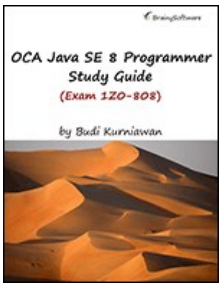

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 6: Inheritance

Overview

Inheritance is a very important object-oriented programming (OOP) feature. It is what makes code extensible in an OOP language. Extending a class is also called inheriting or subclassing. In Java, by default all classes are extendible, but you can use the **final** keyword to prevent classes from being subclassed. This chapter explains inheritance in Java.

6.1 Inheritance Overview

You extend a class by creating a new class. The former and the latter will then have a parent-child relationship. The original class is the parent class or the base class or the superclass. The new class is the child class or the subclass or the derived class of the parent. The process of extending a class in OOP is called inheritance. In a subclass you can add new methods and new fields as well as override existing methods to change their behaviors.

Figure 6.1 presents a UML class diagram that depicts a parent-child relationship between a class and a child class.

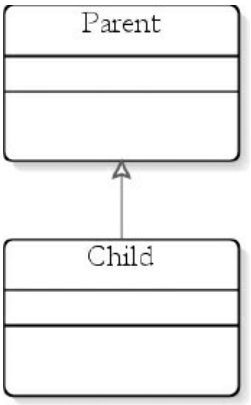

Figure 6.1: The UML class diagram for a parent class and a child class

Note that a line with an arrow is used to depict generalization, e.g. the parent-child relationship.

A child class in turn can be extended, unless you specifically make it inextensible by declaring it final. Final classes are discussed in the section "Final Classes" later in this chapter.

The benefits of inheritance are obvious. Inheritance gives you the opportunity to add some functionality that does not exist in the original class. It also gives you the chance to change the behaviors of the existing class to better suit your needs.

The extends Keyword

You extend a class by using the **extends** keyword in a class declaration, after the class name and before the parent class. Listing 6.1 presents a class named **Parent** and Listing 6.2 a class named **Child** that extends **Parent**.

Listing 6.1: The Parent class

```
public class Parent {
}
```

Listing 6.2: The Child class

```
public class Child extends Parent {
}
```

Extending a class is as simple as that.

Note All Java classes that do not explicitly extend a parent class automatically extend the **java.lang.Object** class. **Object** is the ultimate superclass in Java. **Parent** in Listing 6.1 by default is a subclass of **Object**.

Note In Java a class can only extend one class. This is unlike C++ where multiple inheritance is allowed. However, the notion of multiple inheritance can be achieved by using interfaces in Java, as discussed in Chapter 7, "Interfaces, Abstract Classes and Polymorphism."

The Is-A Relationship

There is a special relationship that is formed when you create a new class by inheritance. The subclass and the superclass has an "is-a" relationship.

For example, **Animal** is a class that represents animals. There are many types of animals, including birds, fish and dogs, so you can create subclasses of **Animal** that model specific types of animals. Figure 6.2 features the **Animal** class with three subclasses, **Bird, Fish** and **Dog**.

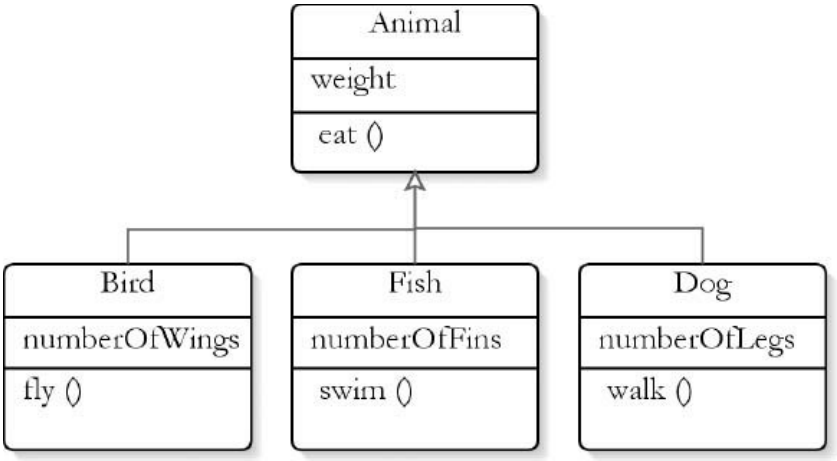

Figure 6.2: An example of inheritance

The is-a relationship between the subclasses and the superclass **Animal** is very apparent. A bird "is an" animal, a dog is an animal and a fish is an animal. A subclass is a special type of its superclass. For example, a bird is a special type of animal. The is-a relationship does not go the other way, however. An animal is not necessarily a bird or a dog.

Listing 6.3 presents the **Animal** class and its subclasses.

Listing 6.3: Animal and its subclasses

```
package app06;
class Animal {
    public float weight;
    public void eat() {
    }
}

class Bird extends Animal {
    public int numberOfWings = 2;
    public void fly() {
    }
}

class Fish extends Animal {
    public int numberOfFins = 2;
    public void swim() {
    }
}

class Dog extends Animal {
    public int numberOfLegs = 4;
    public void walk() {
    }
}
```

In this example, the **Animal** class defines a **weight** field that applies to all animals. It also declares an **eat** method because animals eat.

The **Bird** class is a special type of **Animal**, it inherits the **eat** method and the **weight** field. **Bird** also adds a **numberOfWings** field and a **fly** method. This shows that the more specific **Bird** class extends the functionality and behavior of the more generic **Animal** class.

A subclass inherits all public methods and fields of its superclass. For example, you can create a **Dog** object and call its **eat** method:

```
Dog dog = new Dog();
dog.eat();
```

The **eat** method is declared in the **Animal** class; the **Dog** class simply inherits it.

A consequence of the is-a relationship is that it is legal to assign an instance of a subclass to a reference variable of the parent type. For example, the following code is valid because **Bird** is a subclass of **Animal** and a **Bird** is always an **Animal**.

```
Animal animal = new Bird();
```

However, the following is illegal because there is no guarantee that an **Animal** is a **Dog.:**

```
Dog dog = new Animal();
```

6.2 Accessibility

From within a subclass you can access its superclass's public and protected methods and fields, but not the superclass's private methods. If the subclass and the superclass are in the same package, you can also access the superclass's default methods and fields.

Consider the **P** and **C** classes in Listing 6.4.

Listing 6.4: Showing accessibility

```
package app06;
public class P {
    public void publicMethod() {
    }
    protected void protectedMethod() {
    }
    void defaultMethod() {
    }
}class C extends P {
    public void testMethods() {
        publicMethod();
        protectedMethod();
        defaultMethod();
    }
}
```

P has three methods, one public, one protected and one with the default access level. **C** is a subclass of **P**. As you can see in the **C** class's **testMethods** method, **C** can access its parent's public and protected methods. In addition, because **C** and **P** belong to the same package, **C** can also access **P**'s default method.

However, it does not mean you can expose **P**'s non-public methods through its subclass. For example, the following code will not compile:

```
package test;
import app06.C;
public class AccessibilityTest {
    public static void main(String[] args) {
        C c = new C();
        c.protectedMethod();
    }
}
```

protectedMethod is a protected method of **P**. It is not accessible from outside **P**, except from a subclass. Since **AccessibilityTest** is not a subclass of **P**, you cannot access **P**'s protected method through its subclass **C**.

6.3 Method Overriding

When you extends a class, you can change the behavior of a method in the parent class. This is called method overriding, and this happens when you write in a subclass a method that has the same signature as a method in the parent class. If only the name is the same but the list of arguments is not, then it is method overloading. (See Chapter 3, "Objects and Classes")

You override a method to change its behavior. To override a method, you simply have to write the new method in the subclass, without having to change anything in the parent class. You can override the superclass's public and protected methods. If the subclass and superclass are in the same package, you can also override methods with the default access level.

An example of method overriding is demonstrated by the **Box** class in Listing 6.5.

Listing 6.5: The Box class

```
package app06;
public class Box {
    public int length;
    public int width;
    public int height;

    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }

    @Override
    public String toString() {
        return "I am a Box.";
    }

    @Override
    public Object clone() {
        return new Box(1, 1, 1);
    }
}
```

The **Box** class extends the **java.lang.Object** class. It is an implicit extension since the **extends** keyword is not used. **Box** overrides the public **toString** method and the protected **clone** method. Note that the **clone** method in **Box** is public whereas in **Object** it is protected. Increasing the visibility of a method defined in a superclass from protected to public is allowed. However, reducing visibility is illegal.

An overridden method is normally annotated with **@Override**. It is not required but it is good practice to do so.

What if you create a method that has the same signature as a private method in the superclass? It is not method overriding, since private methods are not visible from outside the class. It is just a method that happens to have the same signature as the private method.

Note You cannot override a final method. To make a method final, use the **final** keyword in the method declaration. For example:

```
    public final java.lang.String toUpperCase(java.lang.String s)
```

6.4 Calling the Constructors of the Superclass

A subclass is just like an ordinary class, you use the **new** keyword to create an instance of it. If you do not explicitly write a constructor in your subclass, the compiler will implicitly add a no-argument (no-arg) constructor.

When you instantiate a child class by invoking one of its constructors, the first thing the constructor does is call the no-argument constructor of the direct parent class. In the parent class, the constructor also calls the constructor of its direct parent class. This process repeats itself until it reaches the constructor of the **java.lang.Object** class. In other words, when you create a child object, all its parent classes are also instantiated.

This process is illustrated in the **Base** and **Sub** classes in Listing 6.6.

Listing 6.6: Calling a superclass's no-arg constructor

```
package app06;
class Base {
    public Base() {
        System.out.println("Base");
    }
    public Base(String s) {
        System.out.println("Base." + s);
    }
}
```

```
public class Sub extends Base {
    public Sub(String s) {
        System.out.println(s);
    }
    public static void main(String[] args) {
        Sub sub = new Sub("Start");
    }
}
```

If you run the **Sub** class, you'll see this on the console:

```
Base
Start
```

This proves that the first thing that the **Sub** class's constructor does is invoke the **Base** class's no-arg constructor. The Java compiler has quietly changed **Sub**'s constructor to the following without saving the modification to the source file.

```
public Sub(String s) {
    super();
    System.out.println(s);
}
```

The keyword **super** represents an instance of the direct superclass of the current object. Since **super** is called from an instance of **Sub, super** represents an instance of **Base**, its direct superclass.

You can explicitly call a parent's constructor from a subclass's constructor by using the **super** keyword, but **super** must be the first statement in the constructor. Using the **super** keyword is handy if you want another constructor in the superclass to be invoked. For example, you can modify the constructor in **Sub** to the following.

```
    public Sub(String s) {
        super(s);
        System.out.println(s);
    }
```

This constructor calls the single argument constructor of the parent class, by using **super(s)**. As a result, if you run the class you will see the following on the console.

```
Base.Start
Start
```

Now, what if the superclass does not have a no-arg constructor and you do not make an explicit call to another constructor from a subclass? This is illustrated in the **Parent** and **Child** classes in Listing 6.7.

Listing 6.7: Implicit calling to the parent's constructor that does not exist

---

```
package app06;
class Parent {
    public Parent(String s) {
        System.out.println("Parent(String)");
    }
}

public class Child extends Parent {
    public Child() {
    }
}
```

This will generate a compile error because the compiler adds an implicit call to the no-argument constructor in **Parent**, while the **Parent** class has only one constructor, the one that accepts a **String**. You can remedy this situation by explicitly calling the parent's constructor from the **Child** class's constructor:

```
public Child() {
    super(null);
}
```

Note It actually makes sense for a child class to call its parent's constructor from its own constructor because an instance of a subclass must always be accompanied by an instance of each of its parents. This way, calls to a method that is not overridden in a child class can be passed to its parent until the first in the hierarchy is found.

6.5 Calling the Hidden Members of the Superclass

The **super** keyword has another purpose in life. It can be used to call a hidden member or an overridden method in a superclass. Since **super** represents an instance of the direct parent, super.*memberName* returns the specified member in the parent class. You can access any member in the superclass that is visible from the subclass. For example, Listing 6.8 shows two classes that have a parent-child relationship: **Tool** and **Pencil**.

Listing 6.8: Using super to access a hidden member

```
package app06;
class Tool {
    @Override
    public String toString() {
        return "Generic tool";
    }
}

public class Pencil extends Tool {
    @Override
    public String toString() {
        return "I am a Pencil";
    }

    public void write() {
        System.out.println(super.toString());
        System.out.println(toString());
    }

    public static void main(String[] args) {
        Pencil pencil = new Pencil();
        pencil.write();
    }
}
```

The **Pencil** class overrides the **toString** method in **Tool**. If you run the **Pencil** class, you will see the following on the console.

```
Generic tool
I am a Pencil
```

Unlike calling a parent's constructor, invoking a parent's method does not have to be the first statement in the caller method.

6.6 Type Casting

You can cast an object to another type. The rule is, you can only cast an instance of a subclass to its parent class. Casting an object to a parent class is called upcasting. Here is an example, assuming that **Child** is a subclass of **Parent**.

```
Child child = new Child();
Parent parent = child;
```

To upcast a **Child** object, all you need to do is assign the object to a reference variable of type **Parent**. Note that the **parent** reference variable cannot access members that are only available in **Child**.

Because **parent** in the snippet above references an object of type **Child**, you can cast it back to **Child**. This time, it is called downcasting because you are casting an object to a class down the inheritance hierarchy. Downcasting requires that you write the child type in brackets. For example:

```
Child child = new Child();
Parent parent = child;// parent pointing to an instance of Child
Child child2 = (Child) parent; // downcasting
```

Downcasting to a subclass is only allowed if the parent class reference is already pointing to an instance of the subclass. The following will generate a compile error.

```
Object parent = new Object();
Child child = (Child) parent; // illegal downcasting, compile error
```

6.7 Final Classes

You can prevent others from extending your class by making it final using the keyword **final** in the class declaration. **final** may

appear after or before the access modifier. For example:

```
public final class Pencil
final public class Pen
```

The first form is more common.

Even though making a class final makes your code slightly faster, the difference is too insignificant to notice. Design consideration, and not speed, should be the reason you make a class final. For example, the **java.lang.String** class is final because the designer of the class did not want you to change the behavior of **String**.

6.8 The instanceof Operator

The **instanceof** operator can be used to test if an object is of a specified type. It is normally used in an **if** statement and its syntax is this.

```
if (objectReference instanceof type)
```

where *objectReference* references an object being investigated. For example, the following **if** statement returns **true**.

```
String s = "Hello";
if (s instanceof java.lang.String)
```

However, applying **instanceof** on a **null** reference variable returns **false**. For example, the following **if** statement returns **false**.

```
String s = null;
if (s instanceof java.lang.String)
```

Also, since a subclass "is a" type of its superclass, the following **if** statement, where **Child** is a subclass of **Parent**, returns **true**.

```
Child child = new Child();
if (child instanceof Parent)    // evaluates to true
```

Self Test

1 Question

Which of the following statements are true with regard to inheritance?

A. In Java a class can extend multiple classes

B. You create a subclass by using the **extends** keyword

C. In Java a class may only extend one parent class

D. You create a subclass by using the **import** keyword

2 Question

Which class is the root of all Java classes?

A. java.lang.Object

B. java.lang.Class

C. java.lang.System

D. java.lang.Runnable

3 Question

What do you call a class that extends another class?

A. A subclass

B. A final class

C. A child class

D. A base class

4 Question

What do you call a class that cannot be extended?

A. A subclass

B. A final class

C. A child class

D. An abstract class

5 Question

What do you call a class from which another class is derived?

A. A superclass

B. A final class

C. A parent class

D. A base class

6 Question

Which of the following statements are true?

A. A child class has an is-a relationship with its parent class

B. A parent class has a has-a relationship with all its children

C. A child class has a has-a relationship with its parent class

D. A child class has a has-a relationship only with its direct parent

7 Question

Consider the following two classes:

```
package com.example.inheritance;
class Parent {
    public void print() { ... }
    String describe() { ... }
    protected String[] copyElements(String[] sources) { ... }
    private float getNumber() { ... }
}

class Child extends Parent {
    public void doIt() {
        ...
    }
}
```

Which methods can be used in the **Child** class?

A. print, describe, getNumber

B. print, describe, copyElements, getNumber

C. print, describe, copyElements

D. print, describe, copyElements, toString

8 Question

Given

```
package biology;
class Insect {
}

class Butterfly extends Insect {
}

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

```
        Butterfly butterfly = new Butterfly();
        Insect insect = new Insect();
        boolean b1 = butterfly instanceof Insect;
        boolean b2 = butterfly instanceof Butterfly;
        boolean b3 = insect instanceof Butterfly;
        boolean b4 = insect instanceof Insect;

        System.out.println(b1);
        System.out.println(b2);
        System.out.println(b3);
        System.out.println(b4);
    }
}
```

After the last line of code is executed, what are the values of b1, b2, b3 and b4?

A. b1, b2, b3 and b4 are all true

B. b1, b2, b3 and b4 are all false

C. b1, b2 and b3 are true, b4 is false

D. b1, b2 and b4 are true, b3 is false

9 Question ?

Consider the following two classes:

```
package test;
class Device {
    public void printDescription() {
        System.out.println("I am a smart device");
    }
    private void printPrice() {}
}

class Computer extends Device {
    protected void printDescription() {
        System.out.println("I am a smart device");
    }
    protected void printPrice() {}
}
```

Which of the following statements is true?

A. The classes will compile with no error

B. There is a compile error caused by the **printDescription** method in **Computer**

C. There is a compile error caused by the **getPrice** method in **Computer**

D. There are compile errors caused by the **getPrice** and **printDescription** methods in **Computer**

10 Question ?

Given

```
package toolbox;
class Tool {
    public void printDescription() {
        System.out.println("I am a tool");
    }
}

class Hammer extends Tool {
    @Override
    public void printDescription() {
        System.out.println("I am a hammer");
        super.printDescription();
    }
}
```

What can be said of the two classes?

A. The @Override annotation in **Hammer** will cause a compile error

B. The two classes will compile and a call to **printDescription** in Hammer will print "I am a hammer" and "I am a tool"

C. There will be a compile error because a call to super must be the first line of code in a method

D. The two classes will compile and calling **printDescription** in Tool will print "I am a tool"

Answers

1 **B, C**.

You extend a class by using the extends keyword and a class can only be derived from a parent class.

2 **A**.

All classes are direct or indirect children of java.lang.Object.

3 **A, C**.

A class that extends another class is called a subclass or child class.

4 **B**.

A class that cannot be extended is called a final class.

5 **A, C, D**.

A class from which another class is derived is called a parent class or a superclass or a base class.

6 **A**.

A child class has an is-a relationship with its parent. For example, if class **Elephant** is derived from **Animal**, an **Elephant** is an **Animal**.

7 **C, D**.

A child class has access to the public, protected and default methods of its parents. The **Child** class can use the **print, describe** and **copyElements** methods defined in **Parent**, so C is a correct answer. Since **Parent** automatically extends **java.lang.Object, Parent** also inherits methods from **java.lang.Object** including its **toString** method. All methods that **Parent** inherit from its parent are also inherited to **Child**, so D is also correct.

8 **D**.

The **instanceof** operator examines whether or not an object is of a certain type. An object of a class is of course an instance of that class. In addition, an object of a child class is also an instance of its parent class, even though it does not work the other way around. Therefore b1, b2 and b4 are true, but not b3.

9 **B**.

The **printDescription** method in **Computer** reduces the visibility of the overriden method in the parent class from public to protected. This is not allowed in Java. The **getPrice** method in **Device** is not visible in **Computer**. Therefore, adding a method with the same signature has no effect.

10 **B, D**.

The @Override annotation type can be used to annotate a method that overrides a method in a parent class. As long as the annotated method truly overrides a method in a parent class, it will not cause a compile error. The use of super to run a method in a parent class does not have to be the first line in a method, unlike the call to super in a constructor.