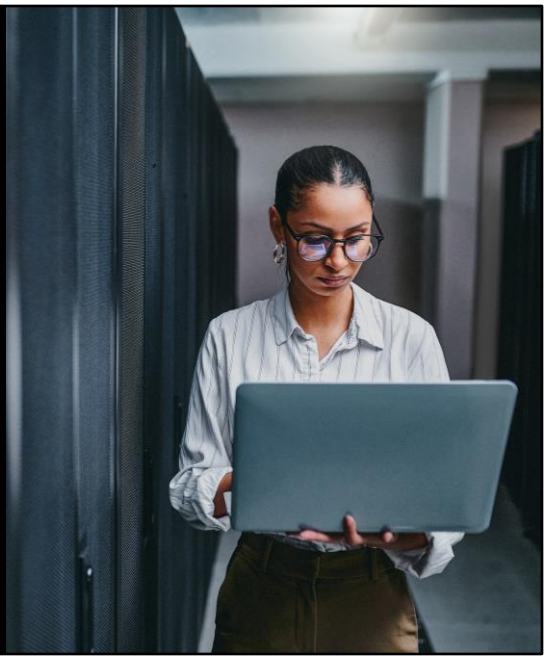
01 Course Introduction

C# The Programming Language Part 1

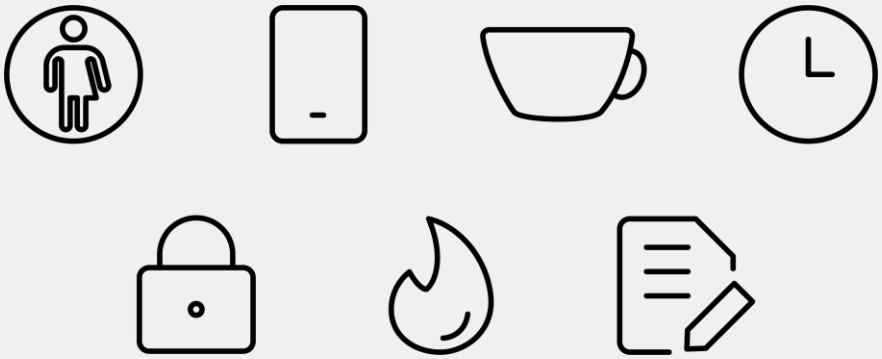
QA

Learn. To Change.




QA

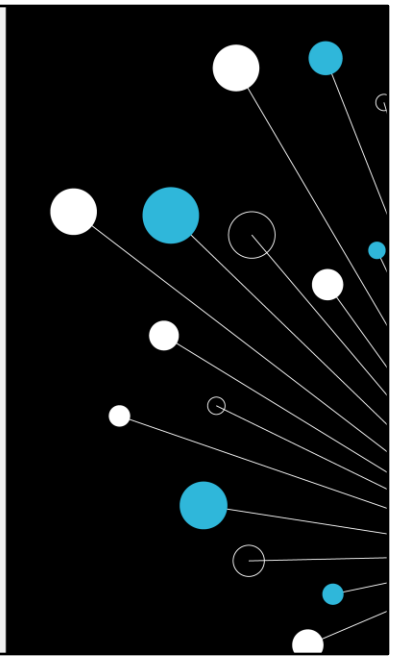
Housekeeping



QA

Learning objectives

- Gain a solid understanding of C# language fundamentals including
  - Variable declaration and datatypes
  - Functions and parameter passing
  - Flow Control
  - Loops
- Be familiar with collections and their specialisms
- Gain a basic understanding of object-oriented programming (OOP) to the level of defining classes, methods and properties.
- Be able to create simple unit tests
- Navigate the Visual Studio Integrated Development Environment (IDE) and make use of its code completion features

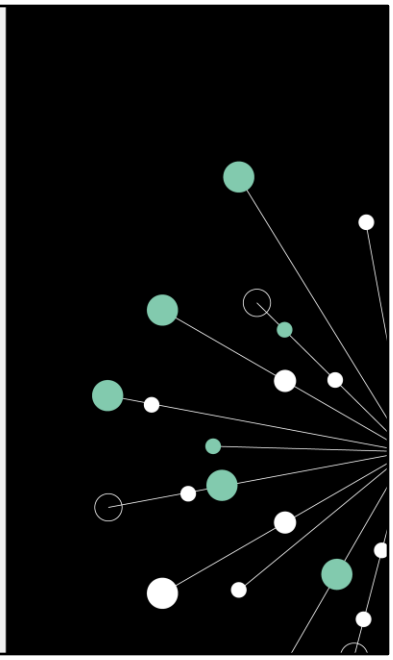


QA

Course Outline

- Introduction to C# and .NET
- Variables, Basic Syntax and Structure
- Functions
- Flow Control
- Arrays, Collections and Loops
- Object-Oriented Programming (OOP) Concepts
- Properties and Constructors
- Unit Testing

QA



Contents

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines

QA

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.

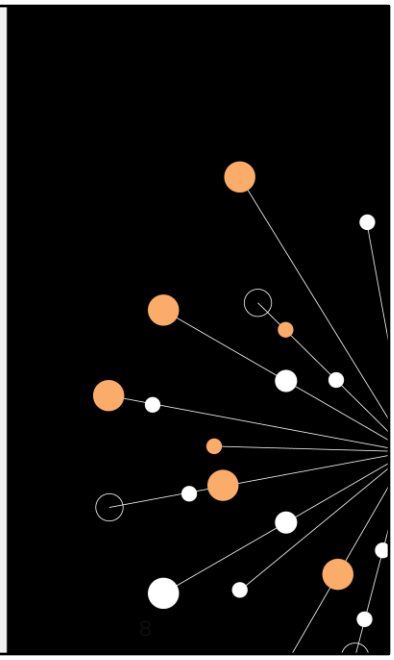
QA

7

This course is designed to teach you how to write object oriented programs for .NET using the C# programming language. To get the most out of this course, it is important that you already have some prior experience in programming, even if it is just creating macros in Excel or writing JavaScript for web pages. This level of knowledge will help when it comes to understanding how to construct the code to actually perform the requisite tasks in C#.

Activity: Introductions

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



QA

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

QA

Allocation of Virtual Machines

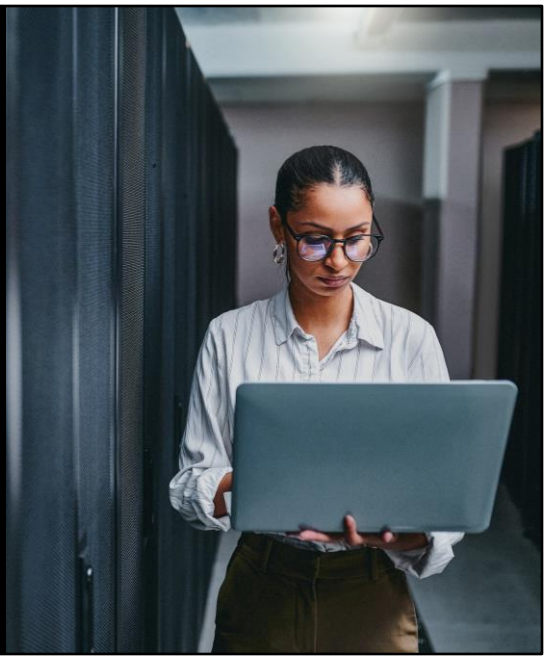
Lab instructions assume you are using Microsoft Visual Studio (Community Edition). You are quite welcome to use a local installation. However, do be aware a later lab makes use of SQL Server and SQL Server Management Studio (SSMS).

For this reason, your tutor can allocate you a virtual machine which has virtual machine has all the necessary software that you need for the course already installed on it.

02 Introduction to C# and .NET

C# The Programming Language Part 1

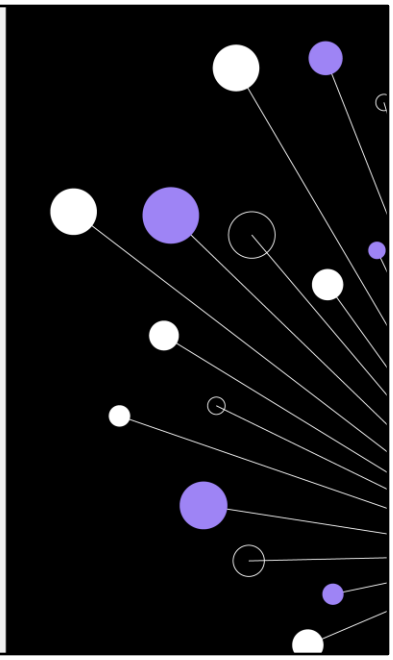
QA
Learn. To Change.



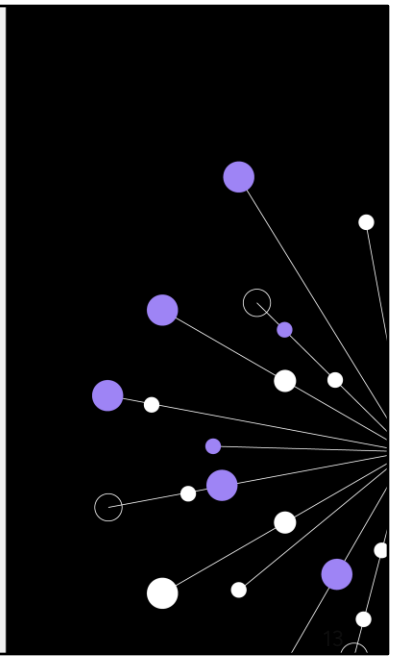
Learning objectives

- Understanding the C# language and its role within .NET
- The structure of a C# program
- Compiling and running C# applications
- Setting up the development environment (Visual Studio/)

QA



Contents



QA

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
- .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

QA

Hello World - Alternative

With .NET 3 (and C# 8) and earlier, a console "Hello World" application must contain 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 . . .
```

QA

The above code still works with all versions of C# and .NET and is often preferred by some developers.

Hello World Explained – Part 1

using System

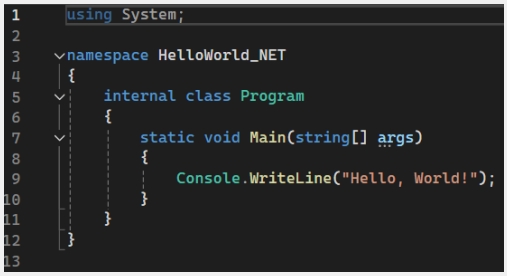
Use the System library to access useful functions such as the Console class's WriteLine method, without having to use its fully-qualified name

namespace HelloWorld_NET

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



QA

Hello World Explained – Part 2

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

```
using System;

namespace HelloWorld_NET
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}

```

QA

Top-level Statements

When you use top-level statements such as the .NET 8 (C# 12) 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 code at the top right to that on the bottom right.

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

```
using System;

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

QA

Starting in .NET 5 (C# 9), you don't have to explicitly include a Main method in a Console application project. Instead, you can use the top-level statements feature to minimise the code you have to write. In this case, the compiler generates a Program class and a Main method entry-point for the application.

Only one file with top-level statements is allowed per project.

A file with top-level statements can also contain namespaces and type definitions, but they must come after the top-level statements.

Note: The entry-point that the compiler synthesizes isn't actually called 'Main' and the signature generated depends on whether your top-level statements include keywords such as await and return.

Referring to namespaces

Option A:

Use the fully-qualified name

```
1  namespace HelloWorld_NET5
2  {
       0 references
3      internal class Program
4      {
           0 references
5          static void Main(string[] args)
6          {
7              Console.WriteLine("This will not work without the using directive");
8              System.Console.WriteLine("This is fully qualified and will work");
9          }
10     }
11 }
```

Option B:

Issue a 'using' directive

```
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("This will work because of the using directive");
10             System.Console.WriteLine("This is fully qualified but verbose");
11         }
12     }
13 }
```

QA

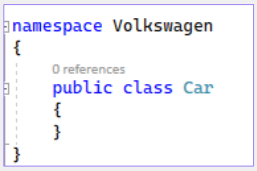
To use types from other assemblies (executables or dynamic link libraries), you must first add a reference to that assembly.

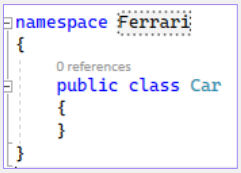
Creating namespaces

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

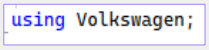
Below there are two classes, both called '*Car*'.

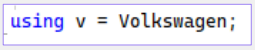
They can be disambiguated using their fully-qualified names of **Volkswagen.Car** and **Ferrari.Car**





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





QA

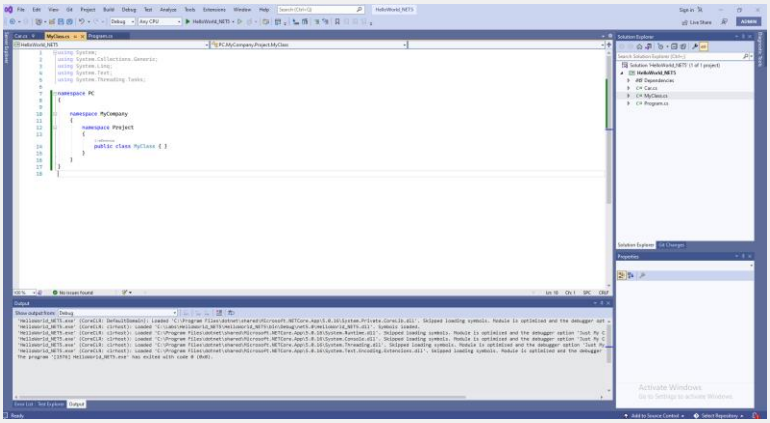
Namespaces can be nested.

```
namespace PC
{
    namespace MyCompany
    {
        namespace MyProject
        {
            internal class MyClass
            {

            }
        }
    }
}
```

Use an alias to shorten a long namespace.

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

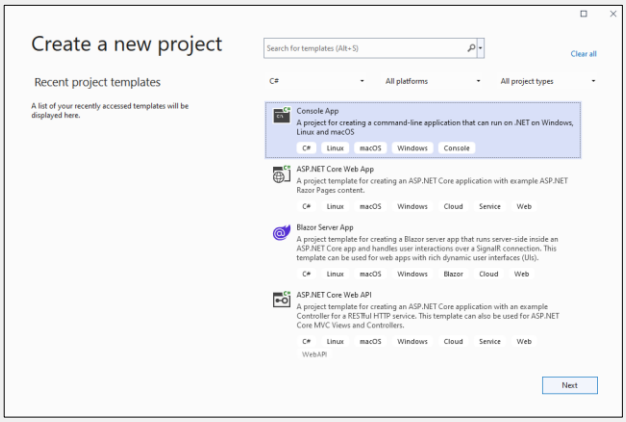
Visual Studio



QA

Visual Studio is the integrated development environment (IDE) you will be using on this course.

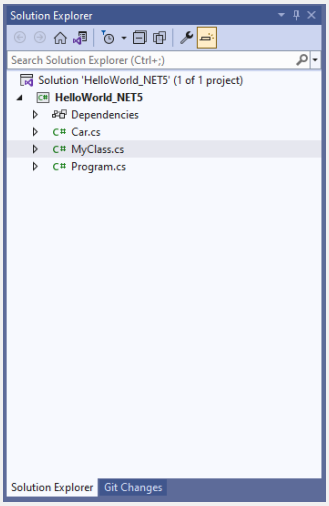
Visual Studio Project Templates



QA

You can create many different types of project for .NET.

Visual Studio: Solution Explorer



QA

Solution Explorer displays your Solutions, Projects and Files. C# files have a **.cs** file extension.
A solution is a group of projects. The project whose name is in bold is the starter project and will act as the entry-point for your application when you run your code.

Visual Studio: Menu and Toolbar

QA

There are many context-sensitive menu and toolbar actions.

Commonly performed menu actions are:
File -> New
File -> Open
File -> Save

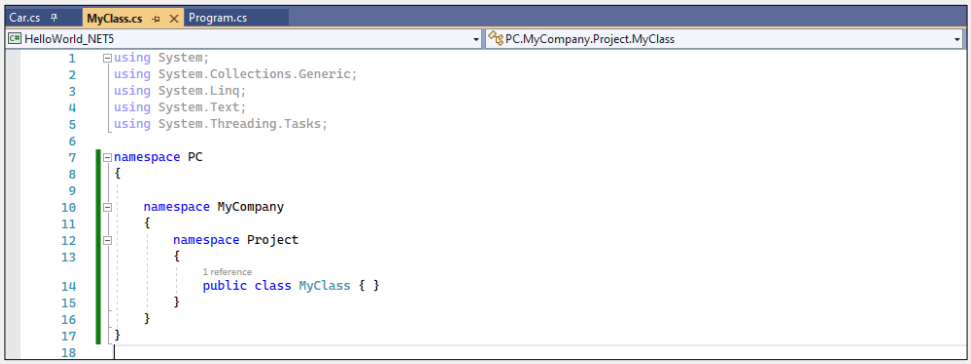
Build -> Build Solution

Debug -> Start Debugging

Test -> Test Explorer
Test -> Run All Tests

Edit -> Advanced -> Comment Selection
Edit -> Advanced -> Uncomment Selection

Visual Studio: Code Editors



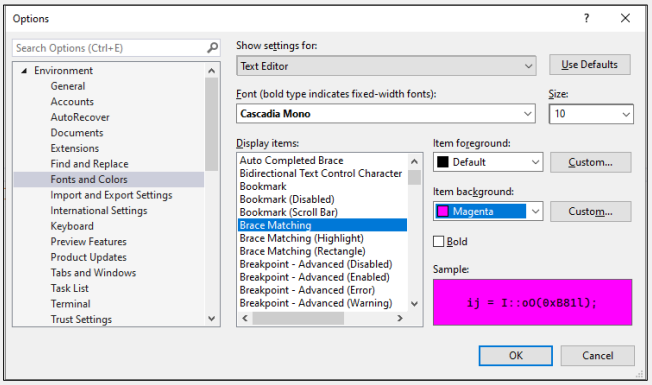
QA

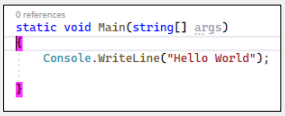
You will write and edit your code in the code editor window.

Visual Studio has a tool called IntelliSense that will provide assistance and feedback as you work on your code.

Brace Matching

Tools -> Options -> Fonts and Colors





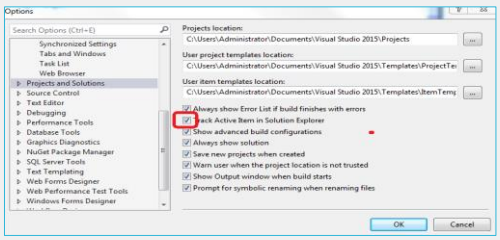
QA

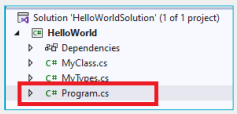
C# uses the C language style of syntax which includes wrapping code into blocks using braces. To make the pairs of braces more visible, you can set the brace matching colour using the Tools menu.

Tools -> Options -> Fonts and Colors -> Brace Matching

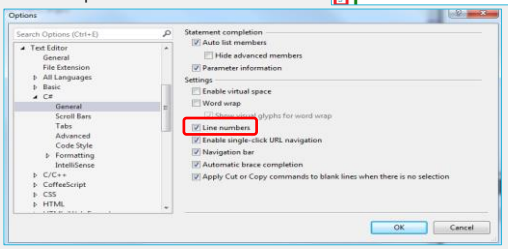
Tracking and Line Numbers

Tools -> Options -> Projects and Solutions





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



QA

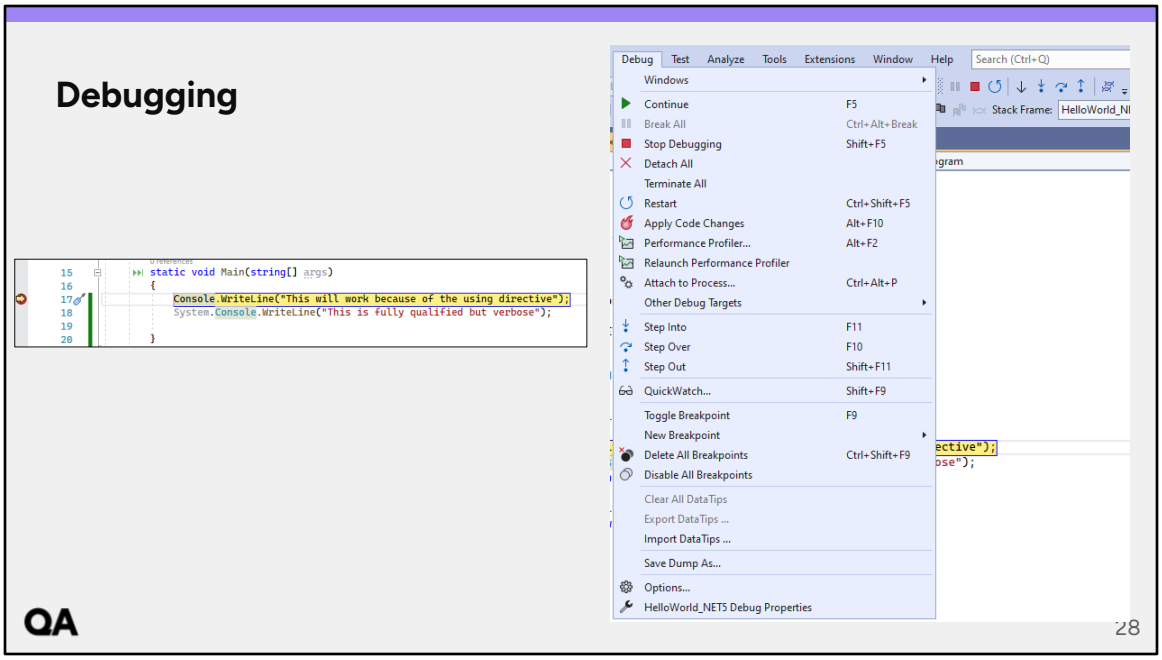
To make the filename of the file being edited more prominent in the Solution Explorer window, you can set to track the active item using the Tools menu:

Tools -> Options -> Projects and Solutions -> Track Active Item in Solution Explorer

To display line numbers in the code editor window:

Tools -> Options -> Text Editor-> C# -> General -> Line numbers

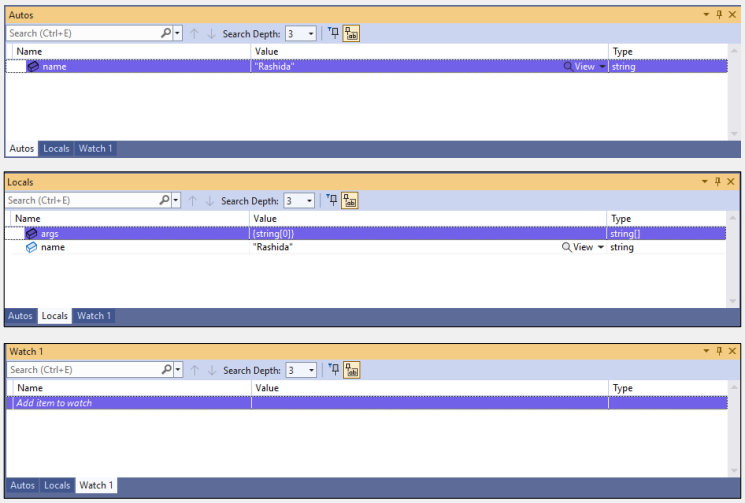
These options have been enabled within the provided virtual machine.



Visual Studio includes a powerful debugger. This tool can be used to step through code to help identify logic flaws.

It is also useful for examining the contents of properties and variables to check they contain expected values.

Debugging Windows

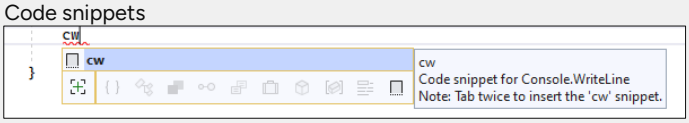


QA

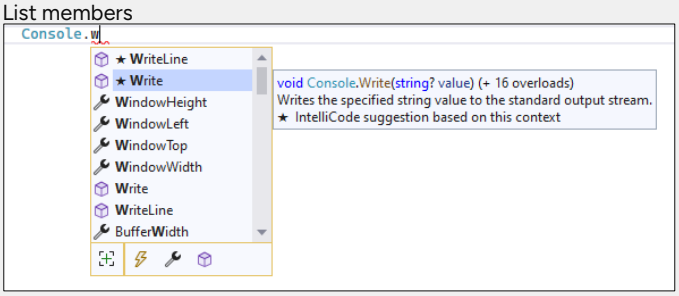
Output – the default output window. You will use it mainly to see build results.
Quick Watch – on the right-click context menu. Used for examining individual symbols.
The Watch window is the result of a Quick Watch ‘save’.
Autos shows the variables in scope.
Locals shows the local variables.
The Call Stack is used to trace back through previous procedures.
The immediate window allows you to enter any valid C# expression and have it evaluated at that point in the code.
All breakpoints can be seen in the Breakpoints window.
The Modules windows shows all loaded modules.
The Process window shows the processes associated with the project.
The Threads (and Tasks) windows show the state of threads (or tasks) under breakpoint conditions.

IntelliSense

Code snippets



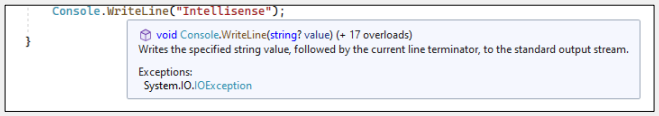
List members



QA

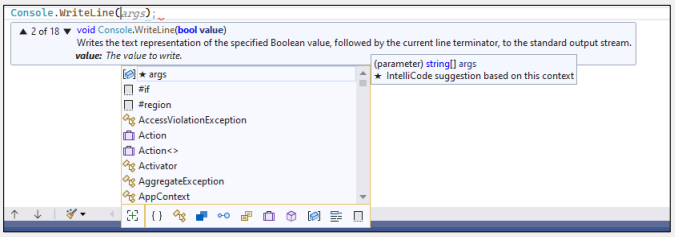
IntelliSense is a code-completion aid that includes a number of features such as code snippets, list members, quick info and parameter info.

IntelliSense



Quick info

Parameter info



QA

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.

QA

Using Git and Git for Source Control

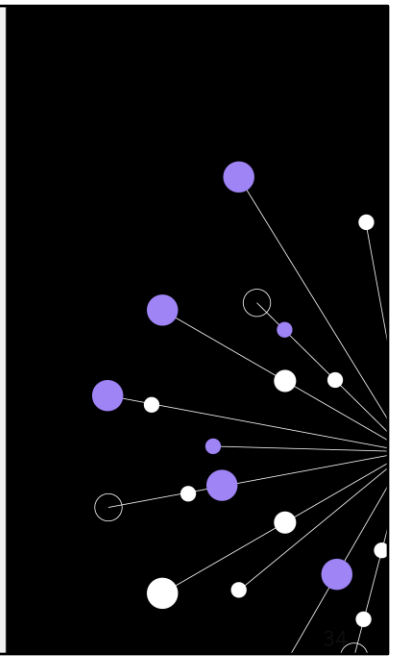
- The practice of tracking and managing changes to code
- Tracking changes to code allows developers to:
  - Centralise all code changes and additions to one code repository
  - Allow for simple and effective collaboration within development teams
  - Control the integration of new code into the codebase
  - Track changes from the entire team over the full lifetime of the project
  - Revert code back to previous versions

The first lab contains a step-by-step guide to configuring Visual Studio to use Git and GitHub to create a source control repository for your code.

QA

Summary

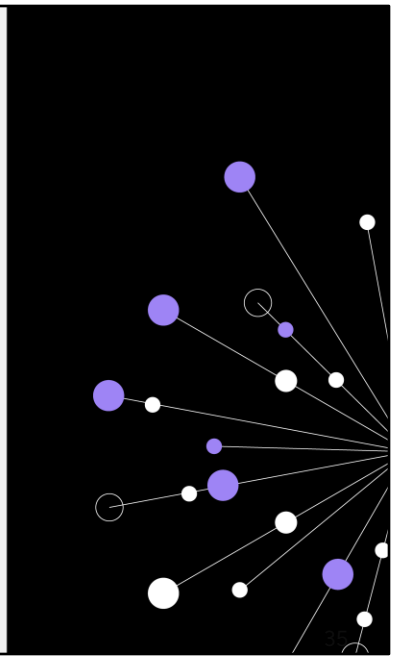
- What is .NET?
- .NET compilation
- What is C#?
- Hello World in .NET 6 and .NET 5
- Hello World explained
- Namespaces
- Visual Studio
- Keyboard shortcuts



QA

ACTIVITY

Exercise 02 Introduction to C# and .NET

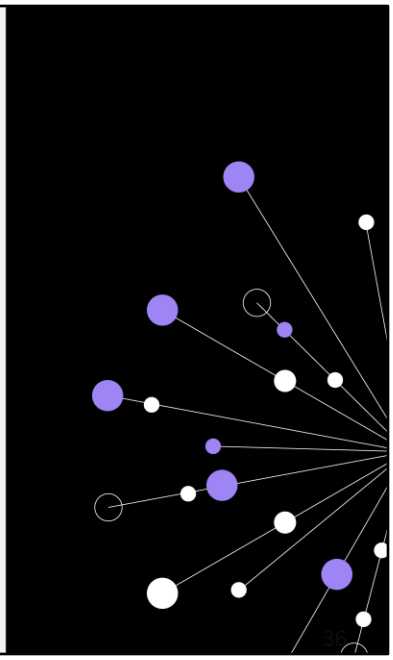


QA

Appendix

More about Namespaces

QA



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.

You cannot include nested namespaces in a file scoped declaration.

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

QA

Above are some of the types you will be working with on this course.

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;`

QA

If you want to remove this behaviour and manually control all namespaces in your project, add the following to your project file:
`<ImplicitUsings>disable</ImplicitUsings>`

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>
```

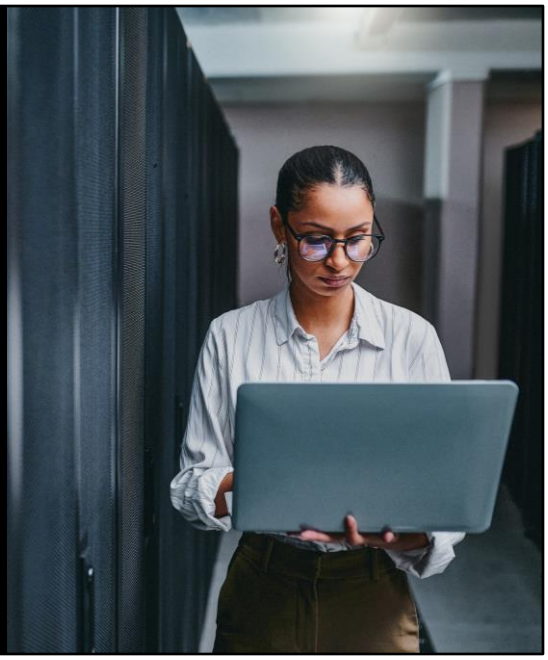
QA

This property is available starting in .NET 6.

03 Variables and Datatypes

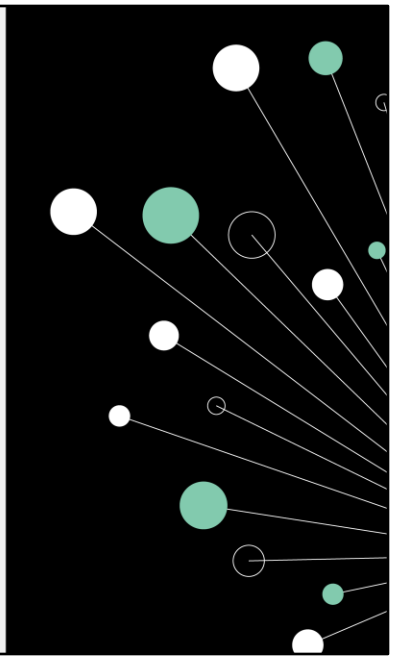
C# The Programming Language Part 1

QA
Learn. To Change.



Learning objectives

- Declare and initialize variables in C#.
- Differentiate between various data types (value types and reference types)
- Understand implicit and explicit type conversions
- Use operators (arithmetic, relational, logical, etc.) to manipulate data
- Understand the scope and lifetime of variables

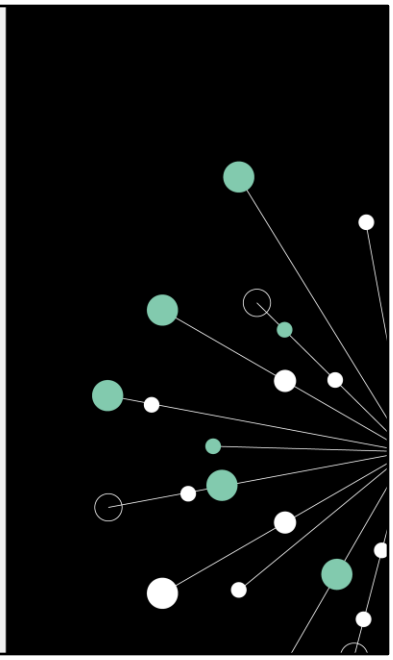


QA

Contents

- Comments
- Identifiers
- Variables
- Built-in types
- Strings
- Creating variables: Value and reference types
- Variable Scope
- Operators
- Parse and casting

QA



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>
```

QA

A C# statement can be a comment, a variable declaration, an expression, a control statement or a block.
The C# compiler ignores most comments although one type is treated differently. The two main types are known as the block comment and the line comment. Everything inside a block comment (/* ... */) is ignored by the C# compiler. With a line comment, only the text between the // and the end of the line is ignored.
To comment out a number of lines, Visual Studio provides a “Comment Block” command (Ctrl+K, Ctrl+C) which uses multiple rest-of-line comments. Ctrl+K, Ctrl+U will “Uncomment” a block.
The final comment is an XML documentation comment which begins with 3 forward slashes ‘///’.This is a special type of comment that can be used to generate documentation automatically. Visual Studio also uses these comments to help provide IntelliSense®.
The compiler switch ‘/doc’ causes the compiler to parse source files for these comments and produce an XML file describing the code.

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'

QA

Identifiers are the names used to define variables, fields, methods, parameters, classes, structs, enums, interfaces and delegates.
An identifier must start with a letter of the alphabet or an underscore. Subsequent characters may also include the digits 0 through 9. Note that C# is case sensitive therefore two identifiers can be differentiated by case alone. There is no restriction on the length of identifiers so best practice is to choose meaningful names that may combine several words.
In general, C# developers tend to use the following conventions:

- Private fields use *camelCasing*, where the first word is in lowercase and all subsequent words have the first letter capitalised e.g., `firstName`.
- Methods names use *PascalCasing*, where each word's first letter is capitalised e.g., `LoginUser()`.
- Parameter names and local variables use camel casing e.g., `amountToAdd`.
- Type names use Pascal casing e.g., `DateTime`.

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;
```

See Appendix for more on data types

QA

A variable is a symbolic name for a block of memory in which a value can be stored.
All variables must be declared before they can be used.
The declaration consists of the type name followed by the variable name.
Variables can be declared anywhere within a block, although it is often preferable to declare it at the start.
Local variables (as opposed to instance and class variables) must also be given values before they can be used in an expression. If this is not done, the C# compiler will indicate an error. We can declare and initialise a variable all at the same time by using the assignment operator '='. Note that if several variables are declared on one line, each variable must be individually initialised. If not, the initial value specified will be assigned only to the last variable in the declaration. The two integers *i* and *j* show an example of this in action, however it is considered to be poor programming practice and best practice is to declare and initialise each variable in separate statements.

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
```

QA

Literal floating-point values (e.g. 7.5) are, by default, assumed to be doubles. Consequently, you may find the IDE complaining about setting a float or decimal variable to a literal floating-point value. For example, the following will be reported as a "Literal of type double cannot be implicitly converted to type float'; use a 'F' suffix to create a literal of this type" syntax error:

```
float f = 7.5;
```

To fix the error you do as the error message suggests and add a suffix to the literal value that specifies its type. E.G.

```
float f = 7.5F;
Decimal money = 2.33M; //Think "M" for money
```

Other suffixes include:
"L" for long, "D" for double, "U" for uint (unsigned integer), "UL" for ulong

When using the **var** keyword, the compiler determines the type of the variable.

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
```

QA

Strings are *immutable* - the contents of a string object can't be changed after the object is created, although the syntax makes it appear as if you can. For example, when you concatenate the word “morning” onto the end of the message text of “good “, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to **message**. The memory that had been allocated for **message** (when it contained the string “good ") is then eligible for garbage collection.

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 **even better** 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
```

QA

Creating Variables: Non-Simple Types: Structs and Enums

- 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
```

QA

Creating Variables: Classes (Reference types)

- Classes (reference types) are also 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.

More about classes later on in course

QA

The '**if**' statement is an example of a conditional expression.
The '**is**' operator checks if the result of an expression is compatible with given type.
The '**nameof**' operator produces the name of a variable, type, or member as a string constant and is useful if the code is refactored and the variable name is changed.

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)
    }
}
```

QA

The part of the program where a particular variable is accessible is termed as the **Scope** of that variable.
Variables declared within one block (set of braces) are not visible outside of that set of braces.
For example a variable declared within a class is not visible outside of that class and a variable declared within a method is not visible outside of that method.

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

QA

As well as the location of a variable or type affecting its scope, the accessibility of a type or type member controls whether that type or member can be used from other code in your assembly or other assemblies.
NOTE: An assembly is a **.exe** or **.dll** created by compiling your code files.

Not all access modifiers are valid for all types or members in all contexts. For example, classes, records and structs declared directly within a namespace can be either **public** or **internal**. If no modifier is specified, the default is **internal**.

Operators

C# provides many operators, such as:

- Arithmetic operators
- Comparison operators
- Boolean logical operators
- Bitwise and shift operators (see appendix)
- Equality operators

QA

Arithmetic Operators

C# provides standard arithmetic operators:

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

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

QA

C# has a range of arithmetic operators.
Incrementing or decrementing a value by one is a very common operation in any language. In common with the other brace languages, C# provides special operators for this purpose. Both the increment (++) and decrement (--) operators can be prefixed or post-fixed. The difference in behaviour is subtle: using the prefix version returns the value of the variable after it has been incremented, using the postfix version returns the value of the variable before it has been incremented, thus:

```
i = 3; x = ++i; // results in both i and x being 4
j = 3; y = j++; // results in j being 4, but y being 3
```

Also, note that the "unary" operators of plus '**+**' and minus '**-**'(so called because they have only one operand) have higher precedence than most operators.

The division operator loses precision when dividing two ints. In the following code, z will have the value of three after the code has been executed, because ints are unable to hold any information other than whole numbers.

```
int x = 10;
int y = 3;
int z = x / y; // z will be 3, NOT 3.333333
int r = x % y; // r will be 1
```

Assignment statements can be simplified. These two statements are identical:

```
x = x + 10;
x += 10;
```

Arithmetic Operator Examples: ++ and --

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

// Can be used in a pre-fix format:
++i;
```

- See notes below to discover more about how the pre and postfix operators work

QA

```
 public static void ArithmeticPreAndPostFixOperatorExamples()
{
    Console.WriteLine("Pre and Post Fix Operators:");
    // 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
    i = 3;
    Console.WriteLine(i); // output: 3
    Console.WriteLine(i--); // output: 3
    Console.WriteLine(i); // output: 2

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

Arithmetic Operators: Add, Subtract, Multiply, Divide, Remainder

```
int a = 1;
int b = 2;
int c = 3;
int d = 10;
int result = 0;

result = a + b;  //3
result = c - a;  //2
result = b * c;  //6
result = d / b;  //5

result = d / c;  //3 (integer result)
result = d % c;  //1 (remainder)
```

See Appendix for more on examples

QA

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;
```

QA

The sequence in which the operators bind with the operands is identical to that of Maths (i.e. multiple/divide bind closer than plus/minus). As with Maths, you can always alter the precedence with brackets.
BODMAS: Brackets Order Divide Multiply Add Subtract

Comparison Operators

C# provides standard comparison operators:

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

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

QA

Comparison Operator Examples:

```
//numeric comparison
Console.WriteLine(5.3 > 2.4); // output: True
Console.WriteLine(5.3 <= 2.4); // 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: True
```

- See notes below for more examples

QA

```
public static void ComparisonOperators()
{
    // 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 or equal <=
    Console.WriteLine(7.0 <= 5.1); // output: False
    //greater than or equal >=
    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: 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**

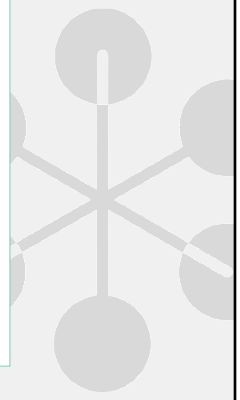
- Operators &, | and ^ perform bitwise operations when the operands are integral numeric types

QA

The unary prefix ! operator computes logical negation of its operand. That is, it produces true, if the operand evaluates to false, and false, if the operand evaluates to true.
For a logical AND, the result of x & y is true if both x and y evaluate to true. Otherwise, the result is false.
For a logical OR, the result of x & y is true if either or both x and y evaluate to true. Otherwise, the result is false.
The ^ operator computes the logical exclusive OR (XOR) of its operands. The result of x ^ y is true if x evaluates to true and y evaluates to false, or x evaluates to false and y evaluates to true. Otherwise, the result is false. The two operands must be different (have exclusive values).
The conditional logical AND is a short-circuiting operator. If the first operand evaluates to false, the second operand is not evaluated.
The conditional logical OR is a short-circuiting operator. If the first operand evaluates to true, the second operand is not evaluated.

Boolean Logical Operator Examples

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

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

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



See notes below **and appendix** for more examples

QA

```
public static void LogicalOperators()
{
    Console.WriteLine("Logical Operators:");
    // 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

    // 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

    // 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
}
```

Equality Operators

- C# provides two equality operators:
- 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**
!=	**Inequality operator**

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

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

See Appendix for more on examples

QA

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");
```

QA

To convert a string to a simple value type, you can use the **Parse** method.
The Parse method is a static method available in the primitive data types that enables a string to be parsed to a numeric value of the corresponding type.
You will see **TryParse** in a later chapter.

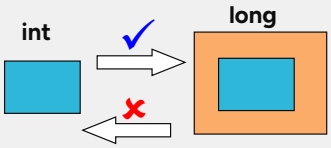
Integer Arithmetic & Conversions

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

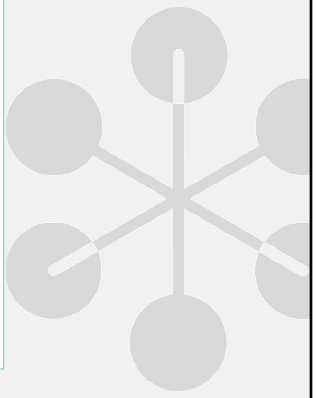


QA

Most of the arithmetic operators in C# are similar to those in other languages. Integer division results in an integer and any remainder is ignored. The modulo operator (%) returns the remainder after an integer division. The multiply and divide operators have higher precedence than the add and subtract operators (BODMAS – Brackets | Orders | Divide | Multiply | Add | Subtract).
It is important to note that, internally, all integer arithmetic is performed with int or larger values. byte, char and short values are automatically widened or promoted to int before an arithmetic operation commences and the result will be an int. Therefore, if the result is to be assigned to anything less than a int, the result must be explicitly cast. Otherwise, the compiler will flag an error. Similarly, if the argument on one side of an arithmetic operator is a long, the argument on the other side will be promoted to a long and the result will be a long.
If necessary, C# will automatically convert or cast a variable or expression of one numeric type to a wider type, e.g. if you assign an int to a long. However, if the reverse is attempted, the compiler will complain because this may result in loss of information. The compiler can be forced to convert a variable of one numeric type to a narrower type by using an explicit cast.
The syntax for an explicit cast is to put the target type in parentheses in front of the expression or variable.

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
```

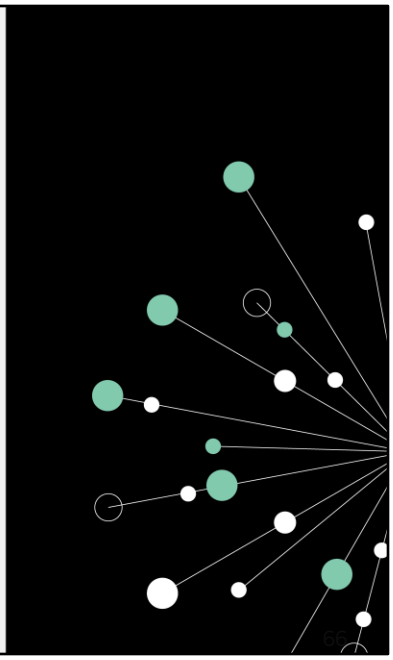


QA

Summary

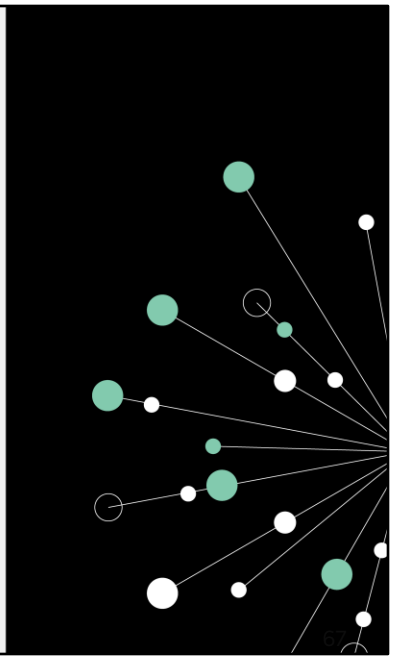
- Comments
- Identifiers
- Variables
- Built-in types
- Strings
- Creating variables: Value and reference types
- Variable Scope
- Operators
- Parse and casting

QA



ACTIVITY

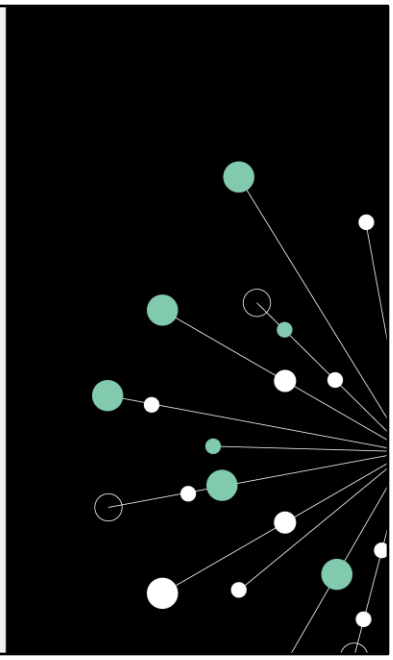
Exercise 03 Variables Basic Syntax and Structure



QA

Appendix

- Value and Reference Types
  - Built-In Value Types
  - Reference Types
- Operators
  - Arithmetic Operators
  - Boolean Logical Operators
  - Equality Operators
- Bitwise and Shift Operators
- Casting



QA

Value and Reference 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

QA

Note: *in*, *ref* and *out* parameter variables change the behaviour of value type variables

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

QA

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

QA

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;
```

QA

The default value of each integral type is zero, 0.
Each C# keyword in the leftmost column is an alias for the corresponding .NET type (apart from *nint* and *nuint*).

The two lines of code declare variables of the same type.

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);
```

QA

The default value of each floating-point type is zero, 0.
Each C# keyword in the leftmost column is an alias for the corresponding .NET type

The two lines of code declare variables of the same type.

A literal number with a decimal point is interpreted as a double by the compiler.
Use a suffix on the literal to create a literal of the correct type.

- F / f – float suffix
- D / d – double suffix
- M / m – decimal suffix

The floating-point types have the following constants:

- Double.NaN
- Double.NegativeInfinity
- Double.PositiveInfinity

When the result of an operation is undefined it returns Nan, for example, dividing zero by zero.
NegativeInfinity is returned when you divide a negative number by zero.
PositiveInfinity is returned when you divide a positive number by zero.

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
```


QA

There are built-in keywords representing the two constants: *true* and *false*. The bool type is the result type of comparison and equality operators.

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';
```


QA

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;
```


QA

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

QA

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());
```

QA

You create an instance of a struct using the **new** keyword.

You cannot change the X or Y properties after construction because they are *init-only* properties.

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);
```

QA

The above enum types represent a choice from a set of mutually exclusive values. To represent a combination of choices, define an enumeration type where the values are powers of two:

```
[Flags]
enum MultiHue : short
{
None = 0,
Black = 1,
Red = 2,
Green = 4,
Blue = 8
};
```

The [Flags] attribute makes it possible to represent a non-listed value as a combination of listed values e.g., 6 would give the value of Red and Green (2 + 4). For more information: https://docs.microsoft.com/en-us/dotnet/api/system.flagsattribute?view=net-6.0

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

QA

The *object* type is an alias for **System.Object**. All types (reference and value types) inherit directly or indirectly from **System.Object**.
The *string* type represents a sequence of zero or more Unicode characters.
The *dynamic* type indicates that use of the variable and references to its members bypass compile-time type checking. Instead, these operations are resolved at run time. Type *dynamic* behaves like type *object* in most circumstances.

- A *class* is a template for a new type.
- An *interface* defines a contract.
- A *delegate* is used to encapsulate a named or anonymous method. .NET has built-in delegate types so you likely don't need to define new custom delegate types.
- A *record* is a type for encapsulating data. The *record* type was introduced in C# 9 and the *record struct* type was introduced in C# 10.

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*

QA

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

QA

A class is a template for a new type. Classes typically contain data in the form of constants (const), fields and properties. Classes also contain behaviour in the form of methods. Methods have brackets where parameters can be defined and then passed when the method is invoked. Classes can contain events which enable them to notify other classes or objects when something of interest occurs.

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");
```

QA

An object instance is created by using the **new** keyword.
julieCar is an object instance and *lisaCar* is an object instance. They are both of type *Car* but are separate instances.
Name, address, postcode and country are the method's parameters. When the method is invoked the parameters are passed positionally.

Arithmetic Operator Examples: Add + and 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
```


QA

Arithmetic Operator Examples: Multiply * and 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
```


QA

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
```


QA

The modulus or remainder operator % computes the remainder after dividing its left-hand operand by its right-hand operand.

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
```


QA

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
```

QA

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
```


QA

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
```


QA

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
```


QA

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

QA

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

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
```

QA

0b is the prefix for binary literals.
The underscore can be used since C# 7 as a digit separator in numeric literals to aid readability.

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
```


QA

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

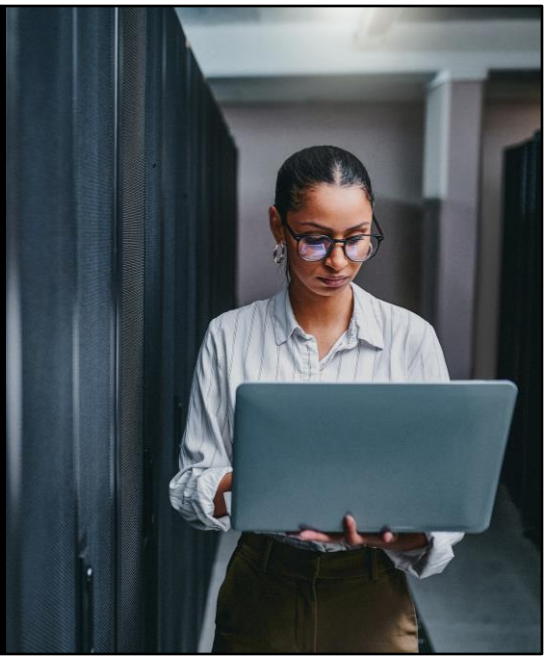
```
((Dog)GetMammal()).Bark();
```

QA

04 Basic Functions

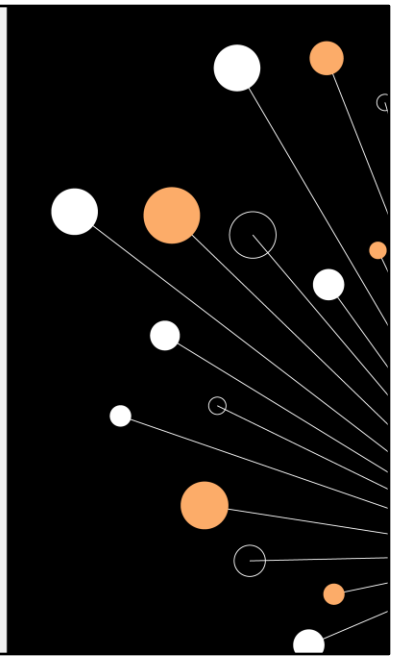
C# The Programming Language Part 1

QA
Learn. To Change.



Learning objectives

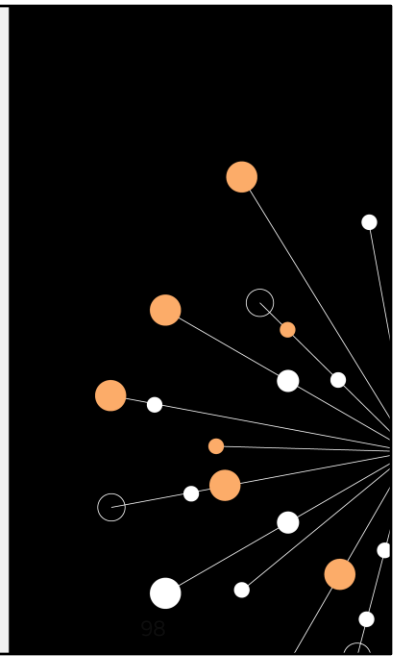
- Know how to create and use functions (aka methods)
- Understand how to declare and pass parameters
- Be able to create and use method overloading
- Recognise how to create and the benefits of using expression bodied methods



QA

Contents

- Code blocks
- Basic Syntax
- Functions
  - Parameters
  - Return types
  - Overloading
  - Expression bodied methods



QA

Code Blocks

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

- Braces group many statements into one code block

```
decimal CalcVAT (decimal amount) {
    return amount * 0.2M;
}
```

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

```
decimal CalcVAT (decimal amount)
{
    return amount * 0.2M;
}
```


QA

Grouping many statements inside a set of curly braces allows them to be collectively treated as the one next statement.
It is regarded as good practice to insert braces even though there is just one statement within the braces because you might revisit the code at a later time and want to add extra statements. The braces ensure this group of statements are treated as a single code block and that all the statements will run.

Terminology

You may have heard these words of the 'programming vocabulary'

- Function, Subroutine, Procedure

In OO languages only the word 'method' is used, consisting of ..

- Signature line plus..
- an associated block of code (1 or more statements in { } braces)
  - Signature and body can NEVER be separated
  - Cannot author a method inside a method, know where the '}' is!

```
public static void DoStuff( int num )
{
   ..;// statements that 'do' stuff
}  // end of method, know where it is!
```

Methods only defined inside the body { } of a class def'n

```
class Program {
   public static void Main(...)  { statements... }
   public static int  Other(...) { statements... }
}
// but not here !!
```


QA

Here is a simple method with 2 mistakes commonly made in the early days.
Try to avoid them!!
public static void Whatever(string data); // no semi colon in a method signature

```
// don't type anything here, it is not part of the
method, when would it run?
{
     ..statements all go here, terminated by semi-colons

}

// next method goes here
// methods can go in any sequence in a class, they can
all see each other
```

Basic syntax of a method

A named block of statements that can be invoked

```
modifiers returnType MethodName( parameterList ) {
   // method body
   ..;
}
```

Often 'void'

Best location?

The key components of a method are:

- Optional modifiers such as public (and for now 'static')
- If the method returns a value, its type; otherwise specify void
- The name of the method - PascalCasing
- An optional parameter list inside mandatory parentheses
- Each parameter has a data type and name e.g. `(int age)`
- Multiple parameters comma-separated `(string name, int age)`
- The method body containing the statements to be executed

QA

Every C# program must contain at least one method, even it is only Main(). However, when you define a class for an object-oriented program, you implement all of the behaviour (or operations) of that class in one or more methods. A C# method is equivalent to a function, procedure or subroutine in other languages. However, all methods must be defined within a type definition. In other words, there is no support for global functions in C#; every method has to be defined in some type or other.
A method definition has five components as shown on the slide. Some points to note are:

If a method does not return anything, specify the return type void
The name of a method can be any valid identifier (like a variable name)
If the method has no arguments (parameters), specify empty parentheses
If you want code outside the scope of the type definition to be able to access the method you must apply an access modifier (such as `public`).
Note that the default modifier in C# is private.

Examples – defining and invoking (1)

A method is named block of statements that can be invoked

```
public static void UpdateAllSalaries() {
   // method body
}
```

```
UpdateAllSalaries();          ✓
UpdateAllSalaries(200);       ✘
int x = UpdateAllSalaries();  ✘
```

Client code in another method

```
public static int CalculateBonus() {
   return ..; // some calculation
}
```

```
CalculateBonus(27);           ✘
int bonus = CalculateBonus(); ✓
CalculateBonus();             ?
```

All statements above that call a method assume it is in 'this' class!

QA

Each method call above is invoking the method without specifying ClassName<dot>.
This can only be done if the method is in the same class as the calling code.

The final method call is valid but pointless. `CalculateBonus()` is an '`int`' method.
That means it provides an '`int`' value, are we not interested in knowing what this bonus is?, or maybe it returns the value 1 or -1 indicating whether there was a bonus?
Either way you usually want to 'catch' a returned value.

Examples – defining and invoking (2)

```
public static void PassNameAndAge(string name, int age) {
   // method body that uses 'name' and 'age'
}
```

Pass correct no of values
- of the right type
- in the correct sequence

```
PassNameAndAge();                    ×
PassNameAndAge("Fred");              ×
PassNameAndAge(27);                  ×
PassNameAndAge(27, "Fred");          ×
PassNameAndAge("Fred", 27.0);        ×
PassNameAndAge("Fred", 27);          ✓
```

Method is 'void'. It simply DOES NOT produce a value

```
????  ??  = PassNameAndAge();                      ×
Console.WriteLine(PassNameAndAge("Fred",27));     ×
```

Produce the string and int arguments from anywhere though

```
PassNameAndAge( GetMeAName(),(int)Math.Sqrt(27.2) );   ✓
```

QA

6 calls to method `PassNameAndAge(string name, int age)`.
The first 5 are invalid, you simply must pass arguments corresponding to the number of parameters defined in the method. 5th line is invalid as you cannot pass a (wider) double where a (narrower) `int` is needed.

The final method call is valid because the double that has been returned from `Math.Sqrt(27.2)` has been 'cast' to `int`. This effectively truncates it to the value 5.

Examples – defining and invoking (3)

```
public static double Average(double d1, double d2) {
   return (d1 + d2) / 2;
}
```

```
int i1 = 3, i2 = 9;
double dub1 = 3.4, dub2 = 5.8, result;
result = Average(2, i2);
result = Average(i1, i2 * 3);
result = Average(dub1, i2);
result = Average(dub1 + dub2, i1 - i2);
result = Average(10, Average(20,60));
Console.WriteLine(result);
Console.WriteLine(Average(10,20));
Average(dub1 * 2, dub2);
int value = Average(i1, i2);
```

Evaluate each one of these method calls

Pass a literal, a variable, an expression, a value from a method call

- No relationship between 'name' of argument and 'name' of parameter
- DO NOT say the type of what you are passing, the compiler KNOWS

QA

The first 5 calls are all valid, they all pass 2 numeric parameters and an int is always safely widened to double.
Note the 5th call uses the double that is returned as the average of 20 & 60 (40) as the 2nd argument of another call to Average. The value produced is therefore (10 + 40) / 2 = 25.

The only invalid call is the final one, Average produces a double, you cannot assign that to an `int` (without casting) as that would be a narrowing conversion.
The correct statement would be either:

```
double value = Average(i1, i2);
```

Or

```
int value = (int)Average(i1, i2);
```

In the latter case 'value' would be a whole number regardless of the values of i1 and i2.

An interesting question here is, should the parameters to Average have more meaningful names than 'd1' and 'd2'? The answer is no as it a way of saying 'look the parameter name has no meaning it is the parameters type that is key. Pass me any double value.

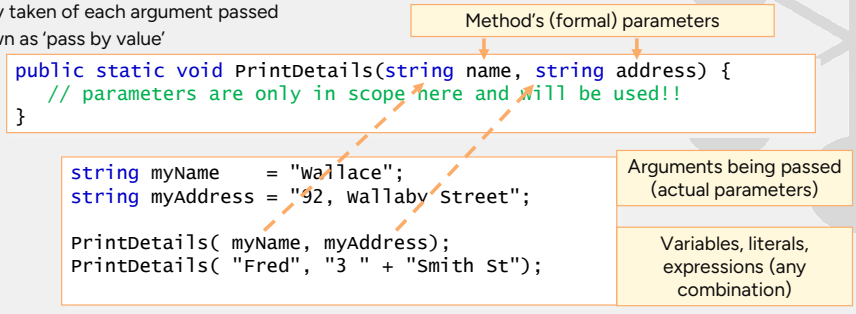
The Average of 4 & 6 is the same as the Average of 6 & 4, it does not matter which value is passed to which parameter.

Positional notation and pass-by-value

Traditional parameter passing notation

Arguments passed in same sequence as parameters defined

- (1st to 1st ..) (2nd to 2nd ..) etc
- Parameters are effectively local variables of the receiving method
- Are a copy taken of each argument passed
  - Known as 'pass by value'



QA

PrintDetails(.. , ..) takes 2 strings, it clearly matters which order the arguments are passed in.
Names have very different values to Addresses.
Also of course the client code will see Intellisense saying 'name' 'address' in that sequence. So harder to pass the arguments in the wrong order.
Remember PrintDetails("Buck House, London", "Charlie"); would compile but produce a very odd mailing address.

Don't ever worry about lines that fail to compile, worry about the ones that do compile but produce erroneous results .

Named and Optional Parameters

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

```
public static void PrintDetails(string name, string address) {
    // parameters are only in scope here and will be used!!
}
```

```
string myName    = "Wallace";
string myAddress = "92, Wallaby Street";

PrintDetails( name: myName, address: myAddress);
PrintDetails(address: "3 " + "Smith St", name: "Sadia");
```

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

```
public static void PrintDetails(string name = "Anon", string address = "") {

}
```

```
string myName    = "Wallace";
string myAddress = "92, Wallaby Street";

PrintDetails();
PrintDetails(myName);
PrintDetails(address: myAddress);
```

QA

Where there is a mix of optional and mandatory parameters all mandatory parameters must be declared before any optional ones

Returning a value from a Method

In the method definition

Method must return a value or expression of appropriate type

- Using a `return` statement
- Compiler checks every path through the method returns a value
  - Of great interest when we get to `if` / `else` and loops

For `void` methods, no `return` statement is necessary

- Implicit `return` at the closing brace, 'drops thru the bottom'

```
public static double Average( double i, double j ) {
  return (i + j) / 2;
}
public static bool IsHugeSalary( decimal salary ) {
  return salary >= 100000;
}
```

```
double ave = Average(20.3, 45.5);
bool isFatCat = IsHugeSalary(23000M);
```

When the method is called

- Can be used in the same way as any other expression
- You can ignore the return value if you want to (rare)

QA

A method can return a single value to the caller of that method. If so, it must be defined with a return type (e.g. `bool`) in front of the method's name. Later we will see a method can also return a reference to some object, which can be useful if you want to return more than one value. The method must also include a return statement. return takes a single value or expression that must be compatible with the return type specified previously. Note that a method returns when the first return statement is executed; it does not 'remember' the return value and return later at the closing brace of the method's body.

If the method doesn't return anything, the return type must be specified as void. A void method returns automatically at its closing brace of its body. It can also return prematurely using an explicit return statement with no expression or value. However, irrespective of whether a method returns anything, it is good practice to provide only a single return statement.

To call a method (from some other method!), you simply specify that method by name. A method with a return type can be used in an expression like any other variable, but don't forget the method's parentheses and any required arguments. If the return value of a method is not required, it can simply be ignored, this is rare though.

Introducing some FCL methods

Read this carefully … Is there is a problem?

```
public static void Main( ) {
  Console.WriteLine("What is your name?");
  string name = Console.ReadLine();
  Console.WriteLine("What is your age?");
  int age = Console.ReadLine();
  Console.WriteLine("Hi " + name + ", next year you will be " + (age + 1));
}
```

Example of statements calling methods of a different class

Well, what does 'ReadLine()' return?, it can't 'change' ½ way through a method!!

```
public static void Main( ) {
  Console.WriteLine("What is your name?");
  string name = Console.ReadLine();          ✓
  Console.WriteLine("What is your age?");    ✓
  string sAge = Console.ReadLine();
  int age = Convert.ToInt32(sAge);
  Console.WriteLine("Hi " + name + ", next year you will be " + (age + 1));
}
```

QA

`ReadLine()` was authored over a decade ago, the author had to define the return type. It simply had to be 'string' otherwise no one could ever type in text to the Console prompt.

You cannot cast a String to an int, they are as different as apples and oranges, chalk and cheese. However, the string may contain numeric data.
The Convert class is an all singing all dancing method for converting something of one type to another.
Let's not worry yet what happens if you try to convert non-numeric string data to int.
Note the `Readline()` call and `Convert.ToInt32()` call could be combined.

```
int age = Convert.ToInt32(Console.Readline());
```

Method Overloading

Method names can be *overloaded*

- Two or more methods with the same name, (in the same class)
  … but having different method *signatures*

The signature is the method name and parameter types (not names)

- You can't overload a method solely by changing the return type

```
public static void PrintPersonalDetails( string name, int age ) { ... }
```

```
public static void PrintPersonalDetails( string name, int age, string hobby) {
 PrintPersonalDetails(name, age);
  Console.WriteLine( hobby );
}
```

Calls the other version to re-use code

```
Print( myName, myAge);
Print( myName, myAge, myHobby );
```

Effectively an alternative to having 'optional parameters'

QA

There will be a number of situations where you will want to pass different parameter types to a method. A classic example of this is the Console.WriteLine method, to which you might want to pass a string, and integer, a boolean etc.
To support this, C# lets you overload a method. All this means is that your type contains multiple definitions for the method, each of which differs from the others in the number, order and types of parameters. In the example above there are two overloads for the Print method, one of which takes 2 parameters and one that takes 3. Each overload must have a unique signature (remember, it's not the names of the parameters that is considered, it is their type).
An important thing to note is that the return type or access modifier is not considered in determining whether methods are unique or not.
Finally, one thing that you can do with overloads is to emulate other languages concept of default parameters.
In C#1, C#2 and C#3 there is no concept of optional parameters with default values, later you will cover how C#4 supports these concepts, but primarily for interoperating with non .NET code.
In the code above, the overload of Print that takes 3 parameters simply calls the overload that takes 2 to reuse code. But an alternative would be for the version that takes 2 parameters to call the one that takes 3 by supplying a 3rd parameter as a 'default' value.
i.e

```
public void Print(string name, int age) {
         Print(name,age, "No hobby specified");
}
```

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

QA

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;
}
```

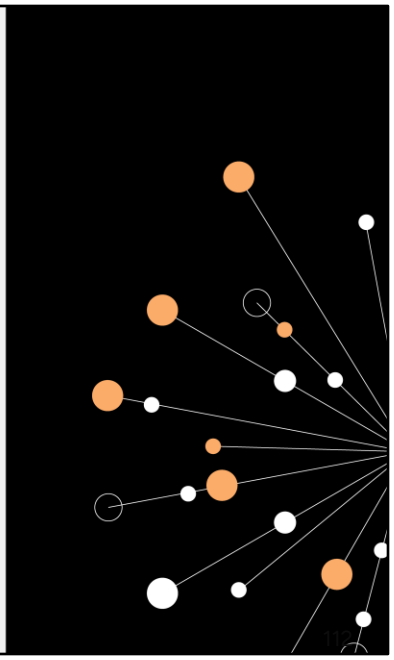
```
public void SquareANumber(int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

QA

Note the **return** keyword is not required in the expression bodied method to pass the value back to the caller.

ACTIVITY

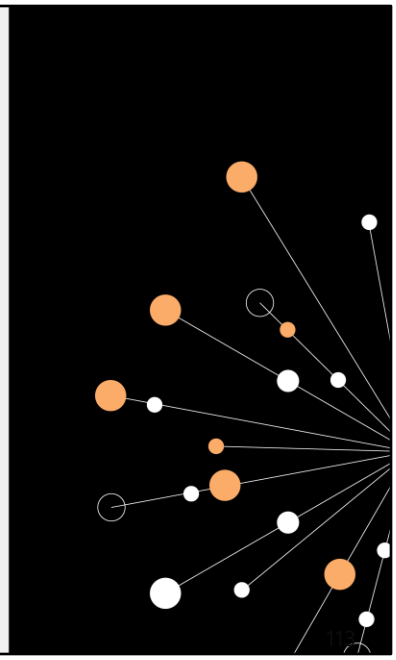
Exercise 04 Functions



QA

Summary

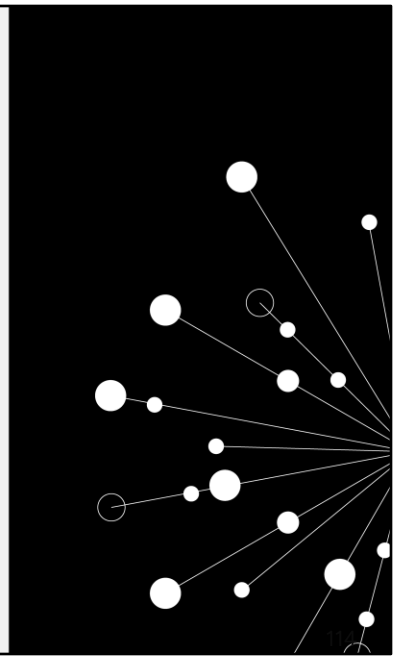
- Code blocks
- Basic Syntax
- Functions
  - Parameters
  - Return types
  - Overloading
  - Expression bodied methods



QA

Appendix

- Extension Methods



QA

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

QA

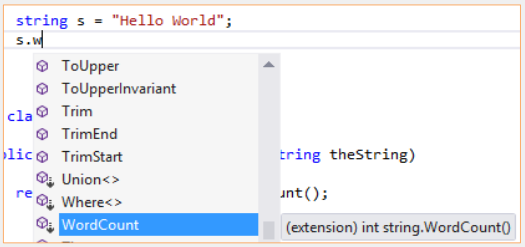
In C#, extension methods are indicated by the **`this`** modifier which must be applied to the first parameter of the extension method.

Extension methods enable us to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, called as if they were instance methods on the extended type.

For client code written in C#, there is no difference between calling an extension method and the methods that are actually defined in a type although the IntelliSense shows (extension) at the beginning of the pop-up dialog box. Extension methods are only in scope when the namespace has been explicitly imported into your source code with a **`using`** directive. In general, you will probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code.

Extension Method Example

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

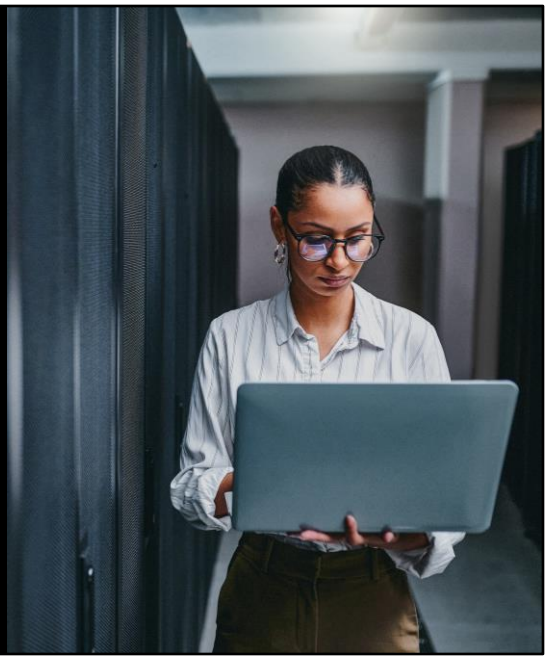


QA

05 Flow Control

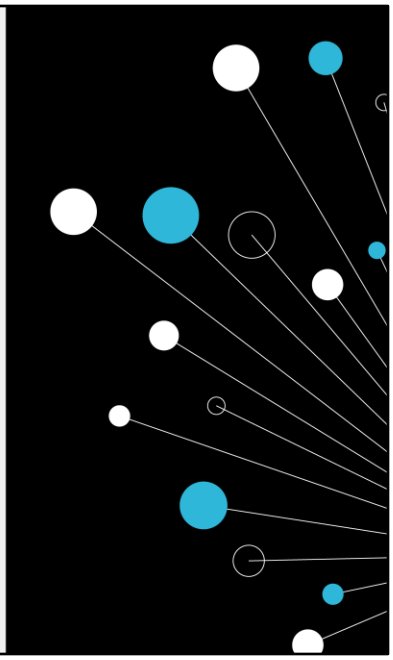
C# The Programming Language Part 1

QA
Learn. To Change.



Learning objectives

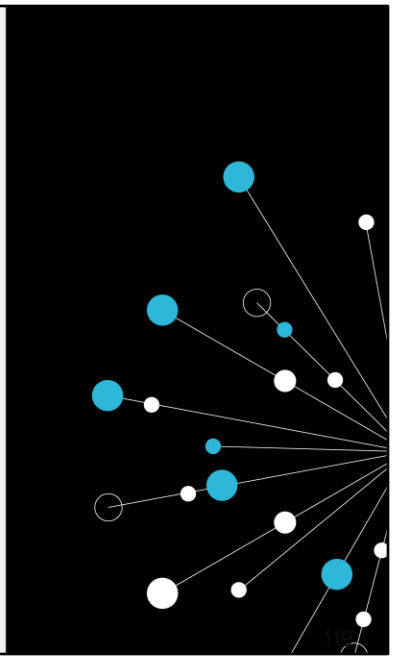
- Understand and implement conditional statements
- Write clean and efficient code using flow control structures



QA

Contents

- Select statements
- The if statement
- The switch statement
- The switch expression
- The ternary conditional operator



QA

If 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++;
}
```

QA

Grouping many statements inside a set of curly braces allows them to be collectively treated as the one next statement.
It is regarded as good practice to insert braces even though there is just one statement within the braces because you might revisit the code at a later time and want to add extra statements. The braces ensure this group of statements are treated as a single code block and that all the statements will run.

Select statements

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

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

QA

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);
}
```

QA

The primary means of decision making is the **if** statement. A boolean control expression determines which branch of the two-way fork is taken. If the expression evaluates to *true*, the first branch (the if-body) is taken. If the expression is *false*, then the second branch (the else-body) is taken. The **else** clause is optional; if it is omitted, nothing is executed if the control expression evaluates to *false*.

You can nest **if** statement to check multiple conditions.

A common coding error is to use the assignment operator (=) rather than the equality operator (==) in the control expression. Fortunately, the C# compiler will pick up this type of error because the control expression *must* evaluate to a bool.

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");
}
```

QA

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;
}
```

QA

The examples show 2 switch statements, one using integers and one using strings. Which one do you think is faster?
The answer is they are almost identical in speed because any string the compiler sees is 'interned' to an index into a table of strings and it then works off the index (which is an integer).
Note that:

1) You cannot "fall-through" from one case statement to the next. You need a break statement and only a single code branch is executed.
2) The final "default" must have a break statement

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);
```


QA

Switch statements are often used with enumerations.

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;
    }
}
```


QA

C# 9 introduced the use of a *relational pattern* to compare an expression result. The above example uses a *constant pattern* to compare if an expression equals the **double.NaN** constant and *relational patterns* to compare an expression with a constant using the relational operators **<** and **>.**

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);
```

QA

C# 8.0 introduced the *switch expression* syntax.
Use a switch expression to evaluate a single expression from a list of candidate expressions based on a pattern match with an input expression.

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*

QA

The example uses a constant pattern and a discard pattern.
The switch expression is being used as the body of an expression-bodied method.

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}");
```


This is a comparison of the switch statement syntax versus the switch expression syntax. The switch expression syntax is more concise.

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");
```

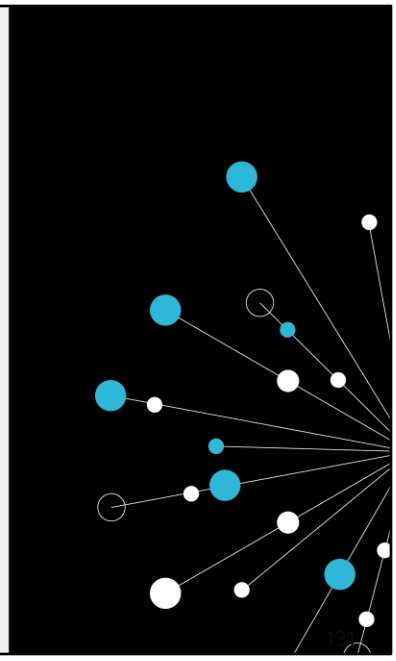
```
enum Coin
{
    Heads,
    Tails
}
```


QA

The ternary conditional operator evaluates a Boolean expression and returns the result of one of the two expressions. If the Boolean expression evaluates to true, the result of the first expression is returned, otherwise the result of the second expression is returned.

Summary

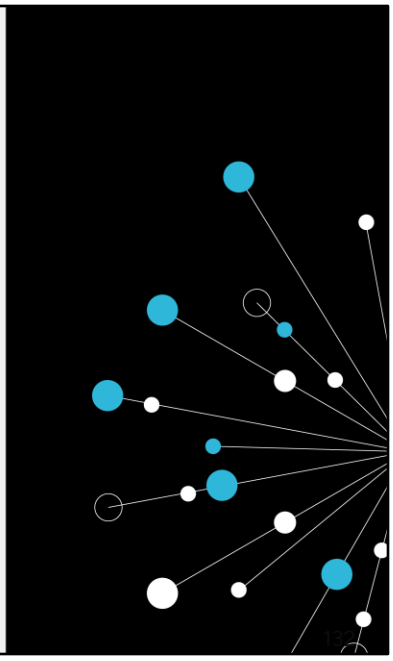
- Select statements
- The if statement
- The switch statement
- The switch expression
- The ternary conditional operator



QA

ACTIVITY

Exercise 04 Flow Control

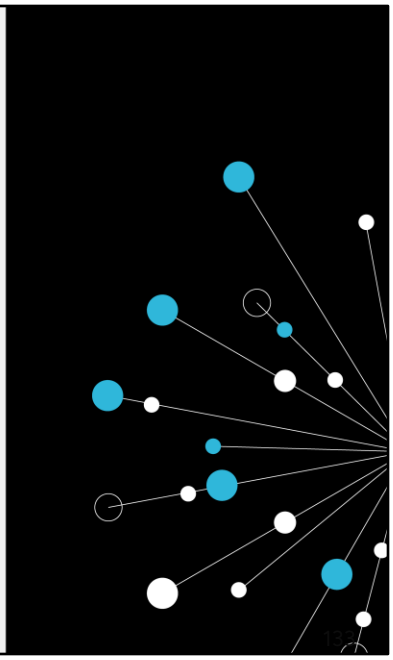


QA

Appendix

- Switch Case Guards
- The Null Operators

QA



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.

QA

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;
    }
}
```

QA

The above example uses positional patterns with nested relational patterns.

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
```

QA

The example uses property patterns with nested var patterns.

Note: A constructor initializes an object's data ready for use.

Null operators ?? ??= ?.

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

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

QA

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

QA

The null-coalescing assignment operator was introduced in C# 8.0

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

QA

The null-conditional operator is also known as the Elvis operator because it looks like a quiff above a pair of eyes which is said to resemble Elvis Presley.

Null operator examples

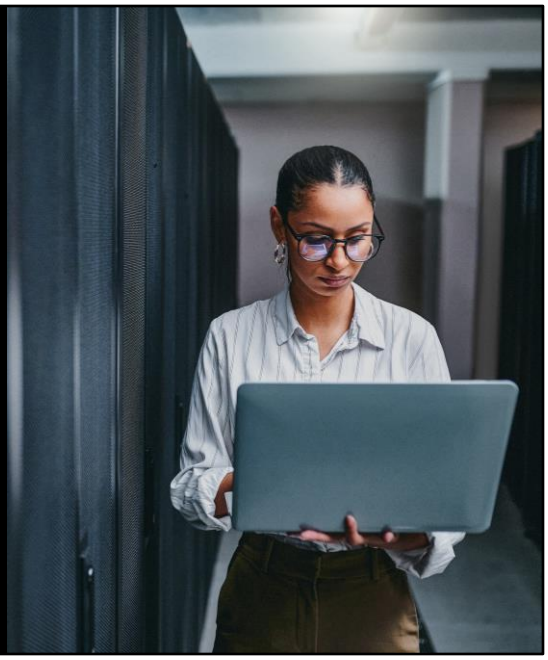
```
// null-coalescing operator
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1

// 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
```


QA

06 Arrays, Loops and Collections

C# The Programming Language Part 1

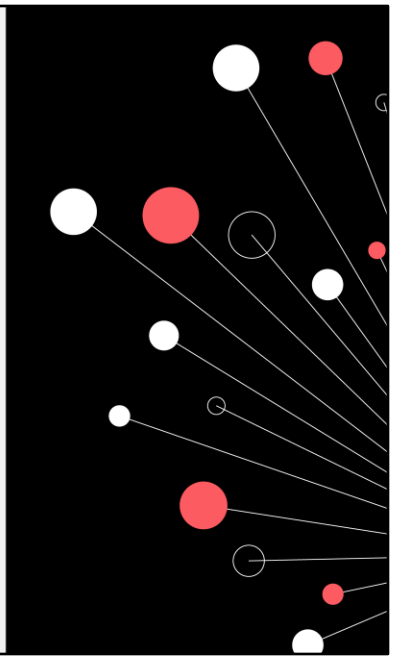
QA
Learn. To Change.



Learning objectives

- Introduction to Arrays
- Introduction to Loops
- Introduction to Collections
- Combining Arrays, Collections, and Loops

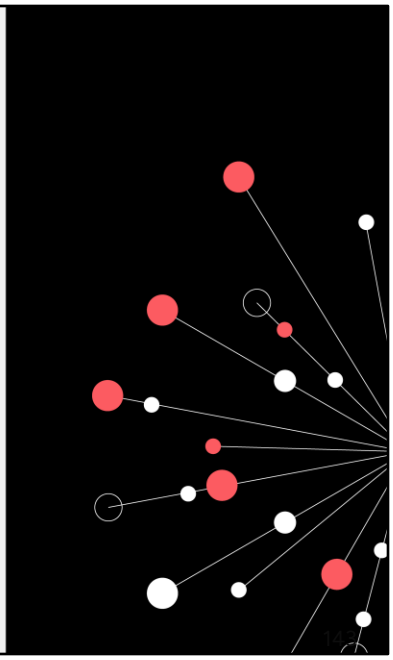
QA



Contents

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

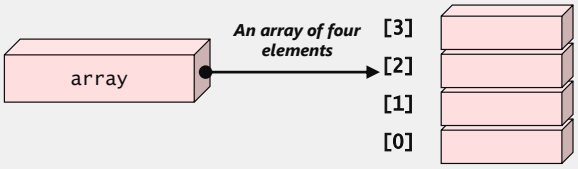
QA



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
```

QA

An array is a collection of objects. We can create arrays of any type, including the intrinsic types as well as arrays of our own classes and structures. Arrays are themselves objects in .NET. The CLR automatically generates a class, which is derived from the **System.Array** class, to support each specific array type. **System.Array** defines many methods and properties for accessing the contents of the array, which are accessible to all array types.
Arrays in .NET have a fixed length that is set when the array object is created*. If we need dynamically sizeable collections of objects we should use one of the classes from the System.Collections.Generic namespace. Like any other object, an array must be created before it can be used. Note that an array variable is a reference. Therefore, if an array is passed to a method, that method can permanently change the contents of the array.
In keeping with the other C-like languages, C# uses the square bracket [ ] notation to signify an array variable and also to provide an indexer into the content of the array so that a specific element can be accessed.
The first line of code on the slide creates an array of four integers with a lower boundary of 0 and an upper boundary of 3.
The third line creates an array of five integers, with a lower bound of 0 and an upper bound of 4. It uses an array initializer to create and populate the array. The length specifier isn't needed because it's inferred by the number of elements in the initialization list.
* .It is possible to resize arrays using the **Array.Resize()** method. However, this still involves creating a new array with the new size and then copying all of the elements from the old array to the new one. This is not very efficient.

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
```

QA

Array initialisers are useful for creating fixed size look-up tables.
For example, a simple calendar might use the following:

```
string[] months = {"Jan", "Feb", "Mar", "Apr",  "May,
"Jun", "Jul", "Aug",  "Sep", "Oct", "Nov", "Dec"};
```

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

QA

Using Foreach to Loop through Arrays and other Collections

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
}
```

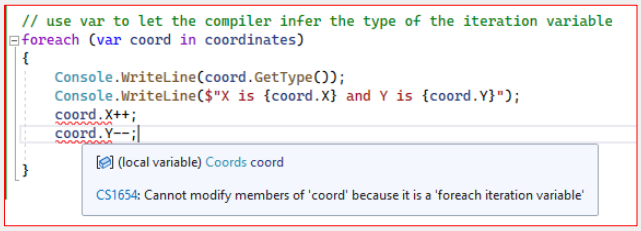
```
Coords[] coordinates = { new Coords(10, 20), new Coords(8, 3), new Coords(5, 5) };

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
```

QA

C# includes a foreach control statement that can be used to iterate over arrays and other collections (such as queues, stacks, array lists, etc.). The foreach loop differs from other loops in one important factor: both the collection and the items within the collection should be considered to be read-only. This is because the types that implement support for the foreach statement don't guarantee how they will extract and present the data.
Therefore, if we want to iterate through a collection and make changes, we should use one of the other loop constructs. When iterating over a collection to simply extract information, foreach is a good choice.

Foreach Does not allow modification of members

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



QA

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
```


QA

The initializer section is executed only once, before entering the loop. If the variable is declared and initialized in this section, it is not accessible from outside the **for** statement because it has block-level scope.
The condition section determines whether or not to run the next iteration in the loop. If it evaluates to *false*, the loop is exited.
The iterator section defines what happens after each execution of the loop body. It is common for the loop variable to be incremented or decremented in the iterator section.

Any of the components of a **for** loop can be omitted. For example, a common way to implement a do-forever loop is as follows:

```
for (;;) // do forever
{
  ...
}
```

This kind of loop must be exited by some other means such as a **break** statement.
Numerous types of statements can be included in the iterator section including variable assignment, method invocation and even the creation of an object using the **new** operator. Therefore, it may not be necessary to do anything in the loop body. In such cases, you would typically have a pair of empty braces.

For loop Examples: Array Enumeration

Iterators in a **for** loop can be incremented and used as array indexers to enable array enumeration

```
string[] iceCreams = { "Vanilla", ..., "Raspberry Ripple" };
string[] tastySynonyms = { "Gorgeous", ..., "Yummy" };
for (int i=0;i < iceCreams.Length;i++)
{
    Console.WriteLine($"{tastySynonyms[i]} {iceCreams[i]}");
}
```

```
//output
Gorgeous Vanilla
Luscious Chocolate
Delectable Strawberry
Scrumptious Mint Choc Chip
Yummy Raspberry Ripple
```

Members can be modified

See Appendix for more for loop examples

QA

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 (Frowned upon because it could cause spaghetti logic. For more see appendix)

```
// 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
```

QA

The **break** statement ends the closest enclosing iteration statement.
In nested loops, the **break** statement terminates only the innermost loop.

NOTE: the keyword **return** is also a jump statement but this is used to terminate a function and return control to the caller.

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.

QA

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.

QA

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

QA

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

QA

A class defines the datatype of an object and specifies its core behaviour and data.
An interface defines extra capabilities of an object, typically extra behaviours that are implemented in the class.

The generic class **List<T>** is pronounced “List of T”.
The term “generics” comes from the type definition which itself is generic and uses a type “**T**” placeholder. When we specify the actual type for the type parameter, the compiler gives us a “specific” type.

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
```

QA

With the List<> class, you can *Add* a value to the end, *Insert* at a specific index, or *Remove* a value.

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
```

QA

The Dictionary class has two type parameters: one for the keys, *TKey* and one for the values, *TValue*.

You can iterate over the dictionary with a foreach loop. The iterator variable in the example is a *KeyValuePair<Tkey, Tvalue>* type.

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
```


QA

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

QA

SortedList and **SortedDictionary** differ only in implementation and therefore have different performance characteristics :

- SortedList<TKey, TValue> uses less memory than SortedDictionary<TKey, TValue>.
- SortedDictionary<TKey, TValue> has faster insertion and removal operations for unsorted data.
- If the list is populated all at once from sorted data, SortedList<TKey, TValue> is faster.

collection Operators

There are three useful operators for working with collections:

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

QA

The *index from end* and *range* operators were introduced in C# 8.0

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 = 84

//Range operator:
List<string> hotDrinks = drinks[1..3].ToList();
Console.Write("Hot Drinks: ");
hotDrinks.ForEach(hd => Console.Write($"{hd} "));
```

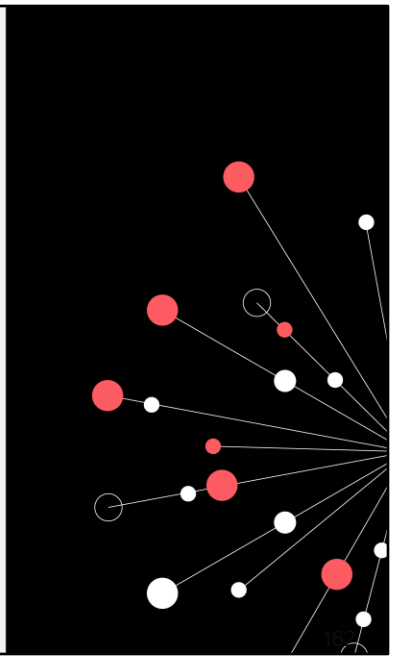

QA

You have seen this operator in previous examples.

Summary

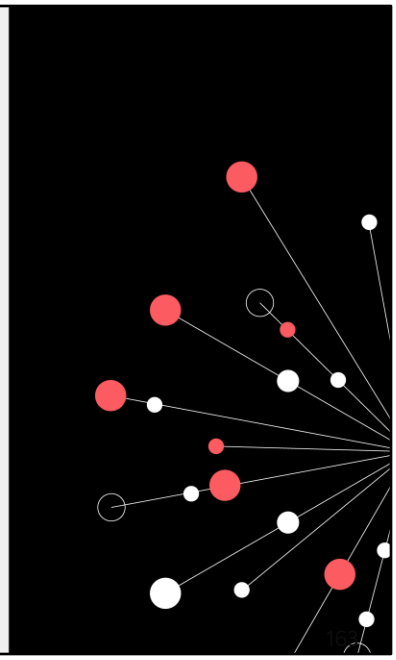
- Arrays
- Foreach loops
- For loops
- While loops
- Do loops
- Generic collections
- List<T>
- Dictionary<TKey, TValue>
- Collection operators

QA



ACTIVITY

Exercise 08 Arrays, Collections and Loops

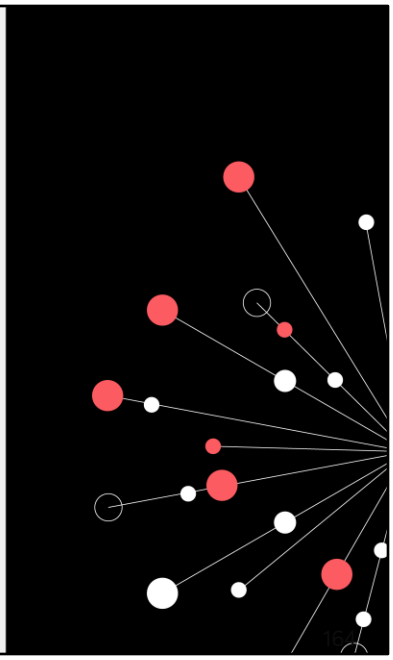


QA

Appendix

- For loop examples
- goto
- While loop examples
- Do loop examples
- List<T> examples
- Dictionary Iteration
- Index from end and range operators

QA



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
```


QA

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
```


QA

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
```


QA

For loop Examples: goto statement

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
```

QA

Use of the **goto** statement is discouraged however, it can be used to exit out of nested loops to jump to a statement defined by a label.

NOTE: the **goto** statement can also be used in switch statements to transfer control to a switch section with a constant case label, enabling multiple sections of the switch statement to execute.

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
```

QA

The first example is to demonstrate that you *could* write a **while** loop to behave like a **for** loop but the purpose of a **while** loop is to test a condition rather than to 'count' and execute a specific number of iterations.

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
```

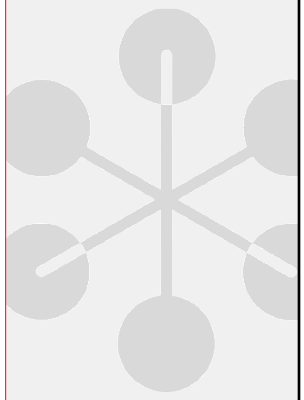
Do loops can be nested:

```
// 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
```

QA

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
```



QA

AddRange is used to add a range of objects to the list.
IndexOf returns -1 if the object is not found within the list.

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]
```


QA

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
```


QA

Index from End Operator and Range Operator

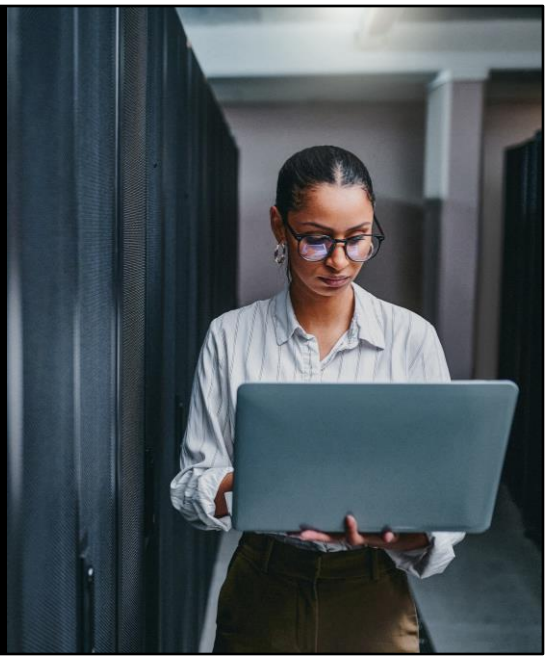
- The **index from end** operator **^** indicates the element position from the end of a sequence
- The **range** operator **..** specifies the start and end of a range of indices
  - The left-hand operand is inclusive
  - The right-hand operand is exclusive

```
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];
```


QA

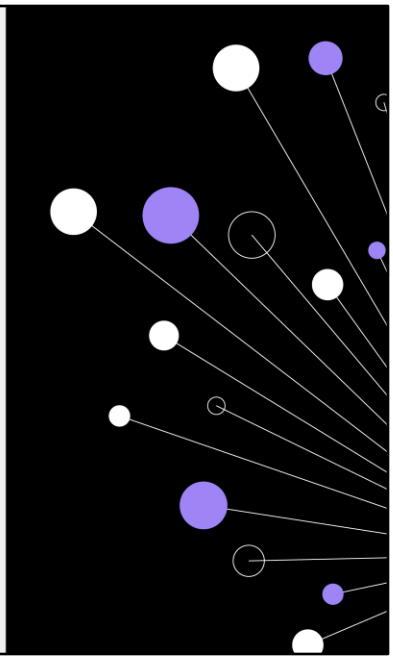
07 Unit Testing

C# The Programming Language Part 1



Learning objectives

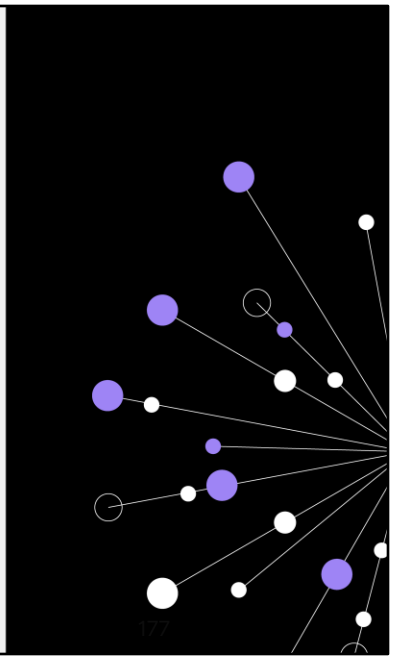
- To understand the purpose of Unit Testing?
- Know how to create and run unit tests



QA

Contents

- What is the purpose of Unit Testing?
- The green-red cycle
- Unit Testing Best Practice



QA

Traditional Development Cycle

Problem: High level of defects

- Lengthy testing phase after a release is frozen
- Cost of fixing bugs is far higher than if bugs are caught when introduced into code

Problem: Poor maintainability

- Legacy spaghetti code that "works"
- Can't be touched – fear of breaking

QA

Testing is not optional

s/w *always* gets tested

If not by you, then by the customer/user

- This may not be for some time after deployment

The longer the time delay between writing the bug and finding it, the more expensive it is to fix.

- Actually, it is not just more expensive, it is much, much more expensive.

Test-Driven Cycle

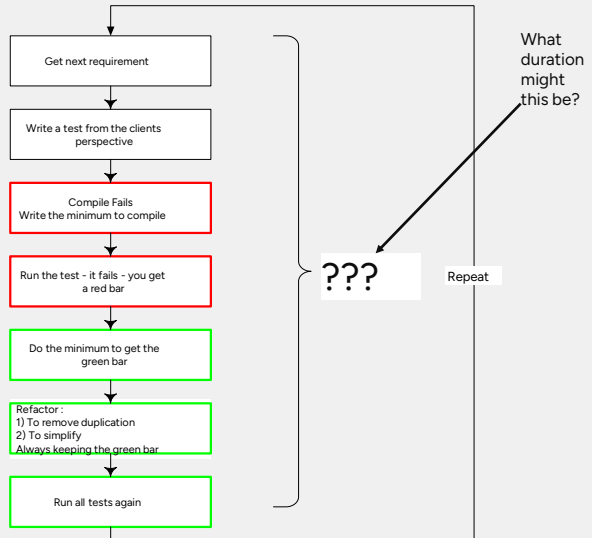
Note the process:

1) Write a test

2) Write the code

3) Run the test

4) Refactor



QA

The mantra of the unit test approach is that you only write operational code under one of 2 circumstances
You have written a test and you have not yet written the operational code to allow the test to even compile
b) The test compiles but does not work.
You write the absolute minimum code to allow the test to work.
It is entirely normal that at some later stage you return to the operational code and refactor it. Clearly after any refactor the tests must still pass.
This brings us to the second unit test mantra
You either write new functionality or refactor existing functionality. You never do both simultaneously.

The Unit Test Development Mindset

Stay on Green – if on Red stop all else & move to Green

There are only 3 conditions for writing operational code

1. Test doesn't compile
2. Test fails
3. Refactor behind existing tests

Only test the s/w through checked-in unit tests

Refactor relentlessly

Do all this in small iterations

The benefits are really for the developer

- You fix all the trivial problems as you go along
- You *know* that they have not recurred
- You document, without effort, how you see other s/w interfacing with your s/w
- You are able to refactor your code to make it more maintainable, faster, … knowing that you haven't broken anything.

Unit Testing Best Practices

Test naming

- Document test methods by a suitably descriptive name eg
- What_You_Are_Testing_AND_Expected_Outcome

Make unit tests as fine-grained as possible

- Test one object at a time
- One of the main purposes of a Unit Test is to quickly identify the reason for a failure – that's why we keep them small

Don't put more than one test into one TestMethod()

- The more complex a test becomes, the greater the risk that the test itself will need debugging

Keep tests independent of each other

- Implies tests are order-independent

Think AAA – see next slide

AAA

You should get in the habit of partitioning your test methods by:

```
//Arrange
//Act
//Assert
```

Arrange: Data for the Code-Under-Test is specified here along with the expected result(s)

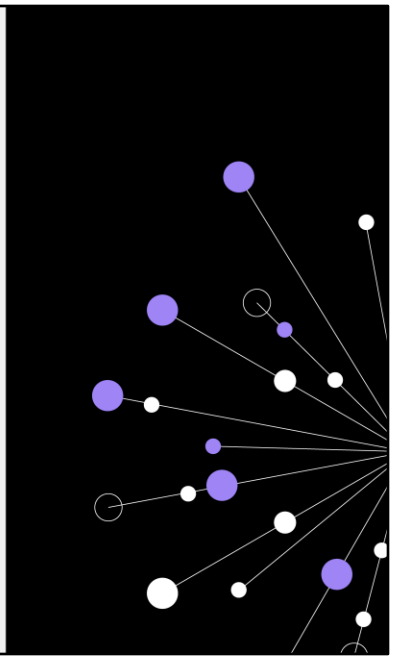
Act: The Code-Under-Test is called.

Assert – Use Assert methods to ensure result(s) is as expected

Summary

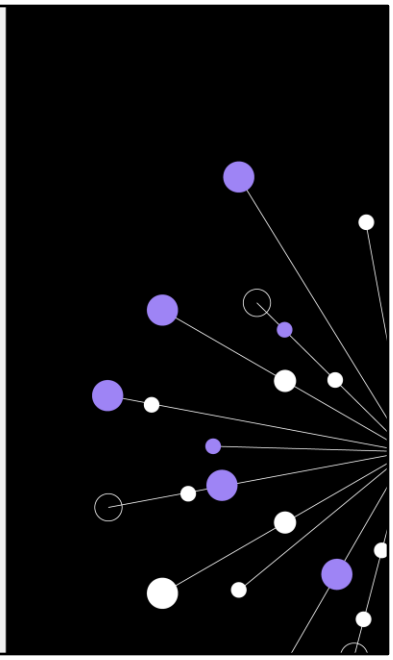
- What is the purpose of Unit Testing?
- The green-red cycle
- Unit Testing Best Practice

QA



Activity

Exercise 09 Unit Testing



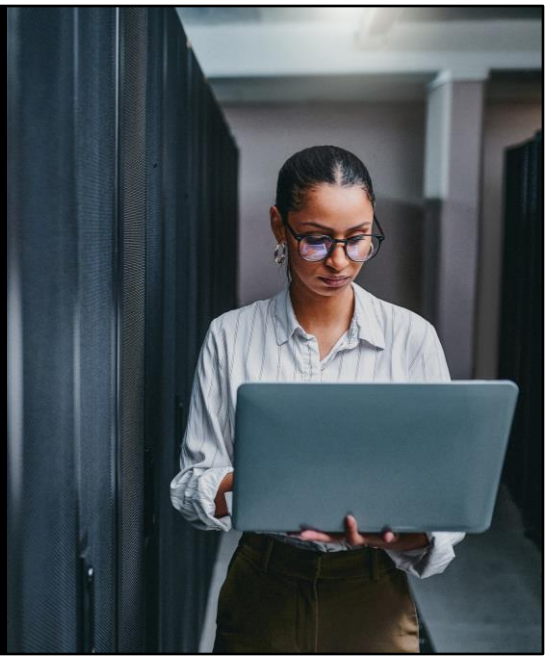
QA

08 Object Oriented Programming Concepts

C# The Programming Language Part 1

QA

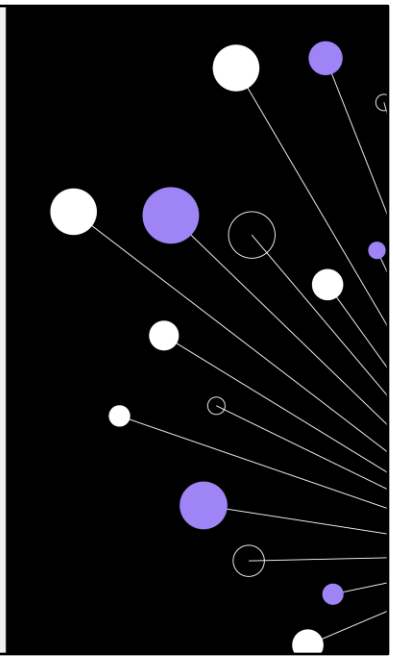
Learn. To Change.



Learning objectives

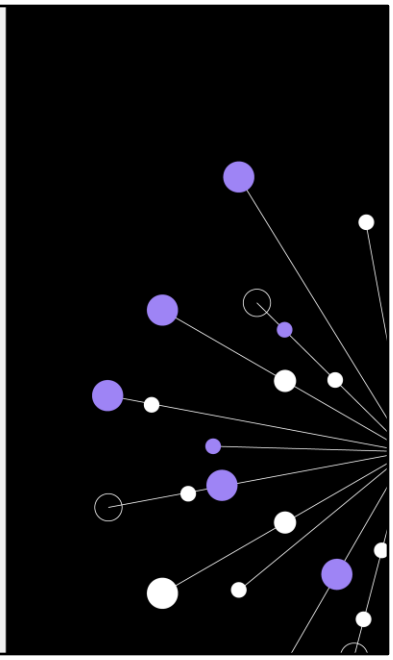
- Understand why Object-Oriented Programming (OOP) is such a useful paradigm
- Understand the Object-Oriented (OO) concepts of Encapsulation and Abstraction
- Understand how to define OO Data Types using classes
- Understand how to instantiate classes
- Gain a deeper understanding of Value and Reference Types
- Recognise why class variables may be null
- Know how to populate collections with class instances

QA



Contents

- Why Object Orientation?
- Four Concepts
- Classes and Instances
- Abstraction, Encapsulation
- Understanding the concept of an ‘object’ reference
- Distinguishing between value and ref type behaviour
- Null Reference



QA

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

QA

Object orientation and the principles of object-oriented programming have been around for a long time. The approach evolved out of programming best practices and simply formalises those practices with language constructs.

A key aim of OOP is to represent the "real world". In life, we don't interact with data structures or database records, we interact with Customers, Employees, Products and Places to name just a few of the "objects" with which we might interact.

Another key aim of OOP is the ease of maintenance: we can write code in one place and modify that one copy of the code. Other parts of software which consume our code will carry on as before, but they will be using the modified code.

C# is a thoroughly object-oriented language.

Four Concepts

Encapsulation

- Keeping related data and methods together

Inheritance

- Hierarchical structure of objects, being able to inherit methods from a base (parent) class with no need to redeclare them

Polymorphism

- Sub-classing allows for methods in derived classes to override methods in the base (parent) class such that they have the same signature but behave differently

Abstraction

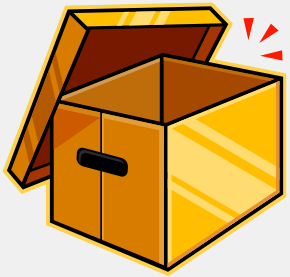
- The process of simplifying complex systems by focusing only on essential attributes and behaviours and hiding unnecessary details from the user/client.

QA

What is an object?

Objects in the real world:

- Car
- Tree
- Computer
- Desk
- Chair
- Rabbit



An object is 'something' that exists in the world

- Contains state and behaviour
  - Fields – information that describe the object
  - Methods – behaviour that the object can take

QA

Another way to look at it is that the nouns in a description would be your objects, verbs are methods in the objects, adjectives describe the object and these are the fields. This doesn't apply for all cases and fields!

Classes and Instances (Objects)

Rabbit Class

Fields
- Name
- Colour
- Breed
- Age
- Current Location

Methods
- Hop
- Eat
- Mischief
- Increase Age

Rabbit Instances

- Jimmy
- Grey
- Argente
- 2
- Hutch #2

- Nudge
- White
- Blanc de Hotot
- 3
- Hutch #7

QA

OO Fundamental – Abstraction (part of Design)

Ability to represent a complex problem in simple terms

- In OO, creation of a high-level definition with no detail
  - Detail added later in the process
- Factoring out common features of a category of data objects

Stresses ideas, qualities & properties not particulars

- Emphasises what an object is or does, rather than how it works
- Primary means of managing complexity in large programs

EXAMPLE:

Students (instances of type Student) attend a Course

- **Have 'attributes'- name, experience, attendance record**
- **Have 'behaviour' - Listen(), Speak(), TakeBreak() DoPractical()**
- **Are part of 'relationships'**
- **A student 'sits on a' course, a course 'has' students**

OO Fundamental – Encapsulation

Hiding object's implementation, making it self-sufficient

Process of enclosing code needed to do one thing well

- Plus, all the data that code needs in a single object
- Allows complexity to be built from (apparently) simple objects
  - Internal representation & complexities are hidden in the objects
  - Users of an object know its required inputs and expected outputs
  - Substantial benefits in reliability , maintainability and re-use

Objects communicate via messaging (method calls)

- Messages allow (receiving) object to determine implementation
- Sender does not determine implementation for each instance

```
foreach(Student s in myStudents){ s.DoNextLab(30);}
```

"I tell you how long you have, you sneak in the 'comfort' breaks"

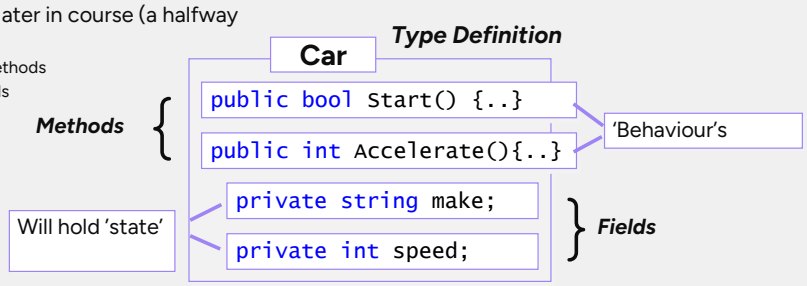
QA

What is an OO data type?

Definition of a data type as a single entity

- Fields - Constituent data parts
- Methods behaviour - Functions that define
- Properties – covered later in course (a halfway house)
  - When written, feel like methods
  - When used , look like fields

Fully explored later



QA

Types define the content and the behaviour of objects. It is a blueprint which the CLR will use to determine what memory needs to be allocated at run-time, and which the compiler will use to validate that your code is correct.
Types consist of three members: Fields, Methods and Properties.
Fields define the data elements from which your types are composed. For example, each and every car has a make and a model; these would therefore make ideal fields in a program that let's users browse car information on the Web.
Methods are the functions (or operations) that determine the behaviour of instances of the type, or of the type itself. Looking at our car example again, methods might include starting the car, steering it, accelerating (and braking!) and parking.
Properties are really nothing more than a pair of get and set methods that are designed to make the type easy to use whilst supporting encapsulation (which is something that we'll examine in more detail later on). For example, each car will need to have a field in which its speed is stored, but we might not want to allow just any code to access that field. Not only that, an external agency shouldn't be able to set the speed directly anyway (that is a by product of accelerating or braking), so we would want to implement a read only property that could return

the value of the field.

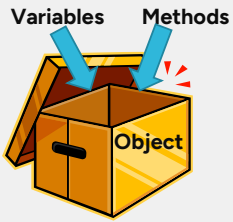
What are C# Objects?

C# objects are instances of something

- Anything that could be a real-world object can be a C# object
- Contain fields which describe the object (variables)
- Contain methods which are actions the object can take
- Contain Properties which are halfway between fields and methods

We can think of an object as being a big empty box

- Fields, methods and properties can be placed inside the box
- Methods and properties can interact with anything else inside the box

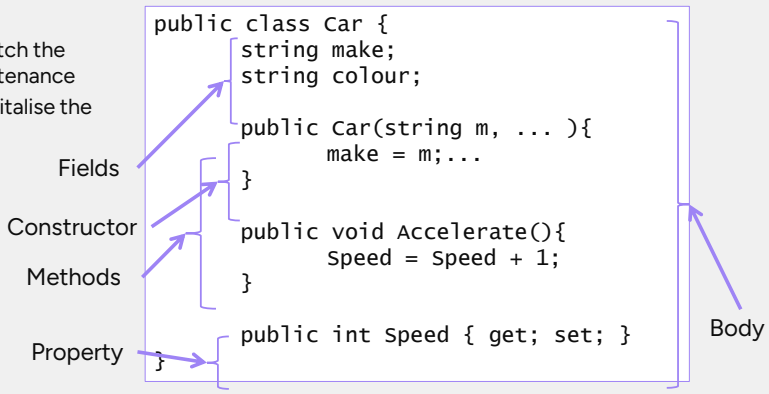


QA

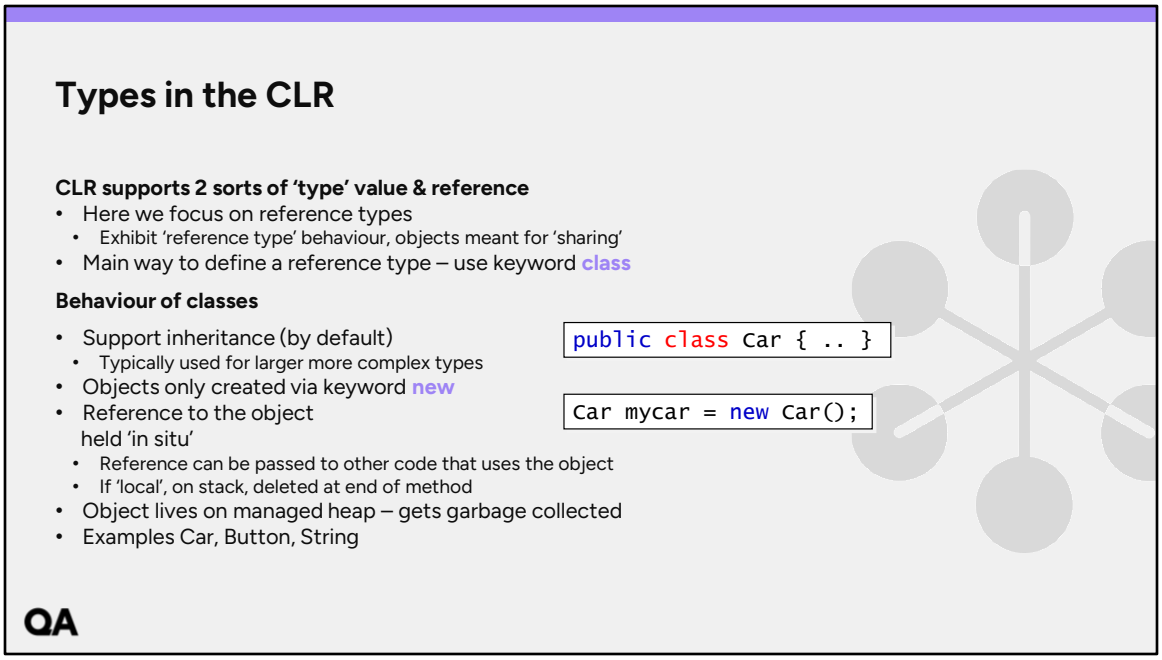
C# Class

A C# class is a description of an object

- It contains the fields, methods and properties
- The class name should match the filename (for ease of maintenance
- Naming convention is to capitalise the words in a name



QA



The CLR supports two different forms of types. These are known as value and reference types, because of their memory allocation and variable passing behaviours. We focus in this chapter on reference types defined using the C# keyword class. So how do you define new reference types in C#? The answer to this is mainly by defining new classes (reference types) using the class keyword. Generally speaking, reference types are efficient for large objects, because the CLR only has to copy the reference when they are passed in as arguments to methods. Value types, on the other hand, are particularly useful for smaller objects (typically less than 16 bytes). You also see them used where a developer would expect value type semantics, such as with the Point, Rectangle and Color classes used in GDI+; you wouldn't expect changing the RGB values of a Color of a circle to change all instances of the colour!

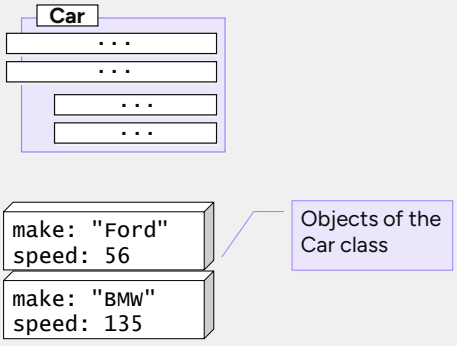
Classes and Objects

class definition is a blueprint, a 'plan' for making *objects*

- Architect can draw up plans, but you can't 'move in'
- Need to create an instance (a house) – then 'move in'

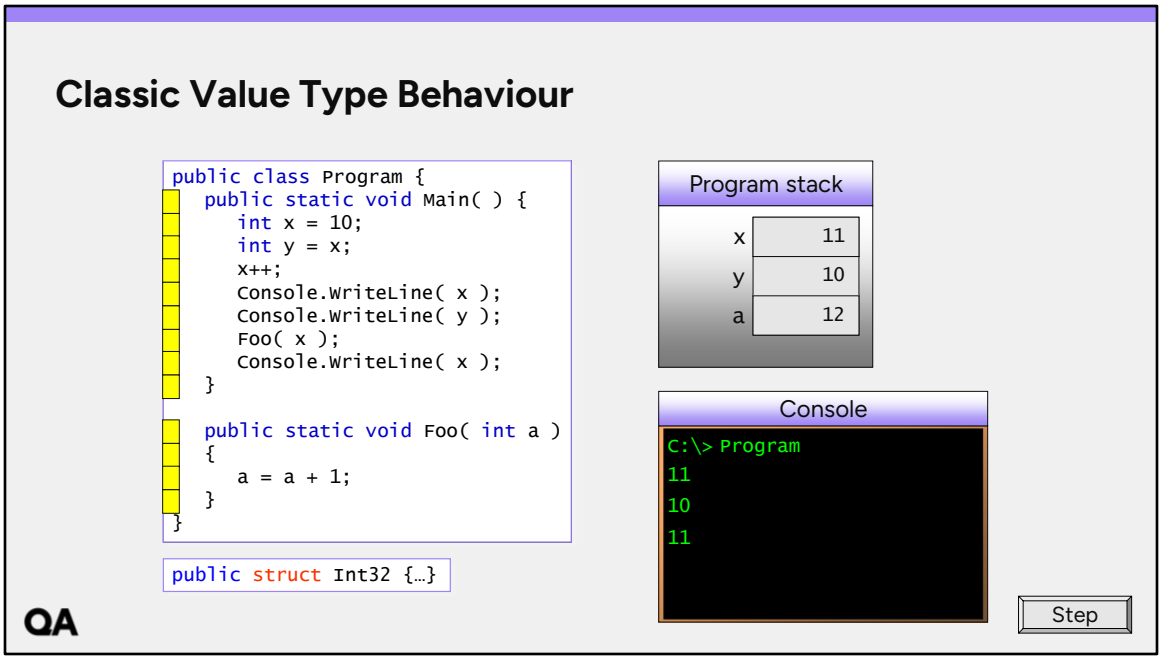
Objects are unique instances of a class

- Have their own *state* (values for their fields)
- Have their own *identity* (address in memory)
- Structure & behaviour of similar objects is defined by their class



QA

Remember, the type definition is simply a blueprint of methods, fields and properties. Objects, on the other hand, are instances of the type. They have their own state (the values in the fields) and their own identity, which in the world of a computer program is the address in memory that the object is stored in.

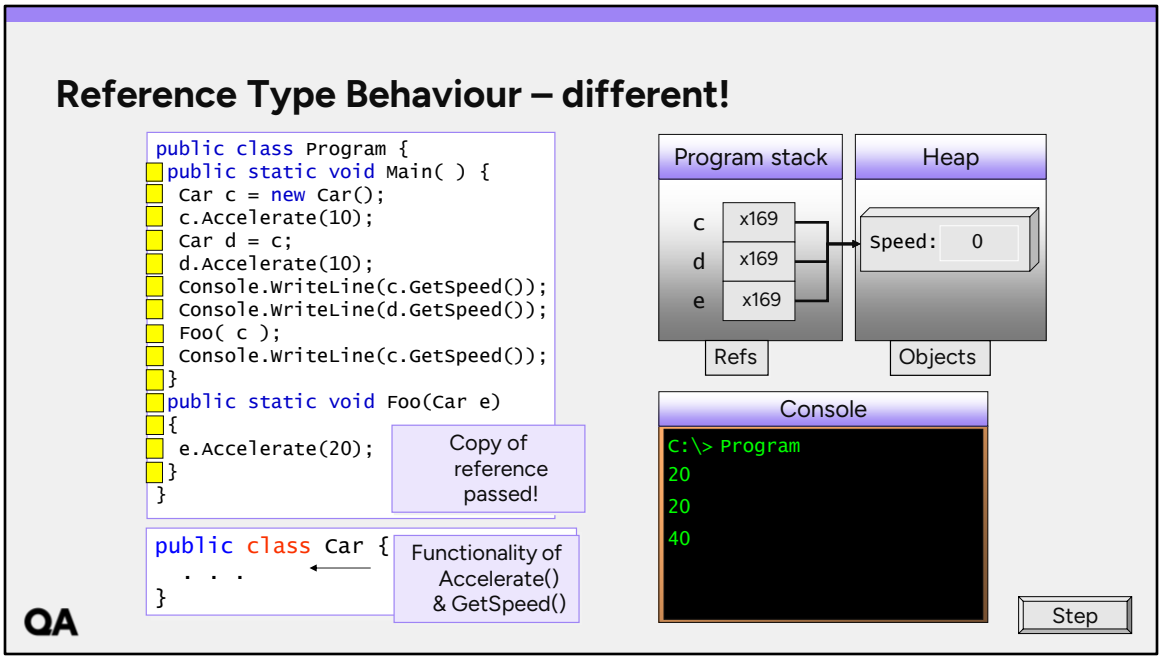


Value types behave very logically. Value types have their memory allocated in situ, which in the case of local variables means on the call stack. When you perform an assignment operation, the contents of the value type is copied to the destination variable, overwriting its contents.

This means that in the code above, when we declare the variable y and assign x into it, the contents of x (10 in this case) are copied into the storage for y. Any operations that are subsequently performed on x will therefore have no effect on the variable y.

Value types are named because of their "pass by value" semantics when they are passed as arguments to method calls. In the call to the method Foo(), a copy of the value type x is passed through to the method. This copy is then accessed using the symbolic variable name a. Operations performed on a have no effect on the variable x, because a is a copy of x. The copy is discarded (and the memory on the stack reclaimed) when the method Foo() returns.

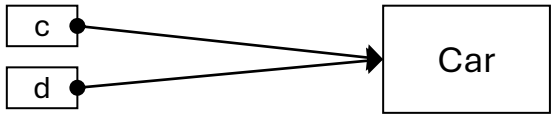
You can see the output for the program in the console window on the right hand side above. Initially, x starts at 10. x is assigned into y (which will now also have the value 10). x is incremented, so now holds the value 11. The method Foo() is called, but x is unaffected by the call, so it still has the value of 11 at the end.



Reference types are a little bit more subtle than value types. A local variable of a reference type consists of a reference, which is on the stack. This variable refers to (in C++ they might have said points to) the actual object, which is allocated on a heap.
In our example above, c is a reference to a Car object that is allocated on the heap. Using this reference, we set the Car object's speed field to the value 10 (via acceleration). The code then declares another reference, d, into which c is assigned. It is here that the main behavioural difference between value and reference types starts to show up: only the reference is copied, not the object. Therefore, the result of the line of code

```
Car d = c;
```

is that there are now two references, c and d, both referring to the same object, as shown in the diagram below:



From this you can see that no matter which reference you use to access the object, you will see the changes when you access the object with the other reference.
It also follows that when Foo() is called, only a copy of the reference is passed to the method, so e also refers back to the same Car object on the heap. Therefore, any changes made using the reference e will be visible via reference c (as the changes occurred on the object referred to by c).

The null Reference

Variables of reference types may be set to null

- The variable does not reference an object
- If variable did reference an object, then old object is 'forgotten'

Can compare an object reference with null

```
public class QA {
  public static Car FindPoolCar() {
    Car aCar = null;
   // attempt to make 'aCar' point to available Car object
    return aCar;
  }
}
```

```
Car car1 = QA.FindPoolCar();

if( car1 != null )  {
  // Drive the car away
} else {
  Console.WriteLine("No car available");
}
```

QA

The default value for a variable whose type is a reference type (think class for now) is null. (provided it is not a local variable as then it is un-initialised like every local variable)
A variable 'theCar' above is initialised with a special reference called null, which indicates that the reference doesn't refer to any object. null is a keyword in the C# language, so you can use it with the equality operator to check whether an object reference has been initialised or not.
When you have finished using an object, you can set its object reference to null, but typically you just allow the reference to go out of scope. If there are no other references to the object, then the object becomes available for garbage collection.
null is meaningless for value types, as they have no reference capability.

Arrays – revisited

All array variables are reference variables

```
public class Car {...}
```

Assuming this class defined

```
int num = 0;
int[] nums1 = new int[3];
int[] nums2 = { 3, 5, 7, 9};
Car[] cars1;
Car[] cars2 = new Car[3];
Car[] cars3 = {new Car(),
               new Car(),
               new Car()};
```

'num' is a value type = 0

'nums1' is a ref type, 3 zeros in

'nums2' is a ref type, .Length = 4

'cars1' is an un-initialised reference variable

'cars2' is a reference variable, .Length = 3
but contains 3 nulls & no cars!!

'cars3' - a reference to
an array of car references

```
ProcessIntArray(nums2);
ProcessCarArray(cars3);
```

Pass a ref to any array – by value (see demo)

QA

The 2 sample methods ProcessIntArray(int[] numbers) & ProcessCarArray(Car[] cars) would typically be void, they don't need to return anything as they can see (and change) the contents of the array passed to them.
"It is not possible to pass an array of cars...", "only an array of car references" which the receiving code can iterate over using either a foreach or a for loop using .Length.

Lists – revisited

All List variables are reference variables

```
public class Car {...}
```

Assuming this class defined

```
int num = 0;
List<int> lnums1 = new List<int> ();
List<int> lnums2 = new List<int> { 3, 5, 7, 9 };
List<Car> lcars1;
List<Car> lcars2 = new List<Car>();
List<Car> lcars3 = new List<Car> { new Car(),
                                   new Car(),
                                   new Car()};
```

'num' is a value type = 0

'nums1' is a ref type, `.Count` = 0

'nums2' is a ref type, `.Count` = 4

'cars1' is an un-initialised reference variable

'cars2' is a reference variable, `.Count` = 0 but contains no cars!!

'cars3' - a reference to a collection of car references

Pass a ref to any collection – by value (see demo)

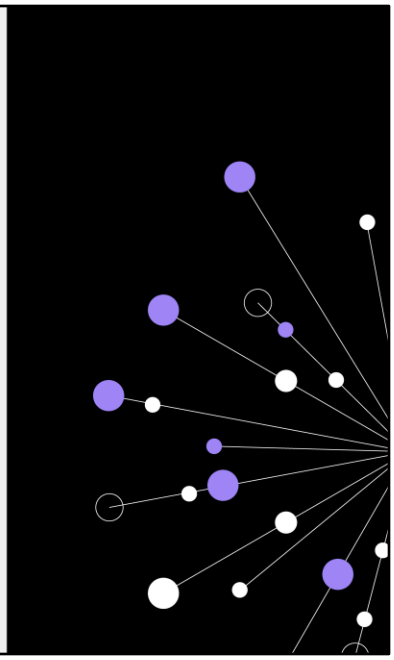
```
ProcessIntList(nums2);
ProcessCarList(cars3);
```

QA

The 2 sample methods ProcessIntList(int[] numbers) & ProcessCarList(Car[] cars) would typically be void, they don't need to return anything as they can see (and change) the contents of the array passed to them.
"It is not possible to pass a List of cars…", "only a collecton of car references" which the receiving code can iterate over, using either a foreach or a for loop using .Count.

Summary

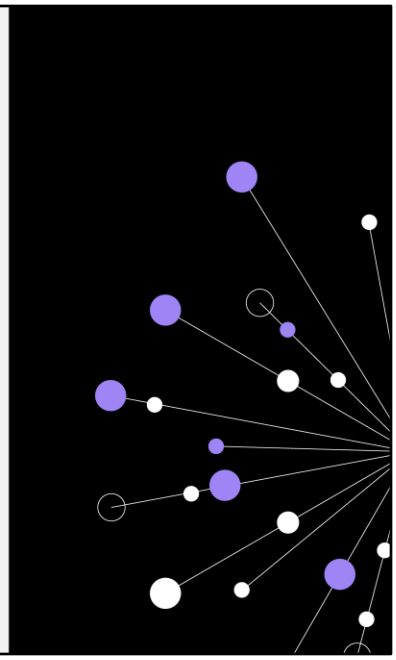
- Why Object Orientation?
- Four Concepts
- Classes and Instances
- Abstraction, Encapsulation
- Understanding the concept of an ‘object’ reference
- Distinguishing between value and ref type behaviour
- Null Reference



QA

ACTIVITY

Exercise 07 Object Oriented Programming Concepts

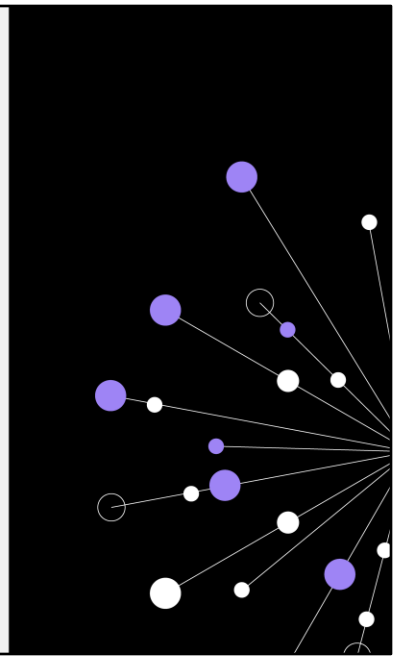


QA

Appendix

Parameter passing ("out" and "in")

Memory Management with the garbage collector



QA

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
```


QA

Use **out** parameters to output multiple values from a method.

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);
```

QA

From C# 7.0, you no longer need to declare the **out** variables outside of the method call. You can declare the **out** variables in the argument list of the method call.

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.

QA

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;
}
```

QA

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;
}
```

QA

The **in**, **ref**, and **out** keywords are not considered part of the method signature for the purpose of overload resolution.

Memory Management in the CLR

The CLR manages memory for you
- Reference types use memory allocated from the heaps
- CLR keeps track of references to objects

CLR uses a *garbage collector* to reclaim memory
- Background thread(s) that compact heap
- Object can be collected if no extant references exist
- Garbage only collected when necessary

Effects of CLR memory management on your code
- Very fast allocation strategy (much quicker than C++ heap)
- Cannot leak memory for objects
- Non-deterministic algorithm; you don't know when GC runs

QA

The CLR allocates memory from one of its heaps* for objects when you create them with new. Obviously, this only applies to reference types; value types are allocated memory directly on the stack. The CLR then tracks all of the references to the objects on the heap that exist within your code.

From time to time, the CLR will decide that it needs to reclaim some memory from the heap. It performs this task using a garbage collector that rapidly scans through the references and determines if any of the objects are unreachable from your code (i.e. all of the references to that object have gone out of scope). The garbage collector can then compact the heap reclaiming the memory from the objects that are no longer accessible.

To improve performance, .NET uses a multi-generational garbage collector. This means that it is very good at reclaiming memory from short lived objects, and doesn't bother to check the longer lived objects every time. It is therefore possible for an object to survive in memory for some time after the last reference to it has been removed, although it will get collected eventually.

What this all means is that memory allocation is very fast in .NET and that it is theoretically impossible to leak memory for a completely managed object (unless the CLR has a bug!). However, the downside of this is that you can never determine when an object will be removed from memory, nor when the GC (which is self-tuning), will run.

* There is one heap for small objects and one for large objects (objects ~ 80Kb or larger).

Possible Problems with GC

Types might also require *guaranteed clean up*
- O/S files need to be closed to flush stream
- GDI+ handles have to be closed

Certain resources need to be released *deterministically*
- Database connections need to be closed to support pooling

CLR and FCL provide support for both
- Finalizer for guaranteed clean up
- Dispose pattern for deterministic resource release

QA

The use of a garbage collector (GC) poses some problems for us, particular when you start to use objects that encapsulate unmanaged resources, such as database connections, file handles, sockets and low level GDI or GDI+ objects. These objects use operating system resources that are written in native code, and the GC is unaware of pressure on these resources.

There are times when you need an object to guarantee clean up of an unmanaged resource. For example, it is all well and good reclaiming the memory from the heap for a File object, but you also need to ensure that the low level O/S file handle is closed (with the Win32 API CloseHandle), otherwise the content of the file will be lost! In these cases you must have code that will execute as part of the object's destruction process.

Let's consider another example where it is important to be able to release a resource in a deterministic manner. The SQL Server managed provider in ADO.NET supports connection pooling of low level (native) connections, with a default pool maximum of just 100 connections. Therefore, if you have no way to tell the pool that you had finished with the object, and you simply relied on the GC to release the resource, a busy ASP.NET application might soon run out of connections. In this case you need a way to tell the object to release its contained unmanaged connection back into the pool as soon as you have finished with it, not when the GC runs.

Fortunately, the CLR and the FCL provide support for both of these requirements through the use of Finalizer methods and the Dispose Pattern.

The Finalizer Method

Purpose is to release resources owned by an object

- Called by the GC except under truly disastrous circumstances
- Necessary if object *directly interacts* with unmanaged memory!

```
public class Font {
   private Font() { ... }
   public static Font CreateFont( string name, int size ) {
      Font f = new Font();
      f.handle = NativeCode.CreateFont( ... );
      ...
      return f;
   }
   ...
   ~Font() {
      NativeCode.SafeReleaseHandle( ref handle );
   }

   System.IntPtr handle = IntPtr.Zero;
}
```

C# Finalizer - NOT a destructor

Code ends up in a try / finally

QA

In C#, a class can define a finalizer method which is used to clean up when the object dies. This is not a destructor (in the C++ sense of the word), as it can not be called deterministically, nor is it automatically called at predefined locations in code. Rather a garbage collector thread calls it when it needs to free up the memory of the object.
In C# the finalizer is written using a syntax that will be familiar to C++ developers. In reality this is actually an override of the Object.Finalize method, and you can check this out if you examine your code with ildasm.exe. Note that you can only override Object.Finalize using this syntax.
A finalizer should be used to free the additional resources held by the object, such as open files, window handles, database connections etc. C# manages memory automatically, so an object does not need to explicitly free up memory unless that memory was allocated by your object using unmanaged (native) APIs.

Dispose Method

Finalizer only called when object is garbage collected

- GC only happens when managed memory is scarce
- GC is not triggered when other resources become scarce

Some resources must be explicitly released

- Provide a method for users to call, typically called `Dispose`
  - Other obvious names may be used, such as `Close`

```
public class Font {
  public void Dispose() {
    ...
  }
}
```

```
public class ChessBoard {
  public void Show() {
    Font f = Font.CreateFont(...);
      ...
    f.Dispose();
  }
}
```

QA

The finalizer method will be called automatically when the object's memory is garbage collected. Unfortunately, as we have already seen, there is no guarantee as to when this will happen or that it will happen before the program exits. Clearly, this is unacceptable if resources are scarce. A manual mechanism has been added in the guise of a design pattern using the IDisposable interface and the Dispose method.

The class which encapsulates a resource should provide a method for disposing of the resource. The method could have any name, but for consistencies sake, and as we shall see later to enable the compiler to automatically generate calls, should be called Dispose. Certain resource types also tend to provide a Close method to perform the same task; developers somehow seem to be more comfortable closing a file rather than disposing of it!

Golden Rules:

- If a class has a finalizer it is generally considered to be good practice to provide a Dispose method.
- If a class has a Dispose method to release an unmanaged resource owned directly by that object then it must have a finalizer, as you can't rely on the developer that uses your class remembering to call Dispose.

Compiler Support for Dispose

Programmer has to remember to call `Dispose`
- Forgetting to do this can cause resource problems

The compiler can help
- The `using` statement ensures `Dispose` is called
- Irrespective of how the statement is exited, including by exception
- Only for objects which implement `IDisposable`
- No need to explicitly call `Dispose`

```
public class MyResourceHolder : IDisposable {
   public void Dispose() { ... }
```

```
using( MyResourceHolder mrh = new MyResourceHolder() ) {
   ...
}
```

Avoids nesting try / finally's

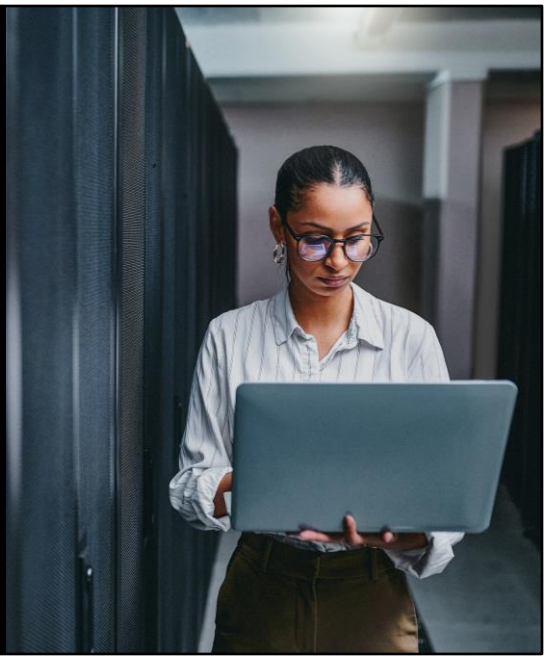
QA

It is very easy for a developer to forget to call Dispose on objects, especially when exceptions are being thrown. The only safe way to use such objects is to use them within a try {} / finally {} block, but this can get very tedious to code.
To simplify the usage of classes that implement Dispose, the C# compiler can generate the try {} / finally {} code for you if you add a using statement block to surround the code where the object will be used.
Note that this use of using has nothing to do with namespaces.
The using statement defines a block of code, at the end of which the resources are no longer required. The compiler therefore ensures that the Dispose method gets called. The compiler needs to know that the object has a Dispose method, which it does by checking that the type implements the IDisposable interface. This interface only has a single method: Dispose.
Let's take a look at how you use using over the page.

09 Methods, Properties & Constructors

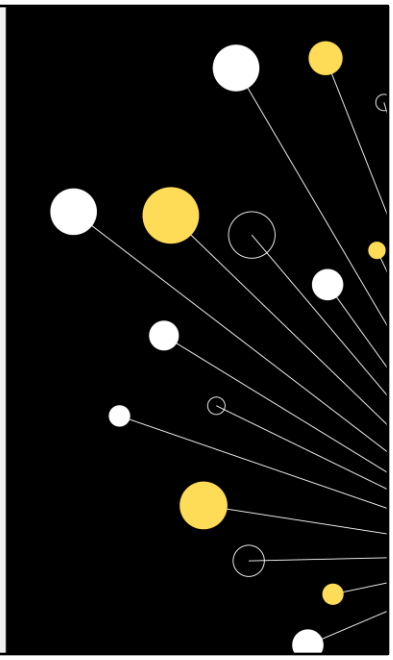
C# The Programming Language Part 1

QA
Learn. To Change.



Learning objectives

- To understand the difference between static and instance methods
- To know why properties are important, their different types and when to use them
- Be able to define, overload and chain to different constructors and initialisers

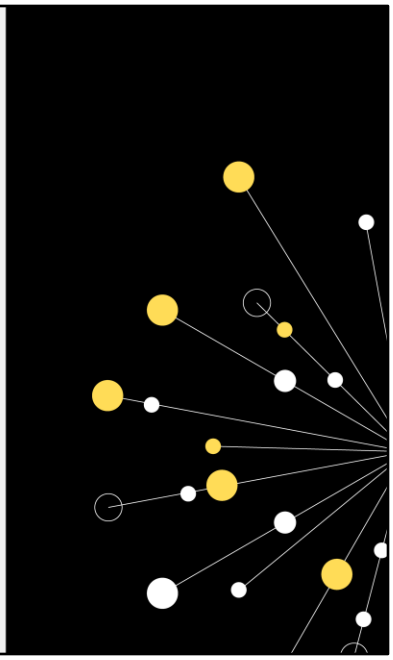


QA

Contents

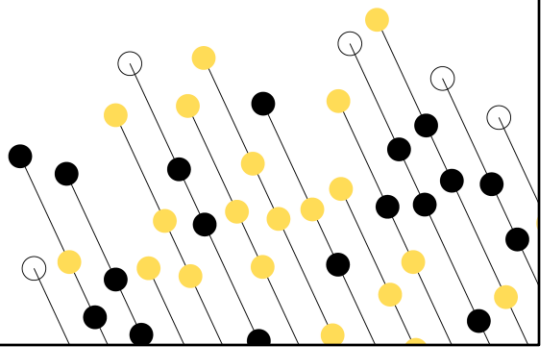
- Instance vs. Static
- Properties
  - Properties with backing fields
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Object initialisers
- The ‘this’ keyword

QA



Instance vs. Static

Methods, Properties and Constructors



Instantiating an object

The class file is just the description of how the object is to be created and what it contains.

We need to instantiate the class and create an object before we can use it

- This uses the **new** keyword
- Calls the constructor in the class

```
Class variableName = new Class();
```

There is only one class, but there can be many objects created from that class

QA

Instantiating an object

```
public class Car{
      String make;
      ...
}
```

```
Car oldBanger = new Car();
Car jalopy = new Car();
Car buggie= new Car();
Car wheels= new Car();
```

QA

Methods – a Reminder

- 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

Note: In an application that uses top-level statements, the Main method is generated by the compiler.

Instance members – Methods, to exhibit functionality

Most methods operate on instances of a type

- Referred to as instance methods as each instance 'has' them
- Can't invoke them though unless you have an instance

Methods directly access instance variables

```
public class Car {

  private int speed;

  public void Accelerate( int weightOfFoot ) {
    speed = speed + ...;
  }
   ...
}
```

Field has 'class' scope but will belong to car 'objects'

No sign of keyword 'static'

QA

Methods are defined in a type and operate on instances of that type. Inside a method, you can declare local variables. Note that the scope of a local variable is limited to the block in which it is declared. This means that if you declare a local variable within a method, if cannot be accessed from outside of that method.

Invoking an Instance (non-static) Method

Use dot operator with a reference to the object

```
objectRef.MethodName(...)
```

```
public class Car {
  public void Accelerate(
      int weightOfFoot ) {
    ...
  }
  ...
}
```

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

Non static!!

Calls the `Accelerate()` method of the object referenced by `car1`

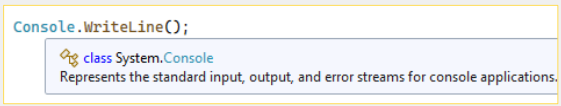
```
Car car1 = new Car();
car1.Accelerate( 5 );
Car.Accelerate( 5 );
```

QA

You use the dot operator to call an instance method of an object. Don't forget to include empty parentheses if the method has no arguments.
In the example on the slide, a new instance of the Car class is created. The Accelerate method is then called on that object, passing in 5 as the argument to represent how much acceleration the car should 'do'.

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



QA

228

A static class is a class that cannot be instantiated. You might want to write a class that only contains static methods or properties such as the framework-provided Math class.

A static class is restricted to having static members only so no instance methods, properties or fields are allowed.

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

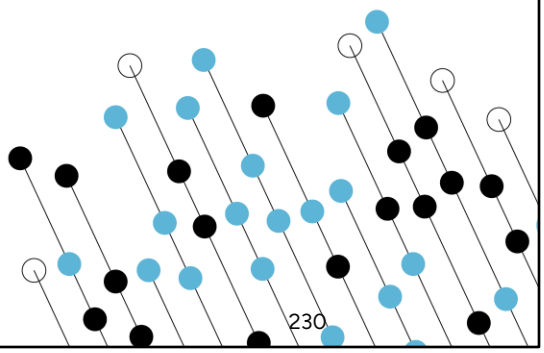
```
internal class PassingParams
{
    1 reference
    public static void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

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

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

See Appendix for more examples

Fields and Properties

Methods, Properties and Constructors



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

```
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
```

QA

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**)

QA

232

It is common to provide validation within the set accessor and perhaps conversion or computation within a get accessor.

The set accessor has a special parameter, called **value**.

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; }
    }
}
```

QA

234

The setter or the getter can be marked with a different visibility to the rest of the property to provide encapsulation. In the example, the get accessor (getter) for Speed is public whilst the set accessor (setter) is private. This means that code within the Car type can change its speed but no other code can.

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
{
  public double TempInDegreesCelcius {get; set;}

  public double TempInDegreesFarenheit
  {
    get { return TempInDegreesCelcius * 1.8 + 32; }
  }
}
```

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

```
public  class Temperature
{
  public double TempInDegreesCelcius {get; set;}

  public double TempInDegreesFarenheitAlt
    => TempInDegreesCelcius * 1.8 + 32;
}
```

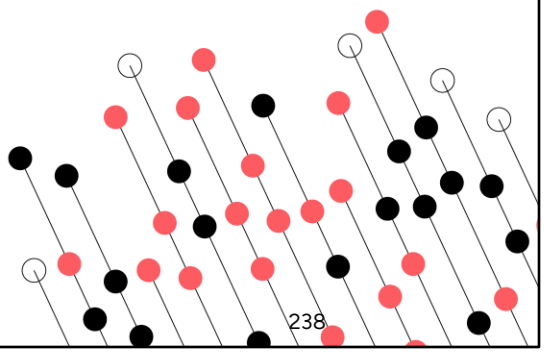
Accessing properties

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

Methods, Properties and Constructors



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

QA

Constructors

- A constructor is called whenever a class or struct is created
  - 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

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

Constructor Example: Setting properties

You have an employee class with three mandatory values

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }

    public Employee(string firstname,
                    string lastname,
                    int employeeID)
    {
        FirstName = firstname;
        LastName = lastname;
        EmployeeID = employeeID;
    }
}
```

See Appendix for more ways to define constructors

QA

241

The properties enable the 3 values to be accessed outside of the class. You also have the option of replacing the automatically implemented behaviour with your own code, if the need arises. For example, you may decide to output the **LastName** in uppercase.

Constructor Optional Parameters

A constructor is a method and can therefore specify optional parameters

```
public class Car
{
    public Car(string make="Ford",
               string model="Fiesta")
    {
        Make = make;
        Model = model;
    }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

See Appendix for more ways to define constructors

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

```
// parameter-less constructor used implicitly with an object
initializer
Car c1 = new Car { Make = "Audi", Model = "TT", Speed = 70 };
Console.WriteLine($"Car {nameof(c1)} is make: {c1.Make} " +
                  $"and model: {c1.Model} and Speed: {c1.Speed}");


// an explicit constructor can be used with an object initializer
Car c2 = new Car("Audi", "TT") { Speed = 70 };
Console.WriteLine($"Car {nameof(c2)} is make: {c2.Make} " +
                  $"and model: {c2.Model} and Speed: {c2.Speed}");
```

QA

243

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

Note: In C# 10 and later, you can explicitly declare a parameterless constructor in a struct. In C# 9.0 and earlier, the compiler always produced an implicit parameterless constructor that initialized all fields and properties to their default value. For example, **0** (zero) for numeric types, **false** for bools and **null** for reference types.

Where should our ‘new’ type be defined?

So far – largely working with Console Applications

- Compile into .exe’s
- Coding mostly in ‘Main’ method of class Program (Occasionally invoking other helper methods of your class)
- Where in Visual Studio do we define class Car, class Person etc?
- If they are in a .exe, are they reusable functionality – no!

Welcome to class libraries (dll’s)

- Project type ‘Class Library’ (in new project dialog)
- The FCL is a large bunch of .dll’s (you add new dll’s to these)
- Are they all magically visible?
  - No, you need a reference to a .dll to ‘consume’ it’s types

QA

Before considering visibility modifiers like public, private we need to think first about where a type will ‘live’
If we just stick class Car in file Car.cs alongside class Program (with its entry point Main) of Program.cs then the compiled code (the IL) of Car is now living in a .exe file.

Is this class reusable outside this .exe? i.e. Can another .exe use this .exe to use Car?
No
Car and maybe Train and Scooter etc perhaps defined in a xxx.xxx.Vehicles namespace should live in a project of type Class Library that compiles into a .dll. An exe, any exe, multiple exe’s can then reference and use that dll and ‘consume’ its types.

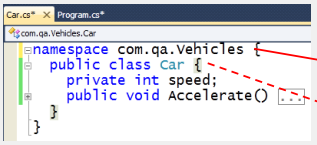
Lets see how.

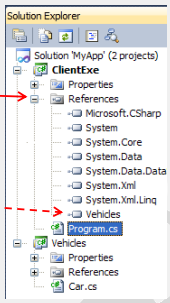
Using types defined in other Assemblies

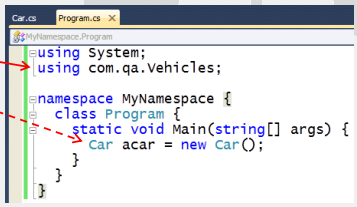
Need a reference to the other Assembly
- Right click for 'Add Reference' dialog

Define your class - 'Car'
- In your namespace - 'com.qa.Vehicles'

`using` statement
- Enables easy use
- Use last part of name only







QA

For a client 'exe' application to consume types (classes) defined in other .NET Assemblies (physical files - 'dll's) then the Client project needs a 'reference' to the 'library' assembly.
In the example above class 'Car' is defined in the Vehicles project that compiles into Vehicles.dll. But 'Car' is defined in the 'com.qa.Vehicles' namespace. This means its full name is com.qa.Vehicles.Car.
By right-clicking References in the ClientExe project an 'Add Reference' dialog appears allowing selection of any installed framework .NET dll and also any Class Library project in the open solution.
Only when a reference exists to a library can you start to use 'using' any namespaces defined in that library.
In the example above the 'using com.qa.Vehicles;' statement enables the Intellisense to show 'Car' as a visible datatype when writing code in the 'Main' method of class 'Program' in the ClientExe Assembly that has a reference to the Vehicles.dll (where com.qa.Vehicles.Car is defined).

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

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

```
class Employee
{
    private readonly string firstName;

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

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

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

QA

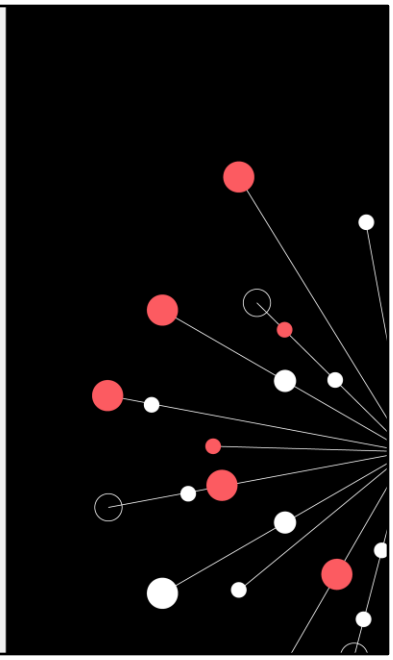
In an ‘instance’ context there is always a ‘this’
It is a reference to the object on which the method was invoked.
It is often referred to as the ‘hidden’ first parameter (of an instance method).

Note: the **this** keyword is also used to define *indexers* and as a modifier of the first parameter of an *extension method*.

Summary

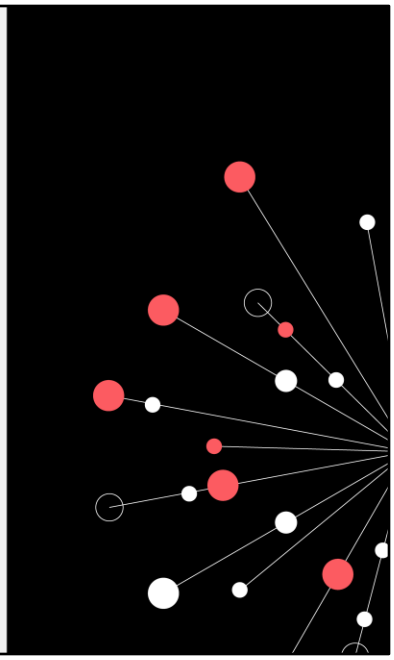
- Instance vs. Static
- Properties
  - Properties with backing fields
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Object initialisers
- The ‘this’ keyword

QA



Activity

Exercise 08 Methods, Properties and Constructors

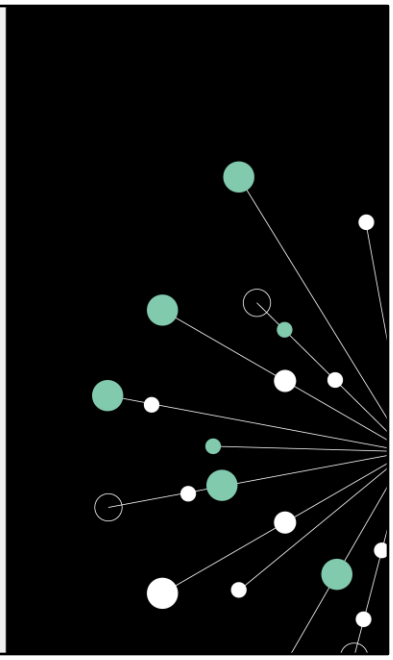


QA

APPENDIX

- Static Methods examples
- Expression Bodied Properties
- Constructor Examples

QA



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
```

QA

250

The **System.Math** class provides constants and static methods for common mathematical functions. This example has a using directive:

using System;

The static methods in the Math class must be prefixed with the class name.

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
```

QA

This example has a using directive:

using static System.Math;

The static methods in the Math class do not need to be prefixed with the class name.

The 'using static' directive operates on a class rather than a namespace. It puts all the static members of that class directly into scope.

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;
    }
}
```

QA

252

You omit the **return** keyword in the getter when using expression body syntax.

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;
    }
}
```

`this` refers to the object on which field belongs

QA

253

The **readonly** modifier on the fields means that the fields' values can only be set in the constructor. Once the construction of an instance of the class is completed, the data within that instance's fields cannot be changed.

Fields do not provide any validation or conversion capabilities.

The fields are private and therefore not accessible outside of the class unless you provide a method to make these values readable. Typically, you would achieve this using the **ToString** method.

The **this** keyword disambiguates the parameter name **firstName** to the field belonging to the object instance **this.firstName**

Note: You can create a constant field using the **const** keyword. A constant must be initialized at the point of declaration whereas a **readonly** field can either be initialized at the point of declaration or in a constructor.

A **const** is a *compile-time* constant. A **readonly** field is a *run-time* constant.

Constructor Example: 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; }
}
```

QA

254

The read-only properties enable the 3 values to be accessed outside of the class. You also have the option of replacing the automatically implemented behaviour with your own code, if the need arises. For example, you may decide to output the **LastName** in uppercase.

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; }
}
```

QA

You have an employee class with 3 mandatory values that should not be able to be changed once set: **firstName**, **lastName** and **employeeID.**

The **init** accessor of the full property enables validation logic to be used whilst restricting the setting of the property to construction time.
The **get** accessor of the full property enables conversion logic to be used whilst reading the property value. The property value is visible outside of the class.

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);
```

QA

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.

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
```

QA

259

this can be used with or without parameters and any parameters in the current constructor are available to be passed to the called (chained) constructor.

This is used to avoid repeating / duplicating code in different overloaded constructors. It is typical for the constructors with the fewest parameters to call constructors with the most parameters.

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";
    }
}
```