The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Interfaces and Inheritance

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Inheritance

In the preceding lessons, you have seen *inheritance* mentioned several times. In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

Definitions: A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

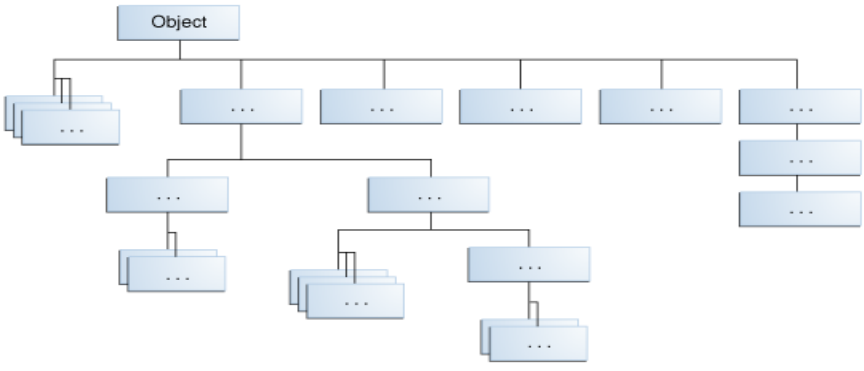
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to `Object`.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The Java Platform Class Hierarchy

The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

An Example of Inheritance

Here is the sample code for a possible implementation of a `Bicycle` class that was presented in the Classes and Objects lesson:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;
```

```
        // the Bicycle class has one constructor
        public Bicycle(int startCadence, int startSpeed, int startGear) {
            gear = startGear;
            cadence = startCadence;
            speed = startSpeed;
        }

        // the Bicycle class has four methods
        public void setCadence(int newValue) {
            cadence = newValue;
        }

        public void setGear(int newValue) {
            gear = newValue;
        }

        public void applyBrake(int decrement) {
            speed -= decrement;
        }

        public void speedUp(int increment) {
            speed += increment;
        }

    }
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
    public class MountainBike extends Bicycle {

        // the MountainBike subclass adds one field
        public int seatHeight;

        // the MountainBike subclass has one constructor
        public MountainBike(int startHeight,
                            int startCadence,
                            int startSpeed,
                            int startGear) {
            super(startCadence, startSpeed, startGear);
            seatHeight = startHeight;
        }

        // the MountainBike subclass adds one method
        public void setHeight(int newValue) {
            seatHeight = newValue;
        }
    }
```

`MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new `MountainBike` class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the `Bicycle` class were complex and had taken substantial time to debug.

What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- 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.
- 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.
- 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`.

The following sections in this lesson will expand on these topics.

Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is *not* a `MountainBike`). This is called *implicit casting*.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can *tell* the compiler that we promise to assign a `MountainBike` to `obj` by *explicit casting:*

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `MountainBike` at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

Previous page: Questions and Exercises: Interfaces
Next page: Multiple Inheritance of State, Implementation, and Type