CENG 1004
Introduction to Object Oriented Programming

Spring 2018

WEEK 7

Hiding Fields

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.

- Hiding fields is not recommended as it makes code difficult to read.

Calling methods of the superclass

- To write a method's definition of a subclass, specify a call to the public method of the superclass
  - If subclass overrides public method of superclass, specify call to public method of superclass:

    ```
    super.MethodName(parameter list)
    ```

  - If subclass does not override public method of superclass, specify call to public method of superclass:

    ```
    MethodName(parameter list)
    ```

class Box

```
public void setDimension(double l, double w, double h)
{
     super.setDimension(l, w);
     if (h >= 0)
          height = h;
     else
          height = 0;
}}
```

Box overloads the method setDimension
(Different parameters)

4

Method *Overloading*

- **Method overloading:** *multiple* methods ...
  - With the *same name*
  - But *different signatures*
  - In the *same class*
- Constructors are often overloaded
- Example:
  - **`MyClass (int inputA, int inputB)`**
  - **`MyClass (float inputA, float inputB)`**

Overloading Example From Java Library

`ArrayList` has two **`remove`** methods:

`remove (int position)`

- Removes object that is at a specified *place* in the list

`remove (Object obj)`

- Removes a *specified object* from the list

It also has two **`add`** methods:

`add (Element e)`

- Adds new object to the *end* of the list

`add (int index, Element e)`

- Adds new object at a *specified place* in the list

Defining Constructors of the Subclass

- Call to constructor of superclass:
  - Must be first statement
  - Specified by super parameter list

```
public Box()
{
    super();
    height = 0;
}

public Box(double l, double w, double h)
{
    super(l, w);
    height = h;
}
```


7

Object as a Superclass

- **`Object`** is the root of the class hierarchy
  - Every *class* has **`Object`** as a superclass
- All classes inherit the methods of Object
  - But may override them

TABLE 3.2
Methods of Class `java.lang.Object`

Method	Behavior
`Object clone()`	Makes a copy of an object.
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.

The `toString()` Method

- The Object's toString() method returns a String representation of the object, which is very useful for debugging.

- You should always override **`toString`** method if you want to print object state

- If you do *not* override it:
  - **`Object.toString`** will return a **`String`**
  - Just not the **`String`** you want!
    Example: **`ArrayBasedPD@ef08879`**
    ... The name of the class, `@`, instance's hash code

The `equals()` Method

- Compares two objects for equality and returns true if they are equal.
- The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.
- To test whether two objects are equal in the sense of containing the same information, you must override the equals() method.

The `getClass()` Method

- The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), etc..

Operations Determined by Type of Reference Variable

- Variable can refer to object whose type is a *subclass* of the variable's declared type
- Type of the *variable* determines what operations are legal
- Java is *strongly typed*
  - Compiler always verifies that variable's type includes the class of every expression assigned to the variable

```
Object obj= new Box(5,5,);
obj.area();                    //  compile-time error.
```

The `hashCode()` Method

- The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.
- By definition, if two objects are equal, their hash code must also be equal.
- If you override the equals() method, you must also override the hashCode() method as well.

Casting Objects

- *Casting* obtains a reference of different, but *matching,* type
- Casting *does not change* the object!
  - It creates an anonymous reference to the object

```
Box box= (Box)obj;
```

- *Downcast:*
  - Cast *superclass* type to *subclass* type
  - Checks *at run time* to make sure it's ok
  - If not ok, throws **`ClassCastException`**

Casting Objects

- Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance

Box box= (Box)obj; //compile-time error

- would get a compile-time error because obj is not known to the compiler to be Box

- However, we can tell the compiler that we promise to assign a MountainBike to obj by explicit casting:

`instanceof` operator

- **`instanceof`** can guard against **`ClassCastException`**

```
Object obj = ...;
if (obj instanceof Box) {
  Box box = (Box)obj;
  int area= box.area();
  ...;
} else {
  ...
}
```

Abstract Methods and Classes

- An abstract class is a class that is declared abstract
  - it may or may not include abstract methods.

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Abstract Methods and Classes

```
public abstract class Shape{

    // declare fields

    // declare nonabstract methods

    abstract void calculateArea();
    abstract void calculatePerimeter();
}
```

Abstract Methods and Classes

- When an abstract class is subclassed,
  - the subclass usually provides implementations for all of the abstract methods in its parent class.
  - if it does not, then the subclass must also be declared abstract.

Abstract Methods and Classes

```
class Circle extends Shape{
    void calculateArea() {
        ...
    }
    void calculatePerimeter() {
        ...
    }
}
```

What You Can Do in a Subclass

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.

What You Can Do in a Subclass

- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

What You Can Do in a Subclass

- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super

Other Issues

- Except for the Object class, a class has exactly one direct superclass.

- The Object class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from Object include
  - toString(), equals(), clone(), and getClass().

Other Issues

- You can prevent a class from being subclassed by using the final keyword in the class's declaration.
- Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods

Polymorhism

The Problem

```
public class Circle {
    private double radius;
    ...
    public double area(){
        return Math.PI * Math.pow(radius, 2);
    }
}
```

The Problem

```
public class Rectangle {

    double width;
    double height;
    ...
    public double area(){
        return height * width;
    }
}
```

The Problem

```
public class Drawing {

    ArrayList<Circle> circles = new ArrayList<Circle>();
    ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();

    public double calculateTotalArea(){
        double totalArea = 0;

        for (Circle circle : circles){
            totalArea += circle.area();    // totalArea = totalArea + circle.area();
        }

        for (Rectangle rect : rectangles){
            totalArea += rect.area();     // totalArea = totalArea + circle.area();
        }
        return totalArea;
    }
}
```

The Problem

- We are asked to introduce a new shape class to the application.

- How Drawing class will be affected?

New Shape: Square

```
public class Square {
    private double side;

    ...

    public double area(){
        return Math.pow(side, 2);
    }
}
```

Drawing

```
public class Drawing {

    ArrayList<Circle> circles = new ArrayList<Circle>();
    ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
    ArrayList<Square> squares = new ArrayList<Square>();
    public double calculateTotalArea(){
        double totalArea = 0;
        for (Circle circle : circles){
            totalArea += circle.area();    // totalArea = totalArea + circle.area();
        }
        for (Rectangle rect : rectangles){
            totalArea += rect.area();     // totalArea = totalArea + circle.area();
        }
        for (Square sq : squares){
            totalArea += sq.area();
        }
        return totalArea;
    }
}
```

Design Principle

- Classes should be open for extension, but closed for modification
- Allow classes to be easily extended to add new behaviour without modifying existing code
- How can we accomplish this?

Drawing (Version 2)

```
public class DrawingV2 {

    ArrayList shapes = new ArrayList();

    public double calculateTotalArea(){
        double totalArea = 0;

        for (Object shape : shapes){
            if (shape instanceof Circle){
                Circle circle = (Circle) shape;
                totalArea += circle.area();
            }else if (shape instanceof Rectangle){
                Rectangle rect= (Rectangle) shape;
                totalArea += rect.area();
            }
        }
        return totalArea;
    }
}
```

Problems with Casting

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);

Object s = c;

((Rectangle)s).changeWidth(4);
```

- Does this work?

Problems with Casting

- The following code compiles but an exception is thrown at runtime

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Object s = c;
((Rectangle)s).changeWidth(4);
```

- Casting must be done carefully and correctly
- If unsure of what type object will be then use the **`instanceof`** operator

instanceof

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Object s = c;
if(s instanceof Rectangle)
     ((Rect)s).changeWidth(4);
```

- syntax: **expression instanceof ClassName**

Casting

- It is always possible to convert a subclass to a superclass. For this reason, explicit casting can be omitted. For example,
  - `Circle c1 = new Circle(5);`
  - `Object s = c1;`

is equivalent to

  - `Object s = (Object)c1;`

- Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.
  - `Circle c2 = (Circle) s;`

Modification to handle Square

```
public class DrawingV2 {
	ArrayList shapes = new ArrayList();
	public double calculateTotalArea(){
		double totalArea = 0;
		for (Object shape : shapes){
			if (shape instanceof Circle){
				Circle circle = (Circle) shape;
				totalArea += circle.area();
			}else if (shape instanceof Rectangle){
				Rectangle rect= (Rectangle) shape;
				totalArea += rect.area();
			}else if (shape instanceof Square){
				Square sq= (Square) shape;
				totalArea += sq.area();
			}
		}
		return totalArea;
	}
}
```

DrawingV2

- Still requires modification to handle new Shapes
- It is possible to add other Objects to the shape list.
  - drawing.add(new String("abc"));
- The common super class for Rectangle, Circle and Square is java.lang.Object

Shape Class

```
public class Shape {

    public double area(){
        return 0;       //default implementation
    }

    public double perimeter(){
        return 0;       //default implementation
    }

}
```

Circle extends Shape

```
public class Circle extends Shape{
    private double radius;
    ...
    public double area(){
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Rectangle extends Shape

```
public class Rectangle extends Shape{

    double width;
    double height;
    ...
    public double area(){
        return height * width;
    }
}
```

Drawing (Version 3)

```
public class DrawingV3 {

    ArrayList<Shape> shapes = new ArrayList<Shape>();

    public void addShape(Shape shape){
        shapes.add(shape);
    }

    public double calculateTotalArea(){
        double totalArea = 0;
        for (Shape shape : shapes){
            totalArea += shape.area();
        }
        return totalArea;
    }
}
```

DrawingV3

- Does not need modification to handle new Shapes
- Only Shape typed objects can be added. Following is not possible now

```
drawing.add(new String("  ")); //compile-time error
```

- What happens if a developer forgets to override area method in a new Shape class?

```
public class Square extends Shape{
    private double side;

    public Square(double side){
        this.side = side;
    }

}
```

Abstract Shape

```
public abstract class Shape {

    public abstract double area();

    public abstract double perimeter();

}
```

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next

Polymorphism

- Suppose we create the following reference variable:

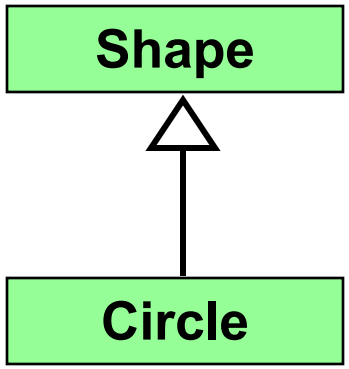
```
Shape shape;
```

- Java allows this reference to point to an `Shape` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

Polymorhism

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Shape` class is used to derive a class called `Circle`, then a `Shape` reference could be used to point to a `Circle` object




```
Shape shape;
shape = new Circle(5);
```

References and Inheritance

- Assigning a child object to a parent reference is called upcasting, and can be performed by simple assignment

  Shape shape;

  shape = new Circle(5);

- Assigning a parent object to a child reference can be done also, but it is called downcasting and must be done manually

  Circle c2 = (Circle) shape;

Polymorphism via Inheritance

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Shape` class has a method called `area`, and the `Circle and Rectangle` classes override it
- Now consider the following invocation:

```
shape.area();
```

- If `shape` refers to a `Circle` object, it invokes the `Circle` version of `area`; if it refers to a `Rectangle` object, it invokes the `Rectangle` version

References

- http://math.hws.edu/javanotes/
- https://docs.oracle.com/javase/tutorial/java/generics/
- http://www2.mta.ac.il/~amirk/java/presentations/03-Collections.ppt
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/20_ch22_Collections.ppt
- https://examples.javacodegeeks.com/core-java/util/comparator/java-comparator-example/