The C# Programming Language

CONTENTS

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions

PRE-REQUISITES

Essential skills:

- Prior experience in programming:
  - Declaring variables
  - Writing loops
  - Passing arguments / parameters
  - Invoking functions

Beneficial skills

- Awareness of object-oriented principles
  - Objects have 'state' and 'behaviour'

- Working with a modern IDE: Visual Studio, Eclipse, etc.

Note: Java and C# are very similar. This course is not intended for those who are already fluent in Java.

COURSE OBJECTIVES

- Gain a solid understanding of the C# programming language
- Develop practical object-oriented programming (OOP) skills
- Navigate the Visual Studio Integrated Development Environment (IDE) and make use of its code completion features

COURSE OUTLINE

- Introduction to C#
- C# variables and datatypes
- Conditional statements
- Loops and collections
- Introduction to object-oriented programming (OOP)
- Methods and parameters
- Properties and constructors
- Inheritance and abstract classes
- Interfaces
- Delegates and Lambdas
- Language Integrated Query (LINQ)
- Exception handling

Activity: Introductions

- → Preferred name
- → Organisation & role
- → Experience of C# and OOP
- → Key learning objective
- → Hobby

QUESTIONS

Golden rule

- 'There is no such thing as a stupid question'

First amendment to the golden rule

- '... even when asked by an instructor'
- Please have a go at answering questions

Corollary to the golden rule

- 'A question never resides in a single mind'
- By asking a question, you're helping everybody

Introduction to C#

OUTLINE

- What is .NET?
- .NET compilation
- What is C#?
- Hello World in .NET 6 and .NET 5
- Hello World explained
- Namespaces
- Visual Studio
- Keyboard shortcuts

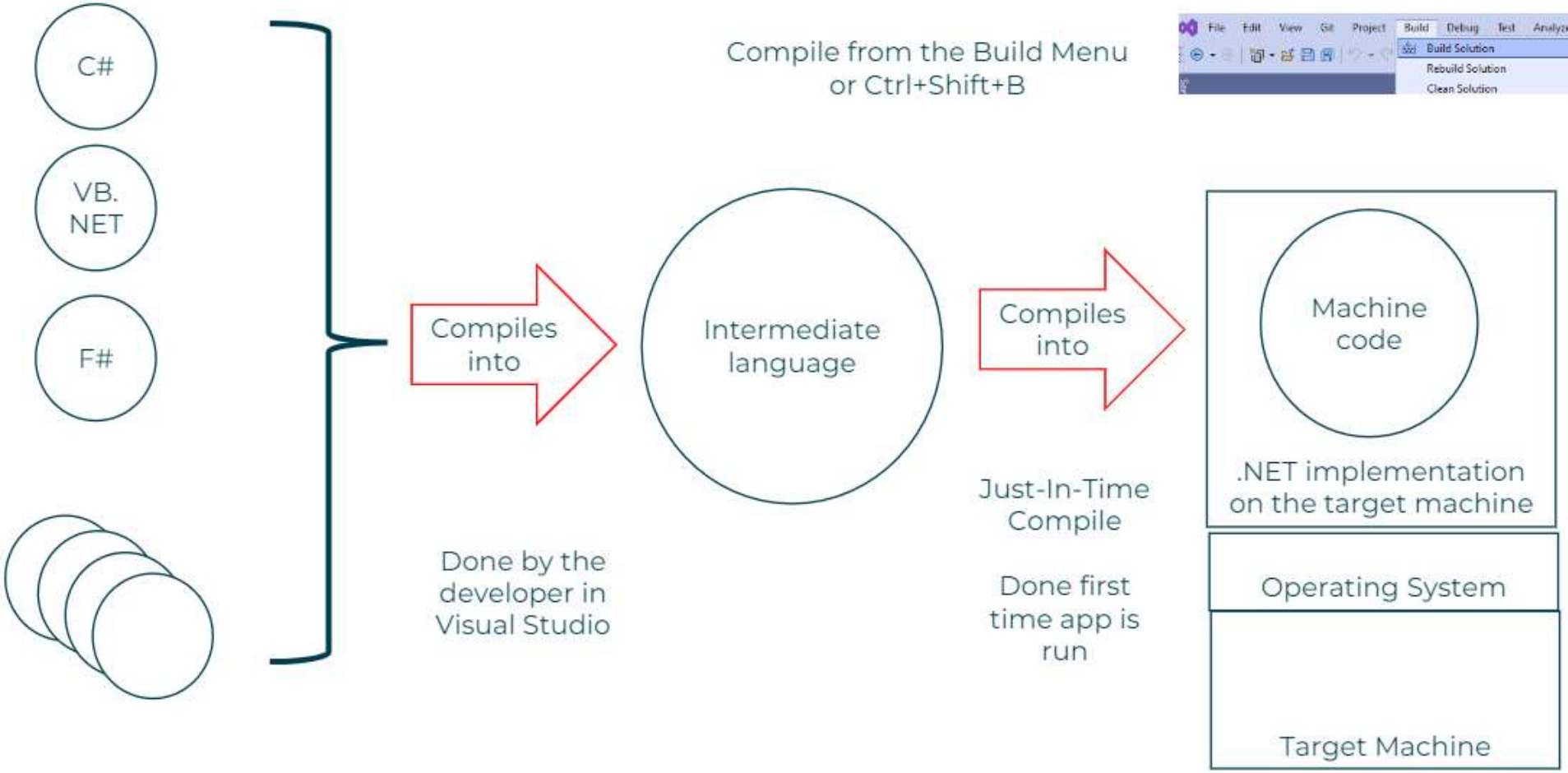
WHAT IS .NET?

- .NET is a free, cross-platform, open source developer platform
- With .NET, you can build different applications such as web, mobile, desktop, games, and IoT
- You can write .NET apps in C#, F#, or Visual Basic
- Different .NET implementations target different operating systems:
  - .NET is a cross-platform implementation for websites, servers, and console apps on Windows, Linux, and macOS
  - .NET Framework supports websites, services, desktop apps, and more on the Windows operating system
  - Xamarin/Mono is an implementation for running apps on all major mobile operating systems
- .NET Standard is a base set of APIs (Application Programming Interfaces) that are common to all .NET implementations
- NuGet is a package manager that stores tens of thousands of packages that can extend the base functionality of .NET

Page 11

Author: Pete Behague Subject: Highlight Date: 11/28/2022 11:21:56 AM

.NET Compilation



WHAT IS C#?

- C# is an object-oriented and type-safe programming language
- Type safety means C# declares data with a specific datatype and it checks either at compile time or runtime that the correct type of data is being used for the operations that are being performed
- C# is case sensitive

- NOTE: This course uses C# 10.0 and .NET 6

Page 13

Author: Pete Behague Subject: Highlight Date: 11/28/2022 11:24:55 AM

HELLO WORLD .NET 6

With .NET 6, a Console Application contains the following code only:

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

Microsoft Visual Studio Debug Console

```
Hello, World!

C:\Labs\HelloWorldSolution\HelloWorld\bin\Debug\net6.0\HelloWorld.exe
To automatically close the console when debugging stops, enable Tools-
le when debugging stops.
Press any key to close this window . . .
```

HELLO WORLD .NET 5

With .NET 5, a console application contains the following code:

```
using System;

namespace HelloWorld_NET5
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Microsoft Visual Studio Debug Console

```
Hello World!

C:\Labs\HelloWorld_NET5\HelloWorld_NET5\bin\Debug\net5.0\HelloWorld_NET5.exe
To automatically close the console when debugging stops, enable Tools->Option
le when debugging stops.
Press any key to close this window . . .
```

Hello World Explained

```
1   using System;
2
3   namespace HelloWorld_NET5
4   {
        0 references
5       internal class Program
6       {
            0 references
7           static void Main(string[] args)
8           {
9               Console.WriteLine("Hello World!");
10          }
11      }
12  }
13
```

using System

Use the System library to access useful functions such as the Console class's WriteLine method, without having to use its fully-qualified name

namespace HelloWorld_NET5

A namespace is used to logically arrange items such as classes and control the scope of their names in large projects

internal class Program

A class defines a type of object

static void Main(string[] args)

Main is a method which is a code block that contains a series of statements

In C#, every executed instruction is performed in the context of a method

Main returns nothing (**void**) and accepts an array (collection) of text strings as input

QA Top-level Statements

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

When you use top-level statements such as the .NET 6 console application above, the .NET compiler synthesises a **Program** class with a **Main** method and places all your top level statements in that Main method.

The compiler effectively converts the above code to code equivalent to that on the right-hand side.

```
1   using System;
2
3   namespace HelloWorld_NET5
4   {
        0 references
5       internal class Program
6       {
            0 references
7           static void Main(string[] args)
8           {
9               Console.WriteLine("Hello World!");
10          }
11      }
12  }
13
```

REFERRING TO NAMESPACES

Option A: Use the fully-qualified name

```
namespace HelloWorld_NET5
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("This will not work without the using directive");
            System.Console.WriteLine("This is fully qualified and will work");
        }
    }
}
```

Option B: Issue a 'using' directive

```
using System;

namespace HelloWorld_NET5
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("This will work because of the using directive");
            System.Console.WriteLine("This is fully qualified but verbose");
        }
    }
}
```

CREATING NAMESPACES

The intent of a Namespace is to uniquely identify the items it contains.

Below there are two classes, both called '*Car*'.

They can be disambiguated using their fully-qualified names of **Volkswagen.Car** and **Ferrari.Car**

```
namespace Volkswagen
{
    0 references
    public class Car
    {
    }
}
```

```
namespace Ferrari
{
    0 references
    public class Car
    {
    }
}
```

If only one class is required, you can issue a 'using' directive to import that namespace or use an alias.

```
using Volkswagen;
```

```
using v = Volkswagen;
```

NAMESPACE ALIASES

Namespaces can be nested.

```
namespace PC
{

    namespace MyCompany
    {
        namespace Project
        {
            1 reference
            public class MyClass { }
        }
    }
}
```

Use an alias to shorten a long namespace.

```
using Project = PC.MyCompany.Project;
```

FILE SCOPED NAMESPACE DECLARATIONS

File scoped namespace declarations are available from C# 10. They enable you to declare that all types in a file are in a single namespace.

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

You cannot include nested namespaces in a file scoped declaration.

IMPLICIT USING DIRECTIVES

The compiler automatically adds a set of using directives based on the project type. For console applications, the following directives are implicitly included in the application:

- `using System;`
- `using System.IO;`
- `using System.Collections.Generic;`
- `using System.Linq;`
- `using System.Net.Http;`
- `using System.Threading;`
- `using System.Threading.Tasks;`

GLOBAL USING DIRECTIVES

A global using directive imports a namespace for your whole application instead of a single file.

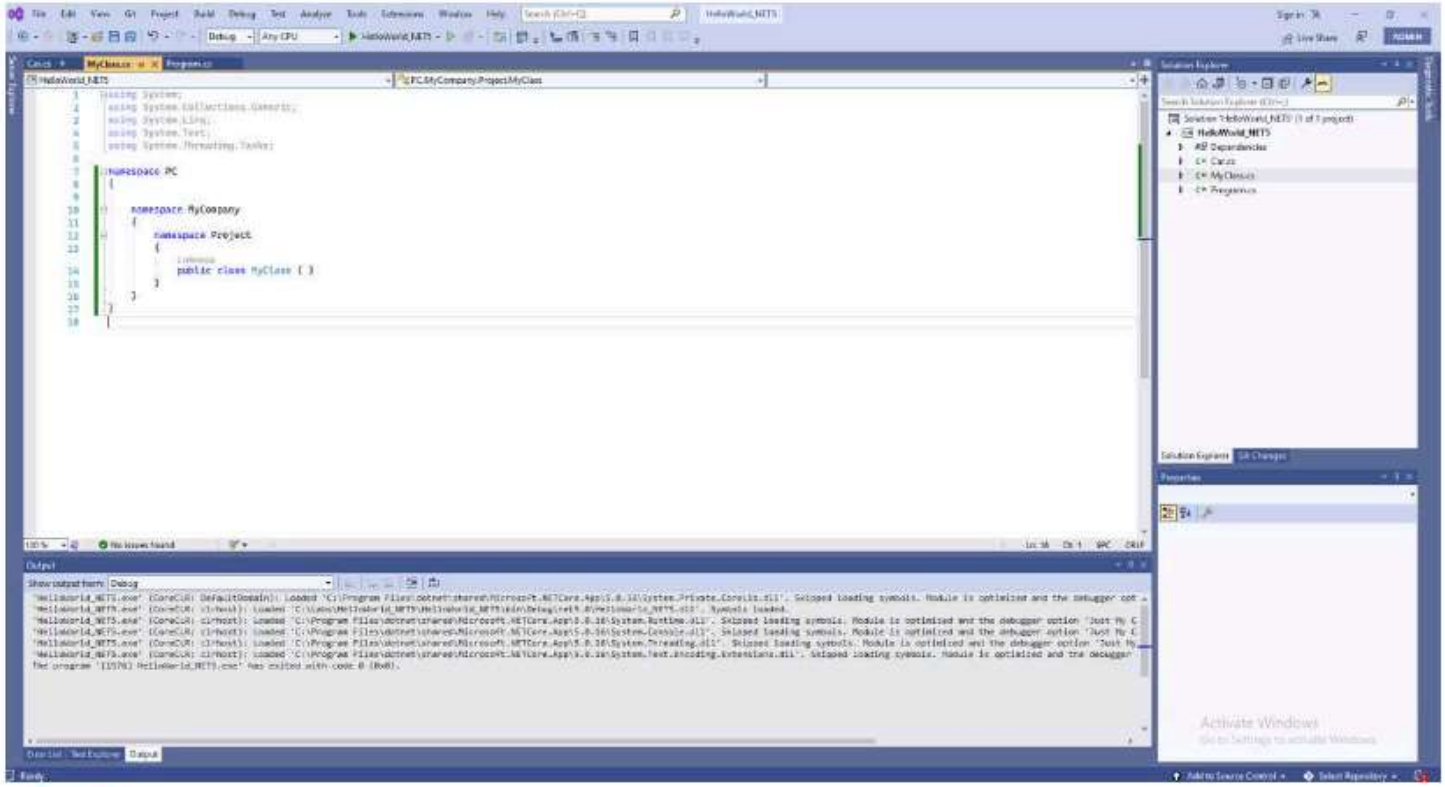
Either add a global using directive to the top of one of your code files:

```
global using PC.MyCompany.Project;
```

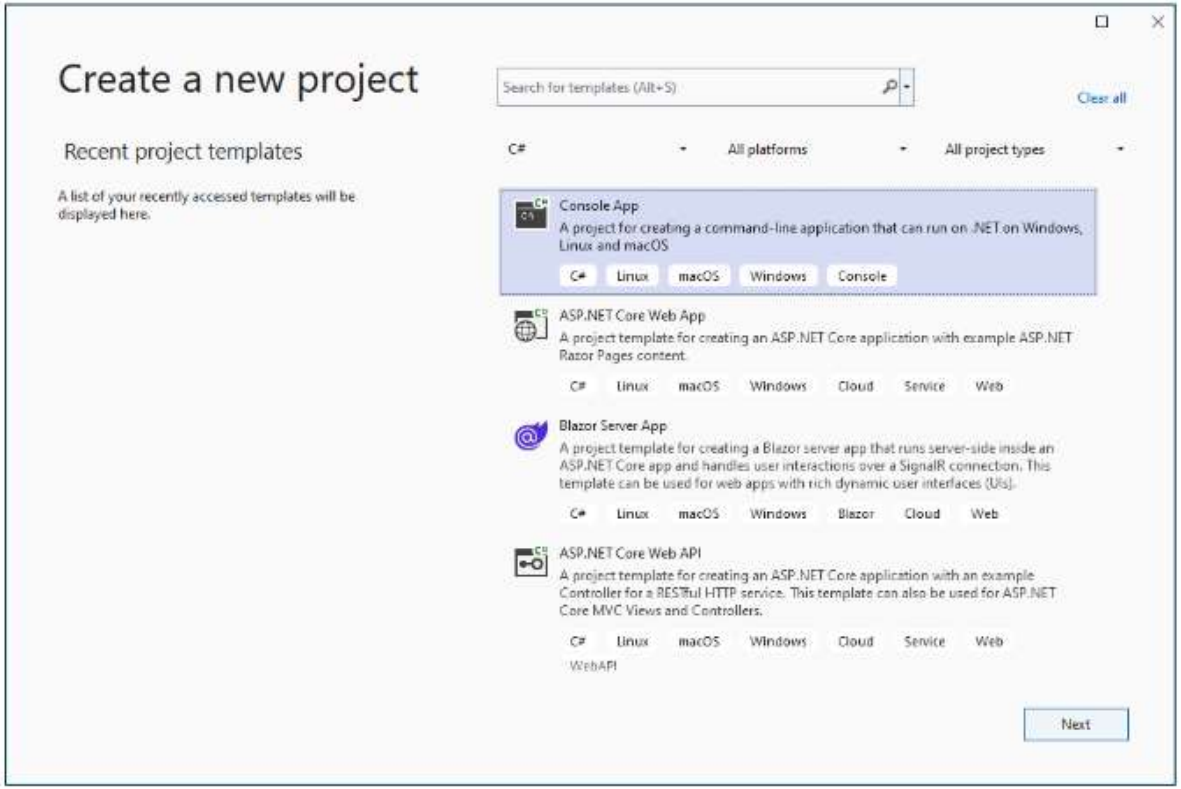
or, add a **<Using>** item to your project file:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

    <ItemGroup>
        <Using Include="PC.MyCompany.Project" />
    </ItemGroup>

</Project>
```

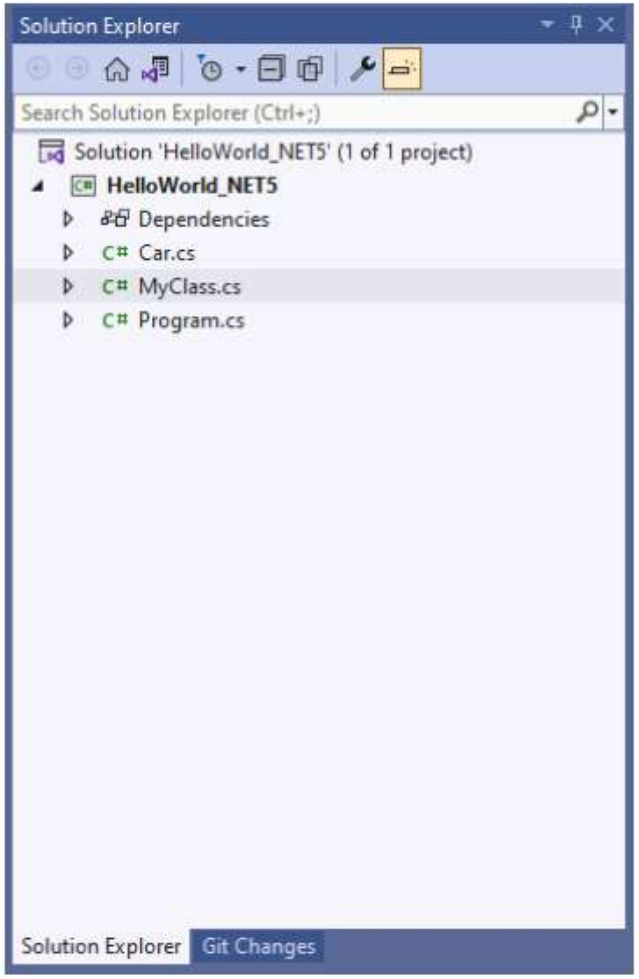
Visual Studio



Visual Studio Project Templates

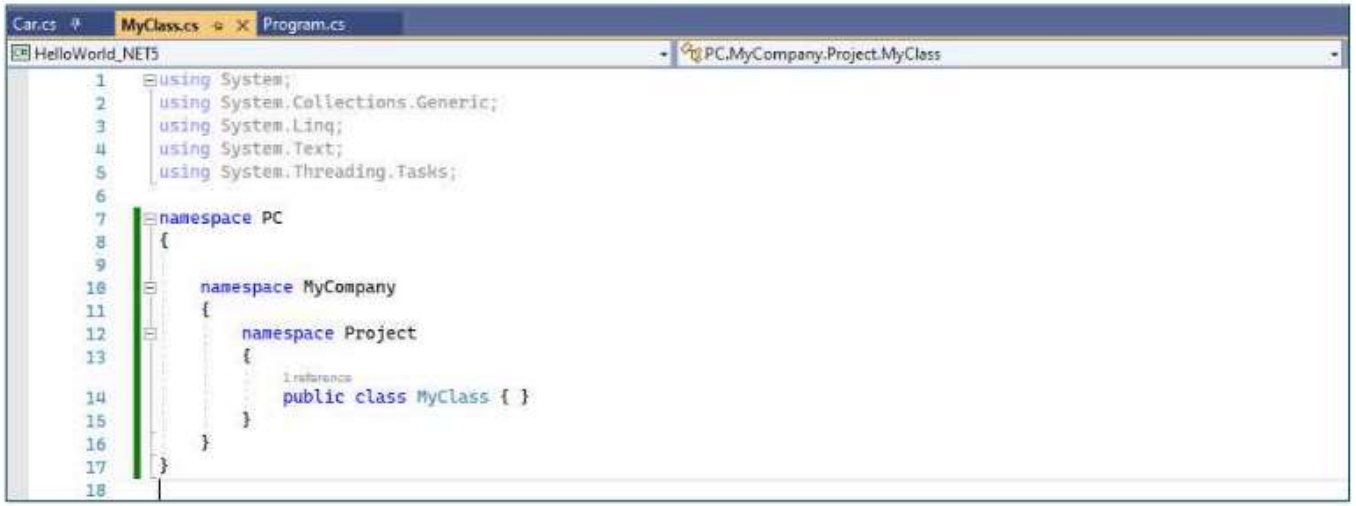


Visual Studio: Solution Explorer



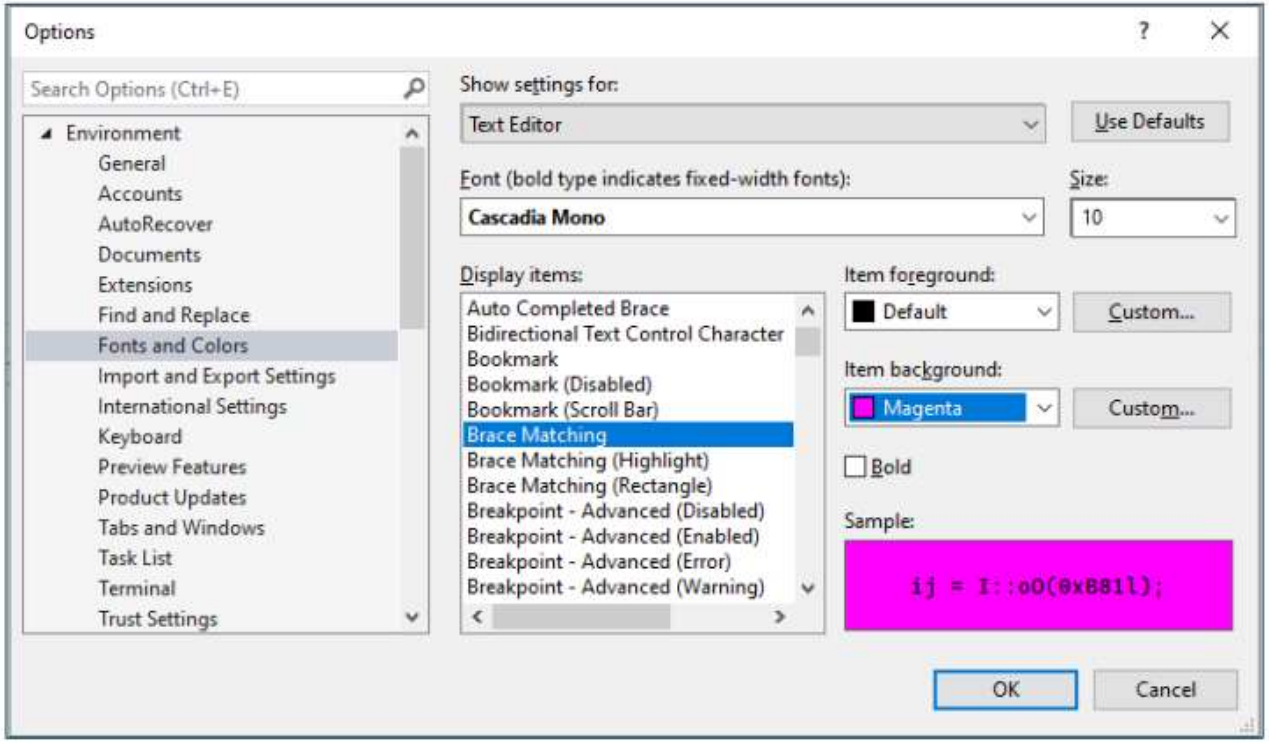
Visual Studio: Menu and Toolbar

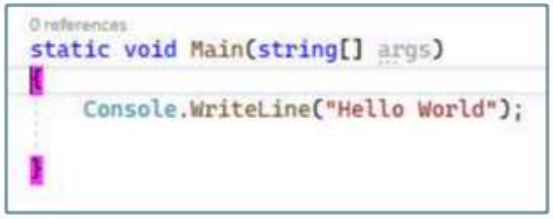
Visual Studio: Code Editors



Brace Matching

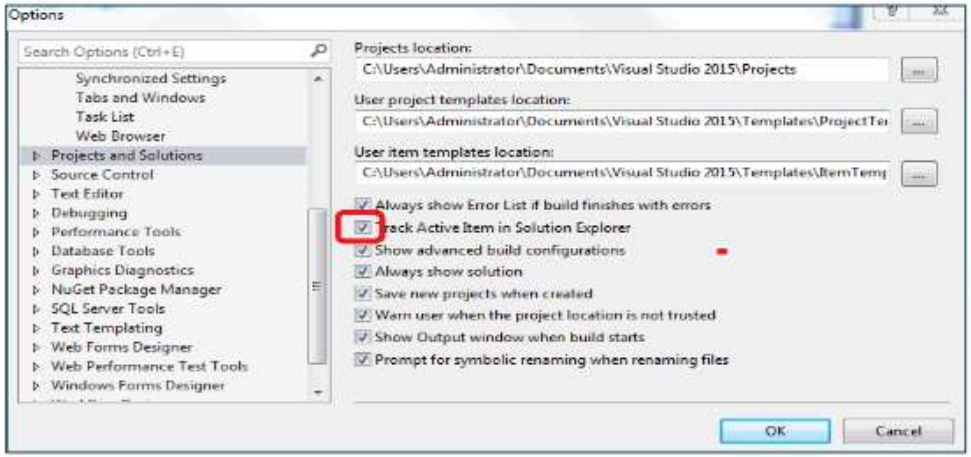
Tools -> Options -> Fonts and Colors

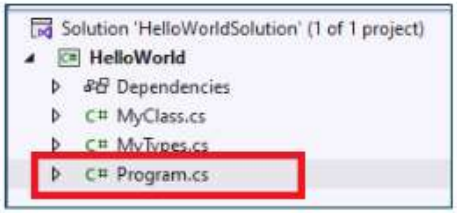




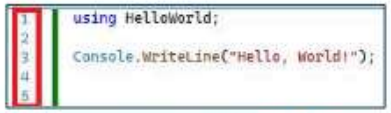
Tracking and Line Numbers

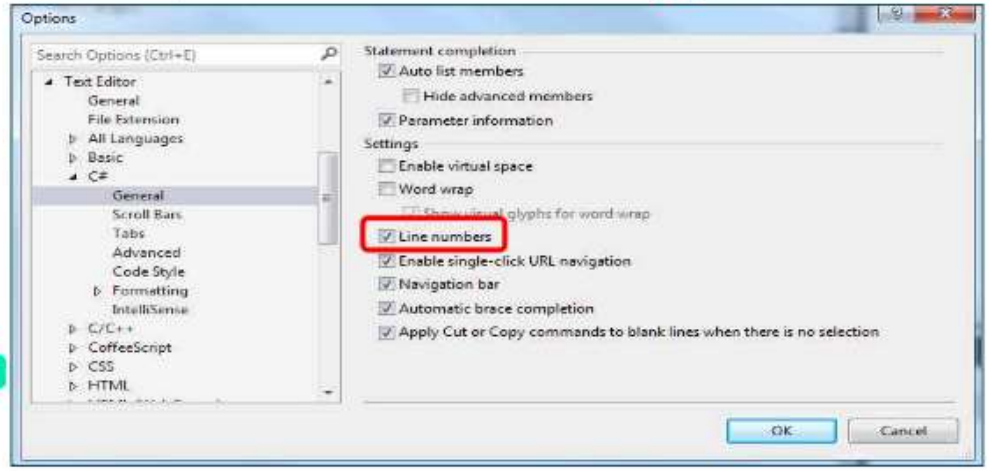
Tools -> Options -> Projects and Solutions





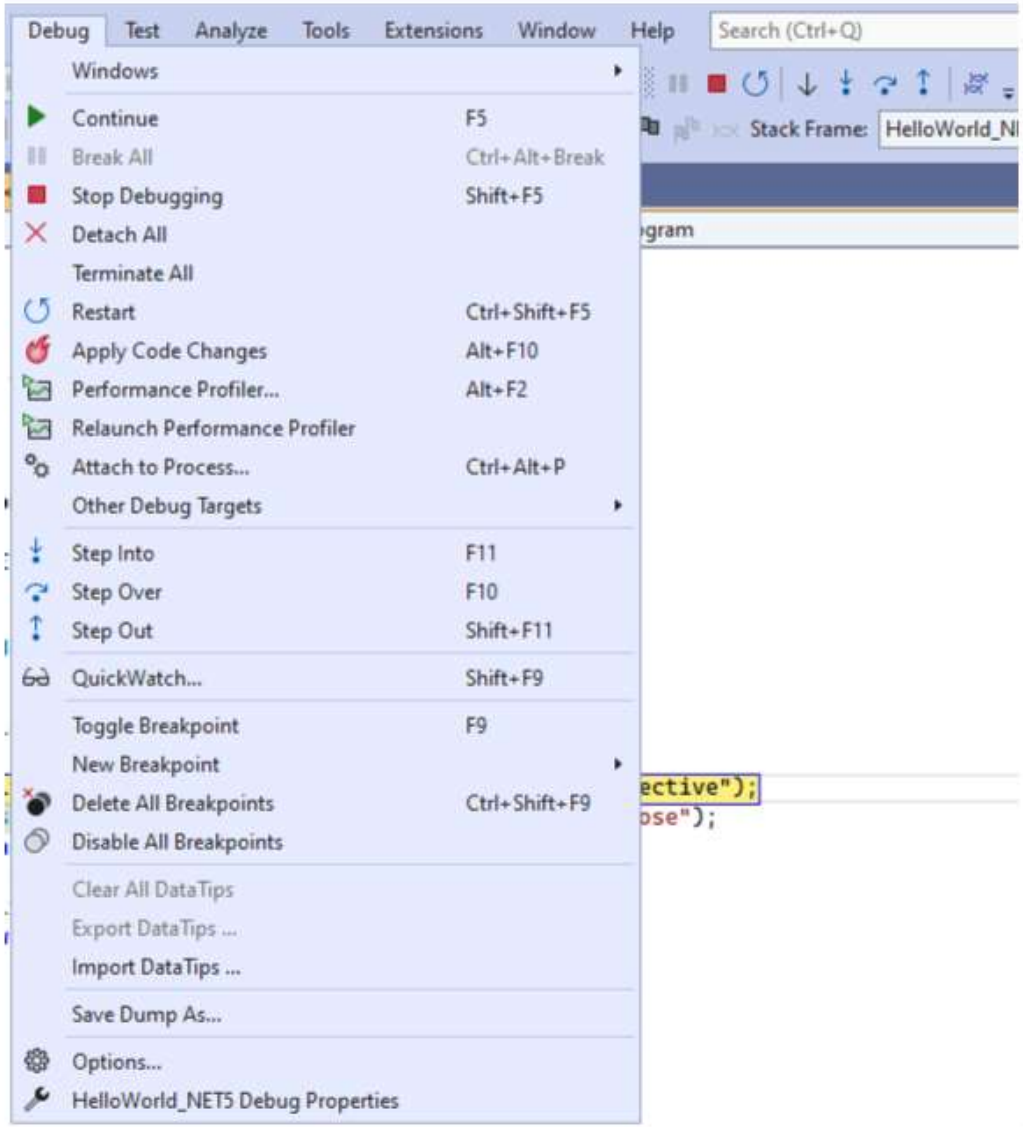
Tools -> Options -> Text Editor -> C#



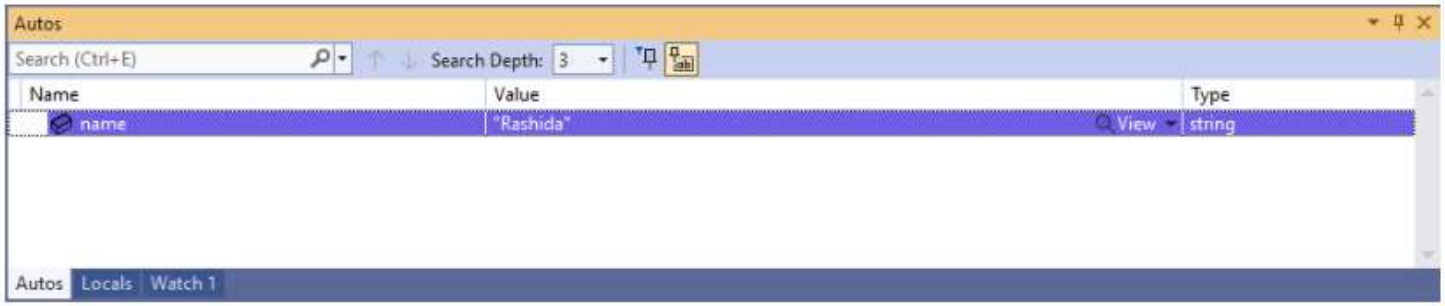


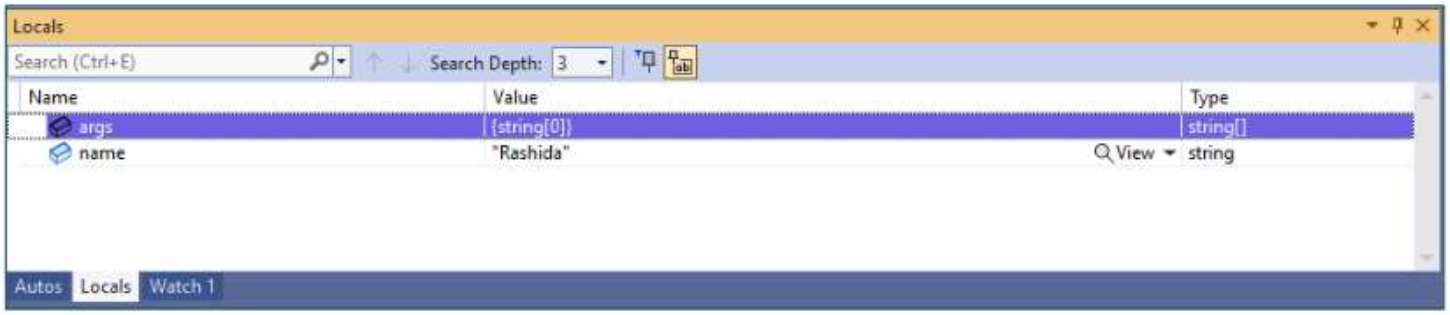
Debugging

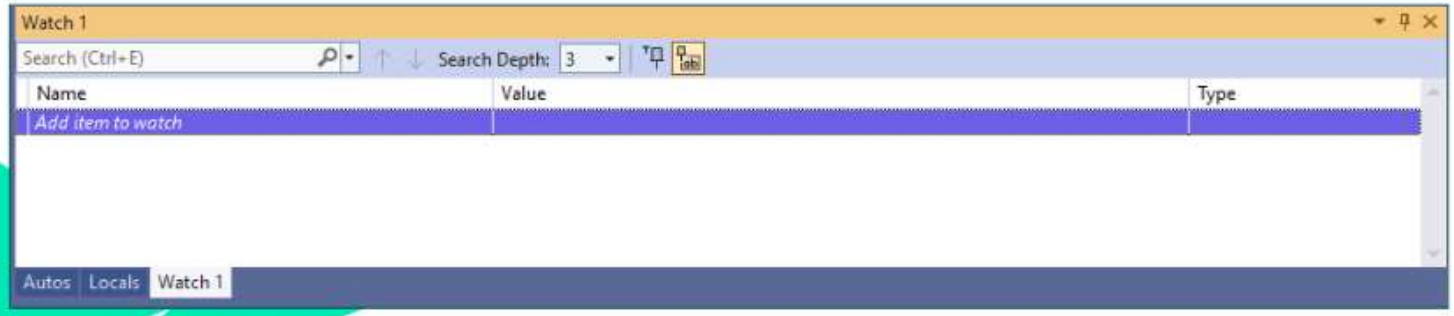
```
15      static void Main(string[] args)
16      {
17          Console.WriteLine("This will work because of the using directive");
18          System.Console.WriteLine("This is fully qualified but verbose");
19
20      }
```



Debugging Windows

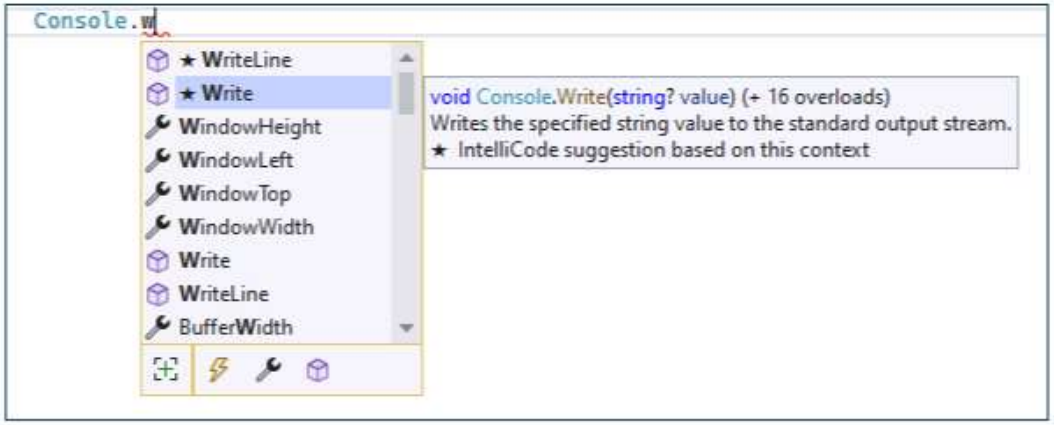






IntelliSense

Code snippets



List members

IntelliSense

```
Console.WriteLine("Intellisense");

}
```

void Console.WriteLine(string? value) (+ 17 overloads)
Writes the specified string value, followed by the current line terminator, to the standard output stream.

Exceptions:
System.IO.IOException

Quick info

Parameter info

```
Console.WriteLine(args);
```

▲ 2 of 18 ▼ void Console.WriteLine(bool value)
Writes the text representation of the specified Boolean value, followed by the current line terminator, to the standard output stream.
value: *The value to write.*

(parameter) string[] args
★ IntelliCode suggestion based on this context

★ args
#if
#region
AccessViolationException
Action
Action<>
Activator
AggregateException
AppContext

KEYBOARD SHORTCUTS

Shortcut (general)	Shortcut (C#)	Purpose
Ctrl + .		'Ctrl+dot' is for quick actions and refactoring. Visual Studio will help resolve by issuing using directives and generating code such as for classes or properties
Ctrl + R, Ctrl + R	F2	Rename an item.
Ctrl + K, Ctrl + D	Ctrl + E, D	Reformat document.
Ctrl + Alt + Spacebar		Toggle IntelliSense completion mode.
F12		Go to definition.
Ctrl + -		Go back to where you were.
F5		Start with debugging.
Ctrl + F5		Start without debugging.
F9		Toggle breakpoint on current line.
F10 / F11		Step over/into.

SUMMARY

- What is .NET?
- .NET compilation
- What is C#?
- Hello World in .NET 6 and .NET 5
- Hello World explained
- Namespaces
- Visual Studio
- Keyboard shortcuts

ACTIVITY: Exercise 2

Variables and Datatypes

OUTLINE

- Comments
- Identifiers
- Variables
- Built-in types
- Value types
- Reference types
- Creating variables: Value and reference types
- The var keyword
- Operators
- Parse and casting

COMMENTS

C# supports three styles of comments:

- Block comments
- In-line comments
- XML documentation comments

```
/*
This is a block comment ...
*/
```

```
  ... // This is a rest-of-line comment
```

```
/// <summary>
/// This is an XML-based comment
/// There are compiler tags
/// And .NET documentation convention tags
/// </summary>
```

IDENTIFIERS

Identifiers are used for the names of types e.g., classes and variables.

- Start with a letter of the alphabet (or an underscore)
  - Subsequent characters can include numeric digits and underscores
- C# is case sensitive
  - Therefore two identifiers can be differentiated by case alone
  - 'speed' & 'Speed' are different
- Follow convention for the casing of identifiers
  - camelCasing - local variables, parameters, & private fields
  - PascalCasing - types and everything they 'expose'

VARIABLES

- A variable is a symbolic name for an address in memory
- A variable is a value that can change
- All variables must be declared before they are used
- A variable must also be initialised before being read
- Local variables (variables declared within a method) have no initial value

```
int myAge;
bool answer = true;
string myName = "Michael Caine";
int i = 0, j = 1;

myAge = 21;
```

BUILT-IN DATATYPES

There are two main categories of types in C#:

- Value types
- Reference types

- A variable of a **value** type contains an instance of the type
- A variable of a **reference** type contains a *reference* to an instance of the type
- With **reference** types, it is possible for two variables to reference the same object and for operations on one variable to affect the object referenced by another variable
- With **value** types, each variable has its own copy of data so it is not possible for operations on one to affect the other

VALUE TYPES VERSUS REFERENCE TYPES

- A **value** type exists mainly for the data it holds rather than functionality
- A **reference** type is created for more complex types that need to exhibit a lot of functionality as well as store data
- The memory used by a **value** type is allocated in an area called the *stack*
- This memory is reclaimed as soon as the value type is no longer needed
- The memory used by a **reference** type is allocated in an area called the *managed heap*
- This memory is reclaimed by a service called the *Garbage Collector* - this happens at some future point when the object is no longer needed

BUILT-IN VALUE TYPES

C# has the following built-in value types, also known as *simple* types:

- Integral numeric types
- Floating-point numeric types
- bool
- char

A value type is one of the following:

- A **structure** type which encapsulates data and related functionality
- An **enumeration** type, which is a set of named constants

INTEGRAL NUMERIC TYPES

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
`nint`	Depends on platform	Signed 32-bit or 64-bit integer	System.IntPtr
`nuint`	Depends on platform	Unsigned 32-bit or 64-bit integer	System.UIntPtr

```
int x = 1234;
System.Int32 y = 1234;
```

FLOATING POINT NUMERIC TYPES

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	System.Single
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	System.Double
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	System.Decimal

```
double x = 12.34;
System.Double y = 12.34;
```

```
float f = 12345.67;
```

readonly struct System.Double
Represents a double-precision floating-point number.

CS0664: Literal of type double cannot be implicitly converted to type 'float'; use an 'F' suffix to create a literal of this type

```
float f = 12345.67F;
double d = 12345.67;
double dd = 12345.67D;
decimal m = 12345.67M;
```

```
Console.WriteLine(double.NaN);
Console.WriteLine(double.NegativeInfinity);
Console.WriteLine(double.PositiveInfinity);
```

BOOL DATATYPE

The bool type can hold only one of **two** values: *true* or *false:*

```
bool check = true;
bool isValid = false;
```

The bool type is the result of **comparison** and **equality** operators:

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(3.4 == 3.4);   // output: True
Console.WriteLine(3.4 != 3.4);   // output: False
Console.WriteLine(double.NaN < 7.0);   // output: False
```

CHAR DATATYPE

The char type is an alias for **System.Char**

It is a structure type that represents a Unicode 16-bit character.

A char can be specified with:

- A character literal in single quotes
- A Unicode escape sequence, which is **\u** followed by the 4-symbol hex character code
- A hexadecimal escape sequence, which is **\x** followed by the hex character code (leading zeros can be omitted)

```
char letter = 'p';
char copyrightUni = '\u00a9';
char copyrightHex = '\xa9';
char atSymbol = '\x40';
```

NULLABLE VALUE TYPES

A nullable value type **T?** represents all values of its underlying value type **T** and an additional *null* value.

For example, the nullable bool type can hold only one of **three** values: *true* or *false* or *null:*

```
bool? check = true;
bool? isValid = false;
bool? flag = null;
```

```
double? pi = 3.14159;
char? letter = 'p';

int luckyNumber = 7;
int? myLuckyNumber = luckyNumber;
```

STRUCTURE TYPES

A structure type (**struct**) is a value type that can encapsulate data and related functionality.

Use structs to design small data-centric types that provide little or no behaviour.

Microsoft recommend you define **immutable** structure types.

- Define the struct as *readonly*
- Define the data members as *readonly* or use an *init* accessor

STRUCT EXAMPLE

```
1 reference
public readonly struct Coords
{
    // Constructor
    // It is called when an instance is created and is used to initialize the instance
    0 references
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    // Auto-implemented properties: X and Y
    // A getter enables a property value to be read
    // An init-only setter assigns a value to the property only during object construction
    1 reference
    public double X { get; init; }
    1 reference
    public double Y { get; init; }

}
```

```
Coords c1 = new Coords(5, 7);
Coords c2 = new(5, 7);
Console.WriteLine(c1.X);
Console.WriteLine(c2.Y);
// c1.X = 10; // compile error because X is readonly
Console.WriteLine(c2.ToString());
```

ENUM TYPES

An enumeration type (**enum**) is a value type defined by a set of named constants of the underlying integral numeric type.

```
enum Level
{
    Low,    // 0
    Medium, // 1
    High    // 2
}
```

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

```
enum FiscalMonths
{

    April = 1,
    May = 2,
    June = 3,
    July = 4,
    August = 5,
    September = 6,
    October = 7,
    November = 8,
    December = 9,
    January = 10,
    February = 11,
    March = 12
}
```

```
FiscalMonths firstMonth = FiscalMonths.April;
Console.WriteLine(firstMonth);
```

REFERENCE TYPES

C# has the following built-in reference types:

- object
- string
- dynamic

The following keywords are used to declare reference types:

- class
- interface
- delegate
- record

OBJECT

- **System.Object** is the ultimate base class of all .NET classes
- Inheritance from Object is implicit
- Every method defined in Object is available in all objects in the system
- Derived classes can override some of these methods:
  - **Equals**: Supports comparisons between objects
  - **Finalize**: Performs clean-up operations before an object is reclaimed
  - **GetHashCode**: Generates a number corresponding to the value of the object to support the use of a hash table
  - **ToString**: Manufactures a human-readable text string that describes an instance of the class
- When a variable of a value type is converted to object, it is said to be *boxed*
- When a variable of type object is converted to a value type, it is said to be *unboxed*

STRING

- A **string** type represents a sequence of zero or more Unicode characters
- string is an alias for **System.String**
- String is a reference type but the equality operators **==** and **!=** are defined to compare the *values* of string objects rather than their *references*
- The **+** operator concatenates strings
- Strings are immutable

```
string greeting = "good morning";
string message = "good ";
message = message + "morning";
Console.WriteLine(greeting == message);
Console.WriteLine(object.ReferenceEquals(greeting, message));
```

```
True
False
```

STRING LITERALS

- A string literal is anything between a pair of double quotes **“ “**
- A *verbatim* string allows special characters to be included in a string literal and is prefixed with **@**
- Use the **string.format** method to format strings or use *interpolated* strings
- Interpolated strings are prefixed with **$**

```
string literalString = "This is a string literal";
string verbatimString = @"This is a verbatim string and can contain special characters such as \ and \n";
string newLine = "\n";
string name = "Bart";
int age = 8;
string formatString = string.Format("My name is {0} and I am {1} years old", name, age.ToString());
string interpolatedString = $"My name is {name} and I am {age} years old";
```

```
This is a string literal
This is a verbatim string and can contain special characters such as \ and \n


My name is Bart and I am 8 years old
My name is Bart and I am 8 years old
```

CLASS

```
public class Car : Object
{
    const int NumWheels = 4;
    int cylinders;

    int Cylinders {
        get { return cylinders; }
        set { cylinders = value;}
    }
    public void Start()
    {
        // code in here
    }
    public event EventHandler AlarmDisabled;
}
```

All classes inherit from System.Object

A constant, immutable field

A field (a class-level variable)

A property with a getter and a setter

A method

An event

OBJECT INSTANTIATION

Class definition with a method:

```
class Car
{
    2 references
    public void Register(string name, string address, string postCode, string country)
    {
        // store this in the DVLA database in Swansea
    }
}
```

Instantiate objects based on the class and invoke the method:

```
Car julieCar = new Car();
julieCar.Register("Julie Dooley", "1 Main Street", "CH12 9DL", "UK");


Car lisaCar = new Car();
lisaCar.Register("Lisa Simpson", "742 Evergreen Terrace", "97394", "USA");
```

CREATING VARIABLES: SIMPLE TYPES

Simple types are declared as variables using the following pseudo-code:

datatype variableName = value;

```
int a = 6; // literal assignment
float b = 7.5F; // literal assignment with a suffix
decimal c = 9.99M; // M or m suffix denotes a decimal literal
bool d = false; // assign special value false (or true) to a bool
char e = 'a'; // single quotes for char literals
char? h = null; // ? for a nulable variable
string f = "hello"; // string is a reference type but gets created like a value type
string? g = null; // nullable string
```

A value type holds its data within its own memory space.

The **var** keyword can be used to implicitly type a variable.

```
int x = 3;
var y = 9;

Console.WriteLine(x.GetType()); // System.Int32
Console.WriteLine(y.GetType()); // System.Int32
```

CREATING VARIABLES: NON-SIMPLE VALUE TYPES

Non-simple value types are declared as variables using the '**new**' keyword for *struct* types and typically a value from the enumeration for an *enum* type:

```
Level level = Level.High;// enum variable is constrained to the defined values
Coords c1 = new Coords(b, a);// a struct can be instantiated using 'new'
Console.WriteLine(c1.GetType());// GetType method gets the type of the current instance
var c2 = new Coords(b, a); // the type can be inferred by the compiler when using 'var'
Console.WriteLine(c2.GetType());
Coords c3 = new(b, a); // C# 9 target-typed constructor invocation syntax
Console.WriteLine(c3.GetType()); // c1, c2 and c3 are all of type Coords
```

C# 9 introduced a shorter syntax for invoking a constructor called *target-typed invocation*.

```
Coords c3 = new(b, a); // C# 9 target-typed constructor invocation syntax
```

CREATING VARIABLES: REFERENCE TYPES

Reference types are instantiated as variables using the '**new**' keyword:

```
Car car1 = new Car(); // reference types are instantiated using 'new'
var car2 = new Car(); // the type can be inferred by the compiler
Car car3 = new(); // C# 9 target-typed constructor invocation syntax
Console.WriteLine(car3.GetType());
// 'if' is a conditional statement with the condition to be evaluated in brackets
if (car3 is Car) {
    // The 'is' operator checks if the result of an expression is compatible with a given type
    Console.WriteLine(nameof(car3) + " is a Car instance");
    // nameof produces the name of a variable, type, or member as a string constant
}
```

Target-typed constructor invocation can be used with any value or reference type that has a constructor.

A reference type holds a pointer (reference) within its own memory space that points to another memory location that holds the real data.

VARIABLE SCOPE

```
0 references
class ScopeOfVariables
{
    int visibleToEntireType; // variables at class scope (Fields)

    0 references
    int MethodA()
    {
        visibleToEntireType = 42;
        int onlyVisibleWithinMethod = 13;
        // block created with braces
        {
            int onlyVisibleWithinBlock = 5;
            return onlyVisibleWithinBlock + onlyVisibleWithinMethod;
        }

        onlyVisibleWithinBlock = 7; //not visible outside of the block (compile error)
    }

    0 references
    void MethodB()
    {
        visibleToEntireType = 21;
        onlyVisibleWithinMethod = 50; //not visible outside of MethodA (compile error)
    }
}
```

ACCESS MODIFIERS

Access Modifier	Description
public	The type or member can be accessed anywhere
private	The type or member can be accessed only by code in the same class or struct
protected	The type or member can be accessed only by code in the same class, or in a class that is derived from that class
internal	The type or member can be accessed by any code in the same assembly
protected internal	The type or member can be accessed by any code in the same assembly, or within a derived class in another assembly
private protected	The type or member can be accessed by types derived from the class that are declared within its containing assembly

OPERATORS

C# provides many operators, such as:

- Arithmetic operators
- Comparison operators
- Boolean logical operators
- Bitwise and shift operators
- Equality operators

ARITHMETIC OPERATORS

- C# provides standard arithmetic operators:

++	**increment by 1**
--	**decrement by 1**
+	**plus**
-	**minus (numeric negation)**
+	**addition**
-	**subtraction**
*	**multiplication**
/	**division**
%	**modulo division (remainder)**

- Operators can use compound syntax:
- x = x + 5;
- x += 5;
- The above statements produce identical results

ARITHMETIC OPERATOR EXAMPLES:

++

--

```
// post-fix increment operator
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4

// pre-fix increment operator
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

```
// post-fix decrement operator
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2

// pre-fix decrement operator
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

ARITHMETIC OPERATOR EXAMPLES: UNARY + -

```
// unary plus and minus operators
Console.WriteLine(+8);      // output: 8

Console.WriteLine(-8);      // output: -8
Console.WriteLine(-(-8));   // output: 8

uint a = 9;
var b = -a;
Console.WriteLine(b);              // output: -9
Console.WriteLine(b.GetType());    // output: System.Int64

Console.WriteLine(-double.NaN);    // output: NaN
```

ARITHMETIC OPERATOR EXAMPLES: ADD + SUBTRACT -

```
// addition operator
Console.WriteLine(5 + 4);        // output: 9
Console.WriteLine(5 + 4.3);      // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3

// subtraction operator
Console.WriteLine(47 - 3);       // output: 44
Console.WriteLine(5 - 4.3);      // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

ARITHMETIC OPERATOR EXAMPLES: MULTIPLY * DIVIDE /

```
// multiply operator
Console.WriteLine(5 * 2);          // output: 10
Console.WriteLine(0.5 * 2.5);      // output: 1.25
Console.WriteLine(0.1m * 23.4m);   // output: 2.34

// division operator
// For integer operands, the result is an int type rounded towards zero
Console.WriteLine(13 / 5);     // output: 2
Console.WriteLine(-13 / 5);    // output: -2
Console.WriteLine(13 / -5);    // output: -2
Console.WriteLine(-13 / -5);   // output: 2

// int / double
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6

Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

ARITHMETIC OPERATOR EXAMPLES: REMAINDER /

```
// modulus / remainder operator
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1

Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

ARITHMETIC OPERATOR PRECEDENCE

- What are the values of *d*?

```
double a = 3;
double b = 5;
double c = 7;
double d = a + b * c;
```

```
double a = 3;
double b = 5;
double c = 7;
double d = (a + b) * c;
```

COMPARISON OPERATORS

- C# provides standard comparison operators:

>	**greater than**
>=	**greater than or equal to**
<	**less than**
<=	**less than or equal to**

Comparison operators can be used to compare:

- All integral types
- All floating-point types
- char types based on the underlying character codes
- Enums based on the underlying integral values

COMPARISON OPERATOR EXAMPLES:

```
// less than <
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True


// greater than >
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False

// less than equals <=  // greater than equals >=
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True

// NaN
Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False

// char comparison
char a = 'a';
char b = 'b';
Console.WriteLine(a > b);   // output: False
Console.WriteLine(a <= b);  // output: True

// enum comparison
Level low = Level.Low;
Level high = Level.High;
Console.WriteLine(low > high);  // output: False
Console.WriteLine(low < high);  // output: True
```

BOOLEAN LOGICAL OPERATORS

- C# provides standard Boolean logic operators:

!	**Logical negation**
&	**Logical AND**
|	**Logical OR**
^	**Logical exclusive OR**
&&	**Conditional Logic AND**
||	**Conditional Logic OR**

- Operators &, | and ^ perform bitwise operations when the operands are integral numeric types

BOOLEAN LOGICAL OPERATOR EXAMPLES

```
// logical negation
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False


// logical exclusive OR
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False

// logical OR
Console.WriteLine(true | true); // output: True
Console.WriteLine(true | false); // output: True
Console.WriteLine(false | true); // output: True
Console.WriteLine(false | false); // output: False

// logical AND
Console.WriteLine(true & true); // output: True
Console.WriteLine(true & false); // output: False
Console.WriteLine(false & true); // output: False
Console.WriteLine(false & false); // output: False
```

BOOLEAN LOGICAL OPERATOR EXAMPLES: & VERSUS &&

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

// Logical AND: method SecondOperand is always called
bool a = false & SecondOperand();
Console.WriteLine($"{nameof(a)} is {a}");
// Output:
// Second operand is evaluated.
// a is False

bool b = true & SecondOperand();
Console.WriteLine($"{nameof(b)} is {b}");
// Output:
// Second operand is evaluated.
// b is True

// Conditional Logical AND: if first operand is false, method SecondOperand is not invoked
// This is short-circuit evaluation
bool c = false && SecondOperand();
Console.WriteLine($"{nameof(c)} is {c}");
// Output:
// c is False

bool d = true && SecondOperand();
Console.WriteLine($"{nameof(d)} is {d}");
// Output:
// Second operand is evaluated.
// d is True
```

BOOLEAN LOGICAL OPERATOR EXAMPLES: | VERSUS ||

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

// Logical OR: method SecondOperand is always called
bool a = true | SecondOperand();
Console.WriteLine($"{nameof(a)} is {a}");
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine($"{nameof(b)} is {b}");
// Output:
// Second operand is evaluated.
// True

// Conditional Logical OR: if first operand is true, method SecondOperand is not invoked
// This is short-circuit evaluation
bool c = true || SecondOperand();
Console.WriteLine($"{nameof(c)} is {c}");
// Output:
// True

bool d = false || SecondOperand();
Console.WriteLine($"{nameof(d)} is {d}");
// Output:
// Second operand is evaluated.
// True
```

BITWISE AND SHIFT OPERATORS

- C# provides standard bitwise and shift operators:

~	**Bitwise complement**
<<	**Left shift**
>>	**Right shift**
&	**Logical AND**
|	**Logical OR**
^	**Logical exclusive OR**

Bitwise operators operate on the integral types at the binary level:

- **~** reverses all bits
- **<<** shifts the bits *n* places to the left
- **>>** shifts the bits *n* places to the right

BITWISE AND SHIFT OPERATOR: EXAMPLES

~ << >>

```
// bitwise complement
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011

// left shift
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000

// right shift
uint i = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(i, toBase: 2),4}");

uint j = i >> 2;
Console.WriteLine($"After:  {Convert.ToString(j, toBase: 2),4}");
// Output:
// Before: 1001
// After:    10
```

BITWISE AND SHIFT OPERATOR: EXAMPLES & ^ |

```
// logical AND &
uint d = 0b_1111_1000;
uint e = 0b_1001_1101;
uint f = d & e;
Console.WriteLine(Convert.ToString(f, toBase: 2));
// Output:
// 10011000

// logical Exclusive OR ^
uint l = 0b_1111_1000;
uint m = 0b_0001_1100;
uint n = l ^ m;
Console.WriteLine(Convert.ToString(n, toBase: 2));
// Output:
// 11100100

// logical OR |
uint o = 0b_1010_0000;
uint p = 0b_1001_0001;
uint q = o | p;
Console.WriteLine(Convert.ToString(q, toBase: 2));
// Output:
// 10110001
```

EQUALITY OPERATORS

- C# provides two equality operators:

==	**Equality operator**
!=	**Inequality operator**

- The equality operator returns *true* if both operands are equal, *false* otherwise
- Value types are equal if their *values* are equal
- Reference types are equal if they *refer* to the same object

EQUALITY OPERATOR EXAMPLES: ==

```
// value type equality
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True


// reference type equality
var car1 = new Car();
var car2 = new Car();
var car3 = car1;
Console.WriteLine(car1 == car2);  // output: False
Console.WriteLine(car1 == car3);  // output: True

// string equality
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

EQUALITY OPERATOR EXAMPLES: !=

```
// value type inequality
int c = 1 + 2 + 3 + 4;
int d = 6;
Console.WriteLine(c != d);  // output: True

// string inequality
string s4 = "Hello";
string s5 = "Hello";
Console.WriteLine(s4 != s5);  // output: False

// reference type inequality
object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

PARSE

The primitive value types are all able to read in a string and convert to their type using the type's **.Parse()** method.

```
int i           = int.Parse("42");
decimal dec     = decimal.Parse("42.0");
double dbl      = double.Parse("123.45");
```

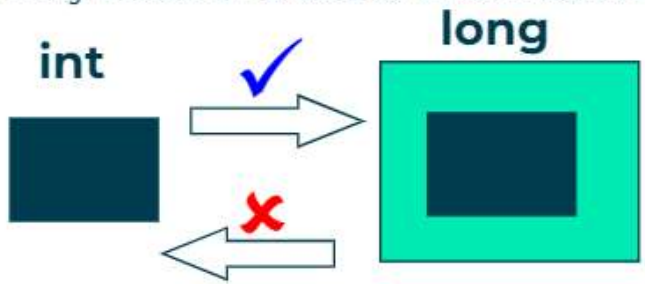
INTEGER ARITHMETIC & CONVERSIONS

Arithmetic works on all numeric types

- Result type is the widest of operand types, but `int` if both narrower
- Cannot mix `decimal` and `float`, or `ulong` and any signed type

You may need to cast the result

- *implicit* casts silently widen to a larger type e.g. `int` to `long`
- *explicit* casting is necessary to do the reverse operation, because it may result in a loss of information



INTEGRAL CONVERSIONS EXAMPLE

```
int i = 4;
long l = i; // implicit cast (widening)
l += 3_000_000; // 3 million
//i = l; // implicit cast (narrowing so compile error)
i = (int)l; // explicit cast required to acknowledge potential data loss
Console.WriteLine(i);
// output: 3000004

int j = 4;
long k = j; // implicit cast (widening)
k += 3_000_000_000; // int has a max value of 2_147_483_647
//j = k; // implicit cast (narrowing so compile error)
j = (int)k; // explicit cast required to acknowledge potential data loss
Console.WriteLine(j); // integer overflow results in an incorrect value
// output: -1294967292
```

CASTING: REFERENCE TYPE EXAMPLE

Forcing the compiler to convert from one datatype to another:

- This won't compile :

```
static void Main(string[] args) {
    Dog d = GetMammal();
    d.Bark();
}
static Mammal GetMammal() {
    return new Dog();
}
```

I can see that it returns a Dog but the compiler is not allowed to use this information

- It needs a cast to compile

```
Dog d = (Dog)GetMammal();
d.Bark();
```

- Or this

```
((Dog)GetMammal()).Bark();
```

ACTIVITY: Exercise 3

SUMMARY

- Identifiers
- Comments
- Variables
- Built-in types
- Value types
- Reference types
- Creating variables: Value and reference types
- The var keyword
- Operators
- Parse and casting

Conditionals

OUTLINE

- Code blocks
- Select statements
- The if statement
- The switch statement
- The switch expression
- Switch case guards
- The ternary conditional operator
- Null-coalescing operators
- Null-conditional operators

CODE BLOCKS

All control statements in C# operate on the one and only next statement

- Braces group many statements into one code block
- The two statements below are identical, but the first is considered best practice. Why?

```
if (condition) {
    count++;
}
```

```
if (condition)
    count++;
```

- Braces can start at the end of the control line or on the next line

```
if (condition)
{
    count++;
}
```

SELECT STATEMENTS

There are two kinds of select or conditional statements in C#:

- The **if** statement
- The **switch** statement

IF STATEMENT

```
if (booleanExpression)
{
  statement(s);
}
```

```
if (booleanExpression)
{
  statement(s);
}
else          // Optional
{
  statement(s);
}
```

```
if (booleanExpression)
{
  statement(s);
}
else if (booleanExpression)  // Optional
{
  statement(s);
}

else          // Optional
{
  statement(s);
}
```

IF STATEMENT EXAMPLES

```
int age = 16;
int votingAge = 18;


if (age >= votingAge)
{
    Console.WriteLine("You are eligible to vote in the election");
}


if (age >= votingAge)
{
    Console.WriteLine("You are eligible to vote in the election");
}
else
{
    Console.WriteLine("You are not eligible to vote in the election");
}
```

```
int age = 16;
var country = Country.Scotland;

if ((country == Country.Wales || country == Country.Scotland) && age >= 16)
{
    Console.WriteLine("You are eligible to vote in the Welsh / Scottish election");
}
else if ((country == Country.England || country == Country.NorthernIreland) && age >= 18)
{
    Console.WriteLine("You are eligible to vote in the English / Northern Irish election");
}
else
{
    Console.WriteLine("You are not eligible to vote in the election");
}
```

SWITCH STATEMENT

```
switch (iMonth) {
  case 1:
    daysInMonth = 28;
    break;
  case 3: case 5: case 8: // etc
    daysInMonth = 30;
    break;
  default:
    daysInMonth = 31;
    break;
}
```

```
switch (month) {
  case "February":
    daysInMonth = 28;
    break;
  case "April": case "June": //etc
    daysInMonth = 30;
    break;
  default:
    daysInMonth = 31;
    break;
}
```

SWITCH STATEMENT EXAMPLE: ENUM

Switch statements are often used with enums:

```
enum Country
{
    England,
    NorthernIreland,
    Scotland,
    Wales
}
```

```
var country = Country.Scotland;
string parliamentName;
switch (country)
{
    case Country.England:
        parliamentName = "Westminster";
        break;
    case Country.NorthernIreland:
        parliamentName = "Northern Ireland Assembly";
        break;
    case Country.Scotland:
        parliamentName = "Holyrood";
        break;
    case Country.Wales:
        parliamentName = "Senedd Cymru";
        break;
    default:
        parliamentName = "Unknown Parliament";
        break;
}
Console.WriteLine(parliamentName);
```

SWITCH STATEMENT EXAMPLE

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

SWITCH EXPRESSION

- A **switch expression** is a more lightweight syntax than a *switch statement*
- They do not use **case, break,** or **default** keywords
- They use patterns and expressions separated by an **=>** arrow token
- The underscore **_** is a *discard pattern* which matches any expression, including **null**

```
var operation = 3;

var result = operation switch
{
    1 => "Option 1",
    2 => "Option 2",
    3 => "Option 3",
    4 => "Option 4",
    _ => "Default option"
};

Console.WriteLine(result);
```

SWITCH EXPRESSION EXAMPLE

```
static string ToParliamentName(Country country) => country switch
{
    Country.England => "Westminster",
    Country.Scotland => "Holyrood",
    Country.Wales => "Senedd Cymru",
    Country.NorthernIreland => "Northern Ireland Assembly",
    _ => "Unknown Parliament"
};

var country = Country.England;
Console.WriteLine($"Country name is {country}");
Console.WriteLine($"The parliament name is {ToParliamentName(country)}");
// Output:
// Country name is England
// The parliament name is Westminster
```

`Country.England => “Westminster”,`

- Is called an *expression arm*

SWITCH STATEMENT VERSUS SWITCH EXPRESSION

```
var country = Country.Scotland;
string parliamentName;
switch (country)
{
    case Country.England:
        parliamentName = "Westminster";
        break;
    case Country.NorthernIreland:
        parliamentName = "Northern Ireland Assembly";
        break;
    case Country.Scotland:
        parliamentName = "Holyrood";
        break;
    case Country.Wales:
        parliamentName = "Senedd Cymru";
        break;
    default:
        parliamentName = "Unknown Parliament";
        break;
}
Console.WriteLine(parliamentName);
```

```
var country = Country.Scotland;
string parliamentName = country switch
{
    Country.England => "Westminster",
    Country.NorthernIreland => "Northern Ireland Assembly",
    Country.Scotland => "Holyrood",
    Country.Wales => "Senedd Cymru",
    _ => "Unknown Parliament"
};
Console.WriteLine($"The parliament name is {parliamentName}");
```

SWITCH CASE GUARDS

Expression arms contain:

- A *pattern*
- An optional **case guard**
- The **=>** arrow token
- An *expression*

A **case guard** is an additional condition that must be satisfied together with the matched pattern.

A case guard must be a Boolean expression.

Specify the case guard after the **when** keyword that follows a pattern.

SWITCH STATEMENT CASE GUARD EXAMPLE

The case guard is:

when a == b

```
DisplayMeasurements(7, 6);  // Output: First measurement is 7, second measurement is 6.
DisplayMeasurements(8, 8);  // Output: Both measurements are valid and equal to 8.
DisplayMeasurements(5, -3);  // Output: One or both measurements are not valid.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case ( > 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case ( > 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

SWITCH EXPRESSION CASE GUARD EXAMPLE

```
16 references
public readonly struct Point
{
    7 references
    public Point(int x, int y) => (X, Y) = (x, y);
    // a constructor expression-bodied method
    // take x and assign it to X
    // take y and assign it to Y

    8 references
    public int X { get; }
    // readonly property X
    8 references
    public int Y { get; }
    // readonly property Y
}
```

```
static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 } => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y } => new Point(2 * x, 2 * y),
};

Point p = new Point(2, 4);
Point transformedP = Transform(p);
Console.WriteLine($"X is {transformedP.X} and Y is {transformedP.Y}");
// Output:
// X is 6 and Y is 4

Point p2 = new Point(7, 3);
Point transformedP2 = Transform(p2);
Console.WriteLine($"X is {transformedP2.X} and Y is {transformedP2.Y}");
// Output:
// X is 4 and Y is 3

Point p3 = new Point(6, 6);
Point transformedP3 = Transform(p3);
Console.WriteLine($"X is {transformedP3.X} and Y is {transformedP3.Y}");
// Output:
// X is 12 and Y is 12
```

TERNARY CONDITIONAL OPERATOR ?:

The *ternary conditional operator* **?:** is a short-hand alternative to an **if** statement with two branches:

```
var coin = Coin.Heads;

// IF statement syntax
string status;
if (coin == Coin.Heads)
{
    status = "won";
}
else
{
    status = "lost";
}
Console.WriteLine($"You {status} the toss");

// Ternary operator ?:
string status2 = (coin == Coin.Heads) ? "won" : "lost";
Console.WriteLine($"You {status2} the toss");
```

```
enum Coin
{
    Heads,
    Tails
}
```

NULL OPERATORS

??

??=

?.

C# provides various operators that evaluate *nulls* instead of Boolean expressions

- Null-coalescing operator **??**
- Null-coalescing assignment operator **??=**
- Null-conditional operator **?.**

NULL COALESCING OPERATORS

??

??=

- The **null-coalescing operator** returns the value of its left-hand operand if it isn't null
- Otherwise it evaluates the right-hand operand
- If the left-hand operand is non-null, the right-hand operand is not evaluated

- The **null-coalescing assignment operator** assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null
- If the left-hand operand is non-null, the right-hand operand is not evaluated

NULL CONDITIONAL OPERATOR ?.

- The **null-conditional operator** applies a member access operation to its operand only if that operand evaluates to non-null
- Otherwise, it returns null
- Often combined with the null-coalescing operator to return something other than null

NULL OPERATOR EXAMPLES

```
// null-coalescing operator
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1

// null-coalescing assignment operator
string address = "1 Main Street ";
string? country = null;
string addressAndCountry = (address + (country ??= "UK"));
Console.WriteLine(addressAndCountry);

// null-conditional operator accessing Length member
// combined with null-coalescing operator
int? countryLength = country?.Length ?? 0;
Console.WriteLine(countryLength);// 2

string? postcode = null;
int? postcodeLength = postcode?.Length ?? 0;
Console.WriteLine(postcodeLength);// 0
```

SUMMARY

- Code blocks
- Select statements
- The if statement
- The switch statement
- The switch expression
- Switch case guards
- The ternary conditional operator
- Null-coalescing operators
- Null-conditional operator

ACTIVITY: Exercise 4

Loops and Collections

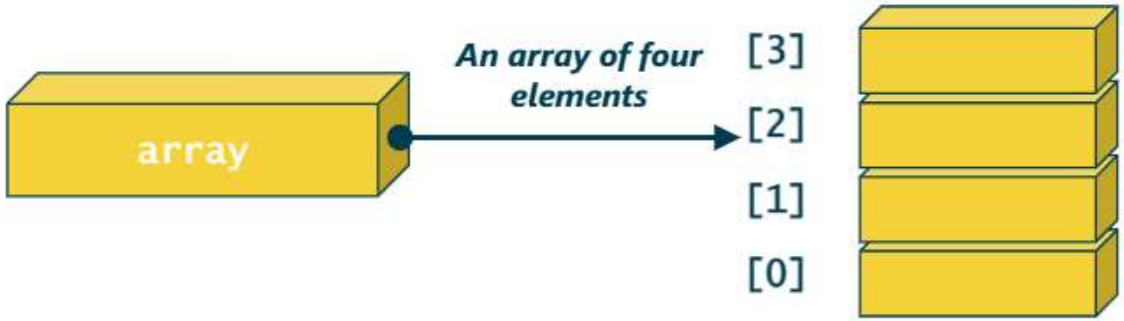
OUTLINE

- Arrays
- Foreach loops
- For loops
- While loops
- Do loops
- Generic collections
- List<T>
- Dictionary<TKey, TValue>
- Collection operators

ARRAYS

An array is a collection of variables all of the same type

- Each element in the array can hold a single item
- Array elements are accessed by a zero-based index number in square brackets




```
int[] arr1 = new int[4]; // 0, 0, 0, 0
Console.WriteLine(arr1[0]); // 0
int[] arr2 = new int[] { 1, 3, 5, 7, 9 }; // array initializer
int x = arr2[2];
Console.WriteLine(x);// 5
```

ARRAY INITIALISATION

- An array can be initialised without **new[ ]** if the type is defined and the values are provided
- An array can be *implicitly typed* using **new[ ]** and providing values whose type can be inferred
- When *declaring* and *initialising* an array variable separately, you must use the **new** operator

```
int[] arr3 = { 2, 4, 6, 8 }; // array initializer without new[]
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
Car[] cars = { new Car(), new Car(), new Car() };

// Implicitly typed arrays
var a = new[] { 1, 10, 100, 1000 }; // int[]
var b = new[] { "hello", null, "world" }; // string[]

// must use 'new' when declaring and initializing separately
int[] arr4;
arr4 = new int[] { 1, 3, 5, 7, 9 };   // OK
//arr4 = {1, 3, 5, 7, 9};   // Error
int[] arr5;
arr5 = new[] { 1, 3, 5, 7, 9 };   // OK
```

LOOPING THROUGH AN ARRAY

- The **foreach** statement enumerates the elements of a collection and executes its body for each element

```
int[] array6 = { 1, 1, 2, 3, 5, 8, 13, 21 };
foreach (int i in array6)
{
    Console.WriteLine(i * 2);// 2 2 4 6 10 16 26 42
}

string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
foreach (string day in weekDays)
{
    Console.WriteLine(day);// Sun Mon Tue Wed Thu Fri Sat
}
```

```
Coords[] coordinates = { new Coords(10, 20), new Coords(8, 3), new Coords(5, 5) };

foreach (Coords coord in coordinates)
{
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
}
// Output:
// X is 10 and Y is 20
// X is 8 and Y is 3
// X is 5 and Y is 5
```

FOREACH

- You can use **var** in the **foreach** loop to let the compiler infer the type of the iteration variable

```
// use var to let the compiler infer the type of the iteration variable
foreach (var coord in coordinates)
{
    Console.WriteLine(coord.GetType());// Coords
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
}
```

- You cannot modify the members of the iteration variable within a **foreach** loop

```
// use var to let the compiler infer the type of the iteration variable
foreach (var coord in coordinates)
{
    Console.WriteLine(coord.GetType());
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
    coord.X++;
    coord.Y--;
}
```

(local variable) Coords coord

CS1654: Cannot modify members of 'coord' because it is a 'foreach iteration variable'

ITERATION STATEMENTS

C# has four types of iteration statement:

- The **foreach** statement
- The **for** statement
- The **do** statement
- The **while** statement

- **Foreach** is used to iterate over a collection
- **For** executes its body while a specified Boolean expression evaluates to *true*
- **Do** conditionally executes its body *one* or more times
- **While** conditionally executes its body *zero* or more times

FOR

For executes its body while a specified Boolean expression evaluates to *true*

A **for** statement is made up of:

- An initialiser — int i = 0
- A condition — i < 5
- An iterator — i++

```
for (int i = 0; i < 5; i++)
{
    Console.Write(i);
}
// Output:
// 01234
```

FOR LOOP EXAMPLES: ITERATOR SECTION

Iterators in a **for** loop can be incremented or decremented and can use compound assignment.

```
// decrement iterator
int x;
for (x = 10; x >= 5; x--)
{
    Console.Write($"{x} ");
}
// Output:
// 10 9 8 7 6 5


// iterator using compound assignment
for (int i = 0; i <= 10; i += 2)
{
    Console.Write($"{i} ");
}
// Output:
// 0 2 4 6 8 10
```

FOR LOOP EXAMPLES: INITIALISER SECTION

Initialisers in a **for** loop can use multiple loop variables that you then increment or decrement in the iterator section.

```
// multiple loop variables
for (int i = 0, j = 0; i + j < 5; i++, j++)
{
    Console.WriteLine($"Value of i: {i}, J: {j}");
}
// Value of i: 0, J: 0
// Value of i: 1, J: 1
// Value of i: 2, J: 2
```

FOR LOOP EXAMPLES: NESTED LOOPS

You can nest **for** loops:

```
// nested for loops
for (int i = 0; i < 2; i++)
{
    for (int j = i; j < 4; j++)
    {
        Console.WriteLine($"Value of i: {i}, J: {j}");
    }
}
// Value of i: 0, J: 0
// Value of i: 0, J: 1
// Value of i: 0, J: 2
// Value of i: 0, J: 3
// Value of i: 1, J: 1
// Value of i: 1, J: 2
// Value of i: 1, J: 3
```

FOR LOOP EXAMPLES: JUMP STATEMENTS

C# has the following jump statements that are used with loops:

- **continue:** Start a new iteration
- **break:** Terminate the closest enclosing loop statement
- **goto:** Transfer control to a statement that is marked by a label

```
// break to conditionally exit out of the loop early
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        break;

    Console.Write($"{i} ");
}// 0 1 2 3 4


// continue to conditionally jump to the top of the loop
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        continue;

    Console.Write($"{i} ");
}// 0 1 2 3 4 6 7 8 9
```

FOR LOOP EXAMPLES: JUMP STATEMENTS

The **goto** statement transfers control to a statement that is marked by a label which can be used to exit out of nested loops:

```
//nested for loops with goto
for (int i = 0; i < 2; i++)
{
    for (int j = i; j < 4; j++)
    {
        if (j == 3)
        {
            goto pickupPoint;
        }
        Console.WriteLine($"Value of i: {i}, J: {j}");
    }
}

pickupPoint:
Console.WriteLine("Code continues here after GOTO");

// Output:
// Value of i: 0, J: 0
// Value of i: 0, J: 1
// Value of i: 0, J: 2
// Code continues here after GOTO
```

WHILE LOOPS

A **while** loop conditionally executes its body *zero* or more times whilst a specified Boolean expression evaluates to *true*.

```
int n = 0; // initialization
while (n < 5) // Boolean expression
{
    Console.Write($"{n} ");
    n++; // increment
}
// 0 1 2 3 4
```

It is important to modify the conditional variable otherwise you will have an infinite loop.

WHILE LOOP EXAMPLES

A **while** loop is typically used when you don't know how many times you want the loop body to execute, which is why it uses a conditional test.

A **for** loop is used when you can count the number of times you want the loop body to execute.

```
// equivalent of a 'for' loop
int i = 0;

while (true)
{
    Console.Write($"{i} ");

    i++;

    if (i > 5)
        break;
}
// 0 1 2 3 4 5
```

While loops can be nested:

```
// nested while loops
int i = 0, j = 1;

while (i < 2)
{
    Console.WriteLine("i = {0}", i);
    i++;

    while (j < 2)
    {
        Console.WriteLine("J = {0}", j);
        j++;
    }
}
// i = 0
// J = 1
// i = 1
```

DO LOOPS

A **do** loop conditionally executes its body *one* or more times whilst a specified Boolean expression evaluates to *true*.

```
int i = 0; // initialization
do
{
    Console.Write($"{i} ");
    i++; // increment

} while (i < 5); // Boolean expression
// 0 1 2 3 4
```

It is important to modify the conditional variable, otherwise you will have an infinite loop.

DO LOOP EXAMPLES

A **do** loop can use the jump statements **continue**, **break,** and **goto:**

```
// do loop with break
int i = 0;
do
{
    Console.Write($"{i} ");
    i++;

    if (i > 5)
        break;

} while (i < 10);
// 0 1 2 3 4 5
```

Do loops can be nested:

```
// nested do while
int a = 0;
do
{
    Console.WriteLine($"a = {a} ");
    int b = a;

    a++;

    do
    {
        Console.WriteLine($"b = {b} ");
        b++;
    } while (b < 2);

} while (a < 2);
// a = 0
// b = 0
// b = 1
// a = 1
// b = 1
```

LOOPS SUMMARY

```
int[] numbers = { 0, 1, 2, 3, 4};
foreach (int i in numbers)
{
    Console.Write(i);
}
//01234
```

foreach

```
for (int i = 0; i < 5; i++)
{
    Console.Write(i);
}
//01234
```

for

```
int n = 0; // initialization
while (n < 5) // Boolean expression
{
    Console.Write($"{n} ");
    n++; // increment
}
// 0 1 2 3 4
```

while

```
int i = 0; // initialization
do
{
    Console.Write($"{i} ");
    i++; // increment

} while (i < 5); // Boolean expression
// 0 1 2 3 4
```

do

GENERIC COLLECTIONS

The **System.Collections.Generic** namespace contains classes and interfaces that define generic collections.

A **generic collection** allows users to create strongly typed collections that provide better performance and type safety than non-generic collections.

Common generic collection classes are:

- **List<T>**
- **Dictionary<TKey, TValue>**

- <T> is the type of elements in the list
- <TKey> is the type of the keys in the dictionary
- <TValue> is the type of the values in the dictionary

LIST<T>

The **List<T>** class represents a strongly typed list of objects

- The objects can be accessed by index
- The list can be manipulated (insert, add, remove)
- The list can be searched
- The list can be sorted

```
List<string> olympicCities = new() { "Sydney", "Athens", "Beijing", "London", "Rio"};
olympicCities.Add("Tokyo");

// access an object by index
string city2012 = olympicCities[3];
Console.WriteLine(city2012);// London

olympicCities.Insert(2, "Bognor");

foreach (var city in olympicCities)
{
    Console.Write($"{city} "); ;
}
// Sydney Athens Bognor Beijing London Rio Tokyo
```

LIST<T> EXAMPLE

```
List<string> upcomingCities = new() { "Paris", "Los Angeles", "Brisbane" };
// search the list and add a range of string objects
if (!olympicCities.Contains("Paris"))
{
    olympicCities.AddRange(upcomingCities);
}

// search and remove an object from the list
int bognorIndex = olympicCities.IndexOf("Bognor");
Console.WriteLine("Bognor is at index position {0}", bognorIndex);
// Bognor is at index position 2

olympicCities.Remove("Bognor");

bognorIndex = olympicCities.IndexOf("Bognor");
Console.WriteLine("Bognor is at index position {0}", bognorIndex);
// Bognor is at index position -1

// sort the list of strings using the default comparer
olympicCities.Sort();
foreach (var city in olympicCities)
{
    Console.Write($"{city} "); ;
}
// Athens Beijing Brisbane London Los Angeles Paris Rio Sydney Tokyo
```

DICTIONARY <TKEY,TVALUE>

The **Dictionary<TKey,TValue>** class represents a strongly typed collection of *keys* and *values*

- The keys must be unique and cannot be null
- The values can be null or duplicates
- The values are accessed by indexing the key in square brackets **[key]**

```
Dictionary<string, int> cities = new()
{
    ["Sydney"] = 4_992_000,
    ["Athens"] = 3_167_000,
    ["Beijing"]= 21_540_000
};

int population = cities["Athens"]; //cities is indexed by the key name
Console.WriteLine("Population of Athens is {0}",population);
// Population of Athens is 3167000

foreach (KeyValuePair<string, int> kvp in cities)
{
    string cityKey = kvp.Key;
    int populationValue = kvp.Value;
    Console.WriteLine($"City {cityKey} has a population of {populationValue}");
}
// City Sydney has a population of 4992000
// City Athens has a population of 3167000
// City Beijing has a population of 21540000
```

DICTIONARY <TKEY,TVALUE> ITERATION

You can iterate over a **dictionary** using:

- Keys
- Values
- Key-value pairs

```
// iterate over the keys
foreach (string cityKey in cities.Keys)
{
    Console.Write($"{cityKey} ");
}
// Sydney Athens Beijing

// iterate over the values
foreach (int populationValue in cities.Values)
{
    Console.Write($"{populationValue} ");
}
// 4992000 3167000 21540000

// iterate over the keyvalue pairs
foreach (KeyValuePair<string, int> kvp in cities)
{
    Console.WriteLine(kvp.ToString());
}
// [Sydney, 4992000]
// [Athens, 3167000]
// [Beijing, 21540000]
```

DICTIONARY <TKEY,TVALUE> EXAMPLE

```
// add objects to a dictionary
cities.Add("London", 8_982_000);

// lookup a value using the key
int populationLondon = cities["London"];
Console.WriteLine(populationLondon);

if (cities.ContainsKey("Rio"))
{
    Console.WriteLine(cities["Rio"]);
}

// update a value
cities["London"] = 9_000_000;
// iterate over the keyvalue pairs
foreach (KeyValuePair<string, int> kvp in cities)
{
    Console.WriteLine(kvp.ToString());
}
// [Sydney, 4992000]
// [Athens, 3167000]
// [Beijing, 21540000]
// [London, 9000000]

// remove an object
cities.Remove("London");

// iterate over the keys
foreach (string cityKey in cities.Keys)
{
    Console.Write($"{cityKey} ");
}
// Sydney Athens Beijing
```

GENERIC COLLECTIONS: STRONGLY TYPED

- Generic collections are strongly-typed
- They ensure the correct datatypes are used and will generate compiler errors if incorrect types are passed

```
// Generics are strongly-typed

//List<string> olympicCities...
olympicCities.Add("Tokyo");//  <string> OK
olympicCities.Add(true);// <bool> compile error
olympicCities.Add(1234);// <int> compile error

//Dictionary<string, int> cities...
cities.Add("London", 8_982_000);// <string, int> OK
cities.Add(8_982_000, "London");// <int, string> compile error
cities.Add("Paris", "2_140_000");// <string, string> compile error
```

MORE GENERIC COLLECTIONS

There are many generic collection classes, such as:

- **Stack<T>** A variable size last-in-first-out (LIFO) collection
- **Queue<T>** A first-in, first-out (FIFO) collection
- **SortedSet<T>** A collection of objects that is maintained in sorted order
- **SortedList<TKey, TValue>** A collection of key/value pairs that are sorted by key
- **SortedDictionary<TKey, TValue>** A collection of key/value pairs that are sorted by key

COLLECTION OPERATORS

There are three useful operators for working with collections:

- Array element or indexer access operator **[ ]**
- Index from end operator **^**
- Range operator **..**

INDEXER ACCESS OPERATOR

- The array element or indexer access operator **[ ]** is used to access elements in an array or collection using an index value or key

```
// indexer access operator
string[] drinks = { "Water", "Coffee", "Tea", "Orange Juice" };
Console.WriteLine($"The zeroth drink is {drinks[0]}");// Water
Console.WriteLine($"The last drink is {drinks[3]}");// Orange Juice

List<string> snacks = new() { "Apple", "Crisps", "Biscuits" };
Console.WriteLine($"The zeroth snack is {snacks[0]}");// Apple
Console.WriteLine($"The last snack is {snacks[2]}");// Biscuits

Dictionary<string, int> foodCalories = new()
{
    ["Banana"] = 89,
    ["Chocolate Digestive"] = 84
};

Console.WriteLine($"Calories in a banana = {foodCalories["Banana"]}");
// Calories in a banana = 89
```

INDEX FROM END OPERATOR AND RANGE OPERATOR

- The **index from end** operator ^ indicates the element position from the end of a sequence
- The **range** operator **..** specifies the start and end of a range of indices
  - The left-hand operand is inclusive
  - The right-hand operand is exclusive

```
List<int> numbers = new() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// index from end operator
var firstFromEnd_10 = numbers[^1];
var thirdFromEnd_8 = numbers[^3];

// range operator
var slice_345678910 = numbers.ToArray()[2..];
var slice_12345678910 = numbers.ToArray()[..];

// range and index from end operators
var slice_34567 = numbers.ToArray()[2..^3];
var slice_1234567 = numbers.ToArray()[..^3];
```

SUMMARY

- Arrays
- Foreach loops
- For loops
- While loops
- Do loops
- Generic collections
- List<T>
- Dictionary<TKey, TValue>
- Collection operators

ACTIVITY: Exercise 5

Object Oriented Programming (OOP)

OUTLINE

- Why object orientation?
- Scenario: Procedural programming versus OOP
- OOP concepts and Unified Modelling Language (UML)
  - Classes
  - Instantiation
  - Association
  - Aggregation
  - Inheritance
  - Interfaces
- Class activity: OOP concepts
- Activity: OOP quiz
- Activity: Draw a class diagram

WHY OBJECT ORIENTATION?

Object-oriented programming (OOP) evolved from programming best practices.

Represents the real world

- People interact with *things*, not database records

Ease of maintenance

- Code is structured
- Functionality and data are together in one place
- Promotes code reuse through object instantiation or other OOP techniques, such as inheritance

C# is thoroughly object-oriented

- E*verything* is an object including value types

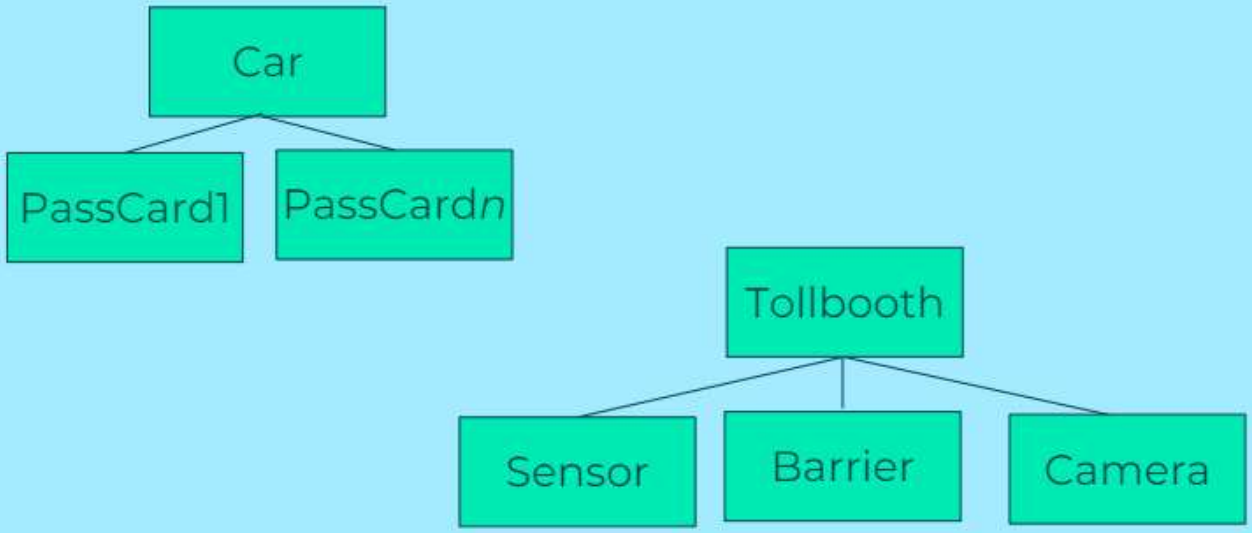
TOLL ROAD EXAMPLE: PROCEDURAL APPROACH

1. Wait for the sensor to detect a car.
2. Take a photograph of the registration plate.
3. Using OCR technology, decode the letters and numbers of the registration plate.
4. If the plate cannot be read then require manual intervention.
5. If it is a UK plate, proceed, otherwise require manual intervention.
6. Use the camera to read the barcode on the pass card displayed on the windscreen. Each car could have a number of pass cards but only one should be on display.
7. Check in a database of valid pass cards that it is not stolen and it has not expired.
8. If it is stolen, inform the authorities: Department of Transport on weekdays and the police at weekends.
9. If it has expired, this will have to be a cash transaction.
10. Check the registration number against the list of approved cars for this pass card.
11. All checks are OK so raise the barrier.

TOLL ROAD EXAMPLE: OBJECT ORIENTED APPROACH

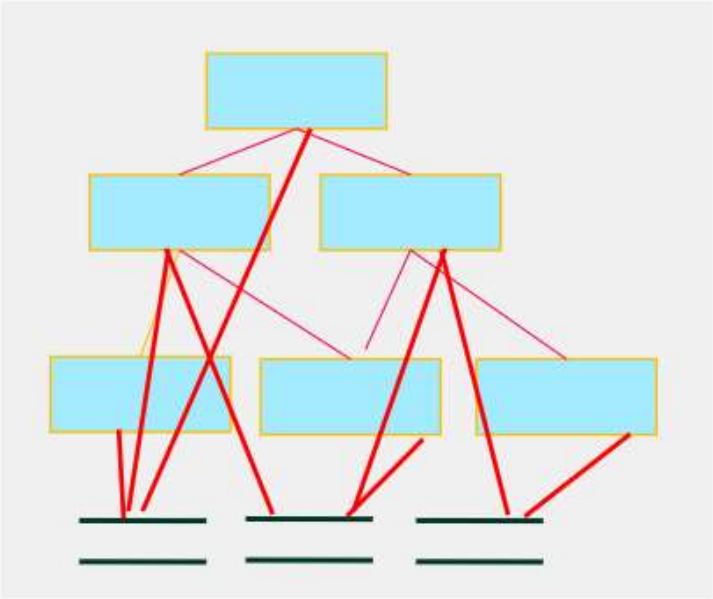
OOP APPROACH

Real-world objects

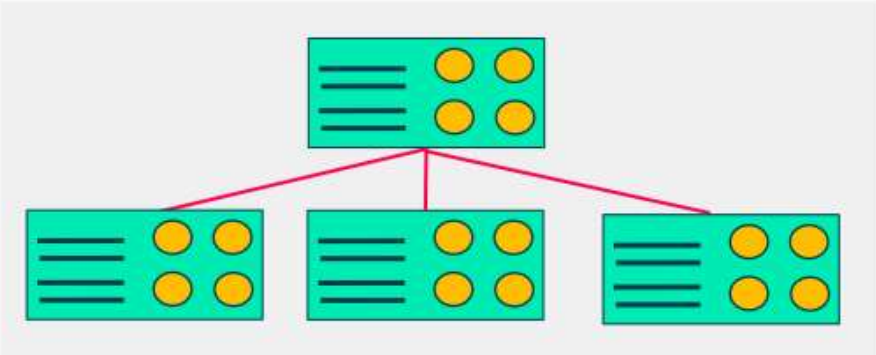


Objects within the computer's RAM

DEPENDENCIES



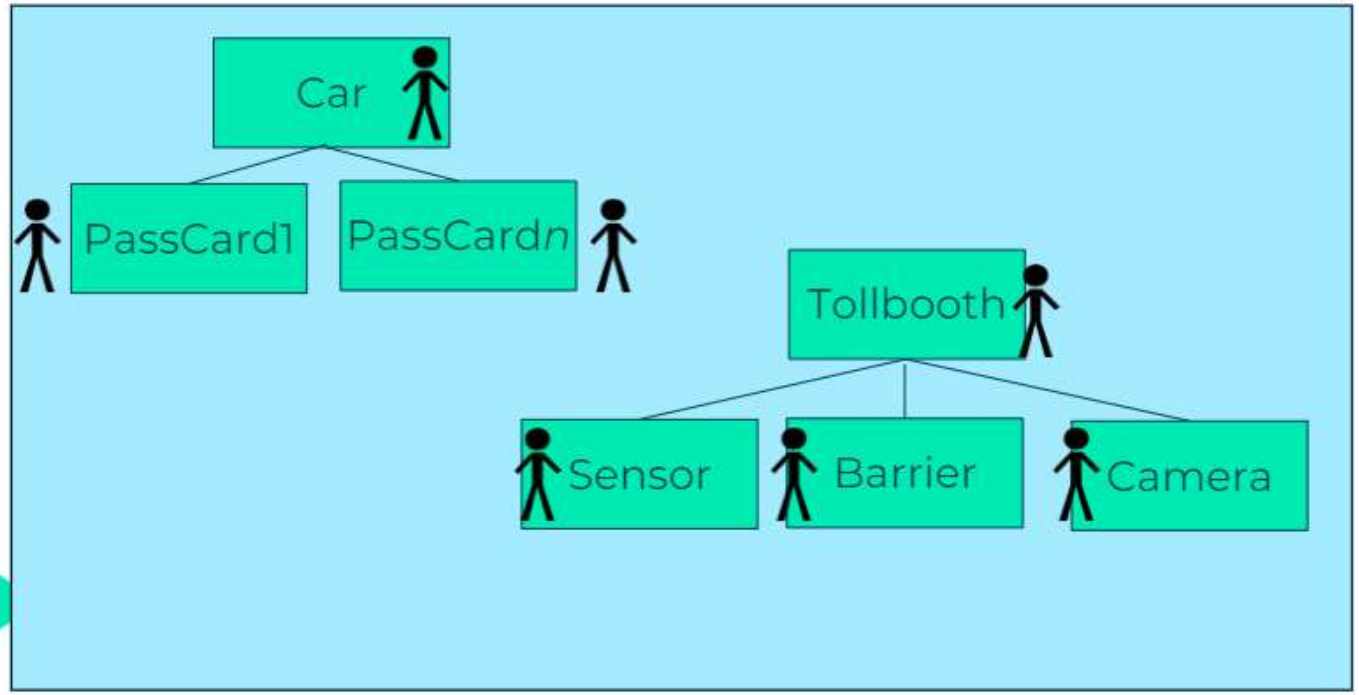
Procedural approach



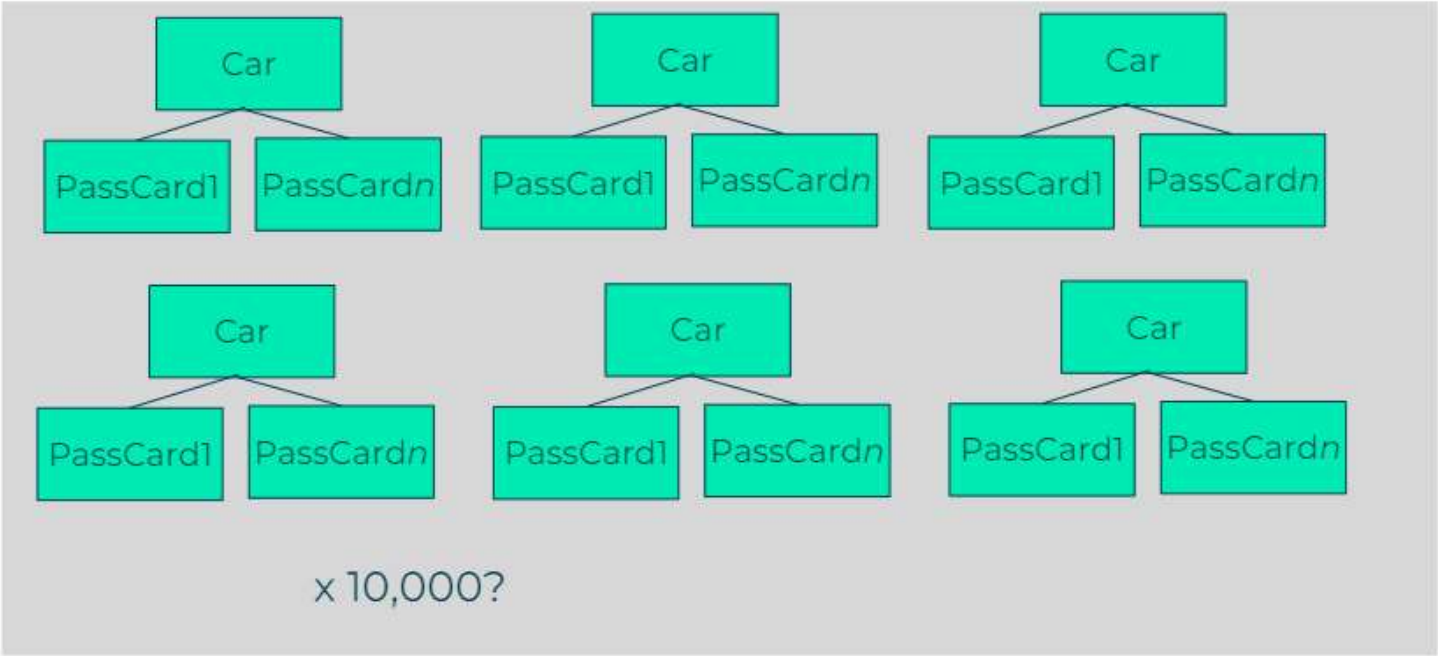
OOP approach

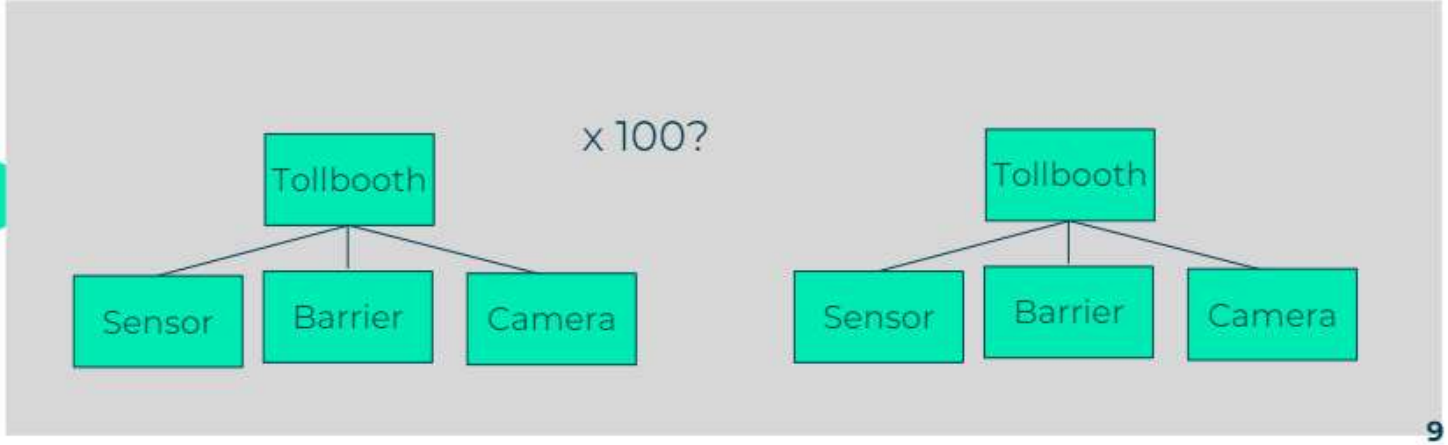
ANTHROPOMORPHISM

The attribution of human characteristics or behaviour to an object.

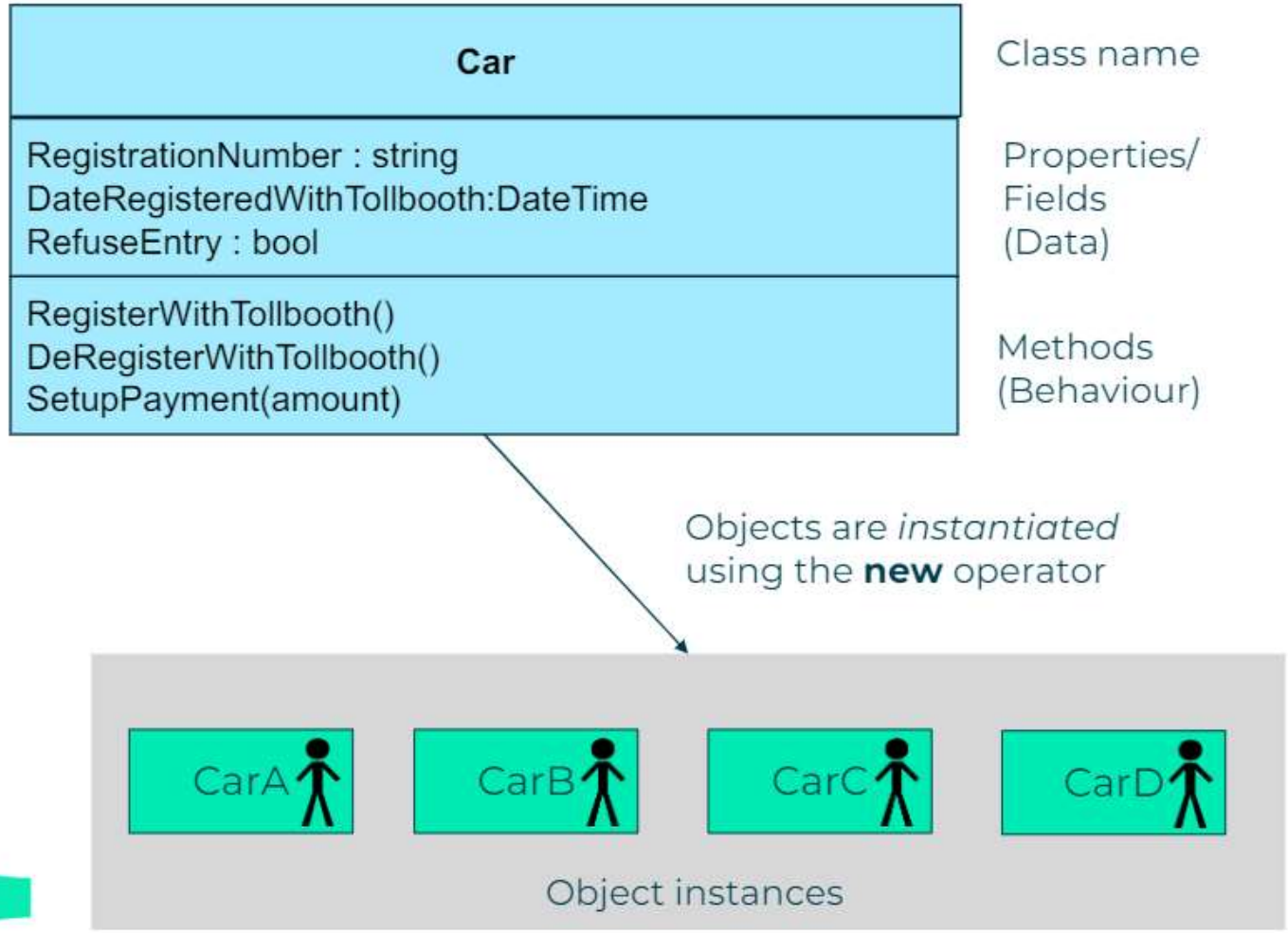


MULTIPLE INSTANCES





INSTANTIATION



CAR CLASS CODE

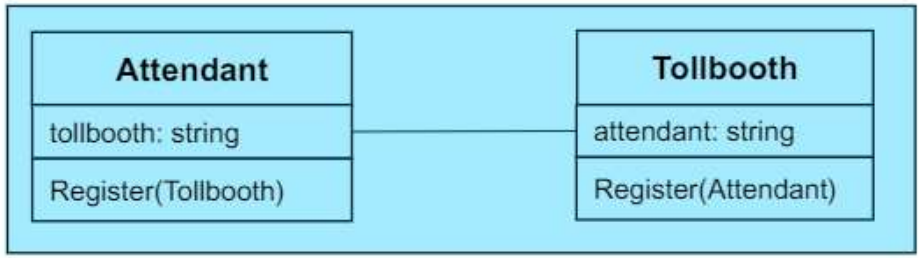
Car
RegistrationNumber : string DateRegisteredWithTollbooth:DateTime RefuseEntry : bool
RegisterWithTollbooth() DeRegisterWithTollbooth() SetupPayment(amount)

```
class Car {
    private string registrationNumber;
    private DateTime dateRegisteredWithTollbooth;
    private bool refuseEntry;

    public void RegisterWithTollbooth() {
    }
    public void DeRegisterWithTollbooth() {
    }
    public void SetupPayment(decimal amount) {
    }
}
```

```
Car myCar = new Car();
```

Instantiating a car

ASSOCIATIONS: 'KNOWS ABOUT' OR 'USES'



Scenario:
An attendant logs on to a particular tollbooth.
The attendant needs to know which tollbooth they are logged into and the tollbooth needs to keep track of which attendant is logged in.

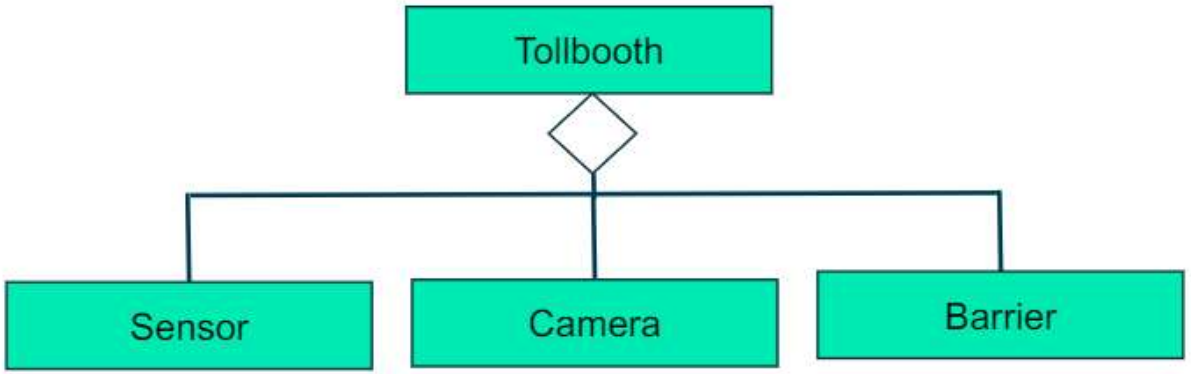
```
class Attendant {
  Tollbooth myTollbooth;

  public void Register(Tollbooth tollbooth)
  {
    myTollbooth = tollbooth;
    myTollbooth.Register(this);
  }
}
```

```
class Tollbooth {
Attendant myAttendant;

  public void Register(Attendant attendant)
  {
        myAttendant = attendant;
  }
}
```

AGGREGATION: 'HAS A'

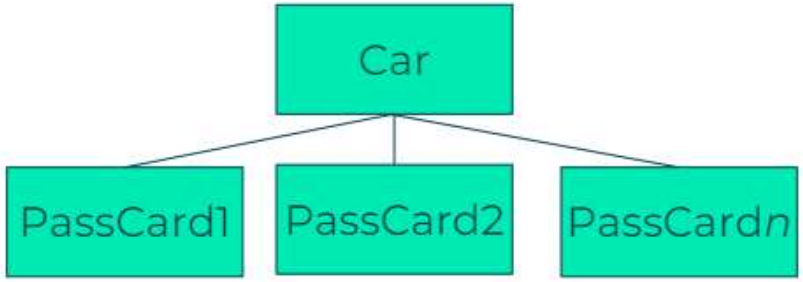
Sometimes there is a container object which is composed of other objects. It does its work by asking its subordinate objects to do work.

We show this association on a UML class diagram with a diamond (**aggregation** symbol). The diamond is unfilled to denote aggregation and filled to denote **composition**. A tollbooth is composed of a sensor, a camera, and a barrier.

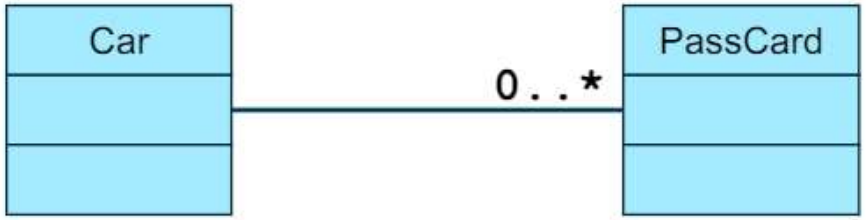


MULTIPLICITY

- Multiplicity represents the number of objects that form a relationship within our system

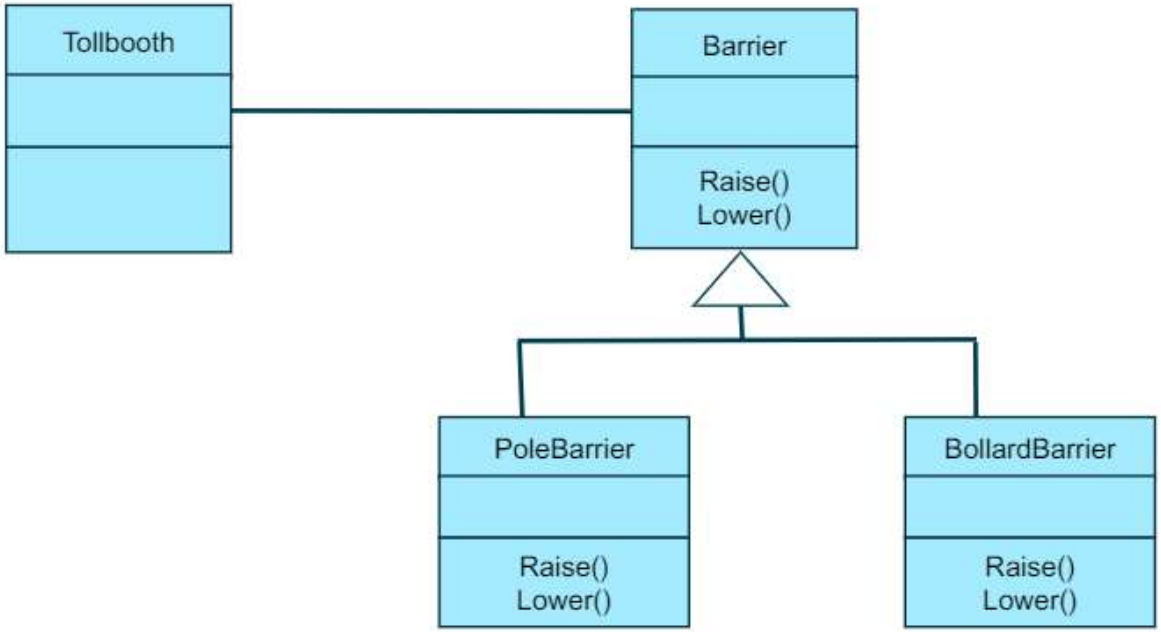


- 'A car can have zero or many pass cards' would be modelled as:



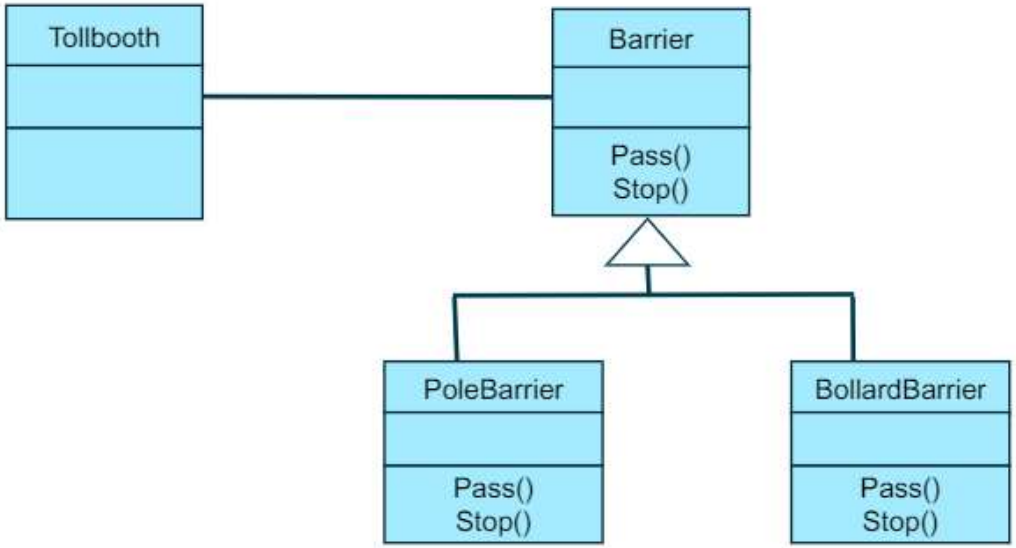
INHERITANCE
'IS A'
OR
'IS A KIND OF'

- Inheritance is basing an object or class upon another object
- The top-level class is called the ***base*** class or ***superclass***
- The inherited class is called the ***derived*** class or ***subclass***



INHERITANCE: IMPLEMENTATION SHOULD BE ENCAPSULATED

- Implementation of a subclass should be well encapsulated and not creep into the superclass
  - A PoleBarier raises to allow a car to *pass*
  - A PoleBarrier lowers to *stop* a car
  - A BollardBarrier raises to *stop* a car
  - A BallardBarrier lowers to allow a car to *pass*

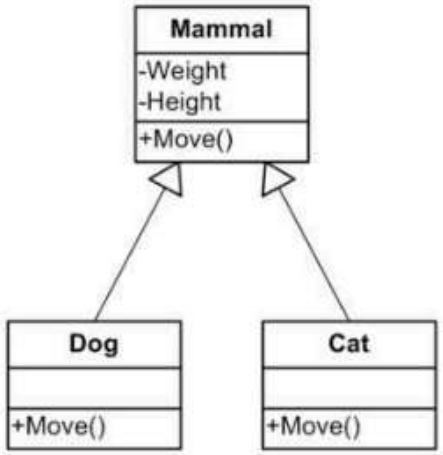


INHERITANCE: EXAMPLE

In the derived (subclasses) we program by difference:

- How is a dog different to our concept of a mammal?
- Does it do some things differently?
- Does it do extra things?

- A dog can do more than a mammal
- It may do some things differently (*overriding*)
- But it can *never* do less than a mammal
- And it can *never* morph into a cat

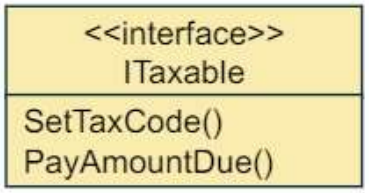


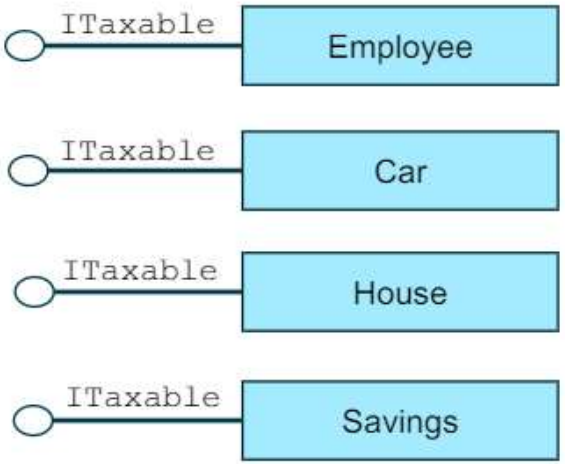
Mammal can be defined as an **abstract** class

- It can be used for inheritance
- It cannot be instantiated

INTERFACE: 'CAN DO'

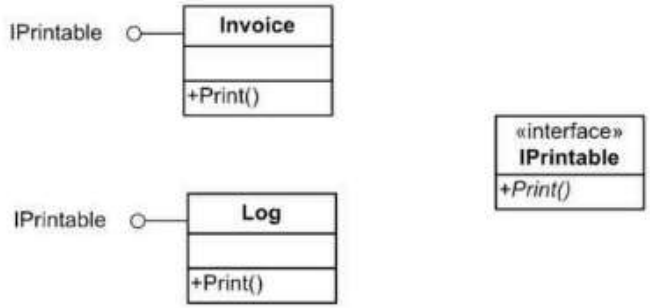
- An interface is a contract or description of all functions that an object must have to be able to act like a specific type
- Interfaces define what an implementing object 'can do', therefore can play the role of the interface type





INTERFACE: EXAMPLE

- You have an **Invoice** and a **Log** of database accesses
- There is no sensible superclass: they are not the same kind of type
- But they do require common functionality: they are both Printable
- Define an **IPrintable** interface and implement it in each class
- Both types must then implement the **Print** method

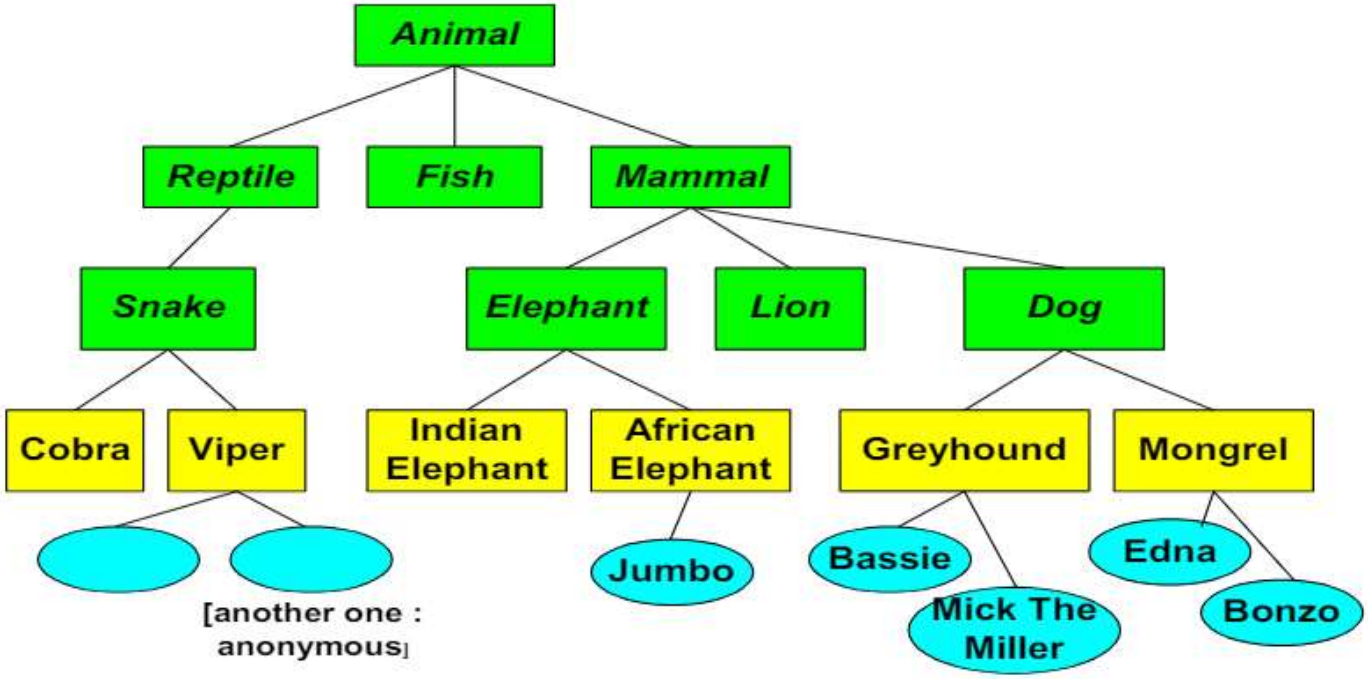


Class activity:

→ Which OOP concept do each of the following UML diagrams convey?

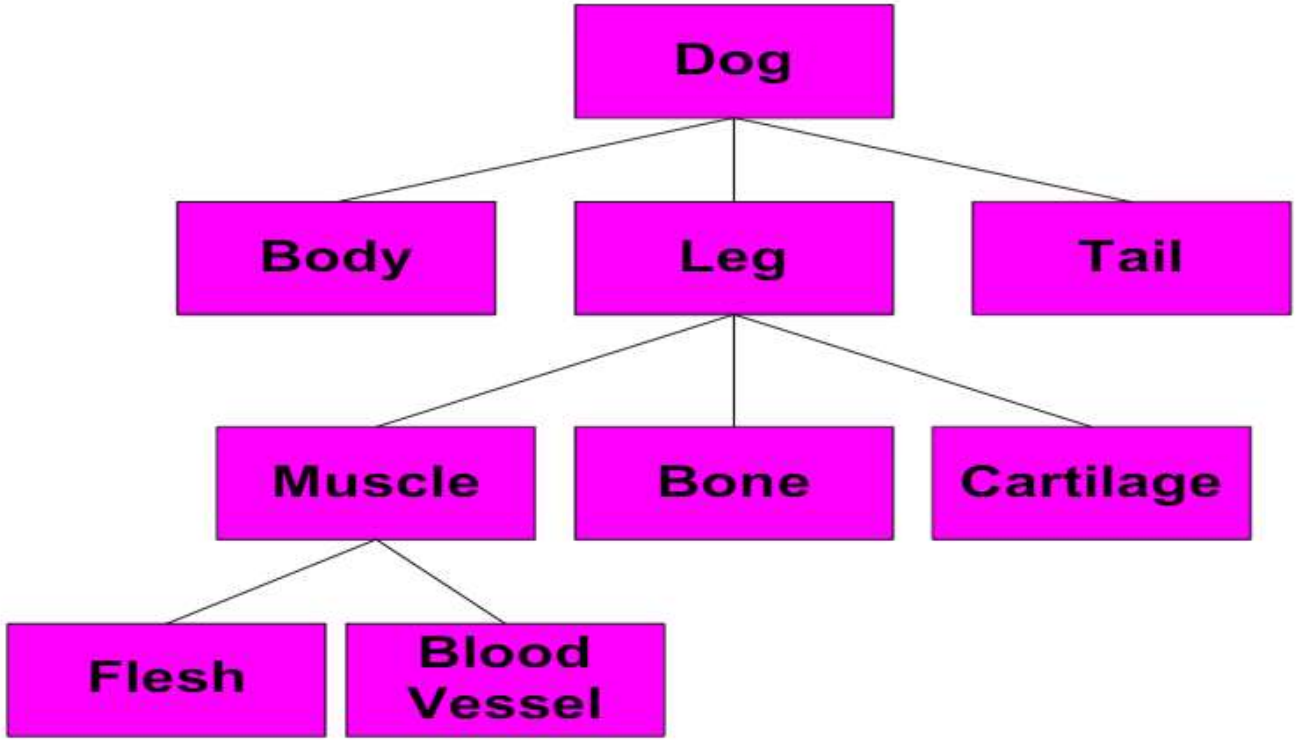
→ The examples are based on a dogs' home scenario.

CONCEPT 1

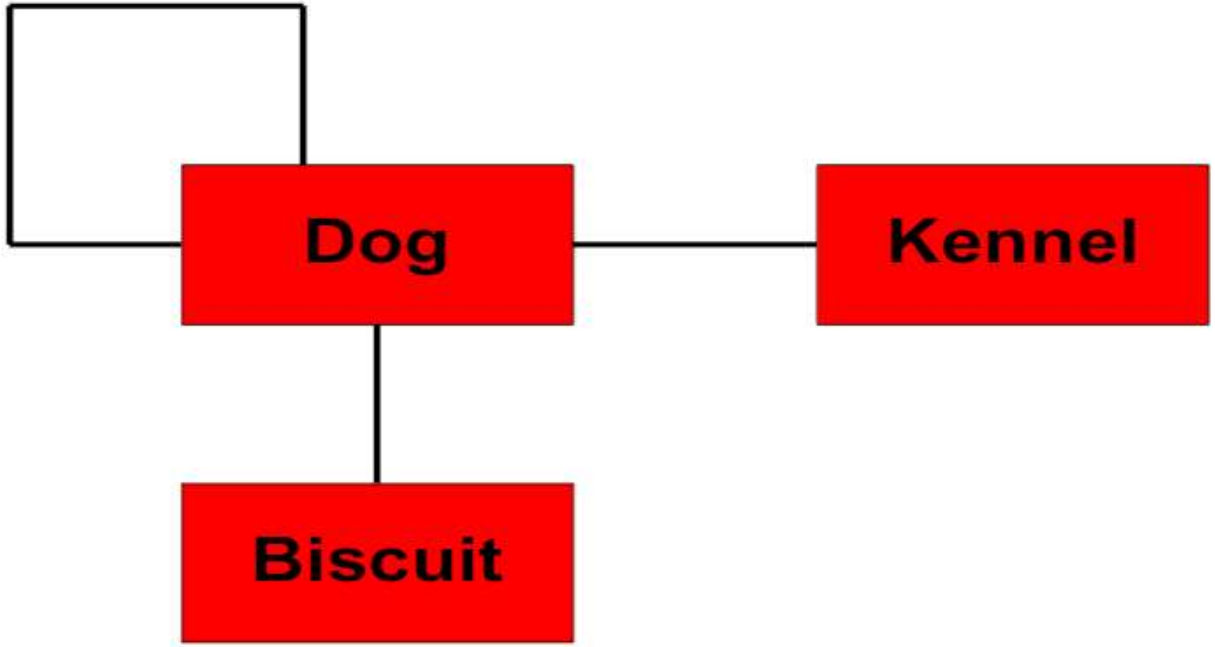

Animal
Reptile
Fish
Mammal
Snake
Elephant
Lion
Dog
Cobra
Viper
Indian Elephant
African Elephant
Greyhound
Mongrel
[another one : anonymous]
Jumbo
Bassie
Mick The Miller
Edna
Bonzo

CONCEPT 2

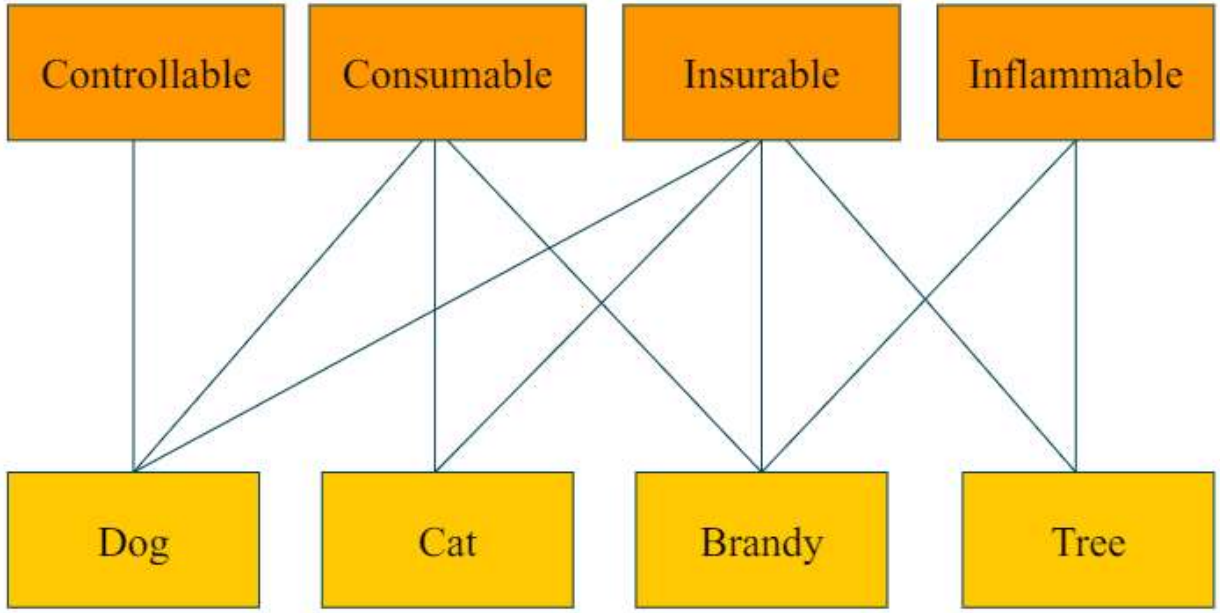
Pincher


Dog
Body
Leg
Tail
Muscle
Bone
Cartilage
Flesh
Blood Vessel

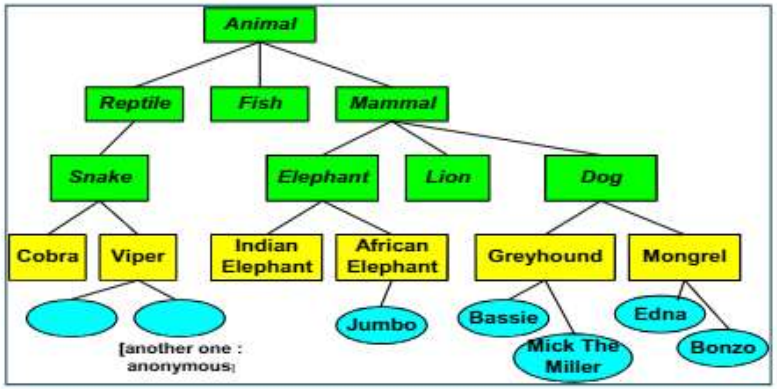
CONCEPT 3

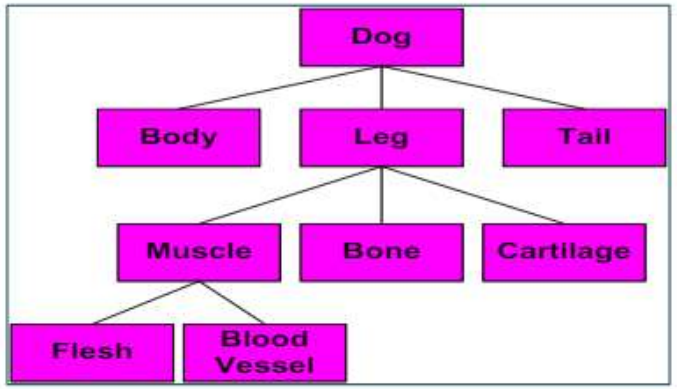


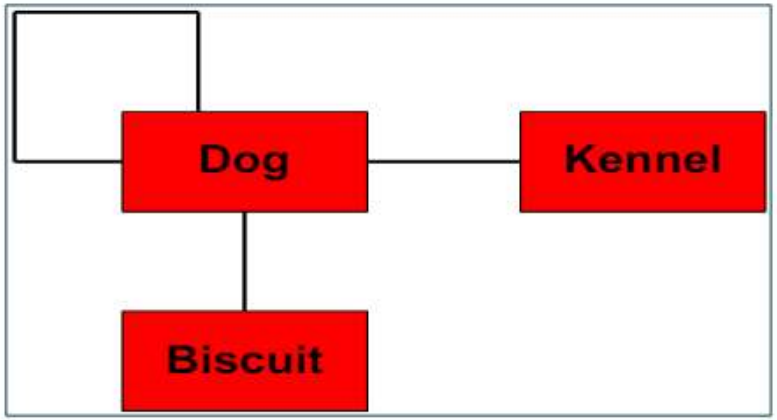
CONCEPT 4

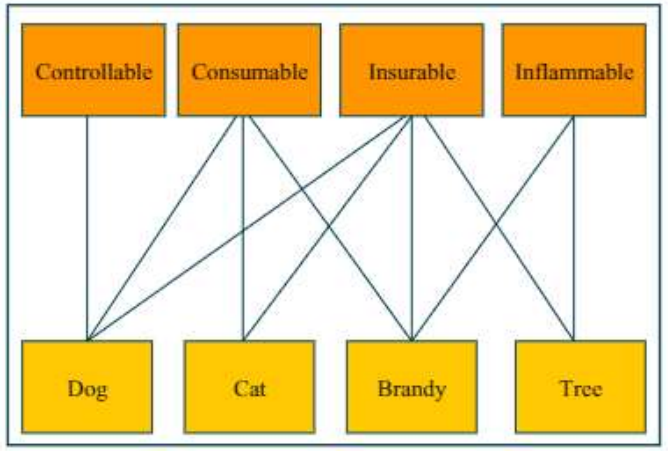

Controllable
Consumable
Insurable
Inflammable
Dog
Cat
Brandy
Tree

ALL CONCEPTS

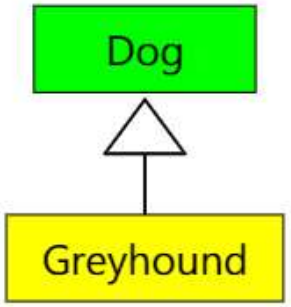




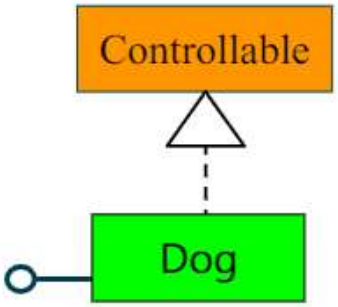




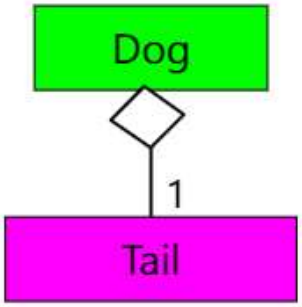
UML NOTATION



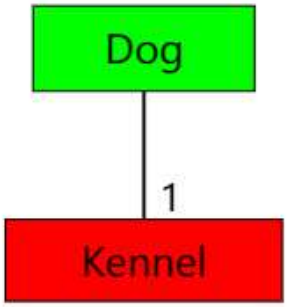
INHERITANCE
Greyhound
'is a kind of'
Dog



INTERFACE
Dog
'exhibits the behaviour of'
Controllable



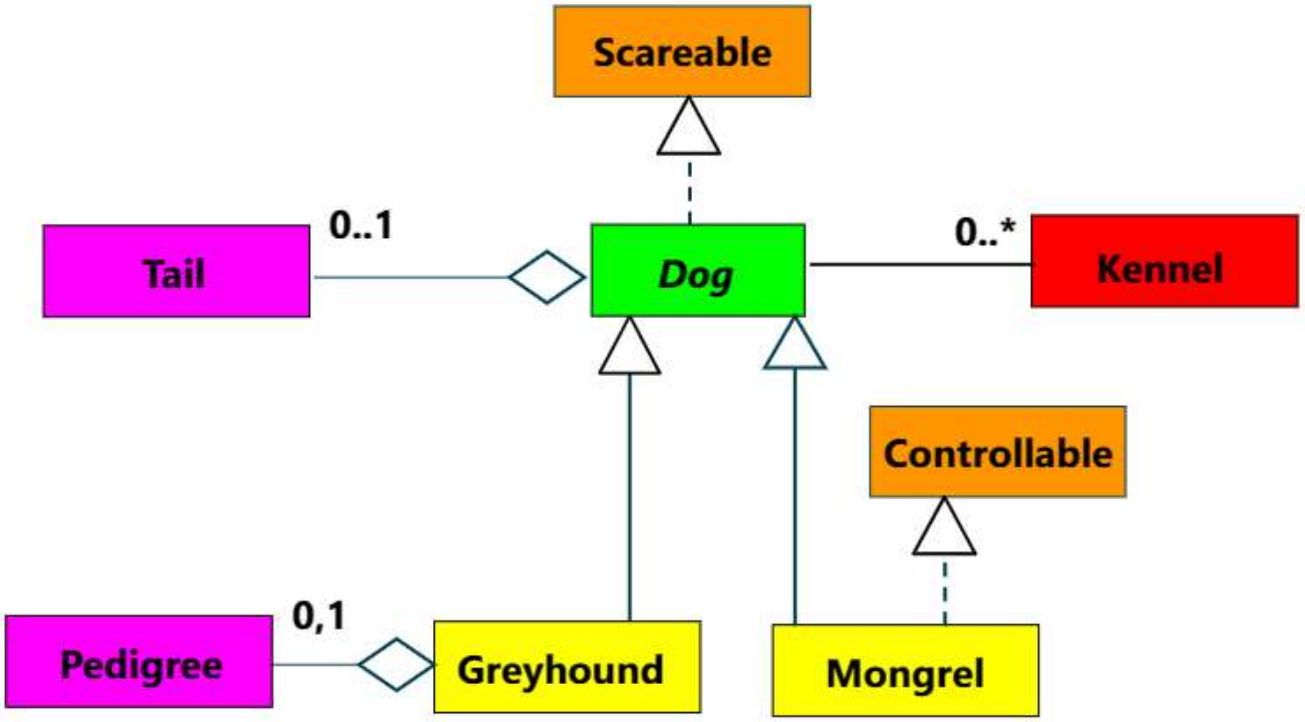
AGGREGATION
Dog
'has the object'
Tail



ASSOCIATION
Dog
'knows about the object'
Kennel

INSTANCE
Bassie
'is an example of a'
Greyhound

COMBINED EXAMPLE



Activity: OOP quiz

- You will consolidate your object-oriented knowledge by taking the OOP Quiz

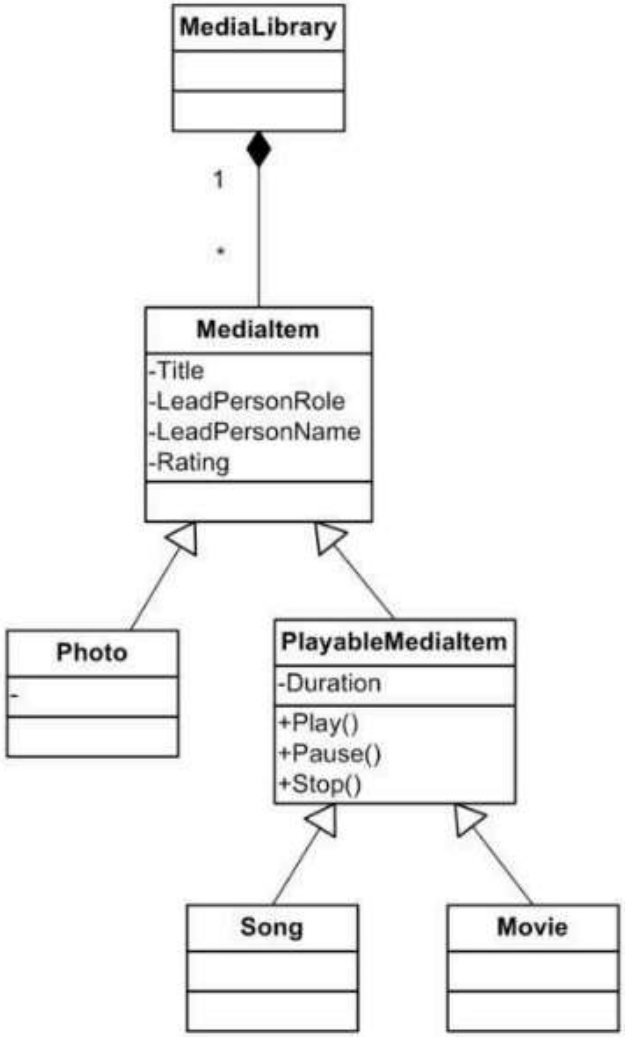
Activity:
Draw a class diagram

- You want to create a Media Library that will contain a list of photos, songs, and movies (*not* one list of photos, one list of songs, and one list of movies)
- Each one of these has a photo, title, lead person, and lead person role (photographer/singer/director) and a rating (rubbish - fantastic)
- Songs and movies are playable (i.e., play, pause, stop) and have a duration

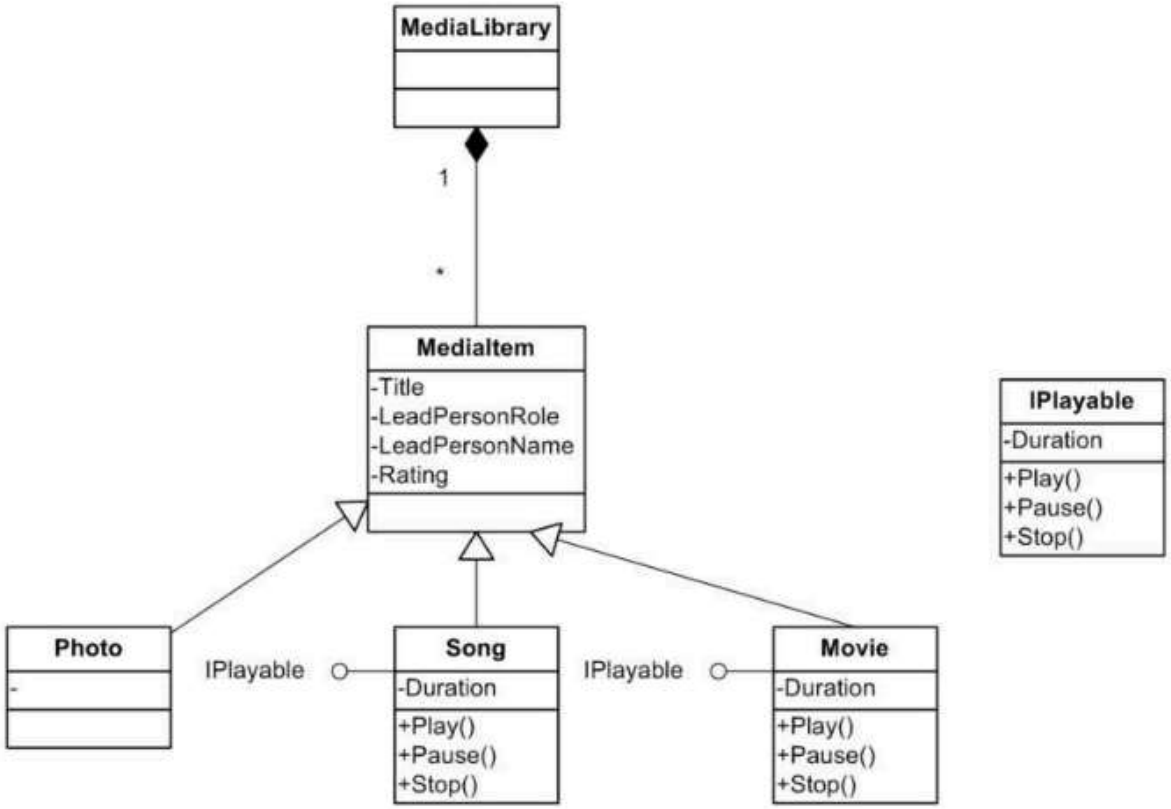
Activity: Solution

- Two possible solutions follow

SOLUTION 1


MediaLibrary
1
*
MediaItem
-Title
-LeadPersonRole
-LeadPersonName
-Rating
Photo
-
PlayableMediaItem
-Duration
+Play()
+Pause()
+Stop()
Song
Movie

SOLUTION 2


MediaLibrary
1
*
MediaItem
-Title
-LeadPersonRole
-LeadPersonName
-Rating
IPlayable
-Duration
+Play()
+Pause()
+Stop()
Photo
-
IPlayable
Song
-Duration
+Play()
+Pause()
+Stop()
IPlayable
Movie
-Duration
+Play()
+Pause()
+Stop()

SUMMARY

- Why object orientation?
- Scenario: Procedural programming versus OOP
- OOP concepts and Unified Modelling Language (UML)
  - Classes
  - Instantiation
  - Association
  - Aggregation
  - Inheritance
  - Interfaces
- Class activity: OOP concepts
- Activity: OOP quiz
- Activity: Draw a Class Diagram

Methods

OUTLINE

- Methods
- Positional parameters
- Named and optional params
- Method overloading
- Return values
- Expression-bodied methods
- Passing parameters
- Passing value-type parameters
  - By value
  - By ref : 'ref', 'out' and 'in'
- Passing reference-type parameters
  - By value
  - By ref
- Static methods and the using static directive
- Extension methods

METHODS

- A method is a code block that contains a series of statements
- Methods are called (invoked) by a program
- Methods are declared in a *class*, *struct*, or *interface* with:
  - An access modifier such as **public** or **private**
  - Optional modifiers such as **abstract**
  - The return type such as **void** or **int**
  - The name of the method
  - Any method parameters in brackets
- This definition is known as the method signature
- Methods can be passed arguments that map to parameters defined in the method signature
- Every C# application has a method called **Main** that is the entry point for the application

METHODS: POSITIONAL PARAMETERS

Method definition:

```
class Car
{
    2 references
    public void Register(string name, string address, string postCode, string country)
    {
        // store this in the DVLA database in Swansea
    }
}
```

Instantiate objects and invoke the method:

```
Car julieCar = new Car();
julieCar.Register("Julie Dooley", "1 Main Street", "CH12 9DL", "UK");


Car lisaCar = new Car();
lisaCar.Register("Lisa Simpson", "742 Evergreen Terrace", "97394", "USA");
```

- Name, address, postcode, and country are the method's parameters. When the method is invoked, the parameters are passed positionally:
  - The zeroth argument 'Julie Dooley' maps to parameter *name*
  - The first argument '1 Main Street' maps to parameter *address*
  - The second argument 'CH12 9DL' maps to parameter *postcode*
  - The third argument 'UK' maps to parameter *country*

NAMED AND OPTIONAL PARAMETERS

- Parameters can be passed by name
- When **named**, the parameters can be passed in any order

```
Car batMobile = new();
batMobile.Register(name: "Bruce Wayne", postCode: "Unknown",
            address: "1007 Mountain Drive, Gotham", country: "USA");
```

- Parameters can be given a default value to make them **optional**:

```
public void Register(string name, string address, string postCode, string country = "USA")
{
}
```

```
batMobile.Register(name: "Bruce Wayne", postCode: "Unknown",
            address: "1007 Mountain Drive, Gotham");
```

METHOD OVERLOADING

You might need to start different cars in different ways:

```
1 reference
public void Start() { }
0 references
public void Start(string keycode) { }
0 references
public void Start(Token token) { }
0 references
public void Start(Fingerprint print) { }
```

You can define different signatures for the same method.

This is an **overloaded method**:

```
batMobile.Start();
▲ 1 of 4 ▼ void Car.Start()
```

```
batMobile.Start();
▲ 2 of 4 ▼ void Car.Start(Fingerprint print)
```

```
batMobile.Start();
▲ 3 of 4 ▼ void Car.Start(string keycode)
```

```
batMobile.Start();
▲ 4 of 4 ▼ void Car.Start(Token token)
```

METHOD RETURN VALUES

- Methods can return a value to the caller if the return type is not **void**
- The value is returned using the **return** keyword
- The type of the value must match the method signature
- The **return** keyword stops execution of the method
- A **void** method can use **return** without a value to stop execution
- A **void** method without return will stop execution at the end of the code block

METHOD RETURN VALUES: EXAMPLE

```
// returns an int
0 references
public int AddTwoNumbers(int number1, int number2) // 2 parameters
{
    return number1 + number2;// execution stops and an int is returned
}

// returns a void
0 references
public void SquareANumber(int number) // 1 parameter
{
    number *= number;
    Console.WriteLine(number);
    return;// execution stops here
    Console.WriteLine(number);// unreachable code
}
```

EXPRESSION BODIED MEMBERS

- Expression body definitions let you provide a member's implementation in a very concise form
- Use an expression body definition whenever the logic required consists of a single expression
- Syntax:
  member => expression

- The following C# members support expression body definitions:
  - Methods
  - Properties
  - Constructors
  - Finalizers
  - Indexers

EXPRESSION BODIED METHODS

- An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for void methods, performs some operation:

```
0 references
public int AddTwoNumbers(int number1, int number2) => number1 + number2;

0 references
public void SquareANumber(int number1) => Console.WriteLine(number1 *= number1);
```

- Block body methods

```
public int AddTwoNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

```
public void SquareANumber(int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

PASSING PARAMETERS

Value types

- When an instance of a **value** type is passed as a parameter to a method, its *copy* is passed instead of the instance
- Changes to the argument within the called method have *no effect* on the original instance in the calling code

Reference types

- When an instance of a **reference** type is passed as a parameter to a method, a *reference* to the object is passed
- Changes to a member of the argument within the called method *are reflected* in the argument in the calling code

Passing value-type parameters

PASSING PARAMETERS: VALUE TYPES BY VALUE

The **value** type variable x is *passed by value* by default, so a copy of the value 7 is passed. This is then squared to give the value of 49 within the called method. The original variable is unchanged.

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
PassingParams pp = new PassingParams();

int x = 7;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(x);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 7
// 49
// The value after calling the method: 7
```

PASSING PARAMETERS: VALUE TYPES BY REFERENCE 'REF'

The value type variable x is *passed by reference* using the **ref** keyword in both the method declaration and the method call. A **reference** to the variable is passed. The value this reference points to is then squared to give the value of 49. The original variable is **changed**.

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(ref int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
PassingParams pp = new PassingParams();

int x = 7;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(ref x);  // Passing the variable by reference
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 7
// 49
// The value after calling the method: 49
```

PASSING PARAMETERS: VALUE TYPES BY REFERENCE 'OUT'

The value type variables are *passed by reference* using the **out** keyword in both the method declaration and the method call. Unlike **ref**, variables do not need to be initialised before being passed.

```
void OutExampleMethod(out int number, out string text, out string optionalString)
{
    number = 42;
    text = "I'm output text";
    optionalString = null;
}

int argNumber;
string argText, argOptionalString;
OutExampleMethod(out argNumber, out argText, out argOptionalString);

Console.WriteLine(argNumber);
Console.WriteLine(argText);
Console.WriteLine(argOptionalString == null);

// Output:
//  42
//  I'm output text
//  True
```

PASSING PARAMETERS: VALUE TYPES BY REFERENCE 'OUT'

From C# 7, you can declare the **out** variables in the argument list of the method call rather than having to declare them beforehand:

```
int argNumber;
string argText, argOptionalString;
OutExampleMethod(out argNumber, out argText, out argOptionalString);
```

```
OutExampleMethod(out int argNumber, out string argText, out string argOptionalString);
```

TUPLES INSTEAD OF OUT

Out parameters are used to return multiple items from a method

An alternative is to return a **collection** when the values belong in a group of the same type e.g., List<string>

Another alternative is to return a **tuple:**

```
(int number, string text, string? optionalString)  OutExampleMethod()
{
    var number = 42;
    var text = "I'm output text";
    string? optionalString = null;
    return (number, text, optionalString);
}

var outputs = OutExampleMethod();
Console.WriteLine($"Outputs: number is {outputs.number}, text is {outputs.text}");
Console.WriteLine($"Outputs: optional string is {outputs.optionalString ?? "NULL"}");
```

A **tuple** is concise syntax to group multiple data elements in a lightweight data structure.

The most common use case is as a method return type within private or internal utility methods.

PASSING PARAMETERS: VALUE TYPES AS 'IN'

- The value type variable x is *passed by reference* using the **in** keyword
- **in** ensures the argument cannot be modified by the called method
- The **in** keyword is optional in the calling code because it is the default passing mechanism

```
PassingParams pp = new PassingParams();

int x = 5;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(x);  // Passing the variable by reference with 'in'
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 5
// 25
// The value after calling the method: 5
```

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

PASSING PARAMETERS: VALUE TYPES AS 'IN'

- The called method cannot modify the **in** parameter, so the code must be changed to reflect this restriction:

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

(parameter) in int number

CS8331: Cannot assign to variable 'in int' because it is a readonly variable

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

Passing reference-type parameters

PASSING PARAMETERS: REFERENCE TYPES

When a **reference-type** parameter is *passed by value*, it is possible to change members belonging to the object but not the object itself.

When a **reference-type** parameter is *passed by reference*, it is possible to change the value of the object itself.

```
internal class SwappingStrings
{
    1 reference
    public void SwapStringsByValue(string s1, string s2)
    // The string parameter is passed by value.
    // Any changes on parameters will not affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }
    0 references
    public void SwapStringsByReference(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }
}
```

PASSING PARAMETERS: REF TYPES BY VALUE

The **reference-type** variables str1 and str2 are *passed by value.*

```
public void SwapStringsByValue(string s1, string s2)
// The string parameter is passed by value.
// Any changes on parameters will not affect the original variables.
{
    string temp = s1;
    s1 = s2;
    s2 = temp;
    System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
}
```

```
SwappingStrings ss = new SwappingStrings();

// ref types by value
string str1 = "Diana";
string str2 = "Prince";
System.Console.WriteLine("Before swapping: {0} {1}", str1, str2);

ss.SwapStringsByValue(str1, str2);   // Passing strings by value
System.Console.WriteLine("After swapping: {0} {1}", str1, str2);


/* Output:
    Before swapping: Diana Prince
    Inside the method: Prince Diana
    After swapping: Diana Prince
*/
```

PASSING PARAMETERS: REF TYPES BY REF

The **reference-type** variables str1 and str2 are *passed by reference*.

```
public void SwapStringsByReference(ref string s1, ref string s2)
// The string parameter is passed by reference.
// Any changes on parameters will affect the original variables.
{
    string temp = s1;
    s1 = s2;
    s2 = temp;
    System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
}
```

```
SwappingStrings ss = new SwappingStrings();

// ref types by reference
string str1 = "Diana";
string str2 = "Prince";
System.Console.WriteLine("Before swapping: {0} {1}", str1, str2);

ss.SwapStringsByReference(ref str1, ref str2);   // Passing strings by reference
System.Console.WriteLine("After swapping: {0} {1}", str1, str2);

/* Output:
    Before swapping: Diana Prince
    Inside the method: Prince Diana
    After swapping: Prince Diana
*/
```

STATIC METHODS

- Use the **static** modifier to declare a static member such as a class or method
- A **static** method belongs to the type itself rather than to a specific instance of the object
- Therefore, **static** methods do not require an object to be instantiated
- A **static** method can't be referenced through an instance
- **WriteLine** is an example of a **static** method of the **Console** class

```
Console.WriteLine();
```

class System.Console
Represents the standard input, output, and error streams for console applications.

STATIC METHODS: EXAMPLE

- The **System.Math** class provides constants and static methods for common mathematical functions
- The directive **using System;** is used in this example

```
int x = 5;
int y = 7;

int lowest = Math.Min(x, y);
int highest = Math.Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Math.Floor(price);
double priceRounded = Math.Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

STATIC METHODS: EXAMPLE 'USING STATIC'

- The directive **using static System.Math;** is used in this example

```
// using static System.Math
int x = 5;
int y = 7;

int lowest = Min(x, y);
int highest = Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Floor(price);
double priceRounded = Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

INSTANCE METHOD EXAMPLE

- For *instance* methods, instantiate an object, then call the method using that object instance

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
// instantiate object
PassingParams pp = new PassingParams();

int x = 7;
pp.SquareANumber(x); // call instance method
```

CREATE AND USE A STATIC METHOD

- For *static* methods, use the **static** modifier on the method definition, then call the method using the class
- Issue a *using directive* to import the static members of the class to make the code less verbose

```
internal class PassingParams
{
    1 reference
    public static void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
int x = 7;
PassingParams.SquareANumber(x); // call static method

// using static PassingParams
int y = 10;
SquareANumber(y); // call static method
```

EXTENSION METHODS

- Extension methods enable a type's functionality to be extended without editing the source code or inheriting from the type
- Extension methods are defined as **static** *methods* in a **static** *class*
- The first parameter defines the *type* that the method 'extends'
- The parameter type is preceded by the **this** modifier

EXTENSION METHOD EXAMPLE

```
static class StringUtils {
        public static int WordCount(this string theString)
        {return theString.Split(' ').Count(); }    }
```

ACTIVITY: Exercise 7

SUMMARY

- Methods
- Positional parameters
- Named and optional params
- Method overloading
- Return values
- Expression-bodied methods
- Passing parameters
- Passing value-type parameters
  - By value
  - By ref : 'ref', 'out' and 'in'
- Passing reference-type parameters
  - By value
  - By ref
- Static methods and the using static directive
- Extension methods

Properties & Constructors

OUTLINE

- Fields
- Properties
  - Properties with backing fields
  - Expression-bodied properties
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Expression-bodied constructors
  - Constructor overloading
  - Constructor chaining
  - Object initialisers
  - Static constructors
- The 'this' keyword

FIELDS

- A **field** is a variable that is declared directly in a class or struct
- A field can be an *instance* field: specific to the object instance
- A field can be a *static* field: shared amongst all objects of that type

```
public class Car {
    public string make;
    public string model;
    public int speed;
}
```

```
// fields do not provide validation beyond the datatype
Car c1 = new();
Car c2 = new();

c1.make = "Ford";
c2.make = "Ferrari";
c1.model = "Fiesta";
c2.model = "Flying submersible"; // not realistic
c1.speed = 9999; // not realistic
```

- A best practice recommendation is to declare fields as **private** or **protected** and define *properties*

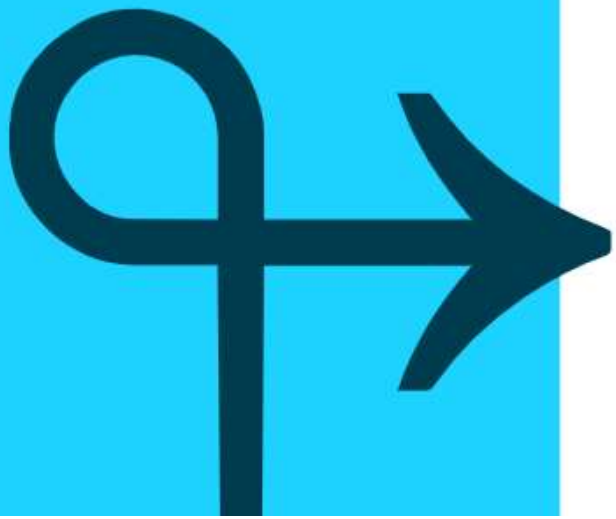
PROPERTIES

- **Properties** are special methods called *accessors*
- They provide a way to read, write, or compute the values of a private field whilst hiding the implementation

- A **get** property accessor is used to return the property value
- A **set** property accessor is used to assign a new **value**
- An **init** property accessor is used to assign a new **value** only during object construction

- Properties can be *read-write* (**get** and **set**)
- Properties can be *read-only* (**get**)
- Properties can be *write-only* (**set**)

PROPERTY SYNTAX



There are three different property syntaxes:

- Properties with backing fields
- Expression body definitions
- Auto-implemented properties

PROPERTIES WITH BACKING FIELDS

- The backing field holds the data
- The accessors can have different visibilities

```
public class Car
{
    private int speed;

    0 references
    public int Speed
    {
        get { return speed; }
        private set { speed = value; }
    }
}
```

EXPRESSION BODIED PROPERTIES

- Property accessors often consist of single-line statements that just assign or return the result of an expression
- Any single-line expression can be implemented using expression body syntax
- If the property is *read-only* (**get**) you can omit the **get** keyword
- If the property is *read-write* (**get** and **set**) you must use the **get** and **set** keywords

```
public class Car
{
    private int speed;

    0 references
    public int Speed => speed;

}
```

```
public class Car
{
    private int speed;

    0 references
    public int Speed
    {
        get => speed;
        private set => speed = value;
    }
}
```

AUTO-IMPLEMENTED PROPERTIES

- Use this syntax if there is no additional logic other than assigning or returning a value
- The C# compiler transparently creates the backing field for you and the implementation code to set / get to / from, the backing field
- You can use an optional initialiser to set a value

```
public class Car
{
    0 references
    public string Make { get; init; } = "Ford";
    1 reference
    public int Speed { get; private set; } = 42;
}
```

CALCULATED PROPERTY EXAMPLE

- **TempInDegreesCelsius** is a read-write auto-implemented property
- **TempInDegreesFahrenheit** is a calculated read-only property

```
public class Temperature
{
    1 reference
    public double TempInDegreesCelsius { get; set; }
    0 references
    public double TempInDegreesFahrenheit
    {
        get { return TempInDegreesCelsius * 1.8 + 32; }
    }
}
```

- **TempInDegreesFahrenheit** is a calculated read-only expression-bodied property

```
public class Temperature
{
    1 reference
    public double TempInDegreesCelsius { get; set; }
    0 references
    public double TempInDegreesFahrenheit => TempInDegreesCelsius * 1.8 + 32;
}
```

ACCESSING PROPERTIES

Property access looks like field access to the client:

```
// instantiate a Car object instance
Car c1 = new();

// set a property value
c1.Model = "Xr2i";

// get property values
Console.WriteLine(c1.Model);
Console.WriteLine(c1.Make);
Console.WriteLine(c1.Speed);
```

Constructing objects

OBJECT CONSTRUCTION

There are two ways to construct an object:

- Constructors
- Object initialisers

Constructors

- Define a constructor (or overloaded constructors) to include all combinations of *mandatory* fields
- This ensures the object is properly setup before being used

Object Initialisers

- Object initialisers can be used for *additional* optional fields that the object creator would like to set
- Object initialisers set properties or fields on the object *after* it has been constructed but before it is used

CONSTRUCTORS

- A constructor is called whenever a class or struct is created
- A class or struct can have multiple overloaded constructors
- A constructor is a method whose name is the same as the name of its type
- Constructors do not include a return type (not even void)
- Constructors are invoked using the **new** operator
- If no constructors are defined for a class or struct, the compiler creates a no-argument parameter-less constructor

```
Employee unknown = new(); //parameter-less constructor
Employee spiderman = new("Peter", "Parker", 1); // 3 arg constructor
```

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH READ-ONLY FIELDS

You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 1

Define *readonly* fields and set their values in the constructor

```
class Employee
{
    private readonly string firstName;
    private readonly string lastName;
    private readonly int employeeID;

    0 references
    public Employee(string firstName, string lastName, int employeeID)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.employeeID = employeeID;
    }
}
```

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH AUTO-IMPLEMENTED PROPERTIES

You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 2

Define *auto-implemented properties* with **get** accessors only.

```
class Employee
{
    1 reference
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }

    // auto-implemented properties
    1 reference
    public string FirstName { get; }
    1 reference
    public string LastName { get; }
    1 reference
    public int EmployeeID { get; }
}
```

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH FULL PROPERTIES

Option 3

Define *full properties* with **get** and **init** accessors.

```
class Employee
{
    1 reference
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }
    // full properties with backing fields
    2 references
    public string FirstName {
        get  { return firstName; }
        init {
            if (value.Length > 0)
            {
                firstName = value;
            }
        }
    }
    2 references
    public string LastName
    {
        get { return lastName.ToUpper(); }
        init
        {
            if (value.Length > 0)
            {
                lastName = value;
            }
        }
    }
    // backing fields
    private string firstName;
    private string lastName;

    // auto-implemented readonly property
    1 reference
    public int EmployeeID { get; }
}
```

EXPRESSION BODIED CONSTRUCTORS

If a constructor can be implemented as a single statement, you can use an expression body definition:

```
public class Location
{
    // private backing field
    private string locationName;

    // expression bodied constructor
    2 references
    public Location(string name) => Name = name;

    3 references
    public string Name
    {
        // expression bodied property accessors
        get => locationName;
        set => locationName = value;
    }
}
```

```
Location home = new("Home");
Location work = new("Work");
Console.WriteLine(home.Name);
Console.WriteLine(work.Name);
```

CONSTRUCTOR OVERLOADING

A constructor is a method and can therefore be overloaded:

```
public class Car
{
    // parameter-less constructor
    1 reference
    public Car()
    {
        Make = "Unkown";
    }
    // 1 arg constructor
    0 references
    public Car(string make)
    {
        Make = make;
    }
    // 2 arg constructor
    0 references
    public Car(string make, string model)
    {
        Make = make; ;
        Model = model;
    }
```

CONSTRUCTOR CHAINING

A constructor can invoke another constructor in the same object (constructor chaining) to avoid duplicating code, using the **this** keyword:

```
public class Car
{
    // parameter-less constructor
    1 reference
    public Car() :this("Unknown")
    { }

    // 1 arg constructor
    2 references
    public Car(string make) : this(make, "Unknown model")
    { }

    // 2 arg constructor
    2 references
    public Car(string make, string model)
    {
        Make = make; ;
        Model = model;
    }
```

```
Car c1 = new();
Car c2 = new("Audi");
Car c3 = new("BMW", "X5");

Console.WriteLine($"Car {nameof(c1)} is make: {c1.Make} and model: {c1.Model}");
Console.WriteLine($"Car {nameof(c2)} is make: {c2.Make} and model: {c2.Model}");
Console.WriteLine($"Car {nameof(c3)} is make: {c3.Make} and model: {c3.Model}");
// Output:
// Car c1 is make: Unknown and model: Unknown model
// Car c2 is make: Audi and model: Unknown model
// Car c3 is make: BMW and model: X5
```

OBJECT INITIALISERS

- To avoid creating many overloaded constructors, *object initialisers* can be used to initialise an object into a ready state
- Object initialisers are often used for *optional* values and constructors are defined for *mandatory* values
- An object initialiser invokes the parameterless constructor unless an explicit constructor is specified
- A **struct** always has a parameterless constructor
- A **class** needs to have an explicit parameterless constructor defined if at least one explicit non-parameterless constructor is defined

```
// parameter-less constructor is used implicitly with an object initializer
Car c4 = new Car { Make = "Audi", Model = "TT", Speed = 70 };
Console.WriteLine($"Car {nameof(c4)} is make: {c4.Make} and model: {c4.Model} and speed: {c4.Speed}");

// an explicit constructor can be used with an object initializer
Car c5 = new Car("Audi","TT") { Speed = 70 };
Console.WriteLine($"Car {nameof(c5)} is make: {c5.Make} and model: {c5.Model} and speed: {c5.Speed}");
```

CONSTRUCTOR ACCESS MODIFIERS

Constructors can be marked as

- public
- private
- internal
- protected internal
- private protected

These define how users of the class can construct an instance of the class.

STATIC CONSTRUCTORS

- Constructors can be marked as **static**
- A **static** constructor *initialises* any *static data* or performs an *action* that needs to be performed only *once*
- Static constructors do not have an *access modifier* or *parameters*
- Any class or struct can only have *one* static constructor
- Static constructors therefore **cannot** be *overloaded*
- Static constructors are called *automatically* by the CLR

```
class Employee
{
    static readonly string companyName;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
     0 references
     static Employee()
    {
        companyName = "QA Ltd";
    }
}
```

THE 'THIS' KEYWORD

The **this** keyword has many uses:

- To qualify members hidden by similar names
- To pass an object as a parameter to other methods
- To chain constructors

```
class Employee
{
    private readonly string firstName;

    0 references
    public Employee(string firstName)
    {
        this.firstName = firstName;
    }
}
```

```
public Employee()
{
    BookSeatInTheOffice(this);
}
```

```
// parameter-less constructor
1 reference
public Car() :this("Unknown")
{ }

// 1 arg constructor
2 references
public Car(string make) : this(make, "Unknown model")
{ }
```

SUMMARY

- Fields
- Properties
  - Properties with backing fields
  - Expression-bodied properties
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Expression-bodied constructors
  - Constructor overloading
  - Constructor chaining
  - Object initialisers
  - Static constructors
- The 'this' keyword

Activity: Exercise 8

Inheritance and Abstract Classes

OUTLINE

- Inheritance
- Derived constructors
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators
- Overriding System.Object methods
- Sealed classes and members

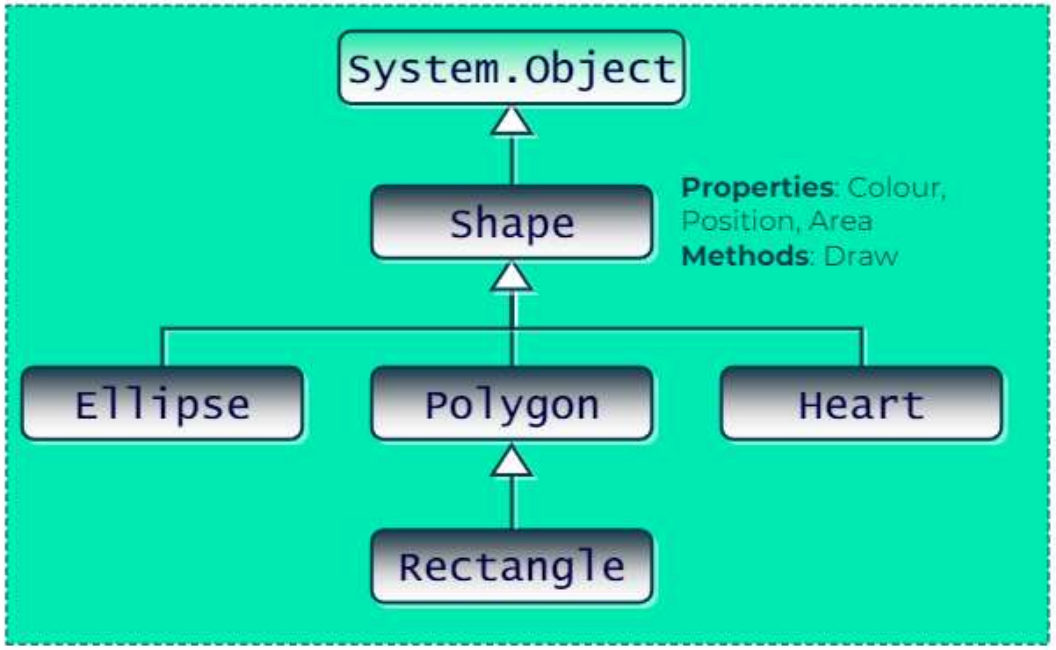
INHERITANCE

- **Inheritance** enables you to create new classes that reuse, extend, and modify the behaviour of other classes
- The class you inherit from is called the *base* class or *super* class
- The class that is being derived is called a *derived* or *sub* class
- Inheritance defines an 'is a kind of' relationship
- In C#, you can only inherit from one base class
- A derived class can be a base class for another class that forms a transitive relationship
  - If ClassA is a base class and ClassB inherits from ClassA, ClassB is the derived class and inherits the members of ClassA
  - If ClassC inherits from ClassB , then ClassC inherits the members of ClassB and ClassA
- Derived classes do not inherit *constructors* or *finalizers*

INHERITANCE EXAMPLE: GRAPHICS APPLICATION

***Scenario**: We need to be able to draw different shapes in our graphics application.*

- Different shapes have common **properties**. Each shape needs to be filled with a *colour* and has a *position* and an *area*
- Different shapes have common **behaviours**. Each shape needs to be able to *draw* itself



INHERITANCE EXAMPLE: GRAPHICS APPLICATION

Declare the base class: **Shape**

```
public class Shape {
  public Color Colour { get; set; }
  public Point Position { get; set; }
  //other Shape properties and methods
}
```

Define the derived class: **Polygon**

Use **derived : base** to specify an inheritance relationship

Add additional properties and methods as required

```
public class Polygon : Shape {
  public int NumberOfSides { get; set; }
}
```

INHERITANCE EXAMPLE: GRAPHICS APPLICATION

- A **Polygon** is a kind of **Shape**
- An **Ellipse** is a kind of **Shape**
- A **Rectangle** is a kind of **Polygon** and a kind of **Shape**
- A **Triangle** is a kind of **Polygon** and a kind of **Shape**

```
public class Polygon : Shape {
  public int NumberOfSides { get; set; }
}
```

```
public class Ellipse : Shape {
  //ellipse-specific properties and methods
}

public class Rectangle : Polygon {
  //rectangle-specific properties and methods
}

public class Triangle : Polygon {
  //triangle-specific properties and methods
}
```

DERIVED CONSTRUCTORS

```
public class Shape
{
    1 reference
    public Point Position { get; set; }
    1 reference
    public Color Colour { get; set; }
    //The only way to instantiate a Shape is to specify a colour and position
    //This is still true for derived classes
    1 reference
    public Shape(Point position, Color colour)
    {
        Position = position; Colour = colour;
    }
}
```

```
public class Ellipse : Shape
{
    1 reference
    public int XRadius { get; set; }
    1 reference
    public int YRadius { get; set; }
    2 references
    public Ellipse(Point position, Color colour, int xRadius, int yRadius)
      : base(position, colour) //chain to base class constructor
    {
        XRadius = xRadius;
        YRadius = yRadius;
    }
}
```

```
Ellipse e1 = new Ellipse((new Point(4, 7)), Color.Azure, 23, 34);

Ellipse e2 = new(new(4, 7), Color.Azure, 23, 34);
```

POLYMORPHISM

- **Polymorphism** is a Greek word meaning *'having many forms'*
- Polymorphism occurs because the *runtime* type of an object can be different to an object's *declared* type
- Objects of a *derived* type may be treated as objects of a *base* class in places, such as when passed as a parameter to a method, or when stored in a collection
- For example:
  - A **Rectangle** instance can be used anywhere a **Rectangle** type is expected
  - A **Rectangle** instance can be used anywhere a **Polygon** type is expected
  - A **Rectangle** instance can be used anywhere a **Shape** type is expected
  - A **Rectangle** instance can be used anywhere a **System.Object** type is expected

POLYMORPHISM SCENARIO

- The **Drawing** class needs to hold a collection of **Shapes** and be able to iterate over the collection to call each shape's *Draw* method

```
//A drawing has a collection of Shapes
0 references
public class Drawing
{
    private List<Shape> shapes;
    1 reference
    public List<Shape> Shapes
    {
        get
        {
        // null-coalescing assignment operator
         shapes ??= new List<Shape>();
         return shapes;
        }
    }
    0 references
    public void Draw(Graphics canvas)
    {
        foreach (Shape shape in Shapes) {
            shape.Draw(canvas); }
            // all shapes must have a Draw method
    }
}
```

POLYMORPHISM WITH VIRTUAL METHODS OR PROPERTIES

- Inherited methods and properties can be defined as **virtual**
- **Virtual** members can be **overridden** in the derived class
- This enables you to *generalise* the type of an object to its base class type, but have the compiler call the more *specialised* derived version of the member

```
public class Shape
{
    2 references
    public virtual int Area
    { get; }
}
```

```
public class Ellipse : Shape
{
    2 references
    public override int Area
    {
        get;
    }
}
```

```
Ellipse e = new Ellipse();
Shape s = e; // runtime type is Ellipse, declared type is Shape
Console.WriteLine(s.Area); // polymorphically gets Ellipse Area not Shape Area
```

MEMBER ACCESS MODIFIERS

Members (methods and properties) can be marked as:

- public
- private
- protected
- internal
- protected internal
- private protected

These define how users of the class or a derived class can access the members of that class.

INVOKING BASE CLASS FUNCTIONALITY

- A **derived** class can access **base** class members
- This avoids code duplication and having to have access to private fields
- To call a **base** class member, use the **base** keyword
- This calls the first matching member in the inheritance hierarchy

```
public class Shape
{
    2 references
    public virtual void Draw() { }
    1 reference
    public virtual void Draw(Graphics canvas) { }
    1 reference
    public Color Colour { get; set; }
    2 references
    public virtual int Area
    { get; }
}
```

```
public class Ellipse : Shape
{
    2 references
    public override void Draw()
    {
        base.Draw();// invoke base class method first, Shape.Draw()
        Brush br = new SolidBrush(base.Colour); // perform additional functionality
                                                // using the Colour property from the base class
                                                // Shape.Colour
    }
```

ABSTRACT CLASSES

- The **abstract** modifier indicates that an item has missing or incomplete implementation
- Use the **abstract** modifier in a **class** declaration to indicate that a class is intended to be *used only as a base class* for other classes
- **Abstract** classes can't be instantiated
- *Abstract members* within an *abstract* class must be implemented by non-abstract derived classes
- Derived classes receive:
  - Zero or more *concrete* methods/properties that they inherit
  - Zero or more *abstract* methods/properties that they inherit and must implement if they are a non-abstract class

ABSTRACT CLASSES EXAMPLE

An abstract class can contain abstract members:

```
public abstract class Shape
{
    // concrete properties and methods

    0 references
    public abstract void Draw();
    // abstract methods have no body
    // They must be overriden and implemented
    // in a non-abstract derived class
}
```

The abstract member must be implemented in a non-abstract derived class.

```
public class Rectangle : Shape
{
}
```

```
CS0534  'Rectangle' does not implement inherited abstract member 'Shape.Draw()'
```

Use the override keyword to implement the member:

```
public class Rectangle : Shape
{
    1 reference
    public override void Draw()
    {
        // implementation code goes here
    }
}
```

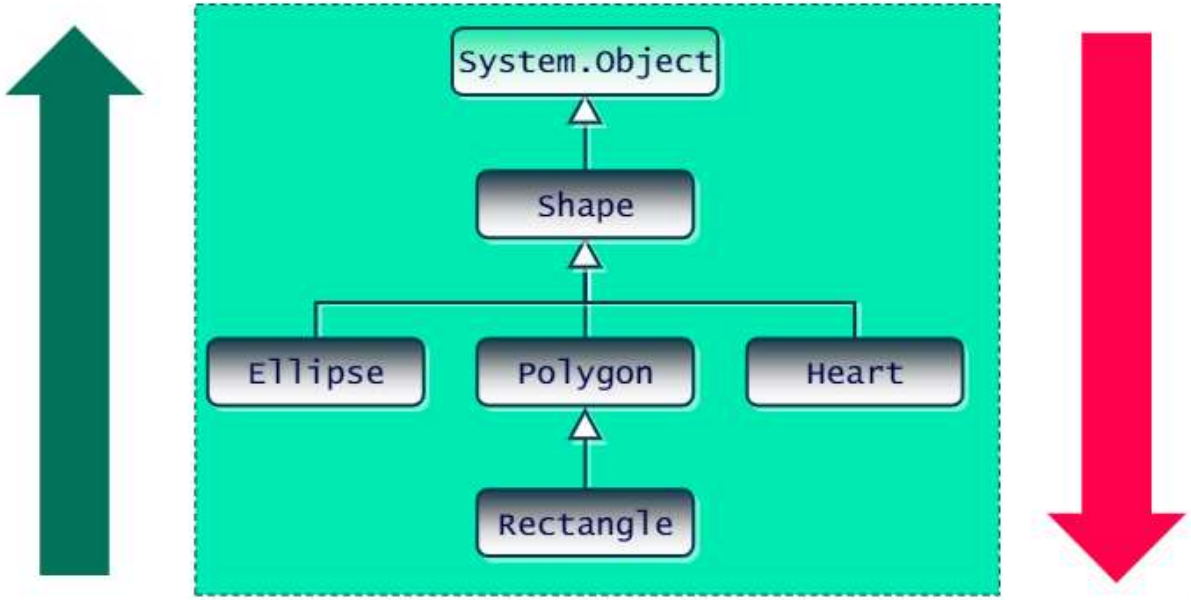
ABSTRACT MEMBERS

- Abstract *members* are declared using the **abstract** modifier and a *signature only*
- They do not contain any implementation code
- A class with even a single abstract member must be declared as abstract and cannot be instantiated
- Each derived class provides its own implementation for the abstract member or declares the inherited member as abstract and itself as an abstract class

```
public abstract class Shape
{
    // concrete properties and methods
    0 references
    public Color Colour { get; set; }

    // abstract method
    1 reference
    public abstract void Draw();

    // abstract property
    0 references
    public abstract double Area { get; }

    // abstract members have no body
    // They must be overriden and implemented
    // in a non-abstract derived class
}
```

CASTING DERIVED AND BASE CLASSES

- An object of a **derived** class can be treated as an object of a **base** class without explicit casting. This is known as an **up-cast** and is safe
- An object of a **base** type needs to be explicitly cast to be used as a **derived** type. This is known as a **down-cast** and is potentially unsafe



UP-CASTING AND DOWN-CASTING

```
Drawing app = new Drawing();
Ellipse e = new Ellipse();
Rectangle r = new Rectangle();
Heart h = new Heart();

app.Shapes.Add(e);// implicit upcasting
app.Shapes.Add(r);// implicit upcasting
app.Shapes.Add(h);// implicit upcasting

Shape s = app.Shapes[0];
s.Draw(); // an Ellipse is drawn

Ellipse ell = (Ellipse)s; // explicit downcasting
double circumference = ell.Circumference;

Console.WriteLine(circumference);
```

```
Drawing app = new Drawing();
Ellipse e = new Ellipse();
Rectangle r = new Rectangle();
Heart h = new Heart();

app.Shapes.Add(e);// implicit upcasting
app.Shapes.Add(r);// implicit upcasting
app.Shapes.Add(h);// implicit upcasting

Shape s = app.Shapes[2];
s.Draw(); // a Heart is drawn

Ellipse ell = (Ellipse)s; // explicit downcasting
double circumference = ell.Circumference;

Console.WriteLine(circumference);
```

Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type 'HelloWorld.Heart' to type 'HelloWorld.Ellipse'.'

View Details | Copy Details | Start Live Share session...

Exception Settings

SAFE DOWNCASTING

To prevent an **InvalidCastException** being thrown, you can use the following operators:

- is
- as

THE 'IS' OPERATOR

The **is** operator checks if the result of an expression is compatible with a given type or matches a pattern:

```
Shape shape = app.Shapes[0];

if (shape is Ellipse ellipse)// declaration pattern
{
    double circumference = ellipse.Circumference;
    Console.WriteLine(circumference);
}
```

```
if (shape is not null)// type pattern null check
{
    Console.WriteLine(shape.Area);
}
```

THE 'AS' OPERATOR

- The **as** operator explicitly converts the result of an expression to a given type
- If the conversion isn't possible, the **as** operator returns **null**

```
Ellipse? ellipse = shape as Ellipse;
if (ellipse != null)
{
    double circumference = ellipse.Circumference;
    Console.WriteLine(circumference);
}

if (ellipse is not null)
{
    double circumference = ellipse.Circumference;
    Console.WriteLine(circumference);
}
```

THE OBJECT CLASS

The ultimate base class of all .NET classes is **System.Object**.

A class implicitly inherits from **Object** if no base class is explicitly specified.

Object contains *virtual* methods that are commonly *overridden* in derived classes:

- Equals: Supports object comparisons
- Finalize: Performs clean-up before garbage collection
- GetHashCode: Generates a number to support the use of a hash table
- ToString: Provides a human-readable text string

OVERRIDING OBJECT METHODS EXAMPLE

```
public class Book
{
    5 references
    public string Title { get; set; }
    5 references
    public string Author { get; set; }
    3 references
    public long ISBN { get; set; }

    1 reference
    public override bool Equals(object? obj)
    {
        // If this and obj do not refer to the same type, then they are not equal.
        if (obj.GetType() != this.GetType()) return false;

        // Return true if Title and Author fields match.
        var other = (Book)obj;
        return (this.Title == other.Title) && (this.Author == other.Author);
    }
    1 reference
    public override int GetHashCode()
    {
        string hashString = ISBN.ToString()[..9];// get first 9 digits
        return int.Parse(hashString);
    }
    1 reference
    public override string ToString()
    {
        return $"Title={Title}, Author={Author}";
    }
}
```

OVERRIDING OBJECT METHODS EXAMPLE

```
Book b1 = new();
b1.Author = "J K Rowling";
b1.Title = "Harry Potter and the Philospher's Stone";
b1.ISBN = 9780590353403;

Book b2 = new();
b2.Author = "J K Rowling";
b2.Title = "Harry Potter and the Philospher's Stone";
b2.ISBN = 9780590353403;

Console.WriteLine(b1.ToString());// Title=Harry Potter and the Philospher's Stone, Author=J K Rowling
Console.WriteLine(b1.GetHashCode());// 978059035
Console.WriteLine(b1.Equals(b2));// True (value equality)
Console.WriteLine(b1 == b2);// False (reference equality)

Book b3 = b1;
Console.WriteLine(b1.Equals(b3));// True (value equality)
Console.WriteLine(b1 == b3);// True (reference equality)
```

SEALED CLASSES AND MEMBERS

- All classes can be inherited from unless the **sealed** modifier is applied
- All virtual members can be overridden anywhere within the inheritance hierarchy unless the sealed modifier is applied
- Structs are implicitly sealed and so cannot be inherited

SEALED CLASSES

ClassA is not sealed so can be used as a base class:

```
1 reference
public class ClassA
{
}
```

ClassB inherits from ClassA but is marked as **sealed**:

```
1 reference
public sealed class ClassB : ClassA
{
}
```

No classes are allowed to inherit from a sealed class:

```
0 references
public class ClassC : ClassB
{
}
```

class SealedMembers.ClassB

CS0509: 'ClassC': cannot derive from sealed type 'ClassB'

Show potential fixes (Alt+Enter or Ctrl+.)

SEALED METHODS

```
1 reference
public class ClassX
{
    2 references
    protected virtual void F1() { Console.WriteLine("X.F1"); }
    2 references
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

1 reference
class ClassY : ClassX
{
    1 reference
    sealed protected override void F1() { Console.WriteLine("Y.F1"); }
    2 references
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

0 references
class ClassZ : ClassY
{
    // Attempting to override F1 causes compiler error CS0239.
    2 references
    protected override void F1() { Console.WriteLine("Z.F1"); }

    // Overriding F2 is allowed.
    2 references
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

CS0239 'ClassZ.F1()': cannot override inherited member 'ClassY.F1()' because it is sealed

SUMMARY

- Inheritance
- Derived constructors
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators
- Overriding System.Object methods
- Sealed classes and members

Activity: Exercise 9

Interfaces

OUTLINE

- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces
- Default implementation
- Interface inheritance
- Member collisions
- Explicit implementation

INTERFACES

- An interface contains definitions for a group of related functionalities
- A class or struct that implements the interface must provide the implementation code for all members
- From C# 8.0, an interface member can provide a default implementation for a member
- A class or struct can implement many interfaces, whereas a class can only inherit from one base class and structs cannot inherit from any base classes
- Interfaces conventionally are identified with a capital 'I', followed by a verb that describes the group of functionalities e.g., IComparable, IControllable, IDisposable,, and IPlayable

DEFINING AN INTERFACE

- Define an interface using the **interface** keyword
- Interfaces can contain *methods*, *properties*, *indexers*, and *events*
- Interface members are implicitly **public** and **abstract**

```
public interface IDrawable
{
    1 reference
    void Draw(Graphics g);// no implementation code
}
```

IMPLEMENTING AN INTERFACE

- List the interfaces after a colon and the base class (if also using inheritance)
- All non-default members must be implemented

```
public abstract class Shape
{
    1 reference
    public abstract double Area { get; }
}
```

```
public interface IDrawable
{
    1 reference
    void Draw(Graphics g);// no implementation code
}
```

```
public class Rectangle : Shape, IDrawable
{
    1 reference
    public int Width { get; set; }
    1 reference
    public int Height { get; set; }
    1 reference
    public override double Area => Width * Height;

    1 reference
    public void Draw(Graphics g)
    {
        // implementation code goes here;
    }
}
```

POLYMORPHISM

- An interface defines a collection of related functionalities
- A class or struct that implements an interface 'can do' those functions, i.e., they play that role
- An interface type can be used as a method parameter, return type, or as the type in a generic collection
- Any implementing class or struct can be used where the interface type is expected

POLYMORPHISM EXAMPLE

```
Graphics canvas = new();

void ProcessDrawable(IDrawable id)
{
    if (id is not null)
    {
        id.Draw(canvas);
    }
}

List<Shape> shapes = new() { new Rectangle(), new Rectangle() };
foreach (Shape shape in shapes)
{
    if (shape is IDrawable s)
    {
        s.Draw(canvas);
        //or:
        ProcessDrawable(shape as IDrawable);
    }
}
```

MULTIPLE INTERFACES

- A class or struct can implement multiple interfaces

```
public interface IComparable<T> {
  int CompareTo(T obj);
}
```

```
public interface IDrawable
{
    1 reference
    void Draw(Graphics g);// no implementation code
}
```

```
public class Rectangle : Shape, IDrawable, IComparable<Rectangle>
{
    3 references
    public int Width { get; set; }
    1 reference
    public int Height { get; set; }
    1 reference
    public override double Area => Width * Height;

    0 references
    public int CompareTo(Rectangle? other)
    {
        return Width - other.Width;
    }
    3 references
    public void Draw(Graphics g)
    {
        // implementation code goes here;
    }
}
```

DEFAULT IMPLEMENTATION

- From C# 8.0, interface members can provide a default implementation
- **IEmployee** provides *default implementation* for the **GetTaxAmount** method

```
public interface IEmployee
{
    1 reference
    public int ID
    {
        get;
        set;
    }
    1 reference
    public string Name
    {
        get;
        set;
    }
    3 references
    public double Salary
    {
        get;
        set;
    }
    0 references
    public double GetTaxAmount()
    {
        return Salary * 0.05;
    }
}
```

```
public class Employee : IEmployee
{
    0 references
    public Employee(double salary)
    {
        Salary = salary;
    }
    1 reference
    public int ID { get; set; }
    1 reference
    public string Name { get; set; }
    3 references
    public double Salary { get; set; }
}
```

```
IEmployee employeeA = new Employee(2500);
Console.WriteLine($"The Tax amount is {employeeA.GetTaxAmount()}");
Console.ReadKey();

Employee employeeB = new Employee(2500);
Console.WriteLine($"The Tax amount is {employeeB.GetTaxAmount()}");
Console.ReadKey();
```

CS1061 'Employee' does not contain a definition for 'GetTaxAmount' and no accessible extension method 'GetTaxAmount' accepting a first argument of type 'Employee' could be found (are you missing a using directive or an assembly reference?)

INTERFACES CAN INHERIT FROM INTERFACES

- **IFullySteerable** inherits **ISteerable**
- Any implementing class or struct must implement all the members of both **ISteerable** and **IFullySteerable**

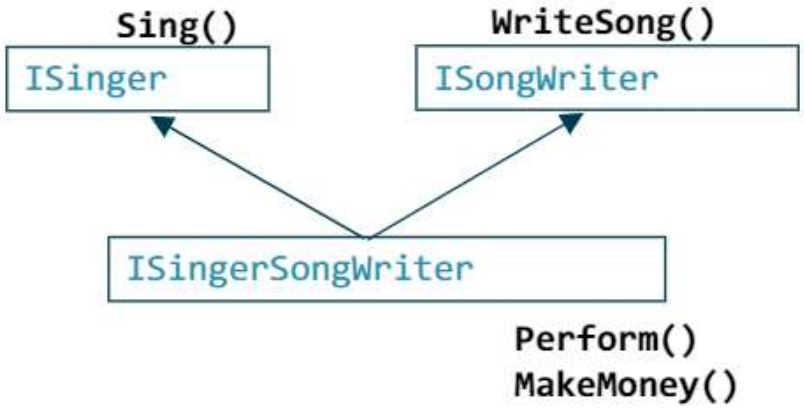
```
public interface ISteerable
{
    1 reference
    void TurnLeft();
    1 reference
    void TurnRight();
}
```

```
public interface IFullySteerable : ISteerable
{
    1 reference
    void GoUp();
    1 reference
    void GoDown();
}
```

```
public class Drone : IFullySteerable
{
    1 reference
    public void GoDown()
    {
        //  go down implementation;
    }
    1 reference
    public void GoUp()
    {
        //  go up implementation;
    }
    1 reference
    public void TurnLeft()
    {
        //  turn left implementation;
    }
    1 reference
    public void TurnRight()
    {
        //  turn right implementation;
    }
}
```

MULTIPLE INTERFACE INHERITANCE

- **ISingerSongWriter** inherits both **ISinger** and **ISongWriter**
- Implementing classes and structs must implement:
  - Sing
  - WriteSong
  - Perform
  - MakeMoney



MULTIPLE INTERFACE MEMBER COLLISIONS

- Member collisions can arise when multiple interfaces use the same member name for semantically different functionality
- You can only implement one version of the member using *implicit* interface implementation

```
public interface ICowboy
{
    0 references
    void Draw(Graphics g);
}
```

```
public interface IDrawable
{
    4 references
    void Draw(Graphics g);
}
```

```
public class CowboyShape : ICowboy, IDrawable
{
    4 references
    public void Draw(Graphics g)
    {
        // only one implementation;
    }
}
```

EXPLICIT INTERFACE IMPLEMENTATION

- You can implement interface members *explicitly*, which includes the interface name as part of the member name

```
public class CowboyShape : ICowboy, IDrawable
{
    3 references
    public void Draw(Graphics g)
    {
        // implicit drawable implementation;
        Console.WriteLine("Drawing a cowboy shape");
    }
    1 reference
    void ICowboy.Draw(Graphics g)
    {
        // explicit cowboy implementation;
        Console.WriteLine("Reach for the sky, mister!");
    }
}
```

```
CowboyShape cs = new CowboyShape();
cs.Draw(canvas);// Drawing a cowboy shape

IDrawable id = cs;
id.Draw(canvas);// Drawing a cowboy shape

ICowboy ic = cs;
ic.Draw(canvas);// Reach for the sky, mister!
```

SUMMARY

- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces
- Default implementation
- Interface inheritance
- Member collisions
- Explicit implementation

Activity: Exercise 10

Delegates and Lambdas

OUTLINE

- Delegates
- The Func delegate
- The Action delegate
- Lambda expressions
- The Predicate delegate
- Delegate and Lambda examples

DELEGATES

- A **delegate** is a type that represents references to methods with a particular parameter list and return type
- You can instantiate a delegate and associate it with any method with a compatible signature and return type
- You invoke the method through the delegate instance
- Delegates are used to pass methods as arguments to other methods

Useful built-in delegate types are:

- Func
- Action

THE FUNC DELEGATE

Func<> is a generic delegate which accepts zero or more input parameters and **one** return type.

The last parameter is the return type.

There are many overloads:

- **Func<TResult>** : Accepts zero parameters and returns a value of the type specified by the TResult parameter
- **Func<T, TResult>** : Accepts one parameter and returns a value of the type specified by the TResult parameter
- **Func<T1, T2, TResult>** : Accepts two parameters and returns a value of the type specified by the TResult parameter

Example: **Func<int, string, bool>** accepts two parameters of type **int** and **string** and returns a value of type **bool**.

THE ACTION DELEGATE

Action<> is a generic delegate which accepts zero or more input parameters and does **not** return a value.

There are many overloads:

- **Action** : Accepts zero parameters and does not return a value
- **Action<T>** : Accepts one parameter and does not return a value
- **Action<T1, T2>** : Accepts two parameters and does not return a value

Example: **Action<int, string, bool>** accepts three parameters of type **int, string,** and **bool** and does not return a value.

FUNC EXAMPLE

The **Add** method accepts *two* **int** parameters and has a return type of **int**.

It is an instance method (not static).

```
public class DelegateExamples
{
    1 reference
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

You can instantiate an object instance and use the **Func** generic delegate to encapsulate the **Add** method.

You invoke the **Add** method by invoking the **delegate** and passing the required parameters.

```
DelegateExamples examples = new();

Func<int, int, int> funcAdd = examples.Add;
Console.WriteLine(funcAdd(20, 40));
Console.WriteLine(funcAdd(2, 5));
```

ACTION EXAMPLE

Delegates can encapsulate static or instance methods:

```
public class DelegateExamples
{
    1 reference
    public void DisplayGreeting(string name)
    {
        Console.WriteLine($"Hello {name}");
    }

    1 reference
    public static void DisplayGreetingStatic(string name)
    {
        Console.WriteLine($"Hello {name}");
    }
}
```

```
DelegateExamples examples = new();

Action<string> funcHello = examples.DisplayGreeting;
funcHello("Everyone");
funcHello("World");

Action<string> funcHelloStatic = DelegateExamples.DisplayGreetingStatic;
funcHelloStatic("Everyone");
funcHelloStatic("World");
```

EXAMPLE PART 1

```
public class Book
{
    3 references
    public Book(string title, decimal price, int yearPublished)
    {
        Title = title;
        Price = price;
        YearPublished = yearPublished;
    }
    6 references
    public string Title { get; set; }
    2 references
    public decimal Price { get; }
    2 references
    public int YearPublished { get; }
```

```
static List<Book> FindCheapBooks(List<Book> books)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (book.Price < 15M)
        {
            output.Add(book);
        }
    }
    return output;
}
```

```
static List<Book> FindRecentBooks(List<Book> books)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (book.YearPublished > 2000)
        {
            output.Add(book);
        }
    }
    return output;
}
```

```
List<Book> books = new List<Book>() {
        new Book("Gulliver's Travels", 10M, 1726),
        new Book("War and Peace", 20M, 1869),
        new Book("This is going to hurt", 5M, 2017),
    };

List<Book> cheapBooks = FindCheapBooks(books);
List<Book> recentBooks = FindRecentBooks(books);
```

The two methods are almost identical and only differ by the conditional test

EXAMPLE PART 2

```
static List<Book> FindBooks(List<Book> books, Func<Book, bool> func)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (func(book))
        {
            output.Add(book);
        }
    }
    return output;
}
```

The two methods are consolidated into one and accept a **Func** delegate which can be used to encapsulate a matching method which contains the conditional test.

```
// these methods accept a Book parameter and return a bool
static bool CheapBook(Book book)
{
    return book.Price < 15M;
}
static bool RecentBook(Book book)
{
    return book.YearPublished > 2000;
}
```

These two methods match the **Func** delegate signature and contain the conditional tests.

```
List<Book> books = new List<Book>() {
        new Book("Gulliver's Travels", 10M, 1726),
        new Book("War and Peace", 20M, 1869),
        new Book("This is going to hurt", 5M, 2017),
    };

// Call FindBooks passing a method that matches the Func<Book, bool> signature
List<Book> cheapBooks = FindBooks(books, CheapBook);
List<Book> recentBooks = FindBooks(books, RecentBook);
```

LAMBDAS

- A **lambda** expression is used to create an anonymous function
- You use the lambda declaration operator **=>** to separate the lambda's parameter list from its body
- A lambda expression can be either an expression lambda (single line) or a statement lambda (multiple lines enclosed in braces)
- Specify input parameters on the left side of the lambda operator or use empty brackets if there are zero parameters
- If there is only one parameter, brackets are optional
- Any lambda expression can be converted to a delegate type

```
Action line = () => Console.WriteLine();// zero parameters
Func<double, double> cube = x => x * x * x; // one parameter x
Func<int, int, bool> testForEquality = (x, y) => x == y; // two parameters x and y
```

FUNC EXAMPLE AS A LAMBDA

This example uses a **delegate** that encapsulates a *named* Add method.

```
public class DelegateExamples
{
    1 reference
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

```
DelegateExamples examples = new();

Func<int, int, int> funcAdd = examples.Add;
Console.WriteLine(funcAdd(20, 40));
Console.WriteLine(funcAdd(2, 5));
```

This example uses a **lambda** expression to encapsulate an *anonymous* method.

```
Func<int, int, int> add = (x, y) => x + y;
Console.WriteLine(add(20, 40));
Console.WriteLine(add(2, 5));
```

ACTION EXAMPLE AS A LAMBDA

These examples use **delegates** that encapsulate *named* methods:

```
public class DelegateExamples
{
    1 reference
    public void DisplayGreeting(string name)
    {
        Console.WriteLine($"Hello {name}");
    }

    1 reference
    public static void DisplayGreetingStatic(string name)
    {
        Console.WriteLine($"Hello {name}");
    }
}
```

```
DelegateExamples examples = new();

Action<string> funcHello = examples.DisplayGreeting;
funcHello("Everyone");
funcHello("World");

Action<string> funcHelloStatic = DelegateExamples.DisplayGreetingStatic;
funcHelloStatic("Everyone");
funcHelloStatic("World");
```

These examples use **lambda** expressions to encapsulate *anonymous* methods:

```
Action<string> greet = (string name) => Console.WriteLine($"Hello {name}");
greet("Everyone");

Action<string> greet2 = name => Console.WriteLine($"Hello {name}");
greet2("World");
```

EXAMPLE PART 3

```
static List<Book> FindBooks(List<Book> books, Func<Book, bool> func)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (func(book))
        {
            output.Add(book);
        }
    }
    return output;
}
```

The two methods are consolidated into one and accept a **Func** delegate, which can be used to encapsulate a matching method which contains the conditional test.

Named methods are no longer required since the conditional tests are now defined as **lambda** expressions which match the Func<Book, bool> delegate signature.

```
List<Book> books = new List<Book>() {
        new Book("Gulliver's Travels", 10M, 1726),
        new Book("War and Peace", 20M, 1869),
        new Book("This is going to hurt", 5M, 2017),
    };

// Call FindBooks passing a lambda expression that matches the Func<Book, bool> signature
List<Book> cheapBooks = FindBooks(books, b => b.Price < 15M);
List<Book> recentBooks = FindBooks(books, b => b.YearPublished > 2000);
```

THE PREDICATE DELEGATE

Predicate<> is a generic delegate which accepts one parameter and a return type of bool.

Example: **Predicate<int>** accepts one parameter of type **int** and returns a value of type **bool**.

```
Predicate<int> oldEnough = a => a >= 21;

Console.WriteLine(oldEnough(22)); //True
Console.WriteLine(oldEnough(18)); //False


Predicate<Book> isCheapBook = b => b.Price <= 5M;

Book book = books[2];
string isCheap =  isCheapBook(book) ? " is " : " is not ";
Console.WriteLine(book.Title + isCheap + "a cheap book");
// This is going to hurt is a cheap book
```

DELEGATE/ LAMBDA EXAMPLES

```
// Arrays and Lists have many methods that use a Predicate delegate
// Find uses a Predicate delegate
Book? recentBook = books.Find(book => book.YearPublished >= 2015);
if (recentBook != null)
{
    Console.WriteLine(rec
}

// FindAll uses a Predica
List<Book> startsWithW = 
```

Book? List<Book>.Find(Predicate<Book> match)

Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.

Returns:

The first element that matches the conditions defined by the specified predicate, if found; otherwise, the default value for type T.

Exceptions:

ArgumentNullException

```
// Arrays and Lists have many methods that use a Predicate delegate
// Find uses a Predicate delegate
Book? recentBook = books.Find(book => book.YearPublished >= 2015);
if (recentBook != null)
{
    Console.WriteLine(recentBook.Title);
}

// FindAll uses a Predicate delegate
List<Book> startsWithW = books.FindAll(book => book.Title.StartsWith('W'));

// ForEach uses an Action delegate
startsWithW.ForEach(book => global::System.Console.WriteLine(book.Title));
//output: War and Peace

// Average uses a Func<T, decimal> delegate
decimal averagePrice = books.Average(b => b.Price);
Console.WriteLine(averagePrice);
//output: 11.66666667
```

SUMMARY

- Delegates
- The Func delegate
- The Action delegate
- Lambda expressions
- The Predicate delegate
- Delegate and Lambda examples

Activity: Exercise 11

Language Integrated Query (LINQ)

OUTLINE

- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Forcing immediate execution
- Joins
- Group join
- Aggregations
- The let clause
- The OfType method
- LINQ expression syntax and keyword reference

LINQ

- Language-Integrated Query (**LINQ**) is a set of technologies based on the integration of query capabilities directly into the C# language
- You can use LINQ to consistently query data from Objects, collections that support IEnumerable, relational databases and XML, all using C# syntax
- LINQ queries are written using *query expression syntax* or *method syntax*
- Some queries can only use the method syntax (e.g., Count and Max)
- A query is **not** executed until you iterate over the query variable

SCENARIO: NON-LINQ AND LINQ EQUIVALENT

Compare the two code snippets.

The left-hand snippet does **not** use LINQ.

The right-hand snippet uses LINQ.

```
// Non-LINQ Example

// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Create a List to store the high scores
List<int> highScores = new();

// Iterate over the array to find the
// scores above 80 and add to the List
foreach (int score in scores)
{
    if(score > 80)
    {
        highScores.Add(score);
    }
}
// Iterate over the highScores List
foreach(int i in highScores)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

```
// LINQ

// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query expression
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

LINQ SYNTAX

Query expression syntax:

```
// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query expression
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

Query method syntax:

```
// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query using method syntax
var query = scores.Where(score => score > 80)
                  .Select(score => score);

// Execute the query
foreach (var i in query)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

LINQ SYNTAX EXAMPLE

Query expression syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using query expression syntax
var queryExpression = from c in customers
                      where c.City == "London"
                      orderby c.Balance
                      select c.CustomerName;

// Execute the query
foreach (var c in queryExpression)
{
    Console.WriteLine(c);
}
```

Query method syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using method syntax
var queryMethod = customers.Where(c => c.City == "London")
                           .OrderBy(c => c.Balance)
                           .Select(c => c.CustomerName);

// Execute the query
foreach (var c in queryMethod)
{
    Console.WriteLine(c);
}
```

LINQ PROJECTIONS

You can explicitly specify the query type if you project (select) a whole class e.g., Customer.

```
// Return type is IEnumerable<Customer>
IEnumerable<Customer> queryA = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select c; // Customer is returned
```

If you project only some of the object's properties, the compiler will generate a new anonymous type.

Use the **var** keyword:

```
// Return type is IEnumerable<<anonymous type: string customerName, string City>>
IEnumerable<Customer> queryB = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select new { c.CustomerName, c.City }; // CustomerName and City are returned

// use var
var queryC = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select new { c.CustomerName, c.City }; // CustomerName and City are returned
```

DEFERRED EXECUTION

A LINQ query is not executed until you iterate over the query variable in a foreach statement.

This allows the query to retrieve different data each time it is executed:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

var query = from s in names
            where s.Length == 5
            select s;

foreach (string s in query)
{
    Console.Write(s + " ");
}
// Output: Tommy Fiona Bobby


names[0] = "Susie";

foreach (string s in query)
{
    Console.Write(s + " ");
}
// Output: Susie Fiona Bobby
```

FORCING IMMEDIATE EXECUTION: AGGREGATE FUNCTIONS

- Queries that perform aggregation functions over a range of source elements must first iterate over those elements, therefore they execute **without** an explicit foreach statement
- These types of queries return a single value not an IEnumerable collection

```
int[] scores = { 97, 92, 81, 60, 40, 54, 80, 75 };

var failingScores =
    from score in scores
    where score < 80
    select score;

int countFailingScores = failingScores.Count();
Console.WriteLine("Number of failing scores is " + countFailingScores);

double avgFailingScores = failingScores.Average();
Console.WriteLine("Average of failing scores is " + avgFailingScores);
```

FORCING IMMEDIATE EXECUTION: TOLIST OR TOARRAY

- To force execution of any LINQ query and cache its results, you can call the **ToList** or **ToArray** methods:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

var query = (from s in names
             where s.Length == 5
             select s).ToList();//force execution

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby

names[0] = "Susie";

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby
```

JOINS

The **join** clause is used to match elements in one collection with elements in another collection.

Find persons who have names that match the 'names' list:

```
string[] names = { "Vinita", "Pete" };

var people = new List<Person> {
    new Person("Pete",20),
    new Person("Rafael",50),
    new Person("Vinita",32),
    new Person("Pete",37),
    new Person("Tom",40),
};


var matchingPeople = from n in names
                     join p in people
                     on n equals p.Name
                     select p;

foreach (var p in matchingPeople)
{
    Console.WriteLine("{0}, {1}", p.Name, p.Age);
}
```

```
Vinita, 32
Pete, 20
Pete, 37
```

GROUP JOIN

The **group join** method is useful for producing hierarchical data structures.

It pairs each element from the first collection with a set of correlated elements from the second collection

The query expression **join ... into** translates to an invocation of **GroupJoin**

```
var groupJoined = from n in names
                  join p in people
                  on n equals p.Name into matchingNames
                  select new { Name = n, Persons = matchingNames };

foreach (var group in groupJoined)
{
    Console.WriteLine(group.Name);
    foreach (Person p in group.Persons)
    {
        Console.WriteLine("..." + p.Age);
    }
}
```

```
Vinita
...32
Pete
...20
...37
```

AGGREGATIONS

- Aggregations execute without an explicit foreach statement
- The **baseQuery** uses lambda expressions to specify which property of Customer is to be aggregated
- **Qry2** projects the Balance property which is a decimal, so no lambdas are required

```
var baseQuery = from c in customers
                where c.City == "London"
                orderby c.Balance
                select c;// Customer

decimal avg = baseQuery.Average(c => c.Balance);
decimal max = baseQuery.Max(c => c.Balance);
decimal total = baseQuery.Sum(c => c.Balance);


IEnumerable<decimal> qry2 = from c in customers
                            where c.City == "London"
                            select c.Balance;// Decimal
avg = qry2.Average();
max = qry2.Max();
total = qry2.Sum();
```

LET CLAUSE

- The **let** clause enables you to store the result of a sub-expression in order to use it in subsequent clauses

```
// Specify the data source
string[] strings =
{
    "Been there, done that.",
    "All Greek to me.",
    "A piece of cake."
};

// Split the sentence into an array of words
// and select those whose first letter is a vowel.
var query =
    from sentence in strings
    let words = sentence.Split(' ')
    from word in words
    let w = word.ToLower()
    where w[0] == 'a' || w[0] == 'e'
        || w[0] == 'i' || w[0] == 'o'
        || w[0] == 'u'
    select word;

// Execute the query.
foreach (var v in query)
{
    Console.WriteLine("\"{0}\" starts with a vowel", v);
}

/* Output:
"All" starts with a vowel
"A" starts with a vowel
"of" starts with a vowel
*/
```

OFTYPE

The **OfType** method filters the elements of an IEnumerable based on a specified type:

```
public class Mammal
{
    0 references
    public string Name { get; set; }
}
```

```
public class Dog : Mammal
{
}
```

```
public class Cat : Mammal
{
}
```

```
List<Mammal> mammals = new() {
    new Dog() {Name="Rover"},
    new Cat() {Name="Tiddles"},
    new Dog() {Name="Snowy"}
};
var dogs = mammals.OfType<Dog>();

foreach (var dog in dogs)
{
    Console.WriteLine(dog.Name);
}

/* Output:
Rover
Snowy
*/
```

LINQ QUERY KEYWORDS & METHODS

```
from
join .. on
 equals
where
group .. by
into
let
orderby
 descending
select
```

```
Join
Where
GroupBy
OrderBy
OrderByDescending
ThenBy
ThenByDescending
Select
```

```
Concat
Distinct
Except
Intersect
Union
GroupJoin
Reverse
SelectMany
SequenceEqual
Union
```

```
Contains
DefaultIfEmpty
ElementAt
Single
```

```
All
Any
```

```
AsEnumerable
Cast
OfType
```

```
ToArray
ToDictionary
ToList
ToLookup
```

```
Aggregate
Average
Count
LongCount
Max
Min
Sum
```

```
First
Last
Skip
SkipWhile
Take
TakeWhile
```

SUMMARY

- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Forcing immediate execution
- Joins
- Group join
- Aggregations
- The let clause
- The OfType method
- LINQ expression syntax and keyword reference

Activity: Exercise 12

- Convert non-LINQ code into the LINQ equivalent

Exception Handling

OUTLINE

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

EXCEPTION HANDLING

The **exception handling** features of the C# language let you deal with any unexpected or exceptional situations that occur whilst your code is running

There are four keywords used:

- **Try**: Try actions that may not succeed
- **Catch**: Handle failures
- **Finally**: Clean up resources
- **Throw**: Generate an exception

EXAMPLE: TRY CATCH FINALLY

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //do things with connection
}
catch (SqlException ex)
{
    Console.WriteLine("Data access error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("General error: " + ex.Message);
}
finally
{
    if (conn != null && conn.State == ConnectionState.Open)
    {
        conn.Close();
    }
}

//remainder of method
```

QA Understanding Execution Flow: 1

```
public class Program {
1   static void Main() {
2       try {
            Task.F1( 0 );
            Task.F2();
        }
3       catch (Exception ex)
4       {
5           Console.WriteLine(ex.Message);
6       }
7   }
}
```

```
public class Task {
11  public static void F1(int a) {
12      F3(a);
        F4();
    }

    public static void F2() { }

31  public static void F3(int y) {
32      int x = 10 / y;
        // Does not run
    }
    public static void F4() { }
}
```

Step

Understanding Execution Flow: 2

```
public class Program {
1   static void Main() {
2     try {
3       Task.F1( 0 );
4       Task.F2();
      }
      catch (Exception ex) {
          Console.WriteLine(
            ex.Message);
      }
    }
55
```

```
public class Task {
11    public static void F1(int a) {
12        F3(a);
13        F4();
14    }

22    public static void F2() { }

31    public static void F3(int y) {
32        int x;
33        try {
34            x = 10 / y;
              // Does not run
          }
35        catch (DivideByZeroException ex)
36        { }
37        // Rest of method
      }
41    public static void F4() { }
}
```

Step

Understanding Execution Flow: 3

```
public class Program {
1   static void Main() {
2     try {
3       Task.F1( 0 );
        Task.F2();
      }
4     catch (Exception exn)
5     {
6       Console.WriteLine(
          exn.Message);
7     }
8   }
}
```

```
public class Task {
11    public static void F1(int a) {
12        F3(a);
          F4();
      }

      public static void F2() { }

31    public static void F3(int y) {
32        int x;
32        try {
33            x = 10 / y;
              Console.WriteLine(x);
          }
34        finally {
35            Console.WriteLine("Other");
36        }
          // does not run if try fails
      }
      public static void F4() { }
}
```

Step

THROWING EXCEPTIONS

To generate an exception, you throw a reference to an exception object:

```
void PrintReport(Report rpt) {
    if (rpt == null) {
        throw new ArgumentNullException
            ("rpt", "Can't print a null report");
    }
    …
}
```

You can re-throw a caught exception which maintains the original stack trace:

```
catch (ArgumentNullException ex) {
    …
    throw;
}
```

CUSTOM EXCEPTIONS

Derive the class from **System.Exception**

```
public class CarFactoryException : Exception
{
    0 references
    public CarFactoryException() { }
    // Other overloaded constructors / properties / fields
}

1 reference
public class InvalidModelException : CarFactoryException
{
    0 references
    public InvalidModelException()
    {
    }
}
```

To provide rich exception details:

- Overload the constructor to pass in information
- Provide public properties to allow retrieval

FILTERED EXCEPTIONS

- **User-filtered exception handlers** catch and handle exceptions based on requirements you define
- Use the **when** keyword with the **catch** statement:

```
try
{
    throw new MyException() { MinorFault = true };
}
catch (MyException ex)
{
    if (!ex.MinorFault)
    {
        throw;
    }
    Console.WriteLine("deal with minor fault");
}


try
{
    throw new MyException() { MinorFault = false };
}
catch (MyException ex) when (ex.MinorFault)
{
    Console.WriteLine("deal with minor fault");
}
catch (MyException ex) when (ex.MinorFault == false)
{
    Console.WriteLine("deal with major fault");
    throw;
}
```

INNER EXCEPTIONS

- The **Exception** class defines an **InnerException** property that enables you to wrap a custom exception around a system exception, whilst maintaining traceability as to the original cause of the exception
- Your custom Exception type requires a constructor that accepts an inner exception
- The **Message** and **InnerException** properties are read-only so call the base class constructor to set their values

```
public class InvalidPaintJobException : Exception
{
    0 references
    public InvalidPaintJobException()
    {
    }
    1 reference
    public InvalidPaintJobException(string message, Exception inner): base(message, inner)
    {
    }
}
```

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //look up the required paint colour
}
catch (SqlException ex)
{
    throw new InvalidPaintJobException(
        message: "not a valid colour spec",
        inner: ex);
}
```

BEST PRACTICES

- Use specific catch blocks for the exceptions you expect within the code
- Include a last catch block to catch **System.Exception** which will catch unexpected exceptions
- Not every method needs **try** and **catch** blocks since exceptions are propagated up the call stack
- Use **finally** blocks to tidy up resources
- Only **throw** exceptions if the situation is exceptional rather than expected
- Ensure your tests check that exceptions are thrown when expected
- Do not disclose sensitive or too much information in error messages

SUMMARY

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

Activity: Exercise 13

XUNIT AND EXCEPTIONS

- xUnit enables you to test when an exception is thrown specifically within the *Act* stage of your test, as opposed to the *Arrange* or *Assert* stage
- Use Assert.Throws<TException> passing a lambda statement to perform the action you are testing
- Assert.Throws returns the exception so you can access any properties and make further assertions on those

```
[Fact]
public void Total_Price_Never_Negative()
{
    Checkout checkout = new Checkout(new TestDiscount());
    order.Add(new Pizza(Size.Small_10, Crust.Regular_2));

    Assert.Throws<NegativePriceException>(() => checkout.GetBestPrice(order));
}
```