CS5000: Foundations of Programming

Mingon Kang, PhD
Computer Science, Kennesaw State University
Three main programming mechanisms that constitute object-oriented programming (OOP)

- Encapsulation
- Inheritance
- Polymorphism
Inheritance

Examples of inheritance

- Bicycle -> Mountain Bike, Racing bicycle
- Student -> high school student, undergraduate student, graduate student
- User -> Administrator, normal user, child account
- Employees -> hourly employees, salaried employees
A class hierarchy

Display 7.1 A Class Hierarchy

- Employee
  - HourlyEmployee
    - FullTimeHourlyEmployee
    - PartTimeEmployee
  - SalariedEmployee
    - TechnicalStaff
      - Engineer
      - Technician
    - Executive
      - ClericalStaff
Inheritance

- Inheritance allows a class (derived class) to use the properties and methods of another class (base class)
- The base class is first defined and compiled, and then more specialized version of the class (derived class) are defined by adding instance variables and methods
  - The new class: derived class, subclass, or child class
  - The original class: base class, superclass, or parent class
Inheritance

- A derived class automatically has all the instance variables and methods that the base class has.
- Can have additional instance variable and methods as well.
- Codes can be reused without having to copy it into the definitions of the derived classes.
Inheritance

- The Java Platform Class Hierarchy
  - The `object` class (defined in the `java.lang` package) defines and implements behaviors common to all classes.
  - E.g., Hierarchy for package `java.util`
    - [http://docs.oracle.com/javase/7/docs/api/java/util/package-tree.html](http://docs.oracle.com/javase/7/docs/api/java/util/package-tree.html)
Inheritance

- Syntax

```java
public class ParentClass{
    ...
}

public class ChildClass extends ParentClass{
    ...
}
```
Inheritance

// A class to display the attributes of the vehicle
class Vehicle
{
    String color;
    int speed;
    int size;
    void attributes() {
        System.out.println("Color : " + color);
        System.out.println("Speed : " + speed);
        System.out.println("Size : " + size);
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
// A subclass which extends for vehicle
class Car extends Vehicle {
    int CC;
    int gears;
    void attributescar() {
        System.out.println("Color of Car : " + color);
        System.out.println("Speed of Car : " + speed);
        System.out.println("Size of Car : " + size);
        System.out.println("CC of Car : " + CC);
        System.out.println("No of gears of Car : " + gears);
    }
}
public class Test {
    public static void main(String[] args) {
        Car b1 = new Car();
        b1.color = "Blue";
        b1.speed = 200;
        b1.size = 22;
        b1.CC = 1000;
        b1.gears = 5;
        b1.attributescar();
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
Inheritance

// A class to display the attributes of the vehicle
class Vehicle {
    String color;

    private int size;

    public int getSize() {
        return size;
    }

    public void setSize(int i) {
        size = i;
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
Inheritance

// A subclass which extends for vehicle
class Car extends Vehicle {
    int CC;
    int gears;
    int color;
    void attributescar() {
        // Error due to access violation
        //System.out.println("Size of Car : " + size);
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
Inheritance

```java
public class Test {
    public static void main(String[] args) {
        Car b1 = new Car();
        b1.color = 500;       b1.setSize(22);
        b1.CC = 1000;         b1.gears = 5;
        System.out.println("Color of Car : " + b1.color);
        System.out.println("Size of Car : " + b1.getSize());
        System.out.println("CC of Car : " + b1.CC);
        System.out.println("No of gears of Car : " + b1.gears);
    }
}
```

http://beginnersbook.com/2013/03/inheritance-in-java/
Method Overriding

- A derived class inherits methods from the base class, but it can change or override an inherited method.
  - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class.
Method Overriding

- A method can only be written in Subclass
- The argument list should be exactly the same as that of the overridden method
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class
class Company{
    public void address(){
        System.out.println("Address of Company");
    }
}

Class eBay extends Company{
    public void address(){
        System.out.println("Address of eBay");
    }
}
public class CompnayTest{
    public static void main(String[] args){
        // Company reference and object
        Company a = new Company();
        // Company reference but eBay object
        Company b = new eBay();
        a.address();
        b.address();
    }
}
Changing the Access Permission of an Overridden Method

- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class.

- However, the access permission of an overridden method cannot be changed from public in the base class to a more restricted access permission in the derived class.
Changing the Access Permission of an Overridden Method

- Given the following method header in a base case:
  `private void doSomething()`
- The following method header is valid in a derived class:
  `public void doSomething()`
- However, the opposite is not valid
- Given the following method header in a base case:
  `public void doSomething()`
- The following method header is **not** valid in a derived class:
  `private void doSomething()`
The final modifier

- If the modifier `final` is placed before the definition of a method, then that method may not be redefined in a derived class.
- If the modifier `final` is placed before the definition of a class, then that class may not be used as a base class to derive other classes.
The **super** constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class.
- In order to invoke a constructor from the base class, it uses a special syntax:
  ```java
  public derivedClass(int p1, int p2, double p3)
  {
    super(p1, p2);
    instanceVariable = p3;
  }
  ```
- In the above example, `super(p1, p2);` is a call to the base class constructor.
The super constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword `super` instead.
- A call to `super` must always be the first action taken in a constructor definition.
- An instance variable cannot be used as an argument to `super`.

2012 Pearson Addison-Wesley
The **super** constructor

- If a derived class constructor does not include an invocation of **super**, then the no-argument constructor of the base class will automatically be invoked.
  - This can result in an error if the base class has not defined a no-argument constructor.

- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to **super** should always be used.
Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class.

The same restrictions on how to use a call to `super` apply to the `this` constructor.

If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action.
The **this constructor**

- Often, a no-argument constructor uses **this** to invoke an explicit-value constructor

  - No-argument constructor (invokes explicit-value constructor using **this** and default arguments):
    ```java
    public ClassName()
    {
        this(argument1, argument2);
    }
    ```

  - Explicit-value constructor (receives default values):
    ```java
    public ClassName(type1 param1, type2 param2)
    {
        . . .
    }
    ```
class Shape {
    private int length;
    private int breadth;

    // default Constructor
    Shape() {
        length = 0;
        breadth = 0;
    }

    // Parameterized Constructor
    Shape(int len, int bdth) {
        length = len;
        breadth = bdth;
    }

    void showattributes() {
        System.out.println("length : " + length);
        System.out.println("breadth : " + breadth);
    }
}
// A subclass which extends for shape

class Rectangle extends Shape {
    private String type;

    // default Constructor
    Rectangle() {
        type = null;
    }

    // Parameterized Constructor
    Rectangle(String ty, int len, int bdth) {
        super(len,bdth);
        type = ty;
    }

    void showattributes() {
        // showattributes() of class Shape is called
        super.showattributes();
        System.out.println("type : " + type);
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
public class Test {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle("Blue", 5, 7);
        // showattributes() in rectangle is called
        rect.showattributes();
    }
}

http://beginnersbook.com/2013/03/inheritance-in-java/
If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed by name

- Inside its own class definition
- Inside any class derived from it
- In the definition of any class in the same package

The `protected` modifier provides very weak protection compared to the `private` modifier

- It allows direct access to any programmer who defines a suitable derived class
- Therefore, instance variables should normally not be marked `protected`
## Access Levels

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>package (No modifier)</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Multiple supers

- JAVA does not allow to use multiple supers.
- It is only valid to use `super` to invoke a method from a direct parent
  - Repeating `super` will not invoke a method from some other ancestor class
- For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived from the class `Employee`, it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class
  
  ```java
  super.super.toString() // ILLEGAL!
  ```
The class object

- Every class is a descendent of the class Object in Java.
- Even though a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class Object.
- The class Object is in the package java.lang which is always imported automatically.
The class object

- Every Java class can call methods of the class Object
  - E.g., equals and toString methods
  - Default implementation of equals() class provided by java.lang.Object compares memory location and only return true if two reference variable are pointing to same memory location i.e. essentially they are same object.
Overriding the equals method

```java
public boolean equals(Object otherObject) {
    if (otherObject == null)
        return false;
    if (getClass() != otherObject.getClass())
        return false;
    Car otherCar = (Car) otherObject;
    return (color.equals(otherCar.color) &&
            speed == otherCar.speed &&
            size == otherCar.size &&
            CC == otherCar.CC &&
            gears == otherCar.gears);
}
```