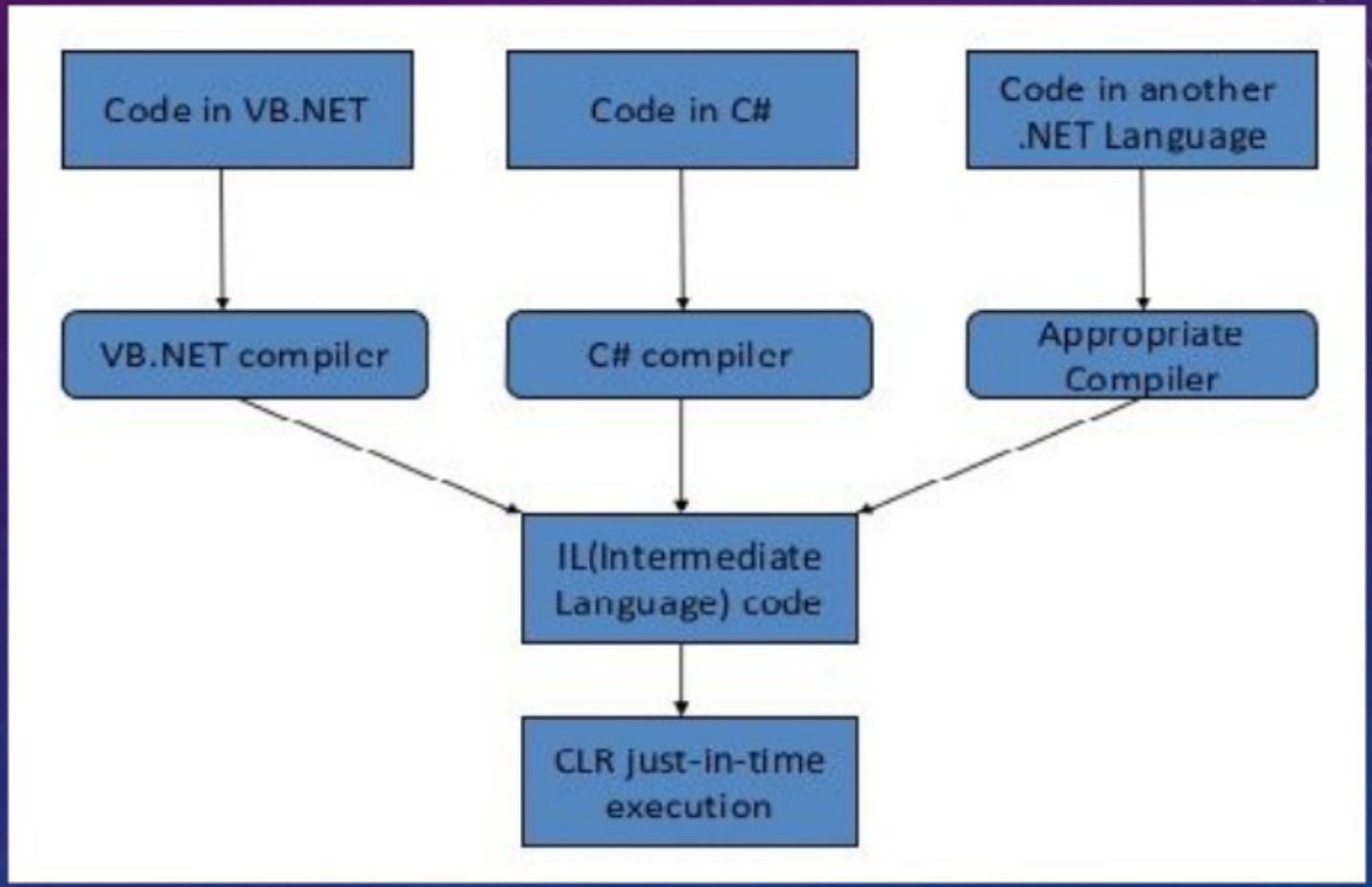
Session 1:
Introduction

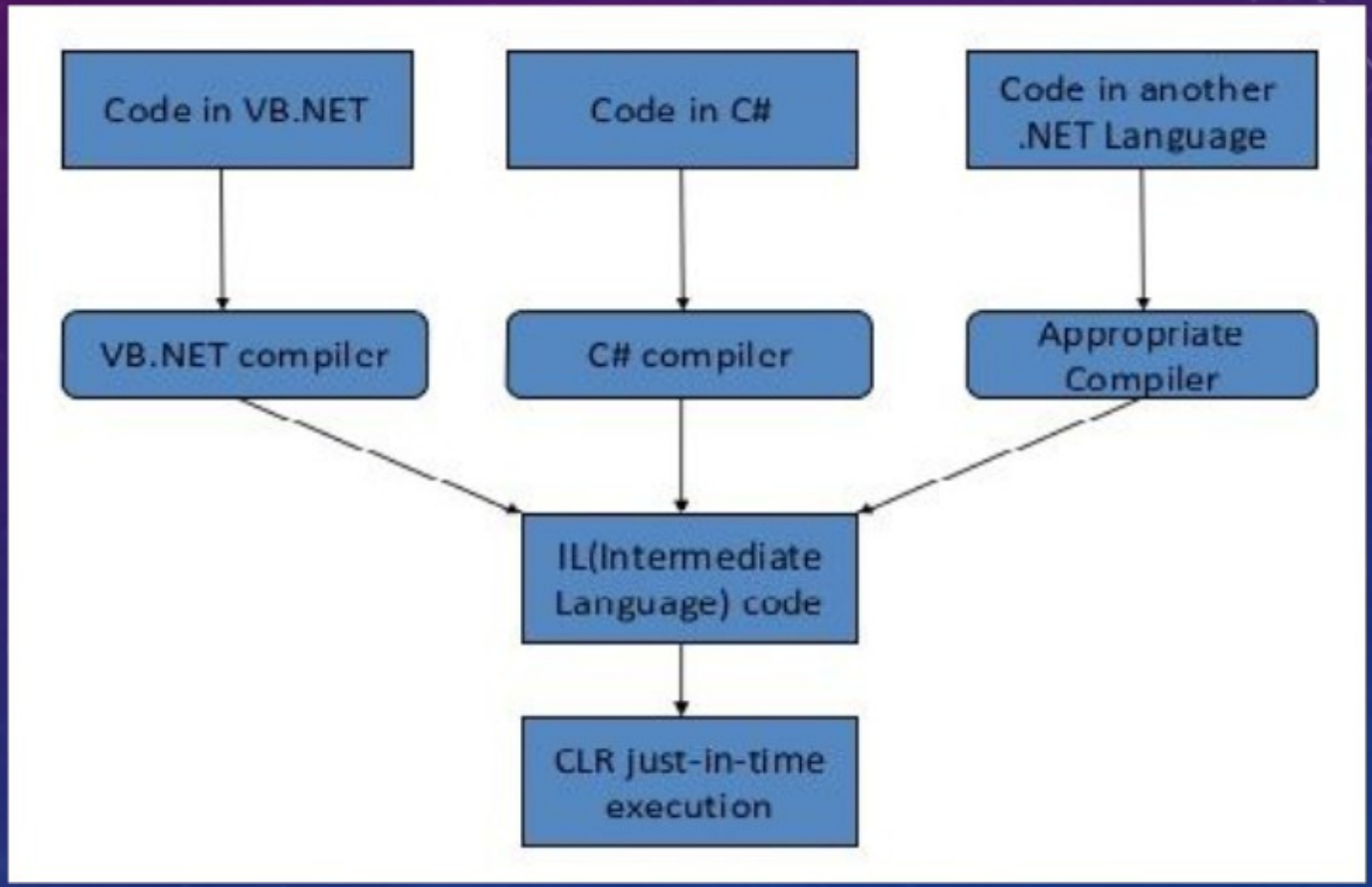
The .NET Environment

- There are three main parts to the .NET programming environment:
- C#
  - This is the programming language that we use.
  - C# is the most popular, although it's not the only option.
- The .NET Runtime
  - This is a program that needs to be installed on the user's machine – it handles actually executing the code.
- The .NET Framework Libraries.
  - These are a set of pre-written pieces of code that can be used in your program.

C# and the .NET Runtime



C# and the .NET Runtime



C# and the .NET Runtime

- C# code gets compiled to *Common Intermediate Language* code (CIL).
- The CIL then gets passed to the platform-specific runtime, which compiles it to machine-code for the target machine.
- This is not limited to C# - any .NET language can be used.
  - F#, VB.NET, etc.

Managed Code

- Code that runs under the CLR rather than directly on the hardware is called *managed code*.
  - The primary advantage of this is portability: managed code can run on any platform with a CLR runtime.
- Unmanaged code compiles directly to machine-code, without an intermediate language.
  - This means each version is platform-specific: the code has to be rebuilt for each platform it is to be used on.

JIT (Just In Time)

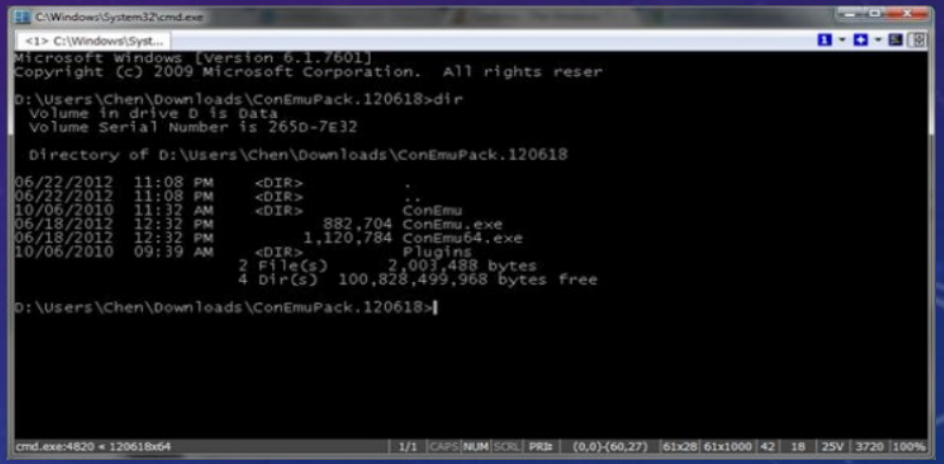
- The CLR uses a JIT compiler.
  - This means that code is only compiled to machine-code at runtime, and only the methods that are actually used will be compiled.
  - Methods only need to be compiled the first time they are run; then they are stored in the cache and can be reused without being recompiled.

Application Types

- The .NET Framework can be used to build many different types of program.
  - Console applications
  - Desktop applications – Windows Forms
  - Desktop applications – WPF
  - Windows Services
  - Websites and web services – ASP.NET

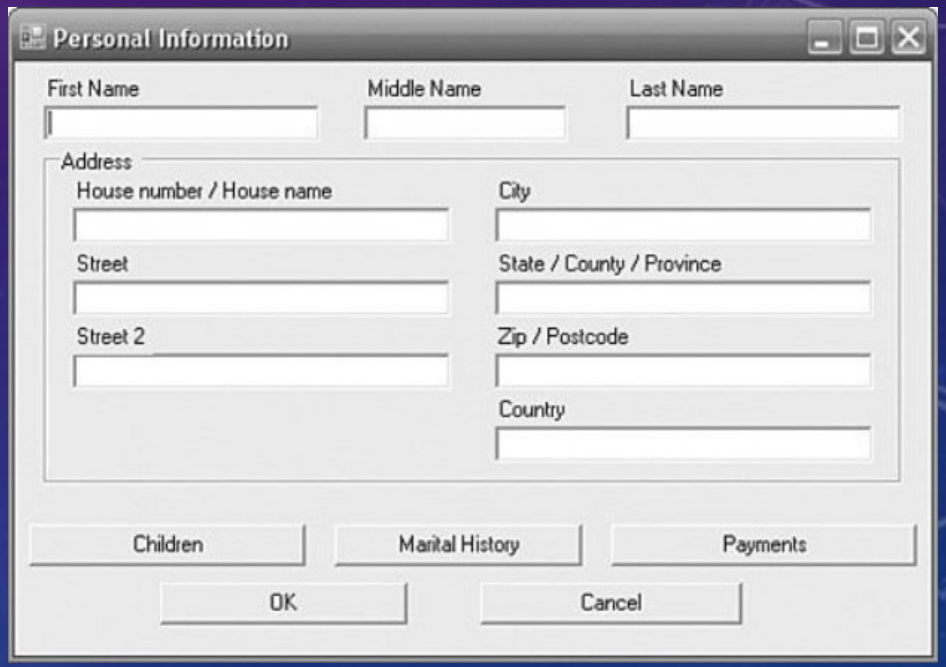
Console Applications

- Minimal 'UI' – both input and output are pure text.
- Quick to build and very small program size.
- Not used for modern 'ordinary user' programs, but are actually still quite popular for specialist tools.



WinForms

- Graphical UI rather than command-line.
- WinForms is an early GUI framework, so the options are relatively limited.
- Old, but not obsolete.



Windows Presentation Foundation

- A more recent GUI framework. WPF was built using the lessons learnt from WinForms.
- Much more powerful, but can also be more complex to use.

WinForms vs WPF

- WinForms is very well established and documented – there are a lot of resources out there.
- Many third-party controls available for purchase.
- Customising the look-and-feel is a lot of work.

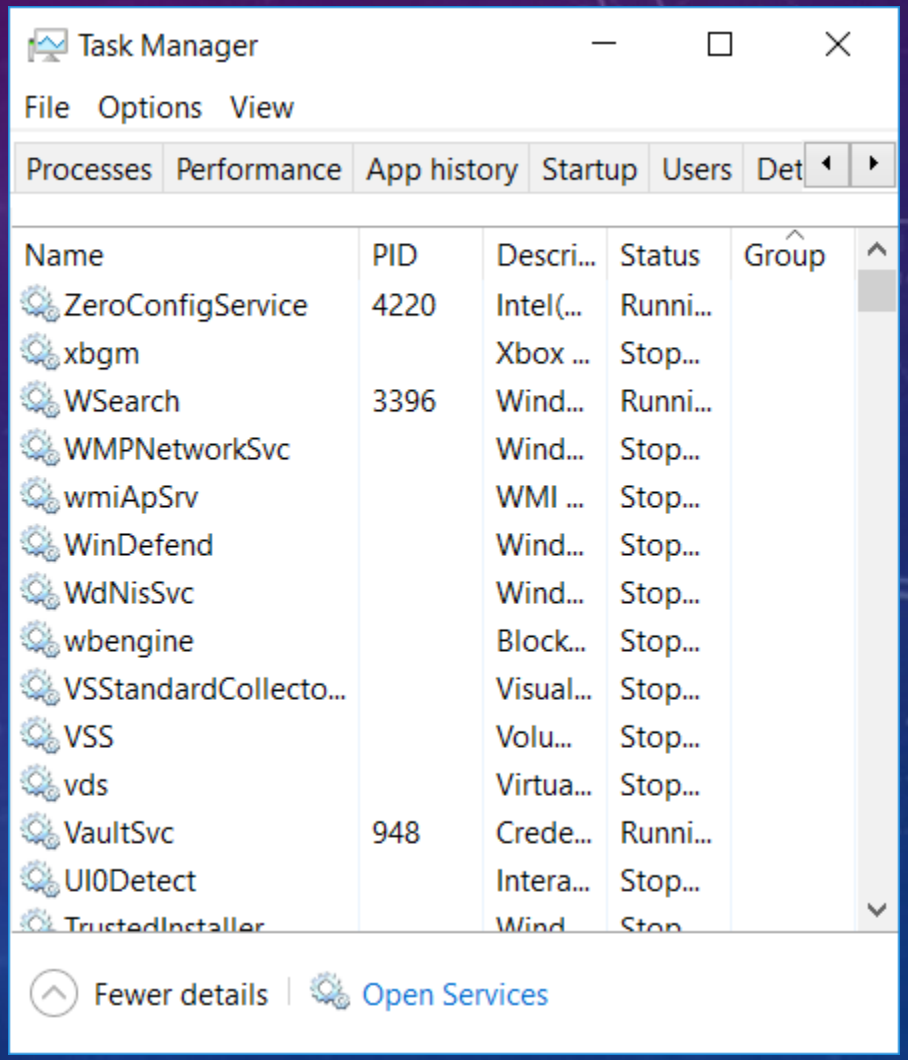
- WPF gives a more powerful set of options for your UI.
- Easy to customise your appearance and build rich data-driven apps.
- Allows for hardware-accelerated graphics.
- Requires .NET 3.0
- Advanced graphics require a DirectX 9 capable video card.

ASP.NET

- ASP.NET is a framework for web programming.
- Websites
  - Static web pages that are read in the browser – websites are defined by their *content*.
- Web applications
  - Dynamic pages that are accessed using the browser – apps are defined by their *behaviour* as much as their content.
- Web services
  - Services return JSON, XML, or some other data-oriented format.

Windows Service

- Runs in the background with no user-interface.
- Once started, they run forever (until manually stopped).
- Used a lot by operating systems and servers, less by anyone else.



C#

- C# is based on a cross between C and C++, although it has many of its own features that are not found in either language.
  - It adds many modern programming tools when compared with C, while simplifying much of the complexity of C++.
- C# is type-safe, garbage-collected, and has a wide range of features like generics, iterators, delegates and lambda expressions, nullable value types, and LINQ.
  - C# is a high-level language, while C and C++ are both low-level languages.

C# Basics

- You should be familiar with:
- Value types: int, bool, char, enum, date, etc.
- Reference types: string, objects, arrays, etc.
- Loops: for, foreach, while, do-while
- Conditionals: if, if-else, switch.
- Mathematical operators: +, -, *, /, %, +=, *=, ++, –
- Logical operators: ==, !=, <, >, <=, >=, &&, ||, !

Visibility Modifiers

- Public:
  - Fully accessible to any part of the program.
- Private:
  - Only accessible to the exact class it is defined in.
- Protected:
  - Accessible to the class it is defined in and any subclasses.
- Internal:
  - Accessible to anything in the same assembly (.exe or .dll).
- Protected Internal:
  - Counts as Protected OR Internal
- Protected Private:
  - Counts as Protected AND Internal

Abstract Classes/Methods

- An abstract class cannot be instantiated: it only exists to be a base class for others to inherit from.
- An abstract method is a method with no body: it will be implemented in the inheriting class(es).
  - All abstract methods *must* be in an abstract class.
- An abstract class *may* include normal methods, fields and properties.
- Abstract classes often also include *virtual* methods: ones which do have a default body, but are still allowed to be overridden in subclasses.

Interfaces

- An interface is similar to an abstract class, in that both only exist so that other classes can implement them.
- Interfaces can ***only*** include unimplemented methods.
  - No virtual/real methods, no fields, etc.
- Everything in an interface is automatically public; it doesn't use any visibility keywords.
- Interfaces are not included in C#'s prohibition on multiple-inheritance: they are *implemented*, not *inherited*.
  - There is no actual functionality there to *be* inherited.

Abstract Class vs Interface

- Abstract Class
  - Must be the only base class for any inheriting class.
  - Can include default behavior for subclasses to use – not every method must be overridden
  - Abstract classes represent something real but generalised; they are usually the core of a class.
- Interface
  - Can implement as many interfaces as you like.
  - Can be used alongside an abstact class.
  - Cannot have any behavior or state defined.
  - Interfaces are a contract or promise: they don't do anything, they just tell you what something else will do.

Sealed Class/Method

- The sealed keyword prevents any inheritance.
- If you use it while overriding an abstract/virtual method, it prevents subclasses from overriding it again.
- If you apply it to an entire class, then subclasses can't be created at all.

Polymorphism

- Polymorphism means having multiple versions of something.
  - The name comes from the Greek words for 'multiple shapes'
- There are two types:
  - Static or compile-time polymorphism
    - This comes from method overloading.
  - Dynamic or run-time polymorphism
    - This comes from using interfaces, base classes, and overriding abstract or virtual methods.

Method Overloading

- Overloading means defining multiple methods with the same name but different parameters.

```
Math.Round(double val);

Math.Round(double val, int precision);

Math.Round(decimal val, int precision, MidpointRounding type);


Console.WriteLine(string text);

Console.WriteLine(string format, params Object[] arg);
```

Method Overriding

- Abstract and virtual methods can be overridden in a subclass.
  - Abstract methods *must* be overridden!
- The signature (name, parameters and return type) must be identical to the original.

```
public abstract class AbstractClass
{
    public abstract void AbstractMethod();

    public virtual void VirtualMethod()
    {
        Console.WriteLine("Virtual method in Base");
    }
}

public class ConcreteClass : AbstractClass
{
    public override void AbstractMethod()
    {
        Console.WriteLine("Implementing Abstract Method");
    }
    public override void VirtualMethod()
    {
        base.VirtualMethod();
        Console.WriteLine("Virtual method is modified");
    }
}
```

C# Lock

- The lock keyword prevents multiple threads from accessing the same location at the same time.
- If a second thread tries to enter the same part of the code it will ***block*** – stop executing and wait until the first thread has finished.

```
public class BankAccount
{
    public object _lock = new Object();

    public void Withdraw()
    {
        lock (_lock)
        {
            // Critical code goes here...
        }
    }
}
```

Using

- Using has two different functions.
  - The import statements at the top of the file.
  - To auto-dispose of an object at the end of a block.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
public class TestUsingClass
{
    public void Hello()
    {
        using (var obj = new MyDisposableClass())
        {
            // code here..

        } // obj.Dispose() is called here.
    }
}

public class MyDisposableClass : IDisposable
{
    public void Dispose()
    {
        // clean up stuff here
        // e.g. close db connection
    }
}
```

Delegates

```
public delegate void Action();
public delegate void Action<in T>(T obj);

public class MyDelegateTestClass
{
    public void Test()
    {
        Action doSomeAction1Delegate = DoSomeAction1;
        DoSomething(doSomeAction1Delegate, DoSomeAction2);

        DoSomething(() => { Console.WriteLine("1"); }, (txt) => { Console.WriteLine("2: {0}", txt); });
    }

    public void DoSomething(Action beforeDoing, Action<string> afterDoing)
    {
        // should always check null for reference type
        beforeDoing();
        Console.WriteLine("Doing it..");
        afterDoing("done");
    }

    public void DoSomeAction1() { Console.WriteLine("1"); }
    public void DoSomeAction2(string text) { Console.WriteLine("2: {0}", text); }
}
```

Delegates

- Delegates are a way to save a reference to a method on its own.
- A delegate type specifies the parameters and return type that are required.
- Once a delegate type has been created, you can instantiate it and associate the instance with any method that matches the parameters and return type.
- Delegates are particularly useful for Events and Callbacks
  - You can tell the program: 'When X happens, do Y in response.'

Events

- Events are encapsulated delegates.
- They are very frequently used to handle user-input.
- When something happens – a keypress, a mouse event, etc. – the system triggers the delegates that have been associated with that event.
  - To associate a method with an event, use the += operator.

```
public Form1()
{
    InitializeComponent();
    hello = new EventHandler(WriteHello);
    button1.Click += hello;
}

private void WriteHello(object sender, System.EventArgs e)
{
    MessageBox.Show("Hello World");
}
```

Extension Methods

- Extension methods allow you to add new methods to a type without modifying the original source or creating a new derived type.
- Many libraries use these (e.g. LINQ)

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Serialization

- Serialization means turning an object into some form of data that can be stored or transmitted.
  - Deserialization means using that data to recreate the original object.
- Very commonly the data storage is text (JSON or XML), since this is convenient for web programming.
- Other formats are possible – some apps use raw byte-streams instead of text data.

Stack vs Heap

- The stack tracks what is being executed.
  - It is stored as a set of frames in a Last-In, First-Out stack (hence the name).
  - Only the top frame can be accessed.
  - The stack is self-maintaining: when a frame has finished executing, it is removed.
- The heap tracks objects that are being stored.
  - It allows random-access: anything can be accessed at any time.
  - Objects on the heap need to be handled by the garbage-collector

Boxing and Unboxing

- Value types are usually stored on the stack, reference types are always stored in the heap.
- If we convert a value type to a reference (or back to a value), then it needs to be moved from one to another.
  - This is a *relatively* expensive operation.
  - It is also a potential interview question! This tests how well you understand the underlying memory systems – very important if you're working on high-performance code.