Inheritance and Abstract Classes

1

OUTLINE

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

2

2

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*

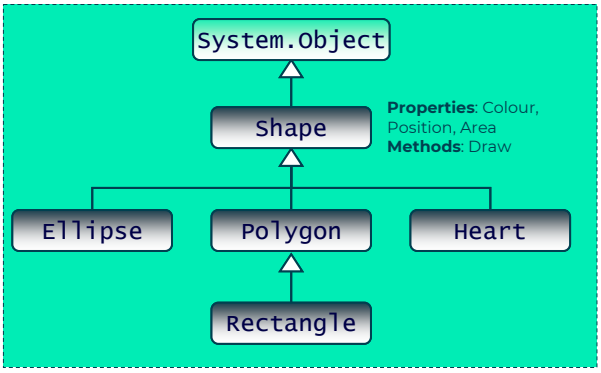
3

3

INHERITANCE EXAMPLE: GRAPHICS APPLICATION

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

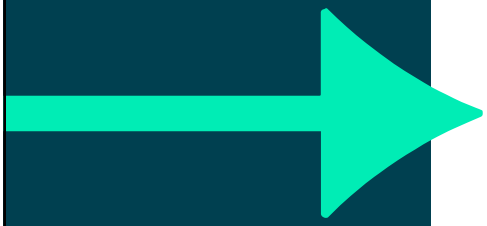
- Different shapes have common **properties**. Each shape needs to be filled with a *colour* and has a *position* and an *area*
- Different shapes have common **behaviours**. Each shape needs to be able to *draw* itself



4

4

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

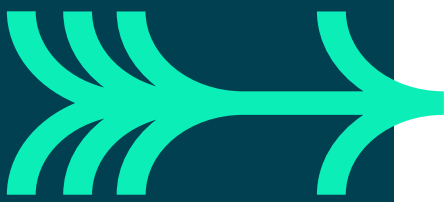
Add additional properties and methods as required

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

5

5

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
}
```

6

6

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);
```

7

7

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

8

8

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
    }
}
```

9

9

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
```

10

10

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.

11

11

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
    }
```

12

12

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

13

13

ABSTRACT CLASSES EXAMPLE

An abstract class can contain abstract members:

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

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

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

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

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

Use the override keyword to implement the member:

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

14

14

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

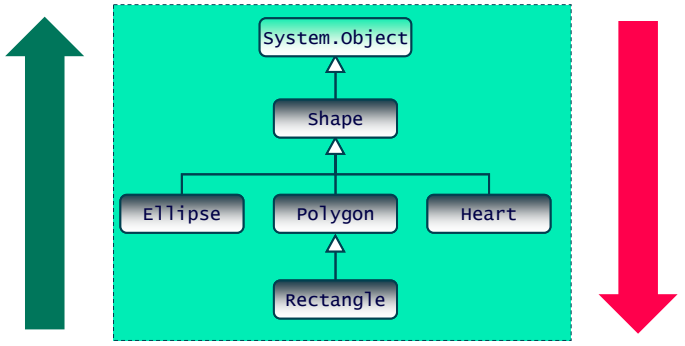
```
public abstract class Shape
{
    // concrete properties and methods
    0 references
    public Color Colour { get; set; }

    // 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
}
```

15

15

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



16

16

UP-CASTING AND DOWN-CASTING

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

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

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

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

Console.WriteLine(circumference);
```

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

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

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

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

Console.WriteLine(circumference);
```

Exception Unhandled

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

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

Exception Settings

17

17

SAFE DOWNCASTING

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

- is
- as

18

18

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);
}
```

19

19

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);
}
```

20

20

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

21

21

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}";
    }
}
```

22

22

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)
```

23

23

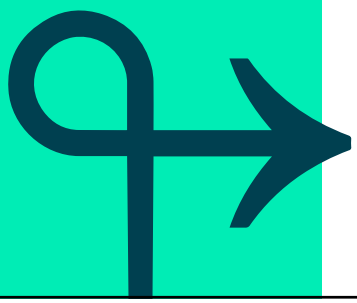
SEALED CLASSES AND MEMBERS

- 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

24

24

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+.)

25

25

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"); }
}
```

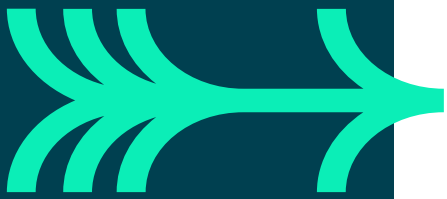
CS0239 'ClassZ.F1()': cannot override inherited member 'ClassY.F1()' because it is sealed

26

26

SUMMARY

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



27

Activity: Exercise 9

28

28