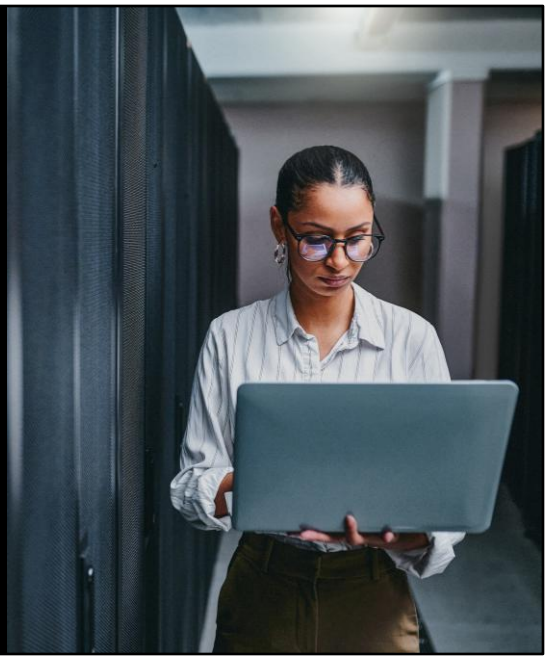
01 Course Introduction

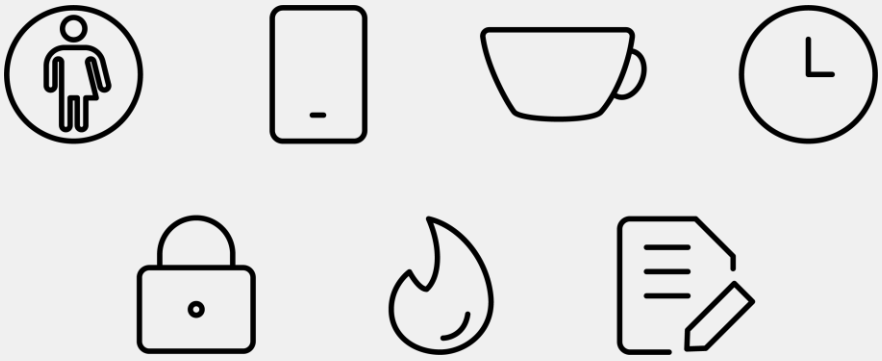
C# The Programming Language Part 2



QA

Learn. To Change.

Housekeeping

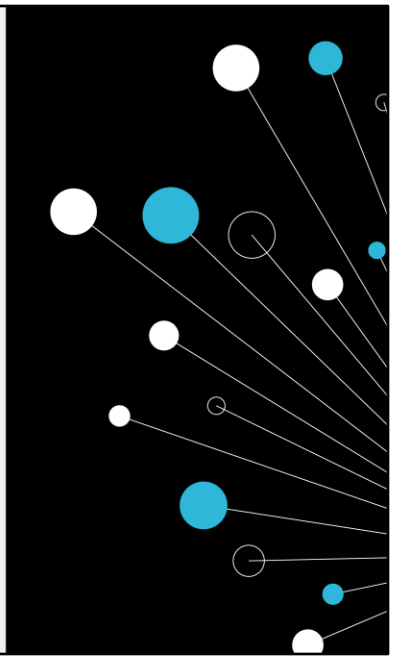


QA

Learning objectives

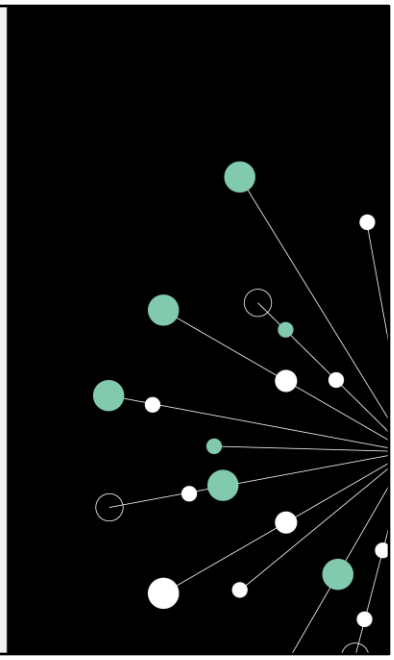
- Gain a solid understanding of the Principals of Object-Oriented Programming (OOP)
- Know how to use the Entity Framework to access a SQL server database
- Understand the use of delegates and lambdas to reduce code duplication
- Be able to create Language Integrated Queries (LINQ)
- Understand how to handle exceptions in C#

QA



Course Outline

- Object-Oriented Programming (OOP) Principles
- Inheritance and Abstract Classes
- Interfaces
- Delegates and Lambdas
- Database Access Using the Entity Framework
- LINQ
- Exception Handling



QA

Contents

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

QA

Pre-requisites

Essential skills:

- Understanding of C# and .NET to the level of the QA C# Fundamentals course

Beneficial skills

- Awareness of object-oriented principles
  - Objects have 'state' and 'behaviour'
- Working with a modern IDE: Visual Studio
- **Note**: Java and C# are very similar. This course is not intended for those who are already fluent in Java.

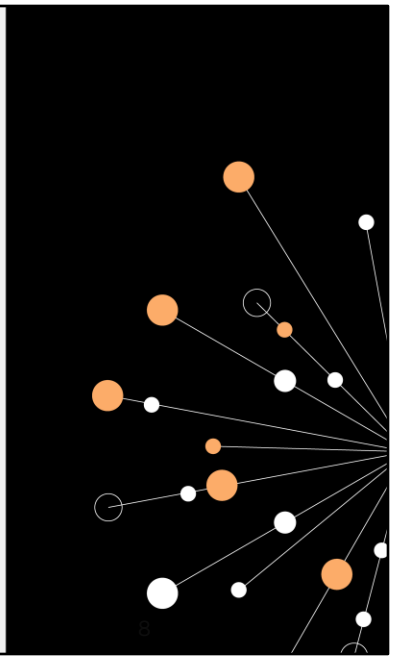
QA

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 C# and .NET to the level of the QA C# Fundamentals course. 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).

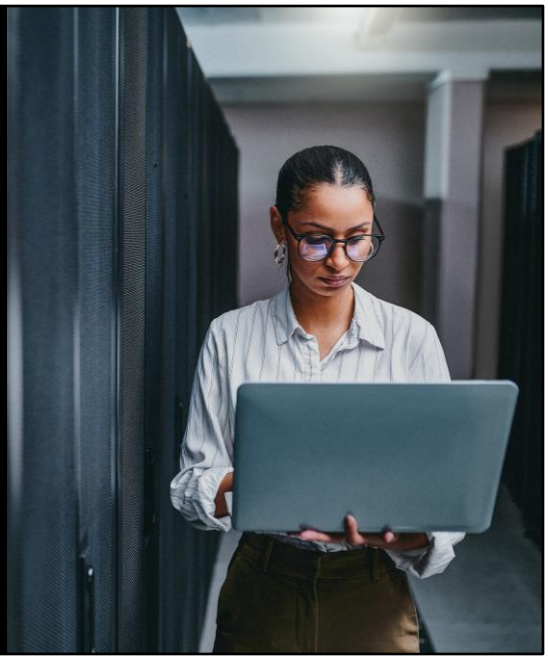
For this reason, your tutor can allocate you a virtual machine which has virtual machine has all the necessary software that you need for the course already installed on it.

02 Object Oriented Programming Principles

C# The Programming Language Part 2

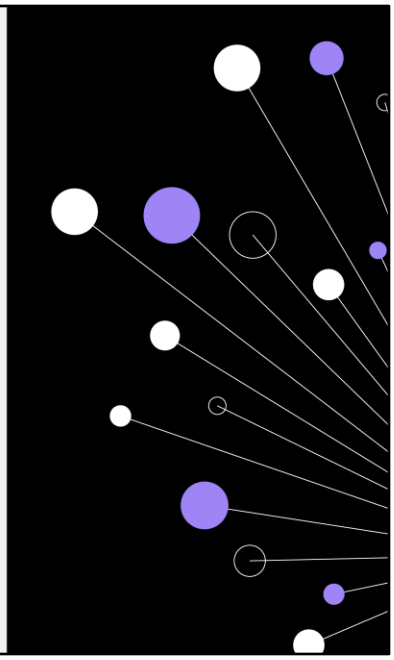
QA

Learn. To Change.



Learning objectives

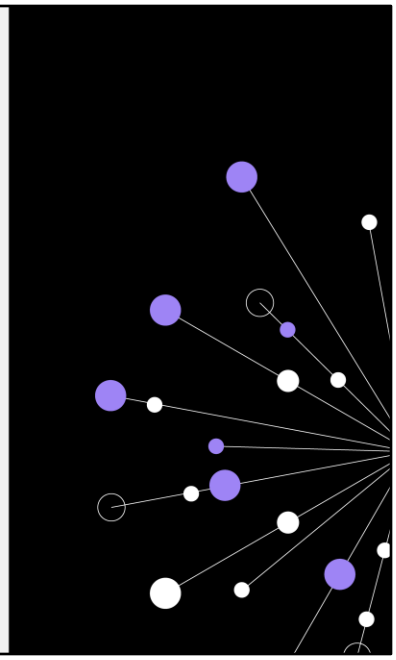
- Review OOP basics (covered in C# Fundamentals course)
- Understand the core principles of Object-Oriented Programming (Encapsulation, Inheritance, Polymorphism, Abstraction)
- Gain an overview of constructors and initialisers
- Know how to create the different types of Property and when to use them
- Know how to populate collections with class instances



QA

Contents

- Why Object Orientation?
- Four Concepts
- Classes and Instances
- Abstraction, Encapsulation, Inheritance, Polymorphism
- Understanding the concept of an ‘object’ reference
- 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

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

OO Concept - Inheritance

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

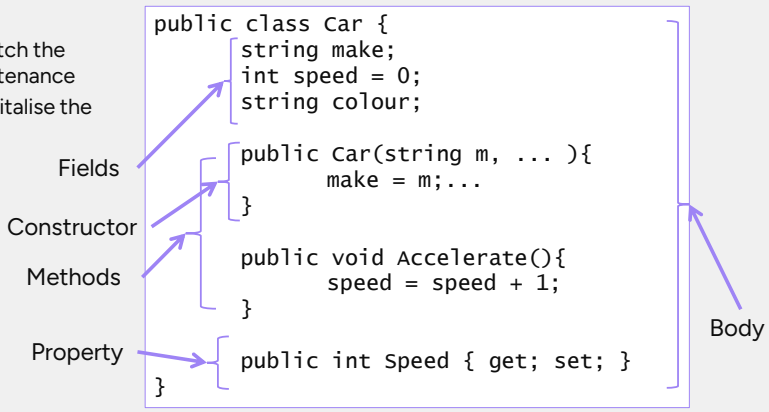
OO Concept - Polymorphism

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

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

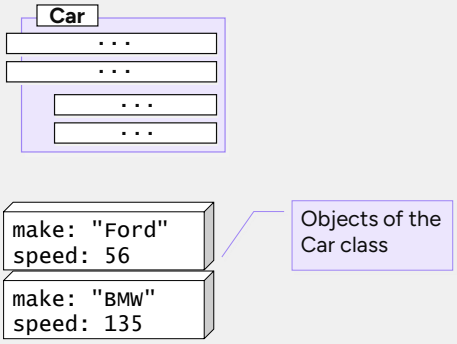
Classes and Objects

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.

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

22

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**.

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

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;
}
```

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
```

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

27

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.

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.

Lists – revisited

All List variables are reference variables

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

Assuming this class defined

```
Car[] cars1;

List<Car> cars2 = new List<Car>();

List<Car> cars3 = new List<Car> { new Car(),
                                  new Car(),
                                  new Car()};
```

'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

```
ProcessCarList(cars3);
```

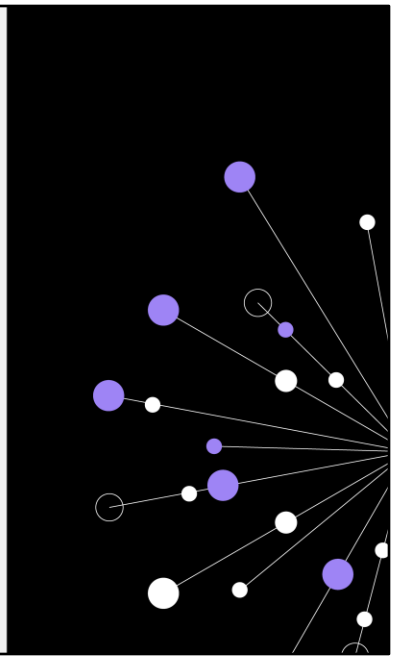
QA

The sample method ProcessCarList(Car[] cars) would typically be void, it doesn't need to return anything as it can see (and change) the contents of the passed in collection.
“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

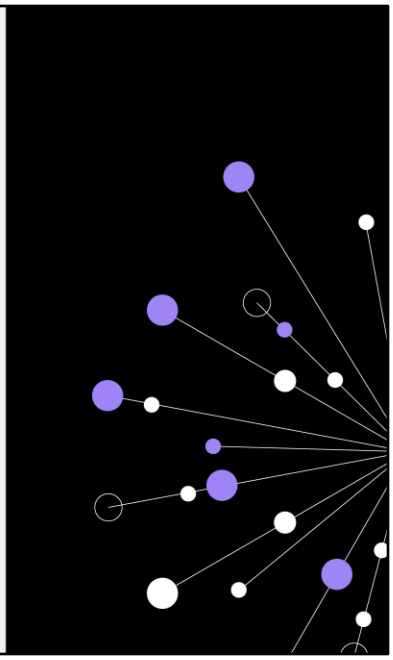
- Why Object Orientation?
- Four Concepts
- Classes and Instances
- Abstraction, Encapsulation, Inheritance, Polymorphism
- Understanding the concept of an ‘object’ reference
- Null Reference

QA



ACTIVITY

Exercise 02 Object Oriented Programming Principles



QA

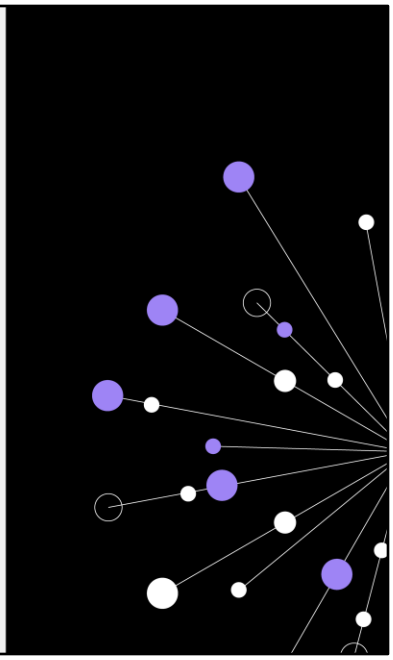
Appendix

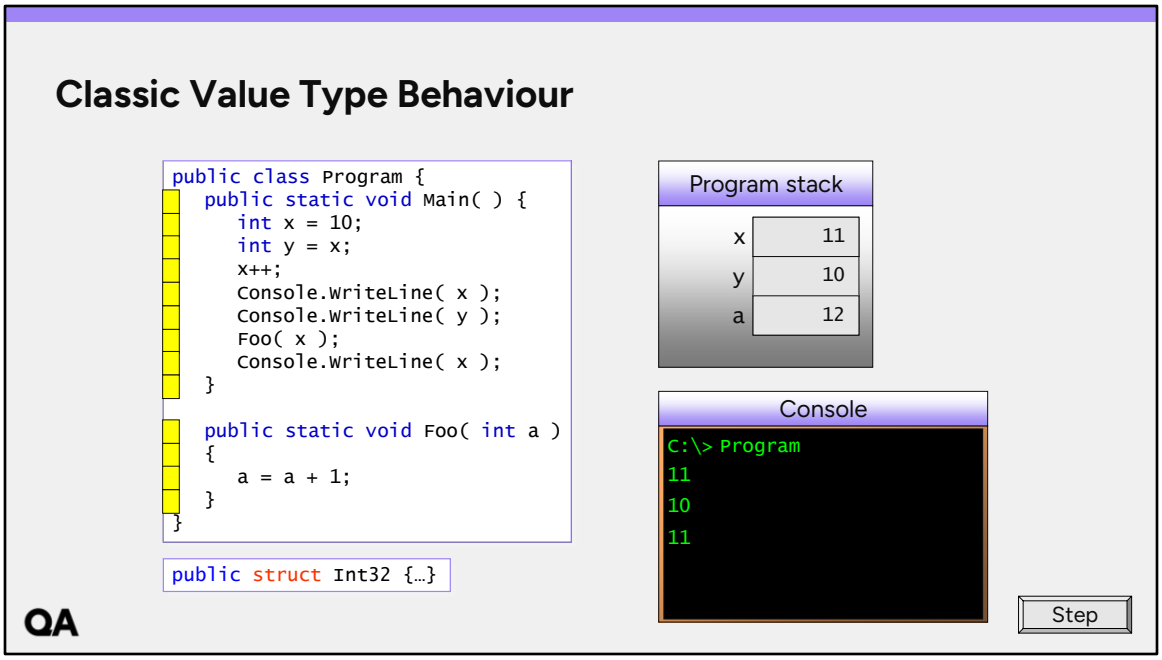
Passing Value and Reference Types as parameters

Parameter passing ("out" and "in")

Memory Management with the garbage collector

QA





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.

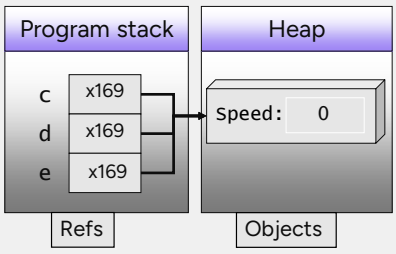
Reference Type Behaviour – different!

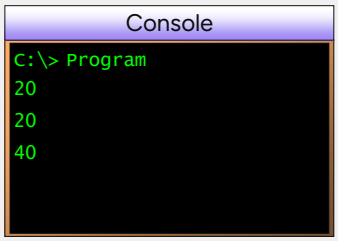
```
public class Program {
public static void Main( ) {
 Car c = new Car();
 c.Accelerate(10);
 Car d = c;
 d.Accelerate(10);
 Console.WriteLine(c.GetSpeed());
 Console.WriteLine(d.GetSpeed());
 Foo( c );
 Console.WriteLine(c.GetSpeed());
}
public static void Foo(Car e)
{
 e.Accelerate(20);
}
}
```

Copy of reference passed!

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

Functionality of Accelerate() & GetSpeed()





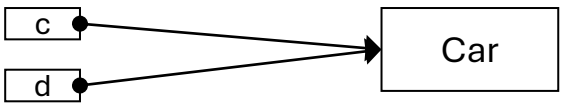
Step

QA

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).

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

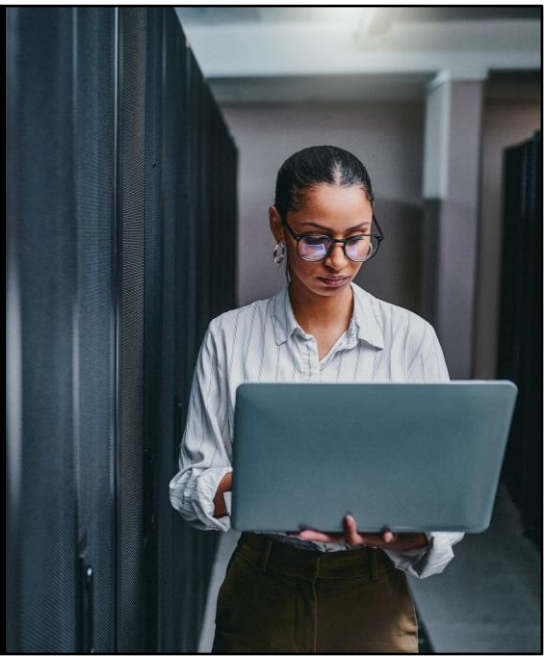
QA

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.

03 Inheritance and Abstract Classes

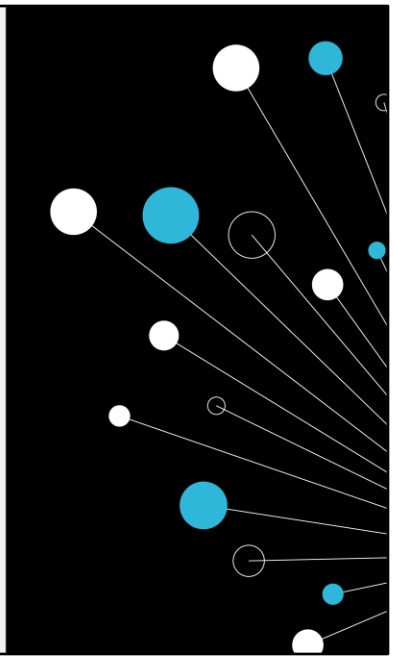
C# The Programming Language Part 2

QA
Learn. To Change.



Learning objectives

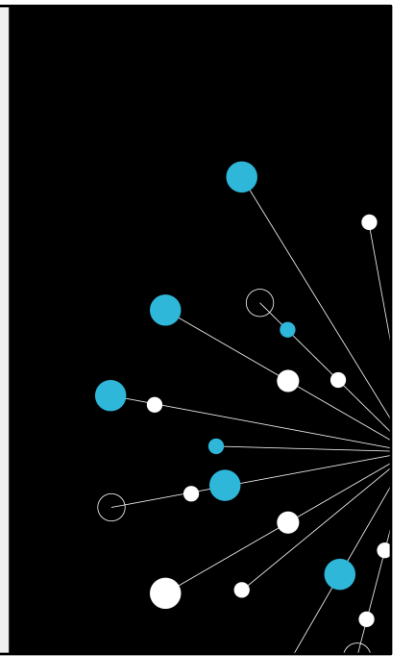
- Understand Inheritance and how to implement it in classes
- Know how marking members as virtual and overriding them in derived classes leads to Polymorphism
- Be able to Invoke base class functionality
- Understand what Abstract classes are and why they can be helpful
- Know when and why casting of types is necessary and to be able to use the is and as operators



QA

Contents

- Inheritance
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators



QA

Inheritance

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

QA

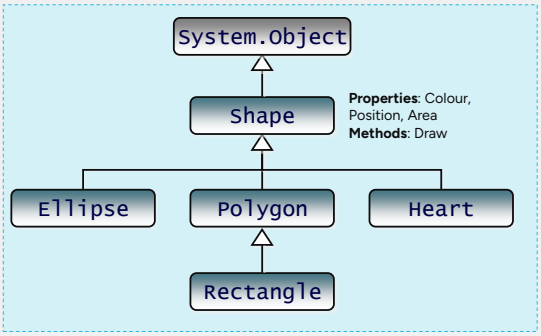
Examples of inheritance are ubiquitous in our model of the real world. Our definition of a cat inherits the features of our definition of a mammal, which in turn inherits the features of our definition of an animal. A chair and a table both inherit the features of furniture. A taxi is a kind of car.
Notice that derived classes take on features of the base class and can add or modify features, but they cannot remove features.

Note: A finalizer is used to perform any necessary clean-up when an object instance is being garbage collected.
Inheritance pertains to classes and not to structs.

Inheritance example: Graphics Application

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

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



QA

An example where inheritance might be a useful tool is a graphics application that needs to draw shapes. Such a program would allow us to create common shapes, such as ellipses, rectangles and triangles. Each of these is likely to be a class in the application's model. However, it quickly becomes apparent that these classes all share common behaviours and properties; shapes are filled with a colour, all need to be drawn, etc.
Good OO practice would have us factor the common elements into a base class, **Shape**, from which all of the other classes would be derived. These derived classes would then gain the benefit of re-use of the code from the base class. However, the derived types would also need to be able to extend and modify the base class functionality; for example, each separate type would need to be able to provide its own algorithm to calculate the area of the shape, and many would need to add specific fields and constructors to support their different data requirements.
One of the key things that we observe is that we can apply a test to see whether inheritance will work in our model: the "is a kind of" relationship test. A triangle is a kind of shape; an ellipse is a kind of shape; a circle is a kind of ellipse. This test confirms that we are introducing inheritance relationships that make logical sense.
Any class that does not explicitly extend another class implicitly extends the **System.Object** class. The Object class is the only class that does not have a base class and is the root class of all other classes.
Single inheritance means that each class can only have one direct *base* class: the direct base class of Circle is Ellipse; the direct base class of Ellipse is Shape.

Inheritance example: Graphics Application

Declare the base class: **Shape**

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

Define the derived class: **Polygon**

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

Add additional properties and methods as required

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


QA

Inheritance example: Graphics Application

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

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

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

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

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

QA

In a derived class, you only need to provide code for the things which are different to the base class.
Use a colon after the derived class name followed by the base class you wish to inherit from.
Note: if a base class is omitted, the compiler will implicitly inherit from the base class **System.Object**.

Derived Constructors

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

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

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

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

QA

In the example the Shape class only has one constructor which takes two parameters: position and colour.

Constructors are not inherited. But the rules they encapsulate are. Any class derived from Shape, *must have* some way of telling its base class what position and colour it should have.

The Ellipse constructor takes four parameters, of which the first two (position and colour) are passed up to the base class constructor. To invoke a constructor in the base class, use the *base* keyword. To invoke a constructor within the same class, use the *this* keyword.

It isn't possible to use both *this* and *base* in the same constructor; it's an either / or situation.

Polymorphism

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

QA

Polymorphism Scenario

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

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


QA

In our drawing program, it seems likely that a collection of all of the different shaped objects will be maintained in some kind of a drawing type.
This collection would hold references to Rectangle, Ellipse and Triangle objects, but the code above treats them all as if they were mere Shape objects when it iterates through the collection to draw all the shapes. This generalised approach to working with objects is made even more powerful because of the fact that we can access behaviours of objects polymorphically.
When you generalise a **Rectangle** as a **Shape**, polymorphism will call Rectangle-specific methods rather than general Shape methods, if they exist.

Polymorphism with virtual methods or properties

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

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

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

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

QA

A derived class inherits all of the instance methods and properties of its base class. However, it can also modify inherited behaviour by overriding it. This means that the derived class defines a method or property with exactly the same signature and return type as one in a base class (not necessarily its immediate base class).

In C#, a class must explicitly allow a method to be overridden through the use of the “virtual” keyword. Otherwise, an attempt to override will generate a compiler warning.

C# will call the runtime’s specialized member rather than the declared type’s generalized member. This is polymorphism: treating objects generally but having them act specifically.

Member access modifiers

Members (methods and properties) can be marked as:

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

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

QA

A “public” member: access is not restricted. Many classes are marked with the public keyword, as are methods and properties that represent the publicly accessible façade of a type.
A “private” member: access is limited to the containing type. This modifier is very commonly applied to fields and occasionally to some methods and properties.
A “protected” member: access is limited to the containing class or types derived from the containing class.
An “internal” member: access is limited to the current assembly. Certain classes, known as helper classes, and some methods and properties will be specified with internal access.
A “protected internal” member: access is limited to the current assembly or types derived from the containing class.
A “private protected” member is accessible by types deriving from the containing class, but only within its containing assembly.

Invoking base class functionality

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

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

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

QA

Common overriding scenarios are:

- Replace the virtual method's code completely
- Call the base class's implementation first, then add extra code
- Perform some of your own code, then call the base class's code

To do this, you use the **base** keyword and call the appropriate method or property. This will call the matching method in the base class, or in one of its base classes if the method is virtual and hasn't been overridden in the immediate base class.

You cannot call "*base.base*". The **base** keyword can only be used to refer to the immediate base class.

If you omit the **base** keyword, you are effectively referring to **this**, which could lead to a stack overflow as your method repeatedly calls itself.

Abstract Classes

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

QA

Abstract classes example – Part 1

An abstract class can contain abstract members:

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

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

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

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

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

QA

Good practice in OO design is to factor out as much common data and behaviour as possible into a shared base class. If this base class becomes so general or abstract that it is used only as a framework by derived classes and is never instantiated, that class is known as an abstract class.
For example, the author of the Shape class has no real idea of how a specific shape will be drawn, nor can they possibly know how to calculate the area of a specific shape. However, they can determine that all shapes can be drawn and have their area calculated. Therefore, they can add an abstract definition of this behaviour without actually providing any implementation at all.
Derived classes must then replace (via overriding) the abstract member with a concrete implementation.
An abstract class can contain anything a non-abstract class can contain, such as instance variables and instance methods. Abstract classes can also contain abstract members. Abstract members do not contain any implementation code.

Abstract classes example – Part 2

Use the override keyword to implement the member:

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

QA

Good practice in OO design is to factor out as much common data and behaviour as possible into a shared base class. If this base class becomes so general or abstract that it is used only as a framework by derived classes and is never instantiated, that class is known as an abstract class.
For example, the author of the Shape class has no real idea of how a specific shape will be drawn, nor can they possibly know how to calculate the area of a specific shape. However, they can determine that all shapes can be drawn and have their area calculated. Therefore, they can add an abstract definition of this behaviour without actually providing any implementation at all.
Derived classes must then replace (via overriding) the abstract member with a concrete implementation.
An abstract class can contain anything a non-abstract class can contain, such as instance variables and instance methods. Abstract classes can also contain abstract members. Abstract members do not contain any implementation code.

Abstract members

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

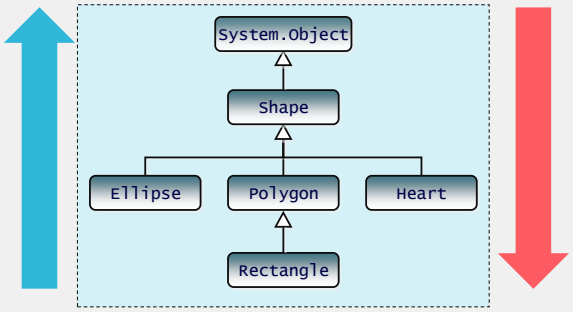
```
public abstract class Shape
{
    // concrete properties and methods
    0 references
    public Color Colour { get; set; }

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

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

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

QA

An abstract method is a method that cannot meaningfully be implemented by a class. Any class that contains one or more abstract methods is an abstract class and must be marked with the abstract keyword. A concrete derived class must implement all of the abstract methods of its base class. The declaration of an abstract method in an abstract base class forces all concrete descendants of that class to implement that method. Any derived class that does not implement all of the abstract methods of a base class is implicitly abstract and cannot be instantiated.
An abstract method has no method body; just a signature that includes the keyword abstract.
If ClassA is abstract and ClassB derives from ClassA and overrides all of ClassA's abstract members (making ClassB a concrete class), then any class that derives from ClassB does not need to provide its own implementation for ClassA's abstract methods. This is because ClassB has overridden all of the necessary members. Any further deriving classes can override these members if they require.

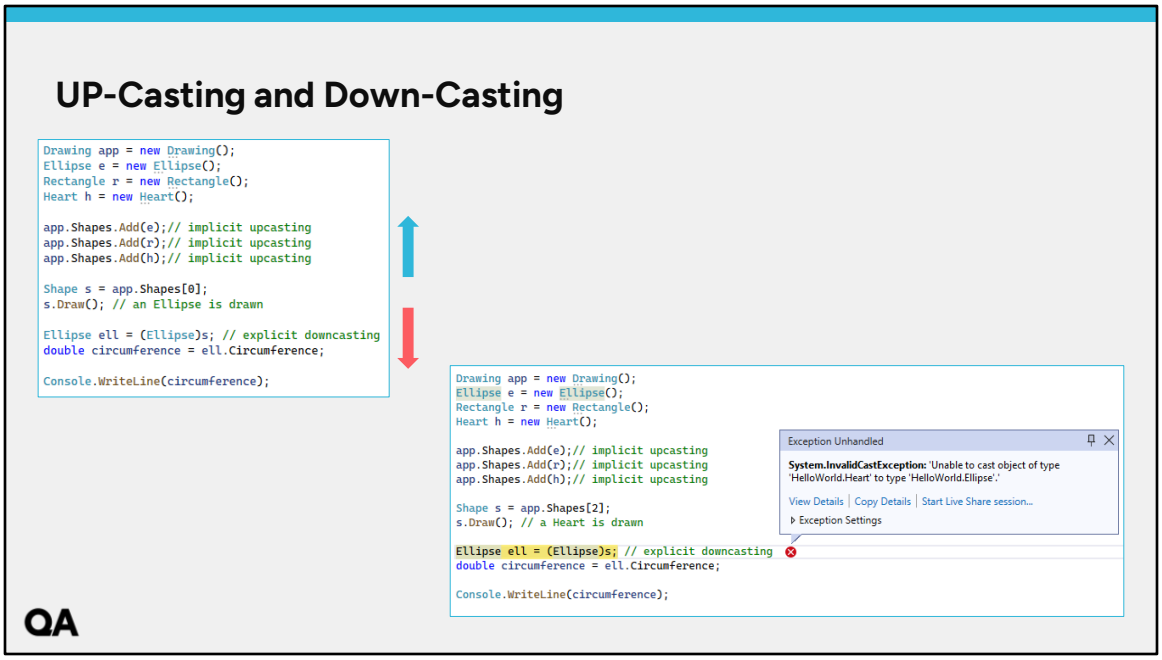
Casting derived and base classes

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



QA

Every Rectangle is a Polygon and every Polygon is a Shape. It is therefore safe to use a variable declared as a Shape and pass a Polygon or Rectangle instance. This is an up-cast and happens implicitly.
The reverse is not true. Not every Shape is a Polygon. A Shape may be an Ellipse or a Heart shape. You must therefore perform a down-cast explicitly. If the type is incompatible, an exception will be thrown.



You can only invoke methods and properties of the base type when working with a base type reference. For example, in the code above the compiler will check to make sure that the method Draw is defined in the base **Shape** class. But what happens when we want to call a method or property that is defined in the derived class, such as the Circumference property defined in Ellipse?
In this case, a down-cast must be performed explicitly. This is potentially unsafe. What happens if the Shape extracted from the collection is not an Ellipse? The cast to Ellipse would fail.
To summarise, it is always possible to use a base reference to refer to a derived object (and to implicitly convert a derived reference to a base reference, which is known as an “up cast”). Conversions down the hierarchy (down-casts) require explicit casts which may throw an exception. Extra checking can be performed to prevent the exception being thrown.

Safe Downcasting

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

- is
- as

QA

The ‘is’ operator

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

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

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

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

QA

The **is** operator checks if the run-time type of an expression result is compatible with a given type **T**. It returns true when an expression is not null and the run-time type matches the type **T** or is derived from type **T**. If an implicit reference conversion exists or a boxing or unboxing conversion exists, it also returns true.

From C# 7.0 onwards, you can also use the **is** operator to test an expression against a pattern. The example uses the *declaration pattern* whereby you declare a local variable to store the cast expression result if the type match succeeds.

From C# 9.0 onwards, you can use the **not**, **and**, and **or** pattern combinators to create *logical patterns*.

Best practice recommendation: use the pattern matching syntax whenever possible because it combines the test and the assignment in a single statement.

The 'as' Operator

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

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

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

QA

The **as** operator prevents an exception being thrown. It returns the cast expression result or null if the conversion is not possible.
The **!=** check uses any overloaded equality operators which may have been customized to return unexpected results when comparing to null.
The **is not null** check uses reference equality and will always return the expected outcome of a null test.

The Object Class

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

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

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

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

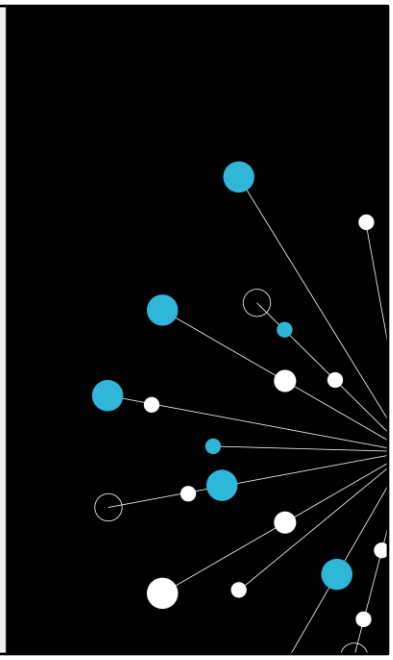
See Appendix for examples

QA

Summary

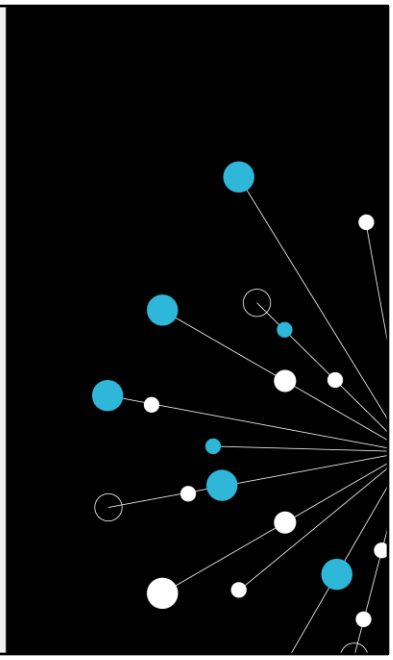
- Inheritance
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators

QA



Activity

Exercise03 Inheritance and Abstract Classes



QA

Overriding Object methods example

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

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

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

QA

The **Book** class implicitly inherits from **System.Object**

The *virtual* **Equals**, **GetHashCode** and **ToString** methods inherited from **Object** have all been *overridden*.

Overriding Object methods example

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

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

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

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

QA

The overridden **Equals** method compares the Author and Title properties. If they match, it deems the two book objects to be equal.
The **==** operator compares the object references. Book **b1** and Book **b2** are different objects in memory and therefore this evaluates as *False*.
Book **b1** and Book **b3** point to the same object in memory and therefore this evaluates as *True*.

Sealed classes and members

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

QA

Sealed Classes

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

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

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

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

No classes are allowed to inherit from a sealed class:

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

class SealedMembers.ClassB

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

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

QA

Note: You cannot seal an abstract class.

Sealed methods

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

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

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

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

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

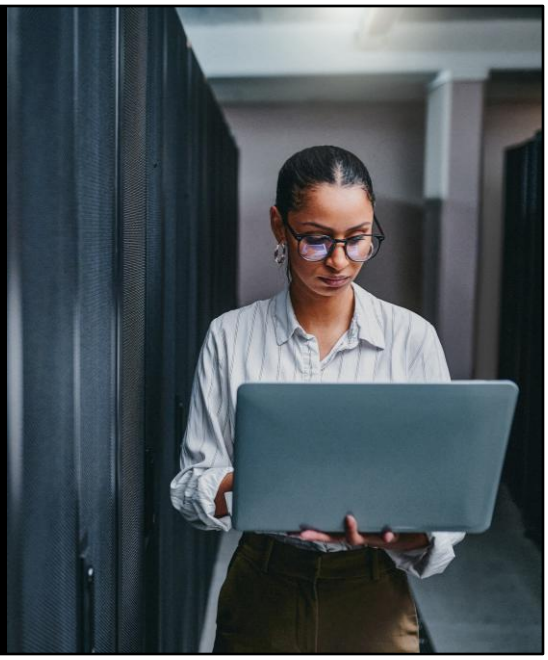
QA

Once a virtual member is sealed, it cannot be further overridden. You must use the sealed modifier with the override modifier for members.

04 Interfaces

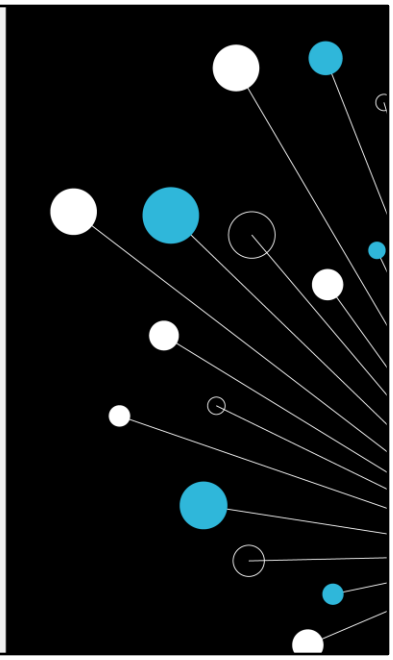
C# The Programming Language Part 2

QA
Learn. To Change.



Learning objectives

- Understand what interfaces are and why they are useful
- Known how to implement interfaces
- Be aware of how interfaces are polymorphistic

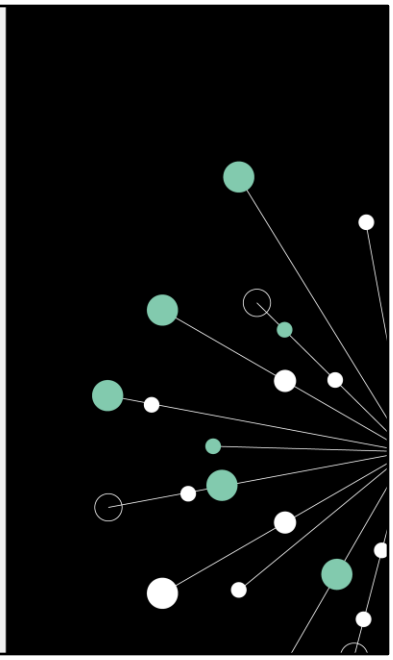


QA

Contents

- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces

QA



Interfaces

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

QA

Classes in different inheritance hierarchies can implement the same interface to create a common group of functionalities across different classes.
Interfaces are implemented. They are not instantiated.
Interfaces cannot have any instance fields.

Defining an interface

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

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


QA

Implementing an interface

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

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

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

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

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

QA

Polymorphism

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

QA

Polymorphism example

```
Graphics canvas = new();

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

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

QA

Multiple interfaces

A class or struct can implement multiple interfaces

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

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

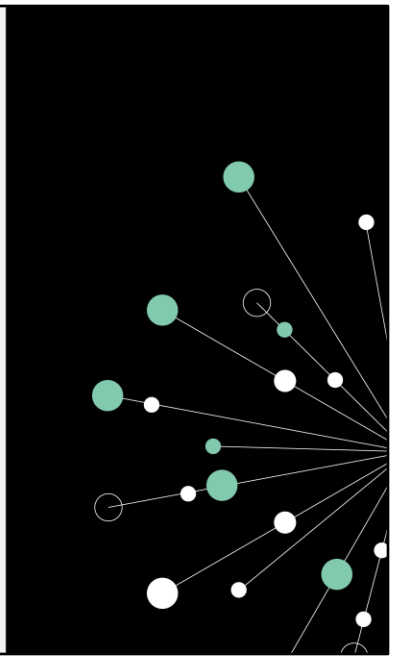
```
public class Rectangle : Shape, IDrawable, IComparable<Rectangle>
{
    3 references
    public int Width { get; set; }
    1 reference
    public int Height { get; set; }
    1 reference
    public override double Area => Width * Height;

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

QA

Summary

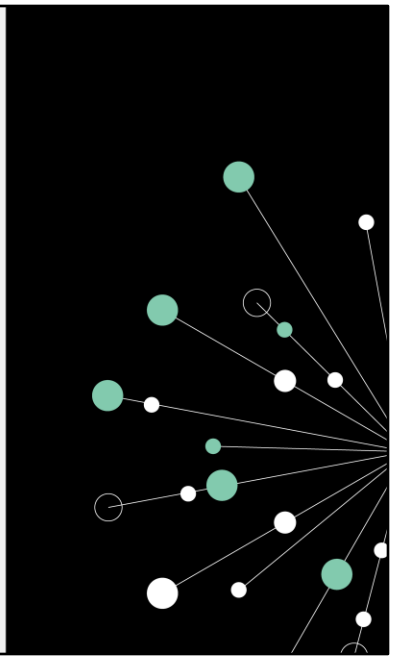
- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces

QA



Activity

Exercise 04 Interfaces



QA

Default implementation

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

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

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

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

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

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

QA

The **GetTaxAmount** method does not have to be implemented in the implementing class (Employee) because the interface provides a default implementation.

This feature enables existing interfaces to be extended without breaking existing implementing classes and structs.

To access the default implementation, the declared type must be the **interface** type not the implementing class or struct type.

Interfaces can inherit from interfaces

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

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

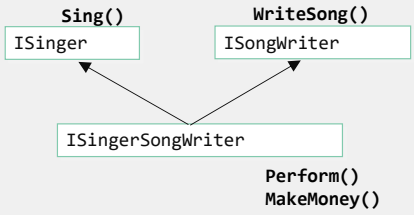
```
public interface IFullySteerable : ISteerable
{
    1 reference
    void GoUp();
    1 reference
    void GoDown();
}
```

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

QA

Multiple Interface inheritance

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



QA

Multiple interface member collisions

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

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

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

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

QA

Explicit Interface Implementation

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

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

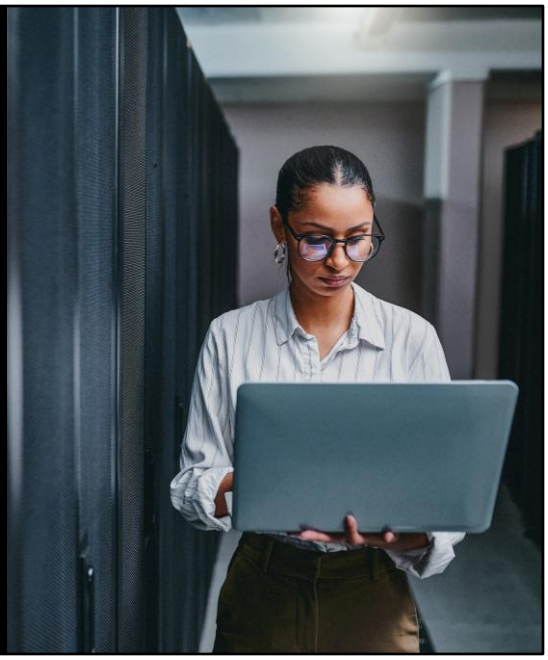
```
CowboyShape cs = new CowboyShape();
cs.Draw(canvas);// Drawing a cowboy shape

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

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

QA

You do not include an access modifier on an explicitly implemented member. The member is implicitly public through the interface reference.

06 Database Access Using the Entity Framework

C# The Programming Language Part 2

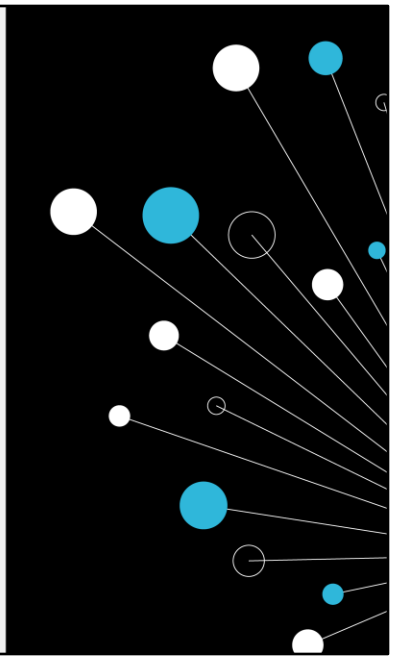


QA

Learn. To Change.

Learning objectives

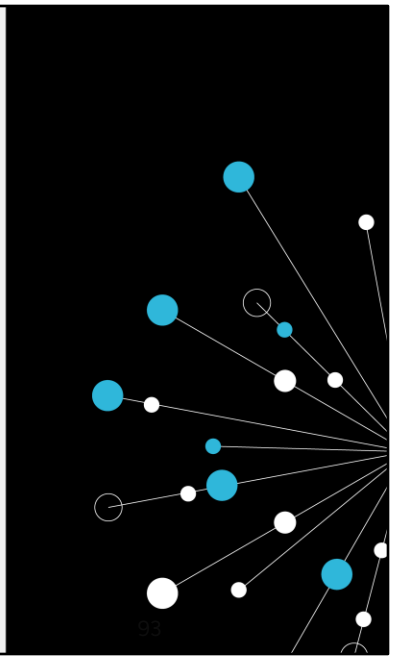
- To give an introduction to Entity Framework
- Understand the main features of Entity Framework
- Be able to use a database first approach to access and update a database using the Entity Framework and its tools



QA

Contents

- Understanding ORMS
- Code first vs Database First
- Models
- Connection strings
- Context
- Querying and Updating
- LINQ (a Sneek Preview)
- Using appsettings.json
- Configuration
- IEnumerable vs IQueryable



QA

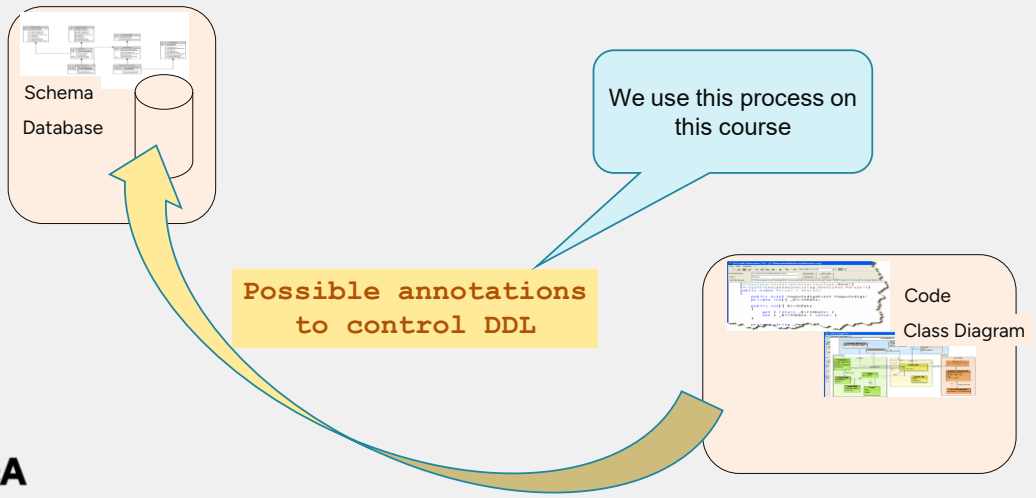
Why have an ORM at all?

Most applications use OO code and store in relational data hence there is a mis-match. To do-it-yourself you need to:

- Write the object model
- Write the plumbing code - synchronising, lazy loading, persistence, track changes, concurrency, validation, data type conversions, relationships and associations, vendor independence, make it a LINQ-provider.
- Typically, the DIY way consumes ~35% of project code & budget. With an ORM this typically drops to <10%.

QA

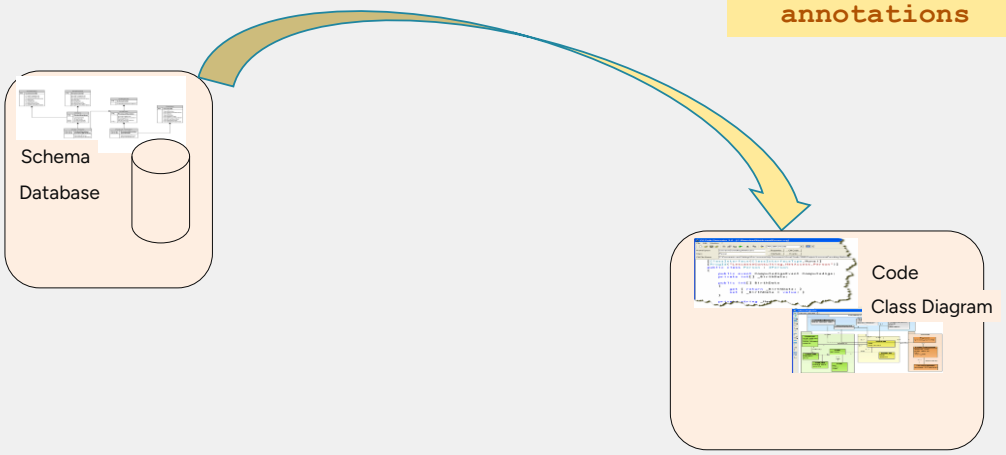
Code First



QA

Code First From Database

Possible 'Buddy' classes to add annotations



QA

To get started

EF favours convention, but almost everything can be modified by configuration

NuGet:

Microsoft.EntityFramework.Core.SqlServer

The minimum you need are:

- Properties NOT Fields!!
- An Id Property
- A Context
- What provider
- What database name, permissions etc
- What database tables will be exposed to the code

Code samples for getting started

Member class

```
//Properties and Ids
public class Zoo {
    public int ZooId { get; set; }
    public string Name { get; set; }
}
```

Context class

```
//Context
public class ZooContext : DbContext {
    public DbSet<Zoo> Zoos { get; set; }
    public DbSet<Animal> Animals { get; set; }
}
```

Connection Strings

- Used to specify connection details such as (for SQL Server):
  - Server (Data Source)
  - Database (Initial Catalog)
  - Credentials (or trusted connection setting)
- Managed in Context class's OnConfiguring method
- Note, it is not recommended to put the connection string directly in the code as shown below

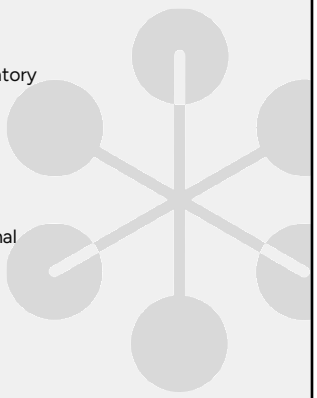
```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
ng  To protect potentially sensitive information in your connection string, you should move it out of source code. You can a
        optionsBuilder.UseSqlServer("Data Source=(local);Initial Catalog=Movies;trusted_connection=true;Encrypt=False");
    }
}
```

Context

```
public class ApplicationDbContext : IdentityDbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);
    }
}
```

Mandatory

Optional



QA

Using the Context Object to Query and Update Data

A regular 'Select'

```
using (var ctx = new NorthwindEntities()) {
    var customers = ctx.Customers.Where(c => c.Country == "Germany");
    foreach(var c in customers)
        Console.WriteLine(c.ContactName + ", " + c.City);
}
```

Change property and save

```
using (var context = new NorthwindEntities()) {
    var customer = context.Customers.First(c => c.CustomerID == "ALFKI");
    customer.City = "Berlin";
    context.SaveChanges();
}
```

QA

LINQ (a Sneek Preview)

Lambda-Style

```
using (var ctx = new NorthwindEntities()) {
    var customers = ctx.Customers.Where(c => c.Country == "Germany");
    foreach(var c in customers)
        Console.WriteLine(c.ContactName + ", " + c.City);
}
```

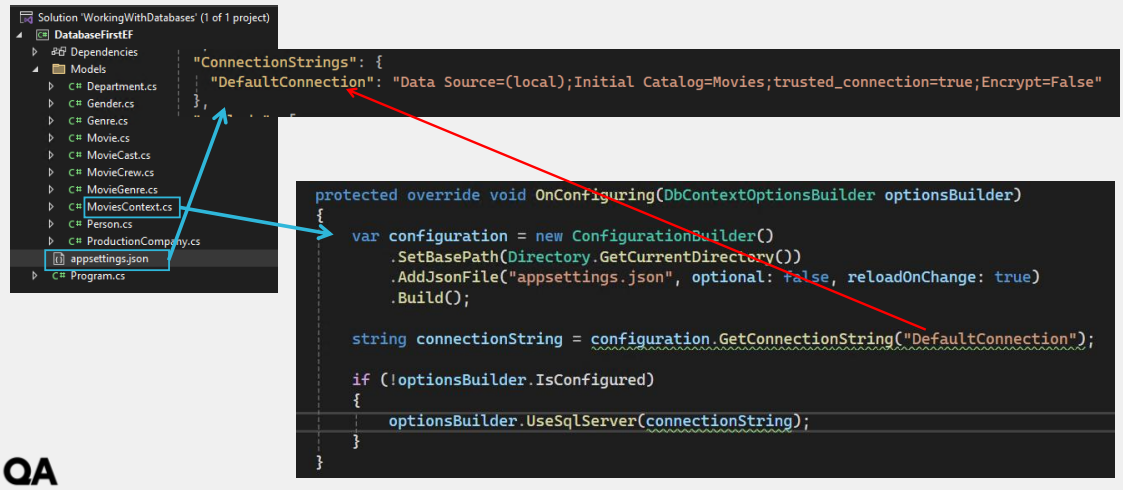
Query Expression Style

```
var customers = from c in ctx.Customers
                where c.Country == "Germany"
                select c;
foreach(var c in customers)
    Console.WriteLine(c.ContactName + ", " + c.City);
```

QA

LINQ is covered in much more detail in the next session

Configuration using appsettings



QA

The name of the database connection string can be passed into the DbContext constructor.
Note that newlines are not permitted in the connection string – this is done only so they can fit onto the page

Configuration

Using Attributes

```
[Required(ErrorMessage="Choose a forum for this thread")]
public int ForumID { get; set; }
[Column("OwnerID")]
public string UserID { get; set; }
```

Using the Fluent API

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    1 reference
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Forum>()
                    .HasOptional(a => a.Threads)
                    .WithOptionalDependent()
                    .WillCascadeOnDelete(true);

        base.OnModelCreating(modelBuilder);
    }
```

QA

When working with Code First, you define your model by defining your domain CLR classes. By default, the Entity Framework uses the Code First conventions to map your classes to the database schema. If you use the Code First naming conventions, in most cases you can rely on Code First to set up relationships between your tables based on the foreign keys and navigation properties that you define on the classes. If you do not follow the conventions when defining your classes, or if you want to change the way the conventions work, you can use the fluent API or data annotations to configure your classes so Code First can map the relationships between your tables.

IEnumerable / IQueryable

IEnumerable allows you to iterate collections

IQueryable does the iteration but also allows you to pass *commands* (rather) than *data* across your application

Beware of multiple unintended database accesses

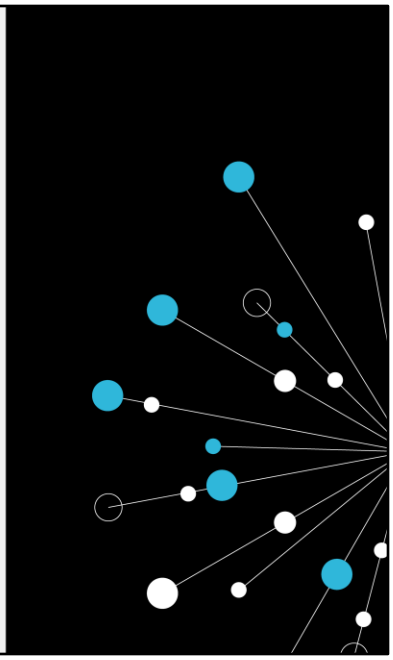
The general recommendation is to pass IList<> (i.e. an evaluated IEnumerable) as your interface

```
public interface IEnumerable
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IQueryable : IEnumerable
{
    string Provider { get;}
}
```

Summary

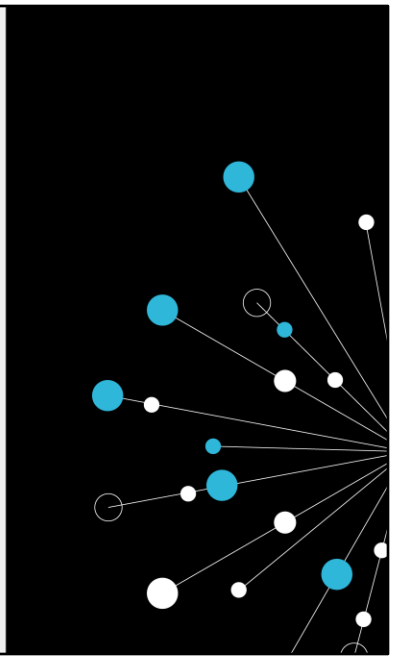
- Understanding ORMS
- Code first vs Database First
- Models
- Connection strings
- Context
- Querying and Updating
- LINQ (a Sneek Preview)
- Using appsettings.json
- Configuration
- IEnumerable vs IQueryable

QA



Activity

Exercise 09 Database Access using the Entity Framework

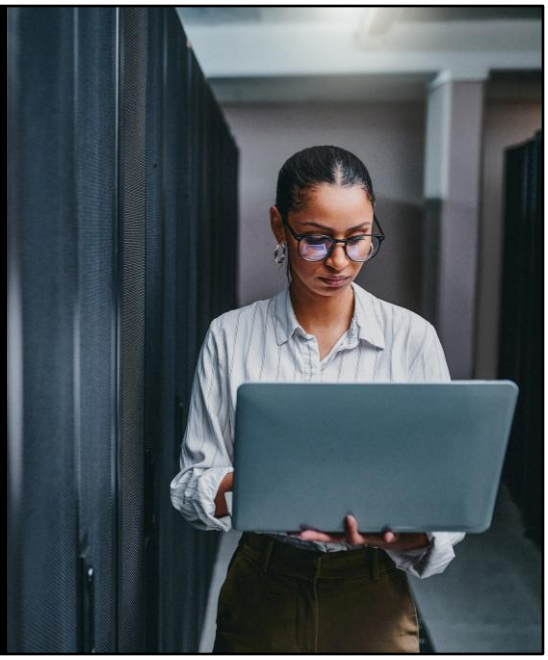


QA

05 Delegates and Lambdas

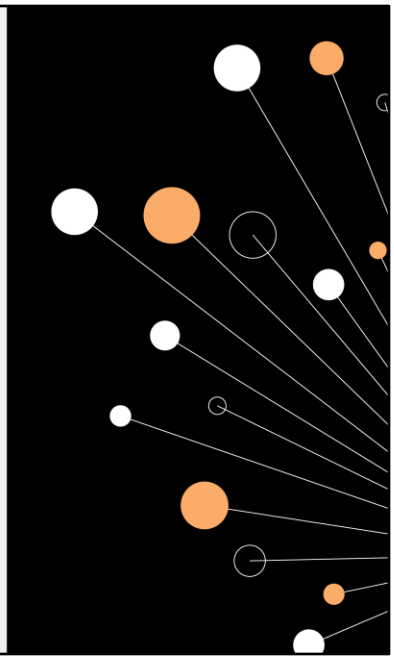
C# The Programming Language Part 2

QA
Learn. To Change.



Learning objectives

- Know what delegates are and understand the benefits they can bring to coding
- Know how to use the preexisting Func, Action and Predicate delegates
- Know how to create your own delegates

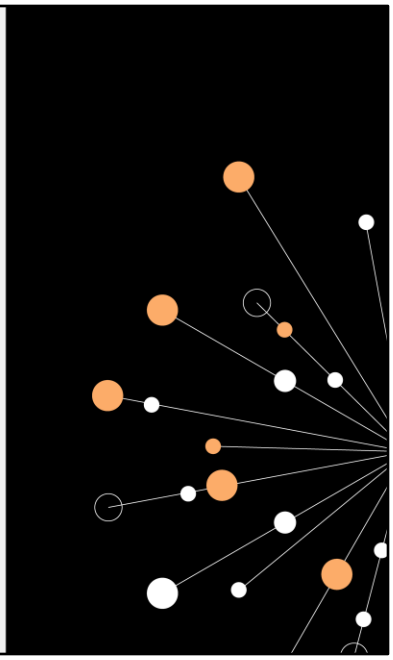


QA

Contents

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

QA



Delegates

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

Useful built-in delegate types are:

- Func
- Action

QA

The Func delegate

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

The last parameter is the return type.

There are many overloads:

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

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

QA

The Action delegate

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

There are many overloads:

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

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

QA

Func example

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

It is an instance method (not static).

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

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

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

```
DelegateExamples examples = new();

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

QA

Action Example

Delegates can encapsulate static or instance methods:

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

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

```
DelegateExamples examples = new();

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

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

QA

Example Part 1

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

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

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

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

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

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

QA

Example Part 2

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

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

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

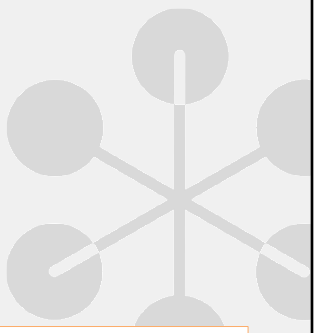
These two methods match the **Func** delegate signature and contain the conditional tests.

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

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

QA

Lambdas

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



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

QA

Func example as a lambda

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

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

```
DelegateExamples examples = new();

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

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

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

QA

Action Example as a lambda

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

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

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

```
DelegateExamples examples = new();

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

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

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

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

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

QA

Example Part 3

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

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

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

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

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

QA

The Predicate delegate

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

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

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

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


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

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

QA

You may come across a built-in delegate called **Predicate**.
Func<T, bool> is the equivalent of the Predicate delegate.

For example:
Func<int, bool> is equivalent to Predicate<int>.
Func<string, bool> is equivalent to Predicate<string>.
Because Func can do everything Predicate can do it is recommended that Func should be used instead of Predicate.

Delegate/ lambda examples

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

// FindAll uses a Predicat
List<Book> startsWithW = b
```

```
Book? List<Book>.Find(Predicate<Book> match)
Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.

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

Exceptions:
ArgumentNullException
```

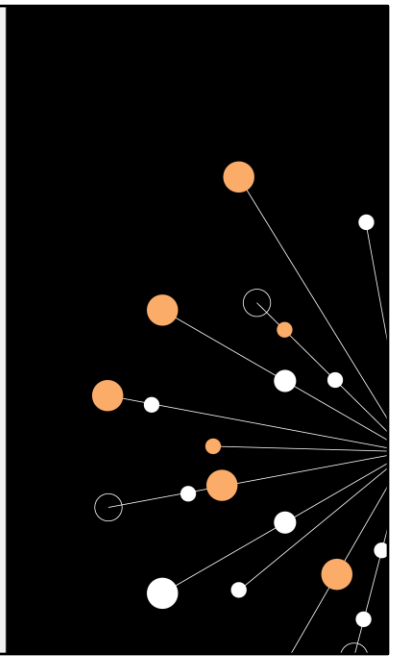
```
// Arrays and Lists have many methods that use a Predicate delegate
// Find uses a Predicate delegate
Book? recentBook = books.Find(book => book.YearPublished >= 2015);
if (recentBook != null)
{
    Console.WriteLine(recentBook.Title);
}

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

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

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

QA

Summary

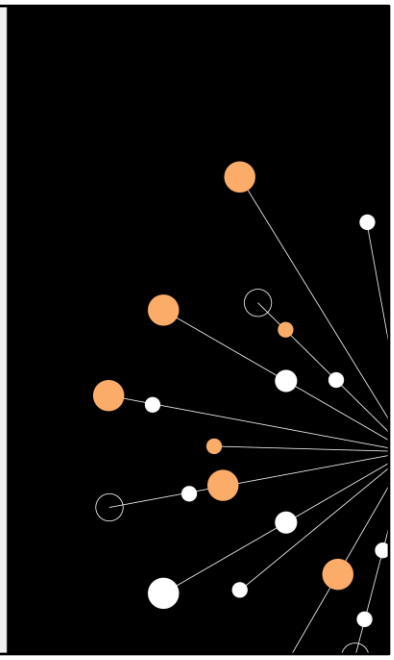
- Delegates
- The Func delegate
- The Action delegate
- Lambda expressions
- The Predicate delegate
- Delegate and Lambda examples

QA



Activity

Exercise 05 Delegates and Lambdas

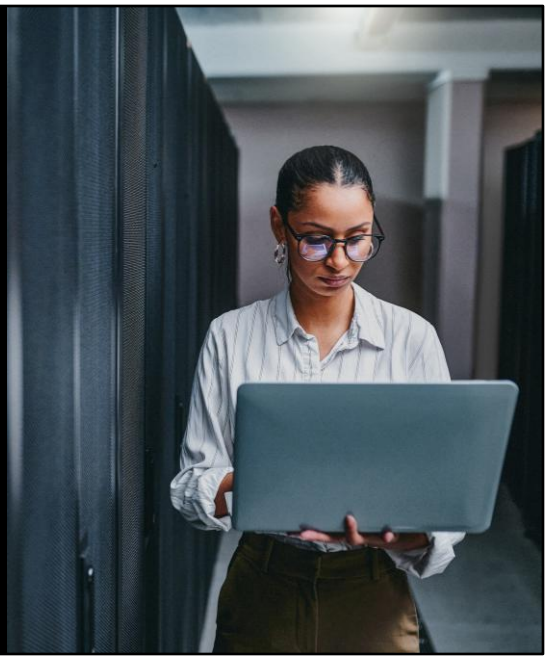


QA

07 LINQ

C# The Programming Language Part 2

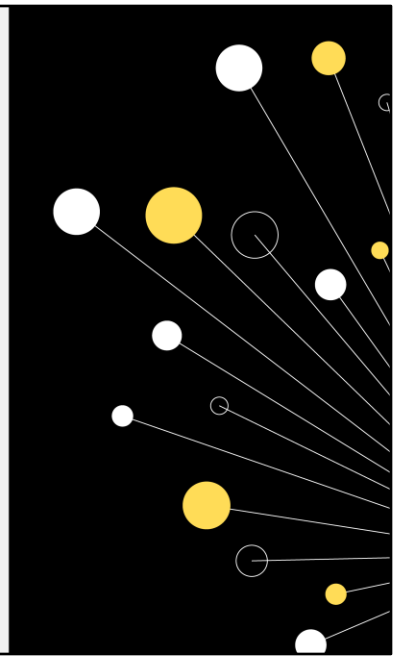
QA
Learn. To Change.



Learning objectives

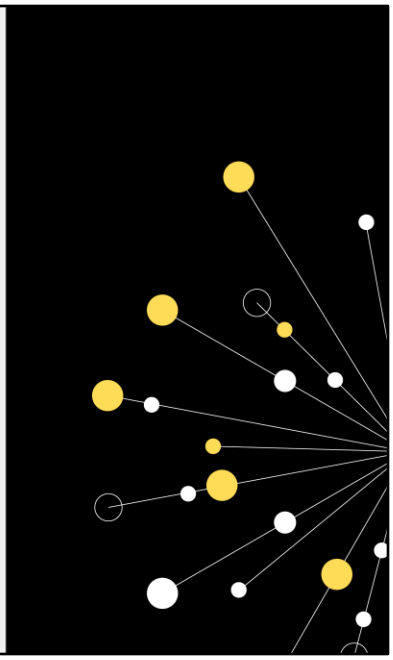
- Understand What LINQ is, why it's useful and how to use it
- Know how to work using the query method or query expression syntax
- Be able to perform joins and aggregations
- Be able to code LINQ using best practice

QA



Contents

QA



LINQ

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

QA

Scenario: Non-LINQ And LINQ equivalent

Compare the two code snippets.

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

The right-hand snippet uses LINQ.

```
// Non-LINQ Example

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

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

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

```
// LINQ

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

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

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

QA

The two code snippets show a non-LINQ approach and a LINQ approach to the same scenario: finding scores that are above 80.
The LINQ snippet uses a query expression. The non-LINQ example requires a collection to store the results. The LINQ equivalent does not require the results to be stored. The LINQ query is not executed until the query variable is iterated over in the foreach loop.

LINQ Syntax

Query expression syntax:

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

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

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

Query method syntax:

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

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

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

QA

The LINQ query expression syntax has been designed to be similar to Structured Query Language (SQL).

LINQ Syntax Example

Query expression syntax:

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

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

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

Query method syntax:

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

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

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

QA

Note the use of the **var** keyword. This provides flexibility in enabling you to project different types including anonymous types from your queries.
Some LINQ queries project (return) an **IEnumerable** type, others return an **IQueryable** type. It is also possible to return a new anonymous type by selecting only a subset of attributes of an object to be returned.
Note the use of LINQ's **OrderBy** clause which raises the question does its use negate the need for classes to implement **IComparable**? The answer is: Not entirely. While LINQ's **OrderBy** clause provides a flexible way to sort collections by specifying key selectors and custom comparers, implementing **IComparable<T>** still has its place in C#.
OrderBy is useful when sorting on the fly: If sorting is needed based on different properties or in multiple ways throughout the application, **OrderBy** allows specifying key selectors dynamically without modifying the class..
IComparable<T> is Useful for "Natural Ordering": If a class has a clear, default way it should be compared (e.g., DateTime, string, or int), implementing **IComparable<T>** allows objects to be compared consistently across the application. Use of **IComparable** can also improve performance when sorting large collections: **List<T>.Sort()** (which uses **IComparable<T>**) is generally faster than **OrderBy** because it sorts in-place rather than creating a new sequence.

LINQ Projections

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

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

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

Use the **var** keyword:

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

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

QA

Deferred Execution

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

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

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

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

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


names[0] = "Susie";

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


QA

Aggregations

- Aggregations execute without an explicit foreach statement
- The **baseQuery** uses lambda expressions to specify which property of Customer is to be aggregated
- **Qry2** projects the Balance property which is a decimal, so no lambdas are required

```
var baseQuery = from c in customers
                where c.City == "London"
                orderby c.Balance
                select c;// Customer

decimal avg = baseQuery.Average(c => c.Balance);
decimal max = baseQuery.Max(c => c.Balance);
decimal total = baseQuery.Sum(c => c.Balance);


IEnumerable<decimal> qry2 = from c in customers
                            where c.City == "London"
                            select c.Balance;// Decimal
avg = qry2.Average();
max = qry2.Max();
total = qry2.Sum();
```


QA

Forcing immediate execution: Aggregate functions

- Queries that perform aggregation functions over a range of source elements must first iterate over those elements, therefore they execute **without** an explicit foreach statement
- These types of queries return a single value not an IEnumerable collection

```
int[] scores = { 97, 92, 81, 60, 40, 54, 80, 75 };

var failingScores =
    from score in scores
    where score < 80
    select score;

int countFailingScores = failingScores.Count();
Console.WriteLine("Number of failing scores is " + countFailingScores);

double avgFailingScores = failingScores.Average();
Console.WriteLine("Average of failing scores is " + avgFailingScores);
```

QA

Forcing immediate execution: ToList or ToArray

To force execution of any LINQ query and cache its results, you can call the **ToList** or **ToArray** methods:

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

var query = (from s in names
             where s.Length == 5
             select s).ToList();//force execution

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

names[0] = "Susie";

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby
```


QA

137

For applications that want to cache the results of query evaluation, two methods, ToList and ToArray, are provided that force the immediate evaluation of the query and return either a List<T> or an Array containing the results of the query evaluation.
Any changes to the underlying original collection would not be reflected when iterating over the List/Array now being used.
Both ToArray and ToList force immediate query evaluation. The same is true for any LINQ operations that return singleton values (for example: First, ElementAt, Sum, Average, All, Any).
Note that the method signature for ToList is:

List<T> ToList<T>(IEnumerable<T> source).

You don't need to specify ToList<string>() in the above code because the compiler knows you are converting an IEnumerable<string> into a List.

Joins

The **join** clause is used to match elements in one collection with elements in another collection.

Find persons who have names that match the 'names' list:

```
string[] firstNames = { "Vinita", "Pete" };

var people = new List<Person> {
    new Person("Pete", 20), new Person("Rafael", 50),
    new Person("Vinita", 32), new Person("Pete", 37),
    new Person("Tom", 40)
};

var matched = from n in firstNames
              join p in people
              on n equals p.Name
              select new { AgeOfPerson = p.Age, FirstName = n };

foreach(var p in matched)
{
    Console.WriteLine($"{p.FirstName} {p.AgeOfPerson}");
}
```

```
Vinita, 32
Pete, 20
Pete, 37
```

QA

138

We want to find elements of 'people' who are in the list 'names'.

The code on the slide could be simplified because the end result is simply a list of Person elements whose Name is in the list of names.
As all the information in the string array is effectively held in the Person array (because people have names) there is no need for the 'select' statement to select 'n' or any portion of it. Often when a join takes place the resulting select is likely to create instances of an anonymous class containing information from both 'n' and 'p' (i.e. both halves of the join).

```
IEnumerable<Person> matched = from n in firstNames
              join p in people
              on n equals p.Name
              select p;

foreach(Person p in matched)
{
    Console.WriteLine($"{p.Name} {p.Age}");
}
```

Group Join

The **group join** method is useful for producing hierarchical data structures.

It pairs each element from the first collection with a set of correlated elements from the second collection

The query expression **join … into** translates to an invocation of **GroupJoin**

```
var groupJoined = from n in names
                  join p in people
                  on n equals p.Name into matchingNames
                  select new { Name = n, Persons = matchingNames };

foreach (var group in groupJoined)
{
    Console.WriteLine(group.Name);
    foreach (Person p in group.Persons)
    {
        Console.WriteLine("..." + p.Age);
    }
}
```

```
Vinita
...32
Pete
...20
...37
```


QA

Let clause

- The **let** clause enables you to store the result of a sub-expression in order to use it in subsequent clauses

```
// Specify the data source
string[] strings =
{
    "Been there, done that.",
    "All Greek to me.",
    "A piece of cake."
};

// Split the sentence into an array of words
// and select those whose first letter is a vowel.
var query =
    from sentence in strings
    let words = sentence.Split(' ')
    from word in words
    let w = word.ToLower()
    where w[0] == 'a' || w[0] == 'e'
        || w[0] == 'i' || w[0] == 'o'
        || w[0] == 'u'
    select word;

// Execute the query.
foreach (var v in query)
{
    Console.WriteLine("\"{0}\" starts with a vowel", v);
}

/* Output:
"All" starts with a vowel
"A" starts with a vowel
"of" starts with a vowel
*/
```

QA

- Once initialized with a value, the range variable created by **let** cannot be used to store another value
- The range variable can hold a queryable type that can be queried

OfType

The **OfType** method filters the elements of an IEnumerable based on a specified type:

```
public class Mammal
{
    0 references
    public string Name { get; set; }
}
```

```
public class Dog : Mammal
{
}
```

```
public class Cat : Mammal
{
}
```

```
List<Mammal> mammals = new() {
    new Dog() {Name="Rover"},
    new Cat() {Name="Tiddles"},
    new Dog() {Name="Snowy"}
};
var dogs = mammals.OfType<Dog>();

foreach (var dog in dogs)
{
    Console.WriteLine(dog.Name);
}

/* Output:
Rover
Snowy
*/
```

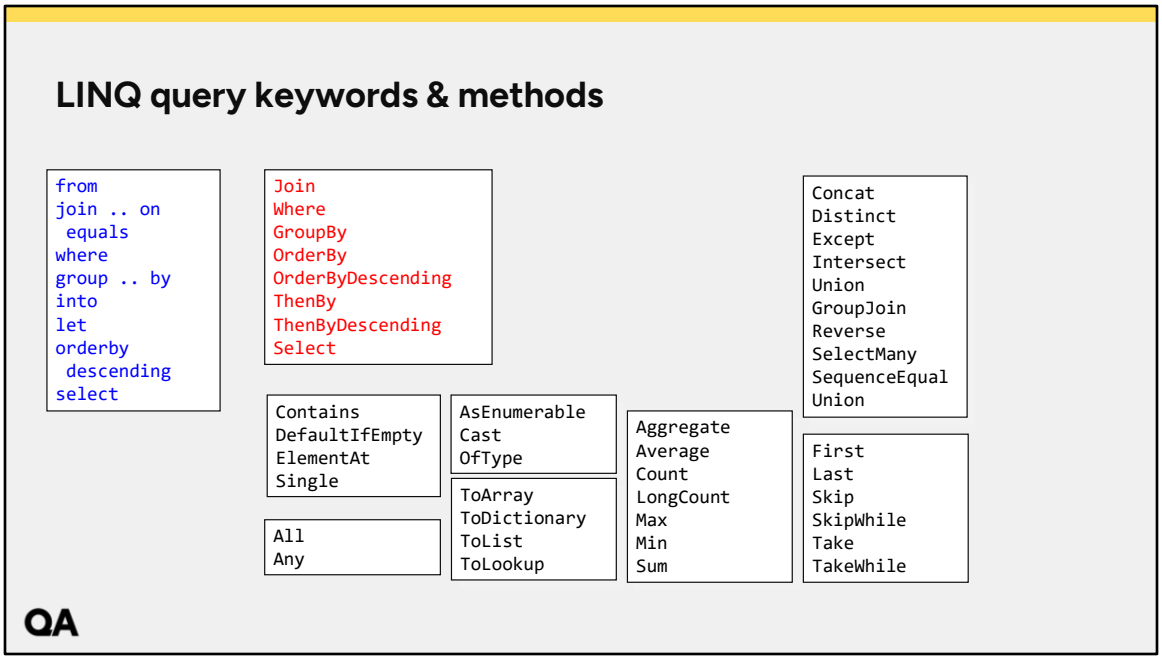
QA

LINQ Best Practice

- Use the right data structure (List<T>, HashSet<T>, Dictionary<TK, TV>)
- Understand deferred execution.
- Know the difference between IQueryable< > and IEnumerable< >
- Understand lazy-loading and performance
- Use Select and Where wisely
- Avoid Multiple Enumerations

QA

- Use the right data structure (List<T>, HashSet<T>, Dictionary<TK, TV>)
- Make sure you understand the concept of deferred execution and how that relates to how queries are generated and executed.
- Know the difference between IQueryable<> and IEnumerable<>
  - Compile time vs runtime query execution
- Understand lazy-loading and how it will affect the performance of your queries.
  - By default LINQ supports deferred execution (the query is not executed until the results are actually needed)
- Use Select and Where wisely
  - Use Select to return just the required properties
  - Use Where to filter out unnecessary objects
- Avoid Multiple Enumerations
  - Mitigate by using ToList()



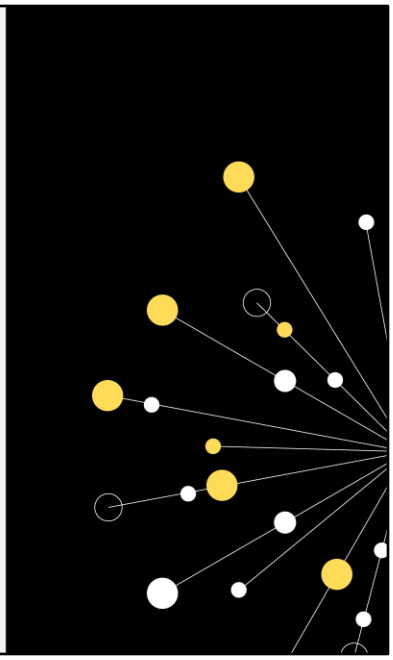
The top left box (blue text) contains the query expression keywords.

The other boxes list the extension methods of class System.Linq.Enumerable. Use of the keywords compiles to calls to the methods in red with a similar name. Many methods are hugely overloaded and many take 1, 2 or even 3 instances of one of the delegate type Func<...>.

Summary

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

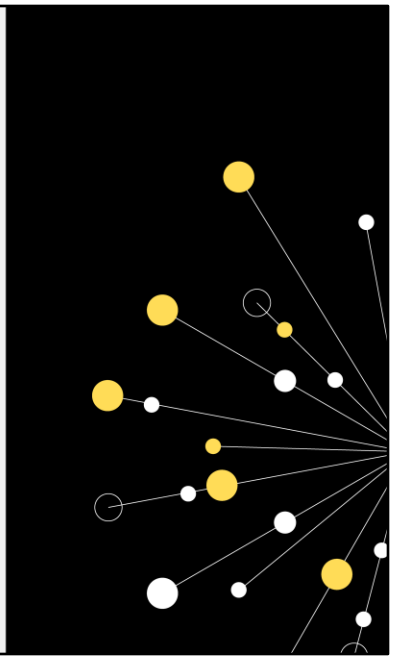
QA



Activity

Exercise 07 LINQ

Convert non-LINQ code into the LINQ equivalent

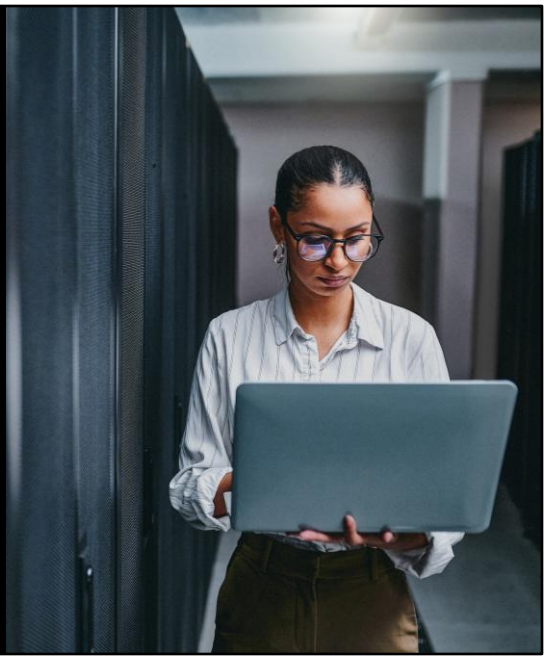


QA

08 Exception Handling

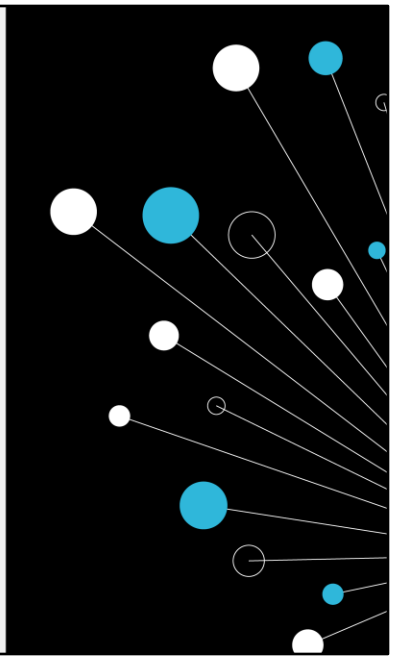
C# The Programming Language Part 2

QA
Learn. To Change.



Learning objectives

- Be able to create robust code that anticipates the potential of exceptions being thrown and handles them appropriately
- Know how to write and throw your own exceptions
- Understand Exception handling best practice

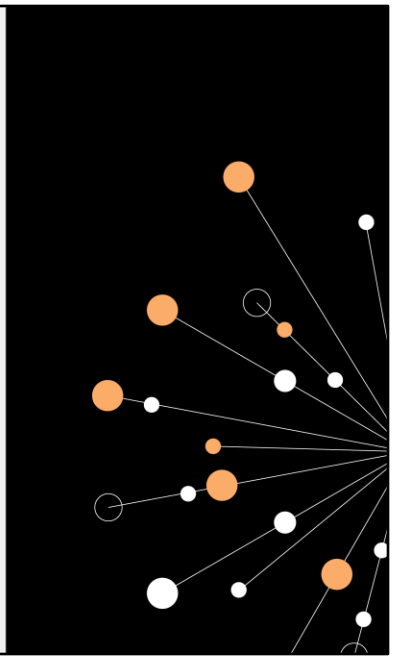


QA

Content

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Best practices

QA



Exception Handling

The **exception handling** features of the C# language let you deal with any unexpected or exceptional situations that occur whilst your code is running

There are four keywords used:

- **Try**: Try actions that may not succeed
- **Catch**: Handle failures
- **Finally**: Clean up resources
- **Throw**: Generate an exception

QA

Programs will fail from time to time, particularly when accessing resources in other processes or across a network: printers can run out of paper, databases can become unavailable and users can enter incorrect security credentials.
The .NET Framework takes a single approach to error reporting: the throwing and catching of exceptions.
Exception handling is designed to ensure that programs take note of problems as they occur, thus hopefully stopping them writing garbage data or causing a traumatic system failure.
Exceptions are types. They have properties and methods.

Example: Try Catch finally

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //do things with connection
}
catch (SqlException ex)
{
    Console.WriteLine("Data access error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("General error: " + ex.Message);
}
finally
{
    if (conn != null && conn.State == ConnectionState.Open)
    {
        conn.Close();
    }
}

//remainder of method
```

QA

The three components of exception handling code are the try, catch and finally blocks. These work as follows:

`try`

The try block is where we write the code that is the normal flow of execution for the application. In this example this consists of opening a connection to a SQL Server database, and performing operations against it (code not shown).

`catch`

Following a try block there can be zero-to-many catch blocks. These contain statements that are to be executed if an exception of the specified type is thrown (raised) by the code that is called in the try block. If there is more than one catch block they must be in a very specific order: the most derived exception type blocks must appear before their base class counterparts. This is because the exception handling code walks the blocks in sequence, looking for the first exception that matches according to the "is" operator. Only ONE of the exception handling blocks will ever be executed.

`finally`

Optionally, there can be a finally block. Unless there are no catch blocks, in which case it is required. The code in this block will always execute, no matter whether an exception is thrown (and/or caught) or not.

A try block must be followed EITHER by one or more catch blocks and optionally a finally block OR by a finally block.

The next slides illustrate how this fits together and how exceptions affect the flow of execution of code.

Understanding Execution Flow: 1

```
public class Program {
1   static void Main() {
2       try {
            Task.F1( 0 );
            Task.F2();
        }
3       catch (Exception ex)
4       {
5           Console.WriteLine(ex.Message);
6       }
7   }
}
```

```
public class Task {
    public static void F1(int a) {
        F3(a);
        F4();
11  }
12
    public static void F2() { }

31  public static void F3(int y) {
32      int x = 10 / y;
        // Does not run
    }
    public static void F4() { }
}
```

Step

QA

In this first example, we have a simple program that, under normal conditions, would execute two methods (Task.F1 and Task.F2). The method F1 calls the methods F3 and F4 respectively.
However, there is a potential problem in F3. When we divide by zero using integer division the result is undefined. Consequently, the author of the System.Int32 class decided that they would **throw** an exception of type **System.DivideByZeroException** under these circumstances.
When this exception is thrown, the CLR starts to "unwind the stack" until it finds a matching exception handler for **DivideByZeroException** (or one of its base classes). Therefore, the code after the divide by zero action in the method F3 doesn't get executed. Since the call to F3 is not inside a **try** { } block there can be no **catch** handler here. Therefore the stack continues to unwind and F4 is not executed.
The stack has now unwound to the point that F1 was invoked (in Main). This call is inside a **try**{ } block, so the exception handling code looks to see if there is a matching **catch**{} block. It locates the block for **System.Exception**, which is the ultimate base class for all exception types, so the thrown exception matches the "is a type of" test. Control is therefore transferred to this **catch**{} block and the stack unwinding process is now complete. The **catch**{} block simply displays the exception's error message information on the console. The code following the **catch**{} block then executes, which means that Main returns as normal. Therefore F2 does not get executed.

Understanding Execution Flow: 2

```
public class Program {
1   static void Main() {
2     try {
3       Task.F1( 0 );
4       Task.F2();
      }
      catch (Exception ex) {
          Console.WriteLine(
            ex.Message);
      }
    }
55
```

```
public class Task {
11    public static void F1(int a) {
12        F3(a);
13        F4();
14    }

22    public static void F2() { }

31    public static void F3(int y) {
32        int x;
33        try {
34            x = 10 / y;
              // Does not run
          }
35        catch (DivideByZeroException ex)
36        { }
37        // Rest of method
      }
41    public static void F4() { }
}
```

Step

QA

In this example, the developer of the Task class has anticipated that someone might call the method F3 and pass in the value zero. They have therefore protected the division operation with a **try**{} **catch**{} block to catch any divide by zero exceptions. Here's the flow of execution when 0 is passed in:
Main calls Task.F1, passing in zero.
Task.F1 calls Task.F3, passing the value 0 through in the 'a' parameter.
Task.F3 attempts to perform its division operation, and a **DivideByZeroException** is thrown by the internals of the System.Int32 class.
The stack begins to unwind. As the code in F3 is inside a **try**{} block, the CLR looks to see if there is a matching **catch**{} block. There is, which means that control is passed to the **catch**{} block. The code after the division by zero operation in F3 doesn't execute.
As the exception has been caught in F3, the method F3 returns as normal to F1. Method F4 then executes as normal, and assuming that no exception is generated, control is returned to F1 as normal. F1 then returns to Main, and F2 executes. Again, assuming that no exception is thrown in F2, control returns to Main as normal. As no exception was thrown, without being caught, by the methods F1 and F2 (and the methods they called in turn) the **catch**{} block is skipped, and Main returns as normal.

Understanding Execution Flow: 3

```
public class Program {
1   static void Main() {
2     try {
3       Task.F1( 0 );
        Task.F2();
      }
4     catch (Exception exn)
5     {
6       Console.WriteLine(
          exn.Message);
7     }
8   }
}
```

Step

QA

```
public class Task {
11    public static void F1(int a) {
12        F3(a);
          F4();
      }

      public static void F2() { }

31    public static void F3(int y) {
32        int x;
32        try {
33            x = 10 / y;
              Console.WriteLine(x);
          }
34        finally {
35            Console.WriteLine("Other");
36        }
          // does not run if try fails
      }
      public static void F4() { }
}
```

This example highlights the use of **finally** to ensure the execution of code, no matter whether an exception is thrown or not. As ever, Main calls Task.F1, passing in the value zero. F1 calls F3, passing the value zero through. F3 attempts to perform the division, but the System.Int32 class throws the **DivideByZeroException**. The CLR starts to unwind the stack and locates the **finally**{} block that follows the **try**{} block in which the division was performed. It therefore executes the code inside the **finally** block, and displays “Other” to the console.

At this point the CLR is still looking for a **catch**{} block to handle the exception; remember the **finally**{} block only guarantees the execution of code and doesn't handle exceptions. The stack is therefore unwound until the **catch**{} block is located in Main, where the exception's message is written to the console.

In this example, the code that writes the value of ‘x’ to the console isn't executed if an exception is thrown by the division operation (nor is the code in F4 or F2). You can therefore see the benefit of the **try** / **finally** construct when you need to ensure that certain code is always executed. This is particularly important for ensuring the database connections and file streams are always closed.

Throwing Exceptions

To generate an exception, you throw a reference to an exception object:

```
void PrintReport(Report rpt) {
    if (rpt == null) {
        throw new ArgumentNullException
            ("rpt", "Can't print a null report");
    }
    …
}
```

You can re-throw a caught exception which maintains the original stack trace:

```
catch (ArgumentNullException ex) {
    …
    throw;
}
```

QA

There will be times when we will want to throw exceptions. For example, when writing a method that takes an object reference as a parameter. Our code might reasonably expect this reference to be set to a valid object, rather than be null. Consequently, we can test it to see whether it is null and if it is we can then throw a **System.ArgumentNullException.**

Most exceptions provide a set of overloaded constructors through which we can provide additional information. In the example above, the exception lets us specify the parameter name and a helpful message.

We can also re-throw exceptions that we have caught. This allows us to perform some clean up code in response to the exception but then propagate the exception back up the call stack. To re-throw the exception simply use the "**throw**" keyword by itself inside a **catch** block.

This type of throw statement preserves the original exception, including the originating stack trace.

It should be noted that many of the exception classes also support overloaded constructors that accept an "inner exception". An example of when this might be of use is in a Data Access Layer component (one that talks to a database), where we want to enclose the low level exception (to preserve traceability) inside a custom exception that we are developing (to provide a domain-specific exception type).

Custom Exceptions

Derive the class from **System.Exception**

```
public class CarFactoryException : Exception
{
    0 references
    public CarFactoryException() { }
    // Other overloaded constructors / properties / fields
}

1 reference
public class InvalidModelException : CarFactoryException
{
    0 references
    public InvalidModelException()
    {
    }
}
```

To provide rich exception details:

- Overload the constructor to pass in information
- Provide public properties to allow retrieval

QA

We can create custom domain-specific exception classes for use in our applications. All exception types are derived from System.Exception. Custom exception types are no different in this respect.
Don't create too many exception types, and never duplicate exceptions that exist in the Framework Class Library. In most cases, one of the predefined exceptions is perfectly adequate. An excess of exceptions can make it difficult to write readable code.

Best Practice

- Use specific catch blocks for the exceptions you expect within the code
- Include a last catch block to catch **System.Exception** which will catch unexpected exceptions
- Not every method needs **try** and **catch** blocks since exceptions are propagated up the call stack
- Use **finally** blocks to tidy up resources
- Only **throw** exceptions if the situation is exceptional rather than expected
- Ensure your tests check that exceptions are thrown when expected
- Do not disclose sensitive or too much information in error messages

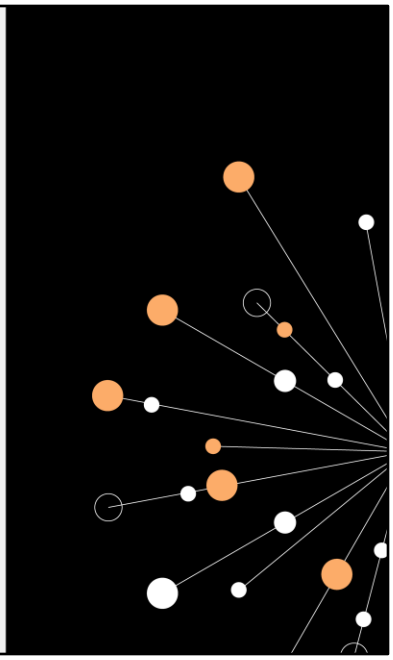
QA

Firstly, we shouldn't simply write catch handlers for System.Exception. This would catch every type of exception, including those from the CLR that are indicating traumatic problems (stack overflow, security exceptions, etc.).
This leads neatly onto the second point: only catch the specific exceptions that are expected. Code which attempts to catch every possible exception type is impossible to read.
Not every method needs try / catch blocks. Exceptions propagate back up the call stack, so only place them where, for example, logging or resource management code is required.
To manage key resources such as database connections and files, which need to be closed when finished with, use try / finally blocks.
Next, exceptions are designed to be used in exceptional circumstances such as attempting to open a file that doesn't exist, not to indicate an end-of-file (which is expected when reading from them).
Many developers dutifully add exception handling code and then never test their program under the circumstances where it is called.
Never just display an exception's Message. SqlExceptions often contain Messages like "table Persons not found". This doesn't seem like much, but a determined malicious user can use these kind of error messages to build up a picture of the system (known as a jigsaw attack). A message saying "invalid password" tells them they've guessed a username correctly.

Summary

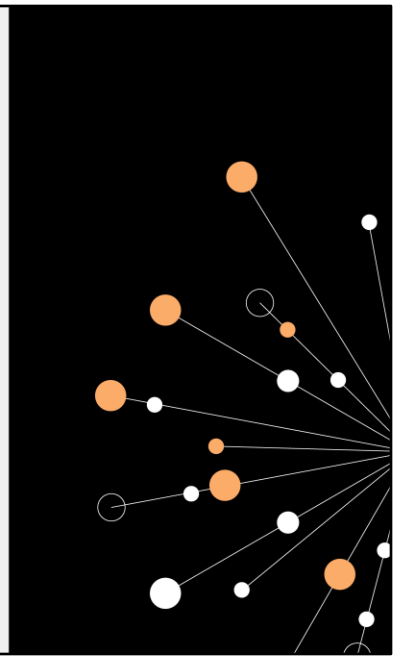
- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Best practices

QA



Activity

Exercise 08 Exception Handling



QA

xUnit and Exceptions

- xUnit enables you to test when an exception is thrown specifically within the *Act* stage of your test, as opposed to the *Arrange* or *Assert* stage
- Use Assert.Throws<TException> passing a lambda statement to perform the action you are testing
- Assert.Throws returns the exception so you can access any properties and make further assertions on those

```
[Fact]
public void Total_Price_Never_Negative()
{
    Checkout checkout = new Checkout(new TestDiscount());
    order.Add(new Pizza(Size.Small_10, Crust.Regular_2));

    Assert.Throws<NegativePriceException>(() => checkout.GetBestPrice(order));
}
```

Filtered Exceptions

- **User-filtered exception handlers** catch and handle exceptions based on requirements you define
- Use the **when** keyword with the **catch** statement:

```
try
{
    throw new MyException() { MinorFault = true };
}
catch (MyException ex)
{
    if (!ex.MinorFault)
    {
        throw;
    }
    Console.WriteLine("deal with minor fault");
}


try
{
    throw new MyException() { MinorFault = false };
}
catch (MyException ex) when (ex.MinorFault)
{
    Console.WriteLine("deal with minor fault");
}
catch (MyException ex) when (ex.MinorFault == false)
{
    Console.WriteLine("deal with major fault");
    throw;
}
```

QA

Inner Exceptions

The **Exception** class defines an **InnerException** property that enables you to wrap a custom exception around a system exception, whilst maintaining traceability as to the original cause of the exception

Your custom Exception type requires a constructor that accepts an inner exception

The **Message** and **InnerException** properties are read-only so call the base class constructor to set their values

```
public class InvalidPaintJobException : Exception
{
    0 references
    public InvalidPaintJobException()
    {
    }
    1 reference
    public InvalidPaintJobException(string message, Exception inner): base(message, inner)
    {
    }
}
```

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //look up the required paint colour
}
catch (SqlException ex)
{
    throw new InvalidPaintJobException(
        message: "not a valid colour spec",
        inner: ex);
}
```

QA

There are a few .NET framework exceptions that will have an inner exception populated, notably the **HttpUnhandledException** in ASP.NET; by the time the server sees an unhandled exception, it has been "wrapped" in this general purpose object, whose **InnerException** property will contain the exception that was originally unhandled.

Use InnerExceptions with care. Your client code should not have to dig down more than one or two levels to find the original exception.

Imagine needing the following code:

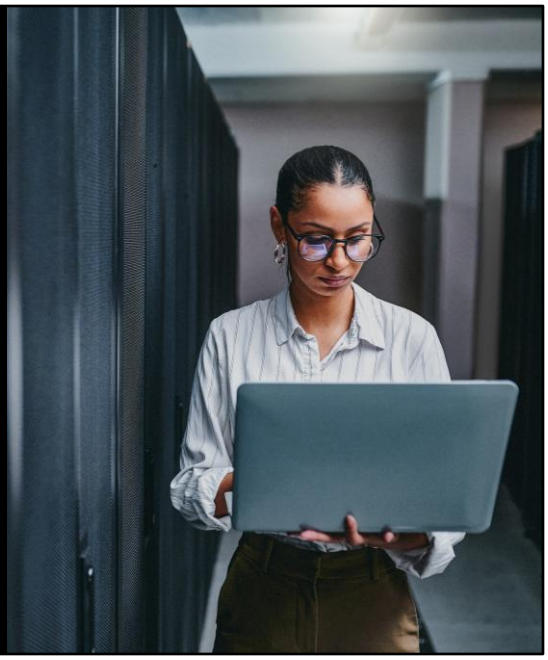
```
Exception eek =
ex.InnerException.InnerException.InnerException.InnerException;
```

There is a **GetBaseException** method on the Exception type which will find the original exception (the one furthest down the chain whose **InnerException** is null).

90 Working with Data and Files

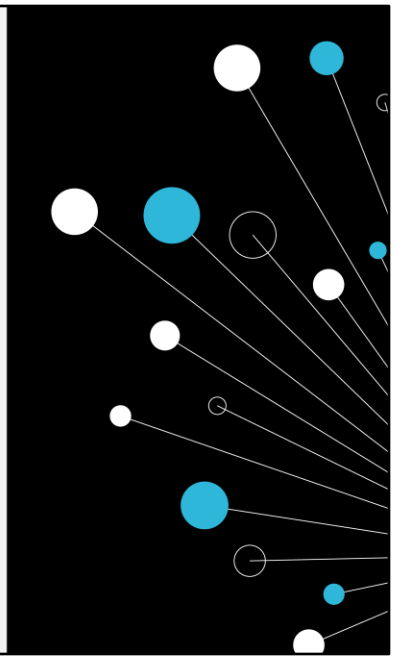
C# The Programming Language Part 2
Appendix

QA
Learn. To Change.



Learning objectives

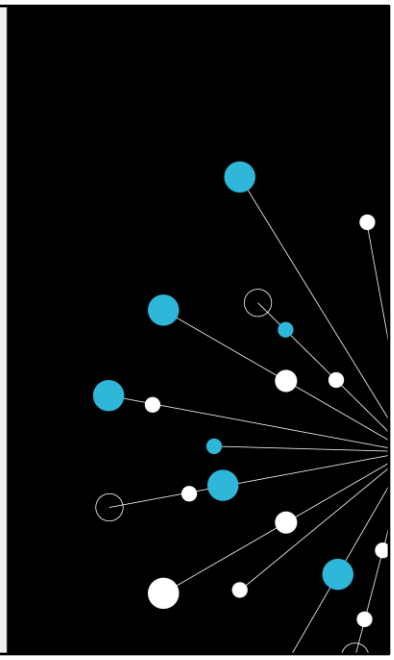
- To understand how to use C# and .NET to read to and from different kinds of files (text, CSV, JSON) and streams



QA

Contents

- Accessing Files and Folders using the File System Classes
- Managing application data by using Reader and Writer classes
- Reading and Writing JSON
- Reading and Writing CSV files



QA

The System.IO Namespace

File and Directory classes
- Allow creation, deletion and manipulation of directories and files

StreamReader and StreamWriter
- Enable a program to access file contents as a stream of bytes or characters

FileStream
- Provide random access to files

QA

The `System.IO` namespace is important because it contains many classes that allow an application to perform input and output (I/O) operations in various ways through the file system. It also provides classes that allow an application to perform input and output operations on files and directories. The `System.IO` namespace is too big to be explained in detail here. However, some of the facilities available include:

The `File` and `Directory` classes that allow an application to create, delete, and manipulate directories and files.

The `StreamReader` and `StreamWriter` classes that enable a program to access file contents as a stream of bytes or characters.

The `FileStream` class that can be used to provide random access to files.

The `BinaryReader` and `BinaryWriter` classes that provide a way to read and write primitive data types as binary values.

File and Directory Classes

FileInfo,DirectoryInfo, DriveInfo and DriveType

- Expose system information about file system objects
- FileInfo & DirectoryInfo derive from FileSystemInfo but DriveInfo does not

File, Directory, Path

- Provide static methods to allow files, directories and paths to be manipulated

FileSystemInfo

- Enumerate files, directories and drives

QA

The System.IO namespace hosts a set of classes used to navigate and manipulate file, directories and drives. The file system classes are separated into two types of class: informational and utility

Most informational classes derive from FileSystemInfo. These classes expose all the system information about file system objects. However, the DriveInfo class does not derive from FileSystemInfo because although it is an informational class (like FileInfo and DirectoryInfo) it does not share the common sorts of behaviour (for example, you can delete files and directories, but you cannot delete a drive).

The utility classes provide static methods to perform operations on file system objects.

Getting File Information

```
FileInfo mf = new FileInfo(@"C:\Labs\WarAndPeace.txt");

if ( mf.Exists ) {
     Console.WriteLine("Filename: {0}", mf.Name);
     Console.WriteLine("Path: {0}", mf.FullName);
     Console.WriteLine("Length: {0}", mf.Length.ToString());
}
```

```
FileInfo fileToCopy = new FileInfo(@"C :\Labs\WarAndPeace.txt");
fileToCopy.CopyTo(@"C:\Labs\WarAndPeace.txt");
```

QA

Unsurprisingly, the FileInfo class can be used to get information about a specified file! It is derived from FileSystemInfo and supports a number of methods and properties to that aim, some of which are listed below:

Properties

CreationTime	Gets or sets the creation time of the current FileSystemInfo object.(inherited from FileSystemInfo)
DirectoryName	Gets a string representing the directory's full path.
Exists	Gets a value indicating whether a file exists.
Extension	Gets the string representing the extension part of the file.(inherited from FileSystemInfo)
FullName	Gets the full path of the directory or file.(inherited from FileSystemInfo)
IsReadOnly	Gets or sets a value that determines if the current file is read only.
LastAccessTime	Gets or sets the time the current file or directory was last accessed.(inherited from FileSystemInfo)
Length	Gets the size of the current file.
Name	Gets the name of the file.

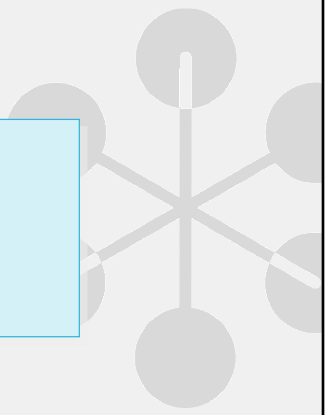
Methods

CopyTo	Copies an existing file to a new file.
Create	Creates a file.
Delete	Permanently deletes a file.
MoveTo	Moves a specified file to a new location, providing the option to specify a new file name.
Open	Opens a file with various read/write and sharing privileges.
OpenRead	Creates a read-only FileStream.
OpenText	Creates a StreamReader with UTF8 encoding that reads from an existing text file.
OpenWrite	Creates a write-only FileStream.
Replace	Replaces the contents of a specified file with the file described by the current FileInfo object, deleting the original file, and creating a backup of the replaced file.

DirectoryInfo

The DirectoryInfo class can be used to get information about directories and host an enumerable GetFiles collection

```
DirectoryInfo di = new DirectoryInfo(@"C:\Windows");
Console.WriteLine("Directory: {0}", di.FullName);
foreach (FileInfo fi in di.GetFiles()) {
  Console.WriteLine("File: {0}", fi.Name);
}
```



QA

You equally won't be surprised to learn that the DirecoryInfo class can be used to get information about a specified directory! It is derived from FileSystemInfo and supports a number of methods and properties some of which are listed below:

Properties

Attributes	Gets or sets the FileAttributes of the current FileSystemInfo.(inherited from FileSystemInfo)
CreationTime	Gets or sets the creation time of the current FileSystemInfo object.(inherited from FileSystemInfo)
Exists	Gets a value indicating whether the directory exists.
Extension	Gets the string representing the extension part of the file.(inherited from FileSystemInfo)
FullName	Gets the full path of the directory or file.(inherited from FileSystemInfo)
LastAccessTime	Gets or sets the time the current file or directory was last accessed.(inherited from FileSystemInfo)
Name	Gets the name of this DirectoryInfo instance.
Parent	Gets the parent directory of a specified subdirectory.
Root	Gets the root portion of a path.

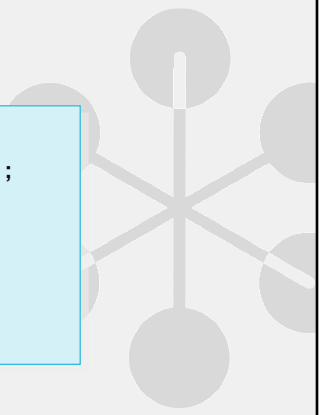
Methods

Create	Creates a directory.
CreateSubdirectory	Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class.
Delete	Deletes a DirectoryInfo and its contents from a path.
GetDirectories	Returns the subdirectories of the current directory.
GetFiles	Returns a file list from the current directory.
GetFileSystemInfos	Retrieves an array of strongly typed FileSystemInfo objects representing files and subdirectories of the current directory.
MoveTo	Moves a DirectoryInfo instance and its contents to a new path.

The Path Class

Allows the interrogation and parsing of individual parts of a file system path and the ability to change a file extension

```
string p = @"C:\Labs\WarAndPeace.txt";
Console.WriteLine("Extension: {0}", Path.GetExtension(p));
// Note, does not actually change the file name
string renamedFile = Path.ChangeExtension(p, "bak");
Console.WriteLine($"Extension: {renamedFile}");
Console.WriteLine($"Extension: {Path.GetExtension(p)}" );
```



QA

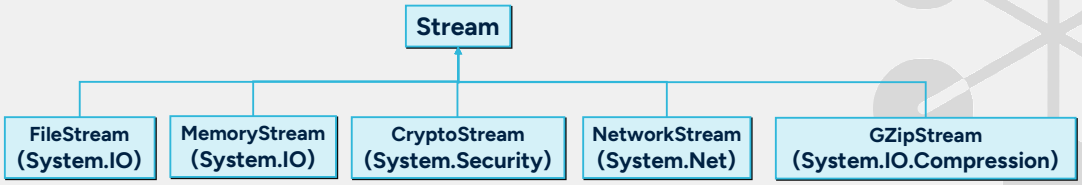
The `Path` class provides methods for manipulating a file system path, some of which are listed below:

Methods

ChangeExtension	Changes the extension of a path string. Note only the path string changes, not the actual file name extension.
Combine	Combines two compatible path strings.
GetDirectoryName	Returns the directory information for the specified path string.
GetExtension	Returns the extension of the specified path string.
GetFileName	Returns the file name and extension of the specified path string.
GetFullPath	Returns the absolute path for the specified path string.
GetPathRoot	Gets the root directory information of the specified path.
GetTempFileName	Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.
GetTempPath	Returns the path of the current system's temporary folder.
HasExtension	Determines whether a path includes a file name extension.
IsPathRooted	Gets a value indicating whether the specified path string contains absolute or relative path information.

Streams

- Common way of dealing with sequential and random access to data within the .NET Framework
- Stream is an abstract base class



QA

A stream is an important concept in modern programming. Earlier programming languages such as Fortran, Cobol and Basic had input and output built in to those languages. Modern languages like C# tend to avoid having this and instead implement input and output via library functions. Output involves sending a stream of bytes to a device and whether that device is the screen, an object in memory, a file on disk, another computer on a network or a printer is unimportant. The same routines can be used with different destinations.
In earlier languages you needed separate routines to handle each device. Streams make this flexibility possible.
Streams involve three fundamental operations:

- You can read from streams. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
- You can write to streams. Writing is the transfer of data from a data structure into a stream.
- Streams can support seeking. Seeking is the querying and modification of the current position within a stream. Seek capability depends on the kind of backing store a stream has. For example, network streams have no unified concept of a current position, and therefore typically do not support seeking.

Stream is the abstract base class of all streams.

Some thoughts before we see real IO Code

Reading/Writing files – you are interacting with the outside world
- File(name) might not exist, error situation, an 'exception' is thrown
- Open 'file handles' - an unmanaged resource (not Garbage Collected)
  - You are responsible for 'closing' files to free up resources
- This 'must happen' regardless of success / failure

C# has a try {…} finally {…} construct
- When something 'must run regardless' - learn it later in 'Exceptions'
- finally blocks are 'guaranteed' to run (even with unhandled exception)
- But in IO code these can end up being nested (and messy)

C# designers invented a 'using' statement for making IO easier
- Not the 'using' directives that sit at the top of a file!
- IO is done via 'using' statements that compile into try / finallys

QA

We are about to go outside our cosy C# 'Visual Studio' world and interact with the file system.
Perhaps create 'open file' handles, these represent unmanaged windows resources that are not garbage collected. However, the Close() methods of the FCL IO classes will free up these resources.
So basically 'Close' has to happen, but it needs to happen if any sort of exception happens whilst the file is being processed. This could be an IO exception or any sort of exception thrown by your code whilst processing the data of a file.
The C# language as you will learn in a later 'Exceptions' chapter has a mechanism for 'always' running some code. It is called a finally block and it sits after a 'try' block.

```
try {
..
..   // exception might happen here
} finally {
..   // but this still runs enabling 'cleanup to happen
}
```

Unfortunately when doing IO you are often working with multiple files and they all need closing so try / finally blocks can become nested and messy.
To solve this problem the language designers invented a 'using' statement that compiles into a try finally and can easily be nested.
Read on ..

The using statement – that simplifies code

```
using System.IO;
public void DoingIO() {
  using (StreamReader rdr = new StreamReader("in.txt")) {
    using (StreamWriter wtr = new StreamWriter("out.txt")) {
      string line;
      while ((line = rdr.ReadLine()) != null) {
        wtr.WriteLine(line);
      }
    }
  }
}
```

Using 'statement' in the code

Each 'using' compiles into '..do stuff..but always invoke Dispose().'

2 using's needed here

Client code handles any IO Exception

```
using (StreamReader rdr1 = new StreamReader("in1.txt"),
                    rdr2 = new StreamReader("in2.txt")) {
  // code that uses both rdr's...
}
```

When 2 refs of same type

QA

StreamReaders and StreamWriters provide basic functionality to read and write from and to a stream. The example above shows how you can perform a simple file copy operation.
To open a file for reading, the code in the example creates a new `StreamReader` object and passes the name of the file that needs to be opened in the constructor. Similarly, to open a file for writing, the example creates a new `StreamWriter` object and passes the output file name in its constructor. The example program then reads one line at a time from the input stream and writes that line to the output stream.
To free unmanaged resources (File handles etc) `Close()` or `Dispose()` can be called in a finally block. This can end up with some very messy try/try/try finally/finally/finally type code (covered in 'Exceptions' later)even. There is an implicit assumption here that any IO Exception wants to be passed back unhandled. Fortunately, the language designers came up with an easy way of getting the compiler to generate the try / finally's…the using statement.
You don't need to worry about scope as you do with ret and finally 'blocks'.
In the 1st example above 'rdr' is clearly in scope in the nested using. If you viewed this in ILDASM you would see
try…try ..finally (including Dispose() of wtr) …finally (including Dispose() of rdr).
In the second example because both rdr1 and rdr2 are of same data type we can cope with one using block.

The File Class

Provides basic functionality to open file streams for reading and writing

```
using (FileStream fs = File.Open(@"C:\bootx.ini",
                          FileMode.Open, FileAccess.Read)) {
  using (StreamReader sr = new StreamReader(fs)) {
    Console.Write(sr.ReadToEnd());
  }
}
```

The `OpenText` method makes it simple to open a file for reading

```
using (StreamReader sr = File.OpenText(@"C:\boot.ini")) {
  Console.Write(sr.ReadToEnd());
}
```

QA

The File class provides the basic functionality to open file streams for reading and writing. It supports a variety of Open methods:
Open, OpenText, OpenRead and OpenWrite
The FileMode enumeration is used to specify how a file is to be opened or created. FileMode.Open opens an existing file. The FileAccess enumeration is used to determine the rights required when opening a table. FileAccess.Read is used to gain read-only access to its contents.
If all you want to do is read out the entire file the ReadAllText method hides all the details of the stream and reader implementation:
Console.WriteLine(File.ReadAllText(@"C:\boot.ini");
To write to a file you must open it for writing, the process being similar to opening a file for reading:

```
 //Explicit filestream
using(FileStream fs =
File.Open(@"C:\myFile.txt",FileMode.Create, FileAccess.Write)){
  using(StreamWriter sw = new StreamWriter(fs)) {
    sw.WriteLine("blah blah blah");
  }
}
//Implicit filestream
using(StreamWriter sw = File.CreateText(@"C:\myOtherFile.txt"))
{
  sw.WriteLine("more blah blah blah");
}
//
File.WriteAllText(@"C:\myFile.txt", "stuff");
```

JSON

- stands for **J**ava**S**cript **O**bject **N**otation
- Lightweight format for storing and transporting data
- Frequently used when data is passed from a web server to a client
- Self-describing and easy to understand
- Use NuGet to install Newtonsoft.Json package

```
{
  "name": "Maria",
  "age": 39,
  "favouriteFilms": [
    "Love Story",
    "Inception",
    "It's a Wonderful Life"
  ]
}
```

Serializing and Deserializing JSON

```
PersonFilms personFilms = new PersonFilms
{
    Name = "Andrew",
    Age = 50,
    FavouriteFilms = new List<string>() {
                     "Toy Story",
                     "Toy Story 2",
                     "Aliens"
    }
};
```

```
using Newtonsoft.Json;

string pfs = JsonConvert.SerializeObject(personFilms, Newtonsoft.Json.Formatting.Indented);
Console.WriteLine(pfs);
PersonFilms pfs2 = JsonConvert.DeserializeObject<PersonFilms>(pfs);
Console.WriteLine(pfs2.Name);
Console.WriteLine(pfs2.Age);
pfs2.FavouriteFilms.ForEach(f => Console.WriteLine(f));
```

JSON Anonymous Objects

```
var obj1 = new
{
    name = "Maria",
    age = 39,
    favouriteFilms = new List<string>() {
                     "Love Story",
                     "Inception",
                     "It's a Wonderful Life"
    }
};
```

```
using Newtonsoft.Json;

string json1 = JsonConvert.SerializeObject(obj1, Newtonsoft.Json.Formatting.Indented);
File.WriteAllText("file1.json", json1);

// Reading JSON into an anonymous, dynamic object then picking out elements
string s1 = File.ReadAllText("file1.json");
dynamic obj2 = JsonConvert.DeserializeObject(s1);
Console.WriteLine(obj2.name);
Console.WriteLine(obj2.age);
Console.WriteLine(obj2.favouriteFilms);
```

CSV

- A comma-separated value (CSV) file is a structured file where individual pieces of data are separated by commas.
- The first line of the file often contains "column" names
- Use NuGet to import CsvHelper package (other packages are available)

```
Title,ReleaseYear,Director,Gross
Revenue
Star Wars IV,1977,George
Lucas,775
Avatar,2009,James Cameron,2800
Inception,2010,Christopher
Nolan,837
Interstellar,2014,Christopher
Nolan,681
Men in Black,1997,Barry
Sonnenfeld,589
```

Reading a CSV File

```
public class Movie {
    public string Title { get; set; }
    public int ReleaseYear { get; set; }
    public  string Director { get; set; }
    public decimal? GrossRevenue { get; set; }
}
```

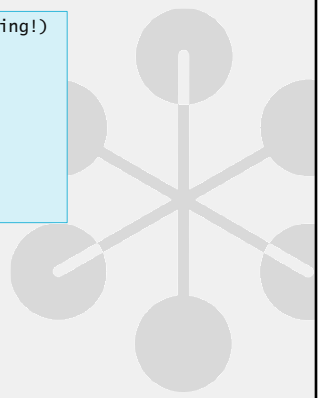
```
Using CsvHelper;

// Reading and displaying a list of people from a CSV file
using (var sr = new StreamReader("movies.csv"))

//InvariantCulture - I don't care, I don't want culture involved in the first place.
using (var reader = new CsvReader(sr, CultureInfo.InvariantCulture)) {
    var list = reader.GetRecords<Movie>().ToList();

    foreach (Movie m in list) {
        Console.WriteLine($"{m.Title} was released in {m.ReleaseYear}");
    }
}
```

QA

Note: For the above code to work you will need to install the CsvHelper package.

Reading a CSV File into a Dynamic Class

```
// Reading records into dynamic class (NB: every property value will be a string!)
using (var sr = new StreamReader("movies.csv"))
using (var reader = new CsvReader(sr, CultureInfo.InvariantCulture)) {
    var list = reader.GetRecords<dynamic>().ToList();

    foreach (var m in list) {
        Console.WriteLine($"{m.Title} was directed by {m.Director}");
    }
}
```



Writing to a CSV File

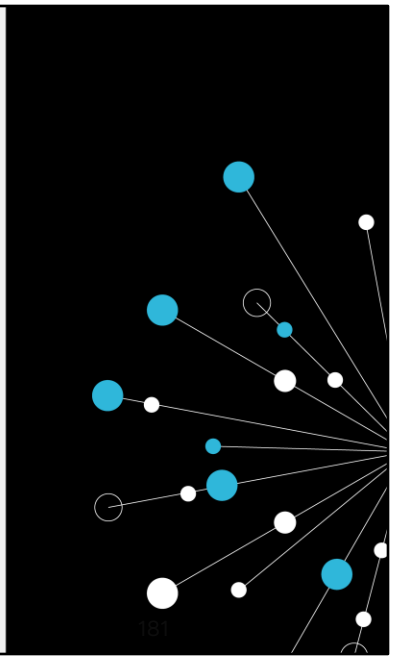
```
// Writing a list to a CSV file
var more_movies = new Movie[] {
  new Movie { Title = "2001: A Space Odyssey",
      ReleaseYear = 1968, Director="Stanley Kubrick", GrossRevenue=146 },
  ...
  new Movie { Title = "The Martian",
      ReleaseYear = 2015, Director="Ridley Scott", GrossRevenue=631 }
};
// Overwrite any existing data
using (var sw = new StreamWriter("updated_movies.csv"))
using (var writer = new CsvWriter(sw, CultureInfo.InvariantCulture)) {
    writer.WriteRecords(more_movies);
}
```

```
// Append to end of existing file and suppress header row
using (var sw = new StreamWriter("updated_movies.csv", true))
using (var writer = new CsvWriter(sw,
CultureInfo.InvariantCulture))
{
    foreach(var movie in more_movies)
    {
        writer.WriteRecord(movie);
        writer.NextRecord(); // Adds a new line (CRLF) to file
    }
}
```

QA

Activity

Exercise 07 Working with Data and Files



QA

Summary

- DirectoryInfo, FileInfo, DriveInfo allow access to Directories, Files and Drives
- Use Streams to Read and Write data
- Reading and Writing JSON
- Reading and Writing CSV files

QA