SIT771 – Lecture 7

Inheritance and polymorphism

DEAKIN UNIVERSITY

Further reading

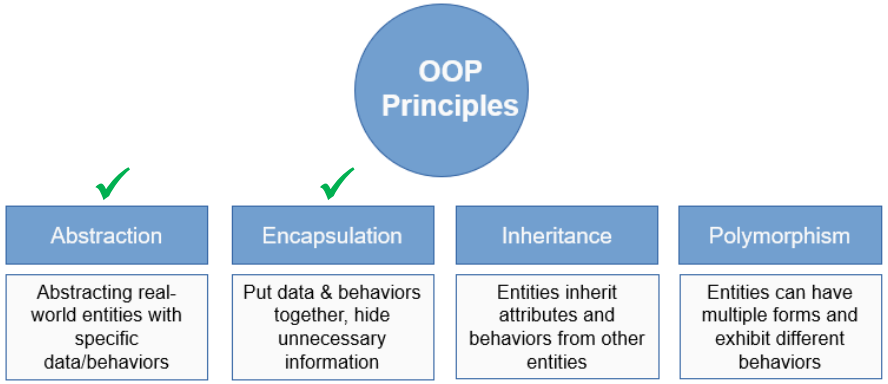
- Paul Deitel and Harvey Deitel (2018). Visual C# how to Program (6th ed). Pearson. Ebook on Deakin Library – Chapter 11 and Chapter 12.

Outline

In this lecture...

- Inheritance (OOP concept 3/4)
- More abstract concepts in OOP
  - Abstract classes (≠ Abstraction)
  - Abstract methods (≠ Abstraction)
  - Access modifiers
- Polymorphism (OOP concept 4/4)
  - Compile-time
  - Run-time
  - Binding types (late binding vs. early binding)
  - Casting

OOP CONCEPT 3/4 … inheritance



The why

Motivation

- Suppose that we want to implement some shapes (e.g., Circle and Square) for a mathematical program. Each shape has a name, color, and a few methods that calculate the geometric features of it...

```
143  public class Circle
144  {
145      private string _name;
146      private Color _color;
147      private double _radius;
148
149      public string Name...
153
154      public Color Color...
158
159      public Circle(string name, Color color, double radius)
160      {
161          this._name = name;
162          this._color = color;
163          this._radius = radius;
164      }
165
166      public double Area()
167      { return Math.PI * Math.Pow(this._radius, 2); }
168
169      public double Perimeter()
170      { return 2 * Math.PI * this._radius; }
171  }
```

Can we remove duplications? What if another shape was added with the same properties?

```
173  public class Square
174  {
175      private string _name;
176      private Color _color;
177      private double _side;
178
179      public string Name...
183
184      public Color Color...
188
189      public Square(string name, Color color, double side)
190      {
191          this._name = name;
192          this._color = color;
193          this._side = side;
194      }
195
196      public double Area()
197      { return Math.Pow(this._side, 2); }
198
199      public double Perimeter()
200      { return this._side * 4; }
201  }
```

What if the developer forgot or decided to ignore the implementation of Perimeter for Square?

The what

Definition

- **Inheritance** is an OOP principle that allows for the extension of existing abstractions through creation of new classes (called *subclasses* or *derived classes*) based on existing ones (called *superclasses* or *base classes*).
- The extension of abstraction occurs through generalization or specialization. Related classes are in the form of parent and child classes, where...
  - Child classes inherit public and protected fields, properties, and methods from their parent.
  - Child classes can <u>change the behavior</u> of inherited methods if needed (a.k.a. overriding, TBD).
  - Child classes can have their own <u>specific properties and methods</u>.

- Are these other <u>inheritance</u> examples…?
  - Car and Vehicle?? Yes, Car is a special type of Vehicle.
  - Car and Wheel?? No! The relation between Car and Wheel is a **<u>composition</u>**.

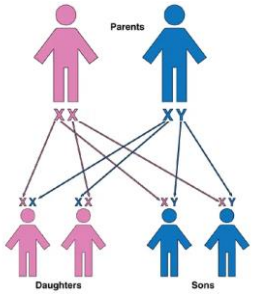


Image taken from www.edurev.in

Image created by https://openart.ai/

The how

Implementation in C#

- The general form:

  [access_modifier] **class** derived_class_name : base_class_name { }

- The mathematical shapes program can now be changed as below…

```
public abstract class Shape
{
    private string _name;
    private Color _color;

    public string Name...

    public Color Color...

    public Shape(string name, Color color)
    {
        this._name = name;
        this._color = color;
    }

    public abstract double Area();

    public abstract double Perimeter();
}
```

```
public class Circle: Shape
{
    private double _radius;

    public Circle(string name, Color color,
        double radius): base(name, color)
    {
        this._radius = radius;
    }

    public override double Area()
    { return Math.PI * Math.Pow(this._radius,

    public override double Perimeter()
    { return 2 * Math.PI * this._radius; }
}
```

Circle is a child class of the superclass **Shape**.

Circle and **Square** do not have to have the same repetitious class members, they are **inherited** from **Shape**.

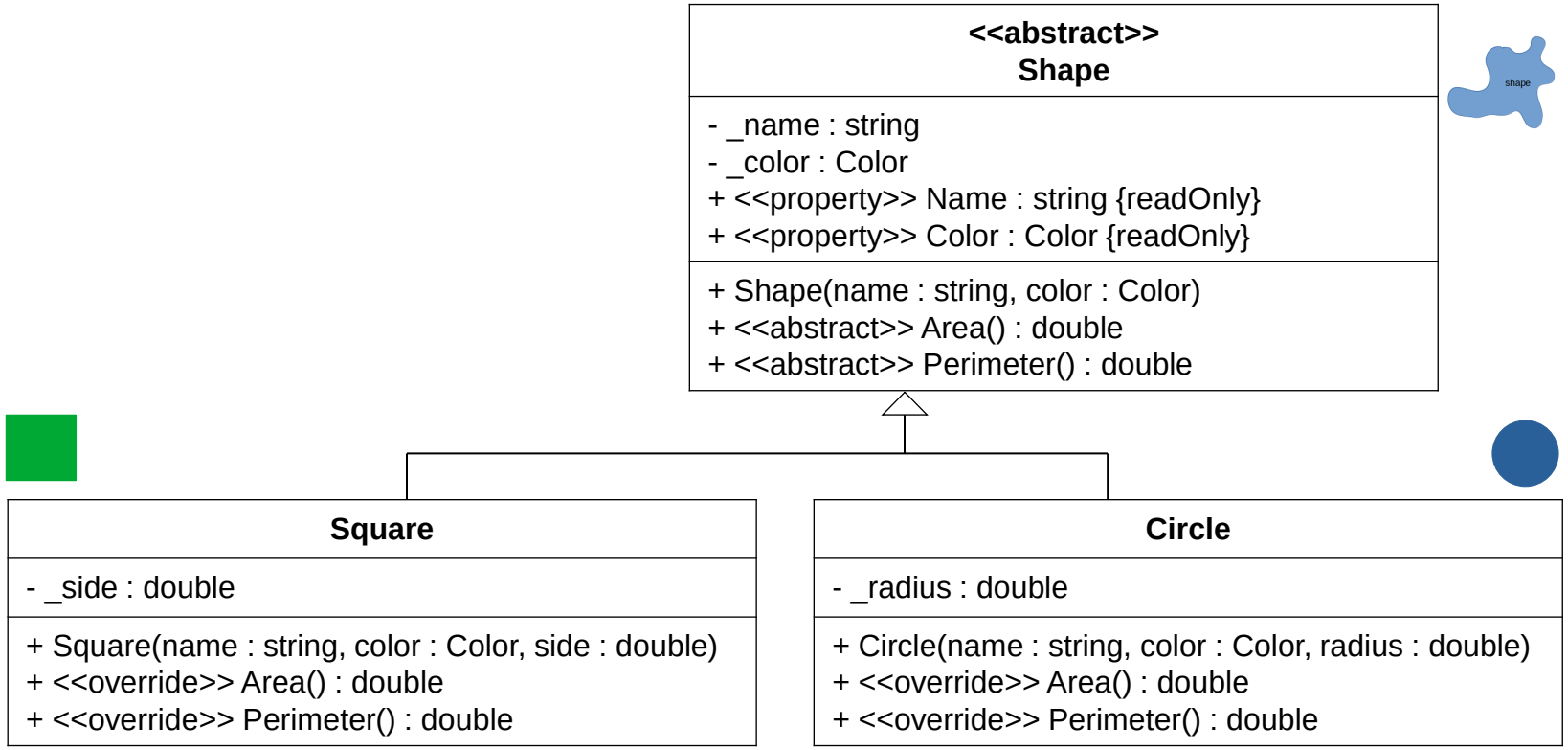
```
public class Square: Shape
{
    private double _side;

    public Square(string name, Color color,
        double side): base(name, color)
    {
        this._side = side;
    }

    public override double Area()
    { return Math.Pow(this._side, 2); }

    public override double Perimeter()
    { return this._side * 4; }
}
```

???

The how

Implementation in UML

- Inheritance is represented using a **hollow triangle** in the UML class diagrams. The triangle points from the subclass (child) to the superclass (parent).



MORE ABSTRACT CONCEPTS IN OOP

Abstract classes

Generic and contract-enforcing

- It is common for OOP developers to have classes that represent generic concepts rather than specific objects...
  - These are known as **abstract** classes.
  - In C#, add the keyword **abstract** to the class declaration.
  - No specific objects of an abstract class can be constructed or instantiated!
  - Abstract classes and Abstraction are **NOT** the same concepts.
  - Abstract classes provide a place for contract-enforcing methods (see the next slide)…

```
84   public abstract class Thing
85   {
86       public string Name;
87       public int Id;
88   }
89
90   public class Computer : Thing
91   {
92   }
93
94   public class UsingAbstractClasses
95   {
96       public static void Process()
97       {
98           Thing comp  = new Computer();
99           comp.Name = "My first computer";
100          comp.Id = 1;
101
102          Thing thing = new Thing();
103      }
104  }
```

Note the use of the **abstract** keyword that indicates that this is an abstract class.

0 % 1 0

ror List

Entire Solution | 1 Error | 0 Warnings | 0 of 1 Message

Code	Description
CS0144	Cannot create an instance of the abstract type or interface 'Thing'

Abstract methods

Contract-enforcing placeholders

- Sometimes, you need/predict some behaviors for child classes where the parent class does not have enough information or specificity to implement it. In such cases, ...
  - Abstract methods can be created <u>only</u> within <u>abstract parent classes</u> using the same keyword **abstract**.
  - Abstract methods will act as a placeholder, a method that the abstract parent class will not implement but the child class **must!**

```
264  public class Circle: Shape
265  {
266      private double _radius;
267
268      public Circle(string name, Color color,
269          double radius): base(name, color)
270      {
271          this._radius = radius;
272      }
273
274      public override double Area()
275      { return Math.PI * Math.Pow(this._radius, 2); }
276  }
```

Note the modifiers **abstract** and **override** in the parent and child classes, respectively.

```
238  public abstract class Shape
239  {
240      private string _name;
241      private Color _color;
242
243      public string Name...
247
248      public Color Color...
252
253      public Shape(string name, Color color)
254      {
255          this._name = name;
256          this._color = color;
257      }
258
259      public abstract double Area();
260
261      public abstract double Perimeter();
262  }
```

100 % 1 0

Error List

Entire Solution | 1 Error | 0 Warnings | 0 of 26 Messages | Build + IntelliSense

Code	Description
CS0534	'Circle' does not implement inherited abstract member 'Shape.Perimeter()'

Access modifiers

C# version 9.0

- Seven different modifiers...
  - **public**: Accessible from any code and is used for members that need to be exposed to external code.
  - **private**: Accessible only within the declaring scope (e.g., class) and is used for information hiding.
  - **protected**: Accessible within the declaring class and its derived subclasses and is used for code reuse and polymorphism.
  - **internal:** Accessible within the same assembly (e.g., exe/dll) and is used for code sharing within a project.
  - **protected internal:** Accessible within the same assembly or within derived classes in any other assembly.
  - **private protected:** Accessible within the declaring class and derived classes in the same assembly only.
  - **file:** Accessible within the same source file only.

Image source: www.learn.microsoft.com

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`	`file`
Within the file	✔	✔	✔	✔	✔	✔	✔
Within the class	✔	✔	✔	✔	✔	✔	✘
Derived class (same assembly)	✔	✔	✔	✔	✔	✘	✘
Non-derived class (same assembly)	✔	✔	✘	✔	✘	✘	✘
Derived class (different assembly)	✔	✔	✔	✘	✘	✘	✘
Non-derived class (different assembly)	✔	✘	✘	✘	✘	✘	✘

- Default access modifiers...
  - **Classes and structs:** internal by default
  - **Class members:** private by default
  - **Enums:** public by default
  - **Interfaces:** public by default
  - **Delegates:** internal by default

Quiz

Consider a scenario where…

- You are working on a C# application that involves animals in a zoo. You have the following class hierarchy:
  - Base Class: Animal

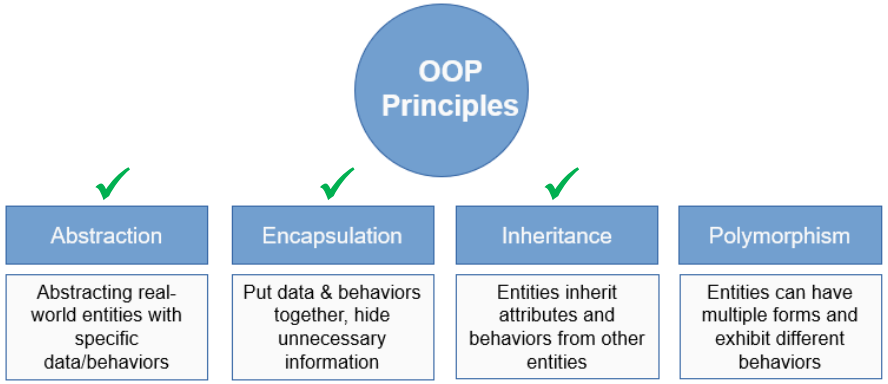
    Method: **public abstract void** Speak()

    Method: **public virtual void** Sleep()

    Constructor: **public** Animal(**string** name) – Initializes the animal's name.
  - Derived Class: Dog (inherits from Animal)

    Method: **public override void** Speak() – Provides a specific implementation for dogs.
- Given this scenario, consider the following statements and choose the one/s that is/are correct:
  - ✓ A) You can call the Sleep method on an instance of Dog through a variable of type Animal.
  - B) The Speak method in the Dog class does not need to use the override keyword because it is implementing a method from the Animal class.
  - C) You can create an instance of the Animal class directly even though it has an abstract method.
  - D) If you do not provide a constructor in the Dog class, the Animal class constructor will not be called automatically.

OOP CONCEPT 4/4 … polymorphism



Introduction

Polymorphism means...

- Things of different types can be accessed through the same interface. This allows, for instance, objects of different types to be treated as if they were of the same type. Things can be…
  - Objects or instances of classes
  - Methods
  - Operators

Polymorphism (with objects and at run-time) is only meaningful where there is an **inheritance** relationship among classes.

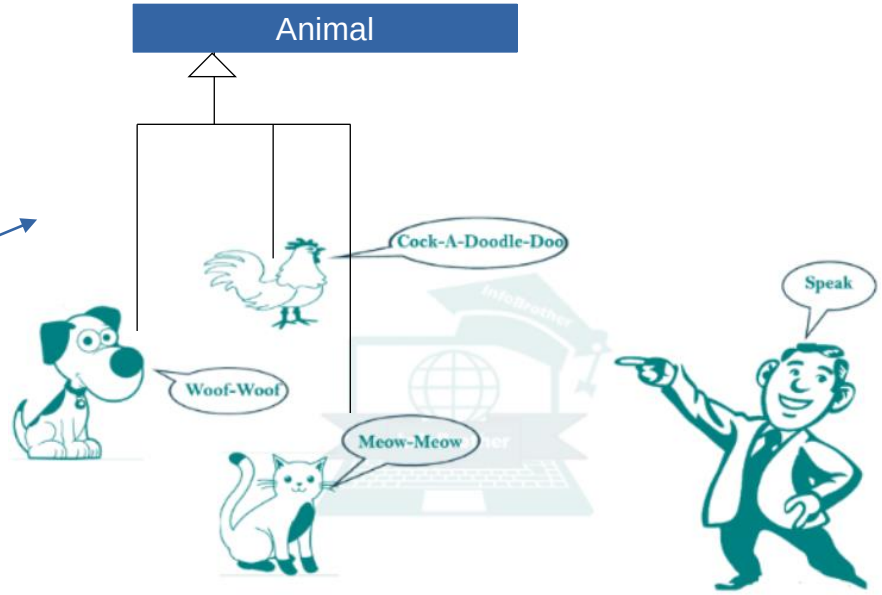


Image source: www.csharp-station.com

Compile-time polymorphism

Static

- At this stage, the program is only being compiled (not running)
  - Methods can have the same name, with different arguments, in the same class. This is also known as **method overloading**.
  - The specific method version is called and used based on the number of input arguments.
  - **Quiz:** Does this apply every time you create a class constructor with some input arguments? 

    Yes! Every time you create such a constructor (in addition to and keeping the default constructor), there will be at least two versions of the class constructor:

    - Version 1: The constructor with no arguments (default constructor)
    - Version 2: The constructor you created with arguments

```
public class Calculator
{
    public static int AddInts(int a, int b)
    {
        return a + b;
    }

    public static int AddInts(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

Run-time polymorphism

Dynamic

- At this stage, the program has been compiled before and is now running
  - A methods in a child class can take the same name and arguments as the same method in its parent class, however with a different implementation or behavior. This is also known as **method overriding**. This can only be done on <u>non-static</u> methods.

```
9   public class Calculator
10  {
11      public virtual int AddInts(int a, int b)
12      {
13          return a + b;
14      }
15
16      public virtual int AddInts(int a, int b, int c)
17      {
18          return a + b + c;
19      }
20  }
21
22  public class MyCalculator : Calculator
23  {
24      public override int AddInts(int a, int b)
25      {
26          Console.Write(a + "+" + b + "=");
27          return a + b + 1;
28      }
29  }
30 }
```

Note the use of these modifiers...

```
9   class main
10  {
11      static void Main(string[] args)
12      {
13          MyCalculator mc = new MyCalculator();
14          Console.WriteLine(mc.AddInts(2, 2) + " !");
15          Console.ReadLine();
16      }
17  }
18 }
19
20
21
22
23
```

C:\Users\bahadorreza\source\repos\SIT771\SIT771\bin\Debug\SIT771.exe

```
2+2=5 !
```

Run-time polymorphism

Dynamic (cont.)

- If you need to access the method from the parent class in C#...
  - Use the keyword **base** to get a handle to the method in the parent class
  - This is useful when you need to **extend** the behavior of the parent method

```
public class Rectangle : Shape
{
    public override void Draw(System.Drawing.Graphics graphics)
    {
        base.Draw(graphics);
        //calls the Draw() method from Shape

        //...
        //do more here...
    }
}
```

base.Draw will call the Draw() method of Shape as the parent class.

Run-time polymorphism

Dynamic (cont.)

- If you need to access the constructor from the parent class in C#...
  - Use the keyword **base** to get a handle to the constructor in the parent class

```
public class Shape
{
    public int X { get; set; }
    public int Y { get; set; }

    public Shape(int _x, int _y)
    {
        X = _x;
        Y = _y;
    }
}
```

```
public class Rectangle : Shape
{
    public Rectangle() : base(15, 25)
    {
    }
}

public class Ellipse : Shape
{
    public Ellipse(int _x, int _y) : base(_x, _y)
    {
    }
}
```

Further code and instructions can be added here...

This will call the constructor of Shape as the parent class.

Binding types

Early binding

- The compiler is in charge
  - The **compiler** knows exactly what the methods and the types of all variables are before any assignment of values and before the program runs.
  - The object types, methods, and properties are all checked during **compile-time** and there will be compile errors for any incorrect code, e.g., int x = 2.1.
  - The performance is <u>higher</u> at run-time.

The compiler knows all the types and available methods/arguments when the code is being developed and compiled.

```
public class Cylinder_v2
{
    public Circle_v2 Top { get; private set; }
    public Circle_v2 Bottom { get; private set; }
    public double Height { get; set; }

    public Cylinder_v2(double radius, double height)
    {
        Top = new Circle_v2(radius);
        Bottom = new Circle_v2(radius);
        Height = height;
    }

    public double GetArea()
    {
        return 2 * Top.GetArea()
            + Top.GetPerimeter() * Height;
    }

    public double GetVolume()
    {
        return Top.GetArea() * Height;
    }
}
```

Binding types

Late binding

- The runtime (common language runtime in .Net) is in charge
  - The compiler will not know the type of objects or properties or the arguments of methods.
  - The types and argument sets are all decided...
    - when the program runs.
    - based on the right-hand-side assignments.
  - The performance is lower at run-time.

dynamic or **object** keywords can be used to define a general object. The class **object** is the parent of all classes in C#.

```
16          dynamic object_1 = 1.23;
17          dynamic object_2 = "coding";
18          dynamic object_3 = 250;
19
20          Console.WriteLine(object_1 + " \t type = " + object_1.GetType());
21          Console.WriteLine(object_2 + " \t type = " + object_2.GetType());
22          Console.WriteLine(object_3 + " \t type = " + object_3.GetType());
23          Console.ReadLine();
24      }
25  }
26  }
27
28
29
30
```

C:\Users\bahadorreza\source\repos\SIT771\SIT771\bin\Debug\SIT771.exe

```
1.23      type = System.Double
coding    type = System.String
250       type = System.Int32
```

Binding types

Late binding (cont.)

- When there is inheritance between classes…
  - A **parent** object (instance) can be referenced by **its own** class reference (variable type).
  - A **child** object (instance) can be referenced by **its own** class reference (variable type).
  - A **child** object (instance) can be referenced by its **parent** class reference (variable type).

```
9   public class Calculator
10  {
11      public virtual int AddInts(int a, int b)
12      {
13          return a + b;
14      }
15
16      public virtual int AddInts(int a, int b, int c)...
20  }
21
22  public class MyCalculator : Calculator
23  {
24      public override int AddInts(int a, int b)
25      {
26          //Console.Write(a + "+" + b + "=");
27          return a + b + 1;
28      }
29  }
```

```
24          Calculator calc_1 = new Calculator();
25          Console.WriteLine("parent referenced as parent, sum = " + calc_1.AddInts(1, 2));
26
27          MyCalculator calc_2 = new MyCalculator();
28          Console.WriteLine("child referenced as child, sum = " + calc_2.AddInts(1, 2));
29
30          Calculator calc_3 = new MyCalculator();
31          Console.WriteLine("child referenced as parent, sum = " + calc_3.AddInts(1, 2));
32
33          Console.ReadLine();
34      }
35  }
36  }
37
38
```

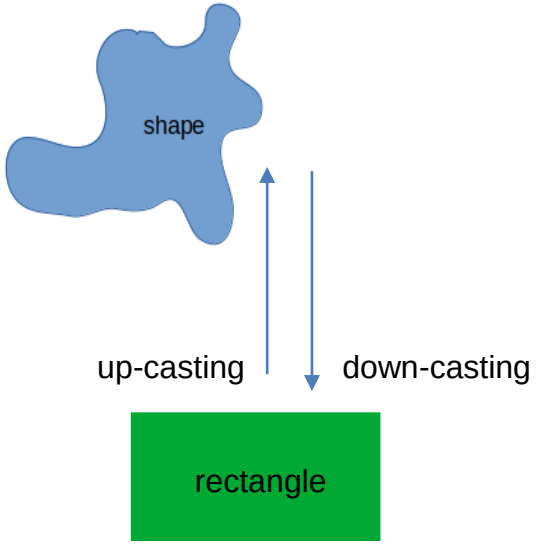
C:\Users\bahadorreza\source\repos\SIT771\SIT771\bin\Debug\SIT771.exe

```
parent referenced as parent, sum = 3
child referenced as child, sum = 4
child referenced as parent, sum = 4
```

Casting

A way for type forcing/changing

- When there is inheritance, it is possible to use an object and get to a…
  - more specific type (**down-casting**)
  - more general type (**up-casting**)
- Can be done using the keyword **as** or using **()**




```
31  public class Shape
32  {
33      public int X { get; set; }
34      public int Y { get; set; }
35
36      public Shape(int _x, int _y)...
41  }
42
43  public class Rectangle : Shape
44  {
45      public string ExtraProperty;
46      public Rectangle(int _x, int _y) ...
49  }
50
51  public class Ellipse : Shape
52  {
53      public string Color;
54      public Ellipse(int _x, int _y) ...
57  }
```

```
58
59  public class UsingShapes
60  {
61      public static void ProcessShapes()
62      {
63          Shape s1;
64          Shape s2;
65          Shape s3;
66
67          //down-casting s1 to Ellipse
68          s1 = new Ellipse(0, 0);
69
70          //setting s2 to a shape object
71          s2 = new Shape(0, 0);
72
73          //down-casting s3 to Rectangle
74          s3 = new Rectangle(0, 0);
75
76          if (s3 is Rectangle)
77              (s3 as Rectangle).ExtraProperty = "some property";
78
79          Console.WriteLine(((Ellipse)s1).Color);
80          Console.WriteLine(s1.Color);
81      }
82  }
```

Quiz

Consider a scenario where…

- A gaming system needs to handle different types of characters (warrior, mage, archer) with unique abilities and attributes. The game should be able to apply generic actions like attack, defend, and heal to any character without knowing the specific character type beforehand. The system aims for optimal flexibility. Given this scenario, which polymorphism approach, considering factors like compile-time efficiency, runtime flexibility, and code maintainability, is most suitable?
  - A) Primarily static polymorphism for performance-critical core gameplay mechanics, complemented by dynamic polymorphism for character-specific behaviours.
  - B) Exclusive use of dynamic polymorphism for maximum flexibility, even at the potential cost of some performance overhead.
  - C) A balanced approach combining static and dynamic polymorphism, leveraging the strengths of each for different aspects of character handling.
  - ✓ D) A hybrid approach utilizing abstract classes for defining character contracts, while employing dynamic polymorphism for behaviour implementation.

Epilogue

A GOOD PROGRAMMER IS SOMEONE WHO LOOKS BOTH WAYS BEFORE CROSSING A ONE-WAY STREET...

DOUG LINDER