Lesson: Classes and Objects

With the knowledge you now have of the basics of the Java programming language, you can learn to write your own classes. In this lesson, you will find information about defining your own classes, including declaring member variables, methods, and constructors.

You will learn to use your classes to create objects, and how to use the objects you create.

This lesson also covers nesting classes within other classes, enumerations, and annotations.

Classes

This section shows you the anatomy of a class, and how to declare fields, methods, and constructors.

Classes

The introduction to object-oriented concepts in the lesson titled Object-oriented Programming Concepts used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a Bicycle class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
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 has
    // 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 has
    // 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 (mountain bikes have seats that can be moved up and down as the terrain demands).

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
    // field, constructor, and
    // method declarations
}
```

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
   // field, constructor, and
   // method declarations
}
```

means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access MyClass, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

- public modifier—the field is accessible from all classes.
- private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {

   private int cadence;
   private int gear;
   private int speed;

   public Bicycle(int startCadence, int startSpeed, int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   public int getCadence() {
      return cadence;
   }

   public void setCadence(int newValue) {
      cadence = newValue;
```

```
    }

    public int getGear() {
        return gear;
    }

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

    public int getSpeed() {
        return speed;
    }

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

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

Types

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions that were covered in the Language Basics lesson, Variables—Naming.

In this lesson, be aware that the same naming rules and conventions are used for method and class names, except that

- the first letter of a class name should be capitalized, and
- the first (or only) word in a method name should be a verb.

Defining Methods

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

calculateAnswer(double, int, double, double)

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
   ...
   public void draw(String s) {
      ...
   }
   public void draw(int i) {
      ...
   }
   public void draw(double f) {
      ...
   }
   public void draw(int i, double f) {
      ...
   }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
   gear = 1;
   cadence = 10;
   speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Note: If another class cannot call a MyClass constructor, it cannot directly create MyClass objects.

Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(
                double loanAmt,
                double rate,
                double futureValue,
                int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                    - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                    - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Note: *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the computePayment method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new Polygon object and initializes it from an array of Point objects (assume that Point is a class that represents an x, y coordinate):

```
public Polygon polygonFrom(Point[] corners) {
   // method body goes here
}
```

Note: The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.

Arbitrary Number of Arguments

You can use a construct called *varargs* to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {
   int numberOfSides = corners.length;
   double squareOfSide1, lengthOfSide1;
   squareOfSide1 = (corners[1].x - corners[0].x)
               * (corners[1].x - corners[0].x)
               + (corners[1].y - corners[0].y)
               * (corners[1].y - corners[0].y);
   lengthOfSide1 = Math.sqrt(squareOfSide1);

   // more method body code follows that creates and returns a
   // polygon connecting the Points
}
```

You can see that, inside the method, corners is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this printf method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

or with yet a different number of arguments.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

```
public class Circle {
   private int x, y, radius;
   public void setOrigin(int x, int y) {
      ...
   }
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the this Keyword."

Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
public class PassPrimitiveByValue {

   public static void main(String[] args) {

      int x = 3;

      // invoke passMethod() with
      // x as argument
      passMethod(x);

      // print x to see if its
      // value has changed
      System.out.println("After invoking passMethod, x = " + x);

   }

   // change parameter in passMethod()
   public static void passMethod(int p) {
      p = 10;
   }
}
```

When you run this program, the output is:

```
After invoking passMethod, x = 3
```

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves Circle objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
   // code to move origin of
   // circle to x+deltaX, y+deltaY
   circle.setX(circle.getX() + deltaX);
   circle.setY(circle.getY() + deltaY);

   // code to assign a new
   // reference to circle
   circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively. These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.

Objects

This section covers creating and using objects. You will learn how to instantiate an object, and, once instantiated, how to use the dot operator to access the object's instance variables and methods.

A typical Java program creates many objects, which as you know, interact by invoking methods. Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small program, called CreateObjectDemo, that creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object
        // and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new
            Rectangle(originOne, 100, 200);
        Rectangle rectTwo =
            new Rectangle(50, 100);

        // display rectOne's width,
        // height, and area
        System.out.println("Width of rectOne: "
                           + rectOne.width);
        System.out.println("Height of rectOne: "
                           + rectOne.height);
        System.out.println("Area of rectOne: "
                           + rectOne.getArea());

        // set rectTwo's position
        rectTwo.origin = originOne;
```

```
        // display rectTwo's position
        System.out.println("X Position of rectTwo: "
                           + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
                           + rectTwo.origin.y);

        // move rectTwo and display
        // its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: "
                           + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
                           + rectTwo.origin.y);
    }
}
```

This program creates, manipulates, and displays information about various objects. Here's the output:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

The following three sections use the above example to describe the life cycle of an object within a program. From them, you will learn how to write code that creates and uses objects in your own programs. You will also learn how the system cleans up after an object when its life has ended.

Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the CreateObjectDemo program creates an object and assigns it to a variable:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the Point class, and the second and third lines each create an object of the Rectangle class.

Each of these statements has three parts (discussed in detail below):

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The new keyword is a Java operator that creates the object.
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

type name;

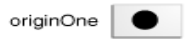
This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

```
Point originOne;
```

If you declare originOne like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the new operator, as described in the next section. You must assign an object to originOne before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, originOne, plus a reference pointing to nothing):



Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point(23, 94);
```

The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

```
int height = new Rectangle().height;
```

This statement will be discussed in the next section.

Initializing an Object

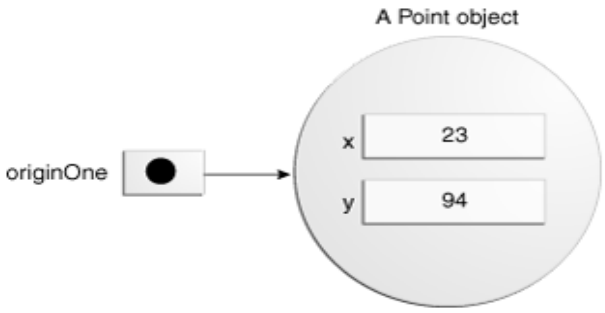
Here's the code for the Point class:

```
public class Point {
   public int x = 0;
   public int y = 0;
   //constructor
   public Point(int a, int b) {
      x = a;
      y = b;
   }
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

```
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:



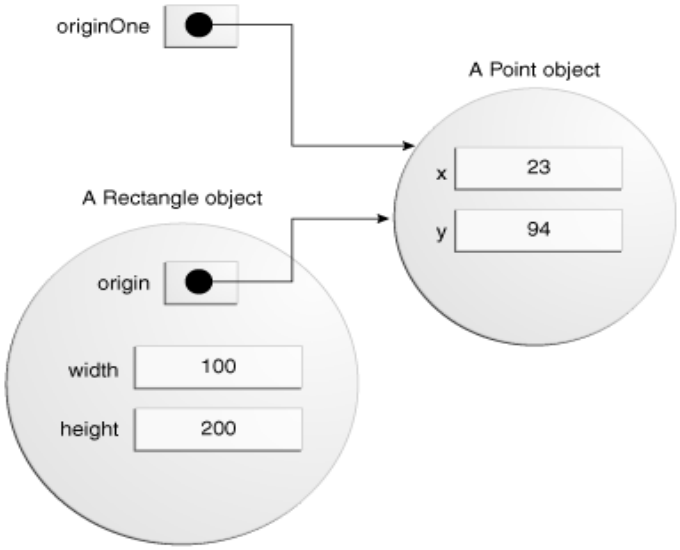
Here's the code for the Rectangle class, which contains four constructors:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area
    // of the rectangle
    public int getArea() {
        return width * height;
    }
}
```

Each constructor lets you provide initial values for the rectangle's size and width, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the same Point object—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a *no-argument constructor*:

```
Rectangle rect = new Rectangle();
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the *default constructor*. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program

Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the Rectangle class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, width and height are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the CreateObjectDemo class is outside the code for the Rectangle class. So to refer to the origin, width, and height fields within the Rectangle object named rectOne, the CreateObjectDemo class must use the names rectOne.origin, rectOne.width, and rectOne.height, respectively. The program uses two of these names to display the width and the height of rectOne:

```
System.out.println("Width of rectOne: "
                   + rectOne.width);
System.out.println("Height of rectOne: "
                   + rectOne.height);
```

Attempting to use the simple names width and height from the code in the CreateObjectDemo class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about rectTwo. Objects of the same type have their own copy of the same instance fields. Thus, each Rectangle object has fields named origin, width, and height. When you access an instance field through an object reference, you reference that particular object's field. The two objects rectOne and rectTwo in the CreateObjectDemo program have different origin, width, and height fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
```

or:

```
objectReference.methodName();
```

The Rectangle class has two methods: getArea() to compute the rectangle's area and move() to change the rectangle's origin. Here's the CreateObjectDemo code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());
...
rectTwo.move(40, 72);
```

The first statement invokes rectOne's getArea() method and displays the results. The second line moves rectTwo because the move() method assigns new values to the object's origin.x and origin.y.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The new operator returns an object reference, so you can use the value returned from new to invoke a new object's methods:

```
new Rectangle(100, 50).getArea()
```

The expression new Rectangle(100, 50) returns an object reference that refers to a Rectangle object. As shown, you can use the dot notation to invoke the new Rectangle's getArea() method to compute the area of the new rectangle.

Some methods, such as getArea(), return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by getArea() to the variable areaOfRectangle:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that getArea() is invoked on is the rectangle returned by the constructor.

The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

```
return returnValue;
```

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The getArea() method in the Rectangle Rectangle class that was discussed in the sections on objects returns an integer:

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

This method returns the integer that the expression width*height evaluates to.

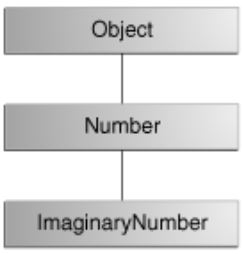
The getArea method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate Bicycle objects, we might have a method like this:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                    Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```

Returning a Class or Interface

If this section confuses you, skip it and return to it after you have finished the lesson on interfaces and inheritance.

When a method uses a class name as its return type, such as whosFastest does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which ImaginaryNumber is a subclass of java.lang.Number, which is in turn a subclass of Object, as illustrated in the following figure.



The class hierarchy for ImaginaryNumber

Now suppose that you have a method declared to return a Number:

```
public Number returnANumber() {
   ...
}
```

The returnANumber method can return an ImaginaryNumber but not an Object. ImaginaryNumber is a Number because it's a subclass of Number. However, an Object is not necessarily a Number — it could be a String or another type.

You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() {
   ...
}
```

This technique, called *covariant return type*, means that the return type is allowed to vary in the same direction as the subclass.

Note: You also can use interface names as return types. In this case, the object returned must implement the specified interface.

Using the this Keyword

Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, the Point class was written like this

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

   //constructor
   public Point(int a, int b) {
```

```
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

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

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

Using this with a Constructor

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*. Here's another Rectangle class, with a different implementation from the one in the Objects section.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose

initial value is not provided by an argument. For example, the no-argument constructor calls the four-argument constructor with four 0 values and the two-argument constructor calls the four-argument constructor with two 0 values. As before, the compiler determines which constructor to call, based on the number and the type of arguments.

If present, the invocation of another constructor must be the first line in the constructor.

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

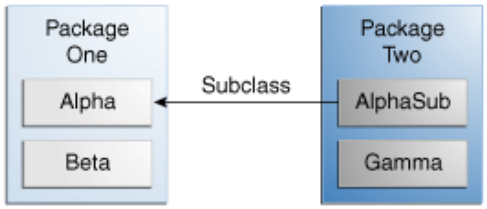
Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class â€" declared

outside this package â€" have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility				
Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Understanding Instance and Class Members

In this section, we discuss the use of the static keyword to create fields and methods that belong to the class, rather than to an instance of the class.

Class Variables

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. In the case of the Bicycle class, the instance variables are cadence, gear, and speed. Each Bicycle object has its own values for these variables, stored in different memory locations.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have the static modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

For example, suppose you want to create a number of Bicycle objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many Bicycle objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, numberOfBicycles, as follows:

```
public class Bicycle{

   private int cadence;
   private int gear;
   private int speed;

   // add an instance variable for
   // the object ID
   private int id;

   // add a class variable for the
   // number of Bicycle objects instantiated
   private static int numberOfBicycles = 0;
      ...
}
```

Class variables are referenced by the class name itself, as in

```
Bicycle.numberOfBicycles
```

This makes it clear that they are class variables.

Note: You can also refer to static fields with an object reference like
myBike.numberOfBicycles
but this is discouraged because it does not make it clear that they are class variables.

You can use the Bicycle constructor to set the id instance variable and increment the numberOfBicycles class variable:

```
public class Bicycle{

   private int cadence;
   private int gear;
   private int speed;
   private int id;
   private static int numberOfBicycles = 0;

   public Bicycle(int startCadence, int startSpeed, int startGear){
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;

      // increment number of Bicycles
      // and assign ID number
      id = ++numberOfBicycles;
   }

   // new method to return the ID instance variable
   public int getID() {
      return id;
   }
      ...
}
```

Class Methods

The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

```
ClassName.methodName(args)
```

Note: You can also refer to static methods with an object reference like
instanceName.methodName(args)
but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields. For example, we could add a static method to the Bicycle class to access the numberOfBicycles static field:

```
public static int getNumberOfBicycles() {
   return numberOfBicycles;
}
```

Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

Constants

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

For example, the following variable declaration defines a constant named PI, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

```
static final double PI = 3.141592653589793;
```

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

The Bicycle Class

After all the modifications made in this section, the Bicycle class is now:

```
public class Bicycle{
```

```
private int cadence;
private int gear;
private int speed;

private int id;

private static int numberOfBicycles = 0;


public Bicycle(int startCadence,
               int startSpeed,
               int startGear){
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;

   id = ++numberOfBicycles;
}

public int getID() {
   return id;
}

public static int getNumberOfBicycles() {
   return numberOfBicycles;
}

public int getCadence(){
   return cadence;
}

public void setCadence(int newValue){
   cadence = newValue;
}

public int getGear(){
return gear;
}

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

public int getSpeed(){
   return speed;
}

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

public void speedUp(int increment){
```

```
        speed += increment;
    }
}
```

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. An overriding method can also return a subtype of the type returned by the overridden method. This is called a *covariant return type*.

When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, it will generate an error. For more information on @Override, see Annotations.

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.

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