# ICT365

# Software Development Frameworks

Dr Afaq Shah

Murdoch
UNIVERSITY

# Inheritance

Murdoch
UNIVERSITY

# In this Topic

Murdoch UNIVERSITY

- Inheritance (subclassing, interfaces, virtual methods, method overriding)
- Abstract classes and interfaces.
- Define new classes by extending existing classes.
- Access specifiers control the access of members of the base class by its derived classes.
- Constructor invocation when instantiating a derived class.
- Virtual methods and method overriding.
- Automatic type conversion.
- Dynamic method dispatch.
- System.Object class.

# Inheritance

Murdoch UNIVERSITY

Assume we already have a class named Person (name, address, age, gender, etc)

We need to design a new class that models a student.

Question:

Should we design the Student class from scratch or design the Student class based on Person class to avoid re-inventing the wheel?

# Inheritance (cont'd)

Murdoch UNIVERSITY

Like any OO language, C# supports inheritance

Instead of writing a new Student class from scratch, extend the existing Person class

Extended class - the derived class, or subclass, or child class,

The original class - base class, or superclass, or parent class.

# Inheritance (cont'd)



Murdoch UNIVERSITY

In C#, every class is either a direct or an indirect descendant of System.Object class.

The FCL forms a tree of classes rooted at System.Object.

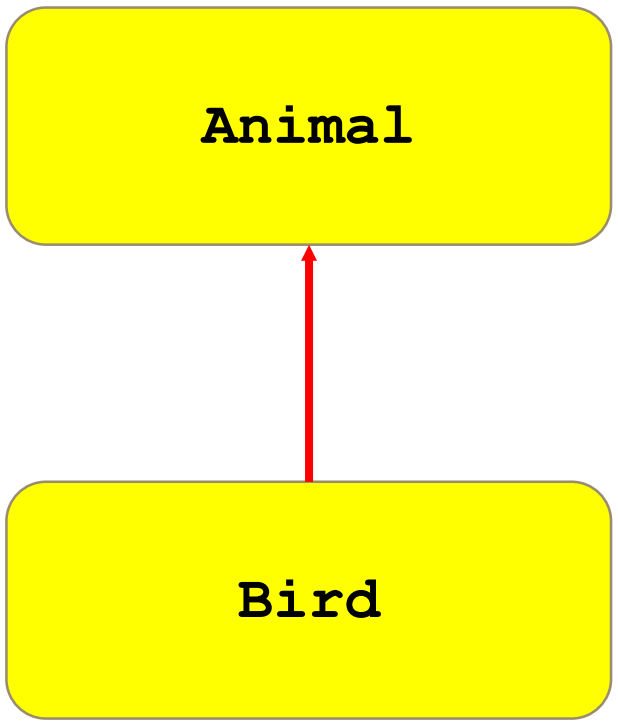
The derived class inherits all of the base class attributes

# Inheritance

Murdoch UNIVERSITY

Inheritance relationships (*class diagram)* - arrow pointing to the parent class.

Inheritance should create an ***is-a*** relationship



# Examples: Base Classes and Derived Classes


| Base class | Derived classes |
|---|---|
| `Student` | `GraduateStudent`<br>`UndergraduateStudent` |
| `Shape` | `Circle`<br>`Triangle`<br>`Rectangle` |
| `Loan` | `CarLoan`<br>`HomeImprovementLoan`<br>`MortgageLoan` |
| `Employee` | `FacultyMember`<br>`StaffMember` |
| `Account` | `CheckingAccount`<br>`SavingsAccount` |

# Declaring a Derived Class

Murdoch UNIVERSITY

- Define a new class DerivedClass which extends BaseClass

```
  class BaseClass
{
    // class contents
}
 class DerivedClass : BaseClass
 {
    // class contents
}
```

# Example1

Assume an existing class Person:

```
class Person
{
        public string Name;
        public string Address;

        public void Show()
        {
                Console.WriteLine("Name: {0},
address: {1}",  Name,  Address);
        }
}
```

# Example 1

Create a new class to model a student

No need to do it from scratch.

Efficient way - extend the class Person:

```
class Student :  Person
        // extending Person class
{
        public int StudentNo;
}
```

The new class Student inherits everything from Person class

New member StudentNo that Person does not have.

# Example 1

Murdoch UNIVERSITY

```
Student s = new Student();
s.Name = "Joe";      // from base class
s.Address  = "2 South St";   // from base class
s.StudentNo = 12345;   // from the derived class
s.Show();    // from base class
```

# Class inheritance

Murdoch UNIVERSITY

```
//The Mammal class can be instantiated.
public class Mammal
{
    public int NumberOfVertebrae{ get; set;}
    public float WarmBloodTemperature{ get; set;}
    public void GrowHair(){ /* something here */ }
    public void ProduceMilk(){ /* something here */ }
}

// Human inherits from Mammal a lot of "biological"
// functions.
public class Human : Mammal
{
     /* human attributes, functionality here */

    //Human constructor calls Mammal() first
     public Human()
     {
        super();
        /* some code goes here */
     }
}
```

# Overriding Methods

Murdoch UNIVERSITY

- A child class can *override* the definition of an inherited method
- The new method

  - the same signature as the parent's method,

  - a different implementation.
- The type of the object executing the method determines which version of the method is invoked.

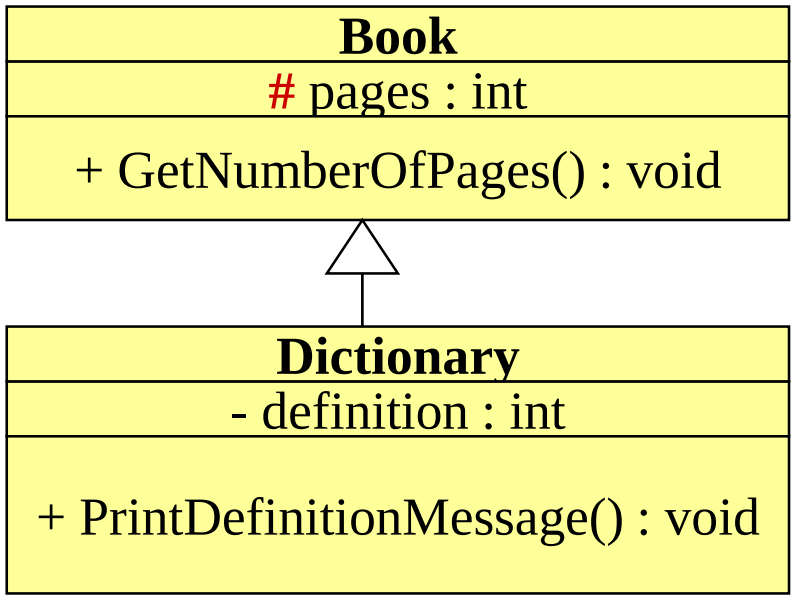
# Controlling Inheritance

Murdoch UNIVERSITY

- A child class inherits the methods and
- *private* Variables and methods are not accessible in the child class
- *public* Variables and methods are accessible (but violate our goal of encapsulation)
- Third visibility modifier: *protected?*

# The protected Modifier

Variables and methods declared with protected visibility in a parent class are only accessible by a child class or any class derived from that class




+ public
- private
# protected

# Single Inheritance

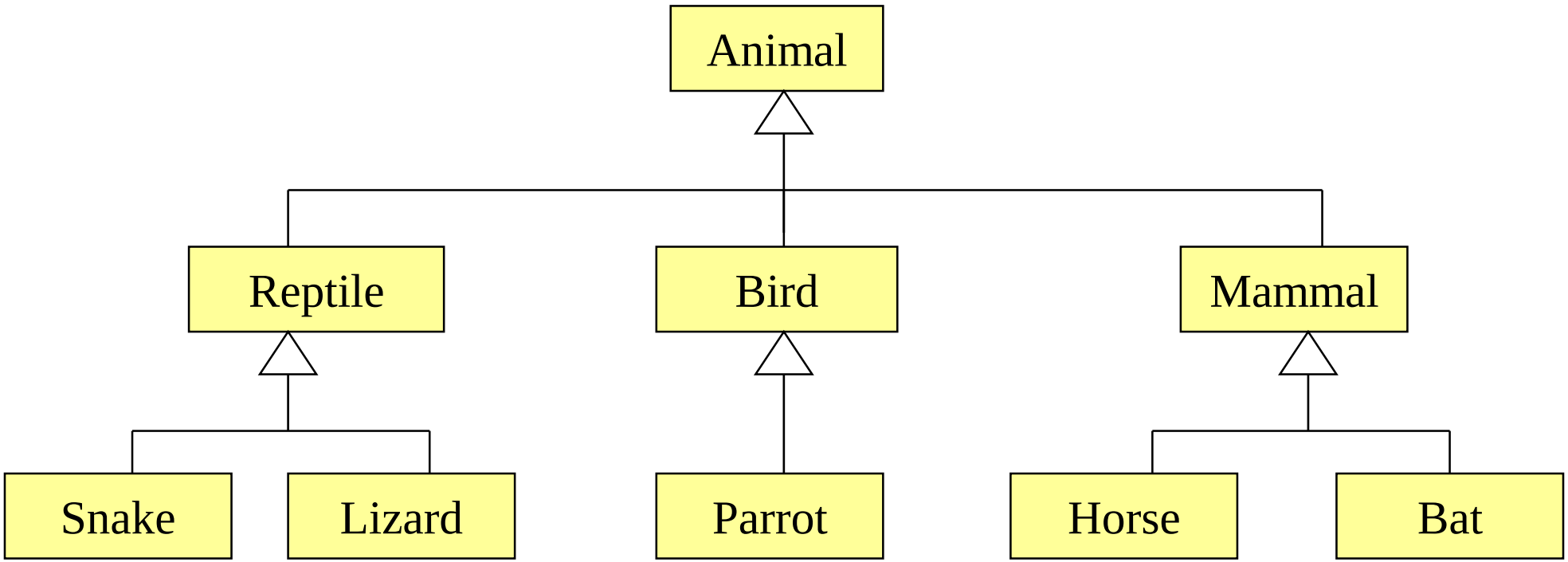
Murdoch UNIVERSITY

- Some languages, e.g., C++, allow *Multiple inheritance*.
- C# and Java support *single inheritance*

  - a derived class can have only one parent class.

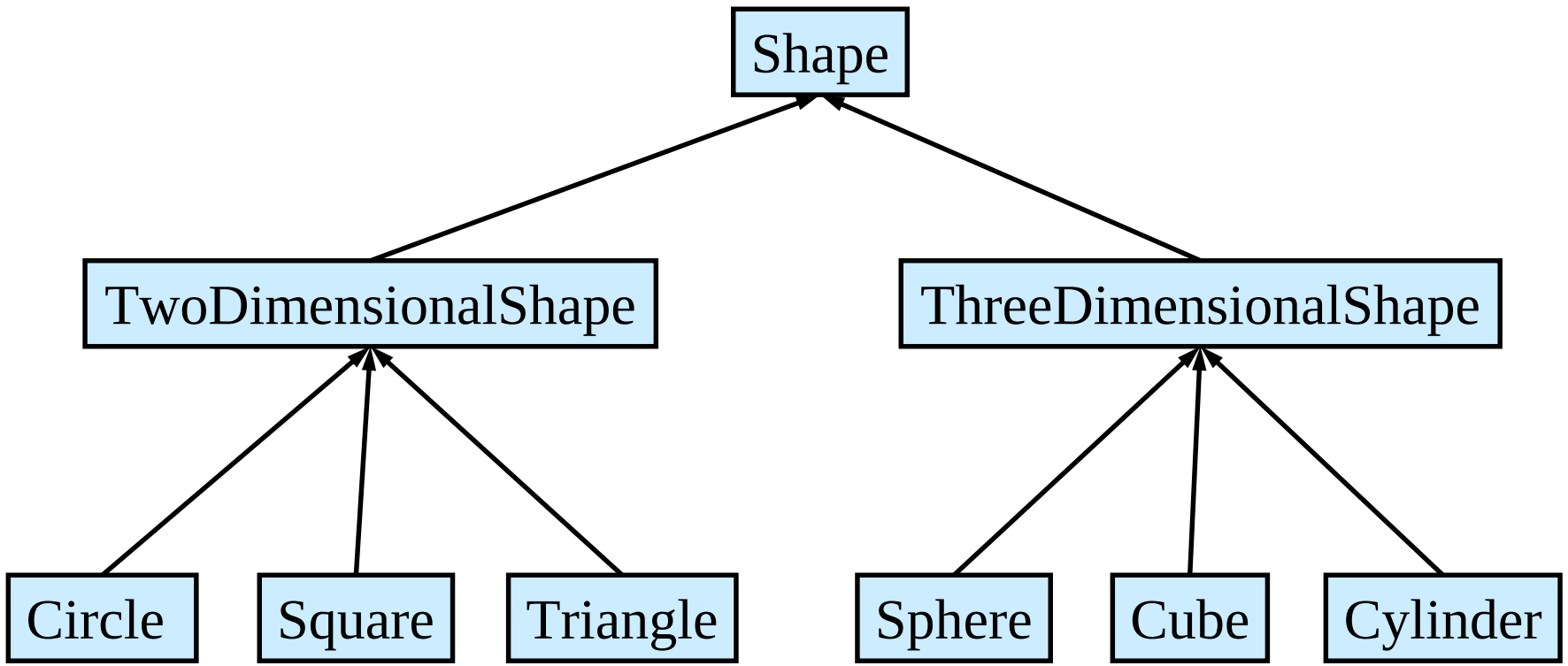
# Class Hierarchies

Murdoch UNIVERSITY



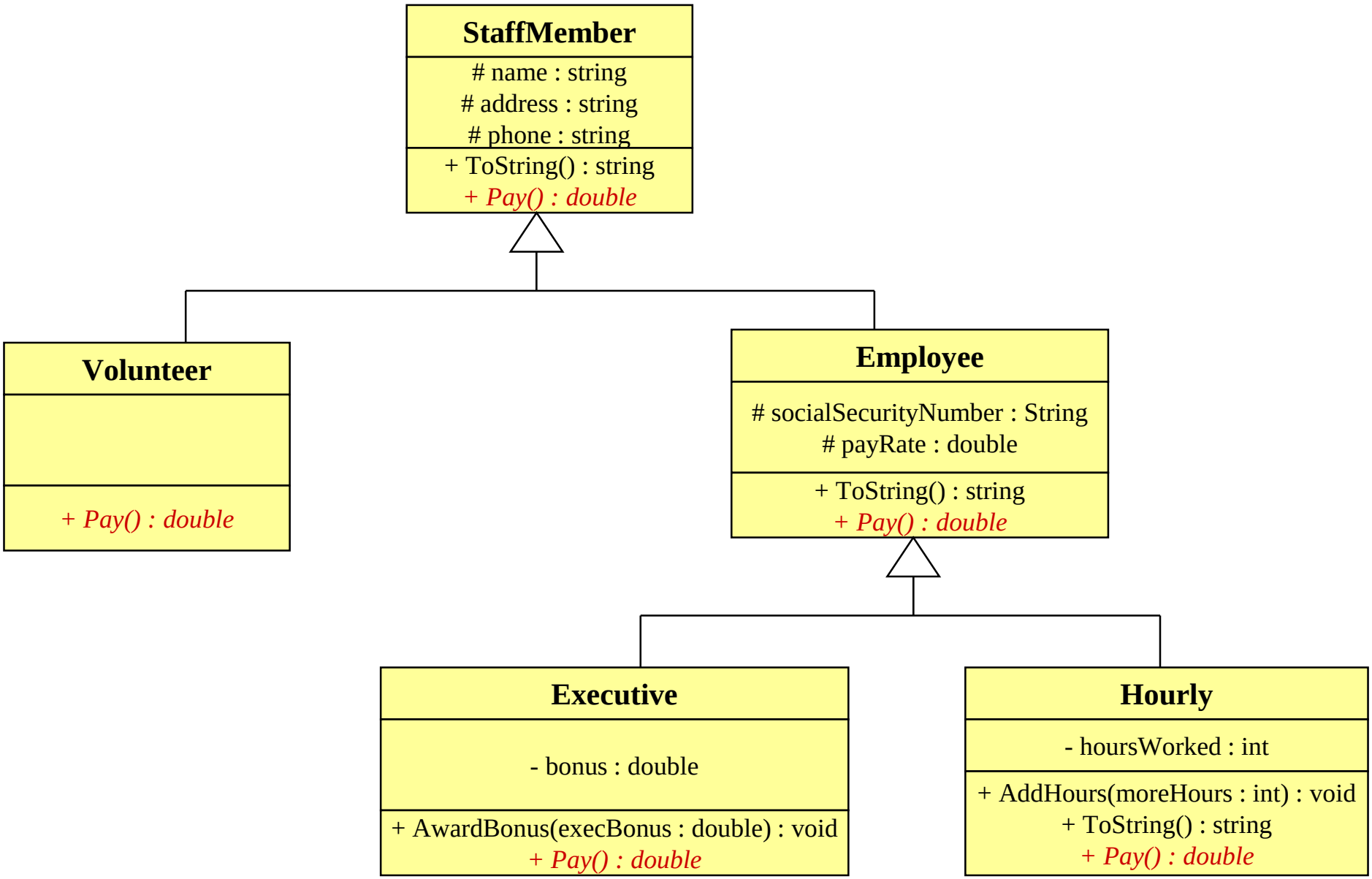
# Class Hierarchies

Murdoch UNIVERSITY



# Polymorphism via Inheritance

Murdoch UNIVERSITY



# Dynamic Binding

Murdoch UNIVERSITY

- Suppose the `Holiday` class has a method called `Celebrate`, and the `SummerHoliday` class redefines it (overrides it).
- Now consider the following invocation:

```
day.Celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `Celebrate`; if it refers to a `SummerHoliday` object, it invokes the `SummerHoliday` version

```
Holiday day;
day = new Holiday();
…
day = new SummerHoliday();
```

# Overriding Methods

Murdoch UNIVERSITY

- The keywords *virtual*, *override* and *new*
- If a base class method is going to be overridden it should be declared *virtual.*
- A derived class would then indicate that it indeed does override the method with the *override* keyword.

# Overriding Methods

Murdoch UNIVERSITY

- If a derived class wishes to hide a method in the parent class, it will use the *new* keyword.
- This should be avoided.

# Overloading vs. Overriding

Murdoch UNIVERSITY

Overloading deals with multiple methods in the same class with the same name but different signatures

Overloading lets you define a similar operation in different ways for different data

Example:

```
int foo(string[] bar);
int foo(int bar1, float a);
```

Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

Overriding lets you define a similar operation in different ways for different object types

Example:

```
class Base {
    public virtual int foo() {} }
class Derived {
    public override int foo() {}}
```

# The object Class

Murdoch UNIVERSITY

In C#, every class is derived from the root class:

System.Object

When you define a new class, if it is not an extension of an existing class, it is implicitly an extension of System.Object class.

Hence every class shares the attributes of System.Object class.

# The object Class

Class object has a number of useful methods, including

pubic virtual string ToString()

public virtual Equals(object ob)

public Type GetType()

We will explain *virtual* later in this Topic

# The `System.Object` Class

Murdoch UNIVERSITY

- All classes in C# are derived from the `Object` class
- The `Object` class - root of all class hierarchies.
- The `Object` class defines methods that will be shared by all objects in C#, e.g.,

  `ToString`: converts an object to a string representation

  `Equals`: checks if two objects are the same

  `GetType`: returns the type of object
- A class can override a method defined in Object to have a different behavior

# What are Accessible?

Murdoch UNIVERSITY

Which members of the base class are directly accessible in the derived class?

Access specifiers of those members:

- public members - accessible in the derived classes
- private members - not directly accessible in the derived classes
- protected members - directly accessible in the derived classes

# Example 2

Murdoch UNIVERSITY

```
class A
{
    public int x;
    protected int y;
    private int z;

     // default constructor
     public A() { x = 10; y = 11; z = 12; }

    public void M1()
     { Console.WriteLine("A.M1"); }
     protected void M2()
     { Console.WriteLine("A.M2"); }
     private void M3()
     { Console.WriteLine("A.M3"); }
}
```

# Example 2

```
class B : A  // (B is a derived class of base
class A)
{
  public void M()
  {
        Console.WriteLine("B.M");
        Console.WriteLine("x = {0}", x);
        Console.WriteLine("y = {0}", y);
        // Console.WriteLine("z = {0}", z);
   }
}
```

public member x is accessible in subclass

protected member y is also accessible in subclass

private member z is not accessible in subclass, so this is a mistake!

# Example 2

Murdoch UNIVERSITY

```
B b = new B();

b.x = 200;  // x is declared public in class A and is inherited
             //from A by class so it is accessible anywhere

//b.y = 300;   // y is declared protected in class A and is
      //inherited from A by class B but it is not accessible
// in any class that is not derived from class A

 b.M(); // M is declared public in class B so it is
      // accessible anywhere

 b.M1();  // M1 is declared public in class A and is inherited by
      //class B, so it is accessible anywhere

  // b.M2(); // M2 is declared protected in class A and is
// inherited by class B but it is not accessible in any class
// that is not derived from class A
```

What about b.M3()?

# Overriding Methods

Murdoch UNIVERSITY

In the previous example:

```
s.Show();
```

only prints the name and address of the Student object s.

It cannot print StudentNo

# Overriding Methods

Murdoch UNIVERSITY

Declare Show method as a virtual method in the base class.

In the derived class, the method is re-defined. The keyword "override" - in the method declaration.

In the derived class, we can use reserved word "base" to access members of the base class.

Another form of polymorphism

# Example 3

Overriding the virtual method in the derived class.

in the base class Person:

```
public virtual void Show()
{
        Console.WriteLine("Name: {0}, address: {1}",
            name, address);
 }
```

in the derived class Student:

```
public override void Show()
{
        base.Show();
// invoke the method in the base class
Console.WriteLine("Student No.: {0}", studentNo);
 }
```

# Which Constructors?

Murdoch UNIVERSITY

Which constructor - the one in the derived class, or the one in the base class, or both?

In C#, both constructors will be used.

For example, if class A is derived from System.Object, class B is derived from class A, and class C is derived class B, then when instantiating class C, the constructor from class A is executed, followed by that of class B, followed by that of class C.

# Example 4

```
class Person
{
      string myName;    // note this is private
      string myAddress;

      public Person(string name, string address) {
            // constructor
             myName=name; myAddress=address;
      }

      public virtual void Show()
      {
            Console.WriteLine("Name: {0},  address: {1}",
                              myName,  myAddress);
      }
}
```

# Example 4

Murdoch UNIVERSITY

```
class Student: Person
{
    int myStudentNo;

    public Student(string name, string address, int studentNo)
: base(name, address)  // see comments in the next slide
    {
          myStudentNo = studentNo;
    }

    public override void Show()
    {
            base.Show();
// invoke the method in the base class
          Console.WriteLine("Student No.: {0}", myStudentNo);
    }
}
```

# Example 4

Murdoch UNIVERSITY

When you instantiate class Student, such as in

```
Student s = new Student("John", "South St", 1234);
```

the matching constructor in Student is invoked.

Note that you cannot call a constructor like you call a normal method. That is why a special syntax is needed.

Even if you did not explicitly invoke the base constructor using

```
: base( . . . . )
```

in the constructor, the base constructor is still invoked first.

# Type Conversion

Murdoch UNIVERSITY

A "narrower" type can usually be automatically converted to a more general type, for example:

```
int x = 10;

double y = x;

// "int" is narrower than "double" in the sense

// every int value has a corresponding double value
```

But a more general type cannot be automatically converted to a narrower type:

```
double x = 10.5;

int y = x;

// error,  as double is more general than int
```

# Type Conversion

Murdoch UNIVERSITY

a subclass (more specific) -> ancestry classes (more general).

For example:

```
Person p = new Person("Joe", "Murdoch Dr");
Student s = new Student("Tim", "Leach Hwy", 1234);
p = s;
```

The above assignment is ok, because Student type is narrower than Person type.

Note that after the assignment, p is still of type Person, but it is pointing to an object of Student type! See the implication later.

# Type Conversion

Murdoch UNIVERSITY

The type of b is B. It can be converted to type A:

```
B b = new B();
A a = b;
// variable b's type is narrower than that of variable a
```

But not:

```
A a = new A();
B b = a;  // this is wrong!
```

# Example 5

```
class Program
{
        public static void Main(string[] args)
        {
            B b = new B();
            A a = b;
// ok as b's type B is derived from a's type A.

            a.x = 400;  // ok, as x is from class A
            a.M1();      // ok, as M1 is from class A
            // a.M();
// oops, as M is from B, it is not available in class A.
// note that a's type is now A, not B, even though
// the object it is pointing to is of type B
        }
  }
```

# Example 6

Murdoch UNIVERSITY

```
class A
{
        public virtual void M()
        {
                Console.WriteLine("I am in class A");
        }
 }


class B:  A
{
        public override void M()
        {
                Console.WriteLine("I am in class B");
        }
}
```

# Dynamic Method Dispatch

Murdoch UNIVERSITY

In the following code

```
B b = new B();
A a = b;
a.M();
```

Which method is invoked with a.M() ?
is it the virtual method defined in class A, or
is it the overriding method defined in class B?

# Example 6 (try this code)

```
   class Program
    {
            public static void Main(string[] args)
            {
                    A a = new A();
                    a.M();
// calling the virtual method M from class A

                    B b = new B();
                    b.M();
// calling the overriding method M from class B

                    a = b;
// ok, b's type is narrower than that of a
                    a.M();
// which method are we calling now, the virtual
// method from A or overriding method in B?
// perform a test to find it out yourself!
            }
     }
```

# Example 6 (Explanation)

Murdoch UNIVERSITY

If you think variable a is of type A, then you may think that a.M() should be the virtual method defined in class A.

However if you think that the object pointed to by a is actually of type B, then a.M() should invoke the overriding method defined in class B.

The compiler cannot decide at compile time which method should be invoked, the virtual method in the base class A, or the overriding method in the derived class B.

This is because at compile time, the compiler does not know the type of the object pointed to by the reference stored in variable a.

The decision is made at runtime. The runtime system resolves the method based on the type of the object, rather than the type of the object reference.

Hence, in the above case, the overriding method in class B is called. This is known as dynamic method invocation.

# Abstract Classes

Murdoch UNIVERSITY

An abstract class is incomplete and is intended to be used only as a base class.

An abstract class may contain fields and methods.

When a class is derived from an abstract class -> must include actual implementations.

An abstract class -> provides a common interface among several different classes.

# Example 7

```
abstract class A
{
        public abstract void M();  // note there
is no method body.
}


class B:  A     // B is a subclass of A
{
        public override void M() // class B must
implement this method!
        {
                // actual implementation of M
                 . . . . . .
        }
}
```

# Abstract class – contains abstract method

Murdoch UNIVERSITY

```
abstract class Motorcycle
 {
       // Anyone can call this.
       public void StartEngine()
       {/* Method statements here */ }

       // Only derived classes can call this.
       protected void AddGas(int gallons)
       { /* Method statements here */ }

       // Derived classes can override the base class implementatio
       public virtual int Drive(int miles, int speed)
       { /* Method statements here */ return 1; }

       // Derived classes must implement this.
       public abstract double GetTopSpeed();
}
```

https://msdn.microsoft.com/en-us/library/ms173114.aspx

```
class TestMotorcycle : Motorcycle
{
      public override double GetTopSpeed()
      {
            return 108.4;
      }

      static void Main()
      {
            TestMotorcycle moto = new TestMotorcycle();
            moto.StartEngine();
            moto.AddGas(15);
            moto.Drive(5, 20);
            double speed = moto.GetTopSpeed();
            Console.WriteLine("My top speed is {0}", speed);
      }
}
```

# Abstract Class Summary

Murdoch UNIVERSITY

- Defines the interface for a hierarchy of classes.
- Abstract Class lets subclasses redefine the implementation of an interface.
- This means that you cannot directly **instantiate** an abstract class.

http://www.agile-code.com/blog/difference-between-interfaces-and-abstract-classes-in-microsoft-net/

Murdoch UNIVERSITY

```
//The Human class cannot be instantiated.
public abstract class Human
{
    public abstract void Talk();
}

//while Person inherits from Human
//it has to implement the Talk method.
public class Person : Human
{
    //note the override keyword
    public override void Talk()
    {
        /* some code goes here */
    }
}
```

- A class may inherit only from **only one** abstract class
- Cannot be instantiated directly
- An abstract class may contain abstract methods and accessors.
- Methods marked as abstract cannot contain any code.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors otherwise a compiler error will occur.
- An abstract method is implicitly a virtual method.
- Abstract properties behave like abstract methods.

http://www.agile-code.com/blog/difference-between-interfaces-and-abstract-classes-in-microsoft-net/

# Interface

Murdoch UNIVERSITY

An interface defines a contract.

Unlike abstract classes, interfaces can contain methods prototypes and properties,

- no fields (which require storage),
- no method implementation.

A class can implement multiple interfaces

# Example 8

```
interface IComboBox
// by convention, we start an interface name with I

{
     void Paint();
     void SetText(string text);
     void SetItems(string[] items);
}

public class EditBox: IComboBox
{
      public void Paint() {...}
      public void SetText(string text){ ... };
      public void SetItems(string[] items){ ... };
}
```

# Interfaces - Summary

Murdoch UNIVERSITY

```
public interface IHuman
{
      void Talk();
}

public class Person : IHuman
{
    public void Talk()
    {
        throw new
NotImplementedException();
    }
}
```

- In its most common form, an interface is a group of related methods with empty bodies, in other words, interfaces describe a group of related functionalities that can belong to any class or struct.
- Some of the characteristics of an Interface:
- An interface cannot be instantiated directly.
- A class or struct can implement **more than one** interface.
- Interfaces can contain events, indexers, methods, and properties.
- An interface itself can inherit from multiple interfaces.
- The interface itself contains no implementations but only the public members signature

http://www.agile-code.com/blog/difference-between-interfaces-and-abstract-classes-in-microsoft-net/

# Interface Example - Passing Listbox

Murdoch UNIVERSITY

As an Exercise you could use a ListBox control to display all clients that are currently stored in your client database.

Assume the ListBox object is named listbox, the event handler for clicking “List All Clients” button would call a method from ClientDB object to get a list of client names for display. Let’s assume the method is named ListAllClients.

The issue then is how do we get the list of client names from the ClientDB object?

There are at least three ways to do it.

# Interface Example - Passing Listbox


Method 1:

In ListAllCLients method from ClientDB, we create a temporary array, and retrieve all client names from the database and store these names in the array. Then we return the array to the event handler.

downside: a lot of overhead associated with creating and destroying the temporary array. In addition, for each name, you need to copy it to that array and then copy it from the array to the list box for display. This is additional overhead.

# Interface Example - Passing Listbox

Murdoch UNIVERSITY

## Method 2:

From the event handler, pass the ListBox object to the method:

```
clientDB.ListAllClients ( listbox );
```

For this to happen, we can declare the method as:

```
public void ListAllClient
(System.Windows.Form.ListBox listbox)
```

Strength: avoid the overheads of Method 1.

Weakness: the method ListAllClients is limited to ListBox. It cannot be used for other list-like objects such as ListView objects. Hence we need to make the method more generic.

# Interface Example - Passing Listbox

Murdoch UNIVERSITY

Method 3:

Since what ListAllClients method really needs is the *collection* where the names can be added to. This collection is the property "Items" from a ListBox which is of type ObjectCollection. We could use this type to declare the formal parameter, however this type is defined in ListBox class and can only be used for the Items from ListBox.Items.

We noticed that ObjectCollection is actually an implementation of the interface *IList* and *IList* is fairly general and many GUI controls implement this interface. In terms of type hierarchy, *IList* is much more general and ObjectCollection is more narrow, hence the type ObjectCollection is compatible to type *IList* .

# Interface Example - Passing Listbox

Murdoch UNIVERSITY

For the above reason, we can use the third method: declare the formal parameter with *IList*:

```
public void ListAllClients
(System.Collection.IList  list)
```

and in the event handler we pass Items from the listbox to the method:

```
ListAllClients (listbox.Items);
```

The method will then retrieve all names from the database and add each of the names to the "list" using method list.Add(name).

# C# Reference

Murdoch UNIVERSITY

- **Inheritance (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173149.aspx
- **Abstract and Sealed Classes and Class Members (C# Programming Guide)**
- https://msdn.microsoft.com/en-au/library/ms173150.aspx
- **Interfaces (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173156.aspx

# C# Reference

Murdoch UNIVERSITY

- Great resource for C#
- https://msdn.microsoft.com/en-us/library/618ayhy6.aspx

- E.g. **C# Programming Guide**
- https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx

- E.g. **Interfaces (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173156.aspx

Murdoch
UNIVERSITY

- **Classes and Structs (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173109.aspx

- **Inheritance (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173149.aspx

Murdoch UNIVERSITY

- **Polymorphism (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173152.aspx

- **Abstract and Sealed Classes and Class Members (C# Programming Guide)**
- https://msdn.microsoft.com/en-au/library/ms173150.aspx

- **Properties (C# Programming Guide)**
- https://msdn.microsoft.com/en-AU/library/x9fsa0sw.aspx

# Reading/ reference

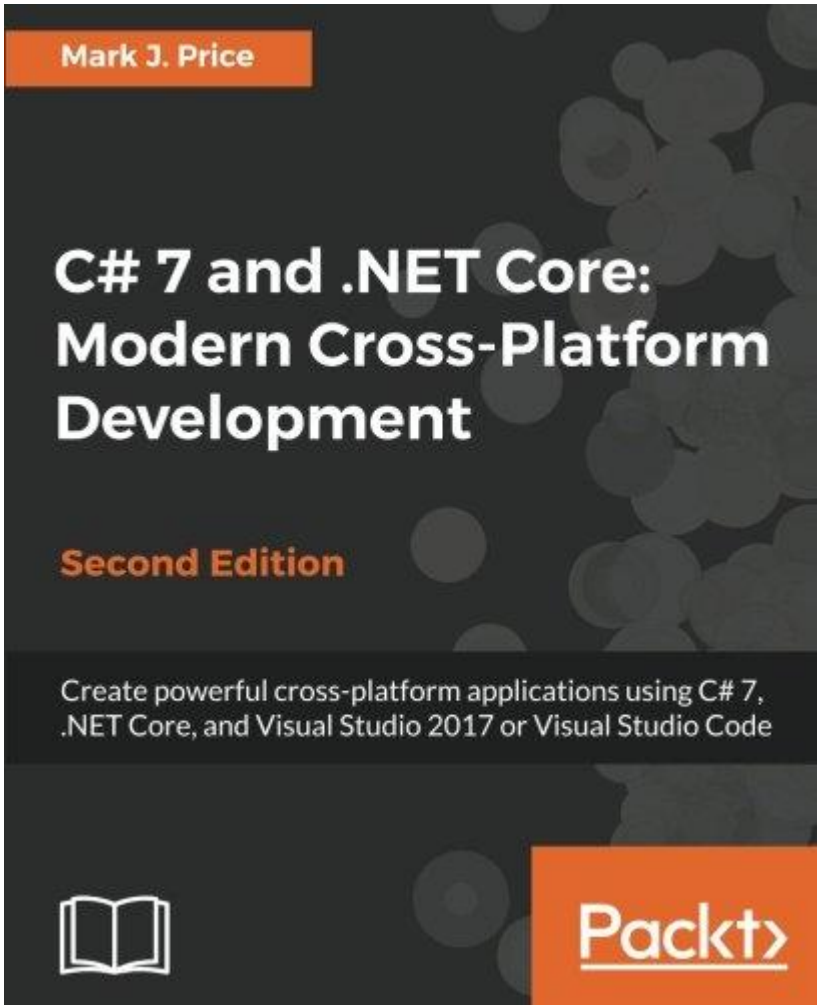
Murdoch UNIVERSITY

You should be okay with
Chapters 1-5

Specifically, review:

Chapter 6. Building Your Own Types with Object-Oriented Programming

Chapter 7. Implementing Interfaces and Inheriting Classes

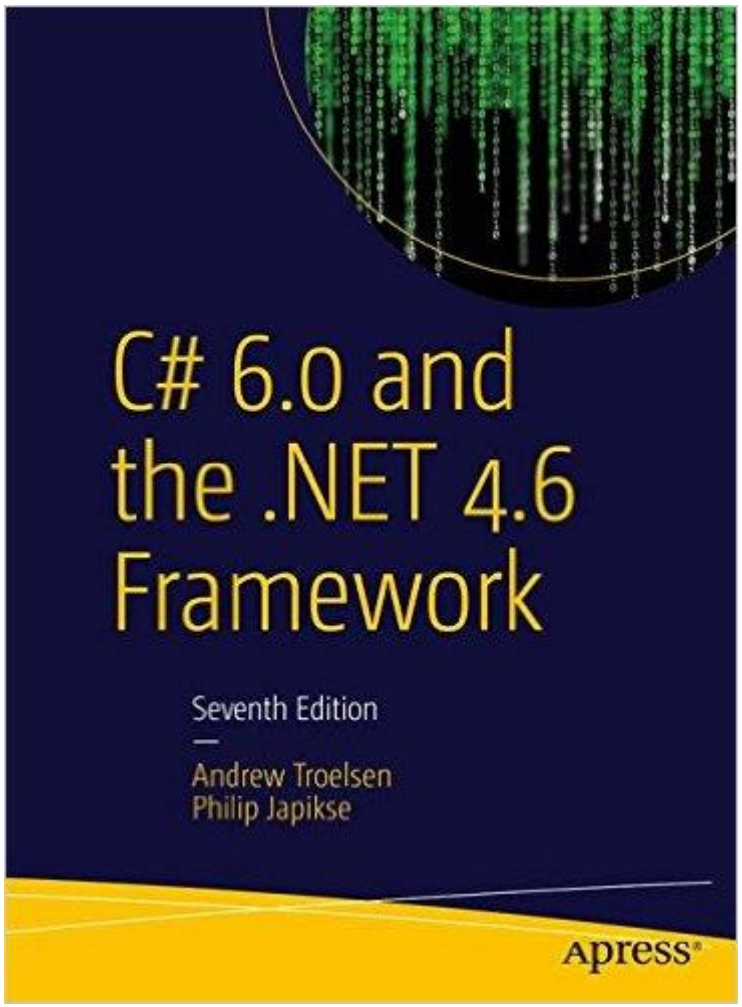


# Reading/ reference

Murdoch UNIVERSITY

Nothing this week, but take a look at the book anyway, its really good..

We will be referring to it later.



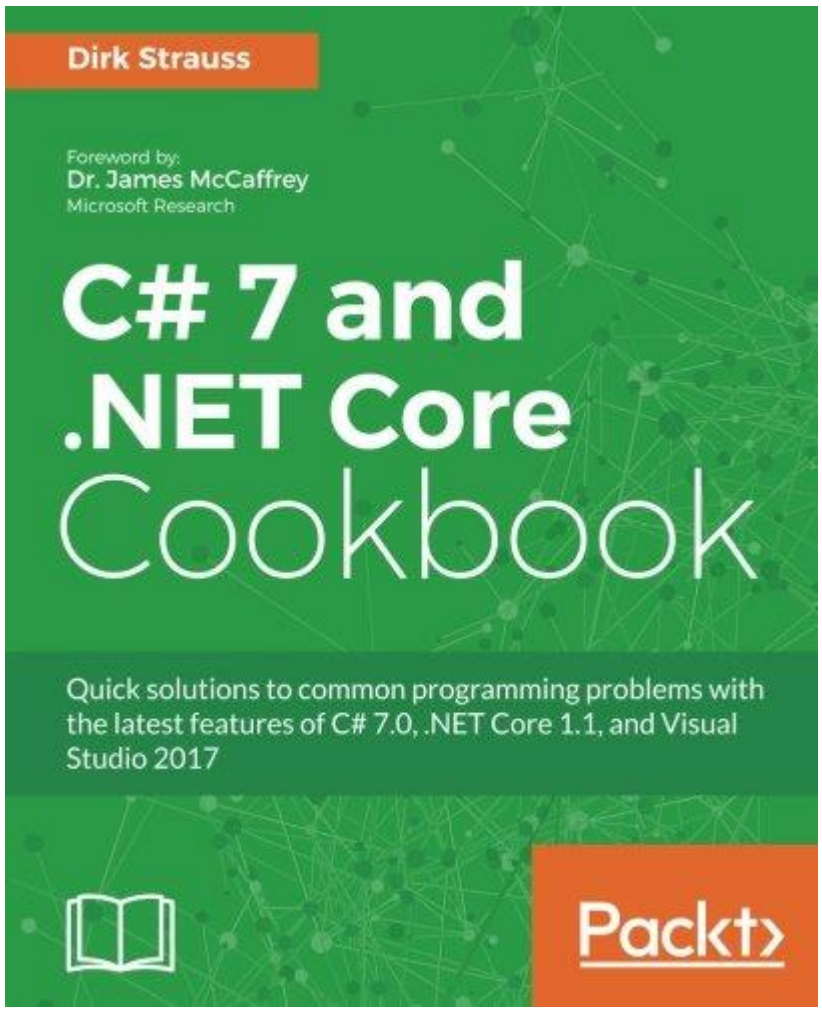
# Reading/ reference

Murdoch UNIVERSITY

You could skim through these:

Chapter: CLASSES AND GENERICS

Chapter: OBJECT-ORIENTED PROGRAMMING IN C#

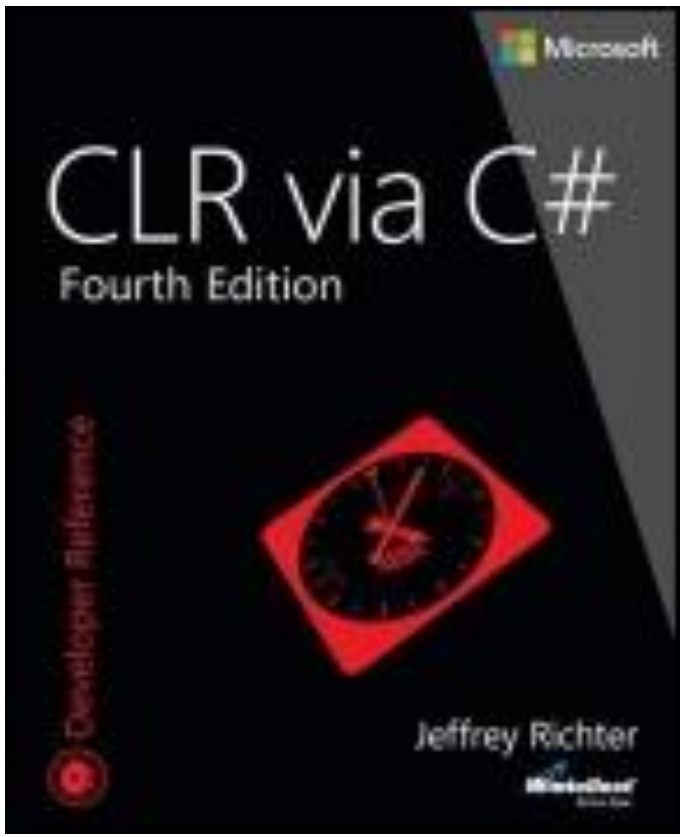


# Reading/ reference

Murdoch UNIVERSITY

We will be covering Interfaces more later on, but if you want to look at this:

Chapter 13. Interfaces



Murdoch
UNIVERSITY

# Few extra slides for you to read

# Recommendations for Abstract Classes vs. Interfaces


- The choice of whether to design your functionality as an interface or an abstract class (a **MustInherit** class in Visual Basic) can sometimes be a difficult one. An *abstract class* is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes.
- An *interface*, by contrast, is a totally abstract set of members that can be thought of as defining a contract for conduct. The implementation of an interface is left completely to the developer.
- Both interfaces and abstract classes are useful for component interaction. If a method requires an interface as an argument, then any object that implements that interface can be used in the argument.

https://msdn.microsoft.com/en-AU/library/scsyfw1d(v=vs.71).aspx

```
' Visual Basic
Public Sub Spin (ByVal widget As IWidget)
End Sub

// C#
public void Spin (IWidget widget)
{}
```

Murdoch UNIVERSITY

- This method could accept any object that implemented IWidget as the widget argument, even though the implementations of IWidget might be quite different. Abstract classes also allow for this kind of polymorphism, but with a few caveats:
- Classes may inherit from only one base class, so if you want to use abstract classes to provide polymorphism to a group of classes, they must all inherit from that class.
- Abstract classes may also provide members that have already been implemented. Therefore, you can ensure a certain amount of identical functionality with an abstract class, but cannot with an interface.
  - https://msdn.microsoft.com/en-AU/library/scsyfw1d(v=vs.71).aspx

# Interface or abstract class..

Murdoch UNIVERSITY

- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
- If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
- If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
- If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

https://msdn.microsoft.com/en-AU/library/scsyfw1d(v=vs.71).aspx

# Widening and Narrowing

Murdoch UNIVERSITY

Few more concepts

- Assigning an object to an ancestor reference is considered to be a ***widening*** conversion, and can be performed by simple assignment

```
Holiday day = new SummerHoliday();
```

- Assigning an ancestor object to a reference can also be done, but it is considered to be a ***narrowing*** conversion and must be done with a cast:

```
SummerHoliday summerDay= new SummerHoliday();
Holiday day = summerDay;
SummerHoliday summerDay = (SummerHoliday)day;
```

# Widening and Narrowing

Murdoch UNIVERSITY

- Widening conversions are most common.

  Used in polymorphism.

- Note: Do not be confused with the term widening or narrowing and memory. Many books use *short* to *long* as a widening conversion. A *long* just happens to take-up more memory in this case.

- More accurately, think in terms of sets:

  The set of animals is greater than the set of parrots.

  The set of whole numbers between 0-65535 (ushort) is greater (wider) than those from 0-255 (byte).

# Type Unification

Murdoch UNIVERSITY

- **Everything** in C# inherits from `object`

  Similar to Java except includes value types.

  Value types are still light-weight and handled specially by the CLI/CLR.

  This provides a single base type for all instances of all types.

  Called **Type Unification**