INHERITANCE

**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.

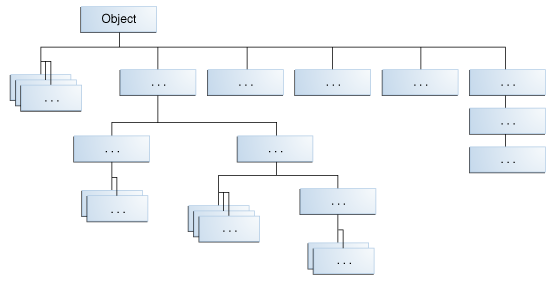
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

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) 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](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) 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 (a no args constructor call is made) or by using the keyword super. Better to be explicit!

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. (NOT in the java subset!) 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.

**Overriding and Hiding Methods**

**Instance Methods**

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method it overrides.

**Class Methods**

If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass *hides* the one in the superclass.

The distinction between hiding and overriding has important implications. The version of the overridden method that gets invoked is the one in the subclass. The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass. Let's look at an example that contains two classes. The first is Animal, which contains one instance method and one class method:

public class Animal {

public static void testClassMethod() {

System.out.println("The class" + " method in Animal.");

}

public void testInstanceMethod() {

System.out.println("The instance " + " method in Animal.");

}

}

The second class, a subclass of Animal, is called Cat:

public class Cat extends Animal {

public static void testClassMethod() {

System.out.println("The class method" + " in Cat.");

}

public void testInstanceMethod() {

System.out.println("The instance method" + " in Cat.");

}

public static void main(String[] args) {

Cat myCat = new Cat();

Animal myAnimal = myCat;

Animal.testClassMethod();

myAnimal.testInstanceMethod();

}

}

The Cat class overrides the instance method in Animal and hides the class method in Animal. The main method in this class creates an instance of Cat and calls testClassMethod() on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

The class method in Animal.

The instance method in Cat.

As promised, the version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in the subclass.

**Modifiers**

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a class method in the subclass, and vice versa.

**Summary**

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

**Defining a Method with the Same Signature as a Superclass's Method**

|  |  |  |
| --- | --- | --- |
|  | **Superclass Instance Method** | **Superclass Static Method** |
| **Subclass Instance Method** | Overrides | Generates a compile-time error |
| **Subclass Static Method** | Generates a compile-time error | Hides |

**Note:** In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods—they are new methods, unique to the subclass.

Overloading is the ability to define more than one method with the same name in a class. The compiler is able to distinguish between the methods because of their [method signatures](http://java.about.com/od/m/g/methodsignature.htm).

**Polymorphism**

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the Bicycle class. For example, a printDescription method could be added to the class that displays all the data currently stored in an instance.

public void printDescription(){

System.out.println("\nBike is " + "in gear " + this.gear

+ " with a cadence of " + this.cadence +

" and travelling at a speed of " + this.speed + ". ");

}

To demonstrate polymorphic features in the Java language, extend the Bicycle class with a MountainBike and a RoadBike class. For MountainBike, add a field for suspension, which is a String value that indicates if the bike has a front shock absorber, Front. Or, the bike has a front and back shock absorber, Dual.

Here is the updated class:

public class MountainBike extends Bicycle {

private String suspension;

public MountainBike(

int startCadence,

int startSpeed,

int startGear,

String suspensionType){

super(startCadence,

startSpeed,

startGear);

this.setSuspension(suspensionType);

}

public String getSuspension(){

return this.suspension;

}

public void setSuspension(String suspensionType) {

this.suspension = suspensionType;

}

public void printDescription() {

super.printDescription();

System.out.println("The " + "MountainBike has a" +

getSuspension() + " suspension.");

}

}

Note the overridden printDescription method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the RoadBike class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the RoadBike class:

public class RoadBike extends Bicycle{

// In millimeters (mm)

private int tireWidth;

public RoadBike(int startCadence,

int startSpeed,

int startGear,

int newTireWidth){

super(startCadence,

startSpeed,

startGear);

this.setTireWidth(newTireWidth);

}

public int getTireWidth(){

return this.tireWidth;

}

public void setTireWidth(int newTireWidth){

this.tireWidth = newTireWidth;

}

public void printDescription(){

super.printDescription();

System.out.println("The RoadBike" + " has " + getTireWidth() +

" MM tires.");

}

}

Note that once again, the printDescription method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: Bicycle, MountainBike, and RoadBike. The two subclasses override the printDescription method and print unique information.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

public class TestBikes {

public static void main(String[] args){

Bicycle bike01, bike02, bike03;

bike01 = new Bicycle(20, 10, 1);

bike02 = new MountainBike(20, 10, 5, "Dual");

bike03 = new RoadBike(40, 20, 8, 23);

bike01.printDescription();

bike02.printDescription();

bike03.printDescription();

}

}

The following is the output from the test program:

Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.

The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.

The RoadBike has 23 MM tires.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.

**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. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

**Using the Keyword super**

**Accessing Superclass Members**

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged). Consider this class, Superclass:

public class Superclass {

public void printMethod() {

System.out.println("Printed in Superclass.");

}

}

Here is a subclass, called Subclass, that overrides printMethod():

public class Subclass extends Superclass {

// overrides printMethod in Superclass

public void printMethod() {

super.printMethod();

System.out.println("Printed in Subclass");

}

public static void main(String[] args) {

Subclass s = new Subclass();

s.printMethod();

}

}

Within Subclass, the simple name printMethod() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to printMethod() inherited from Superclass, Subclass must use a qualified name, using super as shown. Compiling and executing Subclass prints the following:

Printed in Superclass.

Printed in Subclass

**Subclass Constructors**

The following example illustrates how to use the super keyword to invoke a superclass's constructor. Recall from the [Bicycle](http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html) example that MountainBike is a subclass of Bicycle. Here is the MountainBike (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

public MountainBike(int startHeight,

int startCadence,

int startSpeed,

int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

super();

or:

super(parameter list);

With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

**Note:** If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

**Object as a Superclass**

**The equals() Method**

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. 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 *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

public class Book {

...

public boolean equals(Object obj) {

if (obj instanceof Book)

return ISBN.equals((Book)obj.getISBN());

else

return false;

}

}

Consider this code that tests two instances of the Book class for equality:

// Swing Tutorial, 2nd edition

Book firstBook = new Book("0201914670");

Book secondBook = new Book("0201914670");

if (firstBook.equals(secondBook)) {

System.out.println("objects are equal");

} else {

System.out.println("objects are not equal");

}

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the equals() method if the identity operator is not appropriate for your class.

**Note:** If you override equals(), you must override hashCode() as well.

**The toString() Method**

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

System.out.println(firstBook.toString());

which would, for a properly overridden toString() method, print something useful, like this:

ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

Question 1: Consider the following two classes:

public class ClassA {

public void methodOne(int i) {

}

public void methodTwo(int i) {

}

public static void methodThree(int i) {

}

public static void methodFour(int i) {

}

}

public class ClassB extends ClassA {

public static void methodOne(int i) {

}

public void methodTwo(int i) {

}

public void methodThree(int i) {

}

public static void methodFour(int i) {

}

}

Question 1a: Which method overrides a method in the superclass? Answer 1a: methodTwo

Question 1b: Which method hides a method in the superclass? Answer 1b: methodFour

Question 1c: What do the other methods do? Answer 1c: They cause compile-time errors.