. When you want to iterate over, say, a `List` collection you would have some code like the following:

```
var list = new List<int>() { 2, 6, 8, 4, 8, 4, 8, 7};
```

```
foreach(int n in list)  
Console.WriteLine(n);
```

Behind the scenes, the compiler is actually producing code that corresponds to something like the following:

```
var list = new List<int>() { 2,6,8,4,8,4,8,7};  
var enumerator = list.GetEnumerator();  
while (enumerator.MoveNext())  
Console.WriteLine(enumerator.Current);
```

Both code samples produce the same result. The **foreach** syntax is really just there to make the code easier to read.

Lab: Adding Data Validation and Type-Safety to the Application

Lab: Adding Data Validation and TypeSafety to the Application

Lab scenario

Now that the user interface navigation features are working, you decide to replace the simple structs with classes to make your application more efficient and straightforward.

You have also been asked to include validation logic in the application to ensure that when a user adds grades to a student, that the data is valid before it is written to the database. You decide to create a unit test project that will perform tests against the required validation for different grade scenarios.

Teachers who have seen the application have expressed concern that the students in their classes are displayed in a random order. You decide to use the `IComparable` interface to enable them to be displayed in alphabetical order.

Finally, you have been asked to add functionality to the application to enable teachers to add students to and remove students from a class, and to add student grades to the database.

Objectives

Exercise 1:
Implementing the Teacher, Student, and Grade Structs as Classes

Exercise 2:
Adding Data Validation to the Grade Class

Exercise 3:
Displaying Students in Name Order

Exercise 4:
Enabling Teachers to Modify Class and Grade Data

Module Review and Takeaways

Review Questions

Question: Which of the following types are reference types (select all that apply)?

- A: decimal
- B: byte
- C: object
- D: double

Answer: A, B and D

Question: You create a class MyQueue that will contain a collection of items of the same (as yet unknown) type. The class must enable developers to iterate over the items in the MyQueue. How should you declare MyQueue?

- A: public class MyQueue<T> : IDictionary< TKey, TValue >
- B: public class MyQueue : List<T>
- C: public class MyQueue : IEnumerable<int>
- D: public class MyQueue<T> : IEnumerable<T>

Answer: D, MyQueue needs to implement IEnumerable to satisfy the requirement. B could possibly work but is incorrectly defined.

Module 5 – C# Inheritance

Module Overview

Object-oriented programming helps to reduce complexity by offering a development paradigm that is modeled around real-world objects and how you manipulate them. These objects might be physical; a dog, a person, or a building; or more abstract concepts, like a message, or an index. The first task is to understand the problem domain so that you can define the objects and how they relate to each other. Most object-oriented languages have a concept of classes. A class represents a template for an object, and an object is an instance of a class.

A key part of this way of representing real-world objects, and fundamental to the concept of object-oriented programming, is the idea of **inheritance**. By making a class inherit from another we are able to create new classes without duplication. We say that the derived class

extends the parent class – you get everything that the base class has, but you can also add more methods and properties as needed. Typically, as you move up an object hierarchy, the classes become more generalized and abstract in nature, if not by definition. And as you move down the hierarchy the classes become more specific and closer to real-world objects. Code that is common between classes can usually be moved up the hierarchy and shared between the derived classes lower in the hierarchy.

Why does this matter? Because duplication is the antithesis of good programming, and we want to eliminate it at all costs. It's not just to save the effort of writing code twice. When code gets duplicated the end result is usually that code starts to diverge; you end up with functionality defined in multiple places, and bugs get fixed in one place and not another. It leads eventually to unmaintainable software.

A well thought-out class hierarchy eliminates duplication and makes it possible to separate different ‘concerns’ in code. This avoids the situation where changing one thing in one part of an application affects everything else, or where a simple change involves many diverse parts of the code-base. As the scale of a software application increases, these advantages become more and more important.

In this module we'll see how to create classes that take advantage of inheritance, and use them in code.

Objectives

After completing this module, you'll be able to:

- Create classes that implement inheritance.
- Understand the concept of polymorphism.
- Extend framework classes.

Lesson 1 – Hierarchies of Classes

You don't always have to start from scratch when making a new class. Instead, you can often use an existing class as a starting point. We call this "inheritance". Your class gets all of the members from the base class, and then you add some additional data and behaviour as needed, making the new class a more specialized implementation of its parent class. You'll see how this works in this lesson.

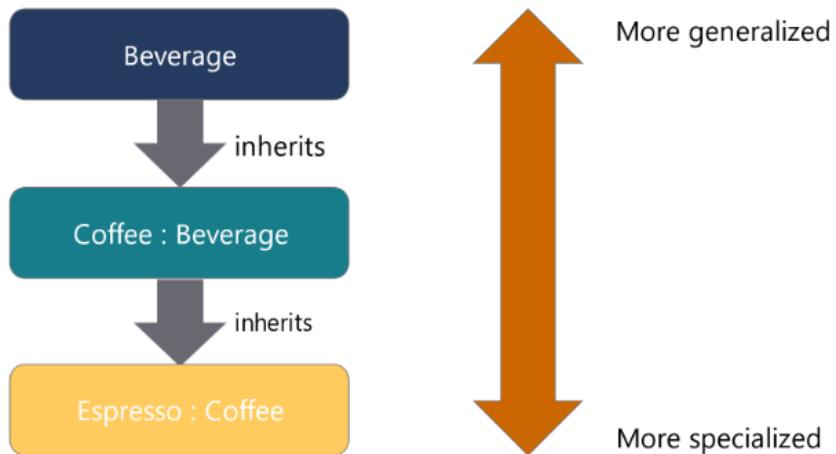
Lesson Objectives

After completing this lesson, you'll be able to:

- Understand inheritance.
- Define base classes.
- Derive classes from base classes.
- Call base class methods from derived classes.

What Is the Inheritance?

What Is Inheritance?



The diagram shows a class hierarchy where a class named Espresso inherits from a class named Coffee, which in turn inherits from a class named Beverage. The inherited classes are increasingly specialized instances of the base class.

At the top of our hierarchy we have a class called “Beverage”. This is a fairly abstract concept – you wouldn’t go into a coffee shop and ask for a “beverage”. If you did, the barista would probably look at you as though you were crazy, before politely asking you to be more specific. If you then asked for a “coffee” you’d be taken more seriously, but you’d know that a follow-up question was coming. If you then asked for a cappuccino, depending on the type of coffee shop, you might get another question, for example “What size?”, and whether you wanted chocolate sprinkled on top.

In terms of class hierarchy design, it’s at this point that you have to make a decision. Should a large espresso be a different class to a small espresso? Probably not. More likely a class called Espresso, or one of its ancestor classes, would have a **property** called Size. We might sense that we’ve probably gone far enough with our class hierarchy, and one of the skills in this kind of design is knowing when to stop.

Having considered where we might stop, there remains the question of where to start. If the application was dispensing food, as well as drink, then we might make a case for an even higher-level base class, perhaps a Comestible class. But let's assume this application is part of a drinks dispensing system, in which case Beverage is a good choice as the most general thing we might want to deal with. Let's see what a Beverage class might look like:

```
public class Beverage  
{  
    public double Temperature { get; set; } // deg C  
    public double Size { get; set; } // ml  
    public override string ToString()  
    {  
        return "Beverage";  
    }  
}
```

Now we need a class for each type of beverage, such as Coffee and Tea. We could make separate Coffee and Tea classes, but we'd need to redefine any members that were common to both. Cups of coffee and cups of tea both have the characteristics of temperature and size, so there's an advantage in deriving both classes from the Beverage base class. They also both need a way of being rendered as a string, but that might be different in each case. In practice it might not make much sense to have an instance of a Beverage, but we've implemented a `ToString()` method anyway. This replaces the default `ToString()` method that every class gets automatically, hence the need for the **override** keyword.

To inherit a class you use the same syntax as for implementing an interface, by adding the class name after a colon. Here's how we might create a Coffee class that derives from the base Beverage class:

```
public class Coffee : Beverage  
{  
    public string Bean { get; set; }  
    public string Roast { get; set; }  
    public string CountryOfOrigin { get; set; }  
    public override string ToString()
```

```
{  
    return "Coffee";  
}  
}
```

Note: Some class-based object-oriented languages allow you to have multiple inheritance, in other words you can have a class that derives from multiple base classes. This is not supported in C#. You can implement multiple interfaces but only inherit from one class.

We've overridden the `ToString()` method and added a few properties. Let's see how we can instantiate and use the `Coffee` class:

```
var drink = new Coffee();  
  
drink.Bean = "Arabica";  
  
drink.CountryOfOrigin = "Columbian";  
  
drink.Temperature = 85;  
  
Console.WriteLine(drink);
```

Notice that we used the `Bean` and `CountryOfOrigin` members of the `Coffee` class, but also the `Temperature` member of the `Beverage` base class. If we run this code, we'll get the output: `Coffee, Columbian Arabica.`

We can also create another class `Tea` that will also derive from `Beverage`:

```
public class Tea : Beverage  
{  
  
    public string Leaf { get; set; }  
  
    public string CountryOfOrigin { get; set; }  
  
    public override string ToString()  
{  
        return "Tea, " + Leaf + ", " + (Temperature > 80 ? "hot" : "warm");  
    }  
}
```

The Tea class is distinct from both the Coffee class and the Beverage base class. Tea isn't made from Beans, it's made from Leaves, so it makes no sense to derive from the Coffee class. But it does have a size and temperature, so deriving from Beverage makes sense from the point of view of the class hierarchy design. It also intuitively makes sense in that tea and coffee are both types of beverage. We also implement `ToString()` a little differently. The possible types of tea is more diverse but the country of origin is not quite such a big thing with tea drinking, but still relevant.

We can now make a cup of tea:

```
var drink2 = new Tea();  
  
drink2.Leaf = "Earl Grey";  
  
drink2.CountryOfOrigin = "Sri Lanka";  
  
drink2.Temperature = 85;  
  
Console.WriteLine(drink2);
```

And the output is, of course, Tea, Earl Grey, hot. But you might notice that both the Tea and Coffee classes have a `CountryOfOrigin` property. We'll address this in the next section.

Creating Base Classes

Creating Base Classes

- Use the `abstract` keyword to create a base class that cannot be instantiated

```
public abstract class Beverage
```

- Create a class that derives from the abstract class
- Implement any abstract members

- Use the `sealed` keyword to create a class that cannot be inherited

```
public sealed class Tea : Beverage
```

We've created two classes, Tea and Coffee, that both derive from the Beverages base class. We also noted that both Tea and Coffee have a `CountryOfOrigin` property. Have we discovered something that is universal to beverages, or is it just a few drinks that have this property? If you take the view that most drinks will have a `CountryOfOrigin`, which seems

reasonable, then we have introduced some duplication. We can fix this by moving `CountryOfOrigin` into the `Beverage` class, and removing it from the derived classes:

Another problem is that there is nothing preventing us from creating an instance of the `Beverage` class. But does the following code really make sense?

```
var drink3 = new Beverage();  
drink3.Temperature = 85;  
Console.WriteLine(drink3);
```

There's nothing stopping us doing this, and it will print out: `Beverage`, as it was meant to. But we could take the view that we don't want to be able to create a `Beverage` instance any more than we want to walk into a coffee shop and ask for one. We can tell the compiler to enforce that by making `Beverage` an **abstract** class. An abstract class, as the name implies, doesn't exist in the real world other than as a concept – you can only normally have a specific drink. Here's a revised version of the `Beverage` base class:

```
public abstract class Beverage  
{  
    public double Temperature { get; set; } // deg C  
    public double Size { get; set; } // ml  
    public string CountryOfOrigin { get; set; }  
    public abstract override string ToString();  
}
```

We've used the **abstract** keyword to make `Beverage` an abstract class. We've also removed the implementation of the `ToString()` override. Now the following code won't compile:

```
var drink3 = new Beverage(); // won't compile
```

We can also mark individual members of `Beverage` as abstract. By doing this, it's a bit like declaring an interface; we're saying if you derive from this class you have to implement that method or property. We've done that with the `ToString()` method override, so that any class derived from `Beverage` that doesn't implement `ToString()` won't compile.

Note: you can only declare members as abstract within an abstract class.

At this point we have two layers in the class hierarchy, but we may want to go further. Espressos and cappuccinos and americanos are distinct drinks, so it makes sense to create classes for these, and these should derive from our `Coffee` class. For example:

```
public class Americano : Coffee
{
    public bool Milk { get; set; }

    public override string ToString()
    {
        return "Coffee, Americano, " + (Milk?"white":"black");
    }
}
```

We can create an instance of Americano in the usual way, with an increasing list of possible parameters:

```
var drink3 = new Americano();

drink3.Bean = "Arabica";

drink3.CountryOfOrigin = "Columbian";

drink3.Temperature = 85;

drink3.Milk = true;

Console.WriteLine(drink3);
```

The output is: Coffee, Americano, white. An americano is a distinct drink, and we describe it in a distinct way.

But with teas it's not so obvious. Is an Earl Grey fundamentally different from a regular black tea? Maybe we should add the Milk property to the class Tea. What about herbal teas? Are they just a tea with different parameters, or are herbal teas a different kind of drink? Sometimes class design becomes something of a philosophical discussion. Suppose we take the view that we need a new class called HerbalTea, derived from Beverage. What about the Tea class – would we ever want to have derived types of Tea, like we did with Coffee.

Imagine that we create our carefully designed Beverage class hierarchy, and it gets used by other developers. Then we discover that there's been a proliferation of new classes derived from Tea. There's a TeaWithMilk class, and a HotTea class, and a LemonAndGingerTea class out there, and everyone's confused. We never intended our classes to be used in this way, with classes that should be derived from Beverage, and others that are really just regular Tea objects with certain specific property values. There is a way you can stop this by use of the **sealed** keyword.

```
public sealed class Tea : Beverage
{
    public string Leaf { get; set; }

    public override string ToString()
    {
        return "Tea, " + Leaf + ", " + (Temperature > 80 ? "hot" : "warm");
    }
}
```

Now, if you try to create a new class derived from Tea, it won't compile.

Note: obviously you can't have a class that's both abstract **and** sealed.

You might wonder what is the point of making Beverage abstract, or Tea sealed – couldn't we just remember how they're meant to be used? This is probably true for a small amount of code that only you will use, but for a larger projects you tend to find classes are written by one person or group, and then used by many others. Things like access modifiers, abstract base classes, and sealed classes, all help to make sure that classes are used the way the designer intended, which ultimately reduces the risk of errors and problems further down the line.

Creating and Inheriting from Base Class Members

- Use the **virtual** keyword to create members that you can override in derived classes
- Use the **protected** access modifier to make members available to derived types
- To override virtual base class members, use the **override** keyword
- To prevent classes further down the class hierarchy from overriding your override methods, use the **sealed** keyword

In our example Coffee and Tea classes we had overrides of the `ToString()` method, which was itself an override in the base class. `ToString()` is a **virtual** method that every class has,

because ultimately they all derive from Object. What if we add a new method to Beverage that we want to have the option of overriding in any derived classes. Let's add another method to Beverage:

```
public string getRecipe()
{
    if (Temperature > 25) return "Heat some water to " + Temperature + " degrees.";
    else return "Take some cold tap water";
}
```

This is our most generic recipe to get you started. If we want to make coffee, there's a bit more to it. We need to return a different string if the getRecipe() method is called on a Coffee object. We could try just adding this to the Coffee class:

```
public override string getRecipe()
{
    return "Use the espresso machine to make a shot.";
}
```

If we omit the **override** keyword we'll get a warning that the new getRecipe() method "hides" the underlying Beverage.getRecipe() method, which indicates a possible mistake. If that's really what you want, you can use the **new** keyword to tell the compiler this was intentional. This will have the desired effect, but there's a better way.

Note: For more on the differences in between the **override** and **new** keywords, see:
<https://aka.ms/3LzeUrU>.

If the intention was to **override** the method on the base class, then you need to use the **override** keyword as shown, but this will give a compiler error. The problem is that we need to indicate that we want to allow the method on the base class to be overridden. To do this we need to change the definition of the getRecipe() method on the Beverage class:

```
public virtual string getRecipe()
```

Using or not using the **virtual** keyword gives us control over whether a method can be overridden in a derived class. We can get a further degree of control by the use of access modifiers. Instead of making it **public**, we could choose to make it **protected**, in which case it would only be accessible to derived classes, or of course within the class itself. We could also use the **internal** keyword which would limit access to code that's part of the same .NET assembly. Finally, there's the **private** modifier (which is the default), which means that the member is only accessible within the same class. You can also

combine **protected** with **internal** or **private**, so that **protected internal** is the union of **protected** and **internal**.

You can also use the **sealed** keyword on an overridden member. For example, we could modify the `getRecipe()` method in the `Coffee` class:

```
public sealed override string getRecipe()
{
    return "Use the espresso machine to make a shot.";
}
```

This will prevent anybody overriding the method in any derived classes. From the point of view of a derived class, it is as if the method was never made virtual further up the hierarchy.

Question: True or false: the members of a **sealed** class are only accessible in derived classes?

Answer: False; the purpose of the **sealed** modifier is to prevent the overriding of methods in derived classes.

Calling Base Class Constructors and Members

Calling Base Class Constructors and Members

- To call a base class constructor from a derived class, add the **base** constructor to your constructor declaration

```
public Coffee(int temp, double size, string bean)
    : base(temp, size)
```

- Pass parameter names to the base constructor as arguments
- Do not use the **base** keyword within the constructor body
- To call base class methods from a derived class, use the **base** keyword like an instance variable
 - `base.ToString();`

Even though you've overridden a method, you still occasionally need to call the method on a base class. For example, let's suppose that we're building up the `getRecipe()` string by calling

the base method and adding to the string in the derived methods. We can do that by using the **base** keyword.

You can also do the same for constructors. Maybe a derived class has a constructor, but you need to call the base class constructor, perhaps with some of the arguments passed through. Let's redefine our Beverage and Coffee classes, except this time we'll define some non-default constructors as well as an explicit default constructor. We'll also modify the `getRecipe()` method to build up the string by calling the Beverage version from the Coffee override. First, here's the revised Beverage class:

```
public abstract class Beverage
{
    public double Temperature { get; set; } // deg C
    public double Size { get; set; } // ml
    public string CountryOfOrigin { get; set; }

    public Beverage() {}

    public Beverage(double temp, double size)
    {
        Temperature = temp;
        Size = size;
        CountryOfOrigin = "UK";
    }

    public abstract override string ToString();

    public virtual string getRecipe()
    {
        if (Temperature > 25) return "Heat some water to " + Temperature + " degrees.";
        else return "Take some cold tap water";
    }
}
```

The Coffee class has been updated with a new constructor that calls the base constructor:

```
public class Coffee : Beverage
{
    public string Bean { get; set; }
    public string Roast { get; set; }
    public Coffee() { }

    public Coffee(double temp, double size, string bean) : base(temp, size)
    {
        Bean = bean;
    }

    public override string ToString()
    {
        return "Coffee, " + CountryOfOrigin + " " + Bean;
    }

    public sealed override string getRecipe()
    {
        return base.getRecipe() + "\r\nUse the espresso machine to make a shot.";
    }
}
```

We can now create an instance of the Coffee class using the new constructor, and call the getRecipe() method:

```
var drink = new Coffee(85, 200, "Arabica");
drink.CountryOfOrigin = "Columbian";
Console.WriteLine(drink);
Console.WriteLine("Recipe:");
Console.WriteLine(drink.getRecipe());
```

Calling this produces the following output:

Coffee, Columbian Arabica

Recipe:

Heat some water to 85 degrees.

Use the espresso machine to make a shot.

Note that this is for illustration of the class hierarchy design principles. Please don't rely on this recipe to make coffee.

By calling methods on the base class we have the option of code re-use. For example, in our Tea class we might add something about infusing the tea after calling the base class `getRecipe()`. For a more realistic Tea constructor, we might change the Tea constructor to specify a fixed temperature of 100 degrees. Tea should always be made with boiling water:

```
public sealed class Tea : Beverage
{
    public string Leaf { get; set; }

    public Tea() {}

    public Tea(double size, string leaf) : base(100, size)
    {
        Leaf = leaf;
    }

    public override string ToString()
    {
        return "Tea, " + Leaf + ", " + (Temperature > 80 ? "hot" : "warm");
    }

    public override string getRecipe()
    {
        return base.getRecipe() + "\r\nFill an infuser with leaf tea and add to the water.";
    }
}
```

This produces the following output:

Tea, Earl Grey, hot

Recipe:

Heat some water to 100 degrees.

Fill an infuser with leaf tea and add to the water.

Classes derived from the Tea or Coffee classes might add further steps while re-using the higher-level steps.

Note: Inheritance doesn't work in the same way for static classes and their members, so you can't use the **base** keyword in a static method.

Question: True or false: you can call the base class constructor in a derived class constructor?

Answer: True; you use the `: syntax` to invoke the base constructor in the definition of the constructor in the derived class.

Lesson 2 - Polymorphism

In computer programming, ‘polymorphism’ means that we have a type of object that can exist in more than one form. In C# this means that we can use a base class that represents instances of more than one subclass. To use a more concrete example, we can use a **Beverage** object in our code without needing to know whether it’s actually a **Coffee** or a **Tea**, or a **Cappuccino**. This may seem rather arcane but it’s actually a powerful tool that allows us to write code at a higher level of abstraction. That means we can write clear code without needing lots of “if **coffee** then do this, but if **tea** then do that” scattered everywhere.

Polymorphism is one of the four pillars of object-oriented programming. Understanding how to use polymorphism in your applications will enable you to write better code that’s easier to maintain.

Note: The word ‘polymorphism’, in the general sense, means things taking different forms.

Lesson Objectives

After completing this lesson, you’ll be able to:

- Use the class hierarchy to implement appropriate behaviour.
- Simplify logic by using polymorphism.

Introducing Polymorphism

Introducing Polymorphism

- Many forms
- Can be a derived class or a base class representation
- Base class implements virtual methods
- Derived class overrides implementation
- Derived class can implement nonvirtual methods to hide base class functionality
- Derived class can be treated as a parent class
- Additional functionality that is added to the derived class morphs the object

In the context of object-oriented programming, polymorphism can be implemented by treating an object of a derived class as if it was an object of the base class. For example, if the **Coffee** class inherits from a base class **Beverage**, we now have the choice of either treating it as a Coffee object or as a Beverage object. When we treat it as a Beverage object we only have access to the methods and properties of the Beverage object. We no longer have access to the specific methods and properties of the Coffee object.

However, remember that base classes can implement virtual methods, and the subclasses can override those virtual methods to provide their own implementation of that functionality. The base classes can also define abstract methods, and in that case the subclasses must always override them with their own implementation.

The important thing to grasp, and this is key to polymorphism, is that **if we call the base method of Beverage, and that base method has been overridden in Coffee, it's the Coffee method that gets called.**

You may remember that a third possibility is that a subclass can hide the implementation of a method in a base class by defining a **new**, nonvirtual method implementation. In that case you don't get the polymorphic behavior; if you want to use the `getRecipe()` method of Coffee then it needs to explicitly be a Coffee object.

In the previous lesson, we created a base class (`Beverage`) and created two derived classes, `Coffee` and `Tea`. Let's recreate the classes in simplified form. Here's the code for the `Beverage` class:

```
public abstract class Beverage
```

```
{  
public double Temperature { get; set; } // deg C  
public virtual string getRecipe()  
{  
return "Make a beverage";  
}  
}
```

The Beverage class implements the `getRecipe()` method, but it's declared as `virtual`. Now let's look at the Coffee class:

```
public class Coffee : Beverage  
{  
public override string getRecipe()  
{  
return "Make a coffee";  
}  
}
```

Coffee has an **override** of the `getRecipe()` method. Finally, we can look at the Tea class:

```
public sealed class Tea : Beverage  
{  
public new string getRecipe()  
{  
return "Brew up tea";  
}  
}
```

The Tea class just has a re-write of the `getRecipe()` method, indicated by the **new** keyword. That's not quite the same as an `override`, because the new `getRecipe()` function is going to "hide" the base class `getRecipe()` method.

This is a minimal class hierarchy with the Beverage class at the top serving as the super class, base class, or parent class, depending on how you choose to use it. The Coffee and Tea classes inherit the attributes and behaviors defined in the Beverage class. However a Coffee is a different kind of a Beverage than a Tea, so the Coffee and Tea classes will show some differences from the Beverage class, otherwise there would be no point in creating them. These differences are expressed by overriding methods that exist in the Beverage class, and also by adding new methods and properties that don't belong in the Beverage class. A Coffee will have differences when compared to a Tea, and both Coffee and Tea will have differences from Beverage.

This is where polymorphism becomes a powerful tool. Because a Coffee is also a Beverage, you can treat the subclass as an object of the base class; you can use a Beverage to represent a Coffee. Look at the following code:

```
Coffee coffee = new Coffee();  
Tea tea = new Tea();  
Console.WriteLine(coffee.getRecipe());  
Console.WriteLine(tea.getRecipe());
```

What will be the output? This one's fairly easy – the Coffee.getRecipe() method will get called, then the Tea.getRecipe() method. The output will be:

Make a coffee

Brew up tea

Nothing too surprising here. In both Coffee and Tea, we get the methods of the subclasses. Because both Coffee and Tea are classes derived from the Beverage base class, we have the option of treating them both as a Beverage. Consider the following code:

```
Beverage drink = new Beverage(); // won't compile
```

Remember that we can't instantiate a Beverage object because it is an abstract class. But we can do the following:

```
Beverage drink1 = new Coffee();
```

```
Beverage drink2 = new Tea();
```

Just think about what we did here; we have two references to objects of type Beverage, but the actual objects are Coffee and Tea instances, respectively. In the next section, we'll take polymorphism to the next level.

Apply Polymorphism to Functions (Overloading, Overriding)

Apply Polymorphism to Functions (Overloading, Overriding)

- Polymorphism through functions
 - Overriding
 - Override virtual methods
 - Override nonvirtual methods
 - Override abstract methods
 - Method Overloading
 - Same name, different parameters

We can really take advantage of polymorphism by making use of method overriding and overloading. When you override a base class method in a derived class, there are three options:

- You optionally override a virtual method that is defined in a parent class (using **override** keyword).
- You optionally override a nonvirtual method that is defined in a parent class (using **new** keyword)..
- You override an abstract method that is defined in a parent class (using **override** keyword).

Overriding a virtual method means that the method in the base class is declared with the **virtual** modifier (or is **abstract**). You can also override a method if the method in the parent class is itself an override – because that means the ultimate base class must be **virtual**. This means that when you implement the subclass you have the choice of overriding the method, or letting the subclass use the base method (as long as it isn't **abstract**). Calling the overridden method invokes the code in the subclass.

Overriding a nonvirtual method means that the base class and the subclass both have a method with the same signature. The signature consists of the return type, name, and any parameters for the method. In this case, the base class and subclass will likely have their own implementation that differs from the other. When you call the method in the subclass, it hides the method in the parent class resulting in the method being executed. You can still call the method in the parent class, but it has to be an explicit call to the method of the parent class.

Remember that in the code from the previous section we have a **virtual** `getRecipe()` method on the base **Beverage** class, an **override** on the `getRecipe()` class in the **Coffee** subclass, and a **new** `getRecipe()` method in the **Tea** subclass. We already saw that we can create **Coffee** and **Tea** objects and the respective methods are called. But we can also treat **Coffee** and **Tea** objects as **Beverage** objects. What happens if we do that? Consider the following code:

```
Beverage drink1 = new Coffee();  
Console.WriteLine(drink1.getRecipe());
```

What will be the result of calling the `getRecipe()` method on each object? You might think they'd both produce the output `Make a beverage`. But if we execute that code, we actually get the following output:

`Make a coffee`

This might not be what you were expecting. Even though we have a **Beverage** object, the `Coffee.getRecipe()` still gets called. This is interesting, because it means we can write very general code that deals with **Beverage** objects without worrying whether they are a **Coffee**, or a **Tea**, or some other type of **Beverage** that we haven't thought of yet. General things about a **Beverage**, like their name, all work as you'd expect, but when we call a method that has a special **Coffee** behavior, the **Beverage** just **does the right thing™**. If we didn't have this kind of polymorphism, we'd have to write lot's of code like:

```
if(drink is Coffee) doCoffeeRelatedStuff();  
else if(drink is Tea) doTeaRelatedStuff();
```

What if we built our application and then realized we needed to account for a new kind of **Beverage** called a **HotChocolate**? We'd have to revisit everywhere in our code that worked out what kind of **Beverage** it was and add all the **HotChocolate** behavior. By using polymorphism we can add a new **HotChocolate** class derived from **Beverage** that includes any special behavior and variables. The only code that needs to change is the creation of the object, where of course it needs to be:

```
Beverage drink3 = new HotChocolate();
```

From that point on, if we use polymorphism correctly, it can be treated as just another **Beverage** object.

We saw that a **Coffee** object's `getRecipe()` method got called correctly, even when we passed it as a **Beverage** object. Now let's try the same with the **Tea** class:

```
Beverage drink2 = new Tea();  
Console.WriteLine(drink2.getRecipe());
```

If we execute that code, we get the following output:

Make beverage

You might have thought it would call the Tea's getRecipe() method, but this is clearly calling the base class method. This is because Tea didn't **override** the getRecipe() method, it replaced it by using the **new** keyword. That's not the same as an override, even though the base method was marked as a virtual. To call the Tea's getRecipe() method we need to have a Tea, and sometimes that's what you want.

If you really want to call the method on the subclass in this situation, you'd have to do a cast, e.g. ((Tea)drink2).getRecipe(); which is pretty ugly, and you'd better be sure that drink2 really is an Tea or you'll get a runtime exception.

If you want the drink2 Beverage reference to automatically use the getRecipe() method on the Tea class, you need to make the method an **override** instead of a **new**:

```
public sealed class Tea : Beverage
{
    public override string getRecipe()
    {
        return "Brew up tea";
    }
}
```

Another way to implement polymorphism is to use method overloading, which allows us to create multiple versions of the same method by just changing the parameter types (the signature). For example, you could create three Add methods; Add(int a, int b), Add(int a, int b, int c), and Add(int[] arrInts). Although each method uses same name, during a method call, the compiler can differentiate between them by looking at the number and type of parameters being passed in. For this to work, the parameters have to differ by type; you can't just change the names, or change the return type. Examples of an overloaded method are shown in the following code.

```
public sealed class Tea : Beverage
{
    public override string getRecipe()
    {
        return "Brew up tea";
    }
}
```

```
public string getRecipe(double temperature)
{
    if (temperature < 90) return "Heat water some more";
    return getRecipe();
}
```

Notice that you have the option of calling another overload of a method from that same method. We don't use the **override** keyword on our second overload of `getRecipe`, because there is no such method in the base class. This also means we can't call that second overload from a `Beverage` reference unless we also added it to the base class. The following code will work, though:

```
Tea drink = new Tea();

Console.WriteLine(drink.getRecipe());
Console.WriteLine(drink.getRecipe(88));
Console.WriteLine(drink.getRecipe(99));
```

The resulting output is:

Brew up tea

Heat water some more

Brew up tea

Lesson 3 - Extending Classes

There are several thousand classes in the .NET framework that provide a wide range of functionality. Where possible you should try to build on these classes by inheriting from built-in .NET types, rather than create your own classes from scratch. This reduces the amount of code you need to write and maintain, and also helps to make sure that your classes are consistent with the rest of .NET.

Sometimes you need to extend the functionality of .NET framework classes, or other classes written by others, where you don't have the option of inheriting from them. An example is the built-in `String` class. So another option in .NET is to create extension methods that add functionality to sealed classes, including sealed .NET library types.

Lesson Objectives

After completing this lesson, you'll be able to:

- Create classes that inherit from .NET Framework types.
- Create custom extension classes, and throw and catch them.
- Create classes that inherit from generic types.
- Create extension methods.

Inheriting from .NET Framework Classes

Inheriting from .NET Framework Classes

- Inherit from .NET Framework classes to:
 - Reduce development time
 - Standardize functionality
- Inherit from any .NET Framework type that is ~~sealed~~ or ~~static~~
- Override any base class members that are marked ~~virtual~~
- Implement any base class members that are marked ~~abstract~~

Not all of the many thousands of .NET public classes are extendable, but many are. These libraries provide implementations for many general purpose types, algorithms, and utility classes. This means that most of the common tasks you'll perform with system access, such as files and memory, network operations, encryption, have already been written for you.

There's often an existing built-in class that is suitable for your needs, saving you the time of writing your own code and, even more importantly, the burden of maintaining it. Some of the classes included are:

- Base C# types
- Exception classes
- Data collections such as lists, stacks, and dictionaries
- I/O operations for streams and files
- Reflection for looking at types
- Access to external data

- APIs for building graphical user interfaces

In contrast to developing a new class from scratch, a class that is derived from a .NET library class has the following advantage of reducing development time. Inheriting from an existing class reduces the amount of logic you have to write yourself. In addition, by inheriting from a standard base class your own class will inherit a standardized interface. It's much easier to work with related classes if they all derive from the same base class, because you can reference instances of your own class as instances of the corresponding base class.

Classes built into .NET follow the same inheritance rules as custom classes:

- A class can derive from a .NET library class, provided that the class is not created as sealed or static.
- Virtual base class members can be overridden.
- You must implement all abstract members if you inherit from an abstract class.

When creating a class, choose a base class that will minimize the amount of required coding. Whenever you need to duplicate functionality that is provided by built-in classes you should generally look for one that provides more specific functionality. If, however, you find you need to override many members, at that point you may want to consider a base class that's more general.

As an example, suppose you want to create a collection class that stores a list of values, and one of the requirements is that the class must be capable of removing duplicate items. You can accomplish this without having to build a class from scratch by inheriting from the generic `List<T>` class. and then just adding a method that will remove the duplicates. Additionally, you have the option of invoking the existing `Sort` method in the `List<T>` base class so that any duplicate items in the collection will be adjacent. You can then easily identify and remove the duplicates with just a small amount of additional code.

Creating Custom Exceptions

Creating Custom Exceptions

To create a custom exception type:

1. Inherit from the `System.Exception` class
2. Implement three standard constructors:
 - `base()`
 - `base(string message)`
 - `base(string message, Exception inner)`
3. Add additional members if required

Most runtime error conditions raise an exception using one of the many exception classes that are built into the .NET framework. These exceptions range from the most general; the `Exception` class, to more specific ones such as `IndexOutOfRangeException` or `DivideByZeroException`. All these exceptions are in the `System` namespace. Most of these exception names are self-explanatory.

Note: to find out more about the exceptions that are defined in the `System` namespace, see: <https://aka.gd/3G4FLuL>.

These built-in exceptions are thrown by framework classes when things go wrong, but you can also choose to instantiate and throw these exceptions yourself. A good practice is to use the built-in exceptions where possible, and to use the most specific exception available. As a last resort you can show the generic `Exception` object:

```
throw new Exception("Oh, noes!");
```

However, it would be very difficult to add exception handling logic elsewhere in your code with such a generic exception. If you can't find a more specific and relevant existing exception type, another option is to create a custom exception type. When you write a catch block you choose the exception type so that you can catch specific exceptions or handle them in a particular way. An exception type serves to provide a general category of problem, while its properties can be used to contain detailed information about the problem. If your handler is going to do nothing more than report the problem like any other exception, then there's little reason to create a custom exception type. You can simply provide the information in the exception message and use one of the built-in exception types.

If you decide you really do need a custom exception, you'll need to make a new class that derives from `System.Exception`, or from any more specific exception type that's suitable. All

exception classes ultimately inherit from the System.Exception class. You can use the following properties of this class to give more information about the error condition:

Property	Description
Message	Lets you add a text string with more information about what happened.
InnerException	Allows you to attach another Exception that was the ultimate cause.
Source	Specifies the item that caused the error.
Data	Lets you add more information in the form of key/value pairs.

Where possible, you should use these properties, but you can also add your own properties when you define your custom exception class. The following example shows a definition of a custom exception:

```
public class BeverageException : Exception
{
    public BeverageException()
    {
        // base class constructor
    }

    public BeverageException(string message) : base("Beverage Issue: " + message)
    {
        // calls base class constructor with customized message
    }
}
```

In this trivial example we've simply augmented the error message. It's a good practice to also define the base(string message, Exception inner) constructor as well.

Note: it's a convention to end the name of an exception class with the word "Exception".

Throwing and Catching Custom Exceptions

- Use the `throw` keyword to throw a custom exception

```
throw new BeverageException();
```

- Use a try/catch block to catch the exception

```
try
{
    // Perform the operation that could cause the exception.
}
catch(BeverageException ex)
{
    // Use the exception variable, ex, to get more information.
}
```

Once you've made your custom exception type, you can treat custom exceptions in the same way you would treat other exceptions. You use the `throw` keyword and make a new instance of your exception type to throw a custom exception.

This is how we would throw our custom exception in code:

```
if(somethingBadHappened == true)

throw new BeverageException("Oh, noes!");
```

Because our custom exception changes the exception message, it will be: 'Beverage Issue: Oh, noes!'.

We can catch the exception elsewhere in code using a try/catch block. It's likely that the motivation for having a custom exception was to perform some special logic when it's thrown. We can therefore wrap any code that's likely to throw the exception with a try/catch block where the catch block accepts our custom exception. If we wanted to catch other exceptions we could add further more general catches:

```
try

{
    // Perform the operation that could cause the exception.

    throw new BeverageException("Oh, noes!");
}
```

```
}

catch (BeverageException ex)

{

    // Code to deal with this specific exception.

    // Use the exception variable, ex, to get more information.

    Console.WriteLine(ex.Message);

}

catch (Exception ex)

{

    // Deal with a more generic exception.

    Console.WriteLine(ex.Message);

}
```

Because the custom exception derives from the System.Exception class, if you had a “catch all” handler for type Exception, it would catch our BeverageException if another handler didn’t get to it. You can also have a hierarchy of exception classes that get more specific the lower down the hierarchy you go, if that makes exception handling easier. It really depends on the application.

Inheriting from Generic Types

Inheriting from Generic Types

For each base type parameter, you must either:

- Provide a type argument in your class declaration

```
public class CustomList : List<int>
```

- Include a matching type parameter in your class declaration

```
public class CustomList<T> : List<T>
```

You can inherit from generic types just like you would for other non-generic types. However, you are presented with a dilemma; should you continue to implement your derived class as a generic type, or does it make more sense to create a custom class that is non-generic? Both are possible.

To give an example, suppose you want to create your own custom list. Perhaps your list will be a collection of numbers and you want to create an Average method that returns the average value. You could define that as a NumberList class that derives from the List<T> collection class in the System.Collections.Generics namespace:

```
public class NumberList<T> : List<T>

{
    public T Average() // won't compile

    {
        T total = 0;

        foreach (var item in this) total += item;

        return total / this.Count;
    }
}
```

At first glance, this looks as though it ought to work. We also get the advantage that we have a generic NumberList type. We could declare a new NumberList<int> or NumberList<float>. The problem is that the compiler doesn't know what T is, so it can't do a numeric conversion. The best you can do is some kind of conditional logic that depends on retrieving the type of T at runtime and choosing a different execution path for each numeric type, otherwise throw an exception. There are many types of logic we can do on any type, and in that case a custom generic collection would make sense. But arithmetic operations are more restrictive and there isn't even a constraint that we can use on the type of T. The problem is in the Average() method, not the idea of having a derived generic class.

In this situation, trying to make a generic collection with an Average method might not be worth the effort. It probably makes more sense to decide what numeric type we're going to support and make our collection specific to that type. We can do this while still deriving from the generic base type:

```
public class NumberList : List<double>

{
    public double Average() // won't compile
```

```
{  
    double total = 0;  
  
    foreach (var item in this) total += item;  
  
    return total / this.Count;  
}  
}
```

This will compile, and if we run the code:

```
var list = new NumberList() { 3, 5, 77, 8, 20 };  
  
Console.WriteLine(list.Average());
```

we get the correct result of the average of the numbers (22.6). The only disadvantage is that our NumberList class isn't generic – internally it's using **double** values. If we try to add a number to the collection that isn't double, we'll get an automatic type conversion. The exception is if we try to convert from a larger type, so for example this won't work:

```
Decimal d = 55;  
  
list.Add(d); // won't compile
```

But this will:

```
Decimal d = 55;  
  
list.Add((double)d); // explicit conversion
```

When we tried to create a generic NumberList class we passed the type T to the base class. The name "T" is arbitrary and is used by convention. If you extend a class with multiple types, you might need multiple types, or pre-define some of them in your derived custom class. If we decided to make a catalog of beverage recipes, we might create some kind of custom BeverageDictionary, with one or more of the types being pre-defined. For example, the following are all possible, at least as far as the definition:

```
public class BeverageDictionary< TKey, TValue > : Dictionary< TKey, TValue > {}  
  
public class BeverageList< T > : Dictionary< string, T > {}  
  
public class BeverageDictionary : Dictionary< int, string > {}  
  
public class BeverageDictionary2< TKey, TValue > : List< TKey > {}  
  
public class BeverageDictionary3< T > : List< string > {}
```

```
public class NonGenericListClass { }

public class BeverageCollection<T> : NonGenericListClass { }
```

Creating Extension Methods

Creating Extension Methods

- Create a static method in a static class
- Use the first parameter to indicate the type you want to extend
- Precede the first parameter with the `this` keyword

```
public static bool ContainsSpaces(this string s) {...}
```

- Call the method like a regular instance method

```
string text = "Text with spaces";
if(text.ContainsSpaces)
{
    // Do something.
}
```

Sometimes you want to extend the functionality of a class, but it's either a built-in or from a library that you don't control, and the author has declared it as **sealed**. In this case inheriting from it is not an option. However, you can still use a feature of C# called "extension methods". These are methods that you can attach to a type to augment it, rather than modifying it directly. In effect you are adding a static method to the underlying type.

In order to create an extension method, you need to create a static class and then add a static method where the first parameter is the class that you want to extend. There's a special syntactic use of the `this` keyword that signals to the compiler that this is an extension method for the target class.

As an example, let's suppose that we want to check strings to see if they contain any spaces. We would ideally like to modify the built-in `String` class, or extend a derived version of it, to add a method called `ContainsSpaces`, but we can't:

```
public class CustomString : System.String { } // won't compile
```

The `System.String` class is sealed. But we can instead create an extension method:

```
public static class CustomExtensions
```

```
{
```

```
    public static bool ContainsSpaces(this String s)
```

```
{  
foreach(char c in s) if(c == ' ') return true;  
return false;  
}  
}
```

You can now use the ContainsSpaces methods on strings wherever the CustomExtensions class is in scope. Typically you would add a **using** statement for its namespace. You can use it just as if it was a method on the System.String class, and Visual Studio's Intellisense will even offer ContainsSpaces as a method on string objects. You can test it using the following code:

```
var s = "abcdefg";  
Console.WriteLine(s.ContainsSpaces());
```

The program will print out false, but if you add any spaces to the string it will print true.

This demonstration shows the tasks that you need to perform in the lab for this module.

Lab: Refactoring

Lab: Refactoring Common Functionality into the User Class

Lab scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Objectives

Exercise 1:
Creating and Inheriting
from the User Base
Class

Exercise 2:
Implementing Password
Complexity by Using an
Abstract Method

Exercise 3:
Creating the
ClassFullException
Custom Exception

Module Review and Takeaways

Review Questions

Question: Of the following method types, which are mandatory to implement in a derived class?

- A: static
- B: public
- C: protected
- D: virtual
- E: abstract

Ans: E; you must implement abstract methods in a derived class.

Question: You create a string extension method. How does the compiler know that it is a method on string?

- A: it inherits from the string class
- B: the first parameter is of type string, preceded by the this keyword
- C: the method returns a string
- D: the method includes a <string> type parameter

Answer: B

Module 6 – Input and Output

Module Overview

A data processing system is of little use unless we can data into it, and get the results out. For example, a text file editor needs to be able to save a file to the file system, and a service needs to be able to write error messages to a log file. The local file system is the obvious place for an application to read and write this kind of data because access is fast and not dependent on connectivity (with the exception of ‘sandboxed’ environments such as script in browsers). There are many I/O classes in .NET that make it easy to add I/O functionality to your applications.

In this module we'll look at the various methods of input and output (I/O) that you can use for reading and writing of disk files.

Objectives

After completing this module, you'll be able to:

- Serialize and deserialize data.
- Read and write local disk files.

Lesson 1 – File I/O

We've already made extensive use of the `System.Console` class to read and write to the computer console. In addition to `System.Console`, the `System.IO` namespace in .NET contains numerous classes to support I/O. This lesson will introduce these classes and show how to manage files and directories and read data from, and write data to, the filesystem.

Lesson Objectives

After completing this lesson, you'll be able to:

- Read files using the `File` class.
- Create files and write data to the filesystem.
- Use the `FileInfo` class and the `Path` class.
- Use the `Directory` and `DirectoryInfo` classes

Reading and Writing Data by Using the File Class

Reading and Writing Data by Using the File Class

- The **System.IO** namespace contains classes for manipulating files and directories
- The **File** class contains atomic read methods, including:
 - `ReadAllText(...)`
 - `ReadAllLines(...)`
- The **File** class contains atomic write methods, including:
 - `WriteAllText(...)`
 - `AppendAllText(...)`

When you want to persist data to the file system you have a wide range of methods available. In most cases, you can use plain text files to store application data or configuration information such as user settings. Configuration data is often stored as XML files or JSON (JavaScript Object Notation), providing a structured way to store complex data in plain text files. There are .NET classes that can be used to for reading and writing plain text files at a fairly low level using the `StreamReader` and `StreamWriter` categories of classes, that require that you open a file handle, then open a stream, perform the read or write operations, and then release the file handle resource. We'll look at these later in this module.

In many cases you can just use one of a number of static methods that the **File** class provides. These wrap these low-level functions into a single, convenient method call. They are a quick way of doing File I/O without the ceremony of using the lower level APIs, and are a good option if you don't need the fine-grained control that more complex applications require.

If you just need to read the contents of a file you can use the `File.ReadAllText` method. It takes an argument of the file path and returns a string containing the file contents. If the file is not too big, this is a very convenient option:

```
var text = File.ReadAllText("TextFile1.txt");
Console.WriteLine(text);
```

This code will look for a file called `TextFile1.txt` in the current directory. We've provided the file name as a string, and also added a file in the project directory. If you try to run this in debug mode in Visual Studio, you'll probably get the following exception:

```
System.IO.FileNotFoundException: 'Could not find file 'C:\Users\BillA\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net6.0\TextFile1.txt'. '
```

When you debug in Visual Studio, the default behaviour is to run the application in the directory where the binary file is produced. This is a few levels down in the directory structure. You have a few options, such as changing the project configuration, or running the app in the project directory. The quickest way is probably to change the path to go up a few levels in the directory structure:

```
var text = File.ReadAllText(@"..\..\..\TextFile1.txt");
```

Note: We have preceded the string with the '@' symbol to avoid having to escape the '\' symbol. You could also escape each symbol and write the string as: "..\\..\\..\\TextFile1.txt".

The `ReadAllText` method will store the file contents as a single string, new line characters included. Sometimes you want to store the individual lines:

```
var text = File.ReadAllLines(@"..\..\..\TextFile1.txt");
```

In this case, `text` will be an array of strings, with each array element being the contents of one line. This is useful if you want to process a file line-by-line, but be aware that the whole of the file will be read into the array in one go. If you have a large file to process, this might not be the most efficient way of doing it.

If your file contains binary data, you can use the `ReadAllBytes` method:

```
var data = File.ReadAllBytes(@"..\..\..\DataFile1.dat");
```

After calling this, `data` will contain a byte array of data corresponding to the file contents.

When you want to write data to a file, you use a similar approach. The one difference is that you have the possibility that the file already exists, in which case you need to decide what to do; overwrite, append, or throw an exception. There are a number of `Write` methods that have the first of these behaviours, and there are `Append` methods for the second case. `WriteAllText` does the opposite of `ReadAllText` by writing the contents of a string to a file. It works in an almost identical way, adding a second argument for the text string. The following code will read a file and write the contents out to a second file, effectively doing a file copy:

```
var text = File.ReadAllText(@"..\..\..\TextFile1.txt");
File.WriteAllText(@"..\..\..\TextFile2.txt", text);
```

If the target file already exists, it is overwritten. If that's not the intended behaviour you'll need to use one of the APIs later in the module to check, and then throw an exception, warn the user, or whatever. You can also use the WriteAllLines method and give it a string array instead of just a string, and if you have binary data in a byte array, you can use the WriteAllBytes method:

```
byte[] data = {  
    109,112,97,197,109,99,112,110,121,241,102,178,101,111 };  
  
File.WriteAllBytes(@"..\..\..\DataFile2.dat", data);
```

There are also append versions of the WriteAllText and WriteAllLines methods, called AppendAllText, and AppendAllLines, that append to the files instead of overwriting them, which is a common requirement if you are doing any kind of logging. For example, we could keep adding more data to TextFile2.dat, so that it gets longer and longer:

```
var text = File.ReadAllText(@"..\..\..\TextFile1.txt");  
  
File.AppendAllText(@"..\..\..\TextFile2.txt", text);  
  
File.AppendAllText(@"..\..\..\TextFile2.txt", text);  
  
File.AppendAllText(@"..\..\..\TextFile2.txt", text);
```

If you're looking for a File.AppendAllBytes method, you're out of luck. If you want that functionality, you'll have to use the lower-level APIs to build it yourself.

Manipulating Files

Manipulating Files

- The `File` class provides static members

```
File.Delete(...);  
bool exists = File.Exists(...);  
DateTime createdOn = File.GetCreationTime(...);
```

- The `FileInfo` class provides instance members

```
FileInfo file = new FileInfo(...);  
...  
string name = file.DirectoryName;  
bool exists = file.Exists;  
file.Delete();
```

If you need to know if a file already exists, or create a directory, you'll want to use some of the other methods of the **File** class. As mentioned before, you might want to warn the user before doing a **File.WriteAllText** and overwriting an existing file, and give them the opportunity to cancel the operation. The **File.Exists** method will return true or false and is pretty self-explanatory. In the example below we write out to a file only if it doesn't already exist:

```
var outputPath = @"..\..\..\TextFile2.txt";
if (!File.Exists(outputPath))
    File.WriteAllText(outputPath, text);
else
    Console.WriteLine("File already exists!");
```

Sometimes you need to programmatically copy a file from one location to another. We already saw how to do this by reading a file into a string and writing it out again, but there's a simpler way:

```
var inputFilePath = @"..\..\..\TextFile1.txt";
var outputPath = @"..\..\..\TextFile2.txt";
File.Copy(inputFilePath, outputPath, true);
```

The third parameter to **Copy** is an overwrite flag. If we set this value to false and then run the code again, we'll get a **System.IO.IOException**, because the file already exists, and overwriting isn't allowed. We can also do most of the other common operations you might get at a command line, such as deleting a file with the **File.Delete(filePath)** method, **File.Create** and **File.Move**.

All the methods we looked at so far are static methods on the **File** class. Because they're static, we have to pass the file path every time. This can get tedious if you need to do a few operations on a file, so another option is to get an instance of the **FileInfo** object. You can get one of these by creating a new **FileInfo** instance and passing the file path to the constructor:

```
var file = new FileInfo(inputFilePath);
```

Now we can call similar methods on our **file** object without continually having to supply the file path.

```
if(file.Exists) file.CopyTo(outputPath, true);
```

A lot of the static methods on the **File** class also work on the **file** object. We can also get a bit more information, such as:

```
var file = new FileInfo(inputFilePath);
Console.WriteLine("File length: " + file.Length + " bytes");
Console.WriteLine("File ext: " + file.Extension);
Console.WriteLine("File last written: " + file.LastWriteTime);
```

Manipulating Directories

Manipulating Directories

- The **Directory** class provides static members

```
Directory.Delete(...);
bool exists = Directory.Exists(...);
string[] files = Directory.GetFiles(...);
```

- The **DirectoryInfo** class provides instance members

```
DirectoryInfo directory = new DirectoryInfo(...);
...
string path = directory.FullName;
bool exists = directory.Exists;
FileInfo[] files = directory.GetFiles();
```

Applications typically need to manipulate directories. For example, you might need to create a working directory structure on application startup to cache settings during operation, and then on shutdown delete any resources the application created.

In the same way as the **File** class works with files, the **Directory** class provides various static methods that enable you to directly manipulate directories. For example, we can create a new directory using the **Directory.CreateDirectory(directoryPath)** static method:

```
string directoryPath = @"..\..\..\temp";
string subdirectoryPath = directoryPath + @"\sub";
Directory.CreateDirectory(directoryPath);
Directory.CreateDirectory(subdirectoryPath);
```

This will create a directory hierarchy that we might then start populating with temporary files as our application does its work. Eventually we'll probably want to clean up. We can delete a

directory using the `Delete` method, but if we try to delete the temporary directory created above:

```
Directory.Delete(directoryPath);
```

we'll get a `System.IO.IOException` with the message 'The directory is not empty'. This is because there's a subdirectory 'sub'. We could delete sub and any files we'd put in the directory structure before attempting to call the `Delete` method. But we can also force everything to be deleted because `Delete` has an optional second argument 'recursive' that's a Boolean:

```
Directory.Delete(directoryPath, true);
```

This will delete our temporary directory and everything within it. This is a powerful option, and should be used with care.

If you want to see what's in the directory you can use the `GetDirectories` method and get an array of strings containing the subdirectory names. If you want a list of files, there's a `GetFiles` method.

You can also use the various instance members provided by the `DirectoryInfo` class, which works in the same way as the `FileInfo` class. It's usually requires less code when you have to do a number of operations on a directory structure. For example we could rewrite the preceding code, and improve on it to avoid an error if we already deleted the directory:

```
var dir = new DirectoryInfo(directoryPath);
if(dir.Exists) dir.Delete(true);
```

Notice that we no longer need to specify the path in the `Delete` method arguments. There's just the single argument for `recursive`. If you're sure the directory is empty, you could just call:

```
dir.Delete();
```

If you want to retrieve more information, you can also get this from the `DirectoryInfo` object:

```
var dir = new DirectoryInfo(directoryPath);
Console.WriteLine("Directory name: " + dir.FullName);
Console.WriteLine("Parent: " + dir.Parent);
Console.WriteLine("Directory last written: " +
dir.LastWriteTime);
```

Question: You want to read the contents of a text file into an array of strings. Which static method of the **File** class should you use?

- A: ReadAllText
- B: ReadAllLines
- C: ReadAllBytes
- D: WriteAllText

Answer: B, `File.ReadAllText(path)`

Manipulating File and Directory Paths

Manipulating File and Directory Paths

- The **Path** class encapsulates file system utility functions

```
string settingsPath = "..could be anything here..";  
  
// Check to see if path has an extension.  
bool hasExtension = Path.HasExtension(settingsPath);  
  
...  
  
// Get the extension from the path.  
string pathExt = Path.GetExtension(settingsPath);  
  
...  
  
// Get path to temp file.  
string tempPath = Path.GetTempFileName();  
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```

You might find yourself doing all kinds of string manipulation to figure out the directory of a file, its file extension, especially when the name of an output file is derived in some way from that of an input file. You can save yourself a lot of time by simply using the built-in **Path** class. It encapsulates a variety of utility type I/O functions that are very useful when working with the file system. These functions can help your application if it needs to write a file to a temporary location, get an element from a file system path, or create a random file name.

In the previous section we created a temporary directory called ‘temp’ in the project folder. A much better way is to use the **Path.GetTempPath** method. This will automatically give you a path for a temporary directory in the appropriate place, depending on the operating system and environment:

```
var directoryPath = Path.GetTempPath();  
var dir = new DirectoryInfo(directoryPath);
```

```
Console.WriteLine("Directory name: " + dir.FullName);
```

This will output the directory location, in this case on a Windows 10 machine:

```
Directory name: C:\Users\BillA\AppData\Local\Temp\
```

It will of course be different on your machine. There are many other utility methods for getting file extensions, paths, filenames and creating temporary file names:

```
string tempPath = Path.GetTempFileName();  
var file = new FileInfo(tempPath);  
Console.WriteLine("File name: " + file.FullName);
```

From the output, we can see the file that was created (and it will be different every time we run the code):

```
File name: C:\Users\BillA\AppData\Local\Temp\tmp299C.tmp
```

For more information on the Path class, see <https://aka.gd/3GbGpXI>.

Question: True or False: The Path.HasExtension(path) method will return true if path has an extension?

Answer: True, it will return true.

Lesson 2 – Serialization and Deserialization

We often need to save and restore .NET objects. **Serialization** is the process of converting a data structure into a serial form, usually as some kind of text format, that can be stored or sent from one place to another. **Deserialization** is the reverse process of turning data that has been serialized back into objects.

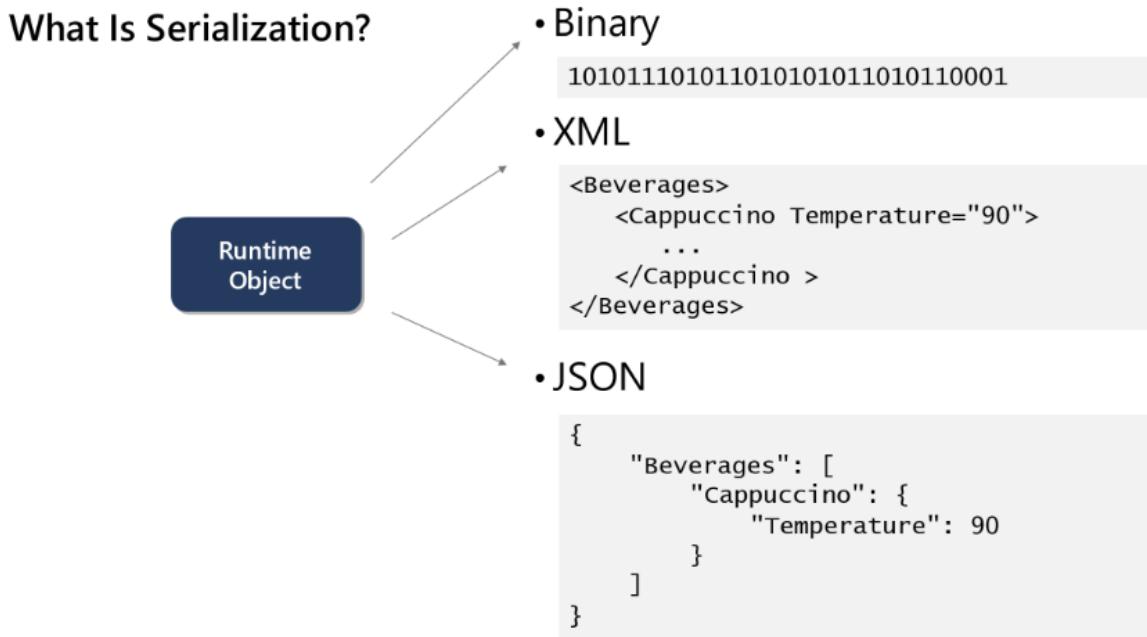
This lesson shows how to serialize objects to binary, XML, and JavaScript Object Notation (JSON), and how to make a custom serializer that allows you to serialize objects into any format you want.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand serialization
- Identify supporting .NET framework classes.
- Serialize and deserialize .NET objects.
- Use XML and JSON serialization formats.

What Is Serialization?



When we create objects in a .NET language like C#, they are stored in memory as binary data. When our applications terminate, all the data in memory is released. If we want the objects to persist between invocations of our applications, it needs to be saved to a storage device such as a disk. If we want to send those objects to another process such as a web service, it has to be sent over a network connection using a protocol such as HTTP. In either case the object needs to be converted into something that can be saved, either as binary data, or increasingly as some kind of text format that is easy to retrieve and process.

In the past, most applications used a binary file format, which is usually very efficient because it matches the internal memory structure and is usually quite compact. The problem with this is that it is not very portable, and can be difficult to use unless you have the original application that created it. Another problem is that binary data can be misinterpreted

depending on the storage device or communications medium. A more modern approach is to use some kind of text format. Initially, XML (extensible markup language) was a popular choice, but in more recent years the JSON format has become more popular. Both are supported by .NET libraries.

By serialising data in the XML format, you are using an open standard that can be read by any application, no matter what platform it runs on. Since serialised data is formatted with XML constructs, the format is more verbose than the binary format. This makes the XML format less efficient than ‘raw’ binary data. But because XML is a plain text and free-form language, it makes it possible for messages to be sent across different platforms and applications. XML also is ‘self-describing’ and you can strictly enforce schemas if needed. As long as the sender and receiver agree on a known schema, they can both send and receive messages and convert them to the right model in any environment. XML serialisation was often used to serialise data that can be sent to and from web services via SOAP (Simple Object Access Protocol). But because SOAP is so verbose and strict, it has fallen out of favour, and is increasingly only found in legacy environments. SOAP has been overtaken in popularity by REST (Representational State Transfer) services that make greater use of the underlying HTTP protocol for security and remote procedure calls.

The XML format is a ‘markup’ language, like HTML, that uses angle brackets to denote tags with attributes. It is a language built on the SGML (Simple Generalized Markup Language) standard, which is the same standard on which HTML is built. An example fragment of XML is shown below:

```
<Beverages>
<Cappuccino Temperature="90">
...
</Cappuccino >
</Beverages>
```

Note: to learn more about the XML standards, see: <https://aka.gd/3a2ji5u>.

The JSON format is more lightweight than XML, and is based on the JavaScript object literal format. This makes it well suited to web development because client-side JavaScript is natively able to serialize and deserialize it. People often use JSON to move data when using AJAX (Asynchronous JavaScript and XML) calls and messages between web client applications and their webservices, for example to create dynamic web pages. This is slightly ironic, since the name AJAX specifically refers to XML, which is because the establishment of the AJAX development pattern preceded the popularity of JSON.

JSON is a simple text format that is relatively easy for people to read, and even write, and for machines to process. It also has a less verbose syntax than XML. Because of these benefits,

JSON is now the most common language for sending data and the de-facto standard in the business world.

JSON uses curly braces to enclose objects and square brackets for arrays. Property names are separated from values by a colon, and multiple properties or array elements are separated by commas. The main syntax difference from the use of object literals in JavaScript code is the need to use quotation marks for property names.

```
{  
  "Beverages": [  
    "Cappuccino": {  
      "Temperature": 90  
    }  
  ]  
}
```

Serializing Objects as JSON

Serializing Objects as JSON

```
var beverage = new Beverage(90, 150, "Columbia");  
var options = new JsonSerializerOptions();  
options.WriteIndented = true;  
var json = JsonSerializer.Serialize(beverage, options);  
File.WriteAllText(@"..\..\..\Beverage.json", json);
```

```
{  
  "Temperature": 90,  
  "Size": 150,  
  "CountryOfOrigin": "Columbia"  
}
```

There is nothing to stop you writing your own serialization code to convert objects in memory into strings or binary arrays, if you have time on your hands. But this is unnecessary because the .NET Framework provides the various namespaces and classes to support serialization. Let's start with a simplified version of our Beverage class:

```
public class Beverage
{
    public double Temperature { get; set; } // deg C
    public double Size { get; set; } // ml
    public string? Countryoforigin { get; set; }
    public Beverage() { }
    public Beverage(double temp, double size, string country)
    {
        Temperature = temp;
        Size = size;
        Countryoforigin = country;
    }
}
```

Converting an instance of this class to a JSON string can be done using the **JsonSerializer** from the **System.Text.Json** namespace. This class has some static methods that we can use to serialize an instance of Beverage:

```
using System.Text.Json;
var beverage = new Beverage(90, 150, "columbia");
var json = JsonSerializer.Serialize(beverage);
Console.WriteLine(json);
```

This will produce the output:

```
{"Temperature":90,"Size":150,"Countryoforigin":"columbia"}
```

This is valid minified JSON, i.e. it has no whitespace, so it's fairly compact and good for machine-to-machine communication, but not especially readable to a human. If you don't like the format of that JSON we do have a degree of control over it by means of a second parameter we can pass to the **Serialize** method. First we have to set up the **JsonSerializerOptions** object, then set the **WriteIndented** method to **true**:

```
using System.Text.Json;
```

```
var beverage = new Beverage(90, 150, "columbia");
var options = new JsonSerializerOptions();
options.WriteIndented = true;
var json = JsonSerializer.Serialize(beverage, options);
Console.WriteLine(json);
```

The resulting output is a little easier to read, and more like what you might see in a text book:

```
{
    "Temperature": 90,
    "Size": 150,
    "Countryoforigin": "Columbia"
}
```

We have a number of options for controlling the output, although if we want complete control, for example over the names of properties in the serialized form, then we need to do a bit more work, as we'll see later in this lesson.

Sending this as output to the console probably isn't that useful. A more practical option would be:

```
var json = JsonSerializer.Serialize(beverage);
File.WriteAllText(@"..\..\..\Beverage.json", json);
```

By default, the JsonSerializer will save all the public properties and ignore any fields (member variables). You can change this behaviour by using attributes from the System.Text.Json.Serialization namespace:

```
using System.Text.Json.Serialization;
public class Beverage
{
    [JsonIgnore]
    public string? _temp_string { get; set; }
    [JsonInclude]
```

```
public double importantNumber;  
public double Temperature { get; set; } // deg C  
public double size { get; set; } // ml  
public string? Countryoforigin { get; set; }  
...  
}
```

If we run the serializer again, we get:

```
{  
    "Temperature": 90,  
    "Size": 150,  
    "Countryoforigin": "Columbia",  
    "importantNumber": 0  
}
```

Here we can see that **importantNumber** has been saved, even though it's a field, and that **_temp_string** has been ignored, even though it's a public property.

Deserializing JSON

Deserializing JSON

```
{  
    "Temperature": 90,  
    "Size": 150,  
    "CountryOfOrigin": "Columbia"  
}
```

```
var json = File.ReadAllText(@"..\..\..\Beverage.json");  
var beverage = JsonSerializer.Deserialize <Beverage>(json);  
Console.WriteLine(beverage.Countryoforigin);
```

We have a file, `beverage.json`, that contains a serialized Beverage object. We can now write another program to pick up the serialized object from the filesystem and ‘rehydrate’ it into memory.

If you look again at the serialized JSON you might notice that nowhere does it say that the object is a Beverage instance. If we want to deserialize it, the question is; how will the JsonSerializer know what type to create? Serialization is pretty easy because the JsonSerializer can look at the object we pass it and, using reflection, work out what it is, what properties it has and so forth. When we go in the other direction we’re going to have to tell the JsonSerializer what type of object we are trying to rehydrate.

We can do this either by passing it a type, or using generics. We can pass the **Deserialize** method a type as the second parameter, which will tell it what to create. **Deserialize** returns an **Object**, so we’ll also need to do a cast to type **Beverage**, which we can do using the **as** keyword:

```
var json = File.ReadAllText(@"..\..\..\Beverage.json");
var beverage = JsonSerializer.Deserialize(json, typeof(Beverage))
as Beverage;
Console.WriteLine(beverage.Countryoforigin);
```

We’ve printed out something from `beverage` to confirm that we got the correct object back into memory.

A more elegant way is to use the generic approach, which returns a strongly typed object and thus avoids the need for the cast:

```
var json = File.ReadAllText(@"..\..\..\Beverage.json");
var beverage = JsonSerializer.Deserialize<Beverage>(json);
Console.WriteLine(beverage.Countryoforigin);
```

Using generics means that we are effectively building a special Beverage deserializer method.

The majority of JSON serialization and deserialization can be handled using the `System.Text.Json.JsonSerializer` class, but it doesn’t handle everything. For example, it does a good job of dealing with collections and other nested objects, but there are occasions where you have to use custom converters.

To learn more about JSON serialization, see <https://aka.ms/3Pv0coW>.

Serializing Objects as Binary

Serializing Objects as Binary

- Serialize as binary using `BinaryFormatter.Serialize` method.
- Deserialize from binary using `BinaryFormatter.Deserialize` method
- **No longer recommended due to security issues**

We often need to save the state of an object to a persistent storage so that it can be retrieved later. If storage and efficiency are of paramount importance, then using a binary format similar to in-memory storage is an attractive option. But it should be remembered that binary serialization comes with risks, because the format isn't human readable. This means if something goes wrong, it will be very difficult to diagnose and fix the problem. It typically means that you need the exact same application code to be available, or data may be very difficult to retrieve. But perhaps the biggest danger is security, because you have to assume that the data you are deserializing is trustworthy.

Deserialization presents a vulnerable attack surface if data or request payloads are not secured. If an attacker is able to intercept and manipulate binary data, they can cause a denial of service (DoS) or information leaks, or run code inside the app from a remote location. This is a common risk category that frequently appears in the OWASP Top 10 list of security risks.

The .NET framework has classes for binary serialization, but they are no longer recommended for new application development because of these security risks. They were designed before deserialization vulnerabilities were fully understood, and there isn't really any satisfactory way of mitigating these risks.

Note: If you need to work with binary serialization in .NET, for example to maintain legacy code, then you can find more information at: <https://aka.gd/3G22GqF>.

Serializing Objects as XML

Serializing Objects as XML

- Serialize as XML

```
var beverage = new Beverage(90, 150, "Columbia");
XmlSerializer ser = new XmlSerializer(typeof(Beverage));
TextWriter writer = new StreamWriter(@"..\..\..\Beverage.xml");
ser.Serialize(writer, beverage);
writer.Close();
```

- Deserialize from XML

```
TextReader reader = new StreamReader(@"..\..\..\Beverage.xml");
var beverage2 = ser.Deserialize(reader) as Beverage;
reader.Close();
Console.WriteLine(beverage.CountryOfOrigin)
```

We can serialize our Beverage class as in XML syntax by using the **XmlSerializer** class in the **System.Xml.Serialization** namespace. First we need to create an **XmlSerializer** object and pass it the type of the object we want to serialize. The XML serializer isn't able to deduce the type of the object we're passing it on the fly, like the JSON serializer. Then we can call the **Serialize** method of **XmlSerializer**. This method takes a stream object (more about these in the next section) and the object we want to serialize:

```
var beverage = new Beverage(90, 150, "columbia");
ser = new XmlSerializer(typeof(Beverage));
TextWriter writer = new StreamWriter(@"..\..\..\Beverage.xml");
ser.Serialize(writer, beverage);
writer.Close();
```

If you look at **Beverage.xml** you'll see the output is a single string; a sort of minified XML. That's good for sending data to another process, but you might want to make it human readable. You could look for some option on the **XmlSerializer**; but you would look in vain. That's actually the job of the **TextWriter**, and we can improve the output format by replacing it with the more specialized **XmlTextWriter** and configuring it. We can also set an empty default XML namespace to prevent some of the default verbosity on the root element:

```
using System.Text;
using System.Xml;
using System.Xml.Serialization;
```

```
var beverage = new Beverage(90, 150, "Columbia");
XmlSerializer ser = new XmlSerializer(typeof(Beverage));
var writer = new XmlTextWriter(@"..\..\..\Beverage.xml",
Encoding.UTF8);
writer.Formatting = Formatting.Indented;
var xns = new XmlSerializerNamespaces();
xns.Add("", "");
ser.Serialize(writer, beverage, xns);
writer.Close();
```

Notice that we needed to add a couple more usings to add more .NET namespaces. The result of running this code is a file containing the following XML.

```
<?xml version="1.0" encoding="utf-8"?>
<Beverage>
<Temperature>90</Temperature>
<Size>150</Size>
<CountryofOrigin>Columbia</CountryofOrigin>
</Beverage>
```

One thing you might notice is that everything is an element. If you want, say, temperature and size to be XML attributes of the Beverage element, than you can do that by adding C# attributes to the Beverage class members. You'll need to add the **System.Xml.Serialization** namespace, and then you'll be able to use the **[XmlAttribute]** on the members as required:

```
using System.Xml.Serialization;

public class Beverage
{
    [XmlAttribute]
    public double Temperature { get; set; } // deg C
    [XmlAttribute]
    public double Size { get; set; } // ml
```

```
public string? Countryoforigin { get; set; }

...
}
```

Now the generated XML will be:

```
<?xml version="1.0" encoding="utf-8"?>
<Beverage Temperature="90" Size="150">
<Countryoforigin>Columbia</Countryoforigin>
</Beverage>
```

There are many other ways you can customize the way the XML is generated. We can now try reading in the serialized data from the file we generated, and make sure that our object is reconstituted as expected:

```
TextReader reader = new StreamReader(@"..\..\..\Beverage.xml");
var beverage2 = ser.Deserialize(reader) as Beverage;
reader.Close();
Console.WriteLine(beverage2.Temperature);
```

Notice that we don't have the option of strong typing or use of generics with the XmlSerializer class. We have to cast it to a Beverage object, which is reasonable because we told the XmlSerializer what the type was. If the the XML isn't a Beverage, you'll get an exception, probably a System.InvalidOperationException from the parser. If it's just that some elements or attributes are missing, the Deserialize method will probably return, but some of the values will be null.

Creating a Custom Serializable Type

Creating a Custom Serializable Type

- Implement the `ISerializable` interface

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    ...
    public ServiceConfiguration(
        SerializationInfo info, StreamingContext ctxt)
    {
        ...
    }

    public void GetObjectData(
        SerializationInfo info, StreamingContext context)
    {
        ...
    }
}
```

The .NET Framework provides the `System.Runtime.Serialization` namespace which provide classes to support serialization. Binary serialization is no longer recommended, but you can use this namespace if you want to implement a custom formatter, or you find yourself working with some legacy code.

To make one of your own types serializable, you need to do the following:

- Ensure that you've defined a default constructor.
- Decorate the class with the `[Serializable]` attribute.
- Implement the `ISerializable` interface.
- Define a deserialization constructor.
- Define the members that you want to serialize.

Let's look at our `Beverage` class, and consider how we can might make it serializable in a more custom way by implementing the `ISerializable` interface:

```
using System.Runtime.Serialization;

[Serializable]
public class Beverage : ISerializable
{
    [NonSerialized]
```

```
private string? _temp_string;
public double Temperature { get; set; } // deg C
public double Size { get; set; } // ml
public string? CountryOfOrigin { get; set; }
public Beverage() { }
public Beverage(double temp, double size, string country)
{
    Temperature = temp;
    Size = size;
    CountryOfOrigin = country;
}
public Beverage(SerializationInfo info, StreamingContext context)
{
    Temperature = info.GetDouble("Temp");
    Size = info.GetDouble("Size");
    CountryOfOrigin = info.GetString("Country");
}
public void GetObjectData(SerializationInfo info,
    StreamingContext context)
{
    info.Addvalue("Temp", Temperature);
    info.Addvalue("Size", Size);
    info.Addvalue("Country", CountryOfOrigin);
}
```

You can see in this code that we have added a using statement for the **System.Runtime.Serialization** namespace, because this isn't one of the namespaces we get automatically. We then add the **[Serialization]** attribute to the class (more about attributes later in the course), and we add the **ISerializable** interface. As soon as you add this

interface you'll start to get errors because you now need to implement **GetObjectData**, which we'll discuss in a moment.

We have a made-up example of a private variable that is really just there for utility, called `_temp_string`. It isn't necessary to define an instance of the Beverage class, so we added the **[NonSerialized]** attribute to prevent it being output as part of the serialization. This is a good practice because you probably don't want to serialize everything in your class as it would be inefficient and make the serialized form unnecessarily verbose. The serialized form of an object should contain enough information to recreate the class in memory, but no more.

We already have a default constructor, but we've added an additional constructor which takes a **SerializationInfo** and a **StreamingContext** object. We need this constructor if we want to be able to 'rehydrate' the object from its serialized form. In this case we don't actually make use of the **StreamingContext** object.

Finally we need to implement **GetObjectData** for this to compile. **GetObjectData** has the same parameter signature as the deserialization constructor, taking a **SerializationInfo** and a **StreamingContext** object. Again, we don't make use of the **StreamingContext** in this particular case, but we use the **SerializationInfo** object to add the properties that will be serialized.

Now that our Beverage class implements the **ISerializable** interface, it can be used by various .NET classes to serialize and deserialize beverages in various formats.

To learn more about custom serialization, see: <https://aka.gd/3wIMQwy>.

Creating a Custom Serializer

Creating a Custom Serializer

- Implement the `IFormatter` interface

```
class IniFormatter : IFormatter
{
    public ISurrogateSelector SurrogateSelector { get; set; }
    public SerializationBinder Binder { get; set; }
    public StreamingContext Context { get; set; }

    public object Deserialize(Stream serializationStream)
    {
        ...
    }

    public void Serialize(Stream serializationStream, object graph)
    {
        ...
    }
}
```

Suppose you want to turn your data into some obscure application-specific CSV or INI file. One way of doing this is to create your own implementation of the `IFormatter` interface in the `System.Runtime.Serialization` namespace.

First, you'll need to define a class that implements the `IFormatter` interface. This means providing the properties `SurrogateSelector`, `Binder`, and `Context`. You will also need to implement the `Deserialize` and `Serialize` methods.

To find out more about the `IFormatter` interface, see: <https://aka.ms/3LsELS5>.

Question: True or False: you can use the `XmlSerializer` class to save an object in JSON format?

Answer: False, use the `JsonSerializer` in the `System.Text.Json` namespace.

Lesson 3 - Streams

Working with data in memory is very fast, but volatile. When your application terminates, you usually need to have a way of saving data so that the next time the application runs, the user can pick up where they left off. Worse, if your application ends unexpectedly, e.g. due to a bug or hardware fault, you need to have a way of persisting data periodically to minimize data loss. There are also situations where the amount of data is so large that it can't be held in memory. In all these situations you need to be able to send and receive data from some

kind of storage device, usually the file system, or some remote service such as a web server that's accessible over HTTP.

The purpose of this lesson is to introduce streams a an efficient way of reading and writing files.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the purpose of streams.
- Make use of streams in your application.
- Describe the different kinds of streams.

What are Streams?

What are Streams?

- The `System.IO` namespace contains a number of stream classes, including:
 - The abstract `Stream` base class
 - The `FileStream` class
 - The `MemoryStream` class
- Typical stream operations include:
 - Reading chunks of data from a stream
 - Writing chunks of data to a stream
 - Querying the position of the stream

A stream represents a series of bytes that can come from a file on the file system, a network connection, or memory. Streams let you read or write small packets of data from or to a data source. We've already seen a number of methods on the `File` object that allow you to read the contents of a file or write out to a file. Streams offer a lower-level set of functions that give you more control over this process. Streams allow you to read or write data in chunks, typically in the form of strings or byte arrays. They can also work through a file or other resource and keep track of the current position, giving you the ability to read and write part of a file without having to have a memory representation of it in its entirety.

In .NET there are a number of stream classes, derived from a base **Stream** class that provides the common capabilities that the various stream classes provide. The different stream classes work with different categories of data sources such as binary data, text, local filesystems or network connections. The stream classes are located in the **System.IO** namespace. You can't instantiate a **Stream** object directly; it is an abstract base class. Instead, you create a **FileStream**, **MemoryStream**, **NetworkStream**, etc.

When you create a stream class and connect it to a data source, you will get a pointer that is at the beginning of the file or stream of data. When you read or write data, the pointer will maintain the current position in the stream.

Types of Streams in the .NET Framework

Types of Streams in the .NET Framework

- Classes that enable access to data sources include:

Class	Description
FileStream	Exposes a stream to a file on the file system.
MemoryStream	Exposes a stream to a memory location.
NetworkStream	Exposes a stream to a network location.

- Classes that enable reading and writing to and from data source streams include:

Class	Description
StreamReader	Read textual data from a source stream.
StreamWriter	Write textual data to a source stream.
BinaryReader	Read binary data from a source stream.
BinaryWriter	Write binary data to a source stream.

You can use the .NET stream classes to read and write different kinds of data from different kinds of data sources. When you use a **FileStream**, **MemoryStream**, or **NetworkStream** object to set up a stream, it is just a raw list of bytes, even if it's text. Typically the data has some structure that you must parse or otherwise extract to turn these bytes into useful data. This kind of code can be very tedious and onerous to write, and coding errors are common. But .NET has classes that you can use to read and write text data and primitive types in a stream that you opened with the **FileStream**, **MemoryStream**, or **NetworkStream** classes. In the table below, some of the stream reader and writer classes are shown along with the underlying stream classes:

Class	Application
-------	-------------

FileStream	Create a stream that connects to the file system, including opening and closing files, and giving raw access to the file system as a byte sequence.
MemoryStream	Create a stream that connects to a location in memory, including accessing and allocating memory, and giving raw access to memory as a byte sequence.
NetworkStream	Create a stream that connects to a network location, including opening and closing the connection and giving raw access to the network resource as a byte sequence.
StreamReader	Reading text data from the underlying stream object.
StreamWriter	Writing text data to an underlying stream object.
BinaryReader	Reading binary data from the underlying stream object.
BinaryWriter	Writing binary data to an underlying stream object.

Reading and Writing Binary Data by Using Streams

Reading and Writing Binary Data by Using Streams

- You can use the `BinaryReader` and `BinaryWriter` classes to stream binary data

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";
// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath);

// BinaryReader object exposes read operations on the underlying
// FileStream object.
BinaryReader reader = new BinaryReader(file);

// BinaryWriter object exposes write operations on the underlying
// FileStream object.
BinaryWriter writer = new BinaryWriter(file);
```

Software applications used to commonly store data in binary form. The advantage of speed and efficiency were compelling; by storing the data in the same binary format that it occupied in memory, you save on time converting to a text format, and it will almost certainly

require less disk space or network bandwidth. But, as mentioned before, there's a serious drawback to this approach that ultimately centres around the lack of human readability. It means that data is very difficult to recover if the original application that wrote it, possibly the exact version, is not available. Binary data formats can also vary, even for the same application, with different hardware or operating systems. But a much bigger issue is the security risk. Binary data has become a serious security issue because of the difficulty of validation and other factors, and there has been a move towards acceptance of the loss of efficiency with text formats, particularly as storage has become less expensive.

If you do want to store binary data you can still use the `BinaryReader` and `BinaryWriter` classes. You need to create a reader or writer object, and supply a stream that is connected to your data. In the following example we'll use a `FileStream` object to connect to both an existing file and create a new one:

```
string filePath = @"..\..\..\Beverage.xml";
FileStream input = new FileStream(filePath, FileMode.Open);
BinaryReader reader = new BinaryReader(input);
var length = reader.BaseStream.Length;
var datablock = new byte[length];
int nByte, pos=0;
while ((nByte = reader.Read()) != -1)
{
    // Set the value at the next index.
    datablock[pos] = (byte)nByte;
    // Advance our position variable.
    pos += sizeof(byte);
}
reader.Close();
input.Close();
```

Compared to some of the File I/O code we looked at earlier in the lesson, this is quite a bit more involved, but we get much lower-level control over the process. Inside the loop we're reading from the text file we created earlier, but treating it as a binary file, and reading in one byte at a time. When the `Read` method returns a value of `-1` we stop the read process. Notice that some of the properties of the reader are available on the `BaseStream` class. This is how we get the length of the file, or rather the stream. The `BinaryReader` class has numerous

other methods and properties to enable you to get random access to file and read parts of it, as needed.

Writing binary data is a little simpler:

```
// Collection of bytes to be written  
byte[] dataCollection = { 121, 74, 106, 7, 122, 233, 226, 198,  
82, 10 };  
  
FileStream output = new FileStream(@"..\..\..\Test.bin",  
FileMode.OpenOrCreate);  
  
BinaryWriter writer = new BinaryWriter(output);  
  
// Write each byte to stream.  
  
foreach (byte data in dataCollection) writer.Write(data);  
  
writer.Close();  
  
output.Close();
```

This code will create a file Test.bin with the data inside it. You can view it using the binary editor in Visual Studio.

Reading and Writing Text Data by Using Streams

Reading and Writing Text Data by Using Streams

- You can use the `StreamReader` and `StreamWriter` classes to stream plain text

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";  
  
// Underlying stream to file on the file system.  
FileStream file = new FileStream(filePath);  
  
// StreamReader object exposes read operations on the underlying  
// FileStream object.  
StreamReader reader = new StreamReader(file);  
  
// StreamWriter object exposes write operations on the underlying  
// FileStream object.  
StreamWriter writer = new StreamWriter(file);
```

You can avoid most of the problems with binary files, at the expense of efficiency and storage or bandwidth, by using text formats of one kind or another. We've already seen that there are built-in .NET classes for saving and retrieving objects in JSON and XML. In this section we'll look at some of the lower-level ways of working with text streams.

Reading text data works in much the same way as reading binary data. We can open the Beverage.xml file and just treat it as a text file. The only difference to the code in the preceding section is that we now use a **StreamReader** instead of a **BinaryReader**:

```
string filePath = @"..\..\..\Beverage.xml";
FileStream input = new FileStream(filePath, FileMode.Open);
StreamReader reader = new StreamReader(input);
string text = "";
while (reader.Peek() != -1) text += (char)reader.Read();
reader.Close();
input.Close();
Console.WriteLine(text);
```

Compared to dealing with binary files, working with text is relatively easy, and also easier to debug. Notice that `reader.Read()` returns an `int`, which is why we did a cast to `char`. If we don't do that, we'll output a string of decimal numbers (corresponding to the ASCII code for the letters) rather than the fragment of XML that was in the file.

Note: for simplicity we have appended characters to a string object. This is actually very inefficient, and where you are repeatedly appending to a string the best practice is to use the `StringBuilder` class.

This final rather trivial code example shows how to use the **FileStream** and **StreamWriter** classes to write a string to a new file on the file system, or overwrite it if the file already exists:

```
string message = "This is a very important message!";
FileStream output = new FileStream(@"..\..\..\Test.txt",
FileMode.OpenOrCreate);
StreamWriter writer = new StreamWriter(output);
writer.Write(message);
writer.Close();
```

```
output.Close();
```

To find out more about the text stream classes in .NET, see: <https://aka.ms/3ltCw6B>.

Lab: Generating the Grades Report

Lab: Generating the Grades Report

Lab scenario

You have been asked to upgrade the Grades Prototype application to enable users to save a student's grades as an XML file on the local disk. The user should be able to click a new button on the StudentProfile view that asks the user where they would like to save the file, displays a preview of the data to the user, and asks the user to confirm that they wish to save the file to disk. If they do, the application should save the grade data in XML format in the location that the user specified.

Objectives

Exercise 1:
Serializing Data for the
Grades Report as XML

Exercise 2:
Previewing the Grades
Report

Exercise 3:
Persisting the Serialized
Grade Data to a File

Module Review and Takeaways

Review Questions

Question: You need to transfer files of up to 100 GB in size from one location to another location on disk. You need to minimize the amount of memory used by the application. Which classes should you use to read and write the files?

- A: BinaryReader and BinaryWriter
- B: FileStream, StreamReader and StreamWriter
- C: MemoryStream, StreamReader and StreamWriter
- D: FileStream, BinaryReader and BinaryWriter

Answer: D, FileStream for file system access, BinaryReader and BinaryWriter to read and write bytes in the video file.

Module 7 – Database Access

Module Overview

A frequent part of almost any business software project is to connect to some kind of database. Whilst you can write your own database access code, most developers find that some kind of object-relational mapping tooling greatly speeds up development against relational databases. These provide or generate entity data models that give you classes that you can work with, rather than having to manage the low-level retrieval and updating of data. In addition, this approach unlocks the capability of Language-Integrated Query (LINQ), which gives a syntax for database queries that is similar to SQL (Structured Query Language).

Objectives

After completing this module, you'll be able to:

- Use Entity Framework.
- Construct queries using LINQ.

Lesson 1 – Entity Framework

Creating applications that let users access data has always been rather repetitive, and error-prone. The data access code usually involves a lot of “plumbing” that is repetitive, and unrelated to the problem domain. They often involve writing queries that are text strings, and therefore don't have any of the rigorous type-checking that we get in the rest of our code. To the compiler, a string is a string. The results of queries are also typically returned as untyped objects.

For many years, database developers have used one of the many ORM tools that are available. Microsoft has its own, called Entity Framework, which is built on top of Microsoft's data access layer, ADO.NET. In its latest incarnation, it is designed as a cross-platform .NET 6 solution called Entity Framework Core. Like other ORM products, Entity Framework makes it easier to build apps that access data.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand Entity Framework.
- Use the Entity Data Model tooling of ADO.NET.
- Use Entity Framework to generate wrapper classes.
- Perform create, read, update and delete operations using EF.

What Is Entity Framework?

What is Entity Framework?

- The ADO.NET Entity Framework provides:
 - EDMs
 - Entity SQL
 - Object Services
- The ADO.NET Entity Framework supports:
 - Writing code against a conceptual model
 - Easy updating of applications to a different data source
 - Writing code that is independent from the storage system
 - Writing data access code that supports compiletime type-checking and syntax-checking

The main function of Entity Framework is to act as a bridge between the world of relational databases and the various SQL-type languages, and the world of object-oriented programming. With Entity Framework, the conceptual entity data model (EDM) that you work with, represents a business view of the data, implemented with strongly typed .NET objects. It describes business entities and relationships that are then mapped to the data source's underlying data model. Entities can be things like employees, jobs, customers, products, orders and branch locations in a business application. Entities are linked by relationships, for example the connection between an order and a customer.

Entity framework has the ability to use the conventional “database-first” approach, where entity classes are generated to align with the database tables. These are POCO classes (plain old CLR objects) that don't require any special base classes or interfaces. EF also offers a “code-first” approach where a data table is generated to map to the pre-built classes.

Once you have entities that map to your database tables, you can use them to write code against your conceptual model. This model is able to capture the relationships in the underlying data model, as well as being able to use inheritance to create derived classes. At the same time you are able to use classes that are strongly-typed and syntax-checked by the

compiler, rather than having to worry in great detail about the underlying database technology being used. There is also tooling available to keep the model and database in synchronization as they develop, without having to start again.

Note: To learn more about EF Core, see: <https://aka.ms/3PypC56>.

Entity Framework Core, or EF Core, is a complete re-write of the original Entity Framework, which is still available as Entity Framework 6 (or EF6). The latter has a few features, and supported databases, that are still not available in EF Core. But the current advice is to use EF Core for any new EF applications, unless you have a need for one of these features. This is because most of the future investment by Microsoft is going to be in EF Core. It's also worth noting that the latest versions of EF Core only run on .NET Core, and not the legacy versions of the .NET Framework. EF Core has many new features that were not available in earlier versions of EF, such as the ability to work with non-relational data.

On the other hand, if you are working on a legacy EF application then you will probably want to update to the latest EF6 version, rather than try to migrate it to EF Core. EF6 is a stable and supported product, but you shouldn't expect to see any development of significant new features.

To compare the features of EF Core and EF6, see: <https://aka.ms/3PB4afS>

Code-first with Entity Framework Core

Code-first with Entity Framework Core

- Create a POCO class
- Create a DbContext class
- Implement the OnConfiguring method to set data source
- Create instance of DbContext and call EnsureCreated()
- Use DbContext to work with data

```
using var db = new BeverageContext();
db.Database.EnsureCreated();

foreach(var bev in db.Beverages)
    Console.WriteLine(bev.ID + " " + bev.Name);
```

Visual Studio tooling supports a code-first design strategy, where you create the entities in code, and then use the tooling to provision database tables based on the code entities. This

is a developer-centric approach, because you can start with a model that captures your business problem and business entities. Obviously, this is only possible when developing a ‘green-field’ application. In many real-life situations you are obliged to work with an existing data source, and work from there.

Note: In earlier versions of EF there was a graphical database designer; the “design-first” option. This is not supported in EF Core, and Microsoft have not indicated any intention to add such a feature.

Let’s start with our Beverage class, and again, we’re going to keep it simple. We’ll keep the three fields from before, but we’re going to add an integer field called ID. We’ll see why in a minute. We’ll use the default constructor, and it’s not going to be an abstract class so that we can create Beverage instances.

```
public class Beverage
{
    public int ID { get; set; }
    public string Name { get; set; }
    public double Temperature { get; set; } // deg C
    public double Size { get; set; } // ml
}
```

The point to note here is that there’s nothing special about this class. It isn’t derived from some special database class or have to implement some particular interface. When we instantiate it we’re going to get a “plain old CLR object” or POCO. These objects are ultimately going to map to rows in a database table.

Using Entity Framework, getting to that point is surprisingly easy. First we need to make sure that we have the right NuGet packages installed. You can go to the Visual Studio menu, and use the **Tools->NuGet package manager->Manage NuGet packages for solution...** option to open the package manager, and then look for **Microsoft.EntityFrameworkCore** and **Microsoft.EntityFrameworkCore.Sqlite**. Alternatively you could use the **Tools->NuGet package manager->Package Manager Console** option and enter commands into the command window:

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

Once you have those two NuGet packages installed, you can use the classes in the Microsoft.EntityFrameworkCore namespace to create a new class derived from the base DbContext class:

```
using Microsoft.EntityFrameworkCore;

public class BeverageContext : DbContext
{
    public DbSet<Beverage> Beverages { get; set; }

    // Use EF to create a Sqlite database file
    protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlite(@"Data Source=..\..\..\beverages.db");
}
```

Our BeverageContext will provide access to our database, which we have chosen to implement using SQLite, which is a very lightweight file-based database. We override the OnConfiguring method of DbContext to provide the data source, which would normally be done by means of a database connection string. For SQLite we only need to provide the path to the database file as the Data Source in the connection string. This database file doesn't exist yet.

Note: hard-coding a database string in a source file is a terrible practice from the point of view of both maintainability and security. This is just for the purposes of the demonstration.

We also add a member property that's a DbSet with a type parameter of Beverages. This is effectively going to give us a table whose rows are mapped to Beverage objects, where each of the fields has the appropriate type (there are ways of customizing this if the defaults don't work). Now we have everything in place, we can create an instance of our BeverageContext so we can start to interact with it:

```
using var db = new BeverageContext();
```

The first time this runs, the database won't exist. We can get at methods of the underlying database by means of the db.Database property and then call the EnsureCreated() method. This will create the database and set up any tables according to the schema defined by our classes. If the database or table already exists, it will leave it alone, even if the schema doesn't match. The first time we run the following code it will create a beverages.db file in our directory:

```
db.Database.EnsureCreated();
```

Now that we have a BeverageContext object, **db**, we can use that to start working with our database. For example, we can add some drinks:

```
db.Add(new Beverage() { Name = "Regular Coffee", Temperature = 80, Size = 100 });

db.Add(new Beverage() { Name = "Large Coffee", Temperature = 80, Size = 200 });

db.Add(new Beverage() { Name = "Herbal Tea", Temperature = 100, Size = 100 });

db.SaveChanges();
```

When SaveChanges() is called, the database will be updated with the new values. Let's list out the Beverages in the database (strictly the beverages in the DbContext object, which should now be the same thing):

```
foreach(var bev in db.Beverages)
    Console.WriteLine(bev.ID + " " + bev.Name);
db.Dispose();
```

After we've finished with our DbContext object we should Dispose() it. This doesn't delete the database; it just releases the database connection which, in this case, contains a file handle. We can look at the output of our console app:

```
1 Regular Coffee
2 Large Coffee
3 Herbal Tea
```

Notice that our integer property ID has been given the values 1, 2, and 3. If we run the app a second time, we'll get:

```
1 Regular Coffee
2 Large Coffee
3 Herbal Tea
4 Regular Coffee
```

5 Large Coffee

6 Herbal Tea

We added a second lot of drinks to the first three that persisted, because they were saved to the database. The ID values of the second three are 4, 5, and 6. When we give Entity Framework a class to work with, it will try to figure out a property that it can use as a primary key. There are a few rules that it uses, including looking for a field starting with ID, or you can designate some property. It does need to have unique values, though. If it can't find a suitable primary key, it won't compile, which is why we added the ID field to the Beverage class. In this case, Entity Framework automatically gave the ID field a unique value, numbering them sequentially. If you look in the database, you can see that it created a table with appropriate types for the properties in the POCO object. Entity Framework has been designed to work most of the time by convention, so that it will automatically do the right thing, most of the time, with minimal manual configuration.

Database-first with Entity Framework Core

Database-first with Entity Framework Core

- Use the Scaffold-DbContext PowerShell cmdlet
- Provide the connection string
- Use the generated DbContext and entity classes

```
using var db = new BeveragesContext();
db.Database.EnsureCreated();

foreach(var bev in db.Beverages)
    Console.WriteLine(bev.Id + " " + bev.Name);
```

If your application needs to connect to a database that already exists, then a different approach is needed. Entity Framework includes support for the database-first approach, where the database is made and planned before the model is created in code. This is the usual approach to building applications where you have an existing data source, which may have existed and been continually refined for many years. However, in the long run, this can make the solution less flexible.

Working with an existing database essentially amounts to reverse-engineering classes to correspond to the data tables. Let's try to recreate our entity classes using just SQLite the

database we created earlier. If you want to be thorough, you can delete the existing Beverage and BeverageContext before you start, to ensure there's no cheating.

First you will need to go back to the NuGet package manager and install two further packages called **Microsoft.EntityFrameworkCore.Design** and **Microsoft.EntityFrameworkCore.Tools**. Once you have that installed in the project you can run the following from the command line, or in the NuGet Package Manager Console, making sure you run it in the directory where beverages.db is located (or modify the connection string accordingly):

```
Scaffold-DbContext "Data Source=beverages.db"
Microsoft.EntityFrameworkCore.Sqlite
```

This command will generate a couple of source files, containing a couple of classes that are not that dissimilar from what we created when we used the code-first method in the previous section. Machine generated code tends to be more verbose, and we kept the preceding code to a minimum. Nevertheless, the **beveragesContext** looks similar to our hand-crafted **BeverageContext**, with a slightly different name that reflects the name of the database it was derived from.

```
public partial class beveragesContext : DbContext
{
    public beveragesContext()
    {
    }

    public beveragesContext(DbContextOptions<beveragesContext>
options)
        : base(options)
    {
    }

    public virtual DbSet<Beverage> Beverages { get; set; } = null!;

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {

        if (!optionsBuilder.IsConfigured)
    {
```

```
optionsBuilder.UseSqlite("Data Source=beverages.db");
}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Beverage>(entity =>
    {
        entity.Property(e => e.Id).HasColumnName("ID");
    });

    OnModelCreatingPartial(modelBuilder);
}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

One slight change is needed to the database location in the connection string, because we ran the command in the same directory as the database.

You should also see that the command has also generated a Beverages class for beverages:

```
public partial class Beverage
{
    public long Id { get; set; }

    public string Name { get; set; } = null!;

    public double Temperature { get; set; }

    public double Size { get; set; }
}
```

This is virtually identical to our original class, except that it is declared as a partial class (which means it can have its declaration spread across multiple source files), and the generated code uses a **long** for the ID instead of an **int**, and capitalized its name differently.

We could now substitute these classes for our original classes and run the same code against them as before with a couple of changes to the names of the DbContext class and the ID:

```
var db = new beveragesContext();
db.Database.EnsureCreated();
db.Add(new Beverage() { Name = "Regular Coffee", Temperature = 80, Size = 100 });
db.Add(new Beverage() { Name = "Large Coffee", Temperature = 80, Size = 200 });
db.Add(new Beverage() { Name = "Herbal Tea", Temperature = 100, Size = 100 });
db.SaveChanges();
foreach(var bev in db.Beverages)
Console.WriteLine(bev.Id + " " + bev.Name);
db.Dispose();
```

Question: You can use Entity Framework Core to create a database and entities from an EDMX design file?

Answer: False, Entity Framework Core does not support the EDMX file format for models. This was a feature of the legacy Entity Framework tooling.

Customizing Generated Classes

Customizing Generated Classes

- Do not modify the automatically generated classes in a model
- Use partial classes and partial methods to add business functionality to the generated classes

```
public partial class Employee
{
    partial void OnDateOfBirthChanging(DateTime? value)
    {
        if (GetAge() < 16)
        {
            throw new Exception("Employees must be 16 or over");
        }
    }
}
```

Sometimes the classes generated by the Scaffold-DbContext command aren't exactly what's required. You might just need a minor change, for example to add an **ID** property that wraps **Id**. Or you might want to add some additional custom logic. It might be tempting to just make that change directly in the generated class. The problem with modifying the generated code is that if you ever need to regenerate the classes, for example as a result of a schema change in the underlying database, then the custom changes will be lost.

Remember that the Scaffold-DbContext command generates partial classes for the entities and context classes. Using the **partial** keyword allows a classes definition to be spread across one or more source files. The compiler will draw together all the partial definitions in the assembly to compose the resulting class, struct, or interface. Rather than make changes to the generated classes, create new source files with partial classes to contain the additional functionality.

To learn more about partial classes, see: <https://aka.ms/K3I>.

Reading and Modifying Data

Reading and Modifying Data

- Reading data

```
var db = new beveragesContext();
foreach(var bev in db.Beverages)
    Console.WriteLine(bev.Id + " " + bev.Name);
```

- Modifying data

```
var db = new beveragesContext();
db.Add(new Beverage() { Name = "Herbal Tea" });
var b = db.Find<Beverage>((long)2);
if (b != null) db.Remove(b);
db.SaveChanges();
```

Whether you create the entity and context classes, or generate them using the tooling, you can use the context object to then read and manipulate data. You need to create an instance of the context class:

```
db.Database.EnsureCreated();
```

Once you have this you can get a collection of rows and iterate through them:

```
using var db = new beveragesContext();
foreach(var bev in db.Beverages)
    Console.WriteLine(bev.Id + " " + bev.Name);
```

This works file for a small number of rows, but if the underlying table is large, you'll want to be a little more careful how many you bring into memory at any one time.

If you want to modify the data, you can use one of the many methods on the base DbContext class. For example we can add a record:

```
db.Add(new Beverage() { Name = "Regular Coffee", Temperature = 80, Size = 100 });
```

This only changes the version of the data in memory. To persist those changes to the database we need to call the SaveChanges() method.

```
db.SaveChanges();
```

The following example will look in the database for a Beverage with an Id of 2, and remove it.

```
var b = db.Find<Beverage>((long)2);  
if (b != null) db.Remove(b);  
db.SaveChanges();
```

It's a good practice to remember to dispose of the DbContext object.

```
db.Dispose();
```

Question: What disk types are you most commonly using in your organization, and do you have a management and provisioning strategy for storage usage in particular scenarios?

Lesson 2 - LINQ

Language Integrated Query (LINQ), is a .NET technology that includes a set of fluent APIs and special language syntax that makes it easy to write type-safe queries in a structured way, similar to SQL (Structured Query Language). It is designed as a natural migration path from writing native database queries, and to some extent from functional programming, into the world of type-safe code. LINQ can query over local object collections in memory, XML Documents, and of course data sources.

The only requirement for LINQ to work is that the collection implements the `IEnumerable<T>` interface. This applies to all the generic collections, as well as ADO.NET data sets. You then can enjoy the benefits of type-safety and syntax checking, and also IntelliSense, while still being able to build complex queries.

This lesson explores the LINQ architecture and the fundamentals of writing queries.

Lesson Objectives

After completing this lesson, you'll be able to:

- Use LINQ expression syntax to query data.
- Use anonymous types.
- Control the timing of query execution.

Querying Data

Querying Data

- Use LINQ to query a range of data sources, including:
 - .NET Framework collections
 - SQL Server databases
 - ADO.NET data sets
 - XML documents
- Use LINQ to:
 - Select data
 - Filter data by row
 - Filter data by column

The LINQ query expression syntax is based on the following basic structure:

```
from variablenames in datasource select variablenames
```

The syntax is intentionally similar to SQL, and like SQL you can use additional modifiers to retrieve just the data you want. This is often very important because you don't want to retrieve more data than you require. We can demonstrate the principles by just using a collection of Beverage objects. We'll use our data set from the previous examples, although we could also simply create a `List<Beverage>`, and it would work in exactly the same way. First, let's do some set up, and we'll recreate the database on each run so that we have a consistent starting point:

```
var db = new beveragesContext();
db.Database.EnsureDeleted();
db.Database.EnsureCreated();
db.Add(new Beverage() { Name = "Regular Coffee", Temperature = 80, Size = 100 });
db.Add(new Beverage() { Name = "Large Coffee", Temperature = 80, Size = 200 });
db.Add(new Beverage() { Name = "Herbal Tea", Temperature = 100, Size = 100 });
db.Add(new Beverage() { Name = "Hot Chocolate", Temperature = 80, Size = 200 });
```

```
db.Add(new Beverage() { Name = "Earl Grey Tea", Temperature = 100, Size = 100 });

db.Add(new Beverage() { Name = "English Breakfast Tea", Temperature = 100, Size = 100 });

db.SaveChanges();
```

Now we can use LINQ to get a collection of beverages, and then print them out:

```
var beverages = from b in db.Beverages select b;

foreach (var bev in beverages)

Console.WriteLine(bev.Id + " " + bev.Name);
```

The LINQ query returns another collection – it's actually of type `IQueryable<Beverage>?` (the question mark simply indicates that the type is nullable, i.e. it is allowed to have a null value). So it's a strongly-typed collection, and `IQueryable` is itself derived from `IEnumerable<T>` so we can use it in a `foreach` statement. This isn't a very exciting first demonstration of LINQ; we could have simply passed the `db.Beverages` object to the `foreach` and it would have had the same effect. But usually database queries do more than just retrieve the whole contents of a table. Let's do something a bit more interesting, and suppose we only like tea. We can change the query to:

```
var beverages = from b in db.Beverages where b.Name.EndsWith("Tea") select b;
```

This will now output just the teas. We can also change the order to list them alphabetically:

```
var beverages = from b in db.Beverages where b.Name.EndsWith("Tea") orderby b.Name select b;
```

Many people prefer to indent each part of the LINQ statement for dramatic effect:

```
var beverages = from b in db.Beverages
where b.Name.EndsWith(" Tea")
orderby b.Name
select b;
```

What the compiler is actually doing is using the special LINQ language syntax to convert the above into something like the following:

```
var beverages = db.Beverages.  
    where(b => b.Name.EndsWith(" Tea")).  
    OrderBy(b => b.Name.EndsWith(" Tea")).  
    Select(b => b);
```

This is the fluent-style API that LINQ provides, where you can see there's extensive use of lambda expressions. You can use the API syntax if you prefer, and the above code works in exactly the same way, but it is more verbose and probably not as easy to read. It might be a better choice if you had an expression where you needed to add more fluent APIs that weren't part of the LINQ language syntax. You'll might also notice that the `Select` method at the end isn't strictly necessary in this case, and we could omit it without affecting the result. We can't omit `select` from the LINQ query, though.

We could also return `Select(b => b.Name)` to return just the beverage name, which is all we actually need. This is, after all, the job of a `select` clause, to filter by column. The LINQ query syntax version would be:

```
var beverages = from b in db.Beverages  
    where b.Name.EndsWith(" Tea")  
    orderby b.Name  
    select b.Name;
```

When `select` returns something other than just a copy of the range variable in the `from`, it's called a "projection". Now, `beverages` will be returned as a `IQueryable<string>?` and we'll have to update our `foreach`:

```
foreach (var bev in beverages)  
    Console.WriteLine(bev);
```

If we still want the `Id` to be output, then we'll have to do a bit more work in the `select` clause. First we can create a new type for the results, which we'll call `TempClass`:

```
public class TempClass  
{  
    public long Id { get; set; }  
    public string Name { get; set; }
```

```
}
```

Then we can use a projection to select just the Name and Id, and we can expect back a collection of tuples, and put the **foreach** loop back the way it was before:

```
var beverages = from b in db.Beverages  
where b.Name.EndsWith(" Tea")  
orderby b.Name  
select new TempClass { Id = b.Id, Name = b.Name };  
  
foreach (var bev in beverages)  
Console.WriteLine(bev.Id + " " + bev.Name);
```

TempClass doesn't look very elegant. In the next section we'll try and improve on it.

Querying Data with Anonymous Types

Querying Data by Using Anonymous Types

Use LINQ and anonymous types to:

- Filter data by column
- Group data
- Aggregate data
- Navigate data

In the previous example we had to define the rather clunky **TempClass** type, just so we could select a couple of columns to return for each row. It was good to have a strongly-typed result – an **IQueryable<TempClass>?**, but we paid quite a price in terms of complexity in having to define an ad-hoc type in order to achieve it. This can start to become pretty tedious if you have to compose a lot of LINQ queries, and rather undermines the elegance of the LINQ query expression syntax.

A solution to this problem is to use anonymous types. The return type is created using an implicitly typed local variable, without needing to construct a type beforehand. We can rewrite the LINQ query more succinctly, without needing a named type, as follows:

```
var beverages = from b in db.Beverages
```

```
where b.Name.EndsWith(" Tea")
orderby b.Name
select new { Id = b.Id, Name = b.Name };
```

If the names of the projected fields are the same as the source fields, you can simplify the **select** even further:

```
select new { b.Id, b.Name };
```

LINQ syntax also has a **group** keyword. You can use it to group together the items in the collection or data set. We can use this to report aggregate data on a collection. Let's display the different drink sizes, and the number of items on the menu for each:

```
var sizegroups = from b in db.Beverages
group b by b.Size into sizes
select new { size=sizes.Key, count=sizes.Count() };
foreach (var group in sizegroups)
Console.WriteLine("size: " + group.size + "ml, " + group.count +
" items");
```

This produces the following output:

```
size: 100ml, 4 items
size: 200ml, 2 items
```

Query Execution

Query Execution

- Deferred query execution—default behavior for most queries
- Immediate query execution—default behavior for queries that return a singleton value
- Forced query execution—overrides deferred query execution:
 - **ToArray**
 - **ToDictionary**
 - **ToList**

```
IList<Employee> emp = (from e in FCEntities.Employees  
                      orderby e.LastName  
                      select e).ToList();
```

Consider the case where we retrieved a query, then used a foreach to list all the beverages. In fact, we'll rewrite the code slightly so that we'll retrieve the values from beverages.db that were saved in the earlier run:

```
var db = new beveragesContext();  
  
var beverages = from b in db.Beverages  
where b.Name.EndsWith(" Tea")  
select new { b.Id, b.Name };  
  
foreach (var bev in beverages)  
Console.WriteLine(bev.Id + " " + bev.Name);
```

The result of the query was not an **IEnumerable<T>**; it was an **IQueryable<T>**. The difference is important, because an **IEnumerable** is just a collection that exists in memory. In contrast, the **IQueryable** is more of an object that has the potential to be enumerated. Precisely when the database call occurs will depend on how you use it. In the code examples above, LINQ will use deferred execution for the query; in other words, the database call won't occur until you start to enumerate the results of the query, at the start of the **foreach** loop. This can be important when you're using a remote database, and database calls are expensive, and you need to build up a query in several steps. It will also ensure that you don't use precious bandwidth to retrieve data that you might not need.

If you use a LINQ query to calculate a scalar aggregate value, for example an average or maximum; the query will execute immediately. “Immediate query execution” is necessary in these situations because the query needs to retrieve the values immediately in order to calculate the result.

Another way you can force the query to execute immediately is to convert the result to an in-memory collection using conversion operators like the **ToArray**, **ToList**, and **ToDictionary** operators. Essentially, anything that causes enumeration of the **IQueryable** will trigger a query execution. The following code will immediately execute the query and return a generic **List<T>** of an anonymous type, containing a tuple of a **long** and a **string**, which can then be enumerated in the **foreach**.

```
var beverages = (from b in db.Beverages  
where b.Name.EndsWith(" Tea")  
select new { b.Id, b.Name }).ToList();  
  
foreach (var bev in beverages)  
Console.WriteLine(bev.Id + " " + bev.Name);
```

Question: True or false: Visual Studio provides compile-time LINQ syntax-checking and type-checking and IntelliSense support?

Answer: True

Lab: Retrieving and Modifying Grade Data

Lab: Retrieving and Modifying Grade Data

Lab scenario

You have been asked to upgrade the prototype application to use an existing SQL Server database. You begin by working with a database that is stored on your local machine and decide to use the Entity Data Model Wizard to generate an EDM to access the data. You will need to update the data access code for the Grades section of the application, to display grades that are assigned to a student and to enable users to assign new grades. You also decide to incorporate validation logic into the EDM to ensure that students cannot be assigned to a full class and that the data that users enter when they assign new grades conforms to the required values.

Objectives

Exercise 1:

Creating an Entity Data Model from The School of Fine Arts Database

Exercise 2:

Updating Student and Grade Data by Using the Entity Framework

Exercise 3:

Extending the Entity Data Model to Validate Data

Module Review and Takeaways

Review Questions

Question: You have built a model from an existing database in Entity Framework. You discover you need to add a new method with some custom functionality. Where should you write your code?

- A: In a new custom class
- B: In a new class derived from the existing entity class
- C: In a partial class for the existing entity class
- D: In the existing entity class

Answer: C, in a partial class. Not D, because changes would be lost if class regenerated.

Module 8 – Using the Network

Module Overview

Many years ago a business would have an on-premises data-centre, and perhaps a leased line or two to unite branch offices. The idea of connecting to any kind of external network was anathema, and if you tried to do it, the IT department would hunt you down. Today's world is very different, and information workers expect full access to the public Internet, in fact their jobs would be impossible without it. Data could be located anywhere in the world, and needs to be accessible at all times.

Internet applications fall broadly into client applications, and server applications. Client applications ask for information and server applications respond with that information when clients requests it. This module examines the many .NET APIs in the System.Net namespace, and how we can use them to build client applications that access remote data sources.

Objectives

After completing this module, you'll be able to:

- Send data to remote web services.
- Access remote data over web services.
- Understand REST and OData

Lesson 1 – Web Services

A common programming task is to call some web service using a REST-style interface in order to retrieve some data or update a server.

Lesson Objectives

After completing this lesson, you'll be able to:

- Make simple requests using HttpClient.
- Understand the lower-level .NET classes for network communications.

Client-side Network Classes

Client-side Network Classes

- **HttpClient** (most commonly used)
- **HttpClientHandler** (helper class)
- Deprecated in .NET 6
 - WebClient
 - WebRequest

There is a bewildering array of network APIs in the System.Net namespace, and many classes that are not designed to be used directly, as well as numerous deprecated classes that have been retained for backwards compatibility. These latter categories should be avoided.

In earlier versions of .NET, the most commonly used base classes were the **WebRequest** and **WebResponse** classes. These two classes use a request/response pattern that maps to the underlying protocol, where a client makes a request and then receives a response, but they have been gradually replaced with newer classes, and should not be used. The principle of request/response remains the same, though, and the latter step might happen some considerable time after the request is made. For this reason, networking APIs usually offer a way of calling these methods asynchronously. Execution of the application then continues, and when the network response eventually arrives, an event handler is used to process it.

Important: From .NET 6, the **WebRequest**, **WebClient**, and related classes, which were included for backwards compatibility, are now officially deprecated:
<https://docs.microsoft.com/en-us/dotnet/core/compatibility/networking/6.0/webrequest-deprecated>

Rather than code against these lower-level APIs directly, it's common to use one of the higher-level classes such as **HttpClient**, which we'll see in action in the next section. **HttpClient** is a relatively new API that is specifically designed for using the HTTP protocol, unlike the more older and more generic **WebClient**.

At one time there were quite a number of popular options if you wanted to build some kind of client-server application, including SOAP (Simple Object Access Protocol), a whole variety of WS standards for managing security around messages, and other standards for invoking methods on remote systems using Remote Procedure Calls (RPC). In recent years, the REST style of network communications has become dominant because of its simplicity. REST (Representational State Transfer) takes advantage of the HTTP protocol and its methods (GET, PUT, POST, PATCH, DELETE), as well as transport-layer security that is now accepted as a minimum security posture in web communications. The HTTP methods, largely ignored for much of the history of the web, already provide a means of performing create, read, update,

and delete operations on remote resources. There has been a gradual realization that it was unnecessary to build another layer of complexity on top of the protocol in order to achieve the same thing.

Because so much development now uses REST-based services, the **HttpClient** can probably offer most of what we need to build web client applications. We'll examine this API in more detail in the next section.

Note: To learn more about REST architectural style and see an example, see:
<https://aka.ms/3MxmQ>.

Using HttpClient

Using HttpClient

Create an **HttpClient** object, and use its methods:

```
var client = new HttpClient();
var res = await client.GetStringAsync("https://microsoft.com/");
Console.WriteLine(res);
```

The easiest way of making an HTTP request is to use the **System.Net.Http.HttpClient** class. We just need to create an instance of **HttpClient** and then call its **client.GetStringAsync(url)** method.

```
var client = new HttpClient();

var msg = client.GetStringAsync("https://microsoft.com/");

Console.WriteLine(msg); // won't work
```

This won't work because the code will return immediately and **msg** will have a Task Async promise type object in it. The fact that this method has the word "Async" in the name should be a clue that we're not going to be able to just call it and expect it to return when it has a result. Like all networking methods, you really have to call this asynchronously, because the nature of networks is that they have unpredictable delays. Some classes have methods that have synchronous and asynchronous methods so that you can choose, but not this one. So we'll need to do a little asynchronous programming.

```
var client = new HttpClient();

var msg = await client.GetStringAsync("https://microsoft.com/");

Console.WriteLine(msg);
```

We've added the keyword **await**, that tells the compiler to generate code that will wait for the **GetStringAsync** to return. Actually, it does something much cleverer than that. The generated code will immediately return from whatever function it is inside, allowing the application to continue doing things like updating UIs or calculating or just waiting. Then, when the network message is received, the execution of the code continues, as if it were an event handler, and does the rest of the function. You could write code to do this yourself, but it would be quite complicated, and the async/await pattern makes this very complex multi-threaded code a lot simpler.

Note: since **HttpClient** is disposable, we should call **client.Dispose()** when we have finished with the connection.

The result of running this code will be to print out a string containing the contents of the Microsoft home web page, except that if you look at it in any detail, you'll likely see that it's not the normal landing page. Instead, you'll probably get a fairly snooty response, something like "Your current User-Agent string appears to be from an automated process".

The Microsoft web site thinks there's something fishy about this request, as if it came from some kind of automated process, rather than a bona-fide web user. It's a fair cop. What we can do is at least add an honest header for the User-Agent string in the appropriate HTTP header. We can do that by adding to the **DefaultRequestHeaders** property of the client:

```
var client = new HttpClient();

client.DefaultRequestHeaders.Add("User-Agent", @"Bill's Auto Web Page Scraper");

var msg = await client.GetStringAsync("https://microsoft.com/");

Console.WriteLine(msg);
```

Unfortunately the Microsoft server probably won't be any more impressed, and the effect will be the same. If you use a tool like Fiddler, you'll be able to see that the custom User-Agent header has been added.

Note: The **HttpClient** object is designed to be re-used if your application makes multiple HTTP requests. It is not recommended to repeatedly create a new **HttpClient** for every call, as there is a risk of exhausting the underlying network resources.

Getting Data using **HttpClient**

Getting Data using HttpClient

- Parse JSON response to deserialize data

```
var client = new HttpClient();
var json = await client.GetStringAsync("https://api.foo.com/repos");
var repos = JsonSerializer.Deserialize<List<Repo>>(json);
```

We retrieved a string using HttpClient. What if we want some kind of data? If it's JSON data, we can parse it using the Deserializer discussed earlier in the course. Let's go to a different endpoint with the code we wrote in the previous section. There's an API for GitHub repositories at <https://api.github.com/>, so we're going to use that to retrieve a list of all the repositories under the dotnet organization:

```
var client = new HttpClient();

client.DefaultRequestHeaders.Add("User-Agent", @"Bill's GitHub Repo-scraper");

var msg = await client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

Console.WriteLine(msg);
```

If we run that code, we should see a very large string of JSON coming back. Let's do something more useful with it by converting it into a list of objects. First we need to decide what our object looks like. By inspection of the JSON that comes back, we can reverse-engineer the data format:

```
[
{
  "id": 12191244,
  "node_id": "MDEwOlJlcG9zaXRvcnkxMjE5MTIONA==",
  "name": "BenchmarkDotNet"
  .... other stuff
},
{
  "id": 12576526,
  "node_id": "MDEwOlJlcG9zaXRvcnkxMjU3NjUyNg==",
```

```
"name": "reactive"
```

It looks as though we're getting an collection or array of about 30 or so objects, with each having a lot of parameters (which we've omitted). We don't need to pull everything from the data string. Let's make a list of **ids** and **names**. For that we're going to need a type to put them in; a struct will do:

```
public struct Repo  
{  
    public long id { get; set; }  
    public string name { get; set; }  
}
```

Now we can modify our code to bring in the **System.Text.Json** namespace and use the **JsonSerializer** object. We need to give the generic **Deserialize** method a type. We can see that we need to capture a collection of **Repo** objects, so we'll choose a **List<Repo>** as the type parameter:

```
using System.Text.Json;  
  
var client = new HttpClient();  
  
client.DefaultRequestHeaders.Add("User-Agent", @"Bill's GitHub Repo-scraper");  
  
var json = await client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");  
  
var repos = JsonSerializer.Deserialize<List<Repo>>(json);  
  
foreach (var repo in repos)  
  
    Console.WriteLine(repo.id + " " + repo.name);
```

Running this code will retrieve a collection of **Repo** objects that we can then print out using the **foreach** loop. The result is a list of repositories with just the id and name:

```
12191244 BenchmarkDotNet
```

```
12576526 reactive
```

```
16157746 efcore
```

```
17620347 aspnetcore
```

```
21291278 Scaffolding
```

```
23337905 Docker.DotNet
```

26295345 corefx

26295349 buildtools

26784827 core

28232663 orleans

...etc...

Question: True or false: you need to create a new HttpClient object for each request?

Answer: False, the HttpClient object is designed to be reused.

Getting Raw Data with HttpClient

Getting Raw Data with HttpClient

- Response as byte array

```
var client =new HttpClient();
byte[] data =await client.
    GetByteArrayAsync("https://www.foo.com/logo.png")
```

In the previous section we used the `HttpClient.GetStringAsync` method to retrieve a JSON payload, then we converted it into .NET objects we could work with using the `System.Text.Json.Serializer` class. Increasingly, this is the way web service calls are being implemented in modern connected services. But what if you want to send some binary data? For example, you might have an image file.

To solve this kind of problem we have another method called `GetByteArrayAsync`. This works in almost exactly the same way but returns a `byte[]`. For example:

```
var client = new HttpClient();

byte[] data = await client.GetByteArrayAsync("https://www.microsoft.com/logo.png");

// do some image processing...
```

There is also a `GetStreamAsync` method, that can be used to get a stream. This might be useful if you wanted to build an application to download files, as you could take the stream and write it straight out to the filesystem without having to load the whole thing into memory first.

Uploading Data with HttpClient

Uploading Data with HttpClient

- Create an HttpRequestMessage object

```
var client = new HttpClient();
var request = new HttpRequestMessage(
    HttpMethod.Post, "http://foo.com/upload");
request.Content = new StringContent(
    "Test data to upload with HTTP body.");
HttpResponseMessage response =
    await client.SendAsync(request);
```

Uploading content, for example in a request body, can be achieved by creating an HttpRequestMessage:

```
var client = new HttpClient(handler);

var request = new HttpRequestMessage(HttpMethod.Post, "http://microsoft.com/upload");

request.Content = new StringContent("Test data to upload with HTTP body.");

HttpResponseMessage response = await client.SendAsync(request);
```

Dealing with Network Problems

Dealing with Network Problems

- Modify HttpClient behaviour

```
var handler = new HttpClientHandler();
handler.AllowAutoRedirect = false;
var client = new HttpClient(handler);
```

- Wrap in try/catch block

```
try
{
    var client = new HttpClient();
    var json = await client.GetStringAsync("https://foo.com");
    Console.WriteLine(json);
}
catch (HttpRequestException)
{
    Console.WriteLine("HTTP request failed: " + e.Message);
}
```

Whenever you build a solution that requires a network connection, it's only a question of time before problems occur. In that case you might want to fine-tune things like the `HttpClient` behaviour. A quick look at the `HttpClient` class using Intellisense reveals a fairly spartan set of options. Many of the other options you might look for are actually contained in another class called `HttpClientHandler`. You can set up the `HttpClientHandler` object first and pass it to the `HttpClient` constructor:

```
var handler = new HttpClientHandler();

handler.AllowAutoRedirect = false;

var client = new HttpClient(handler);
```

Given that failure is almost inevitable, we should wrap our network code in a try/catch. We'll be particularly interested in any `System.Net.Http.HttpRequestException` that we catch.

```
try
{
    var client = new HttpClient();

    var json = await client.GetStringAsync("https://microsoft.com");

    Console.WriteLine(json);
}
```

```
catch (HttpRequestException e)
{
    Console.WriteLine("HTTP request failed: " + e.Message);
}
```

Authorizing a Web Request

Authorizing a Web Request

- Attach credentials

```
var handler = new HttpClientHandler();
handler.Credentials =
    new NetworkCredential(username, password);
var client = new HttpClient(handler);
```

- Authorization header

```
client.DefaultRequestHeaders.Authorization =
    new System.Net.Http.Headers.
        AuthenticationHeaderValue("Bearer", accessToken)
```

Most web services are protected against unauthorized use. The data at these endpoints may be very valuable. If you put an unprotected data service on the public Internet, it will be a matter of minutes before bad actors are trying to gain access to it. In the early days of the Internet, data was protected with a rudimentary username and password combination, often sent in plain text. You can still do this of course, but it's not secure. Nowadays it's recommended to use libraries written by security specialists, and industry standard algorithms and techniques.

If you need to pass credentials to an endpoint, this can be done through the `HttpClientHandler` object:

```
var handler = new HttpClientHandler();

handler.Credentials = new NetworkCredential(username, password);

var client = new HttpClient(handler);
```

For more information about the `NetworkCredential` class, see: <https://aka.gd/3z5Rvfd>.

There are a numerous ways to make sure that remote data sources are safe. Normally this is achieved in modern infrastructures by use of asymmetrical encryption and access tokens. An

access token acts as a limited time authorization to get access to a resource. Protocols like those based on OAuth2 use bearer tokens, which means there is no associated identity that is checked in addition to the validity of the token. For this reason it's vitally important that all such communication occurs over HTTPS.

The HttpClientHandler object has a Credentials property that you can use to store credentials to send in the request. You can also set headers directly, so for example if you had a security library that used OAuth2 and was able to acquire an access token, you could supply that as the authorization header when you make the request.

```
client.DefaultRequestHeaders.Authorization =  
    new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", accessToken);
```

To learn about authorization and authentication, see: <https://aka.ms/3G5uz15>.

Question: You are using a web service secured by OAuth2. What do you need to send to the service endpoint in the Authorization header?

- A: An SSL certificate
- B: An access token
- C: A refresh token
- D: A username and password

Answer: B

Lesson 2 – REST and OData

For many years, WCF Data Services was the framework of choice for developing distributed applications that required network communications to share information over the public Internet, and had support for multiple protocols and technologies. WCF then began to embrace the Representational State Transfer (REST) architectural model and open web standards like the Open Data Protocol (OData). Over time, these newer technologies have become dominant to the point that a large multi-model framework like WCF is no longer needed. As .NET has moved to become a cross-platform framework, WCF has been left behind, but you may find yourself working on a legacy project that still uses it.

In this lesson, we'll briefly look at WCF, and possible migration approaches, and then look in a little more detail at the REST and OData technologies that have largely replaced it.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand WCF as a legacy network communications framework.
- Use REST-style interfaces.
- Understand OData.

WCF Data Services

WCF Data Services

- WCF Data Services is based on the `System.Data.Services.DataService<T>` generic class

```
public class BeverageDataService : DataService<FourthCoffee>
{
    ...
}
```

- URLs are mapped to entity sets by a data service:

`http://FourthCoffee.com/SalesService.svc/SalesPersons`



WCF came out at the end of 2006 as a part of the .NET Framework 3.0. At that time the developer landscape for distributed applications looked very different; there was widespread use of Remote Procedure Calls (RPC), XML, and SOAP messages, and JSON was yet to become popular. In the intervening time, technology and the industry have moved forward, and we've discovered better ways of solving the problems that WCF was designed for.

If you have a legacy project that uses WCF, and you want to use the latest versions of .NET and C#, you have three choices; you can migrate the project, continue on legacy .NET Framework versions, or re-architect it. If you decide to migrate, then you'll discover that there's support in .NET 6 for the client-side WCF APIs, but not the server-side APIs. Fortunately there is now a community supported version of WCF called CoreWCF that provides server-side APIs. As with any community project, it's up to you to decide whether there is a sufficient level of support for your needs.

The second option is to continue to use the legacy .NET Framework. The main drawback here is that you won't then be able to enjoy new features of the .NET platform as they become available. This option might be attractive though, if you want to keep a project running while concurrently building a replacement with newer technology.

The third option is to re-architect the solution with newer technologies such as HTTP/REST and OData. This is the topic of the rest of this module.

HTTP Services

HTTP Services

- HTTP is a first class application protocol
- An HTTP URI has the following basic structure:

http://blueyonder.com:8080/travelers?id=1



- HTTP defines a set of methods or verbs that add action-like semantics to requests

The original HTTP protocol was designed to allow web applications to be scalable, with ideas like caching and stateless architecture taken into account. HTTP is now used by a wide range of devices and platforms, including almost all computer systems. HTTP is also easy to use because it uses text messages and follows the request-response pattern for sending messages. HTTP is different from most application layer protocols because it was not originally designed to be an RPC or RMI mechanism. Instead, HTTP tells you how to get and change resources at an address or URI. Both websites and web services use HTTP. The Uniform Resource Identifier (URI or sometimes called a URL) is a standard for identifying network resources that's used by HTTP.

The structure of a URI consists of a schema, the hostname, a port (default 80), an absolute path to the resource, and a query string. The hostname identifies a network resource, typically an individual server, and is usually given as a domain. By using the Domain Name Service (DNS), the name can be mapped to an actual IP address on the network. The path part of the URI is usually used to locate the resource, and the query string can be used to add additional parameters if needed.

Using REST APIs

Using REST APIs

- REST-style interfaces:
 - Use HTTPS to secure data
 - Use HTTP methods
 - Url defines the resource

<https://myservice.org/v2/Store/Beverages>

Representational State Transfer (REST) is a style of architecture that builds on the resource-based HTTP protocol. It was first described by Roy Fielding in 2000, who was one of the authors of the specifications for HTTP, URI, and HTML. Fielding's doctoral thesis was about an architectural style that uses HTTP and Web technologies to make applications that are simple and easy to scale and expand. REST is now widely used in implementing web services, sometimes called “RESTful services” using the various HTTP verbs to build a full CRUD (Create, Read, Update, Delete) API based on these resources.

When you make a request to a REST endpoint, you use the security layer of HTTPS to protect the traffic, you use the HTTP methods to tell the server what you want to do (create, read, update, or delete), and use the URL path to indicate the exact resource on the server you want to interact with.

A typical URL for a REST request takes the following form:

<https://myservice.org/v2/Store/Beverages>

where **v2** is a version for the API, and **Store/Beverages** is the resource, the list of beverages in the store. Making a REST call to this endpoint URL will return all the Beverages with all their information.

The OpenAPI Specification (formerly Swagger) is a specification for machine-readable metadata to describe, consume, and visualize RESTful interfaces. It can be used to document REST APIs by reading the API metadata from the endpoint and then generate a language-specific client library. Once a client library has been created, for example in C#, this can be included in the project and called to perform operations, rather than having to create the HTTP requests using a lower-level API like HttpClient.

Note: you can find out more about the OpenAPI standard at: <https://aka.gd/3wEk2We>.

OData

OData

- OData - Query to limit data returned:
 - Format results: \$select, \$ orderby
 - Control results: \$ fileter, \$expand
 - Paging: \$top, \$skip, \$ skiptoken

When you make a request to a REST endpoint, you use the security layer of HTTPS to protect the traffic, you use the HTTP methods to tell the server what you want to do (create, read, update, or delete), and use the URL path to indicate the exact resource on the server you want to interact with.

The one thing that's missing is a way of efficiently querying the data, in the same way that we make database queries that limit the specific rows and columns that we require.

OData, the Open Data Protocol, provides a standard way of doing this, by adding additional query parameters. You can add a **\$select** parameter to choose the columns you want to return, in much the same way you would use **select** in SQL or in a LINQ query. You can also use the **\$filter** parameter to further limit the resources returned, like a **where** clause. You can also order results using the **\$orderby** parameter. If you have too much data to return from a single request, you can perform paging by using the **\$top**, **\$skip**, and **\$skiptoken** parameters.

Referring back to the typical URL for a REST request:

<https://myservice.org/v2/Store/Beverages>

Making a REST call to this endpoint URL will return all the Beverages with all their information. If we only need the teas, and only the names and ID of the teas, then retrieving all the data is very wasteful of both time and bandwidth.

In order to be more selective about the information returned, we can use OData parameters to limit this. We could use the following Url:

[https://myservice.org/v2/Store/Beverages?\\$filter=contains\(Name,"Tea"\)&select="Id,Name"](https://myservice.org/v2/Store/Beverages?$filter=contains(Name,)

If we have a lot of beverages, and only limited display space to list them, we could choose to limit the response to the first 50 results, and then only fetch more data if the user scrolls the display:

[https://myservice.org/v2/Store/Beverages?\\$filter=contains\(Name,"Tea"\)&select=Id,Name&top=50](https://myservice.org/v2/Store/Beverages?$filter=contains(Name,)

OData provides a consistent standard for the way queries, methods and updates are performed over HTTP, responses are formatted, dealing with concurrency issues, and more.

Note: you can find out more about the OData standard at: <https://aka.ms/3wuJf6G>.

Lesson 3 – ASP.NET Core MVC

In the previous lessons we've looked at how to write client applications that use web services. In this section we'll very briefly look at a .NET technology for hosting web services. This is just to introduce the topic. To learn more about ASP.NET Core MVC, there is a separate dedicated course in this series called "Developing ASP.NET Core MVC Web Applications".

Lesson Objectives

After completing this lesson, you'll have been:

- Introduced to ASP.NET Core MVC.
- Given an overview of how REST-style interfaces are implemented on ASP.NET Core MVC.

What Is ASP.NET Core MVC?

What is ASP.NET Core MVC?

- Web application framework
- Cross-platform and Open Source
- Model View Controller pattern
- Implemented on ASP.NET
- ASP.NET Core (built on .NET Core, now .NET6)
- Web applications using Razor view engine
- Web API for web services

Microsoft introduced ASP.NET in 2002 as a successor to Active Server Pages, built on .NET. For many years this built on Web Forms as a web page rendering technology, and WCF for hosting web services. More recently, with the growing popularity of web application frameworks built on the MVC (Model View Controller) pattern, Microsoft introduced ASP.NET MVC. This left behind the Web Forms model and also acted as a basis for web services. With the cross-platform versions of .NET, introduced with .NET Core and now at .NET 6, a new version of ASP.NET was developed, called ASP.NET Core. This version didn't include Web

Forms and only offers MVC and a simplified model called ASP.NET Web Pages. The name “Core” was dropped when .NET Core became the mainstream version of .NET, but ASP.NET Core has retained the “Core” name to distinguish it from legacy versions that work on the older .NET Framework.

ASP.NET Core is completely cross-platform and open source (<https://github.com/dotnet/aspnetcore>).

The tooling in ASP.NET Core MVC makes it fairly easy to host web applications that expose sophisticated web services using Web API. Web API is well suited to creating REST endpoints, and makes use of the same MVC programming model that provides web applications. In an MVC application the application code is split into three separate classes:

- A Controller. This class handles the request sent to the server by the client. It typically connects to a data source, gets the information it needs, updates the data source if necessary, and sends back the HTTP response with the data.
- A Model. This class represents the business objects that the web application manages.
- A View. For a web page, the job of the view is to render the contents according to the needs of the application. In the case of a web service, the view doesn’t participate, and you can think of the view as being rendered by the client.

It’s not uncommon to find that in an MVC application, the effort to add a web service is minimal. This is because the controllers and models have already been set up to retrieve data, and then render it in the form of web pages using the view. In this case it’s just a case of adding another controller that, instead of invoking a view and rendering it, will simply return the raw data in JSON format.

Introducing Web APIs

Introducing Web APIs

- A Web API controller class

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
    public string Get()
    {
        return "Response from Web API";
    }
}
```

HTTP is an application layer protocol that was originally designed to move hypertext resources across computer networks. It is the main protocol for many applications, including the World Wide Web. HTTP is now one of the most widely used protocols for building apps and services. Web API is a framework that has everything you need to build HTTP-based services.

Web API uses controllers and actions to handle requests. A controller is a class that derives from the base class **ControllerBase**. By convention, controllers are usually named with the word "Controller" at the end. Actions are the methods in the controllers that handle requests and send back the results. A routing system in ASP.NET Core MVC ensures that the correct controller and action gets called depending on the request URI.

The code sample below shows a simple controller class called **HomeController**. Although there is a default convention-based routing mechanism in ASP.NET Core MVC, the class has been decorated with an explicit route by means of an attribute.

```
[Route("api/[controller]")]
public class HomeController : ControllerBase
{
    public string Get()
    {
        return "Response from Web API";
    }
}
```

A controller class for a REST API will typically implement methods called Get, Post, Put and Delete, to correspond to the HTTP methods GET, POST, PUT and DELETE. The routing engine will then map the correct methods depending on the request, although this convention can also be overridden by using attributes on the methods.

The example just returns a string, although in a real application it would probably be returning JSON data.

Note: To learn more about ASP.NET Core MVC, see the documentation:
<https://aka.ms/3LxJvGh>.

Lab: Using the Network

Lab: Using the Network

Lab scenario

You have been asked to create a web-based application for the school. To do this you need to create a page showing all the school departments, enable users to book a meeting with staff of a particular department, add a job vacancies page, and allow submitting an application to a selected job.

You will create a server-side ASP.NET Core Web API application and a clientside ASP.NET Core web application. In the client-side application, you will call the Web API actions by using the HttpClient class and by using jQuery.

Objectives

Exercise 1:

Adding Actions and Calling them by using Microsoft Edge

Exercise 2:

Calling the Schools Web API by using ServerSide Code

Exercise 3:

Calling a Web API by Using jQuery

Module Review and Takeaways

Review Questions

Question: Which OData query parameter would you use to ensure that only the required fields are returned?

- A: \$filter
- B: \$select
- C: \$orderby
- D: \$skiptoken

Answer: B

Module 9 – Graphical User Interfaces

Module Overview

We've built a number of console applications to provide a simple framework to demonstrate a piece of code, and you can certainly build console apps for production use. But in general these will only appeal to other developers and IT personnel. If you want to make a successful app for end users to use on desktop computers or mobile platforms, you need to build some kind of graphical user interface. In this module, we'll look at some of the options for UI development, as well as considering some of the cross-platform development options.

Objectives

After completing this module, you'll be able to:

- Understand UI frameworks based on XAML.
- Use data binding for UI views.
- Be aware of some of the options for cross-platform UI development.
- Understand the Blazor C# web programming framework.

Lesson 1 – Using UI Frameworks

XAML (Extensible Application Markup Language) is a declarative markup language based on XML that can be used to build user interfaces for .NET applications. By using declarative markup instead of imperative code to define a user interface, you can make it more flexible and portable. This means that the same app can be used on many different devices, while separating your application logic from the presentation details of your application's user interface (UI).

This lesson will teach you how to use XAML to build simple graphical user interfaces (GUIs).

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the different flavours of XAML.
- Use XAML to express the layout of a UI.
- Create XAML controls.
- Handle events raised by user interaction.

Introducing XAML

Introducing XAML

- Use XML elements to create controls
- Use attributes to set control properties
- Create hierarchies to represent parent controls and child controls

```
<Button Content="Button1"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Height="44" Width="100"
        Click="click_handler"/>
```

XAML, in the most general sense, is a markup language for serializing .NET objects. It has been used in a number of Microsoft products, but its most important application is probably to define UI elements with an XML-based syntax. But its worth bearing in mind that there is nothing you can do in XAML that you couldn't do declaratively in code, creating instances of objects. For example, if we have a Button class in our UI framework, we could laboriously create it in code, something like:

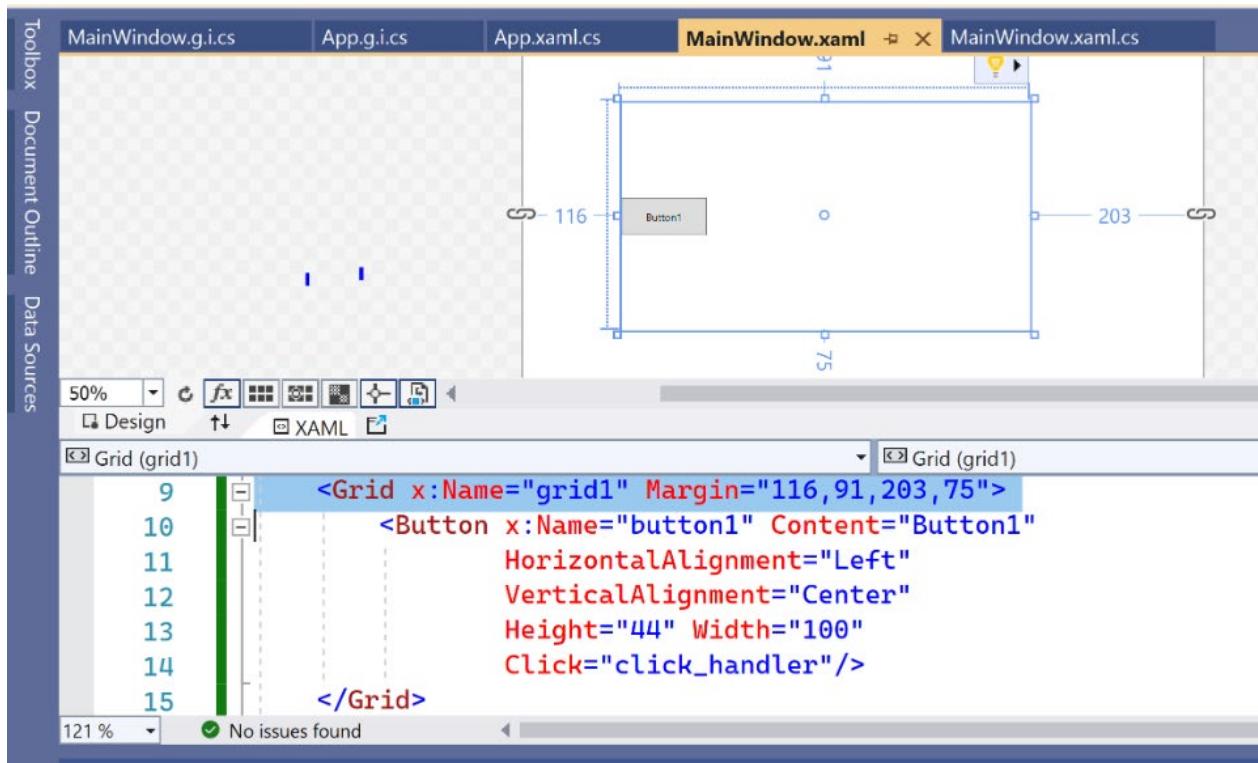
```
var button = new Button();
button.Content = "Button1";
button.Width = 100;
button.Height = 44;
button.HorizontalAlignment = HorizontalAlignment.Center;
button.VerticalAlignment = VerticalAlignment.Center;
button.Click += click_handler;
```

```
grid1.Children.Add(button);
```

Alternatively we could define it with the following markup:

```
<Button Content="Button1"  
       HorizontalAlignment="Center"  
       VerticalAlignment="Center"  
       Height="44" Width="100"/>
```

When you use XAML, you represent controls and their properties with XML elements and attributes. But Visual Studio can make it even easier, because you can use a built-in graphical editor. You can drag controls from a toolpane, position them, resize them, and edit them in a property panel. You can also edit the markup directly if needed. The screen capture below shows the Visual Studio XAML editor in use.



When you build the application, the build engine turns the XAML declarative markup into code that creates UI objects at runtime.

UI Frameworks

UI Frameworks

- WPF
- UWP
- Xamarin Forms
- .NET MAUI

As we mentioned before, XAML is just a .NET object serialization syntax. To build a UI, we need a UI framework that can be expressed in XAML. The most mature of these UI frameworks is WPF or Windows Presentation Foundation. This was introduced at about the same time as Windows Vista, and at the time was a new UI development model to replace Windows Forms. At the time, markup languages like XML were at the height of their technological popularity. WPF is primarily a UI platform for building Windows applications.

Subsequently, Microsoft has introduced a number of other XAML languages for building UIs. The Universal Windows Platform (UWP) was an initiative to offer a single development platform that could support all the versions of Windows, meaning desktop Windows, Windows RT for tablet devices, Windows Phone, Hololens and the Xbox computer game. Unfortunately, despite having a great user interface, Microsoft wasn't able to make Windows Phone a success. Hololens is a brilliant piece of technology but is primarily a niche product. And the use of Windows RT on smaller devices, which supported a Windows Store app distribution model but wasn't able to run older Win32 applications, hasn't really gained a huge amount of traction in the marketplace. Xbox has been very successful in its market, but is of limited interest in a business context, unless your work requires you to spend all day pretending to be a commando.

The two XAML dialects have sufficient differences, as do the resulting GUIs, so that you really need to focus on one or the other. Many people have concluded that a UWP app really isn't that "universal", and that they might as well stick with the more mature WPF. If they want cross-platform capabilities, then another option is Xamarin Forms, which is yet another XAML language. Xamarin is a company Microsoft bought after they developed a cross-platform mobile development framework based on the Mono open source implementation of the .NET Framework. Xamarin Forms apps can run on Android and iOS devices. The available controls are necessarily limited because they have to support two native platforms, although you can build your own controls (twice).

If you are planning a new project that needs this kind of cross-platform mobile reach, then it's probably worth waiting for .NET MAUI (Multi-platform Application User Interface), which is in preview at the time of writing. This framework is in the process of replacing Xamarin and eventually, presumably, WPF and UWP. The framework as a whole is capable of delivering .NET applications that will run on Windows, Apple Mac, Android and iOS. The UI is expressed in XAML, although there is currently no visual designer available.

In the rest of this section we'll see how to build a UI using the WPF version of XAML. In the longer term, it's likely that we'll be using .NET MAUI, which has a slightly different set of

controls and XAML dialect. Nevertheless, most of the principles, if not some of the precise details, will remain the same.

Note: You can find out more about .NET MAUI at: <https://aka.ms/3lubIDh>

WPF XAML

WPF XAML

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="400">
    <Grid>
        <Button x:Name="button1" Content="Order Coffee!"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                Height="44" Width="100"
                Click="click_handler"/>
    </Grid>
</Window>
```

In XAML for building a UI, such as WPF XAML, you define UI elements in markup. In a sense we are doing exactly the same thing as we do in HTML. An important difference is that HTML primarily deals with a **flow** of content, and HTML elements tend to flow as content on a page. There are HTML elements and CSS rules to do this kind of thing as well, but the main design goal of HTML was to be good at displaying content, such as you might find on a typical web page.

In contrast, when designing a UI, you are usually trying to position items on a screen according to some layout, but without necessarily knowing the exact dimensions you are going to be working with. This means that a UI design language usually has a set of controls for buttons, text boxes, and so forth, and a set of containers. The containers are given names like Stack, Grid, or WrapPanel. They are designed to arrange items in a constrained way, so that the layout makes sense in terms of the application's UI design, even with different sizes and dimensions of viewing area. In a UI language like WPF this is the main purpose of the set of controls provided in the toolkit.

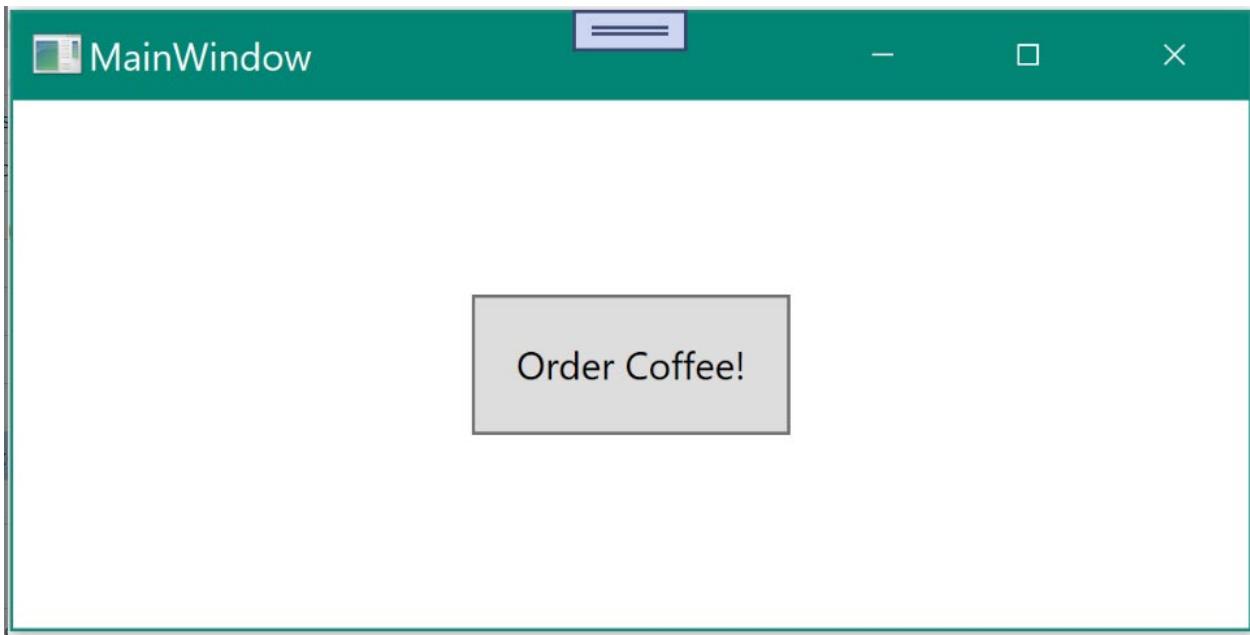
The example below shows an example of a WPF screen defined in XAML. Notice that there's a hierarchy of elements, starting from the outer Window, which contains a Grid, that is used to hold the Button:

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="200" width="400">
<Grid>
    <Button x:Name="button1" Content="Order Coffee!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Height="44" width="100"
        click="click_handler"/>
</Grid>
</window>
```

A **Window** element can have one child element that defines the content of the user interface (UI), as well as a number of attributes and namespace declarations. Most applications need more than one control for the user interface, hence the need for "container" controls that you can use to group and arrange other, lower-level controls. The Grid control is the most common container control, and it will try to lay out its controls in a grid arrangement according to the parameters you set. You can also nest containers inside each other if needed. When you add a new Window to a WPF project in Visual Studio, the Window template adds a Grid control to the Window automatically.

The designer will show a graphical version of the form. We can also debug the app, to verify that it is a big button on a blank background:



Question: What cross-platform UI framework does Microsoft offer that supports Windows, MacOS, iOS, and Android platforms?

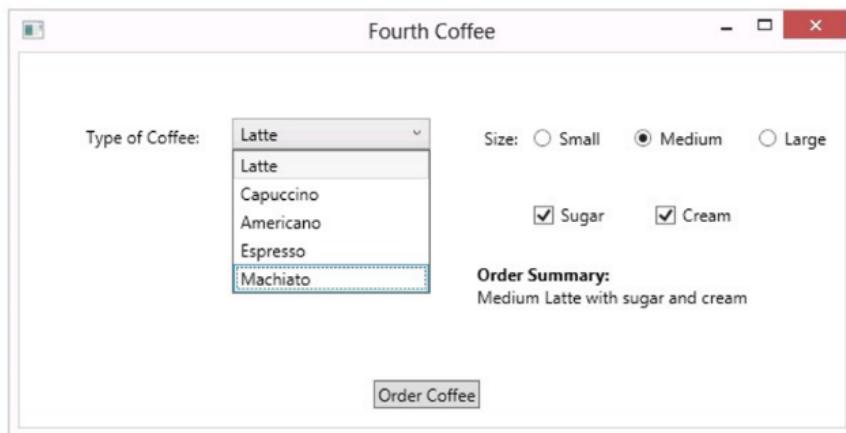
- A: WPF
- B: UWP
- C: .NET MAUI
- D: Xamarin

Answer: C

Common Controls

Common Controls

- Button
- Checkbox
- ComboBox
- Label
- ListBox
- RadioButton
- TabControl
- TextBlock
- TextBox



WPF gives you a rich set of controls to work with out-of-the-box. The Toolbox tab in Visual Studio gives you a palette from which you can drag controls onto the design surface. You can of course add them manually to the XAML if you prefer, or if you have some pre-existing XAML code you want to re-use. If the built-in controls don't do everything you need, you can also build your own custom controls.

The table lists some commonly used controls:

Control	Function
Button	User clickable button
CheckBox	Allow user set a true/false value

ComboBox	Show the user a drop-down list of items to select from
Label	Static text
ListBox	Show user a list of items to select from
RadioButton	Allow user to pick from a range of mutually exclusive options
TabControl	Organize controls in tabbed pages
TextBlock	Show multiline read-only text
TextBox	Show editable text

Setting Control Properties

Setting Control Properties

- Use attribute syntax to define simple property values


```
<Button Content="Click Me" Background="Yellow" />
```
- Use property element syntax to define complex property values


```
<Button Content="Click Me">
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5, 0.5"
        EndPoint="1.5, 1.5">
      <GradientStop Color="AliceBlue" Offset="0" />
      <GradientStop Color="Aqua" Offset="0.5" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

There are multiple ways to set the properties of the controls. Usually this is done with attributes:

```
<Button Content="Order Coffee!" Background="Aqua" Height="44"
Width="100" />
```

This should look familiar if you are used to working with HTML or XML. But this syntax doesn't quite give you the freedom you need to define property values that are more complicated. Suppose that, instead of setting the background of a button to a simple text value like

"Aqua", you wanted to give the button a complex gradient. You can't encapsulate all the details in a simple attribute value. To overcome this limitation, XAML lets you use something called "property element syntax" to set the properties of a control. Instead of making the Background property an inline attribute of the Button element, we add a child element called Button.Background. This then acts as a container for more elements that we can then use to create a gradient brush, as in the following XAML:

```
<Button Content="Order Coffee!" Height="44" width="100">  
  <Button.Background>  
    <LinearGradientBrush StartPoint="0.5, 0.5" EndPoint="1.5, 1.5" >  
      <GradientStop Color="LightGray" offset="0" />  
      <GradientStop Color="Aqua" offset="0.5" />  
    </LinearGradientBrush>  
  </Button.Background>  
  <Button.Foreground>  
    <SolidColorBrush Color="Black" />  
  </Button.Foreground>  
</Button>
```

It's probably fair to say that it's not going to win any design awards, but it works:



Most controls have a **Content** property. You can make use of the **Content** property, for example to add an image to the button. The content is also the equivalent of the XML contents of the element. By this means we can replace our gradient background with a jpg file containing a background pattern, or any other image for that matter. Notice that the syntax is similar, but not identical, to an HTML **img** element:

```
<Button>
<Image Source="images/background.jpg" Stretch="Fill" />
</Button>
```

Handling Events

Handling Events

- Specify the event handler method in XAML

```
<Button x:Name="btnMakeCoffee"
        Content="Make Me a Coffee!"
        Click="btnMakeCoffee_Click" />
```

- Handle the event in the code-behind class

```
private void btnMakeCoffee_Click(object sender,
                                RoutedEventArgs e)
{
    lblResult.Content = "Your coffee is on its way.";
}
```

- Events are bubbled to parent controls

Each XAML page in a WPF application has a code-behind page. When you create a new WPF project in Visual Studio, the first page you create is called **MainWindow.xaml**. That comes with a code-behind file **MainWindows.xaml.cs**.

When you want a UI element to trigger some behaviour, you attach an event handler to an event on the control. You can do this in Visual Studio in several ways, for example by double-clicking on the Button control. This will make a **Click** event handler and take you to the **MainWindow.xaml.cs** file, where a new event handler method will have been created.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

```
}
```

You can add code to this method in order to implement the desired behaviour. For example, you could add code to call a web service to order a coffee. You'll also notice that the Click event has been added to the MainWindow.xaml file:

```
<Button Content="Order Coffee!" Height="44" Width="100"  
Click="Button_Click">
```

This attribute on the Button object is what binds the event handler to the control's **Click** event.

Note: You can also manage events by going to the property pane and clicking on the events toggle (lightning bolt), and then adding a handler name for the event you are interested in.

Let's add some extra logic to the button click event handler. First we'll create a text label that we can update, underneath the button, in the MainWindow.xaml file. We can do this by going to the Visual Studio toolbox and dragging a Label control onto the design canvas. By default, the content gets a value of "Label". This new element doesn't have a name, which will make it difficult to find programmatically. We give it a name of "label1" by going to the properties and setting the Name (make sure the right control is selected before modifying the properties).

```
<Label x:Name="label1" Content="Label" HorizontalAlignment="Left"  
Margin="173,148,0,0" VerticalAlignment="Top"/>
```

Now we can update the Button_Click method with some code:

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    label1.Content = "Coffee order has been placed.";  
}
```

Now, when we click on the button, the label text will change.



Coffee order has been placed.

Note: WPF uses “routed events”, whereby an event will “bubble up” through the control hierarchy until it is handled. To learn more about routed events, see:
<https://aka.ms/3G2bgFC>.

Using Layout Controls

Using Layout Controls

- Canvas
- DockPanel
- Grid
- StackPanel
- VirtualizingStackPanel
- WrapPanel

One of the main goals of WPF is to support relative positioning, so that your app works correctly no matter how the user moves or changes the size of the app window. WPF has a number of layout controls, also called "container" controls, that let you place and size child controls in various ways. The most common controls for layout are shown in the table:

Container	Effect on Child Controls
Canvas	Fixed positioning by canvas coordinates

DockPanel	Attached to the edges of the DockPanel
Grid	Added to rows and columns on the grid
StackPanel	Stacked either vertically or horizontally.
VirtualizingStackPanel	Stacked either vertically or horizontally with only visible child items instantiated
WrapPanel	Added from left to right, or overflow to next line if too many to fit

Here's an example of defining a Grid layout. Notice how the definitions of the rows and columns are defined using property elements.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition MinHeight="100" MaxHeight="200" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="3*" />
    <ColumnDefinition Width="7*" />
  </Grid.ColumnDefinitions>
  <Label Content="This is Row 0, Column 0" Grid.Row="0"
        Grid.Column="0" />
  <Label Content="This is Row 0, Column 1" Grid.Row="0"
        Grid.Column="1" />
  <Label Content="This is Row 1, Column 0" Grid.Row="1"
        Grid.Column="0" />
  <Label Content="This is Row 1, Column 1" Grid.Row="1"
        Grid.Column="1" />
</Grid>
```

You get a number of ways of defining grids, such as prices grid units (corresponding to 1/96"), or a setting of Auto, which will use the minimum width needed to render the children.

You can also use a starred value, e.g. `Width="*"`, which tells the column to expand into the available remaining space after all the other columns have been allotted their space.

You use the `Grid.Row` and `Grid.Column` attributes to tell each child control which grid square to go into.

To learn more about the Grid control, see: <https://aka.ms/3wwBgG7>.

Creating User Controls

Creating User Controls

- To create a user control:
 - Define the control in XAML
 - Expose properties and events in the code -behind class
- To use a user control:
 - Add an XML namespace prefix for the assembly and namespace
 - Use the control like a standard XAML control

You can use XAML to make your own controls that are self-contained and reusable. They are sometimes called "composite controls" because they are made up of other controls. Suppose you use a certain combination of a combo box, a label, and a button together in a UI in multiple places. Rather than build it each time, it might be easier to make a user control. You could also distribute a user control for reuse by developers in different applications.

A user control is a XAML file along with its code-behind file. Unlike an application page, instead of using a `Window` control as the top-level element, a user control has a `UserControl` top-level element. If you right-click on your project in the Solution Explorer in Visual Studio, you can choose the `Add` menu item and then select `User Control (WPF)...`

```
<UserControl x:Class="WpfApp1.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="450" Width="800">
    <StackPanel Grid.Row="0">
        <Label Content="Do you want coffee or tea?" />
        <RadioButton x:Name="radCoffee" Content="Coffee"
            HorizontalAlignment="Left" />
        <RadioButton x:Name="radTea" Content="Tea"
            HorizontalAlignment="Left" />
    </StackPanel>
</UserControl>
```

```

    VerticalAlignment="Top" Margin="5" GroupName="Beverage"
    IsChecked="True" Checked="radCoffee_Checked" />
    <RadioButton x:Name="radTea" Content="Tea"
    HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="5" GroupName="Beverage"
    Checked="radTea_Checked"/>
    <Button x:Name="btnOrder" Content="Place Order" Margin="5"
    Grid.Row="3" Click="btnOrder_Click"/>
</StackPanel>
</UserControl>

```

The user control is scaffolded, and we have added some components to the `UserControl1.xaml` file, as listed above. Everything in the user control works in much the same way as for a regular Window. In the code-behind file, `UserControl1.xaml.cs`, you can define public properties, and those will then be accessible when the user control is used. You can also define public events. For example in the following code we've defined a public property **beverage**.

```

public partial class UserControl1 : UserControl
{
    public string beverage {
        get
        {
            if (radTea.IsChecked == true) return "Tea";
            if (radCoffee.IsChecked == true) return "Coffee";
            return "None";
        }
    }
    public UserControl1()
    {
        InitializeComponent();
    }
}

```

```
}
```

You add the user control to your XAML in the same way as a regular control, although you'll need to define a namespace prefix of the form `xmlns:[prefix]="clr-namespace:[namespace],[assembly name]` to pick it up in a different XAML file.

To learn more about WPF user controls, see: <https://aka.ms/3lwn5ui>.

Question: What XAML top-level element should a user control have?

- A: Window
- B: UserControl
- C: Control
- D: StackPanel

Answer: B

We've built a number of console applications to provide a simple framework to demonstrate a piece of code, and you can certainly build console apps for production use. But in general these will only appeal to other developers and IT personnel. If you want to make a successful app for end users to use on desktop computers or mobile platforms, you need to build some kind of graphical user interface. In this module, we'll look at some of the options for UI development, as well as considering some of the cross-platform development options.

Objectives

After completing this module, you'll be able to:

- Understand UI frameworks based on XAML.
- Use data binding for UI views.
- Be aware of some of the options for cross-platform UI development.
- Understand the Blazor C# web programming framework.

Lesson 1 – Using UI Frameworks

XAML (Extensible Application Markup Language) is a declarative markup language

based on XML that can be used to build user interfaces for .NET applications. By using declarative markup instead of imperative code to define a user interface, you can make it more flexible and portable. This means that the same app can be used on many different devices, while separating your application logic from the presentation details of your application's user interface (UI).

This lesson will teach you how to use XAML to build simple graphical user interfaces (GUIs).

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the different flavours of XAML.
- Use XAML to express the layout of a UI.
- Create XAML controls.
- Handle events raised by user interaction.

Introducing XAML

XAML, in the most general sense, is a markup language for serializing .NET objects. It has been used in a number of Microsoft products, but its most important application is probably

to define UI elements with an XML-based syntax. But it's worth bearing in mind that there is nothing you can do in XAML that you couldn't do declaratively in code, creating instances of objects. For example, if we have a Button class in our UI framework, we could laboriously create it in code, something like:

```
var button = new Button();
button.Content = "Button1";
button.Width = 100;
button.Height = 44;
button.HorizontalAlignment = HorizontalAlignment.Center;
button.VerticalAlignment = VerticalAlignment.Center;
button.Click += click_handler;
grid1.Children.Add(button);
```

Alternatively we could define it with the following markup:

```
<Button Content="Button1"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Height="44" Width="100"/>
```

When you use XAML, you represent controls and their properties with XML elements and attributes. But Visual Studio can make it even easier, because you can use a built-in graphical editor. You can drag controls from a toolpane, position them, resize them, and edit them in a property panel. You can also edit the markup directly if needed. The screen capture below shows the Visual Studio XAML editor in use.

When you build the application, the build engine turns the XAML declarative markup into code that creates UI objects at runtime.

UI Frameworks

As we mentioned before, XAML is just a .NET object serialization syntax. To build a UI, we need a UI framework that can be expressed in XAML. The most mature of these UI frameworks is WPF or Windows Presentation Foundation. This was introduced at about the same time as Windows Vista, and at the time was a new UI development model to replace Windows Forms. At the time, markup languages like XML were at the height of their technological popularity. WPF is primarily a UI platform for building Windows applications.

Subsequently, Microsoft has introduced a number of other XAML languages for building UIs. The Universal Windows Platform (UWP) was an initiative to offer a single development platform that could support all the versions of Windows, meaning desktop Windows, Windows RT for tablet devices, Windows Phone, Hololens and the Xbox computer game. Unfortunately, despite having a great user interface, Microsoft wasn't able to make Windows Phone a success. Hololens is a brilliant piece of technology but is primarily a niche product. And the use of Windows RT on smaller devices, which supported a Windows Store app distribution model but wasn't able to run older Win32 applications, hasn't really gained a huge amount of traction in the marketplace. Xbox has been very successful in its market, but is of limited interest in a business context, unless your work requires you to spend all day pretending to be a commando.

The two XAML dialects have sufficient differences, as do the resulting GUIs, so that you really need to focus on one or the other. Many people have concluded that a UWP app really isn't that "universal", and that they might as well stick with the more mature WPF. If they want cross-platform capabilities, then another option is Xamarin Forms, which is yet another XAML language. Xamarin is a company Microsoft bought after they developed a cross-platform mobile development framework based on the Mono open source implementation of the .NET Framework. Xamarin Forms apps can run on Android and iOS devices. The available controls are necessarily limited because they have to support two native platforms, although you can build your own controls (twice).

If you are planning a new project that needs this kind of cross-platform mobile reach, then it's probably worth waiting for .NET MAUI (Multi-platform Application User Interface), which is in preview at the time of writing. This framework is in the process of replacing Xamarin and eventually, presumably, WPF and UWP. The framework as a whole is capable of delivering .NET applications that will run on Windows, Apple Mac, Android and iOS. The UI is expressed in XAML, although there is currently no visual designer available.

In the rest of this section we'll see how to build a UI using the WPF version of XAML. In the longer term, it's likely that we'll be using .NET MAUI, which has a slightly different set of controls and XAML dialect. Nevertheless, most of the principles, if not some of the precise details, will remain the same.

Note: You can find out more about .NET MAUI at: <https://aka.gd/3lubIDh>

WPF XAML

In XAML for building a UI, such as WPF XAML, you define UI elements in markup. In a sense we are doing exactly the same thing as we do in HTML. An important difference is that HTML primarily deals with a **flow** of content, and HTML elements tend to flow as content on a page. There are HTML elements and CSS rules to do this kind of thing as well, but the main design goal of HTML was to be good at displaying content, such as you might find on a typical web page.

In contrast, when designing a UI, you are usually trying to position items on a screen according to some layout, but without necessarily knowing the exact dimensions you are going to be working with. This means that a UI design language usually has a set of controls for buttons, text boxes, and so forth, and a set of containers. The containers are given names like Stack, Grid, or WrapPanel. They are designed to arrange items in a constrained way, so that the layout makes sense in terms of the application's UI design, even with different sizes and dimensions of viewing area. In a UI language like WPF this is the main purpose of the set of controls provided in the toolkit.

The example below shows an example of a WPF screen defined in XAML. Notice that there's a hierarchy of elements, starting from the outer Window, which contains a Grid, that is used to hold the Button:

```
<window x:Class="wpfApp1.Mainwindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="200" width="400">
<Grid>
<Button x:Name="button1" Content="Order Coffee!"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Height="44" width="100"
click="click_handler"/>
</Grid>
</window>
```

A **Window** element can have one child element that defines the content of the user interface (UI), as well as a number of attributes and namespace declarations. Most applications need more than one control for the user interface, hence the need for "container" controls that you

can use to group and arrange other, lower-level controls. The Grid control is the most common container control, and it will try to lay out its controls in a grid arrangement according to the parameters you set. You can also nest containers inside each other if needed. When you add a new Window to a WPF project in Visual Studio, the Window template adds a Grid control to the Window automatically.

The designer will show a graphical version of the form. We can also debug the app, to verify that it is a big button on a blank background:

Question: What cross-platform UI framework does Microsoft offer that supports Windows, MacOS, iOS, and Android platforms?

- A: WPF
- B: UWP
- C: .NET MAUI
- D: Xamarin

Answer: C

Common Controls

WPF gives you a rich set of controls to work with out-of-the-box. The Toolbox tab in Visual Studio gives you a palette from which you can drag controls onto the design surface. You can of course add them manually to the XAML if you prefer, or if you have some pre-existing XAML code you want to re-use. If the built-in controls don't do everything you need, you can also build your own custom controls.

The table lists some commonly used controls:

Control	Function
Button	User clickable button
CheckBox	Allow user set a true/false value
ComboBox	Show the user a drop-down list of items to select from
Label	Static text
ListBox	Show user a list of items to select from

RadioButton	Allow user to pick from a range of mutually exclusive options
TabControl	Organize controls in tabbed pages
TextBlock	Show multiline read-only text
TextBox	Show editable text

Setting Control Properties

There are multiple ways to set the properties of the controls. Usually this is done with attributes:

```
<Button Content="Order Coffee!" Background="Aqua" Height="44"
Width="100" />
```

This should look familiar if you are used to working with HTML or XML. But this syntax doesn't quite give you the freedom you need to define property values that are more complicated. Suppose that, instead of setting the background of a button to a simple text value like "Aqua", you wanted to give the button a complex gradient. You can't encapsulate all the details in a simple attribute value. To overcome this limitation, XAML lets you use something called "property element syntax" to set the properties of a control. Instead of making the Background property an inline attribute of the Button element, we add a child element called Button.Background. This then acts as a container for more elements that we can then use to create a gradient brush, as in the following XAML:

```
<Button Content="Order Coffee!" Height="44" width="100">
<Button.Background>
<LinearGradientBrush StartPoint="0.5, 0.5" EndPoint="1.5, 1.5" >
<GradientStop Color="LightGray" offset="0" />
<GradientStop Color="Aqua" offset="0.5" />
</LinearGradientBrush>
</Button.Background>
<Button.Foreground>
<SolidColorBrush Color="Black" />
```

```
</Button.Foreground>  
</Button>
```

It's probably fair to say that it's not going to win any design awards, but it works:

Most controls have a **Content** property. You can make use of the **Content** property, for example to add an image to the button. The content is also the equivalent of the XML contents of the element. By this means we can replace our gradient background with a jpg file containing a background pattern, or any other image for that matter. Notice that the syntax is similar, but not identical, to an HTML **img** element:

```
<Button >  
  <Image Source="images/background.jpg" Stretch="Fill" />  
</Button>
```

Handling Events

Each XAML page in a WPF application has a code-behind page. When you create a new WPF project in Visual Studio, the first page you create is called `MainWindow.xaml`. That comes with a code-behind file `MainWindows.xaml.cs`.

When you want a UI element to trigger some behaviour, you attach an event handler to an event on the control. You can do this in Visual Studio in several ways, for example by double-clicking on the Button control. This will make a `Click` event handler and take you to the `MainWindow.xaml.cs` file, where a new event handler method will have been created.

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
}
```

You can add code to this method in order to implement the desired behaviour. For example, you could add code to call a web service to order a coffee. You'll also notice that the `Click` event has been added to the `MainWindow.xaml` file:

```
<Button Content="Order Coffee!" Height="44" Width="100"  
Click="Button_Click">
```

This attribute on the Button object is what binds the event handler to the control's **Click** event.

Note: You can also manage events by going to the property pane and clicking on the events toggle (lightning bolt), and then adding a handler name for the event you are interested in.

Let's add some extra logic to the button click event handler. First we'll create a text label that we can update, underneath the button, in the MainWindow.xaml file. We can do this by going to the Visual Studio toolbox and dragging a Label control onto the design canvas. By default, the content gets a value of "Label". This new element doesn't have a name, which will make it difficult to find programmatically. We give it a name of "label1" by going to the properties and setting the Name (make sure the right control is selected before modifying the properties).

```
<Label x:Name="label1" Content="Label" HorizontalAlignment="Left"  
Margin="173,148,0,0" VerticalAlignment="Top"/>
```

Now we can update the `Button_Click` method with some code:

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    label1.Content = "Coffee order has been placed.";  
}
```

Now, when we click on the button, the label text will change.

Note: WPF uses "routed events", whereby an event will "bubble up" through the control hierarchy until it is handled. To learn more about routed events, see:
<https://aka.ms/3G2bgFC>.

Using Layout Controls

One of the main goals of WPF is to support relative positioning, so that your app works correctly no matter how the user moves or changes the size of the app window. WPF has a number of layout controls, also called "container" controls, that let you place and size child controls in various ways. The most common controls for layout are shown in the table:

Container	Effect on Child Controls
Canvas	Fixed positioning by canvas coordinates
DockPanel	Attached to the edges of the DockPanel
Grid	Added to rows and columns on the grid
StackPanel	Stacked either vertically or horizontally.
VirtualizingStackPanel	Stacked either vertically or horizontally with only visible child items instantiated
WrapPanel	Added from left to right, or overflow to next line if too many to fit

Here's an example of defining a Grid layout. Notice how the definitions of the rows and columns are defined using property elements.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition MinHeight="100" MaxHeight="200" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="3*" />
    <ColumnDefinition Width="7*" />
  </Grid.ColumnDefinitions>
  <Label Content="This is Row 0, Column 0" Grid.Row="0"
        Grid.Column="0" />
  <Label Content="This is Row 0, Column 1" Grid.Row="0"
        Grid.Column="1" />
  <Label Content="This is Row 1, Column 0" Grid.Row="1"
        Grid.Column="0" />
```

```
<Label Content="This is Row 1, Column 1" Grid.Row="1"  
Grid.Column="1" />  
</Grid>
```

You get a number of ways of defining grids, such as prices grid units (corresponding to 1/96"), or a setting of Auto, which will use the minimum width needed to render the children. You can also use a starred value, e.g. Width="*", which tells the column to expand into the available remaining space after all the other columns have been allotted their space.

You use the Grid.Row and Grid.Column attributes to tell each child control which grid square to go into.

To learn more about the Grid control, see: <https://aka.ms/3wwBgG7>.

Creating User Controls

You can use XAML to make your own controls that are self-contained and reusable. They are sometimes called "composite controls" because they are made up of other controls. Suppose you use a certain combination of a combo box, a label, and a button together in a UI in multiple places. Rather than build it each time, it might be easier to make a user control. You could also distribute a user control for reuse by developers in different applications.

A user control is a XAML file along with its code-behind file. Unlike an application page, instead of using a Window control as the top-level element, a user control has a **UserControl** top-level element. If you right-click on your project in the Solution Explorer in Visual Studio, you can choose the **Add** menu item and then select **User Control (WPF)**...

```
<UserControl x:Class="wpfApp1.UserControl1"  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Height="450" Width="800">  
<StackPanel Grid.Row="0">  
<Label Content="Do you want coffee or tea?" />  
<RadioButton x:Name="radCoffee" Content="Coffee"  
HorizontalAlignment="Left"  
VerticalAlignment="Top" Margin="5" GroupName="Beverage"  
IsChecked="True" Checked="radCoffee_Checked" />
```

```

<RadioButton x:Name="radTea" Content="Tea"
HorizontalAlignment="Left"
VerticalAlignment="Top" Margin="5" GroupName="Beverage"
Checked="radTea_Checked"/>

<Button x:Name="btnOrder" Content="Place Order" Margin="5"
Grid.Row="3" Click="btnOrder_Click"/>

</StackPanel>
</UserControl>

```

The user control is scaffolded, and we have added some components to the `UserControl1.xaml` file, as listed above. Everything in the user control works in much the same way as for a regular Window. In the code-behind file, `UserControl1.xaml.cs`, you can define public properties, and those will then be accessible when the user control is used. You can also define public events. For example in the following code we've defined a public property **beverage**.

```

public partial class UserControl1 : UserControl
{
    public string beverage {
        get {
            if (radTea.IsChecked == true) return "Tea";
            if (radCoffee.IsChecked == true) return "Coffee";
            return "None";
        }
    }

    public UserControl1()
    {
        InitializeComponent();
    }
}

```

You add the user control to your XAML in the same way as a regular control, although you'll need to define a namespace prefix of the form `xmlns:[prefix]="clr-namespace:[namespace],[assembly name]` to pick it up in a different XAML file.

To learn more about WPF user controls, see: <https://aka.ms/3lwn5ui>.

Question: What XAML top-level element should a user control have?

- A: Window
- B: UserControl
- C: Control
- D: StackPanel

Answer: B

Lesson 2 – Data binding

Most applications work in some way with data, whether it's from a database, a web service, or just the filesystem. An app that has a graphical user interface needs to connect UI controls to the underlying data source so that users can view, enter, or change the data. You can write ad-hoc logic to update views, and keep everything in synchronization using event handlers, but it becomes increasingly complex and error-prone for anything but the simplest of situations. A better and more robust solution is to use data binding.

In this lesson, you'll learn how to connect controls in WPF applications to underlying data by using data binding.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain data binding in principle.
- Use data binding in WPF.
- Set up data binding in XAML and in code.
- Use data templates to control how data is displayed.

Introduction to Data Binding

Introduction to Data Binding

- Data binding has three components:
 - Binding source
 - Binding target
 - Binding object
- A data binding can be bidirectional or unidirectional:
 - TwoWay
 - OneWay
 - OneTime
 - OneWayToSource
 - Default

Data binding is the process of linking a data source to a UI element so that the two stay in synchronization. We can define the data binding **source** as the data that is being represented in the user interface; the data binding **target** as the UI control that will display, and possibly allow the user to edit, the data; and the data binding **object** that makes the connection, and that might also include a conversion if the source and target have different types.

To work as a data binding target, a control must have a special kind of property called a “dependency property”. These properties are used by the .NET runtime data binding mechanism to trigger updates when their values change. Most UI properties that you’re likely to want to bind to are dependency properties.

To learn more about dependency properties, see: <https://aka.ms/3Gc8ePq>.

You can create a data binding using a binding expression in XAML. For example, we could set use a data binding expression instead of a static string for a TextBlock element:

```
<TextBox Text="{Binding Source={StaticResource beverage},  
Path=Name, Mode=TwoWay}" />
```

This binding expression will bind the text inside the text box to the **Name** property of the **beverage** object. The effect of this is that the TextBlock will display whatever is in `beverage.Name`, even as that value changes. But also, if the user changes the value of the text in the TextBox, the value of `beverage.Name` will be updated. This is because the **Mode** was set to **TwoWay**. This gives us two-way data binding between the two objects, which is very powerful. In addition to **TwoWay** binding, you could also choose **OneWay** (updates target from source whenever source changes), **OneTime** (updates the target from the source when the application starts or when binding is

established), **OneWayToSource** (updates the source from the target whenever the target changes), and **Default** (uses the default mode defined for the target property).

To learn more about data binding in WPF, see: <https://aka.ms/3wx90rW>.

Binding Controls to Data in XAML

Binding Controls to Data in XAML

- Use a binding expression to identify the source object and the source property

```
<TextBlock  
    Text="{Binding Source=$staticResource coffee1},  
    Path=Name}" />
```

- Specify the data context on a parent control

```
<StackPanel>  
    <StackPanel.DataContext>  
        <Binding Source="$staticResource coffee1" />  
    </StackPanel.DataContext>  
    <TextBlock Text="{Binding Path=Name}" />  
    ...  
</StackPanel>
```

You can see two-way binding very visually simply by creating two TextBox controls and databinding one to the other:

```
<StackPanel>  
  
<TextBox x:Name="textbox1" Text="{Binding ElementName=textbox2,  
Path=Text, Mode=OneWay}" />  
  
<TextBox x:Name="textbox2" Text="{Binding ElementName=textbox1,  
Path=Text, Mode=OneWay}" />  
  
</StackPanel>
```

You can use the above sample to try different data binding options, like having one-way binding between textbox1 and textbox2, but not the other way round. If you feel brave, you can try using two-way binding on both textboxes; will it result in some kind of infinite recursion? Fortunately, the framework is clever enough to stop you from getting into this kind of trap by detecting endless loops, usually.

If your source data is going to be static at runtime, you can use a **static resource** and define that in the XAML. This could, for example, be an instance of our Coffee class. To do this, you need to add a Resources property element to any containing element, for example the Window. You can then add your object using a namespace prefix declaration, and add an x:Key attribute:

```
<Window x:Class="DataBinding.Mainwindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:loc="clr-namespace>DataBinding"
Title="Data Binding Example" Height="350" width="525">
<Window.Resources>
<loc:coffee x:Key="coffee1" Name="Fourth Coffee Quencher"
Bean="Arabica" Countryoforigin="Brazil" Strength="3" />
</Window.Resources>
<Grid>
<TextBlock Text="{Binding Source={StaticResource coffee1},
Path=Name}" />
</Grid>
</window>
```

In the above example, we've used the static resource as the source in our data binding for the Text property of the TextBlock.

A very common type of page, such as a data entry form, will involve multiple controls on the page that are bound to different properties of the same underlying data object. To make this easier, you can use a DataContext property element on an outer container element (in this case a StackPanel), so that the context is available to all the controls inside it. In the example below we've added a static resource as the data context, and then bound the controls to various properties:

```
<StackPanel>
<StackPanel.DataContext>
<Binding Source="{StaticResource coffee1}" />
</StackPanel.DataContext>
```

```
<TextBlock Text="{Binding Path=Name}" />
<TextBlock Text="{Binding Path=Bean}" />
<TextBlock Text="{Binding Path=CountryofOrigin}" />
<TextBlock Text="{Binding Path=Strength}" />
</StackPanel>
```

Binding Controls to Data in Code

Binding Controls to Data in Code

- Create data binding entirely in code
- Create Path bindings in XAML and set the DataContext in code

```
<StackPanel x:Name="stackCoffee">
<TextBlock Text="{Binding Path=Name}" />
<TextBlock Text="{Binding Path=Bean}" />
<TextBlock Text="{Binding Path=CountryOfOrigin}" />
<TextBlock Text="{Binding Path=Strength}" />
</StackPanel>
```

```
stackCoffee.DataContext = coffee1;
```

Static binding doesn't really solve the problem of binding controls to data that's been retrieved from a database or other remote service. In that case you usually need to run some code that will set up the bindings at runtime. In these cases you can do the bulk of the configuration declaratively in advance using XAML, and just use procedural code to make the final connection to the data.

Let's consider the previous example where we had a StackPanel with a number of TextBlocks bound to various properties. Rather than use a DataContext defined in XAML, we'll just make sure that the StackPanel has a name so that we can get a handle to it in code:

```
<StackPanel x:Name="stack1">
<TextBlock Text="{Binding Path=Name}" />
<TextBlock Text="{Binding Path=Bean}" />
<TextBlock Text="{Binding Path=Countryoforigin}" />
```

```
<TextBlock Text="{Binding Path=Strength}" />  
</StackPanel>
```

The binding for each `TextBlock` has now been set up, but at the moment this won't work, because there is no `DataContext` defined. We can now do that in the code-behind file. At the point where the data has been retrieved and is available, we can simply get the `StackPanel` instance, and set its `DataContext` property:

```
stack1.DataContext = beverage1;
```

Binding a Control to a Collection

Binding a Control to a Collection

- Set the `ItemsSource` property to bind to an `IEnumerable` collection

```
listbox1.ItemsSource = beverages;
```

- Use the `DisplayMemberPath` property to specify the source field to display

```
<ListBox x:Name="listbox1"  
        DisplayMemberPath="Name" />
```

So far, we've bound a control to a property of an object. Sooner or later you'll need to render a list of objects using a `ListBox`, `ListView`, `TreeView`, or some other custom control that can render a collection. All these controls derive from the `ItemsControl` class, which has an `ItemsSource` property that you bind to the underlying collection, and a `DisplayMemberPath` property that you use to specify the source property to display. The only constraint on the collection class is that it needs to implement the `IEnumerable` interface.

You can set the `ItemsSource` and `DisplayMemberPath` properties in code or in the XAML file. For example, you could set the `DisplayMemberPath` in XAML, but defer the actual binding to the `ItemsSource` until the data is loaded, by means of procedural code. Here's an example of the XAML:

```
<ListBox x:Name="listbox1" DisplayMemberPath="Name" />
```

Assuming we have a list of `Beverage` instances in a `List<Beverage>` called `beverages`, we can then bind that to the control programmatically:

```
listbox1.ItemsSource = beverages;
```

This will populate the contents of **listbox1**, but it won't give us the kind of dynamic binding that would cause the list to automatically update itself if the underlying list changed. For this, implementing **IEnumerable** is not enough; we need to implement the **INotifyPropertyChanged** interface. You can achieve this by using a class that derives from **ObservableCollection<T>**. Using a collection that implements **INotifyPropertyChanged** will raise a **PropertyChanged** event when the collection is updated, and the list control will handle this event to update its contents.

To read more about observable collections, see: <https://aka.gd/3a2hwkQ>.

Creating Data Templates

Creating Data Templates

- Specify how each item in a collection should be displayed

```
<DataTemplate>
  <Grid>
    ...
    <TextBlock Text="{Binding Path=Name}" Grid.Row="0"
      FontSize="22" Background="Black"
      Foreground="White" />
    <TextBlock Text="{Binding Path=Temperature}"
      Grid.Row="1" />
    <TextBlock Text="{Binding Path=CountryOfOrigin}"
      Grid.Row="2" />
  </Grid>
</DataTemplate>
```

UI controls that are based on **ItemsControl** or **ContentControl** can have a data template specifies how your items should be displayed. Let's say, for instance, that you want to use a **ListBox** control to show a list of **Beverage** objects. Each **Beverage** instance has a number of properties for name, temperature, and the country where the coffee came from. You might want to show these individual properties in different fonts, colours, etc. Perhaps the **Name** property should be formatted as a title. By mapping the properties of each **Beverage** instance to child controls in the data template, the data template fully defines how each individual **Beverage** instance should be shown and styled.

In the following XAML we've defined a data template within a **ListBox**:

```
<ListBox x:Name="lstCoffees" width="200">
```

```
<ListBox.ItemTemplate>
<DataTemplate>
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="2*" />
<RowDefinition Height="*" />
<RowDefinition Height="*" />
</Grid.RowDefinitions>
<TextBlock Text="{Binding Path=Name}" Grid.Row="0"
FontSize="22" Background="Black" Foreground="White" />
<TextBlock Text="{Binding Path=Temperature}" Grid.Row="1" />
<TextBlock Text="{Binding Path=Countryoforigin}" Grid.Row="2" />
</Grid>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
```

For more about data templates in WPF, see: <https://aka.gd/3NpwwaN>.

Question: You need to configure data binding so that it only updates the target from source whenever source changes. Which mode should you choose?

- A: TwoWay
- B: OneWay
- C: OneTime
- D: OneWayToSource

Answer: B

Lesson 3 – Styling the UI

In the previous lesson we added some inline style to UI elements. As anyone who is familiar with HTML knows, inline style is a bad idea, and you need to put styling in a separate, maintainable location. In fact HTML has a special language called CSS that is just for styling. There isn't an equivalent to CSS in XAML, but there are various resources that you can create in XAML that save you having to make the same changes in multiple places. You can define styles as reusable resources that can be used on more than one control.

We'll see how to use styles and animations in this lesson.

Lesson Objectives

After completing this lesson, you'll be able to:

- Create reusable XAML resources.
- Apply style resources to more than one control.
- Make use of animations to declaratively build a dynamic UI.

Creating Reusable Resources in XAML

Creating Reusable Resources in XAML

- Define resources in a Resources collection
- Add an x:Key to uniquely identify the resource

```
<Window.Resources>
    <SolidColorBrush x:Key="brush1" Color="Aqua" />
    ...
</Window.Resources>
```

- Reference the resource in property values

```
<TextBlock Text="Foreground"
    Foreground="{StaticResource brush1}" />
```

- Use a resource dictionary to manage large collections of resources

We've seen how XAML allows you to use data templates and sources as reusable resources. You can also use the same approach to make reusable styling resources. This isn't always necessary – a one-off piece of styling can be applied directly to the element and that's perfectly reasonable. But when you start to see the same bits of styling code being repeated, it's a sure sign that it's time to bring the styling into a single resource. Bringing this code into

one place means that it can be edited, without needing to revisit every control that uses that style. It will make the XAML more manageable and less repetitive.

You can add resources to the Resources property of any control, because all the controls are ultimately derived from the FrameworkElement class. Typically, you define resources on the XAML file's root element, like the Window or UserControl element, because the resources will then be available to every control. You can also define resources that are available application-wide, by adding them to the App.xaml file. As well as being the entry point for your application, the App.xaml file can be used for any globally-available resources you might need.

The following sample XAML shows how a solid brush can be defined as a resource:

```
<window x:Class="DataBinding.Mainwindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Resources Example" Height="350" width="525">
<window.Resources>
<SolidColorBrush x:Key="brush1" color="Aqua" />
</window.Resources>
<!-- controls that can use MyBrush -->
</window>
```

You use the syntax **{StaticResource[resourcekey]}** to use the resource. If the resource is in scope, you can use it as the value of any property that accepts the same type as the resource. Using the brush defined above, we could use it in any property that accepts brush types, like **Fill**, **Foreground**, or **Background**. The following sample shows this:

```
<StackPanel>
<Button Content="Click Me" Background="{StaticResource brush1}" />
<TextBlock Text="Foreground" Foreground="{StaticResource brush1}" />
<TextBlock Text="Background" Background="{StaticResource brush1}" />
<Ellipse Height="50" Fill="{StaticResource brush1}" />
</StackPanel>
```

If we ever needed to change the background colour, we'd only need to modify the resource itself.

If you need to make several resources that can be used more than once, it can be helpful to do so in a separate file called a "resource dictionary." A resource dictionary is a XAML file with a **ResourceDictionary** element at the root. You can add resources that can be used more than once to the **ResourceDictionary** element just as you would add them to a **Window.Resources** element. Generally, you'd want to make that available application-wide by putting it in the App.xaml file:

```
<Application x:Class="ReusableResources.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="FourthCoffeeResources.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Defining Styles as Resources

Defining Styles as Resources

- Identify the target control type
- Provide an x:Key value if required
- Use Setter elements to specify property values

```
<Style TargetType="TextBlock" x:Key="BlockStyle1">
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Background" Value="Black" />
    ...
</Style>
```

- Reference the style as a static resource

```
<TextBlock Text="Order Coffee!" 
    Style="{StaticResource BlockStyle1}" />
```

You often find you have many controls which need the same set of property values in order to give them a consistent style. It's quite likely that you'd want all your controls on a form to have the same foreground colour, background colour, font, etc. You can make the management of styles easier by using **Style** elements as resources. A **Style** element lets you apply multiple property values to a control in one go. You can define a **Style** element in the same way as other resources, and then reference them in controls by means of the x:Key attribute. The **Style** element contains a set of **Setter** elements that are used to define the individual properties.

```
<Window.Resources>

<Style TargetType="TextBlock" x:Key="BlockStyle1">
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Background" Value="Black" />
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Offset="0.0" Color="Orange" />
                    <GradientStop Offset="1.0" Color="Red" />
                </LinearGradientBrush.GradientStops>
            </Setter.Value>
        </Setter>
    </Setter>
</Style>
```

```
</LinearGradientBrush>
</Setter.value>
</Setter>
</Style>
</Window.Resources>
```

Having defined a style, you can then use the style on a control, wherever it's in scope:

```
<TextBlock Text="Order Coffee!" style="{StaticResource
BlockStyle1}" />
```

To learn more about styles in WPF, see: <https://aka.ms/3LFkEk0>.

Using Property Triggers

Using Property Triggers

Use triggers to apply style properties based on conditions:

- Use the **Trigger** element to identify the condition
- Use **Setter** elements apply the conditional changes

```
<Style TargetType="Button">
  <Style.Triggers>
    <Trigger Property='IsMouseOver' Value="True">
      <Setter Property="FontWeight" Value="Bold" />
    </Trigger>
  </Style.Triggers>
</Style>
```

One way of adding behaviour to your application is to implement it in code. However, there are quite a few things you can do declaratively, such as change the appearance of a control when the user hovers over it, or interacts with it. If you want to modify styling properties in response to user actions, you can add **Trigger** elements to your styles. A **Trigger** element is defined in terms of the **Property** that should trigger the change, and the **Property** that needs to be changed. The latter is defined in a **Setter** element inside the **Trigger**. You can think of the **Trigger** element as being a special wrapper for the **Setter** that modifies its effect inside the **Style**.

Let's imagine that we wanted a button to become bold when the user hovers over it:

```
<window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="400">
    <Window.Resources>
        <Style TargetType="Button">
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="FontWeight" Value="Bold" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <Grid>
        <Button Content="Order Coffee!" Width="100" Height="50" />
    </Grid>
</window>
```

Creating Dynamic Transformations

Creating Dynamic Transformations

- Use an EventTrigger to identify the event that starts the animation
- Use a Storyboard to identify the properties that should change
- Use a DoubleAnimation to define the changes

```
<EventTrigger RoutedEvent="Image.MouseDown">
  <BeginStoryboard>
    <Storyboard>
      <DoubleAnimation
        Storyboard.TargetProperty="Height"
        From="200" To="300" Duration="0:0:2" />
    </Storyboard>
  </BeginStoryboard>
</EventTrigger>
```

Applications often use animations to make them more appealing to users. Animations can provide affordance to user interface controls, or indicate to the user the effects of actions. An example is where you might make a UI element become slightly larger, or acquire a glow, when the user hovers over it, in order to convey to the user that the element supports some kind of interaction. Used wisely, these techniques can make functionality discoverable, and improve the user experience. If used to excess, this kind of animation can make the UI look gimmicky or distracting. Generally it's a good idea to keep these animations subtle and unobtrusive.

A lot of animation effects can be achieved directly in XAML, without necessarily requiring any code. The first task is to add an element that supports animation such as **DoubleAnimation**. This element can specify how a property evolves over time. You supply a start and end value and a duration. You then need to enclose your animation element in a **Storyboard** element that will define the target element with the **TargetName** property, and the property on the target element by means of the **TargetProperty** property. You then need to create a trigger for the animation by wrapping the **Storyboard** element in an **EventTrigger** element.

The sample below shows how to make a control rotate and expand when clicked on:

```
<window x:Class="WpfApp1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="200" Width="400">
  <Window.Resources>
    <Style TargetType="Button" x:Key="style1">
```

```
<Setter Property="Height" value="200" />
<Setter Property="RenderTransformOrigin" value="0.5,0.5" />
<Setter Property="RenderTransform">
<Setter.value>
<RotateTransform Angle="0" />
</Setter.value>
</Setter>
<Style.Triggers>
<EventTrigger RoutedEvent="Button.Click">
<BeginStoryboard>
<Storyboard>
<DoubleAnimation Storyboard.TargetProperty="Height"
From="50" To="70" Duration="0:0:2" />
<DoubleAnimation
Storyboard.TargetProperty="RenderTransform.Angle"
From="0" To="30" Duration="0:0:2" />
</Storyboard>
</BeginStoryboard>
</EventTrigger>
</Style.Triggers>
</Style>
</window.Resources>
<Grid>
<Button Content="Order Coffee!" width="100" Height="50"
Style="{StaticResource style1}" />
</Grid>
</window>
```

Again, this is unlikely to win any awards for UI design, but shows the technique.

To find out more about WPF animations, and discover more examples, see:
<https://aka.gd/3MxITSr>.

Lab: Customizing Student Photographs and Styling the Application

Lab: Customizing Student Photographs and Styling the Application

Lab scenario

Now that you and The School of Fine Arts are happy with the basic functionality of the application, you need to improve the appearance of the interface to give the user a nicer experience through the use of animations and a consistent look and feel.

You decide to create a StudentPhoto control that will enable you to display photographs of students in the student list and other views. You also decide to create a fluid method for a teacher to remove a student from their class. Finally, you want to update the look of the various views, keeping their look consistent across the application.

Objectives

Exercise 1:
Customizing the Appearance of Student Photographs

Exercise 2:
Styling the Logon View

Exercise 3:
Animating the StudentPhotoControl (If Time Permits)

Module Review and Takeaways

Review Questions

Question: You want to use rows and columns to lay out a UI. Which container control should you use?

- A: The DockPanel control.
- B: The Canvas control.

C: The WrapPanel control.

D: The Grid control.

E: The StackPanel control.

Answer: D, the Grid control lets you define rows and columns, and position child controls using the Grid.Row and Grid.Column attached properties.

Module 10 – Application Performance

Module Overview

Very early computers were designed to perform a single task and then stop. But the need to maximize the use of such a valuable resource meant that multi-tasking operating systems were soon devised. These meant that jobs could be run concurrently, so that every second of the computer's time could be used efficiently. They can do this by periodically switching from one task to another, while storing the state of the CPU for each task so that it can be restored when that task's "turn" comes round again. Thus, the computer's resources can be shared between multiple running programs.

Modern processors have taken this a step further by having multiple CPU cores, so that concurrent applications can be run on their own processor, while sharing memory and peripheral devices. In the meantime, multi-tasking has further evolved into multi-threading, where a single application can have multiple concurrent paths of execution. This enables much better performance, particularly with graphical user interfaces where the UI needs to process user interactions while simultaneously performing calculations. Modern responsive user interfaces would be impossible without multi-threading, as would be any application that depended on a laggy network connection. And if an algorithm can be "parallelized" in such a way that a computation can be broken down into sections that can be performed concurrently, then a multiprocessor architecture can deliver even bigger performance improvements.

But multi-threading comes at the cost of complexity. When an application follows a single thread of execution, it's easy to plan and predict the order in which code will execute. But as soon as you introduce multi-threading, life becomes much more complicated. Events occur in an unpredictable order. A method may be called before a particular variable has been assigned, an object pointer might still be null, and the user interface component that you are updating might not yet have been created.

In this module we'll learn how to use threads safely while optimizing application performance.

Objectives

After completing this module, you'll be able to:

- Use multitasking with the Task Parallel Library.
- Manage long-running operations without blocking the UI.
- Manage multiple threads and concurrent resource access.

Lesson 1 – Multitasking

Applications that have any kind of user interface execute code when something happens; a user clicks a button or a menu item. If the code is trivial, like updating the UI, then the code can run on the UI thread. But if there is any danger that the code might take a significant amount of time, such that a user would notice the delay, it should be run on a separate thread. Otherwise the UI will become unresponsive. We've all used applications that were written in this way. You initiate a calculation, or a network call, and the UI freezes up. Next thing you know you're opening Task Manager to figure out what the problem is.

The Task Parallel Library is part of .NET that provides a set of classes that make it easy to split your code execution into different threads. It allows you to use the parallelism that this model gives you to run multiple threads on different processor cores, or to put long-running tasks on a separate thread, allowing the UI to remain responsive.

In this lesson, you'll learn how to use the Task Parallel Library to make applications that are able to make efficient use of multiple processor cores and continue to respond to user input.

Lesson Objectives

After completing this lesson, you'll be able to:

- Create tasks on a different thread.
- Manage task execution.
- Communicate between processes.
- Run tasks in parallel.
- Handle exceptions in a multi-threaded application.

Creating Tasks

Creating Tasks

- Use an Action delegate

```
Task task1 = new Task(new Action(MyMethod));
```

- Use an anonymous delegate/anonymous method

```
Task task2 = new Task(delegate
{
    Console.WriteLine("Task 2 reporting");
});
```

- Use lambda expressions (recommended)

```
Task task2 = new Task(() =>
{
    Console.WriteLine(" Task 2 reporting");
});
```

The fundamental class in the Task Parallel Library is the **Task**. A Task represents a unit of work or an execution path. You can have multiple such tasks each running on their own thread, which can be thought of as a lightweight process. The Task Parallel Library manages a pool of available threads, starts and stops tasks, and orchestrates task execution.

Note: A **thread** is not quite the same thing as a **process**. Both are a separate path of execution, but a process is typically an execution context that is managed by the operating system, and usually has its own address space. A thread is a much lighter weight construct, that is quickly created, is typically managed by an application, and often shares its address space with other threads.

To create a **Task** you need to call the **Task** constructor, and pass a block of code to the constructor by means of a delegate. The code below shows a task being created using the Action delegate, and a method definition that matches the delegate:

```
Task task1 = new Task(new Action(GetTheTime));

task1.Start();

static void GetTheTime()
{
    Console.WriteLine("The time now is {0}", DateTime.Now);
}
```

After creating the task, we call the Task.Start() method to start the task running. If you run this code as a simple console app, you make look at the output expecting to see the time displayed. Instead, you get nothing. What's going on?

The problem is that the code in the GetTheTime method is running asynchronously. When we call Start(), the task starts to execute, but the Start() method returns immediately. At this point the program exits, but the task is still running. The Console.WriteLine is about to get called, but the program has already finished! The process has already gone before anything gets sent to the terminal. If we want to see the output we need to do something like add task1.Wait() after the call to task1.Start(). This will make execution on the main thread pause at this point and wait for the code on the Task thread to complete. Then the program ends. If you make that change then you'll see the time output to the console.

Note: .NET provides a couple of delegates to use to pass code to the Task. The **Action** delegate has a void return value and a variable number of generic parameters, and the **Func** delegate that can return a value. You can also create your own delegate.

In order to have some code to pass to the Task constructor, we had to create a method to provide to the delegate. Creating a method is a good option where code is being re-used. But for an ad-hoc piece of code it's a bit clumsy to keep creating new methods and thinking up names for them. In this case it's usually better to use anonymous methods or lambda expressions. To write the above code using an anonymous method we could write:

```
Task task1 = new Task(delegate {
    Console.WriteLine("The time now is {0}", DateTime.Now);
});
```

Precisely where you choose to break the lines and how to indent is largely a matter of personal programming style. The same thing can be achieved using a lambda expression, which is really just more concise way of writing an anonymous delegate. The syntax is to provide the input parameters in a pair of parentheses (if there's only one parameter, you can omit the parentheses), followed by a “fat arrow” symbol (=>), and then either an expression return, or a method body.

The lambda expression version of the above is as follows:

```
Task task1 = new Task(() => {
    Console.WriteLine("The time now is {0}", DateTime.Now);
});
```

To learn more about lambda expressions in C#, see: <https://aka.ms/3wDWUXJ>.

Controlling Task Execution

Controlling Task Execution

- To start a task:
 - `Task.Start` instance method
 - `Task.Factory.StartNew` static method
 - `Task.Run` static method
- To wait for tasks to complete:
 - `Task.Wait` instance method
 - `Task.WaitAll` static method
 - `Task.WaitAny` static method

Once you have a `Task` object you have several ways of actually starting the execution of the task. You also get methods to control the task. We already saw one way of starting a task, which is to call the `Start()` method on the `Task` object. When you do this, the Task Parallel Library immediately assigns a thread to your task a thread and begins execution. Because the task is running on its own thread, the code that called `Start()` doesn't have to wait for it to finish. Instead, the `Start()` method returns immediately and the task and the code that started it keep running at the same time.

You can verify this by running the following code:

```
var task1 = new Task(() =>
    Console.WriteLine("Task code finishing")
);
task1.Start();
Console.WriteLine("Calling code finishing");
task1.Wait();
```

You might expect the two messages to print to the console in the same order as in the code, but you will probably see the following when you run this:

```
Calling code finishing
Task code finishing
```

The calling code executes before the code in the task. We added a **Wait()** method call to ensure that the program didn't exit before the task had a chance to run.

Calling **task1.Wait()** causes the main thread to pause until the **task1** thread finishes.

Another option is to use the **TaskFactory** static class to set everything up in one go:

```
var task1 = Task.Factory.StartNew(() => Console.WriteLine("Task code finishing"));

Console.WriteLine("Calling code finishing");

task1.Wait();
```

You can also use **Task.Run** which is just a shortcut version of **Task.Factory.StartNew**:

```
var task1 = Task.Run(() => Console.WriteLine("Task code finishing"));
```

If you have multiple threads running, you can make your main thread wait for all of them to finish. You just need to call the **Task.WaitAll** method and pass it an array of threads:

```
Task[] tasks = new Task[3]

{
    Task.Run( () => Console.WriteLine("Task code finishing 1")),
    Task.Run( () => Console.WriteLine("Task code finishing 2")),
    Task.Run( () => Console.WriteLine("Task code finishing 3"))
};

Task.WaitAll(tasks);

Console.WriteLine("All done!");
```

This produces output something like the following, although the exact order of the finishing of the tasks is unpredictable:

```
Task code finishing 1
Task code finishing 3
Task code finishing 2
All done!
```

If we were to replace `Task.WaitAll` with `Task.WaitAny`, then all we could say is that at least one of the tasks would print out their “finishing” before the “All done!” message.

Returning a Value from a Task

Returning a Value from a Task

- Use the `Task<TResult>` class
- Specify the return type in the type argument

```
Task<string> task1 = Task.Run<string>(() =>  
    DateTime.Now.DayOfWeek.ToString());
```

- Get the result from the `Result` property

```
Console.WriteLine("Today is {0}", task1.Result);
```

A task that executes on a separate thread isn’t going to be of much use if we don’t have a way of getting data back to the main thread. To enable us to do this, there’s a generic class `Task<TResult>` that returns a value of type `TResult`. You create an instance of `Task<TResult>` in the same way as before, but the delegate returns a value of type `TResult`. Here’s a code sample that returns a string:

```
var task1 = Task.Run<string>(() => "Result string from task");  
Console.WriteLine(task1.Result);
```

This time we don’t need a `task1.Wait()`, because the fact that we are using the return value forces the main thread to wait for `task1` to complete.

Question: True or false: a task starts executing when it is instantiated by using `new Task()`?
Answer: False, you need to call the `Task.Start()` method.

Cancelling Long-Running Tasks

Cancelling Long-Running Tasks

- Pass a cancellation token as an argument to the delegate method
- Request cancellation from the joining thread
- In the delegate method, check whether the cancellation token is cancelled
- Return or throw an `OperationCanceledException`

Because they run independently of the UI thread, tasks can be used for long-running tasks without stopping the UI thread and freezing the application's user interface. A well-designed application will also give the user the option of cancelling a time-consuming process. They might not have realized how long it was going to take, and change their mind. It's always a good idea to allow users to undo their changes and reverse their decisions if it's possible. But we can't just stop the task and leave the application in some unknown state. So the Task Parallel Library supports the use of cancellation tokens to allow threads to be stopped in a managed way.

`CancellationToken` objects are created from a `CancellationTokenSource` and then passed to your task delegate when you create the task. In your task logic, you periodically check the cancellation token to see if it has been cancelled. If it has, then you stop processing, for example by breaking out of a processing loop, doing any cleanup that's required, and then returning from the task. Once the task has started, if you subsequently decide to abort the thread execution, in the main thread you call the `Cancel` method on the cancellation token source. It's up to your task logic to then "do the right thing" – the `Cancel` method doesn't terminate the process, it just asks politely.

Here's how it works in practice:

```
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken ct = cts.Token;
var task1 = Task.Run<double>(() => doCalculation(ct));
cts.Cancel(); // for demo: cancel after it started
Console.WriteLine(task1.Result);
double doCalculation(CancellationToken ct)
{
    double result = 0.0;
    for (int i = 0; i < 1000; i++)
    {
```

```
// some time-consuming operation here...

result += Math.Sqrt(i);

Console.Write(".");
if (ct.IsCancellationRequested) return 0.0;
}

return result;
}
```

To find out more about cancellation tokens, see: <https://aka.ms/3LzmS4d>.

Running Tasks in Parallel

Running Tasks in Parallel

- Use Parallel.Invoke to run multiple tasks simultaneously

```
Parallel.Invoke( () => MethodForFirstTask(),
                 () => MethodForSecondTask(),
                 () => MethodForThirdTask() );
```

- Use Parallel.For to run for loop iterations in parallel
- Use Parallel.ForEach to run foreach loop iterations in parallel
- Use PLINQ to run LINQ expressions in parallel

One of the applications of multi-threading is in parallel processing. This is where a compute-intensive task is broken up into sections that can be run concurrently, in separate threads. The advantage of doing this is that if you have a multi-core processor, you can gain a performance advantage by having each thread running on a separate processor. The performance benefit is theoretically in proportion to the number of processors, which may be 16 cores on some CPUs, and as many as thousands on specialized hardware such as GPUs. Unfortunately the actual performance gain will be less, because you rarely get close to 100% utilization of the processors. Depending on the algorithm, the performance benefit might be considerably lower, or even zero. Some problems simply don't lend themselves to parallelization, especially where there are interdependencies, so that each section of the calculation is dependent on the previous one.

If you do have a computational problem that can be divided up into separate, independent tasks, then you can use the Task Parallel Library to ensure that your calculation is multi-

threaded. This will give the hardware the opportunity to distribute the computation across the available processors.

In an earlier section we created three threads and awaited their completion. We can use the **Parallel.Invoke** method in a similar way, without the need to explicitly create the Task objects:

```
Parallel.Invoke(() => doCalculation("1"),
() => doCalculation("2"),
() => doCalculation("3"));
void doCalculation(string s)
{
    for (int i = 0; i < 1000; i++)
    {
        // some time-consuming operation here...
        Console.WriteLine(s);
    }
}
```

This code will output 1000 each of 1, 2 and 3, which doesn't do anything very useful but demonstrates that they all ran. You'll also notice that the numbers are jumbled up, but probably grouped, as each thread gets access to resources. In fact it depends on the hardware and the low-level operating system details, so your mileage may vary.

Our **doCalculation** function included a **for** loop. In fact we can use the Task Parallel Library to provide a loop that is automatically parallelizable. This works for both **for** loops and **foreach** loops. You have to write your loop using the **Parallel.For** method, rather than the built-in language syntax. The following code demonstrates the normal sequential loop, and the parallel version:

```
int start = 0;
int length = 500000;
double[] array = new double[length];
// sequential implementation...
for (int i = start; i < length; i++)
```

```
{  
array[i] = Math.Sqrt(i);  
}  
  
// parallel implementation...  
Parallel.For(start, length, i =>  
{  
array[i] = Math.Sqrt(i);  
});
```

There's also a **Parallel.ForEach** method to iterate over an **IEnumerable**. The example above works because each array value is independent. The values could be calculated in any order. This is not true for all operations. For example, we couldn't compute an average or a sum in this way, or any computation where the computed values were interdependent. Consider the following modified version:

```
int start = 0;  
int length = 500000;  
double[] array1 = new double[length];  
double[] array2 = new double[length];  
double sumSeq = 0, sumPar = 0;  
  
// sequential implementation...  
for (int i = start; i < length; i++)  
{  
array1[i] = Math.Sqrt(i);  
if (i > start) sumSeq += array1[i - 1]*array1[i];  
}  
  
// parallel implementation...  
Parallel.For(start, length, i =>  
{  
array2[i] = Math.Sqrt(i);  
});
```

```
if (i > start) sumPar += array2[i - 1]*array2[i];  
});  
Console.WriteLine("Sequential sum = " + sumSeq);  
Console.WriteLine("Parallel sum = " + sumPar);
```

If you run this code, you'll find you get two different results. The first one is correct, and the second one is a different value each time you run the program. For numerical computing, that's a nightmare! The reason is that the calculation depends on the values of two adjacent elements, and there's no guarantee that `array2[2344]` will have been calculated when we come to calculate `array2[2345]`. You have to be very careful when you try to make loops parallel.

For more information about parallel processing with the Task Parallel Library, see:
<https://aka.ms/3sNV7ym>.

Linking Tasks

Linking Tasks

- Use task continuations to chain tasks together:
 - `Task.ContinueWith` method links continuation task to antecedent task
 - Continuation task starts when antecedent task completes
 - Antecedent task can pass result to continuation task
- Use nested tasks if you want to start *an independent task* from a task delegate
- Use child tasks if you want to start *a dependent task* from a task delegate

As we saw in the previous section, sometimes it helps to do things in a certain order. One way of resolving the concurrency problem would be to break the calculation into separate tasks. You might notice that the initial step, taking the square root, was probably the most computationally expensive. We could have one parallel loop to do all the square roots, and put them in the array, and then a second sequential loop, to do the multiplications. That would still give us a performance benefit while giving the right answer. Similarly, we might want to execute a Task, and then have another task run afterwards, but only if the first task finishes successfully.

A task that only runs when the one before it has completed is called a “continuation”. This allows you to set up a chain of tasks that run in the background. The following code segment illustrates a task continuation.

```
var firstTask = new Task<string>(() => "Hello");
var secondTask = firstTask.ContinueWith((antecedent) =>
String.Format("{0}, world!", antecedent.Result));
firstTask.Start();
Console.WriteLine(secondTask.Result);
```

The first task, `firstTask`, is a regular task that returns a string, and `secondTask` is a continuation task that takes the result of the first task as an argument. We then call `firstTask.Start()` to kick off the process, in which `firstTask` runs and then starts `secondTask`, which takes the result of the first task and combines it into a string value to return. Finally, when `secondTask` completes, we print out the result of the `secondTask`. This will be the string “Hello, World!”.

This may seem like an unnecessarily complicated way to output the familiar string, but in a practical application of this you might be processing some data in multiple steps, and this is one way of orchestrating those steps.

You might have a task that starts multiple continuation tasks, so that the task “fans out” into many threads of execution. The “parent” task can then wait for the “child” tasks to finish before it completes, or it can just complete and leave the child tasks running.

Creating child tasks is, in principle, just a case of nesting the tasks:

```
var outer = Task.Run(() =>
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Run(() =>
    {
        Console.WriteLine("Nested task starting...");
        Thread.Sleep(500000);
        Console.WriteLine("Nested task completing...");
    });
    outer.Wait();
    Console.WriteLine("Outer task completed.");
});
```

```
Thread.SpinWait(500000);
```

This is fairly simple, and if you run it you'll see the outer task starts and finishes and then the nested task also starts and finishes. We've made the main thread sleep for half a second before the program terminates, otherwise the inner "child" thread never gets a chance to run.

There are various options to attach the child thread to the parent to make the parent wait until the child completes. Nested tasks let you break up asynchronous tasks into smaller sub-tasks that can be spread across all available threads. Parent and child tasks, on the other hand, are better for controlling synchronization, by making sure that child tasks are done before the parent task finishes.

For more about continuations, see: <https://aka.ms/3MALdYC>.

Handling Task Exceptions

Handling Task Exceptions

- Call Task.Wait to catch propagated exceptions
- Catch AggregateException in the catch block
- Iterate the InnerExceptions property and handle individual exceptions

```
try
{
    task1.Wait();
}
catch(AggregateException ae)
{
    foreach(var inner in ae.InnerExceptions)
    {
        // Deal with each exception in turn.
    }
}
```

What happens when an exception is thrown inside a **Task**? The exception is sent back to the thread that started the task (referred to as the "joining thread"). There may well be multiple exceptions to deal with, because of continuations or child tasks, so the Task Parallel Library packages up all the exceptions into an **AggregateException** object so they all get sent back to the joining thread. This has an **InnerExceptions** collection that contains all the original exceptions.

On the joining thread, you have to use the `Task.Wait` method to wait for the task to finish before you can catch the exceptions. You do this by calling `Task.Wait` inside a try block and adding a catch block for `AggregateException`. It's also a good idea to catch the `TaskCanceledException` which will be thrown when you cancel a task.

```
var task1 = Task.Run(() => doCalculation());  
try  
{  
    task1.Wait();  
}  
catch (AggregateException ae)  
{  
    foreach (var e in ae.InnerExceptions)  
    {  
        Console.WriteLine("Exception thrown: " + e.Message);  
    }  
}
```

For more on exception handling in the Task Parallel Library, see: <https://aka.ms/3wCxt8l>.

Question: True or false: when an exception is thrown inside a task, the exception is sent back to the thread that started the task?

Answer: True, the exception is sent back to the joining thread.

Lesson 2 – Asynchronous Calls

Normally in code, when we call a method, the next line of code doesn't get executed until the method returns. When we talk about asynchronous calls, we mean that we make a method call and it returns immediately. The code inside the method may get executed much later, so the code doesn't get executed synchronously, or in sequence. To some extent we have surrendered control over the sequence in which statement executions occur.

An asynchronous operation is one that runs on a different thread. The thread that starts an asynchronous operation doesn't have to wait for it to finish before moving on. Tasks and asynchronous operations are very similar, they are both ways of getting code to run on a different thread. The most common reason is that you want to run some code but you can't guarantee that it will return quickly, for example a network request, or an operation on a slow peripheral device. In .NET and C# there are some tools to make writing asynchronous operations easier, so that you don't have to think about threads, and semaphores, and callback methods. This lets you focus on the business logic of your application.

In this lesson, you'll learn how to use and manage asynchronous operations.

Lesson Objectives

After completing this lesson, you'll be able to:

- Use the `async` and `await` keywords.
- Invoke callback methods.
- Create asynchronous methods.

Using `async` and `await`

Using `async` and `await`

- Add the `async` modifier to method declarations
- Use the `await` operator within `async` methods to wait for a task to complete without blocking the thread

```
private async void btnLongOperation_Click(object sender, RoutedEventArgs e)
{
    ...
    Task<string> task1 = Task.Run<string>(() =>
    {
        ...
    });
    lblResult.Content = await task1;
}
```

The `async` and `await` keywords were added to make it easier to write asynchronous code. The `async` keyword is used to show that a method is asynchronous and will return immediately. You can also use `Async` as part of the method name if you want to make it clear to users of the method that it's asynchronous. When you use an `async` method, you use the `await` keyword to show where the method's execution can be paused while you wait for a

long-running operation to finish. This allows the thread that called the method to continue executing while the method is waiting.

Consider the following code that is similar to some we used earlier in the course:

```
public void showwebsite(string uri)
{
    var client = new HttpClient();
    var msg = await client.GetStringAsync(uri);
    Console.WriteLine(msg);
}
```

Because the **HttpClient.GetStringAsync** method is making a network call, we really have no idea how long it will take to return. It may take so long that a network timeout occurs, which could be of the order of a minute. That's why the **GetStringAsync** method is asynchronous, as indicated by the name, but also by the fact that it returns a **Task<string>**. By using the **await** keyword, the **showWebSite** method will return immediately (strictly we should say that it "yields" to the caller), and execution of the main thread will continue. In the meantime, the **GetStringAsync** method will execute on another thread. When the network request returns, execution continues from the "await" point, executing the **Console.WriteLine**. Code execution continues until it reaches a point where **showWebSite** would have returned.

By using the **async/await** pattern, you get asynchronous code without having to worry about managing threads. It also solves a perennial problem with UI code, which is the need for all UI logic to be on the UI thread. If you've done UI development in the past, you've probably had to do all kinds of message passing between threads so that you can update the UI. The **async/await** pattern lets you effectively run logic asynchronously on a single thread, allowing you to run asynchronous logic without blocking the UI:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    label1.Content = "Working on it...";
    var client = new HttpClient();
    label1.Content = await
        client.GetStringAsync("http://foo.com/coffee");
}
```

In this code sample, we have a button click event that updates the text of a Label control, then makes an asynchronous call to a web endpoint, and updates the label with the returned

value. That asynchronous step might take a while, but the `await` keyword ensures that the event handler returns immediately, and the UI continues to be responsive. When the `GetStringAsync` method eventually returns, the label will get updated asynchronously.

To learn more about `async/await`, see: <https://aka.ms/3yMHaob>.

Creating Awaitable Methods

Creating Awaitable Methods

- The `await` operator is always used to wait for a task to complete
- If your synchronous method returns `void`, the asynchronous equivalent should return `Task`
- If your synchronous method has a return type of `T`, the asynchronous equivalent should return `Task<T>`

In the previous example we used an asynchronous method of the `HttpClient` framework class. The `await` operator is used with a method that returns a `Task` instance. If you want to make your own asynchronous method that you can use with the `await` keyword, you must return a `Task` object. If you need to return a value then you can return a `Task<TResult>` object.

Suppose we want to use a version of our `showWebSite` function to update the `label1` content in our button click handler. We want to do this:

```
label1.Content = await showwebsite("http://spdoctor.com/");
```

Currently, this won't work, because `showWebSite` isn't asynchronous:

```
public string showwebsite(string uri)
{
    var client = new HttpClient();
    var msg = await client.GetStringAsync(uri);
    return msg;
}
```

Worse still, the method won't compile because we can't use `await` inside a function that isn't `async`. At this point you might wonder if `async` is going to spread throughout the application. How can we make `showWebSite` `async`? First, we need to add the `async` keyword, and then change the return type to `Task<string>`:

```
public async Task<string> showwebsite(string Uri)
{
    var client = new HttpClient();
    var msg = await client.GetStringAsync(Uri);
    return msg;
}
```

Now everything will compile and have the desired behaviour.

For more detail about asynchronous return types, see: <https://aka.gd/3NIT2Bs>.

Creating and Invoking Callback Methods

Creating and Invoking Callback Methods

- Use the `Action<T>` delegate to represent your callback method
- Add the delegate to your asynchronous method parameters

```
public async Task GetCoffees(Action<IEnumerable<string>> callback)
```

- Invoke the delegate asynchronously within your method

```
    await Task.Run(() => callback(coffees));
```

There are times when you need an asynchronous method to call a callback method. The asynchronous method sends data to the callback method, which then does something with the data. You could also use the callback method to add the processed results to the user interface.

To set up an asynchronous method to call a callback method, you have to give the method a delegate for the callback method as a parameter. Most of the time, a callback method will take one or more arguments, but with no return a value. This makes the `Action<T>` delegate a good way to represent a callback method, where `T` is the type of your argument. There are overloads of the `Action` delegate that can take from 1 to 16 type parameters as needed.

Suppose we implemented our showWebSite method but wanted to make it configurable as to what it did with the response. We could add a callback method to the definition:

```
public async Task showwebsite(string Uri, Action<string>
callback)

{
    var client = new HttpClient();
    var msg = await client.GetStringAsync(Uri);
    await Task.Run(() => callback(msg));
}
```

For more on the Action<T> delegate, see: <https://aka.ms/3wltuaY>.

Question: True or false: the `await` keyword will block all execution until the awaited method returns?

Answer: False, the outer method will return ("yield") and execution of the rest of that method will continue after the inner method returns.

Lesson 3 – Dealing with Conflicts

Adding multithreading to your applications can improve their speed and responsiveness in many ways. But it also brings about a set of new problems. When multiple threads can concurrently update the same resource, the resource can be left in an inconsistent or unknown state.

In this lesson, we'll learn about some of the synchronization techniques we can use to make sure that resources are accessed in a thread-safe way.

Lesson Objectives

After completing this lesson, you'll be able to:

- Use the lock keyword to prevent concurrent updates.
- Restrict access to resource by use of synchronization.
- Use concurrent collections.

Using Locks

Using Locks

- Create a private object to apply the lock to
- Use the `lock` statement and specify the locking object
- Enclose your critical section of code in the lock block

```
private object lockingObject = new object();
lock (lockingObject)
{
    // Only one thread can enter this block at any one time.
}
```

For all the advantages of multithreading, it introduces a new class of bug. If two threads try to modify some resource at the same time, you can end up with a conflict. Suppose you have a collection of items, and some code to modify the items. With a single-threaded application it's relatively easy to update that collection when, say, a user edits a form, or an automated process performs some kind of clean-up of the data.

But if you have a multi-threaded app, there's a risk that two threads might try to change an item at the same time. Suppose two users are both updating the item. The logic probably involves taking a copy of the data, modifying it, and writing it back, perhaps to a database. If the first user makes some changes, and then the second user also makes changes, then the thread handling the second user's actions will overwrite the changes made by the first user, and they'll be lost. Worse, if one user deletes an item, and a second user tries to make a change, then depending how the items are referenced, all manner of strange results might arise with null references and inconsistent values.

The worst part about these kinds of errors is that they are very dependent on precise timing, and therefore difficult to reproduce. They might happen only very occasionally, making them very difficult to track down. For this reason we have to be very careful when writing multi-threaded code.

One way of fixing or preventing this kind of problem is by putting a mutual-exclusion lock on parts of the code that are likely to perform operations concurrently. You can do this with the `lock` keyword, which can protect a critical segment of the code, ensuring that all other threads are mutually excluded. You put a lock on a code block as follows:

```
public class BeverageList
{
    private readonly object mylock = new object();
```

```

public List<string> beverages = new List<string>();
public void UpdateNames(int iBev)
{
    lock (mylock)
    {
        beverages[iBev] += " (improved)";
    }
}
}

```

The **lock** statement guarantees that only one thread can enter the code inside the block, and therefore that only one thread can update the `beverages` collection at a time.

The `mylock` object is used to apply the lock. This object should be private, and not be used for locking other code, or otherwise used as part of the application logic.

Note: Behind the scenes, the lock syntax is actually compiled to a mechanism using the **Monitor API**.

Synchronization Primitives with the Task Parallel Library

Synchronization Primitives with the Task Parallel Library

- Use the `ManualResetEventSlim` class to limit resource access to one thread at a time
- Use the `SemaphoreSlim` class to limit resource access to a fixed number of threads
- Use the `CountdownEvent` class to block a thread until a fixed number of tasks signal completion
- Use the `ReaderWriterLockSlim` class to allow multiple threads to read a resource or a single thread to write to a resource at any one time
- Use the `Barrier` class to block multiple threads until they all satisfy a condition

There are a number of synchronization primitives; classes that allow you to control threads at a lower level. These are sometimes needed for particular situations. This allows you to manage how threads work together. The Task Parallel Library has many such primitives to let you control access to resources. The most frequently used of these are:

Primitive	Purpose
-----------	---------

ManualResetEventSlim	Limit resource access to one thread at a time. Can be in a signaled or unsignaled state.
SemaphoreSlim	Limit resource access to a fixed number of threads, by means of an integer counter that tracks the number of threads currently accessing the protected resource(s). When a thread wants to access the resource, the count is decremented, and when it has finished with the semaphore, it calls the Release method and the counter is incremented. If the counter is zero, threads can't use the resource.
CountdownEvent	Block a thread until a fixed number of tasks signal completion. It works in a similar way to SemaphoreSlim, but with threads blocked until the counter reaches zero.
ReaderWriterLockSlim	Allow multiple threads to read a resource or a single thread to write to a resource at any one time.
Barrier	Block multiple threads until they all satisfy a condition. When a thread reaches a certain point, it calls the SignalAndWait method of the Barrier object. It can be used where a set of interdependent calculations all need to reach the same point before they can continue.

Using Concurrent Collections

Using Concurrent Collections

The System.Collections.Concurrent namespace includes generic classes and interfaces for thread-safe collections:

- ConcurrentBag<T>
- ConcurrentDictionary<TKey, TValue>
- ConcurrentQueue<T>
- ConcurrentStack<T>
- IProducerConsumerCollection<T>
- BlockingCollection<T>

None of the collection classes in the **System.Collections.Generic** namespace, or the older ones in the **System.Collections** namespace, are thread-safe, in other words they're not safe for multiple threads to use at the same time. When you use these collections from multithreading methods like tasks, you need to make sure to preserve the integrity of the collections. You could do this by using the synchronisation primitives from the previous section. But an easier way is to make use of another set of collections in

the `System.Collections.Concurrent` namespace. These collections are designed to be safe for access by multiple threads. The table below shows some of these thread-safe collection classes and their equivalents:

System.Collections.Generic	System.Collections.Concurrent
<code>List<T></code>	<code>ConcurrentBag<T></code>
<code>Dictionary<TKey, TValue></code>	<code>ConcurrentDictionary<TKey, TValue></code>
<code>Queue<T></code>	<code>ConcurrentQueue<T></code>
<code>Stack<T></code>	<code>ConcurrentStack<T></code>

To learn more about concurrent collections, see: <https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent>.

Lab: Improving the Responsiveness and Performance of the Application

Lab: Improving the Responsiveness and Performance of the Application

Lab scenario

You have been asked to update the Grades application to ensure that the UI remains responsive while the user is waiting for operations to complete. To achieve this improvement in responsiveness, you decide to convert the logic that retrieves the list of students for a teacher to use asynchronous methods. You also decide to provide visual feedback to the user to indicate when an operation is taking place.

Objectives

Exercise 1:
Ensuring That the UI Remains Responsive When Retrieving Teacher Data

Exercise 2:
Providing Visual Feedback During Long-Running Operations

Module Review and Takeaways

Review Questions

Question: You have three tasks named task1, task2, and task3. You create and start the tasks, and want to block the joining thread until all three are complete. Which code sample should you use?

- A: Task.WhenAny(task1, task2, task3);
- B: Task.WaitAny(task1, task2, task3);
- C: Task.WaitAll(task1, task2, task3);
- D: Task.WhenAll(task1, task2, task3);
- E: task1.Wait(); task2.Wait(); task3.Wait();

Answer: C

Module 11 – C# Interop

Module Overview

Real production software systems, especially ones that have developed over time, are often composed of many components that use different programming languages and technologies. Some applications might be built in .NET 6, others in older versions of .NET such as the .NET Framework, while others might be unmanaged C++ applications, or Visual Basic, or Java. It's also usual to find that at least some parts of a modern application are implemented in JavaScript in the browser.

In this module, you'll learn how to make your applications interoperate with unmanaged code, and how to make sure that your code releases any underlying unmanaged resources.

Objectives

After completing this module, you'll be able to:

- Understand the Dynamic Language Runtime.
- Use unmanaged resources.

Lesson 1 – Dynamic Objects

There are a lot of different programming languages, and each one has its own pros and cons and situations where it works best. You may find there is existing, tested code in your organization that you want to reuse, but not necessarily written in C#, or even for the .NET platform. When you can use components written in other languages, you may find different type systems have been used, and that can make interop a challenge. That's why the Dynamic Language Runtime was written, so that .NET could support dynamically typed languages.

We'll learn about the DLR in this lesson, and see how to use it to work with unmanaged code.

Lesson Objectives

After completing this lesson, you'll be able to:

- Describe dynamic objects and the DLR.
- Create a dynamic object and call its methods.

What Are Dynamic Objects?

What Are Dynamic Objects?

- Objects that do not conform to the strongly typed object model
- Objects that enable you to take advantage of dynamic languages, such as Python
- Objects that simplify the process of interoperating with unmanaged code

In this course we've talked a lot about C# as a strongly-typed language, and the benefits that brings. By using strong typing, we get a lot of support from the development environment in the form of Intellisense, and compile-time checking. Many classes of error are caught by the compiler, rather than at runtime, making them much easier to fix.

Dynamic objects, on the other hand, don't have their types checked until they are run. This lets you write code quickly because you don't have to define each member or worry about its type until you need to call it in your code. For writing code quickly, and scripting, dynamic typing is a good choice. For building large complex applications, the balance tips in favour of strong typing. Some examples of dynamic programming languages are JavaScript, Ruby, Python, Lisp, Smalltalk, Lua, Cobra, and Groovy.

The Dynamic Language Runtime (DLR) is a runtime environment that adds dynamic languages to the Common Language Runtime (CLR) that is the basis of languages like C#. The DLR makes it easier to build dynamic languages that run on .NET and to add dynamic features to statically typed languages.

Unmanaged Code

C# is a managed programming language, which means that the Common Language Runtime is responsible for running the IL code that is the output of the C# compiler. Amongst other things, the CLR performs memory management and handles exceptions. Unmanaged code, like C or C++, runs outside of the CLR, and for these languages it's up to the programmer to allocate and deallocate memory and other resources.

When you're building solutions with C# and .NET, it's not uncommon to find yourself in a situation where you want to reuse some unmanaged code. You might have an old C++ system that was built ten years ago, or you might just want to use a function in the user32.dll assembly that Windows gives you. Interoperability means the ability to use functionality built with technologies other than the .NET Framework, such as COM, C, C++ or the Win32 API. Dynamic objects make it considerably easier to support this interoperability.

To find out more about the Dynamic Object class, see: <https://aka.ms/38dTgfb>.

What Is the Dynamic Language Runtime?

What Is the Dynamic Language Runtime?

The DLR provides:

- Support for dynamic languages, such as IronPython
- Run-time type checking for dynamic objects
- Language binders to handle the intricate details of interoperating with another language

The DLR has services and components to support dynamic languages and gives a way for unmanaged components to be used with .NET. The DLR is in charge of managing how the running CLR assembly and the dynamic object, such as an object written in IronPython or a COM assembly, talk to each other. The DLR adds dynamic objects to C#, to support dynamic behavior and enable interop with dynamic languages. Type checking is deferred until run-time for dynamic objects, and the interop details like marshalling of parameters is simplified.

The DLR doesn't have features that are specific to a language, but it does have a set of language binders. A language binder tells both the unmanaged language, like IronPython, and .NET, how to call methods and marshal parameters. The language binders also do run-time type checks, such as making sure that an object has a method with a certain signature.

To learn more about the Dynamic Language Runtime, see: <https://aka.ms/3Nvj3ym>.

Creating a Dynamic Object

Creating a Dynamic Object

- Dynamic objects are declared by using the `dynamic` keyword

```
dynamic d = 1; // type: dynamic {int}
var testSum = d + 3; // type: dynamic {int}
var i = 2; // type: int
```

- Dynamic objects are variables of type object
- Dynamic objects do not support:
 - Type checking at compile time
 - Visual Studio IntelliSense

With the `dynamic` keyword, which is part of the DLR infrastructure, you can define dynamic objects in your applications. The resulting variable has the type “dynamic”, which means a dynamic object. You can set a `dynamic` variable to any type of value and try to call any method, and it will not be checked by the compiler. At run time, the DLR will use the language binders to type-check your dynamic code, and make sure that any method you are trying to call exists.

One thing you don’t get with dynamic types is any kind of IntelliSense in Visual Studio. This pop-up assistance is one of the benefits of using strong-typing of variables. Note that declaring a variable using the keyword `var` is not the same as `dynamic`. A `var` is still strongly typed. It’s just that the compiler infers the type from the context. Consider the following code:

```
dynamic d = 1; // type: dynamic {int}
var testSum = d + 3; // type: dynamic {int}
d = 2.0; // type: dynamic {double}
var i = 2; // type: int
```

On the first line we create a dynamic variable `d`, and assign it to a value of `1`, which is an integer literal. On the second line we create a var `testSum`, which is assigned to `d + 3`. That expression is an addition of a dynamic variable and an integer, and the result is still dynamic, so `testSum` has the type of `dynamic` also. On the third line we assign our dynamic

variable **d** a value of **2.0**, which is a double precision literal value. As a result, **d** now contains a double value, but its type is still dynamic. On the fourth line, we just have a **var**, and we assign it an integer literal, and the compiler infers the type as **int** from the assignment. As a result, is a normal strongly-typed integer variable.

When it comes to interoperating with unmanaged assemblies, the dynamic type becomes very useful. Suppose we wanted to use a COM assembly like `Microsoft.Office.Interop.Word`. We could use the NuGet package manager inside Visual Studio to find the `Microsoft.Office.Interop.Word` assembly for Office 2013 Word interop. We could then easily create an instance of the Word application class in code using a dynamic variable declaration:

```
dynamic word = new Microsoft.Office.Interop.Word.Application();
```

Once you've created a Word application object, you just use its methods and properties using the normal notation in C#, and pass arguments. The DLR removes the need for special marshaling syntax.

```
// Create a new document...
dynamic doc = word.Documents.Add();
doc.Activate();
doc.SaveAs("mydocument.docx");
```

To learn more about the dynamic keyword, see: <https://aka.gd/3NviFQr>.

Question: True or false: variables declared using the dynamic keyword can be assigned different types during the life of the variable?

Answer: True, unlike a variable declared using **var**, dynamic variables are not strongly-typed.

Lesson 2 – Managing Resources

It's important to release unmanaged resources when you've finished working with them. When managed resources are no longer used, they are marked for deletion, but precisely when that will happen is non-deterministic.

In this lesson, you'll learn about a typical .NET object's life cycle, and how to make sure that when an object is destroyed, the resources it has been using are released.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand the lifecycle of a .NET object.
- Use the dispose pattern.

The Object Lifecycle

The Object Lifecycle

- When an object is created:
 1. Memory is allocated
 2. Memory is initialized to the new object
- When an object is destroyed:
 1. Resources are released
 2. Memory is reclaimed

A .NET object's life cycle starts with object creation with the **new** operator (or some kind of factory method), and ends with its destruction by the garbage collector. When you use **new** to create an object, the CLR allocates a sufficiently large block of memory and initializes it. The CLR also keeps a reference count, and when the object eventually goes out of scope, and there are no remaining references to it in your code, it is marked for deletion. Eventually the garbage collector process will release the memory for reuse. The garbage collector has an algorithm that determines when it runs and how it prioritizes the release of memory, unless you specifically instruct it to run. This makes it difficult to predict when an object will get deleted.

In languages that don't have managed memory, like C for example, you have to do all this yourself, using instructions like **malloc** to allocate memory, and **free** to make it available again. If you lose track of the memory, you get a "memory leak" and eventually the application crashes with an out-of-memory error. Managed code removes this problem by taking care of memory management automatically.

To read in immense detail how .NET garbage collection works, see: <https://aka.gd/3Gc6TYV>

While this works well, it's important to remember that the CLR manages memory automatically. But it doesn't take care of any unmanaged objects you might use. In that case you need to make sure that you add code to release those resources when you've finished using them, otherwise you'll get a "resource leak". This can have negative consequences for your application in the same way as a memory leak.

In the next section we'll look at how to manage resources with the Dispose pattern.

Implementing the Dispose Pattern

Implementing the Dispose Pattern

- Implement the IDisposable interface

```
public class ManagedWord : IDisposable
{
    bool _disposed;

    ~ManagedWord
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool isDisposing) { ... }

}
```

In the previous topic we saw how the CLR manages memory automatically. But applications use other resources that are unmanaged, such as file handles, network connections, COM objects, and so forth. We can define a destructor, although they're rarely used in C# because of managed memory:

```
class Foo

{

public Foo()

{

// parameterless constructor

}

~Foo()

{

// destructor

}
```

```
}
```

We could release resources in the destructor, but there's a problem. We don't know when the garbage collector will delete our object. We can politely ask for the garbage collector to run:

```
GC.Collect(0, GCCollectionMode.Forced, true);
```

But that's no guarantee of when, or if, our destructor will get called. We could end up with .NET objects with a very small memory footprint, that are responsible for creating enormous unmanaged resources. That can result in very serious resource leaks, so we need a better solution.

The **Dispose** pattern is designed to solve this exact problem. It frees up resources that an object has used. The **IDisposable** interface is in the **System** namespace and facilitates this pattern. The **IDisposable** interface defines a single method called **Dispose()** that you should implement to release all of the resources that your object used. If your object is derived from a base class, the **Dispose** method should also call the **Dispose** method on the base type. Calling the **Dispose** method is not the same as calling a destructor, and the object is not destroyed. It will stay in memory like any other .NET object, until the last reference to it is gone, and the garbage collector cleans it up. The **IDisposable** interface is used by many .NET framework classes that wrap unmanaged resources, like the **HttpClient** class.

When you make your own classes that use unmanaged types and resources, you should also implement the **IDisposable** interface. To do this you need to add **IDisposable** to your class and then implement the **Dispose()** method. You should then add a private field to track the disposal state of your object, so that you don't execute the **Dispose** code more than once. To make your methods more robust, you can also add checks to see if the object was already disposed, and if it has, throw an **ObjectDisposedException**.

The following shows a very simple implementation of a disposable class:

```
class Foo : IDisposable
{
    bool isDisposed;
    public Foo()
    {
        // allocate some unmanaged resources
        isDisposed = false;
    }
}
```

```
public void DoSomething()
{
    if (disposed)
        throw new ObjectDisposedException("Foo fail in DoSomething!"); ;
}

public void Dispose()
{
    if (disposed) return;
    // release any unmanaged resources
}
}
```

You can add more logic and overloaded Dispose methods to deal with the case where the garbage collector disposes of your object. Refer to the documentation if you need to implement these features.

To learn more about the `IDisposable` interface, see: <https://aka.ms/3wISPIf>.

Managing the Lifetime of an Object

Managing the Lifetime of an Object

- Explicitly invoke the `Dispose` method

```
HttpClient client = new HttpClient();
var response = await client.GetAsync("https://spdoctor.com/");
Console.WriteLine(response.StatusCode.ToString());
client.Dispose();
```

- Implicitly invoke the `Dispose` method

```
using (HttpClient client = new HttpClient())
{
    var response = await client.GetAsync("https://spdoctor.com/");
    Console.WriteLine(response.StatusCode.ToString());
}
```

To manage resources that implement the **IDisposable** interface you need to call the **Dispose** method in your code. In order to avoid resource leaks, you should always dispose of these objects when you've finished with them. You can do this by calling **Dispose** directly:

```
HttpClient client = new HttpClient();

var response = await client.GetAsync("https://spdoctor.com/");

client.Dispose();
```

What if your code throws an exception before you get a chance to explicitly call **Dispose**? To prevent that occurring, you should put the **Dispose** call in a **finally** block in your exception handling clauses. Code in the **finally** block is guaranteed to execute:

```
HttpClient client = new HttpClient();

try
{
    var response = await client.GetAsync("https://spdoctor.com/");

    Console.WriteLine(response.StatusCode.ToString());
}

catch (Exception e)
{
    Console.WriteLine(e.Message);
}

finally
{
    client.Dispose();
}
```

Hint: as an extra precaution, check that an object isn't null before calling **Dispose** on it.

You should also be careful not to accidentally do a “premature disposal”, where you dispose an object you haven't finished with. That is likely to result in exceptions or more obscure bugs when you try to use the disposed object. For this reason, a safer way of using the dispose pattern is through the **using** syntax:

```
using (HttpClient client = new HttpClient())
{
    var response = await client.GetAsync("https://spdoctor.com/");
    Console.WriteLine(response.StatusCode.ToString());
}
```

With the above syntax the call to Dispose is done automatically by the compiler. It makes it quite a bit more difficult to get the dispose pattern wrong.

You can find out more about **using** statements at: <https://aka.gd/3z3ht2y>.

Lab: Upgrading the Grades Report

Lab: Upgrading the Grades Report

Lab scenario

You have been asked to upgrade the grades report functionality to generate reports in Word format. In Module 6, you wrote code that generates reports as an XML file; now you will update the code to generate the report as a Word document.

Objectives

Exercise 1:
Generating the Grades Report by Using Word

Exercise 2:
Controlling the Lifetime of Word Objects by Implementing the Dispose

Module Review and Takeaways

Review Questions

Question: True or false: a `using` statement can implicitly invoke the `Dispose` method on an object that implements `IDisposable`?

Answer: True

Question: What kind of type checking is carried out on `dynamic` variables?

- A: compile time checking
- B: run time checking
- C: no checking
- D: dynamic checking

Answer: B

Module 12 – Designing for Reuse

Module Overview

One of the worst things that happen in software development is when you discover that you, or your team, have written the same code twice. When writing software, it's important to consider how it can be reused, and how the features you're developing could be useful in other apps. Complex systems are built of numerous components, and some of those components can probably be shared with other applications developed by your company.

In this module, you'll learn how to use reflection to discover the structure of existing assemblies, and how to use attributes to add extra metadata to types and type members. You'll also learn how to use the CodeDOM to dynamically generate code at run time, and how to version assemblies.

Objectives

After completing this module, you'll be able to:

- Understand .NET reflection and how to use it.
- Create custom attributes.
- Get information about assemblies.

Lesson 1 – Metadata

You might have an existing assembly, perhaps one that you didn't write, or that you don't have the source code for. It would be useful to look inside that assembly, look at what it contains, what types, and properties, and methods, and run its code. This is the purpose of reflection, and the reflection API. There are many applications for reflection, e.g. if you build a unit testing utility, you might need to be able to get a reference to a type and run some of its public methods.

This lesson will look at reflection, and how it can be used to look at the types in an assembly that has already been compiled.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the purpose of reflection.
- Load an existing assembly.
- Use reflection to look at type metadata.
- Use reflection to invoke methods.

What Is Reflection?

What Is Reflection?

- Reflection enables you to inspect and manipulate assemblies at run time
- The System.Reflection namespace contains:
 - Assembly
 - TypeInfo
 - ParameterInfo
 - ConstructorInfo
 - FieldInfo
 - MemberInfo
 - PropertyInfo
 - MethodInfo

Reflection is a feature that lets you look at assemblies, types, and type members at run time and change them on the fly. It is a powerful form of runtime introspection that the .NET platform gives us.

Reflection is used throughout .NET. Many of the .NET framework classes use reflection, as do some of the tools that come with Microsoft Visual Studio®. As an example, many of the serialization classes, that we looked at earlier in the course, make considerable use of reflection to deduce which type members should be serialized, and in what format.

You can use reflection to examine the dependencies of an assembly, or to identify members of a type that have been decorated with some attribute. You might be building an application that supports a plugin architecture, whereby assemblies are loaded at runtime. If it were not for reflection, you would have to know exactly which assemblies you were going to use at build time, before you compiled your code. There are many applications where the precise method or class is not known until runtime. Reflection makes all these things possible.

Note: calling a method by means of reflection is much slower than executing static code, so it should only be used where it is needed.

The **System.Reflection** namespace contains many classes that you can use to support reflection. The table below lists some of the most important ones:

Class	What is exposed
Assembly	Metadata and types of an assembly.
TypeInfo	Characteristics of a type.
ParameterInfo	Any parameters that a member accepts.
ConstructorInfo	Any constructors available for a type.
FieldInfo	Characteristics of any fields defined for a type.
MemberInfo	Any members that a type exposes.
PropertyInfo	Any properties defined for a type.
MethodInfo	Any methods that are defined for a type.

Note: Additionally there is the **Type** class, which is part of the **System** namespace. This has a number of useful members that you can access when using reflection. For more information, see: <https://aka.gd/3PAtkLu>. For more detail on .NET Reflection, see: <https://aka.gd/3IELwWo>.

Loading Assemblies by Using Reflection

Loading Assemblies by Using Reflection

- The Assembly.LoadFrom method

```
using System.Reflection;

var path = Assembly.GetExecutingAssembly().Location;
var assembly = Assembly.LoadFrom(path);
foreach(var type in assembly.GetTypes())
    Console.WriteLine(type.FullName);
```

- The Assembly.Load method

```
var path = Assembly.GetExecutingAssembly().Location;
var rawBytes = File.ReadAllBytes(path);
var assembly = Assembly.Load(rawBytes);
```

You can use the **System.Reflection.Assembly** class to get information about an assembly. Use the static **Load** or **LoadFrom** methods to load an assembly by passing the path to the assembly file.

Here's some sample code for loading an assembly for reflection:

```
using System.Reflection;

var assembly = Assembly.LoadFrom("ConsoleApp5.dll");
foreach(var type in assembly.GetTypes())
    Console.WriteLine(type.FullName);
```

You'll get some output similar to the following, depending on the code:

```
Microsoft.CodeAnalysis.EmbeddedAttribute
System.Runtime.CompilerServices.NullableAttribute
System.Runtime.CompilerServices.NullableContextAttribute
Program
Foo
ConsoleApp5.BeverageList
Program+<<Main>$>d__0
Program+<>o__0
```

If you need to know the path to the currently executing assembly, you can get that from the static **Assembly** object:

```
var path = Assembly.GetExecutingAssembly().Location;
```

You could also just directly use the assembly returned by **GetExecutingAssembly()**:

```
var assembly = Assembly.GetExecutingAssembly();
foreach(var type in assembly.GetReferencedAssemblies())
    Console.WriteLine(type.FullName);
```

Again, depending on the code, you'll get a list of referenced assemblies with their full names:

```
System.Runtime, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Net.Http, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Net.Primitives, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Diagnostics.Process, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Linq.Expressions, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Collections, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Threading, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

System.Console, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a

Microsoft.CSharp, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
```

Note: you have a choice with the older .NET Framework about whether to load an assembly in a “reflection-only” context, or “execution” context, where the former will only let you look at the assembly's metadata and won't allow you to run code. Reflection-only context is deprecated in .NET 6.

Examining Types by Using Reflection

Examining Types by Using Reflection

- Get a type by name

```
var assembly = Assembly.GetExecutingAssembly();
var type = assembly.GetType("Foo");
```

- Get all of the constructors

```
var constructors = type.GetConstructors();
```

- Get all of the fields

```
var fields = type.GetFields();
```

- Get all of the properties

```
var properties = type.GetProperties();
```

- Get all of the methods

```
var methods = type.GetMethods();
```

With reflection, you can look at how any type in an assembly is defined. You can use the `Assembly` class's `GetType` method to find a type by name, or you can get a collection of types using the `GetTypes` method. Having got a `Type` object, you can then use various methods such as `GetConstructors`, `GetFields`, `GetProperties`, and `GetMethods`. The names of these methods are fairly self-explanatory.

Armed with this set of APIs, we can build a small utility to look at a class "Foo" in our current assembly, that we defined in an earlier module when we were looking at the Dispose pattern:

```
using System.Reflection;

var assembly = Assembly.GetExecutingAssembly();

var type = assembly.GetType("Foo");

Console.WriteLine("Class: " + type.FullName);

Console.WriteLine();

Console.WriteLine("====constructors====");
foreach(var constructor in type.GetConstructors())
    Console.WriteLine(constructor.ToString());

Console.WriteLine("====fields====");
```

```
foreach (var field in type.GetFields())
Console.WriteLine(field);

Console.WriteLine("====props====");
foreach (var prop in type.GetProperties())
Console.WriteLine(prop);

Console.WriteLine("====methods====");
foreach (var method in type.GetMethods())
Console.WriteLine(method);
```

Here's the output for the class Foo:

```
Class: Foo

====constructors====

Void .ctor()

====fields====

====props====

====methods====

Void DoSomething()

Void Dispose()

System.Type GetType()

System.String ToString()

Boolean Equals(System.Object)

Int32 GetHashCode()
```

Notice the default constructor, no fields or properties, but quite a few methods. We only defined the Dispose and DoSomething methods in this class. The rest have been inherited from the **Object** base class from which all classes are ultimately derived.

Question: True or false: you can completely reconstruct an application's source code by means of reflection?

Answer: False, not all of the details of the source code, such as white space and comments, is preserved in the compiler output.

Invoking Members by Using Reflection

Invoking Members by Using Reflection

- Instantiate a type

```
var type = assembly.GetType("Foo");
// get the default constructor...
var constructor = type.GetConstructor(new Type[] { });
var foo = constructor.Invoke(new object[0]);
```

- Invoke methods on the instance

```
var DoSomething = type.GetMethod("DoSomething");
DoSomething.Invoke(foo, null);
```

- Get or set property values on the instance

```
var property = type.GetProperty("Text");
var text = property.GetValue(foo) as string;
```

As well as simply looking at properties of types, the .NET Reflection API also enables you to instantiate those objects, get and set their properties, and invoke methods on them. In fact you can use them just as if they were referenced at build time.

In order to instantiate a type with reflection, you need to get a constructor and then **Invoke** that constructor method.

```
using System.Reflection;

var assembly = Assembly.LoadFrom("ConsoleApp5.dll");
var type = assembly.GetType("Foo");
// get the default constructor...

var constructor = type.GetConstructor(new Type[] { });
var foo = constructor.Invoke(new object[0]);
var DoSomething = type.GetMethod("DoSomething");
DoSomething.Invoke(foo, null);
```

In the above sample, we retrieved the type “**Foo**”, and then retrieved the default constructor (returned as a **ConstructorInfo** object). Once we have the constructor, we can invoke it to get a **Foo** instance in the variable **foo**. We can then get the method from the type (as a **MethodInfo**). Finally we call the **Invoke** method on **DoSomething**, passing it the **foo** object and an empty second parameter, the arguments, because **DoSomething** doesn’t take any parameters.

This is admittedly more involved than calling these methods directly, but remember that we are doing all this dynamically, at run time. We could do this even if we didn’t know anything about **Foo**, or its methods, at build time.

We could also get at the properties of **foo** in a similar fashion, if it had any. Let’s pretend there’s a property **Text**:

```
var property = type.GetProperty("Text");
var text = property.GetValue(foo) as string;
Console.WriteLine(text);
```

Question: True or false: you can construct an object and execute its methods by using reflection?

Answer: True.

Lesson 2 - Attributes

Attributes are used by the .NET Framework to give more information about a type or a type member. You can think of it as a way of adding a bit of metadata to a class, method, or property. There are already a number of attributes that are provided as part of the .NET framework.

This lesson will explore attributes, and how you can make your own custom attributes, and also how to use reflection to read the metadata that is stored in custom attributes at run time.

Lesson Objectives

After completing this lesson, you’ll be able to:

- Explain the concept of attributes in .NET.
- Create your own custom attributes.

- Retrieve attributes using .NET Reflection.

What are Attributes?

What Are Attributes?

- Use attributes to provide additional metadata about an element
- Use attributes to alter run-time behavior

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
    [Obsolete("This property will be removed in the next release.")]
    [DataMember]
    public string Name { get; set; }

    ...
}
```

In C#, we have a way of “decorating” properties, methods, and types, by using attributes. They let you add metadata to in a declarative fashion, to add more information about the various elements in your code. Attributes can be used to control behaviour, or how different properties are handled at runtime. They also facilitate programming tools like unit test frameworks and other tools that help with the development process.

.NET already makes considerable use of attributes, with many pre-defined and ready to use. For example, the **Obsolete** attribute that you can use to decorate a type or type member that has been deprecated. We've already seen a number of attributes like **JsonIgnore** and **JsonInclude** in the `System.Text.Json.Serialization` namespace, that we used to control serialization. In .NET, all the attributes ultimately derive from the **Attribute** base class.

To use an attribute, you need to ensure that the namespace is in scope (by means of a **using** statement, if it is not one of the default namespaces). You apply the attribute to a code element by placing it in square brackets preceding the declaration. If the attribute takes parameters, they are added when the attribute is applied. Let's suppose that our **DoSomething** method was superseded by the new **DoSomethingBetter** method. We've continued to maintain **DoSomething** for backwards compatibility for a couple of years now, but it's finally time for it to go. Rather than simply delete it, at the risk of breaking things, we can add the **Obsolete** attribute to it, and add some explanatory text as a parameter:

```
[obsolete("This method is deprecated. Please use
DoSomethingBetter instead.")]
public DoSomething()
```

```
{  
}
```

Interestingly, the attribute is actually called **ObsoleteAttribute**, but the compiler knows what we mean if we drop the **Attribute** suffix. It's a convention to end the names of all attribute classes with the word "Attribute".

You can apply as many attributes as you want to any one code element.

To learn more about attributes, see: <https://aka.gd/3802dMi>.

Creating and Using Custom Attributes

Creating and Using Custom Attributes

- Derive from the **Attribute** class or another attribute

```
[AttributeUsage(AttributeTargets.All)]  
public class AuthorAttribute : Attribute  
{  
    public string Email { get; }  
    public int Revision { get; }  
    public AuthorAttribute(string email, int revision)  
    {  
        Email = email;  
        Revision = revision;  
    }  
  
    [Author("Bill", 2)]  
    class Foo : IDisposable  
    {  
        . . .  
    }  
}
```

There's a wide range of attributes that the .NET framework gives you out-of-the-box. But sometimes you need an attribute that does something specific that isn't covered by the built-in options. You might have a custom testing or auditing framework that required additional metadata to be associated with each class. Or you might build some kind of documentation tool, or an error logging system, that needed to associate some additional information about every method and property.

Let's suppose you wanted a way for the developer to include some information about themselves, and attach it to a type. You need to create a class that derives from the **Attribute** base class (or some other attribute class), and give it a name ending in "Attribute", e.g. **AuthorAttribute**. You can optionally control the way your attribute is used by applying another attribute, the **AttributeUsageAttribute**, to your custom attribute class to

describe which elements you can apply this attribute to. The **AttributeUsageAttribute** can take the **ValidOn**, **AllowMultiple**, and **Inherited** parameters. The **ValidOn** parameter is probably the most useful, and specifies the program elements it can be placed on, from the **AttributeTargets** enumeration. The **ValidOn** parameter is positional, so you don't need to name it. The **AllowMultiple** parameter is just a Boolean that specifies whether the attribute can be added to an element more than once, and **Inherited** defines whether the attribute applies to derived classes.

We can define our **AuthorAttribute** class as follows:

```
[AttributeUsage(AttributeTargets.All)]
public class AuthorAttribute : Attribute
{
    public string Email { get; }
    public int Revision { get; }

    public AuthorAttribute(string email, int revision)
    {
        Email = email;
        Revision = revision;
    }
}
```

The parameters are effectively defined by whatever constructors there are, and this drives the IntelliSense you get when you add the attribute. We've added an **AttributeUsage** attribute and provided the **AttributeTargets** of **Class** and **Struct**. If we try to decorate a method or a property with this attribute, we'll get a compiler error.

We can now add that attribute to our **Foo** class, in exactly the same way as we would use a built-in attribute:

```
[Author("Bill@foo.com", 2)]
class Foo : IDisposable
{
    . . .
}
```

To find out more about custom attributes, see: <https://aka.ms/3wIGD3T>.

Processing Attributes by Using Reflection

Processing Attributes by Using Reflection

- Use reflection to access the metadata that is encapsulated in custom attributes

```
using System.Reflection;

var assembly = Assembly.LoadFrom("ConsoleApp5.dll");
var type = assembly.GetType("Foo");
var attributes = type.GetCustomAttributes<AuthorAttribute>();
Console.WriteLine("The following authors have contributed:");
foreach(var attribute in attributes)
{
    Console.WriteLine(attribute.Email);
}
```

Although we talk about “decorating” types, methods, etc. with attributes, they would not be much use if they were there merely for decoration. The main purpose of attributes is to automate various aspects of the development process. Since they are essentially metadata on code elements, the primary means of using them is through reflection. We might want to get the information about the authors of the various classes, so that we can make a document with a list of all the developers who helped make the application.

Like other information about classes, we can use reflection to extract information in attributes, including our custom ones. The **System.Reflection** namespace includes the **GetCustomAttribute** and **GetCustomAttributes** methods to get a particular attribute, or to retrieve all the attributes. Both of these methods are available in various non-generic and generic forms. There are overloads that take a type as the first parameter (referring to the type of the attribute), and versions that uses a generic type parameter, as well as overloads that don’t specify an attribute type at all. It’s probably easiest to look at an example:

```
var type = assembly.GetType("Foo");

var attributes = type.GetCustomAttributes();

foreach (var attribute in attributes)

Console.WriteLine(attribute.ToString());
```

This is one of the simpler forms and the code will list all the attributes of the Foo type. You’ll actually see something like the following:

```
System.Runtime.CompilerServices.NullableContextAttribute  
System.Runtime.CompilerServices.NullableAttribute  
AuthorAttribute
```

Here, we see our AuthorAttribute, as well as a couple of built-in attributes that the compiler has added because we have some nullable types, that we can ignore.

We could also use the generic **GetCustomAttributes** method to retrieve all the Author attributes on the **DoSomething** method:

```
var assembly = Assembly.LoadFrom("ConsoleApp5.dll");  
var type = assembly.GetType("Foo");  
var DoSomething = type.GetMethod("DoSomething");  
var attributes =  
DoSomething.GetCustomAttributes<AuthorAttribute>();
```

That's going to return an empty list – remember we didn't allow the Author attribute to be added to a member. But notice that we used the generic form that also returns a strongly typed attribute. If we used the non-generic method to return specific attributes (this time on the type again):

```
var type = assembly.GetType("Foo");  
var attributes =  
type.GetCustomAttributes(typeof(AuthorAttribute));
```

we get a collection of just the attribute we're interested in, but its an **IEnumerable<Attribute>**. To use it we'd have to cast to the more specific **AuthorAttribute**. On the other hand, the generic form gives us an **IEnumerable<AuthorAttribute>** which is more useful. Let's use that to get a list of attributes that we can then use to make our list of authors. Here's the full code:

```
using System.Reflection;  
  
var assembly = Assembly.LoadFrom("ConsoleApp5.dll");  
var type = assembly.GetType("Foo");  
var attributes = type.GetCustomAttributes<AuthorAttribute>();  
Console.WriteLine("The following authors have contributed:");  
foreach(var attribute in attributes)
```

```
{  
    Console.WriteLine(attribute.Email);  
}
```

To learn more about accessing custom attributes with reflection, see:
<https://aka.ms/3Nr1OhC>.

Lesson 3 – Generating Code

There are occasions when you need to generate code at runtime. Examples of this include various types of code generation tools, such as building language-specific APIs from a REST API specification, building wrapper classes from a database schema or a model, and cross-compilers or transpilers. In these situations, you need a framework that lets you define operations and code constructs in a general way, that can then be converted into code that can be run. There is a feature of .NET called CodeDOM that provides this.

This lesson will show how to generate code at runtime using CodeDOM.

Lesson Objectives

After completing this lesson, you'll be able to:

- Explain the purpose of CodeDOM.
- Define types and other code elements using CodeDOM.
- Generate source code files from a CodeDOM model.
- Build assemblies and executable from source files.

What is CodeDOM?

What Is CodeDOM?

- Define a model that represents your code by using:
 - The `CodeCompileUnit` class
 - The `CodeNamespace` class
 - The `CodeTypeDeclaration` class
 - The `CodeMemberMethod` class
- Generate source code from the model:
 - Visual C# by using the `CSharpCodeProvider` class
 - Visual Basic by using the `VBCodeProvider` class

You can use the .NET Code Document Object Model (CodeDOM) to generate .NET source code at runtime. The generated code is initially in the form of a generic code model that can then be emitted as source code in multiple programming languages. The built-in generators in .NET are for C# and Visual Basic. In addition, various third party code generators and community-driven projects are available to support other programming languages.

You can use CodeDOM to create source files with boilerplate template code. Another typical use is to generate source code files that act as a proxy type between your application and a web service API or a database. You use a number of namespaces in your application to use CodeDOM, including the `System.CodeDom` that has types for defining a model that represents the code you are generating, and `System.CodeDom.Compiler`, that has classes for generating source code from the code model. There are also language-specific classes, code providers, that actually generate the source code, such as the `Microsoft.CSharp.CSharpCodeProvider` class and the `Microsoft.VisualBasic.VBCodeProvider` class.

Some of the classes from `System.CodeDom` are used to create the model, which could represent a single, simple class with a few members, or a complete, complicated class hierarchy. Some of the most frequently used classes are described in the table.

Class	Purpose
<code>CodeCompileUnit</code>	Encapsulate a collection of types that will compile into an assembly.
<code>CodeNamespace</code>	Define a namespace to organize class hierarchy.
<code>CodeTypeDeclaration</code>	Declare a type, structure, interface, or enum.
<code>CodeMemberMethod</code>	Add a method to a type or interface.

CodeMemberField	Add a field to a type.
CodeMemberProperty	Add a property to a type.
CodeConstructor	Add a constructor so that you can create an instance type in your model.
CodeTypeConstructor	Add a static constructor so you can create a singleton type.
CodeEntryPoint	Define an entry point in your type, typically a static method with the name Main.
CodeMethodInvokeExpression	Create an expression that you want to execute.
CodeMethodReferenceExpression	Create a set of instructions in a method, usually used with CodeMethodInvokeExpression.
CodeTypeReferenceExpression	Use a reference type as part of an expression.
CodePrimitiveExpression	Create an expression value that can be passed as a parameter or stored in a variable.

As you will see in the next section, creating code using CodeDOM is more complex than simply writing the equivalent source code. Before embarking on using the CodeDOM, it's worth considering the possibility of simply generating source code programmatically, as text. If you only anticipate generating code in a single language, and you don't need the rigour of using CodeDOM, this might be a simpler and more pragmatic option.

To find out more about CodeDOM classes and usage, see: <https://aka.ms/3wMtFCp>.

Defining a Type and Type Members

Defining a Type and Type Members

- Defining a type with a Main method

```
using System.CodeDom;
using System.CodeDom.Compiler;

Console.WriteLine("Creating CodeDOM objects...");
var unit = new CodeCompileUnit();
var dynamicNamespace = new CodeNamespace("HelloWorld");
unit.Namespaces.Add(dynamicNamespace);
dynamicNamespace.Imports.Add(new
CodeNamespaceImport("System"));
var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);
var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);
var expression = new CodeMethodInvokeExpression(
    new CodeTypeReferenceExpression("Console"),
    "WriteLine", new CodePrimitiveExpression("Hello,
World!"));
mainMethod.Statements.Add(expression);
```

Using CodeDOM to define a class works in much the same way as writing source code in C# to define a type. The only difference is that when you use CodeDOM, you write a set of instructions that a code generator provider will use to create the source code that represents your model. This can make it a little more involved.

The following example shows how we can create a CodeDOM model for a simple “Hello World” application:

```
using System.CodeDom;
using System.CodeDom.Compiler;

Console.WriteLine("Creating CodeDOM objects...");
var unit = new CodeCompileUnit();

var dynamicNamespace = new CodeNamespace("HelloWorld");
unit.Namespaces.Add(dynamicNamespace);

dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));

var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);

var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);
```

```
var expression = new CodeMethodInvokeExpression(
    new CodeTypeReferenceExpression("Console"),
    "writeLine", new CodePrimitiveExpression("Hello, world!"));
mainMethod.Statements.Add(expression);
```

In the above listing, you can see that we start by creating a new **CodeCompileUnit** object, and then create a new namespace. We then define the namespace imports we need, in this case **System**, so that we can use the **Console** class. We then use a **CodeTypeDeclaration** object to create the new class, which will be called **Program**, and add it to the namespace. We also need to add a **CodeEntryPointMethod**, which will provide the method **Main**, after which we can provide the contents of the **Main** method which will call **Console.WriteLine**.

Generating the Source Code

Generating the Source Code

- Generate source code from CodeDOM model

```
Console.WriteLine("Generating source code...");
var provider = new Microsoft.CSharp.CSharpCodeProvider();
var fileName = @"..\..\..\HelloWorld.cs";
var stream = new StreamWriter(fileName);
var textWriter = new IndentedTextWriter(stream);
var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;
provider.GenerateCodeFromCompileUnit(unit, textWriter, options);
textWriter.Close();
stream.Close();
```

Using the classes in the **System.CodeDom** and **System.CodeDom.Compiler** namespaces, you can generate the source code equivalent of your CodeDOM model.

The following code will generate the C# version of our HelloWorld app:

```
Console.WriteLine("Generating source code...");
var provider = new Microsoft.CSharp.CSharpCodeProvider();
var fileName = @"..\..\..\HelloWorld.cs";
var stream = new StreamWriter(fileName);
var textWriter = new IndentedTextWriter(stream);
```

```
var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;
provider.GenerateCodeFromCompileUnit(unit, textwriter, options);
textwriter.Close();
stream.Close();
```

Looking at the output file, `HelloWorld.cs`, you can see the generated code:

```
//-----
// -----
// <auto-generated>
// This code was generated by a tool.
//
// Changes to this file may cause incorrect behavior and will be
// lost if
// the code is regenerated.
// </auto-generated>
//-----
// -----
namespace HelloWorld {
using System;
public class Program {
public static void Main() {
Console.WriteLine("Hello, world!");
}
}
}
```

Notice that there is a comment section at the beginning. This acts as a warning that the code is generated, and is important to avoid the risk that a developer edits the code subsequently. Editing of generated code is problematic because any changes will be lost when, inevitably, for some reason the code generator is run again. This is one of the reasons that .NET

supports **partial classes**, so that a class can be composed of both generated code and hand-written code, in separate source files.

We can make a slight change to the instantiation of the code provider, to switch it from the C# provider to the Visual Basic provider. We can also change the filename to have a “vb” extension, otherwise things get confusing:

```
Console.WriteLine("Generating source code...");  
var provider = new Microsoft.VisualBasic.VBCodeProvider();  
var fileName = @"..\\..\\..\\Helloworld.vb";  
var stream = new StreamWriter(fileName);  
var textwriter = new IndentedTextWriter(stream);  
var options = new CodeGeneratorOptions();  
options.BlankLinesBetweenMembers = true;  
provider.GenerateCodeFromCompileUnit(unit, textwriter, options);  
textwriter.Close();  
stream.Close();
```

Running this version of the code will produce the following output:

```
'-----  
-----  
' <auto-generated>  
' This code was generated by a tool.  
'  
' Changes to this file may cause incorrect behavior and will be  
lost if  
' the code is regenerated.  
' </auto-generated>  
'-----  
-----  
Option Strict Off  
Option Explicit On
```

```
Imports System  
Namespace Helloworld  
Public Class Program  
Public Shared Sub Main()  
Console.WriteLine("Hello, world!")  
End Sub  
End Class  
End Namespace
```

You can either create a new Visual Studio project, or use the command line tools to build a console app using the above code, in either language, and verify that it compiles and runs as expected.

Note: CodeDOM providers included compilation using methods like `CompileAssemblyFromFile` in older versions of .NET Framework, but this is no longer supported in .NET 6.

Lesson 4 – Assemblies

When you compile a .NET application or library, the resulting output is an **assembly**. An assembly contains Intermediate Language code (IL), along with any other resources that the application needs, and metadata. For historical reasons, a .NET assembly file has the `.dll` extension, or it can be in the form of an executable file with a `.exe` extension.

In this lesson we'll look in more detail at .NET assemblies.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand .NET assemblies.
- Identify the parts of an assembly name.

What Is an Assembly?

What Is an Assembly?

- An assembly is a collection of types and resources
- An assembly is a versioned deployable unit
- An assembly can contain:
 - IL code
 - Resources
 - Type metadata
 - Manifest

The unit of code deployment in .NET is called an **assembly**, essentially a group of types and resources that work together to perform a single task. An assembly could be made up of a portable executable (PE) file, like an executable (.exe), or a dynamic link library (.dll) file. It could also be made up of multiple such files along with external resource files. A .NET application is made up of one or more assemblies.

An assembly contains the IL code that the compiler produces from the source code. The IL code is platform independent, and is like a set of instructions for an abstract set of computer hardware and operating system. Before this code runs for the first time, a just-in-time compiler (JIT) converts the IL to machine code for the target system. The assembly also contains the code for any libraries your application depends on, and any resources such as fonts, images, or language files. The assembly also has some metadata about the types, properties, and methods that the code contains. There is also an assembly manifest that contains the version numbers, application name, description, files, references, and other metadata. The manifest is usually part of the DLL or EXE file.

Not everything makes it through the compilation process, for example the code comments will be lost. But because there is a direct mapping from the source code to the IL, it's theoretically possible to recover the original source code from the assembly, and there are many tools to do this. If your source code contains valuable intellectual property, you can try to protect it by using an **obfuscator**. This will scramble the names of variables, types and method names, so that any "decompiled" code is a lot more difficult to use. Ultimately, somebody with enough resources could figure out how your code works, and perhaps recreate it. But this was also true of binary code from a conventional compiler.

To learn more about .NET assemblies, see: <https://aka.ms/39EX1dT>

Naming Assemblies

Naming Assemblies

A version number of an assembly is a four-part string:

```
<major version>.<minor version>.<build number>.<revision>
```

Assembly name consists of the name, version, culture, token, e.g.:

ConsoleApp5, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

```
using System.Reflection;

var name = Assembly.GetExecutingAssembly().FullName;
Console.WriteLine(name);
```

When you create a new project in Visual Studio, the assembly gets an initial version number, which is usually 1.0.0.0. As the assembly changes, it is the developer's job to make sure that the version number gets incremented so it's possible to distinguish between versions. Without a version number, it's very hard to debug and reproduce production problems, so versioning assemblies is important to be able to track which version of your app your users are using.

An assembly's version number is a four-part string that is in the form: <major version>.<minor version>.<build number>.<revision>, so the default 1.0.0.0 has a major version of 1, and zero for everything else, whereas 1.2.3.4 would be a minor version of 2, a build number of 3, and a revision number of 4. Precisely how you use these versions is up to you, but a good practice is to update a major version if there are breaking changes, and update the minor version if you are just adding features. That notwithstanding, in commercial software development the use of major version updates is often a marketing decision.

An assembly has a “full assembly name” that uniquely identifies it (provided you keep updating the version). This name consists of the name of the assembly itself, along with the version number, the culture and its public key. The CLR uses this full name in order to load the right version of an assembly when it is referenced in another application. You can find the full name of the currently executing assembly using reflection:

```
using System.Reflection;

var name = Assembly.GetExecutingAssembly().FullName;

Console.WriteLine(name);
```

Applications only run with the version of an assembly that they were built with. If you want to be able to use a different version, you can create a version policy in the application configuration file (or a policy file at a higher level). This policy might redirect an assembly from the old version to a different specific version, using an assembly binding redirect.

For more detailed information about assembly versioning, see: <https://aka.gd/38DXkWn>.

Lab: Specifying the Data to Include in the Grades Report

Lab: Specifying the Data to Include in the Grades Report

Lab scenario

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reuse of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

Objectives

Exercise 1:

Creating and Applying the `IncludeInReport` attribute

Exercise 2:

Updating the Report

Exercise 3:

Storing the `Grades.UtilitiesAssembly` Centrally (If Time Permits)

Module Review and Takeaways

Review Questions

Question: Which abstract base class is used as the basis of all attributes?

- A: `ExtensionAttribute`
- B: `BaseAttribute`
- C: `AttributeAttribute`
- D: `Attribute`

Ans: D

Module 13 – Securing Data

Module Overview

Many years ago, a computer was an isolated device, usually inside a physically secure environment. Computer security was a case of remembering to lock up the computer room at night. A criminal or a spy would need to somehow get physical access to the computer, and then have enough technical knowledge to be able to extract the data. Computer security just wasn't that big an issue.

The Internet has changed all that. Most modern computers have an Internet connection and are therefore potentially exposed to every criminal on the planet. And the criminals themselves have become more sophisticated, with a black market of user-friendly tools to exploit vulnerabilities. Nowadays, you have to assume that any computer and any communications channel can be hacked. If the data is valuable, it's important to protect it from being compromised, both at rest and in transit.

In this module, you'll learn how to use the cryptography classes provided by .NET. You'll implement symmetric and asymmetric encryption and learn how to use hashes to generate mathematical representations of your data. You'll also learn how to create and manage X509 certificates and how to use them in the asymmetric encryption process.

Objectives

After completing this module, you'll be able to:

- Understand symmetric and asymmetric encryption.
- Use security libraries in the System.Security namespace.
- Understand hashes and signatures.

Lesson 1 – Application Security

Application Security is a complicated topic, and in today's security landscape it's vitally important. A common mistake is to only start thinking about security towards the end of a software development project. It can be difficult to get buy-in to the idea of setting aside resources for threat modelling and remediation because there's no immediate pay-off, and so no-one is interested. Or rather, no one is interested until something goes wrong. But at this stage it is often late in the project, or it has already deployed to production, and it can be much more difficult and expensive to retro-fit security measures.

Another mistake is to try to write your own low-level security code. Cryptographic algorithms are a specialist field of expertise, so you should use pre-built libraries, preferably ones that are baked into the platform. We'll look at those .NET libraries, and in the following lessons we'll look at how to apply these in more detail.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand threat modelling in application development.
- Understand the System.Security namespace.

What is Threat Modelling?

What is Threat Modelling?

- STRIDE
 - Spoofing
 - Tampering
 - Repudiation
 - Information disclosure
 - Denial of service
 - Elevation of privilege

Threat modelling is the process of identifying potential threats when designing software. The STRIDE model was developed by Microsoft and is a way of identifying the types of attack an application might be vulnerable to. It lets software architects find potential security problems early, when they are easier and less expensive to rectify. The meaning of the terms is as follows:

Spoofing: An attack where a person or program successfully pretends to be someone else by changing data. This is done to deceive the recipient of the message. For example, an email may appear to come from your bank, because the domain of the email address is the bank's domain, although in fact it is trivially easy to provide any *from* address of the sender's choosing. This can be prevented by using email security mechanisms like DKIM. Another common spoofing attack is to have a plausible looking website with a fake domain name. Where an application involves any kind of message passing, you need to consider steps to prevent spoofing, such as using HTTPS certificates to verify that websites are *bona fide*.

Tampering: Any intentional modification of a document or message in a way that would mislead the recipient. For example, a bad actor might change the amount of a transaction in a commercial document to obtain some advantage. One way of mitigating this kind of threat is by using some kind of digital fingerprint such as a hash function, that can be used to detect if the data has been changed in any way.

Repudiation: A situation where the originator of a document or message can deny having sent it or can claim that it isn't valid. If this is a risk, you might need to use something like a signature to verify that the document originates from the sender and hasn't been tampered with. For example, a dishonest consumer might order a product, and then deny having purchased it after delivery. In terms of mitigation, non-repudiation is typically dealt with using certificates.

Information disclosure: The leaking of valuable information that could be useful to a bad actor. Examples include the risk of identity theft, or the accidental disclosure of financial information. Encryption is the main tool for protecting confidential data.

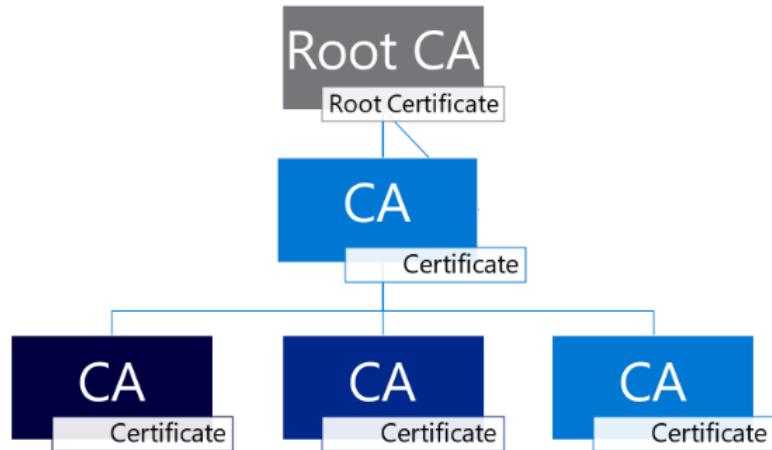
Denial of service: The overwhelming of a computer system or network resource to make it unavailable to its users. A DoS attack is usually from a single point, whereas a Distributed DoS (DDoS) attack involves multiple computers, usually compromised by means of malware, that then flood the system with requests, rendering it unavailable.

Elevation of privilege: Exploiting a bug, design flaw, or configuration mistake in an operating system or software application to get elevated access to resources that are normally protected from an application or user. For example, this could be used by an attacker to raise the authority of a regular user to that of an administrator.

Note: To learn more about threat modelling and STRIDE, see <https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats> .

The Web of Trust

The Web of Trust



One of the challenges of dealing with security threats is deciding who you can trust. Ultimately there needs to be some authority that everyone trusts. Public key infrastructure (PKI) relies on having a certificate authority that can vouch for the public keys and provide certificates (we'll look at asymmetric cryptography in more detail in a later lesson). For practical reasons there are a number of Certificate Authorities (CAs) that maintain these trusted root certificates, and then sign further certificates. These in turn can be used to sign more certificates so that they all lead back to the root certificate. It's rather like a network of personal recommendations that lead back to some established authority figure.

Typically, a commercial certificate authority will sell certificates and do some due diligence to verify the person or organization requesting it. If a root certificate gets compromised (in other words its private key is leaked in some way) it's a disaster, because all of the derived certificates are no longer secure, and sadly this has happened many times in the past. For this reason, there are mechanisms to invalidate certificates, and they typically have a finite duration of validity, just in case. This can result in problems in managing the replacement of expired certificates. This does happen from time to time, even with well-known websites, and a service outage occurs because a certificate has expired.

Web browsers and operating systems like Windows come with a number of root certificates already installed. In other words, certain certificate authorities' certificates are inherently trusted at an operating system level, which enables everything to work without having to explicitly install certificates. You can also install a certificate in an operating system's certificate store, but you need to be careful that you trust that certificate. If a bad actor were able to persuade you to install a certificate on their behalf, it would compromise your system.

In addition to certificate authorities, you can create your own certificates. These are called 'self-signed certificates' and are primarily used for development and testing. These too can be installed in your operating system's certificate store, as we'll see in a later lesson.

The System.Security.Cryptography Namespace

The System.Security.CryptographyNamespace

- Symmetric encryption (data privacy) **Aes**
- Hash signature (data integrity) **HMACSHA256, HMACSHA512**
- Certificate (digital signature) **ECDsa, RSA**
- Asymmetric encryption (key exchange) **RSA, ECDiffieHellman**
- Random number: **RandomNumberGenerator**
- Generate key from password **Rfc2898DeriveBytes**

The .NET **System.Security** namespace has a set of classes and sub-namespaces for security, including the important **System.Security.Cryptography** namespace. These classes consist of base classes for, e.g. **SymmetricAlgorithm**, which is then inherited by the more specific algorithms such as **Aes** and **TripleDES**. These will then have further inherited classes, such as **AesCryptoServiceProvider**.

AesCryptoServiceProvider is the default provider for **Aes**, and in general you won't need to work with **AesCryptoServiceProvider** directly, but rather you'll use the base algorithm class, **Aes**, which will then invoke **AesCryptoServiceProvider**. Where possible, the classes like **AesCryptoServiceProvider** are wrappers around the corresponding implementations of the algorithm provided by the operating system. For example, on Windows machines it will use the underlying Windows Cryptography API.

Not all algorithms are suitable for new development; many are no longer recommended but have been kept available for backwards compatibility. The currently recommended algorithms are:

- Symmetric encryption (data privacy): **Aes**
- Hash signature (data integrity): **HMACSHA256, HMACSHA512**
- Certificate (digital signature): **ECDsa, RSA**
- Asymmetric encryption (key exchange): **RSA, ECDiffieHellman**
- Random number: **RandomNumberGenerator**
- Generate key from password: **Rfc2898DeriveBytes**

Note: For a general introduction to the .NET cryptography model, see <https://learn.microsoft.com/en-us/dotnet/standard/security/cryptography-model> .

Question: What are the different threats represented by the STRIDE model?

Answer: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

Lesson 2 – Symmetric Encryption

Symmetric encryption is used to cryptographically transform data using a mathematical procedure. Symmetric encryption is a reliable and established way of safeguarding private data.

In this section, we'll look at several .NET classes that allow your applications to protect confidential data using encryption and hashing.

Lesson Objectives

After completing this lesson, you'll be able to:

- Understand symmetric encryption.
- Use symmetric encryption to encrypt and decrypt data.
- Uses hash functions to create digital fingerprints of data for verification.

What Is Symmetric Encryption?

What Is Symmetric Encryption?

- Symmetric encryption is the cryptographic transformation of data by using a mathematical algorithm
- The same key is used to encrypt and decrypt the data
- The `System.Security.Cryptography` namespace includes:
 - `DESCryptoServiceProvider` class
 - `AesManaged` class
 - `RC2CryptoServiceProvider` class
 - `RijndaelManaged` class
 - `TripleDESCryptoServiceProvider` class

The word "symmetric" comes from the fact that both encrypting and decrypting the data use the same secret key. Because of this, when you use symmetric encryption, the secret key must be kept safe. Historically, the secret key would be a notebook or pad containing the key. This codebook had to be protected at all costs, and getting it to the other party, for example a secret agent in the field, could be challenging. If the codebook was intercepted, then all was lost. This is the main weakness of symmetric encryption: there has to be initial contact to exchange the key in a secure way. Once this is achieved, symmetric encryption is very effective and efficient for large amounts of data.

In addition, the key must be large enough to avoid the possibility of it being worked out by cryptanalysts. Famously, the keys used by the Germans during World War II were frequently obtained by the British, using machines of increasing complexity and ingenuity. The culmination of that effort also gave rise to the first machines we could call electronic computers. As computers have become more powerful, the requisite key length has also increased. Encryption algorithms and key lengths that were considered unbreakable at the turn of the 20th century (to the extent that the US Government considered them 'munitions') can now be broken routinely with today's general-purpose computers. The larger the key, i.e. the more bits, the better the strength of the encryption, and the more difficult it is for a bad actor to decrypt the data without the key.

Many symmetric encryption algorithms use an initialization vector (IV) in addition to a secret key to make symmetric encryption work better. The IV is a random group of bytes that helps to make the first block of encrypted data more random. The IV makes it much harder for a bad person to read your encrypted data.

There are a number of implementations of different symmetric encryption algorithms in the `System.Security.Cryptography` namespace. These include for Advanced Encryption Standard (AES - recommended), and legacy algorithms such as Data Encryption Standard (DES), and TripleDES. Each of these classes is derived from the abstract `SymmetricAlgorithm` base class, and is a block cipher, meaning that the algorithm will convert the unencrypted data into fixed-length blocks before encrypting each block. The characteristics of these classes is shown in the following table:

Algorithm	Class	Technique	Block Size	Key Size
AES	Aes	Advanced Encryption Standard - Substitution Permutation Network (SPN)	128 bits	128, 192, or 256 bits
DES	DES	Data Encryption Standard - Bit shifting and bit substitution	64 bits	64 bits
RC2	RC2	Rivest Cipher 2 - Feistel network	64 bits	40-128 bits
Rijndael	Rijndael	SPN	128-256 bits	128, 192, or 256 bits
TripleDES	TripleDES	Bit shifting and bit substitution	64 bits	128-192 bits

Important: The current recommendation is to use **AES** for all future symmetric encryption applications. The other algorithms are provided for backwards compatibility and legacy applications.

Note: To learn more about the symmetric encryption classes in .NET, see <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.symmetricalgorithm?view=net-6.0> .

Encrypting Data by Using Symmetric Encryption

Encrypting Data by Using Symmetric Encryption

To encrypt and decrypt data symmetrically, perform the following steps:

1. Create an Rfc2898DeriveBytes object
2. Create an Aes object
3. Generate a secret key and an IV
4. Create a stream to buffer the transformed data
5. Create a symmetric encryptor or decryptor object
6. Create a CryptoStream object
7. Write the transformed data to the buffer stream
8. Close the streams

You can use any of the symmetric encryption classes in the **System.Security.Cryptography** namespace to protect data. These classes provide implementations of a specific encryption algorithm. For example, the **Aes** class provides an implementation of the AES algorithm. Aside from using an algorithm to encrypt and decrypt data, and read or write encrypted data from a stream, they can also be used to get a secret key and an IV from a password, or from a password and a salt. In this context, a salt means a set of random bits that, along with a password, can be used to make a more secure secret key and IV.

In addition to the encryption algorithms, there are additional supporting classes that you use to implement data security. The **Rfc2898DeriveBytes** class implements a password-based key derivation function (PBKDF2), and is compliant with the Public-Key Cryptography Standards (PKCS). You can use **Rfc2898DeriveBytes** to derive secret keys and IVs from a password and salt. The **CryptoStream** class is based on the abstract **Stream** base class in the **System.IO** namespace. It lets you read and write cryptographic transformations using I/O streams.

Note: To see the documentation for these classes, see <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rfc2898derivebytes> and <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptostream>.

Using these classes requires some understanding of the whole process of encrypting and decrypting data. In practice, you would probably write your own **Encrypt** and **Decrypt** methods that encapsulate the encryption details, using your chosen algorithm. Here's what these methods might look like in a typical implementation using the AES algorithm and a simple password:

```
static void Encrypt(Stream stream, string password, byte[] salt, string plaintext)

{

    // create instance of encryption class – we are using AES

    using (var aes = Aes.Create())

    {

        var passwordHash = new Rfc2898DeriveBytes(password, salt, 1000,
        HashAlgorithmName.SHA1);

        aes.Key = passwordHash.GetBytes(aes.KeySize / 8); // create key from passwordHash

        // Write the Initialization Vector to the filestream

        stream.Write(aes.IV, 0, aes.IV.Length);

        using (var cryptoStream = new CryptoStream(stream,
```

```

aes.CreateEncryptor(), CryptoStreamMode.Write))

{

using (var writer = new StreamWriter(cryptoStream))
{
    writer.WriteLine(plaintext);

}
}
}
}

```

We first create an instance of the algorithm, and in this case we've chosen AES. We then create a key from the password and the salt using the **GetBytes** method of the **Rfc2898DeriveBytes** class. We use the `aes.KeySize` variable to ensure that the key is the correct size for the algorithm used. We could also generate our own Initialization Vector using **Rfc2898DeriveBytes**, or we can let the algorithm generate this automatically. Either way, a best practice is to generate a new IV each time a message is encrypted, and then write the IV to the stream, otherwise encrypting the same plaintext always generates identical ciphertext. We then create a **CryptoStream** object from the supplied stream. The **CryptoStream** constructor requires an Encryptor object corresponding to the algorithm, which we create using the **Aes.CreateEncryptor** method. We can then use the **CryptoStream** to write out our plaintext, in this case through a filestream to a file. The file will then contain the IV bytes, followed by the encrypted text.

The corresponding Decrypt function essentially is the reverse of the **Encrypt** function. It reads the IV from the stream, and then uses a **CryptoStream** in Read mode. The key must be the same as that used to encrypt, so we have to use the same password and salt to generate the key (using the same algorithm). The **Decrypt** method code would be as follows:

```

static string Decrypt(Stream stream, string password, byte[] salt)

{
    // create instance of encryption class – we are using AES
    using (var aes = Aes.Create())
    {
        var passwordHash = new Rfc2898DeriveBytes(password, salt, 1000,
        HashAlgorithmName.SHA1);
    }
}

```

```
aes.Key = passwordHash.GetBytes(aes.KeySize / 8); // create key from passwordHash

// Read the Initialization Vector from the filestream

byte[] iv = new byte[aes.BlockSize / 8];

stream.Read(iv, 0, iv.Length);

aes.IV = iv;

using (var cryptoStream = new CryptoStream(stream,
aes.CreateDecryptor(), CryptoStreamMode.Read))

{

using (var reader = new StreamReader(cryptoStream))

{

return reader.ReadToEnd(); ;

}

}

}

}
```

A simple driver for these functions would need to provide a password string and a value for the salt, which should be a byte array of minimum length 16:

```
byte[] salt = Encoding.UTF8.GetBytes("CryptoConsole123");
```

The stream could be a **FileStream** or a **MemoryStream**. The following example code calls the **Encrypt** method to save a string to an encrypted file, and then calls the **Decrypt** method to read the file and send the decrypted string to the console:

```
using System.Security.Cryptography;

using System.Text;

Console.WriteLine("Crypto test");

Console.WriteLine();

var password = "password"; // Demo code only: never hard-code a password!

byte[] salt = Encoding.UTF8.GetBytes("CryptoConsole123");
```

```

var filename = "..\\..\\..\\EncryptedData.txt";

using (var stream = new FileStream(filename, FileMode.OpenOrCreate))
{
    Encrypt(stream, password, salt, "Hello World of plaintext!");
}

using (var stream = new FileStream(filename, FileMode.Open))
{
    var decrypted = Decrypt(stream, password, salt);
    Console.WriteLine("Decrypted text:");
    Console.WriteLine(decrypted);
}

```

If you run this code, you'll be able to check that the contents of TestData.txt is, indeed, encrypted.

Hashing Data

- A hash is a numerical representation of a piece of data
- A hash can be computed by using the following code

```

public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
    using (var hashAlgorithm = new HMACSHA1(secretKey))
    {
        using (var bufferStream = new MemoryStream(dataToHash))
        {
            return hashAlgorithm.ComputeHash(bufferStream);
        }
    }
}

```

The purpose of hashing data is to derive a mathematical ‘signature’ or ‘fingerprint’. The signature doesn’t contain the data itself and is usually much smaller than the data and of fixed length. But if the hash is sufficiently large there is a negligible likelihood that two documents will generate the same hash. The hash function therefore becomes a very efficient way of verifying that a document hasn’t been altered since the hash was generated.

Suppose our FourthCoffee service receives coffee orders through a payment processing system, that then require fulfillment. What if a bad actor were to change the coffee order, perhaps to fraudulently obtain extra supplies. We need a way of knowing that the order details have not been tampered with. One strategy would be to compute an MD5 hash of the order before sending. When the FourthCoffee service receives the order it also computes a hash and sends it back to the payment processing system for checking. The two hashes should match.

But what if the coffee thieves figured out that we were using the MD5 hash algorithm? Since they presumably have a way of intercepting and tampering with the communications, they could generate their own hash and substitute it. To prevent this, we can have a secret key value that’s known only to the FourthCoffee and the payment system, and then compute the hash of the message plus the key. The criminals now need to know the value of the key in addition to knowing the algorithm, otherwise the hashes they generate won’t match.

The **System.Security.Cryptography** namespace includes a number of classes that provide the more commonly used hash algorithms. These are derived from the **HashAlgorithm** base class, as tabulated below:

Class	Description
SHA512	Implementation of the Secure Hash Algorithm (SHA) able to compute a 512-bit hash
SHA384	Implementation of the Secure Hash Algorithm (SHA) able to compute a 384-bit hash
SHA256	Implementation of the Secure Hash Algorithm (SHA) able to compute a 256-bit hash
HMAC	Base class for hash-based Message Authentication Code (HMAC) algorithms such as HMACSHA256, HMACMD5, HMACRIPEMD160, etc.
MACTripleDES	Algorithm that computes a Message Authentication Code (MAC) using TripleDES
MD5	Message Digest algorithm with 128 bit hash size
RIPEMD160	Algorithm designed to replace MD5 with a 160-bit hash size

Important: The current recommendation is to use **SHA256**, **SHA384**, or **SHA512** for all future applications. The MD5, RIPEMD160 and SHA1 algorithms are provided for backwards compatibility and legacy applications.

The following example code shows how to generate the hash of a byte array.

```
var text = @"The current recommendation is to use SHA256,  
SHA384, or SHA512 for all future applications. The MD5,  
RIPEMD160 and SHA1 algorithms are provided for backwards  
compatibility and legacy applications.";  
  
byte[] data = Encoding.UTF8.GetBytes(text);  
  
Console.WriteLine();  
  
Console.WriteLine("SHA512 hash of text:");  
  
var hash = SHA512.HashData(data);  
  
Console.WriteLine(BitConverter.ToString(hash));
```

Most of this code is concerned with setting up a byte array to feed in to the **SHA512.HashData** hash function. This is a static method, but you can also create an instance of the class and use methods on, for example if you have a log of documents that you need to compute the hash of. If you run this code, you get the same output every time:

SHA512 hash of text:

E7-E0-78-6C-0C-FC-2A-DB-7C-E3-4E-D6-95-2A-D1-B4-40-A5-A4-51-4F-BE-2C-A9-35-65-F0-63-86-55-30-A3-B4-70-2C-49-F2-3B-A8-BF-8E-8B-DE-EE-81-E0-35-2F-85-CF-25-DA-E0-33-89-E4-7C-6D-9B-ED-EC-DB-3C-03

If we change just a single character in the input data, the entire hash will be different. For example, if we change the first letter of the input text from upper case to lower case, we get the following:

SHA512 hash of text:

29-90-2E-E9-BD-71-85-88-FF-60-56-BB-CE-04-A1-BB-64-FC-E3-1C-23-19-C6-5A-5F-20-88-9A-24-D8-56-19-FD-49-B6-35-99-4D-F6-DA-57-1A-56-3D-D4-A8-C5-69-9E-48-87-FF-C0-4C-2A-38-5A-25-4D-06-FD-CD-B0-1F

Note: To see the documentation for .NET hash classes in the **System.Security.Cryptography** namespace, see <https://learn.microsoft.com/en-us/dotnet/standard/security/ensuring-data-integrity-with-hash-codes>.

Demonstration: Encrypting and Decrypting Data

Demonstration: Encrypting and Decrypting Data

In this demonstration, you will use symmetric encryption to encrypt and decrypt a message

In this demonstration, you will use symmetric encryption to encrypt and decrypt a message.

Question: Is it possible to have two different documents that generate the same hash code?

Answer: While it is theoretically possible, it's astronomically unlikely that this would happen in normal operation.

Lesson 3 – Asymmetric Encryption

Unlike symmetric encryption which uses the same key to encrypt and decrypt data, asymmetric encryption uses an asymmetric encryption algorithm and a combination of a public and a private key. These keys are related mathematically in such a way that one can be fairly easily derived from the other, but only in one direction.

In this lesson, you'll learn about the classes and tools that can be used to implement asymmetric encryption in your C# applications.

Lesson Objectives

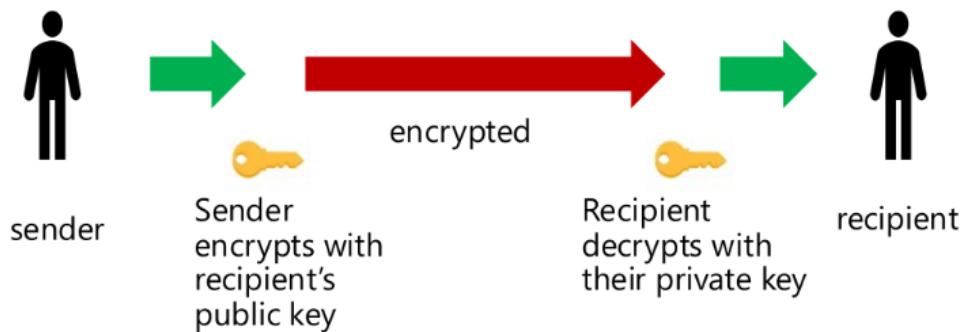
After completing this lesson, you'll be able to:

- Understand asymmetric encryption.
- Encrypt and decrypt data using asymmetric encryption.
- Create and manage X509 certificates.
- Manage encryption keys in applications.

What Is Asymmetric Encryption?

What Is Asymmetric Encryption?

- Asymmetric encryption uses:
 - A public key to encrypt data
 - A private key to decrypt data



If you take two numbers, it is fairly trivial to compute the product. For example, $17 \times 23 = 391$. But if you were given the number 391 and asked to work out what the factors were, you'd have quite a job to figure it out. As long as the two numbers are prime numbers, there's a unique relationship between the product and its two factors, but it's a lot easier to go in one direction than the other. Although finding the factors of 391 is time consuming, it's not unrealistic. If the two prime numbers were enormous, you'd need a special large integer library to compute the product, but it wouldn't be a difficult calculation. But factorizing the product by 'brute force' might be impossible in any reasonable time scale. This is an example of a "one-way function" that's easy to compute in one direction, but nearly impossible to do in the other, unless you know the answer. There are many other mathematical operations that have this characteristic, and they form the basis of asymmetric cryptography.

You could think of the product, 391, as the ‘public key’, and the two factors, 17 and 23, as the ‘private key’, the secret knowledge that can be easily verified against the public key if you know it. Such a pair of keys have some interesting characteristics when used as an encryption key in an asymmetric encryption algorithm. If you encrypt something with the *public* key, it can only be decrypted with the corresponding *private* key. And if you encrypt something with the *private* key it can only be decrypted with the *public* key. Suppose we encrypt a message with the public key: only the holder of the private key can decrypt it. That means you can send me a message using my (non-secret) public key, and only I will be able to decrypt it. Significantly, we don’t need to exchange any keys for this to work, which overcomes the main weakness of symmetric cryptography, and makes it ideal for encrypting traffic over public networks. Obviously it’s critical that I protect my private key – if anybody else is able to obtain it then they’ll also be able to decrypt any messages encrypted with the public key.

There’s a second use for asymmetric cryptography, and that’s to provide a way of verifying that a message is genuine. By encrypting a message with the private key, it’s possible for the receiver of the message to verify that it can be decrypted with the public key, thus proving that it came from the owner of the public key. This is the basis of certificates and digital signatures. An example application is where browsers, for example, are able to check that when you go to <https://www.microsoft.com/>, you really are visiting the Microsoft web site and not some spoof site that belongs to a fraudster.

The one drawback with asymmetric encryption is that it’s much less efficient than symmetric encryption. It’s relatively slow and only encrypts data in small chunks. For this reason, most bulk data encryption works by generating symmetric keys for the communications session, and then using asymmetric encryption to securely exchange the symmetric keys. The rest of the communication is carried out using the symmetric keys. Here’s how that would typically work at a high level:

1. Encrypt the data by using a symmetric algorithm, such as AES.
2. Encrypt the symmetric key by using an asymmetric algorithm.
3. Create a stream and write bytes for the following:
 - a. The length of the IV
 - b. The length of the encrypted secret key
 - c. The IV
 - d. The encrypted secret key
 - e. The encrypted data

The decryption process is essentially the reverse of the above, extracting the IV and key, decrypting the key, and using the decrypted key to decrypt the data.

Encrypt/Decrypt Data by Using Asymmetric Encryption

Encrypt/Decrypt Data by Using Asymmetric Encryption

```
var plainText = "Hello world...";  
var rawBytes = Encoding.Default.GetBytes(plainText);  
var decryptedText= string.Empty;  
  
using (var rsaProvider = RSA.Create())  
{  
    var padding = RSAEncryptionPadding.Pkcs1;  
    var encryptedBytes = rsaProvider.Encrypt(rawBytes, padding);  
    var decryptedBytes = rsaProvider.Decrypt(encryptedBytes,  
padding);  
    decryptedText= Encoding.Default.GetString(decryptedBytes);  
    // The decryptedTextvariable will now contain "Hello world..."  
}
```

There are several classes for asymmetric encryption in .NET, using the RSA (Rivest–Shamir–Adleman) and DSA (Digital Signature Algorithm) algorithms. The DSA algorithm is no longer being actively supported, but the classes are still available for backwards compatibility. For new development the RSA algorithm should be used, and there are a few classes including the default **RSACryptoServiceProvider**. This is the best choice for cross-platform support, and the recommendation is to use the base **RSA** class which will automatically use the default provider. In other words, you will get an instance of **RSACryptoServiceProvider** if you just use the **RSA.Create()** method:

```
using (var RSA = RSA.Create())  
{  
    Var encryptedKey = RSA.Encrypt(SymmetricKeyToEncrypt, RSAEncryptionPadding.Pkcs1);  
    // more code using RSA...  
}
```

The RSA base class has methods, inherited by the specific implementations, that let you use asymmetric encryption in your applications. For example, you can import and export key information, and encrypt and decrypt data. In the following example we are using the Encrypt method to encipher the contents of a string, and then Decrypt to recover the plaintext.

```
using System.Security.Cryptography;
using System.Text;

var plainText = "Hello world...";

var rawBytes = Encoding.Default.GetBytes(plainText);

var decryptedText = string.Empty;

using (var rsaProvider = RSA.Create())

{

    var padding = RSAEncryptionPadding.Pkcs1;

    var encryptedBytes = rsaProvider.Encrypt(rawBytes, padding);

    var decryptedBytes = rsaProvider.Decrypt(encryptedBytes, padding);

    decryptedText = Encoding.Default.GetString(decryptedBytes);

    // The decryptedText variable will now contain "Hello world..."

}

Console.WriteLine(decryptedText);
```

Note: The padding parameter determines whether the Encrypt and Decrypt methods use Optimal Asymmetric Encryption Padding (OAEP) or PKCS#1 v1.5 padding. The most important thing is to use the parameter consistently.

Typically, applications do not encrypt and decrypt data in the scope of the same **RSA** object. One application may perform the encryption, and then another performs the decryption. If you attempt to use different **RSA** objects to perform the encryption and decryption, without sharing the keys, the **Decrypt** method will throw a **CryptographicException**. For this reason, the **RSA** class has methods to export and import the public and private keys.

In the following code example, you can see how to instantiate different **RSA** objects and use the **ExportParameters** and **ImportParameters** methods to share the public and private keys between separate instances.

```
using System.Security.Cryptography;
using System.Text;

var plainText = "Hello world...";

var rawBytes = Encoding.Default.GetBytes(plainText);
```

```
byte[] encryptedBytes;

var decryptedText = string.Empty;

var padding = RSAEncryptionPadding.Pkcs1;

RSAParameters keyparams;

var exportPrivateKey = true;

using (var rsaProvider = RSA.Create())

{

    keyparams = rsaProvider.ExportParameters(exportPrivateKey);

    encryptedBytes = rsaProvider.Encrypt(rawBytes, padding);

}

using (var rsaProvider = RSA.Create())

{

    rsaProvider.ImportParameters(keyparams);

    var decryptedBytes = rsaProvider.Decrypt(encryptedBytes, padding);

    decryptedText = Encoding.Default.GetString(decryptedBytes);

}

Console.WriteLine(decryptedText);
```

Note: The `exportPrivateKey` parameter instructs the `ExportParameters` method to include the private key in the return value. If you pass `false` into the method, the return value will not contain the private key. If you then try to decrypt data without importing the private key, the `Decrypt` method will throw a `CryptographicException`. To learn more about the `RSACryptoServiceProvider` class see <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsacryptoserviceprovider>.

Creating X509 Certificates

Creating X509 Certificates

Use the `New-SelfSignedCertificate` PowerShell cmdlet to create certificates

```
New-SelfSignedCertificate -Subject "CN=Grades" -HashAlgorithm sha1  
-KeyExportPolicy Exportable -CertStoreLocation  
"Cert:\CurrentUser\My" -KeySpec KeyExchange
```

For the code described in the last example to be useful, you probably need some way to persist these keys in your application. One way of doing this is in an X509 certificate, which can be saved in the certificate store on your computer. An X509 certificate is a digital document that has information like the name of the organization that is providing the data. It can also be used to store public and private keys for asymmetric encryption and decryption.

X509 certificates are valuable, do they are kept in certificate stores, and these can be under the user's account, service accounts, or scoped to the local computer in a typical Windows installation.

Typically, an organization will obtain certificates from a certificate authority, who may or may not do this on a commercial basis. The reason is that the certificate authority is trusted, and so there is some verification of the certificates and who they're supplied to. This is important for an SSL certificate, for example, because you need a certificate authority whose root certificates are trusted by the suppliers of web browsers.

You can also create your own certificates. These are called self-signed certificates, and you can create them as required. It's then up to the consumer of the certificate to decide if they trust you or not. For internal use, and for development purposes, this very local web of trust may be sufficient. It's not recommended to use self-signed certificates for production use - they aren't trusted by default and thus can be a challenge to maintain, and may not comply with the latest security standards. Instead, purchase a certificate from a well-established certificate authority.

The recommended way to create a certificate on Windows is to use the `New-SelfSignedCertificate` PowerShell cmdlet from an administrator PowerShell window.

Note: There used to be a `makecert` command-line tool. This has been replaced by the PowerShell equivalent and is not normally installed on newer Windows systems. You will still see `makecert` referred to in security articles, etc.

Here's an example of creating a self-signed certificate in the LocalMachine store:

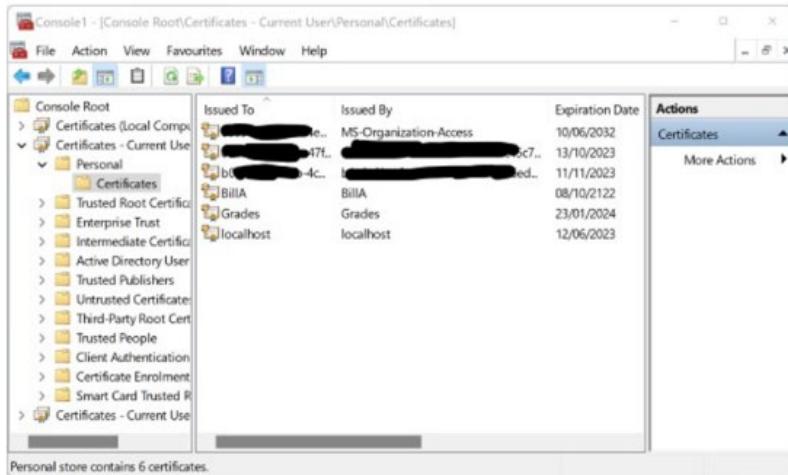
```
New-SelfSignedCertificate -Subject "CN=MyCert" -HashAlgorithm sha1 -KeyExportPolicy Exportable -CertStoreLocation "Cert:\LocalMachine\My" -KeySpec KeyExchange
```

For more information about the **New-SelfSignedCertificate** cmdlet, see <https://learn.microsoft.com/en-us/azure/active-directory/develop/howto-create-self-signed-certificate>.

Managing X509 Certificates

Managing X509 Certificates

Use the MMC Certificates snap-in to manage your certificate stores



The Microsoft Management Console (MMC) Certificates snap-in lets you manage any X509 certificates installed on your user account, service account, or local computer. Follow these steps to use the MMC snap-in to open a certificate store:

1. Log on as an administrator. If you log on without administrative privileges, you will only be able to view certificates in your user account certificate store.
2. Go to Windows Start, and type mmc.exe, and then return.
3. At the User Account Control prompt, click on Yes.
4. In the MMC window, on the File menu, click Add/Remove Snap-in.
5. In the Add or Remove Snap-ins dialog box, in the Available snap-ins list, click Certificates, and then click Add.

6. In the Certificates snap-in dialog box, click either My user account, Service account, or Computer account, and then perform one of the following steps:
 - a. If you chose My user account, click Finish.
 - b. If you chose Service account, click Next, and then perform the following steps:
 - i. In the Select Computer dialog box, click Next.
 - ii. In the Certificates snap-in dialog box, in the Service account list, click the service account you want to manage, and then click Finish.
 - c. If you chose Computer account, click Next, and then click Finish.
7. In the Add or Remove Snap-ins dialog box, repeat steps 4 and 5 if you want to add additional certificate stores to your session, and then click OK.

After you have opened one or more certificate stores, you can view the properties that are associated with any X509 certificate in any of the certificate stores, such as Personal or Trusted Root Certificate Authorities stores. You can also export an X509 certificate from a certificate store to the file system, manage the private keys that are associated with an X509 certificate, issue a request to renew an existing X509 certificate, or delete a certificate from the store.

You can also use PowerShell to remove a certificate as follows:

```
$cert=Get-ChildItem -Path "Cert:\LocalMachine\My" | Where-Object {$_.Subject -Match "MyCert"} | Select-Object Thumbprint
```

```
Remove-Item -Path "Cert:\LocalMachine\My\$($cert.Thumbprint)" -DeleteKey
```

As with all PowerShell delete operations, you should use **Remove-Item** with extreme caution.

Managing Encryption Keys

Managing Encryption Keys

The System.Security.Cryptography.X509Certificates namespace contains classes that enable access to the certificate store and certificate metadata

```
var store = new X509Store(  
    StoreName.My,  
    StoreLocation.LocalMachine);  
  
store.Open(OpenFlags.ReadOnly);  
  
foreach (var storeCertificate in store.Certificates)  
{  
    // Code to process each certificate.  
}  
  
store.Close();
```

The **System.Security.Cryptography.X509Certificates** namespace has several classes that help you use X509 certificates and their keys in your applications. The following are part of these classes:

- **X509Store.** This class lets you access a store of certificates and do things like look for an X509 certificate with a certain name.
- **X509Certificate2.** With this class, you can make an in-memory copy of an X509 certificate that is already in a certificate store. When you create an X509Certificate2 object, you can use its members to get information, like the public and private keys of the X509 certificate.
- **PublicKey.** This class lets you change the value or metadata associated with an X509 certificate's public key.

The code below shows how to use the X509Store and X509Certificate2 classes to list the contents of the personal certificate store on the local machine:

```
var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);  
  
var certificateName = "CN=MyCert";  
  
store.Open(OpenFlags.ReadOnly);  
  
foreach (var storeCertificate in store.Certificates)  
{  
  
if (storeCertificate.SubjectName.Name == certificateName)
```

```
Console.WriteLine("Found cert with thumbprint = " + storeCertificate.Thumbprint);
}

store.Close();
```

After making an **X509Certificate2** object, you can use its members to find out, for example, if an X509 certificate has a public key or a private key. The list below shows some of the **X509Certificate2** properties and methods you can use:

HasPrivateKey. Shows whether the certificate contains a private key.

FriendlyName. Gets the friendly name (alias) associated with the certificate.

GetPublicKeyString. Extracts the public key as a string value.

GetRSAPublicKey. Gets the RSA public key.

GetRSAPrivateKey. Gets the RSA private key.

The public and private keys can be used to create an instance of the **RSACryptoServiceProvider** class:

```
// Code to set the public and private keys.

// Create an RSA encryptor.

var rsaEncryptorProvider = storeCertificate.GetRSAPublicKey();

// Create an RSA decryptor.

var rsaDecryptorProvider = storeCertificate.GetRSAPrivateKey();
```

Note: to learn more about managing encryption keys, see the documentation at <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates> .

Question: In asymmetric cryptography, what are the consequences of someone getting access to your public key from a public/private key pair?

Answer: The public key does not need to be kept secret, and in fact it must be divulged to a party that you want to enable to send you encrypted messages. The private key, on the other hand, must be kept secret.

Lab: Protecting the Grades Report Data

Demonstration: Encrypting and Decrypting Grade Reports Lab

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

Module Review and Takeaways

Review Questions

Question: Fourth Coffee wants you to implement an encryption utility that can encrypt and decrypt large image files that exceed 200 megabytes (MB) in size. Only a small internal team will be expected to use this tool, so controlling who can encrypt and decrypt the data is not a concern. Which of the following techniques will you choose?

- Symmetric encryption
- Asymmetric encryption
- Hashing

Answer: A: Symmetric encryption

Question: True or false: asymmetric encryption uses public keys to encrypt data?

Answer: True, the recipients public key is used to encrypt the data, and the recipient then uses their private key to decrypt it.