

Inheritance and Java

The following are aspects of Java inheritance that you need to understand.

- **Extension.** When you use the `extend` keyword to create a subclass, the subclass has access to all of the fields and methods of the superclass as if they had been defined locally. A class can only extend one superclass.
- **Hiding.** If the subclass creates a field of the same name as a field in the superclass, then the field of the superclass is hidden from normal access. However, although it is hidden, it may still be accessed by coercing the type of the access from the subclass to the superclass. **All** methods defined in the subclass access the fields and methods defined in the subclass first, and only access those in the superclass if they have not been hidden or overridden. An field in the subclass is permitted to hide a field of the superclass even if:
 - They are of different types
 - The superclass field is static and the subclass field is an instance variable.
 - The superclass field is an instance variable and the subclass field is a static variable.
- **Overriding.** If the subclass creates a method of the same name as a method in the superclass, then the method is redefined as the new version. No amount of coercion will allow an instance variable to access the overridden method of the superclass. This fact requires that overridden methods not be static.

- **Abstract Methods.** A class declaration may declare a method abstract, and provide no implementation for it. If this happens for any one methods, then the class must also be declared abstract, and no instance of it can be created. Any non-abstract subclass of an abstract class must define an implementation for all abstract methods of the superclass.

Interfaces are abstract classes with no implementations specified for any methods. Classes are permitted to implement multiple interfaces.

- **Constructors.** When you define a constructor for a class you may still want to call another version of that constructor for the same class, or pass parameters to the constructor of the superclass. Java allows you to do both. To call the default constructor for the current class, we can use the statement:

```
this();
```

as the **first** statement of our own constructor. To call a constructor of the superclass, passing the single parameter `cnt` we can use the statement:

```
super(cnt);
```

as the **first** statement of our own constructor.

The following examples are meant to provide you with an understanding of how Java handles inheritance. These are taken from the Java Language Specification book.

Example 1:

```
class Point {
    int x = 0, y = 0;

    void move(int dx, int dy) {
        x += dx; y += dy; }

    int getX() { return x; }

    int getY() { return y; }

    int color;
}

class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;

    void move(int dx, int dy) {
        move((float)dx, (float)dy); }

    void move(float dx, float dy) {
        x += dx; y += dy; }

    int getX() { return (int)Math.floor(x); }

    int getY() { return (int)Math.floor(y); }
}
```

Here the integer fields `x` and `y` of `Point` are *hidden* by the float fields `x` and `y` of `RealPoint`.

The *overriding* methods `getX` and `getY` in class `RealPoint` have the same return types as the methods of class `Point` that they override, so this code can be successfully compiled.

Consider, then, this test program:

```
class Test {
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        Point p = rp;
        rp.move(1.71828f, 4.14159f);
        p.move(1, -1);
        show(p.x, p.y);
        show(rp.x, rp.y);
        show(p.getX(), p.getY());
        show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
        System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

The output from this program is:

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

The first line of output illustrates the fact that an instance of `RealPoint` actually contains the two integer fields declared in class `Point`; it is just that their names are hidden from code that occurs within the declaration of class `RealPoint` (and those of any subclasses it might have). When a reference to an instance of class `RealPoint` in a variable of type `Point` is used to access the field `x`, the integer field `x` declared in class `Point` is accessed. The fact that its value is zero indicates that the method invocation `p.move(1, -1)` did not invoke the method `move` of class `Point`; instead, it invoked the overriding method `move` of class `RealPoint`.

The second line of output shows that the field access `rp.x` refers to the field `x` declared in class `RealPoint`. This field is of type `float`, and this second line of output accordingly displays floating-point values. Incidentally, this also illustrates the fact that the method name `show` is overloaded; the types of the arguments in the method invocation dictate which of the two definitions will be invoked.

The last two lines of output show that the method invocations `p.getX()` and `rp.getX()` each invoke the `getX` method declared in class `RealPoint`. Indeed, there is no way to invoke the `getX` method of class `Point` for an instance of class `RealPoint` from outside the body of `RealPoint`, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding.

Example 2:

```
interface I { int x = 0; }

class T1 implements I { int x = 1; }

class T2 extends T1 { int x = 2; }

class T3 extends T2 {
    int x = 3;
    void test() {
        System.out.println("x=\t\t"+x);
        System.out.println("super.x=\t\t"+super.x);
        System.out.println("((T2)this).x=\t\t"+((T2)this).x);
        System.out.println("((T1)this).x=\t\t"+((T1)this).x);
        System.out.println("((I)this).x=\t\t"+((I)this).x);
    }
}

class Test {
    public static void main(String[] args) {
        new T3().test();
    }
}
```

which produces the output:

```
x=          3
super.x=    2
((T2)this).x=  2
((T1)this).x=  1
((I)this).x=  0
```

This example demonstrates that we can access the hidden fields of the parent class directly from within the code of the subclass by using the keyword `super` or by type casting.

Example 3:

```
public class testInh {
    public static void main (String args[] ) {
        Tester test = new Tester();
        test.runIt();
    }
}

abstract class Entity {
    abstract String getClassName();
}

class Person extends Entity {

    String ClassName = "Person";
    String name;
    public static int numberOfPersons = 0;

    public String getClassName(){return ClassName;}

    public String WhoAmI(){return ClassName;}

    public Person() {numberOfPersons++;}

    public Person(String nm){this(); name = nm;}

    public String getName() {return name;}

    public void Who() {
        System.out.println("I am "+name+
            " and I KNOW I am a "+getClassName());}
}

class Student extends Person {
    String ClassName = "Student";
    String major;
    float GPA;

    public String getClassName(){return ClassName;}

    public String getMajor(){return major;}
}
```

```
public float getGPA(){return GPA;}

public Student(String nm, String maj, float G) {
    name = nm; major = maj; GPA = G; }

public void Who() {
    System.out.println("I am "+name+
        " and I KNOW I am a "+ClassName);}
}

class Faculty extends Person {
    String ClassName = "Faculty";
    String dept;

    public Faculty(String nm, String dep) {
        super(nm);
        dept = dep;
    }

    public String getClassName(){return ClassName;}

    void setDept (String dep) {dept = dep;}
    public String getDept () {return dept;}

    public void Who() {
        System.out.println("I am "+name+
            " and I KNOW I am a "+ClassName);}
}

class Tester{

    public void runIt()
    {Person per;
    Student stu;
    Faculty fac;

    PrintCnt();
    stu = new Student("Bill", "LSCS", 3.4f);
    PrintCnt();
    stu.Who();
    PrintMyIdentity(stu);

    stu = new Student("Jane", "ECS", 3.95f);
    per = stu;
    PrintCnt();
}
```

```

per.Who();
PrintMyIdentity(per);
System.out.println(per.getName()+
    " as a Person has a class name: "+per.ClassName);
System.out.println(stu.getName()+
    " as a Student has a class name: "+stu.ClassName);

per = new Person("Bob");
PrintCnt();
per.Who();
PrintMyIdentity(per);

fac = new Faculty("Jim", "CS");
PrintCnt();
fac.Who();
PrintMyIdentity(fac);
}

void PrintCnt() {
    System.out.println("\n There are currently "+
        Person.numberOfPersons+" in my database.");}

void PrintMyIdentity(Person per) {
    System.out.println("The person known as "+per.getName()+
        " is a "+per.WhoAmI());
}
}

```

The output from the previous example is:

There are currently 0 in my database.

There are currently 1 in my database.
 I am Bill and I KNOW I am a Student
 The person known as Bill is a Person

There are currently 2 in my database.
 I am Jane and I KNOW I am a Student
 The person known as Jane is a Person
 Jane as a person has a class name: Person
 Jane as a Student has a class name: Student

There are currently 3 in my database.
 I am Bob and I KNOW I am a Person
 The person known as Bob is a Person

There are currently 4 in my database.
 I am Jim and I KNOW I am a Faculty
 The person known as Jim is a Person

Why?