UCL DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

CLASS ENCAPSULATION & INHERITANCE

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ucl.ac.uk

Referring to the Current Object’s Members with the *this* Reference

- In every object’s methods, the **this** reference can be used to refer to its members
  - This can be used when a method parameter has the same name with a member, although this can be avoided by using different names
  - public Complex (double real, double imag)
    { this.real = real; this.imag = imag }
- Another more important (and necessary) use of the this reference is when overloaded constructors call each other – they **cannot** call each other by name
  - public Time (int h, int m, int s)
    { hour = h; min = m; sec = s; }
  - public Time ()
    { this(0, 0, 0); }

Default Constructor

- Constructors initialise an object’s members and every class needs to have a constructor
  - In a subclass through inheritance, a constructor should also call the superclass constructor
- If the programmer has not provided a constructor, the Java compiler produces implicitly a default one with no parameters (but NOT the C++ one!)
  - This initialises instance variables with no initial values to their default values i.e. 0 for primitive numeric types, false for boolean and null for object references
  - It also calls the superclass default constructor
- It is good programming practice though to include explicitly a default constructor

final Variables

- A variable with a “constant” value that should not be changed can be declared inJava as final
  - The compiler will not allow any changes to it
  - In Java: final int bufferSize = 4096;
  - In C/C++: const int bufferSize = 4096;
- Final instance variables are initialised when they are declared as above

Public, Private and Protected Class Members

- Instance variables and methods can be public, private or protected
  - Public members can be accessed from outside the class, i.e. by objects of other classes that have a reference to an object of this class
  - Private members can only be accessed from methods of this class
  - Protected members can only be accessed from methods of this class or of any derived classes through inheritance

Get and Set Methods

- Private or protected instance variables that need to be accessed externally typically have associated *<getVar>* and *<setVar>* methods
  - But is this not the same as making them public? **Well, no!**
- Using get and set methods controls how these variables are accessed
  - For example, the time of the day may be held in seconds, with a getHour method calculating the current hour and returning it
  - Also if the time is held as hour, min, sec, setting any of these values could perform checks to avoid incorrect settings i.e. hour < 24 and min, sec < 60
- Get and Set methods support data encapsulation and represent good software engineering practice

Example: a Time Class

```
public class Time {
    private int hour, min, sec;

    // constructors
    public Time () { this(0, 0, 0); }
    public Time (int h, int m, int s) { setTime(h, m, s); }
    public Time (int s) { setTimeInSecs(s);

    // get methods
    public int getHour () { return hour; }
    public int getMin () { return min; }
    public int getSec () { return sec; }
    public int getTimeInSecs () { return 3600*hour+60*min+sec; } // time in secs

    // set methods that do validation
    public  void setTime (int h, int m, int s) { setHour(h); setMin(m); setSec(s); }
    public  void setTimeInSecs (int s) {  /* implementation not provided here */ }
    private void setHour (int h) { hour = (h >= 0 && h < 24) ? h : 0; }
    private void setMin (int m) { min = ( m >= 0 && m < 60) ? m : 0; }
    private void setSec (int s) { sec = (s >= 0 && s < 60) ? s : 0; }

    public String toString () { return hour + “:” + min + “:” + sec; }
}
```

Data Encapsulation and Abstraction

- Instance variables are typically private or protected, hiding internal implementation details
  - This is called data encapsulation or information hiding
- In fact, a class should provide an abstract view of a "service" through its public methods
- For example, a stack object stores objects of other types that can be added ("pushed") and removed ("popped") only from the top - like a stack of dishes
  - A stack can use internally an array or a linked list in two different ways for its implementation (we will see this later), but its clients do not know this, they just use its methods push and pop
  - A stack can change its internal implementation (and get recompiled) as far as the public method interface remains the same and the rest of the system will not be affected
  - **Good software engineering practice!**

Inheritance

- We can define a class that extends an existing one, a subclass that inherits from a superclass
  - In principle a new class can inherit from multiple superclasses in object-oriented software engineering
  - But we can only inherit from a single superclass in Java i.e. we have single as opposed to multiple-inheritance
- Inheritance allows software reuse as the subclass inherits the instance variables and methods of the superclass
  - Some of the inherited methods can be redefined to reflect the properties and behaviour of the derived class
- With inheritance we achieve both software reusability but also extensibility

Inheritance (cont'd)

- Inheritance hierarchies can have many levels, the top of the hierarchy being always the Java class Object
  - Every class that we define without inheritance inherits from Object, with the compiler setting implicitly the superclass to Object
  - Object keeps meta-data information such as the actual class etc., this is 8 bytes in 32-bit and 16 bytes in 64-bit machine architectures
  - A key method supported by Object is toString which subclasses typically redefine (override) to return the stringified object's content
- Inheritance allows us to deal with common software aspects by defining generic superclasses from which we derive more specific subclasses
  - This increases **abstraction**, i.e. we focus first on general properties rather than specific details
  - It also achieves **software reusability** and **extensibility**
- Another way to achieve reusability is through composition: a class may contain other classes as instance variables

Inheritance and Composition

- Inheritance represents an "is-a" relationship
  - A subclass "is-a" (i.e. behaves as an) object of its superclass(es) e.g. a cylinder "is-a" circle (see later)
- Composition represents a "has-a" relationship
  - An object may contain as instance variable an object of another class e.g. a Stack "has-a" List object (see later)
- Inheritance achieves reusability and extensibility while composition achieves only reusability. They both promote increased software abstraction
  - We will see a relevant comparative example later when we will examine the data structure Stack

Abstract Classes

- Sometimes we define superclasses that will never be instantiated e.g. a class Shape
  - These are effectively incomplete classes that provide methods that have to be redefined in subclasses
- Some of the methods of an abstract class do not have an implementation but only provide a parameter interface
  - These should be declared as abstract
  - If at least one method is abstract, the class should also be declared abstract
- For, example a class Shape:
  - public abstract class Shape {

```
        // …
        public abstract String getName (); // no impl.{ …}
}
```

Abstract Classes and Interfaces

- An abstract class typically provides an implementation for some methods and just a parameter interface for some others (its abstract methods)
- There may exist classes that are completely abstract in the sense that ALL their methods are abstract
- In this case, instead of a class we can declare an interface
  - public interface Shape {

    ```
        // …
        // no implementation for any method { … }
        public String getName ();
    }
    ```

Interfaces

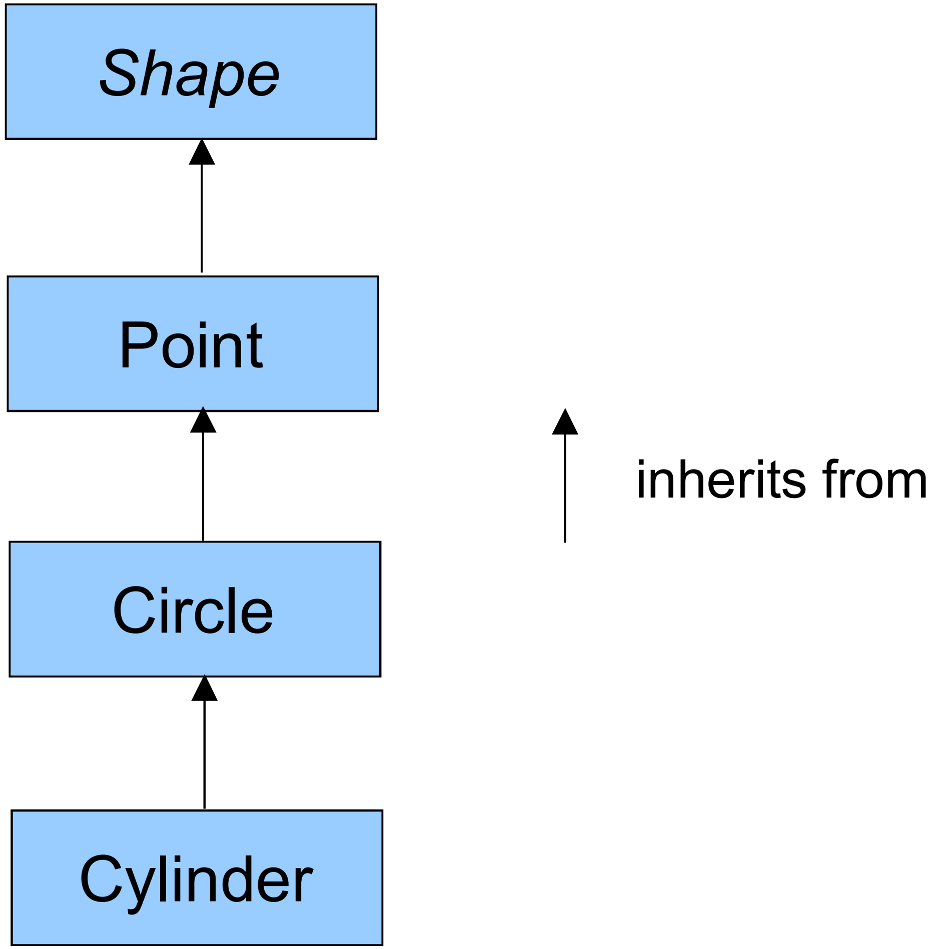
- By inheriting from a class we inherit the superclass methods implementation
- By inheriting from an interface we only inherit …an interface i.e. abstract methods or "a way of doing things"
- The key point though behind introducing interfaces is that a new class can inherit (or implement) multiple interfaces in addition to extending a single superclass
  - This effectively circumvents the lack of multiple inheritance in Java in an elegant way (C++ supports multiple inheritance) although with interfaces one inherits only "behaviour" and not code
- E.g. class Point inherits from Object and Shape, although inheriting from Object is superfluous (i.e. default case)
  - public class Point extends Object implements Shape {

```
        // …
        public String getName () { return "Point"; }
}
```

An Example Inheritance Hierarchy

- We will consider the following class hierarchy to demonstrate the use and benefits of inheritance
  - Shape (abstract class), Point, Circle, Cylinder
- Point, Circle and Cylinder are all shapes, Circle extends Point and Cylinder extends Circle
  - This is "structural inheritance" in the sense that the subclasses inherit the data and methods of the superclasses but "is-a" relationships are unnatural
  - For example, saying that a circle "is-a" point only holds for the degenerate case of a zero radius
  - Also saying that a cylinder "is-a" circle also only holds for the degenerate case of a zero height
- Still this is a suitable example to demonstrate inheritance and polymorphism

A Graphical Depiction



Abstract Class Shape

```
public abstract class Shape {

    // abstract method redefined by subclasses
    public abstract String getName ();

    // the following two methods could also be abstract,
    // in which case we could have defined Shape to be an interface

    // shape area, default 0
    public double getArea () { return 0; }

    // shape volume, default 0
    public double getVolume () { return 0; }

} // end class Shape
```

Class Point that Extends Shape

Class Circle that Extends Point

Class Cylinder that Extends Circle

Inheritance and Constructors

- When a constructor of a subclass is called, it should first call the superclass constructor using the super keyword
  - This needs to be done in the first constructor line
  - Cylinder does this for Circle which in turn does it for Point
  - Point does not need to do it because Shape has no instance variables – is an abstract class - hence no constructor, but the compiler arranges to call the Object constructor
- In all object-oriented languages, objects are constructed from the top of the inheritance hierarchy downwards
  - The constructors are called upwards (Cylinder, Circle, Point, Object) but then the relevant initialisation takes place downwards (Object, Point, Circle, Cylinder)
  - In C++ there is also a destructor per class releasing any allocated memory in the constructor; destructors are called and executed upwards (Cylinder, Circle, Point, Object)

Calling Superclass Methods from a Subclass

- A public superclass method can be called by the subclass which inherits all such methods
- If a method though is redefined (overriden) in the subclass, we can call the same superclass method using the super keyword: super**.**<superclassMethod>
  - This is similar in principle to calling the superclass constructor, it is just the syntax that is different (using the dot syntax)
- This is exactly what the toString methods of Cylinder and Circle did do for Circle and Point respectively

Final Methods

- As we saw, a subclass may redefine (or override) methods of its supeclass(es)
- If a method is defined as final, it cannot be redefined in a subclass
  - This allows the compiler to optimise relevant code
  - Remember that a variable can also be declared final when its value is not allowed to change
- In Java every method can be redefined by default unless declared final, while in C++ methods that can be redefined should be declared virtual
  - Java targets flexibility while C++ performance
- Private and static methods are implicitly final as they cannot be redefined

Protected vs. Private Variables

- We mentioned that if instance variables are declared as protected, they can be directly accessed by subclasses but not from other “outside classes”
- On the other hand, providing direct access to them even by subclasses defeats encapsulation and information hiding when protection is necessary
  - In all the examples, we defined instance variables private
  - This way the subclasses benefit from checks the superclass access methods perform
- In general, using private instead of protected instance variables is considered better software engineering practice

Using Interface Shape Instead of Abstract Class

- Abstract class Shape could have also been an interface as its implementation does not do much
  - Just returns 0 for the getArea and getVolume methods
- We redefine Shape as an interface next and we rewrite Point to implement the Shape interface instead of extending the Shape class
  - The only difference is that now Point needs to also implement the getArea and getVolume methods
  - A class implementing an interface needs to implement all the interface’s abstract methods

Class Point that Implements Shape

```
public interface Shape {          // interface instead of abstract class
    public String  getName ();
    public double getArea ();
    public double getVolume ();
} // end interface Shape

public class Point implements Shape {   // implements effectively means “inherits”
    // …

    // everything is as when inheriting from abstract class Shape
    // but here we also need to implement abstract methods getArea and getVolume

    // implement the Shape abstract method getArea, default 0
    public double getArea () { return 0; }

    // implement the Shape abstract method getVolume, default 0
    public double getVolume () { return 0; }

} // end class Point
```

Triggering Sublass Behaviour from References to Superclasses

- Having created objects of subclasses, we can assign them (“cast” them) to superclass references because a subclass “is a” superclass
  - Circle circle = new Circle(10, 20, 3.5);
  - Shape shape = circle;
- Calling redefined methods through the supeclass reference will cause the right subclass method to be called
  - String string = shape.toString();
    will result in string value “C = [10,20], R = 3.5”
- Treating subclass objects as objects of a common generic superclass and triggering the right behaviour through redefined methods is **polymorphism**

Polymorphic Behaviour Text

```
public class PolymorphicTest {
    public static void main (String[] args) {

        Point point = new Point(10, 20);
        Circle circle = new Circle(30, 40, 5);
        Cylinder cylinder = new Cylinder(50, 60, 10, 20);

        Shape[] shapes = new Shape[3]; // an array of shapes with size 3
        shapes[0] = point;
        shapes[1] = circle;
        shapes[2] = cylinder;

        // call the right polymorphic behaviour through superclass references
        for (int i = 0; i < shapes.length; i++) {
            System.out.println(shapes[i].getName() + “: “ + shapes[i].toString());
            System.out.println(“Area: “ + shapes[i].getArea());
            System.out.println(“Volume: “ + shapes[i].getVolume());
            System.out.println();
        }
    }
}
```

Polymorphism

- With polymorphism, having defined carefully an abstract class or interface, it is possible to add new specific subclasses with minimal modifications to the system's software
  - For example, a ScreenManager may treat screen objects through an abstract class or interface ScreenObject that provides methods such as draw, iconize, move, etc.
  - We can add new subclasses of ScreenObject to the system without having to modify (or even recompile!) the code of ScreenManager and this enhances <u>reusability</u> and <u>extensibility</u>
- With polymorphism, the same redefined method, e.g. draw, can have many forms of results depending on the subclass object, hence the term polymorphism

Summary

- Classes are the cornerstone of object-oriented programming and achieve data encapsulation and abstraction, promoting software reusability
- Inheritance allows subclasses to reuse data and methods of superclasses, achieving increased abstraction, reusability and also extensibility
- With polymorphism, a carefully designed abstract class or interface can allow adding new classes to a system with minimal modifications
  - We can treat diverse subclass objects as if they were objects of a common superclass