NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Chapter 3: Inheritance

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)

Learning Objectives

After the completion of this chapter, you should be able to:

- Explain inheritance in Object Oriented Programming (OOP).
- Describe method overloading and method overriding.
- Describe visibility modifiers.
- Write final classes and methods.
- Describe abstract classes and methods.
- Describe multiple inheritance and interfaces.

Topic 1: Inheritance

Inheritance

- Inheritance: An important OO feature that allows us to derive new classes from existing classes by:
  - Absorbing their attributes and behaviours (inherits).
  - Adding new capabilities in the new classes.
    - This enables code reuse and can greatly reduce the programming effort in implementing new classes
    - "is-a" relationship
- Examples:
  - Person (superclass) and Student (subclass).
  - Student is a person.

- In Java, the *root/ ancestor* ('great^n – grandfather') of all classes is java.lang.Object (http://docs.oracle.com/javase/7/docs/api/).

Inheritance

- The generalisation relationship:
  - The superclass is a generalisation of the subclasses.
  - The subclasses are specialisations of the superclass.

Inheritance Hierarchy

Superclass
(parent class or base class)

Super, base, parent, generalised

Shape

- xPos: double
- yPos: double

+ Shape()
+ Shape(xCoor: double, yCoor: double)
+ getXPos(): double
+ getYPos(): double
+ print(): void

Common attributes and methods

is-a

is-a

Rectangle

- width: double
- height: double

+ Rectangle()
+ Rectangle(xCoor: double,
yCoor: double,
w: double, h: double)
+ getWidth(): double
+ getHeight(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Additional attributes and methods

Circle

- radius: double

+ Circle()
+ Circle(xCoor: double, yCoor: double
rad: double)
+ getRadius(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Sub, derived, child, specialised

Subclasses: Inherits all the instance variables and methods of the superclass.

(EXCEPT those declared as private - accessibility)

Superclass: Shape Class

```
public class Shape {
   private double xPos ;
   private double yPos ;

   public Shape() {
      xPos = 0 ;
      yPos = 0 ;
   }
   public Shape( double xCoor , double yCoor ) {
      xPos = xCoor ;
      yPos = yCoor ;
   }
   public double getXPos() { return xPos ; }
   public double getYPos() { return yPos ; }
   public void   print()   {
     System.out.println( "[x, y] = [" + xPos
                                + ", " + yPos + "]");
   }
}
```

Constructors

Subclass: Rectangle Class

```
public class Rectangle extends Shape {
  private double width  ;
  private double height ;
  public Rectangle()
    { super() ; width = 0 ; height = 0 ; }
  public Rectangle( double xCoor , double yCoor ,
                    double w     , double h     )
    { super( xCoor , yCoor ); width = w ; height = h ; }
  public double getWidth()       { return width  ; }
  public double getHeight()      { return height ; }
  public double findPerimeter()  { return 2*(width+height) ; }
  public double findArea()       { return width * height   ; }
  public void   print() {
      System.out.println( "Rectangle print() method: " );
      System.out.print(   "Center " );
      super.print();
      System.out.println( "Width = "      + width
                        + ";Height = "   + height );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()      );
  }
}
```

Inherits superclass's attributes and methods

Constructors

From superclass

Subclass Definition

- Subclass is defined using the extends keyword.

- super:
  - Can be used in subclass constructor to call the superclass's constructor. The call to **super()** must be the **first statement** in the constructor for the class.
  - Can also be used in method definition to call the superclass's method, i.e., **super.print();**.

Subclass: Circle Class

```
public class Circle extends Shape {
  private double radius ;

  public Circle()
    { super() ; radius = 0 ; }
  public Circle( double xCoor , double yCoor , double rad )
    { super( xCoor , yCoor ); radius = rad ; }

  public double getRadius()      { return radius;                }
  public double findPerimeter()  { return 2*Math.PI*radius;      }
  public double findArea()       { return Math.PI*radius*radius; }

  public void print() {
     System.out.println( "Circle print() method: " );
     System.out.print(   "Center "                  );
     super.print();
     System.out.println( "Radius = "    + radius           );
     System.out.println( "Perimeter = " + findPerimeter() );
     System.out.println( "Area = "      + findArea()       );
  }
}
```

Constructors

The Application Class

```
public class ShapesApp {
 public static void main(String[] args) {
  Rectangle rect = new Rectangle( 5 , 5 , 10 , 10 ) ;
  System.out.print( "Rectangle: " );
  System.out.println( "Center Coordinates [x, y] = ["
                    + rect.getXPos()  + ", "
                    + rect.getYPos()  + "]" );
  System.out.println( "Width = "       + rect.getWidth()
                    + "; Height = "   + rect.getHeight()      );
  System.out.println( "Perimeter = " + rect.findPerimeter() );
  System.out.println( "Area = "       + rect.findArea()       );
  rect.print();
  Circle circ = new Circle( 5 , 10 , 5 );
  System.out.print( "Circle: " );
  System.out.println( "Center Coordinates [x, y] = ["
                    + circ.getXPos()  + ", "
                    + circ.getYPos()  + "]" );
  System.out.println( "Radius = "     + circ.getRadius()      );
  System.out.println( "Perimeter = " + circ.findPerimeter() );
  System.out.println( "Area = "       + circ.findArea()       );
  circ.print();
 }
}
```

Methods in superclass

Program Input and Output

```
Program Input and Output
Rectangle: Center Coordinates [x, y] = [5.0, 5.0]
Width = 10; Height = 10.0
Perimeter = 40.0
Area = 100.0
Rectangle print() method:
Center [x, y] = [5.0, 5.0]
Width = 10.0; Height = 10.0
Perimeter = 40.0
Area = 100.0
Circle: Center Coordinates [x, y] = [5.0, 10.0]
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
Circle print() method:
Center [x, y] = [5.0, 10.0]
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
```

Topic 2: Method Overloading

Method Overloading

(Mentioned here for comparison with Method Overriding, ***NOT** a behaviour due to Inheritance.*)

When a method is overloaded, it is designed to perform differently when it is supplied with different ***signatures***, i.e.

Same Method Name but:

1) **different number of parameters.**

OR

2) **different parameter *types*.**

Advantage:

- Overloading allows **methods** to have **similar tasks**, but **differ** only in the number of parameters required by the method **or** the data types of the parameters.
- Without overloading, we need to come up with **different method names** for each similar task instead of just one.

```
printString(…), printInt(…), printBool(….)??
print(String), print(int), print(boolean)
```

Overloading of Method

```
public class Minimum {

    public int findMin(int num1, int num2){
        if (num1 < num2)
            return num1;
        else
            return num2;
    }
    public double findMin(double num1, double num2){
        if (num1 < num2)
            return num1;
        else
            return num2;
    }
    public int findMin(int num1, int num2, int num3){
      return findMin(findMin(num1,num2),findMin(num2,num3));
    }
```

```
public int findMin(int a, int b, int c){
      ..................
    }
```

Overloading of Method

```
    public static void main(String[] args){
        int x=10, y=20, z=5;
        double i=4.5, j=5.5;
         Minimum m = new Minimum();

        System.out.println("findMin(x,y) with int
            args = "    + m.findMin(x,y));

        System.out.println("findMin(i,j) with double
            args ="     + m.findMin(i,j));

        System.out.println("findMin(x,y,z) with int
            args = "    + m.findMin(x,y,z));
    }
}
```

```
Program Output
findMin(x,y) with int args = 10
findMin(i,j) with double args = 4.5
findMin(x,y,z) with int args = 5
```

Topic 3: Method Overriding

Method Overriding

- A subclass inherits properties and methods from the superclass. Sometimes, it is necessary to alter the methods of the superclass.
- When a subclass alters a method inherited from a superclass, it overrides that method.
- To override superclass method, we define a method with *exactly* the same signature (argument(s)) and return type as the method in the superclass.
- Two ways:
  1) Refinement: Reuse the implementation of superclass method with some refinement – using the super keyword.
  2) Replacement: Replace the method completely.

Subclass: Disc Class

Subclass: Disc Class

```
public class Disc extends Circle {
  private double thickness ;
  public Disc() { super() ; thickness = 0 ; }
  public Disc( double xCoor , double yCoor ,
               double rad , double t      )
    { super( xCoor , yCoor , rad ) ; thickness = t ; }
  public double getThickness() { return thickness ; }
  public double findArea()
    { return 2 * super.findArea()
           + 2 * Math.PI * getRadius() * thickness ; }
  public double findVolume()
    { return super.findArea() * thickness ; }
  public void print() {
      System.out.println( "Disc print() method: " );
      super.print();
      System.out.println( "Radius = "    + getRadius()   );
      System.out.println( "Thickness = " + thickness     );
      System.out.println( "Area = "      + findArea()    );
      System.out.println( "Volume = "    + findVolume() );
  }
}
```

method overriding

Testing the Disc Class

```
public class DiscApp {
  public static void main( String[] args )
  {
    Circle circ = new Circle( 1   ,  5 , 10       );
    Disc   disc = new Disc(   10 , 20 , 5   , 10 );

    circ.print() ;
    disc.print() ;

    System.out.println( "Volume = " + disc.findVolume() );
    System.out.println( "XPos = "   + circ.getXPos()
                      + " YPos = "  + circ.getYPos() );
    System.out.println( "XPos = "   + disc.getXPos()
                      + " YPos = "  + disc.getYPos() );
  }
}
```

Resolving Method Calls

- When a message is sent (method call) to an object, the search for a matching method begins at the class of the object.

- If not found, the search continues to the **immediate superclass**.

- This proceeds through each immediate superclass until:
  1) A matched method is found; or
  2) No further superclass remain, where an error is reported.

Topic 4: Visibility Modifiers

Visibility Modifiers

- Visibility (accessibility/ access control):
  - public
    - Visible (accessible) to anywhere in an application.
  - private
    - Visible (accessible) only within that class's implementation.
  - protected *Stay in the family*
    - Visible (accessible) to methods of the class, the methods of subclasses, or any classes in the ***same package***.

Note: Accessible means read and modify.

- Protected variables and methods are denoted as '#' in an UML (Unified Modeling Language) diagram.

Inheritance Hierarchy

Superclass
(parent class or base class)

Shape

xPos: double
yPos: double

+ Shape()
+ Shape(xCoor: double, yCoor: double)
+ getXPos(): double
+ getYPos(): double
+ print(): void

Common attributes and methods

is-a

is-a

Rectangle

- width: double
- height: double

+ Rectangle()
+ Rectangle(xCoor: double,
yCoor: double,
w: double, h: double)
+ getWidth(): double
+ getHeight(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Additional attributes and methods

Circle

- radius: double

+ Circle()
+ Circle(xCoor: double, yCoor: double
rad: double)
+ getRadius(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Subclasses: Inherits all the instance variables and methods of the superclass. (EXCEPT those declared as private - accessibility)

Defining Package

- A package contains a set of classes that are **grouped** together in the same directory.
- Non-private data can be accessed by any object in the same package.

Example: Two classes named Class1 and Class2 in packageOne:

- The source file **Class1** contains:

```
package packageOne;
public class Class1 { ... }
```

- The source file **Class2** contains:

```
package packageOne;
public class Class2 { ... }
```

```
package packageOne;

import java.util.Scanner;

public class Class1 {
// the usual……


}
```

Package Access

- In **package access**, instance variables and methods (except private) can be accessed by name inside the definition of any class in the <u>same package</u>.

Visibility for Java

If Visibility is *(Access Level)*	Visible			
	Within Class?	Within Package?	to a Subclass?	to the World? (outside Classes)
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Not defined	Y	Y	N	N
private	Y	N	N	N

Topic 5: Final Classes and Methods

Final Classes and Methods

Final method: When a method is declared as final, the method **cannot be overridden in subclasses**. This aims to prevent the method to be overridden.

Final class: When a class is declared as final, the class **cannot be a superclass**. This means the final class will not have subclasses. The methods in this class is implicitly final.

This helps to:

1) Improve security: Ensures the behaviour of the method will not be changed by subclasses.
2) Improve efficiency: Compile-time type checking and binding can be made instead of waiting till runtime (refer 'Chapter 5: Polymorphism and Dynamic Binding').

Topic 6: Abstract Classes and Methods

Abstract Classes and Methods

- In a class inheritance hierarchy, a superclass is more **general** than its subclasses. As we move down in the hierarchy, subclasses become more **specific (specialise)** and concrete.

- Therefore, a superclass should contain general features that can be shared by subclasses. ***abstraction***

- But if a superclass is too general, then no meaningful object can be created (instantiated) from it, such classes are called abstract classes (Abstract Base Class (ABC)).

- Abstract method (with keyword abstract) <u>does not have the implementation (no code)</u> in the abstract class. The implementation must be provided by the subclasses.

Inheritance Hierarchy with Abstract Classes

In Class Diagram,
to show abstract,
class name or method name
is either *italicised* OR
with {abstract} constraint

The Abstract Class Figure

```
public abstract class Figure
{
   private String color;
   public  Figure()            { color      = "black"; }
   public  Figure( String c ) { this.color = c;       }

   public String getColor() { return color; }

   // abstract methods - no method body
   public abstract double findArea()       ;
   public abstract double findPerimeter()  ;
   public abstract void   print()          ;
}
```

no method implementation

Subclass: Rectangle Class

```
public class Rectangle extends Figure {
   private double width  ;
   private double height ;

   public Rectangle()
      { super()     ; this.width = 0 ; this.height = 0 ; }
   public Rectangle( String c , double w , double h )
       { super( c ) ; this.width = w ; this.height = h ; }
   public double getWidth()  { return width  ; }
   public double getHeight() { return height ; }

   // implementation of the abstract methods
   public double findArea()      { return width * height   ; }
   public double findPerimeter() { return 2*(width+height) ; }
   public void  print() {
      System.out.println( "Rectangle print() method: " );
      System.out.println( "Width = "    + width
                        + "; Height = " + height          );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()      );
   }
}
```

Subclass: Circle Class

```
public class Circle extends Figure
{
  private double radius;
  public Circle() { super()  ; this.radius = 0   ; }
  public Circle( String c , double rad )
                 { super(c) ; this.radius = rad ; }

  public double getRadius() { return radius; }

  // implementation of the abstract methods
  public double findArea()
    { return Math.PI * radius * radius ; }
  public double findPerimeter()
    { return 2 * Math.PI * radius ; }
  public void print() {
    System.out.println( "Circle print() method: "        );
    System.out.println( "Radius = "    + radius           );
    System.out.println( "Perimeter = " + findPerimeter() );
    System.out.println( "Area = "      + findArea()       );
  }
}
```

Testing the Figure Abstract Class

```
public class FigureApp {
 public static void main( String[] args )
 {
  Rectangle rect = new Rectangle( "Red" , 10 , 10 );
  System.out.println( "Rectangle: " );
  System.out.println( "Color : "     + rect.getColor()      );
  System.out.println( "Width = "     + rect.getWidth()      );
  System.out.println( "Height = "    + rect.getHeight ()    );
  System.out.println( "Perimeter = " + rect.findPerimeter() );
  System.out.println( "Area = "      + rect.findArea()      );
  rect.print();
  System.out.println();

  Circle circ = new Circle( "Orange" , 5 );
  System.out.println( "Circle: " );
  System.out.println( "Color : "     + circ.getColor()      );
  System.out.println( "Radius = "    + circ.getRadius()     );
  System.out.println( "Perimeter = " + circ.findPerimeter() );
  System.out.println( "Area = "      + circ.findArea()      );
  circ.print();
 }
}
```

Program Input and Output

```
Program Input and Output
Rectangle:
Color : Red
Width = 10.0
Height = 10.0
Perimeter = 40.0
Area = 100.0
Rectangle print() method:
Width = 10.0; Height = 10.0
Perimeter = 40.0
Area = 100.0
Circle:
Color : Orange
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
Circle print() method:
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
```

Topic 7: Multiple Inheritance and Interfaces

Multiple Inheritance (Multiple is-a)

Interfaces

- Java does not support multiple inheritance.
- Java supports **multiple interface**. A class can implement more than one interface and allow it to reflect the behaviour of two (or more) 'parents'.
- An interface is like an abstract class, except it contains **only abstract methods** and **constants** (**with static final**).
- A class implementing/ realising an interface promises that it will provide an implementation to **all the abstract methods** in the interface. Otherwise, the new class will also be abstract.

Example

```
public interface IntFigure
// interface is similar to abstract class
{
  // static final constants for instance variables
  // abstract methods
  public abstract double findPerimeter();
  public abstract double findArea();
  public          void   print();
}

public class Circle implements IntFigure // circle example
{
  private double radius;
  public Circle()                 { this.radius = 0   ; }
  public Circle( double rad )     { this.radius = rad ; }
  public double getRadius()       { return radius ; }
  public double findArea()        { return Math.PI*radius*radius; }
  public double findPerimeter()   { return 2*Math.PI*radius;      }
  public void   print()           { ... }
}
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface.
  - This is called extending the interface.
  - The derived interface must include the phrase extends <BaseInterfaceName>.

- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface.

Derived Interfaces

```
public interface IOrdered
{
    public boolean precedes(Object other);

    public boolean follows(Object other);
}
```

```
public interface IShowablyOrdered extends IOrdered
{
    public void showOneWhoPrecedes();
}
```

```
public class Order implements IShowablyOrdered
{
    public boolean precedes(Object other) {….}
    public boolean follows(Object other) {….}
    public void showOneWhoPrecedes() {….}
}
```

Abstract vs. Interfaces

- An abstract class is a real parent (base class); an interface is not a real parent (~~base class~~).
  type

- An abstract class can have object attributes (data members) and non-abstract methods. An interface can only have constants and abstract methods.

  Constructor?

- A class can implement more than one interface, but can extend (inherit) from only one abstract class.

Note: A Concrete class has implementation for all methods, i.e. NO abstract methods.

Example

```
public class Shape  {      // CONCRETE
        String color ;
        public Shape(String color )  {  this.color = color ; }
        public String getColor()  { return color ; }
        public double findArea()  { return 0 ; }
        public double findPerimeter() { return 0 ; }
}
```

```
public abstract class Shape  {
        String color ;
        public Shape(String color )  {  this.color = color ; }
        public String getColor()  { return color ; }
        public abstract double findArea()  ;
        public abstract double findPerimeter() ;
}
```

```
public interface IShape  {

        public double findArea()  ;
        public double findPerimeter() ;
}
```

Interface Use

Interfaces are particularly useful for assigning common functionality to possibly ***unrelated classes.***

Abstract vs. Interfaces

Abstract	Interface
May have some methods declared as `abstract`.	Can only have abstract methods.
May have `protected` properties and `static` methods.	Can only have `public` methods with no implementation.
May have `final` and non-final data attributes.	Limited to only constants (`static final`).

Both Abstract class and Interface CANNOT be instantiated with *new*, i.e.,

..... = new <AbstractClass>( ) ;

..... = new <Interface>( ) ;

Multiple Inheritance in Codes

```
public class SubClass1 extends SuperClass implements Interface1,
  Interface2, Interface3
{
  ..............
}
```

Topic 8: Java Packages

Java Packages

- Packages: Organise/ group java classes (files) into different directories according to their functionality, usability as well as category they should belong to.
- Derives from Internet domain space (namespace), i.e., com.foo.users, sg.edu.ntu.projects.
- Package namespace is global. Same class name can be unique via using namespace.
- If no name is given, classes are in an anonymous "default package".

Java Packages

- Using 2 Classes with same class name
- Example:
  - **Deck** class for deck of cards
  - **Deck** class for ship deck (level/ floor of a ship)

Casino System in a Cruise Ship

casino.game.Deck for Deck of cards

star.cruise.Deck for Ship Deck

```
public class CruiseTour {
  casino.game.Deck cards = new casino.game.Deck()
  star.cruise.Deck deck = new star.cruise.Deck();
……

}
```

Java Packages

Notes

```
Scanner sc = new Scanner(System.in) ;
char choice = sc.nextLine().charAt(0) ;
```

String

```
System.out.println("blah blah…");
```

PrintStream

```
casino.game.Deck.getTotalCards()  ;
```

```
java.lang.System.out.println("");
```

Method Header

General format

public <static> Return_Type Method_Name (Parameter_List)

Method Signature

1. Public <static>

reserved keywords

2. Return_Type

can be int, double, float, char,..., void

Class type

(Abstract Data Type)

3. Parameter_List

Example

public static **int** successor(*double num*) { …… }

public **boolean** *savetoDB(String name, int age, char gender)*
{……}

Summary

Key points from this chapter:

- Inheritance is an important OO feature that allows us to derive new classes from existing classes.
- Methods overloading: When a method is overloaded, it is designed to perform differently, when it is supplied with different signatures.
- Method overriding: When a subclass alters a method inherited from a superclass, it overrides that method.
- Visibility modifiers: Visibility (accessibility/ access control); public, private, and protected.

Summary

Key points from this chapter:

- Final method: When a method is declared as final, the method cannot be overridden in subclasses.
- Final class: When a class is declared as final, the class cannot be a superclass.
- Abstract method (with keyword abstract) does not have the implementation (no code) in the abstract class.
- Java does not support multiple inheritance, but supports multiple interface.
- Java Packages: Organise/ group java classes (files) into different directories according to their functionality, usability as well as category they should belong to.