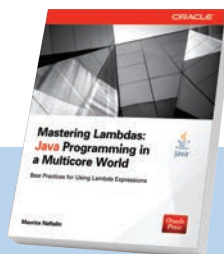


Java 8 eSampler

Preview exclusive excerpts
from brand-new and forthcoming
Oracle Press Java JDK 8 books



Learn more. Do more.™
MHPROFESSIONAL.COM

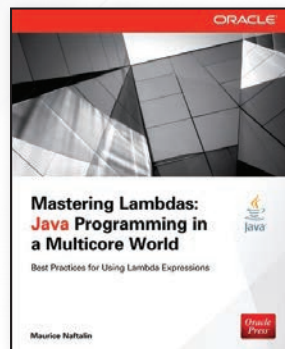
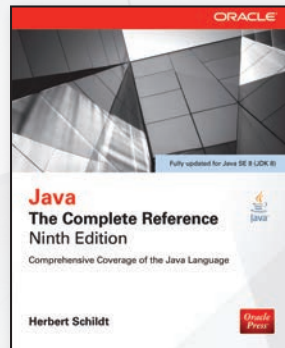


Preview exclusive excerpts from brand-new and forthcoming Oracle Press Java JDK 8 books

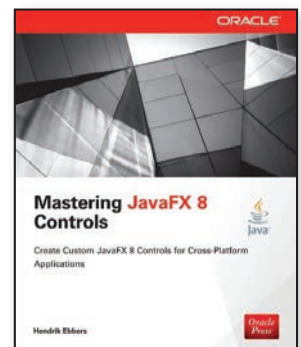
Featuring an introduction
by bestselling programming
author **Herb Schildt!**

Written by leading Java
experts, Oracle Press
books offer the most
definitive, complete, and
up-to-date coverage of the
latest Java release.

**Order today
and save 20%!**



Oracle



“It should come as no surprise that the release of JDK 8 includes new features that will once again change the way that Java code is written. Moreover, those changes will be both deep and profound, affecting virtually all types of Java applications.”

—Herbert Schildt

JDK 8 Will Change the Way You Program

by Herbert Schildt

In the world of programming, nothing stands still very long. Languages evolve, execution environments expand, and the art of programming advances. For those of us who program, it is a fact of life that change is constant. And, it is this process that keeps the profession of programming alive, exciting, and at times demanding. Thus, it should come as no surprise that the release of JDK 8 includes new features that will once again change the way that Java code is written. Moreover, those changes will be both deep and profound, affecting virtually all types of Java applications. Simply put: JDK 8 will change the way you program.

As with previous Java releases, JDK 8 contains a large number of new features. Although all are important, three stand out. They are:

- Lambda expressions
- The stream API in **java.util.stream**
- Default interface methods

Combined, lambda expressions, the stream API, and default methods fundamentally *expand the scope, power, and range* of Java. Let's take a brief look at each.

Lambda Expressions

The single most important new JDK 8 feature is the lambda expression. Java programmers have been anticipating lambda expressions for some time, and JDK 8 delivers a powerful, yet flexible implementation. Lambda expressions are so important because they add functional programming features to Java. Their use can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes. This is particularly helpful when implementing a number of commonly used event handlers, for example. Lambdas also make it easy to pass what is, in essence, a piece of executable code as an argument to a method. To support lambda expressions Java has been expanded by the inclusion of a new operator (the `->`) and a new syntax element. Make no mistake, the impact of the lambda expression will be significant, affecting both the way you design and implement Java code.

To give you an idea of the benefits that lambda expressions bring, consider the following **ActionEvent** handler, which uses the traditional, anonymous class, approach:

```
myButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        myLabel.setText("Button pressed.");  
    }  
});
```

With JDK 8, this event handler can be written using a lambda expression, as shown here:

```
myButton.addActionListener(  
    (ae) -> myLabel.setText("Button pressed.")  
);
```

As you can see, this is shorter code that is more direct and to the point. Of course, lambda expressions have many uses beyond simplifying event handlers. They offer a powerful solution to many programming challenges.

The Stream API

JDK 8 adds many new features to Java's API library. Arguably, the most important is the new stream API, which is packaged in **java.util.stream**. In the context of the stream API, a *stream* represents a sequence of data. The key aspect of the stream API is its ability to perform pipeline operations that search, filter, map, or otherwise manipulate data.

Assume that you have a list that stores employee names, the department in which they work, their e-mail addresses, and their phone numbers. Using the stream API, you can efficiently pipeline the operations that search for entries that match some criterion, such as department name, sort the matching items, and then extract only the e-mail addresses, for example. Often, you will use lambda expressions to specify the behavior of these types of operations. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

Default Methods

In the past, no method in an interface could include a body. Thus, all methods in an interface were implicitly abstract. With the release of JDK 8, this situation has changed dramatically. It is now possible for an interface method to define a default implementation. This new capability is called the *default method*.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking preexisting code. As you know, when a non-abstract class implements an interface, there must be implementations for all methods defined by that interface. In the past, if a new method was added to a popular, widely-used interface, then the addition of that method would break preexisting code, because no implementation would be found for that new method in preexisting classes. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break. This enables interfaces to be gracefully evolved over time without negative consequences.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. In the past, optional methods defined by an interface were still required to be implemented even

though they were unused. Often, this was done by providing an empty implementation. Today, a default implementation can be provided for an optional method, thus eliminating the tedium of creating empty, placeholder implementations.

Of Course, There is More

Although lambda expressions, the stream API, and default methods are the features that have the most profound impact on the character and nature of Java, JDK 8 includes several others. Among these are method references, repeating annotations, and annotations on type uses. As you would expect, there are also substantial updates and enhancements to the Java library, including a new date and time API and the functional interfaces packaged in **java.util.function**. A *functional interface* is an interface that defines one and only one abstract method. Functional interfaces provide support for lambda expressions and method references. JDK 8 also puts JavaFX front and center. This powerful GUI framework is something that no Java programmer should ignore.

With the release of JDK 8, the world of Java programming is once again changing. Many of the techniques that programmers have relied on in the past are now being replaced by better, more powerful constructs. Programmers who fail to adopt the new strategies will soon find themselves left behind. Frankly, in the competitive world of programming, no Java programmer can afford to be left behind.

Fully updated for Java SE 8 (JDK 8)

Java

The Complete Reference

Ninth Edition

Comprehensive Coverage of the Java Language



Herbert Schildt

Oracle
Press™

CHAPTER

6

Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-variable1;
```



```

    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
        // body of method
    }
    type methodname2(parameter-list) {
        // body of method
    }
    // ...
    type methodnameN(parameter-list) {
        // body of method
    }
}

```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a **main()** method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```

class Box {
    double width;
    double height;
    double depth;
}

```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have “physical” reality. For the moment, don’t worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the dot *operator*, the formal specification for Java categorizes the . as a separator. However, since the use of the term “dot operator” is widespread, it is used in this book.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;

        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
    }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this

program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in Figure 6-1.

NOTE Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname ( );
```

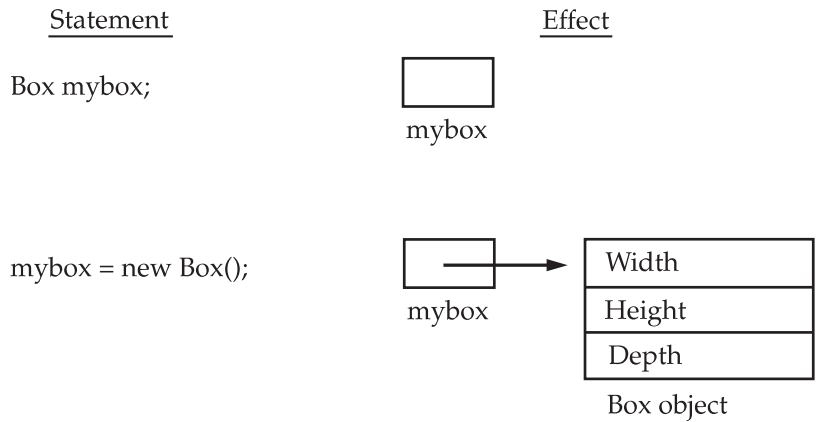


Figure 6-1 Declaring an object of type **Box**

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java’s primitive types are not implemented as objects. Rather, they are implemented as “normal” variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in Chapter 10.) For the sample programs in this book, you won’t need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let’s once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

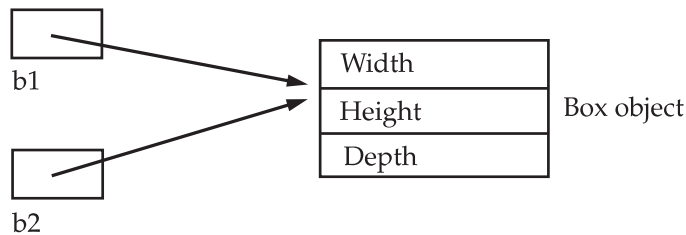
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

REMEMBER When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```
type name(parameter-list) {
    // body of method
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.
```

```
class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
```

```

Box mybox1 = new Box();
Box mybox2 = new Box();

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/* assign different values to mybox2's
   instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// display volume of first box
mybox1.volume();

// display volume of second box
mybox2.volume();
}
}

```

This program generates the following output, which is the same as the previous version.

```

Volume is 3000.0
Volume is 162.0

```

Look closely at the following two lines of code:

```

mybox1.volume();
mybox2.volume();

```

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume()** is executed, the Java run-time system transfers control to the code defined inside **volume()**. After the statements inside **volume()** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of `volume()` does move the computation of a box's volume inside the `Box` class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement `volume()` is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, when `volume()` is called, it is put on the right side of an assignment statement. On the left is a variable, in this case `vol`, that will receive the value returned by `volume()`. Thus, after

```
vol = mybox1.volume();
```

executes, the value of `mybox1.volume()` is 3,000 and this value then is stored in `vol`.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as `vol`, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the `vol` variable. The call to `volume()` could have been used in the `println()` statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when `println()` is executed, `mybox1.volume()` will be called automatically and its value will be passed to `println()`.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
    return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make `square()` much more useful.

```
int square(int i)
{
    return i * i;
}
```

Now, `square()` will return the square of whatever value it is called with. That is, `square()` is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
```

```

y = 2;
x = square(y); // x equals 4

```

In the first call to `square()`, the value 5 will be passed into parameter `i`. In the second call, `i` will receive the value 9. The third invocation passes the value of `y`, which is 2 in this example. As these examples show, `square()` is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in `square()`, `i` is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, `square(100)` passes 100 as an argument. Inside `square()`, the parameter `i` receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```

mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```

// This program uses a parameterized method.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }

    // sets dimensions of box
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

class BoxDemo5 {

```

```

public static void main(String args[] ) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
}
}

```

As you can see, the **setDim()** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim()** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim()**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```

/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

When this program is run, it generates the following results:

```

Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box()** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println()** statement

inside `Box()` is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the `new` operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname ( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

`new Box()` is calling the `Box()` constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of `Box` that did not define a constructor. The default constructor automatically initializes all instance variables to their default values, which are zero, `null`, and `false`, for numeric types, reference types, and `boolean`, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the `Box()` constructor in the preceding example does initialize a `Box` object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct `Box` objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of `Box` defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how `Box` objects are created.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```

class BoxDemo7 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

The output from this program is shown here:

```

Volume is 3000.0
Volume is 162.0

```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box()**:

```

// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}

```

This version of **Box()** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box()**, **this** will always refer to the invoking object. While it is

redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class. If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize()** method on the object.

The **finalize()** method has this general form:

```
protected void finalize()
{
    // finalization code here
}
```

Here, the keyword **protected** is a specifier that limits access to **finalize()**. This and the other access modifiers are explained in Chapter 7.

It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize()** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

NOTE If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The **finalize()** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.

A Stack Class

While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in Chapter 2, one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a "data engine." No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its

inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
    int stck[] = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

As you can see, the **Stack** class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack()** constructor initializes **tos** to **-1**, which indicates an empty stack. The method **push()** puts an item on the stack. To retrieve an item, call **pop()**. Since access to the stack is through **push()** and **pop()**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push()** and **pop()** would remain the same.

The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}

```

This program generates the following output:

```

Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10

```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stack**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

***Note that this excerpt is not the complete chapter but only part of Chapter 15**

CHAPTER

15

Lambda Expressions

During Java's ongoing development and evolution, many features have been added since its original 1.0 release. However, two stand out because they have profoundly reshaped the language, fundamentally changing the way that code is written. The first was the addition of generics, added by JDK 5. (See Chapter 14.) The second is the *lambda expression*, which is the subject of this chapter.

Added by JDK 8, lambda expressions (and their related features) significantly enhance Java because of two primary reasons. First, they add new syntax elements that increase the expressive power of the language. In the process, they streamline the way that certain common constructs are implemented. Second, the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multi-core environments, especially as it relates to the handling of for-each style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the default method (described in Chapter 9), which lets you define default behavior for an interface method, and the method reference (described here), which lets you refer to a method without executing it.

Beyond the benefits that lambda expressions bring to the language, there is another reason why they constitute an important addition to Java. Over the past few years, lambda expressions have become a major focus of computer language design. For example, they have been added to languages such as C# and C++. Their inclusion in JDK 8 helps Java remain the vibrant, innovative language that programmers have come to expect.

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions are reshaping Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

Introducing Lambda Expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression, itself. The second is the functional interface. Let's begin with a simple definition of each.

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run()**. Therefore, **run()** defines the action of **Runnable**. Furthermore, a functional interface defines the *target type* of a lambda expression. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. One other thing: a functional interface is sometimes referred to as a *SAM type*, where SAM stands for Single Abstract Method.

NOTE A functional interface may specify any public method defined by **Object**, such as **equals()**, without affecting its “functional interface” status. The public **Object** methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

Let's now look more closely at both lambda expressions and functional interfaces.

Lambda Expression Fundamentals

The lambda expression introduces a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is **->**. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The **->** can be verbalized as “becomes” or “goes to.”

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. We will begin with lambdas that define a single expression. Lambdas with block bodies are discussed later in this chapter.

At this point, it will be helpful to look a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from **Math.random()**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2) == 0
```

This lambda expression returns **true** if the value of parameter **n** is even. Although it is possible to explicitly specify the type of a parameter, such as **n** in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

Functional Interfaces

As stated, a functional interface is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract. Although this was true prior to JDK 8, the situation has changed. As explained in Chapter 9, beginning with JDK 8, it is possible to specify default behavior for a method declared in an interface. This is called a *default method*. Today, an interface method is abstract only if it does not specify a default implementation. Because nondefault interface methods are implicitly abstract, there is no need to use the **abstract** modifier (although you can specify it, if you like).

Here is an example of a functional interface:

```
interface MyNumber {
    double getValue();
}
```

In this case, the method **getValue()** is implicitly abstract, and it is the only method defined by **MyNumber**. Thus, **MyNumber** is a functional interface, and its function is defined by **getValue()**.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let's work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

```
// Create a reference to a MyNumber instance.
MyNumber myNum;
```

Next, a lambda expression is assigned to that interface reference:

```
// Use a lambda in an assignment context.
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the `getValue()` method. As a result, the following displays the value 123.45:

```
// Call getValue(), which is implemented by the previously assigned
// lambda expression.
System.out.println("myNum.getValue());
```

Because the lambda expression assigned to `myNum` returns the value 123.45, that is the value obtained when `getValue()` is called.

In order for a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two `int` parameters, then the lambda must specify two parameters whose type either is explicitly `int` or can be implicitly inferred as `int` by the context. In general, the type and number of the lambda expression's parameters must be compatible with the method's parameters; the return types must be compatible; and any exceptions thrown by the lambda expression must be acceptable to the method.

Some Lambda Expression Examples

With the preceding discussion in mind, let's look at some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing section.

```
// Demonstrate a simple lambda expression.

// A functional interface.
interface MyNumber {
    double getValue();
}

class LambdaDemo {
    public static void main(String args[])
    {
        MyNumber myNum; // declare an interface reference

        // Here, the lambda expression is simply a constant expression.
        // When it is assigned to myNum, a class instance is
        // constructed in which the lambda expression implements
        // the getValue() method in MyNumber.
        myNum = () -> 123.45;
```

```

// Call getValue(), which is provided by the previously assigned
// lambda expression.
System.out.println("A fixed value: " + myNum.getValue());

// Here, a more complex expression is used.
myNum = () -> Math.random() * 100;

// These call the lambda expression in the previous line.
System.out.println("A random value: " + myNum.getValue());
System.out.println("Another random value: " + myNum.getValue());

// A lambda expression must be compatible with the method
// defined by the functional interface. Therefore, this won't work:
// myNum = () -> "123.03"; // Error!
}
}

```

Sample output from the program is shown here:

```

A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129

```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue()**.

The next example shows the use of a parameter with a lambda expression:

```

// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
    boolean test(int n);
}

class LambdaDemo2 {
    public static void main(String args[])
    {
        // A lambda expression that tests if a number is even.
        NumericTest isEven = (n) -> (n % 2)==0;

        if(isEven.test(10)) System.out.println("10 is even");
        if(!isEven.test(9)) System.out.println("9 is not even");

        // Now, use a lambda expression that tests if a number
        // is non-negative.
        NumericTest isNonNeg = (n) -> n >= 0;

        if(isNonNeg.test(1)) System.out.println("1 is non-negative");
        if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
    }
}

```


The output from this program is shown here:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for evenness. It is shown again here:

```
(n) -> (n % 2)==0
```

Notice that the type of **n** is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of **test()** as defined by the **NumericTest** interface, which is **int**. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid way to write the preceding:

```
(int n) -> (n % 2)==0
```

Here, **n** is explicitly specified as **int**. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it.

This program demonstrates another important point about lambda expressions: A functional interface reference can be used to execute any lambda expression that is compatible with it. Notice that the program defines two different lambda expressions that are compatible with the **test()** method of the functional interface **NumericTest**. The first, called **isEven**, determines if a value is even. The second, called **isNonNeg**, checks if a value is non-negative. In each case, the value of the parameter **n** is tested. Because each lambda expression is compatible with **test()**, each can be executed through a **NumericTest** reference.

One other point before moving on. When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2)==0
```

For consistency, this book will surround all lambda expression parameter lists with parentheses, even those containing only one parameter. Of course, you are free to adopt a different style.

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
    boolean test(int n, int d);
}

class LambdaDemo3 {
```

```

public static void main(String args[])
{
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

    if(isFactor.test(10, 2))
        System.out.println("2 is a factor of 10");

    if(!isFactor.test(10, 3))
        System.out.println("3 is not a factor of 10");
}
}

```

The output is shown here:

```

2 is a factor of 10
3 is not a factor of 10

```

In this program, the functional interface **NumericTest2** defines the **test()** method:

```
boolean test(int n, int d);
```

In this version, **test()** specifies two parameters. Thus, for a lambda expression to be compatible with **test()**, the lambda expression must also specify two parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, **n** and **d**, are specified in the parameter list, separated by commas. This example can be generalized. Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesized list on the left side of the lambda operator.

Here is an important point about multiple parameters in a lambda expression: If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas*. In an expression body, the code on the right side of the lambda operator must consist of a single expression. While

expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body*. Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify **if** and **switch** statements, create nested blocks, and so on. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

```
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
    int func(int n);
}

class BlockLambdaDemo {
    public static void main(String args[])
    {
        // This block lambda computes the factorial of an int value.
        NumericFunc factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the block lambda declares a variable called **result**, uses a **for** loop, and has a **return** statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. One other point. When a **return**

statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
    String func(String n);
}

class BlockLambdaDemo2 {
    public static void main(String args[])
    {

        // This block lambda reverses the characters in a string.
        StringFunc reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
            reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
            reverse.func("Expression"));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface **StringFunc** declares the **func()** method. This method takes a parameter of type **String** and has a return type of **String**. Thus, in the **reverse** lambda expression, the type of **str** is inferred to be **String**. Notice that the **charAt()** method is called on **str**. This is legal because of the inference that **str** is of type **String**.

Generic Functional Interfaces

A lambda expression, itself, cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some “generic-like” qualities.) However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is

determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

To understand the value of generic functional interfaces, consider this. The two examples in the previous section used two different functional interfaces, one called **NumericFunc** and the other called **StringFunc**. However, both defined a method called **func()** that took one parameter and returned a result. In the first case, the type of the parameter and return type was **int**. In the second case, the parameter and return type was **String**. Thus, the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
    T func(T t);
}

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

        // Use a String-based version of SomeFunc.
        SomeFunc<String> reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Lambda reversed is " +
            reverse.func("Lambda"));
        System.out.println("Expression reversed is " +
            reverse.func("Expression"));

        // Now, use an Integer-based version of SomeFunc.
        SomeFunc<Integer> factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("The factorial of 3 is " + factorial.func(3));
        System.out.println("The factorial of 5 is " + factorial.func(5));
    }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, the generic functional interface **SomeFunc** is declared as shown here:

```
interface SomeFunc<T> {
    T func(T t);
}
```

Here, **T** specifies both the return type and the parameter type of **func()**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The **SomeFunc** interface is used to provide a reference to two different types of lambdas. The first uses type **String**. The second uses type **Integer**. Thus, the same functional interface can be used to refer to the **reverse** lambda and the **factorial** lambda. Only the type argument passed to **SomeFunc** differs.

Passing Lambda Expressions as Arguments

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
// Use lambda expressions as an argument to a method.

interface StringFunc {
    String func(String n);
}

class LambdasAsArgumentsDemo {

    // This method has a functional interface as the type of
    // its first parameter. Thus, it can be passed a reference to
    // any instance of that interface, including the instance created
    // by a lambda expression.
    // The second parameter specifies the string to operate on.
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
```

```

{
    String inStr = "Lambdas add power to Java";
    String outStr;

    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) -> {
        String result = "";
        int i;

        for(i = 0; i < str.length(); i++)
            if(str.charAt(i) != ' ')
                result += str.charAt(i);

        return result;
    }, inStr);

    System.out.println("The string with spaces removed: " + outStr);

    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to an
    // instance of StringFunc.
    StringFunc reverse = (str) -> {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    };

    // Now, reverse can be passed as the first parameter to stringOp()
    // since it refers to a StringFunc object.
    System.out.println("The string reversed: " +
        stringOp(reverse, inStr));
}
}

```

The output is shown here:

```

Here is input string: Lambdas add power to Java
The string in uppercase: LAMBIDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL

```

In the program, first notice the `stringOp()` method. It has two parameters. The first is of type `StringFunc`, which is a functional interface. Thus, this parameter can receive a reference to any instance of `StringFunc`, including one created by a lambda expression. The second argument of `stringOp()` is of type `String`, and this is the string operated on.

Next, notice the first call to `stringOp()`, shown again here:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface `StringFunc` is created and a reference to that object is passed to the first parameter of `stringOp()`. Thus, the lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter. Because the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique—especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to `stringOp()`. This lambda removes spaces from a string. It is shown again here:

```
outStr = stringOp((str) -> {
    String result = "";
    int i;

    for(i = 0; i < str.length(); i++)
        if(str.charAt(i) != ' ')
            result += str.charAt(i);

    return result;
}, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case, however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to `reverse`, which is a reference to a `StringFunc` instance. Thus, `reverse` can be used as an argument to the first parameter of `stringOp()`. The program then calls `stringOp()`, passing in `reverse` and the string on which to operate. Because the instance obtained by the evaluation of each lambda expression is an implementation of `StringFunc`, each can be used as the first parameter to `stringOp()`.

One last point: In addition to variable initialization, assignment, and argument passing, the following also constitute target type contexts: casts, the `?` operator, array initializers, `return` statements, and lambda expressions, themselves.

Lambda Expressions and Exceptions

A lambda expression can throw an exception. However, if it throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface. Here is an example that illustrates this fact. It computes the average of an array of **double** values. If a zero-length array is passed, however, it throws the custom exception **EmptyArrayException**. As the example shows, this exception is listed in the **throws** clause of **func()** declared inside the **DoubleNumericArrayFunc** functional interface.

```
// Throw an exception from a lambda expression.

interface DoubleNumericArrayFunc {
    double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
    EmptyArrayException() {
        super("Array Empty");
    }
}

class LambdaExceptionDemo {

    public static void main(String args[]) throws EmptyArrayException
    {
        double[] values = { 1.0, 2.0, 3.0, 4.0 };

        // This block lambda computes the average of an array of doubles.
        DoubleNumericArrayFunc average = (n) -> {
            double sum = 0;

            if(n.length == 0)
                throw new EmptyArrayException();

            for(int i=0; i < n.length; i++)
                sum += n[i];

            return sum / n.length;
        };

        System.out.println("The average is " + average.func(values));

        // This causes an exception to be thrown.
        System.out.println("The average is " + average.func(new double[0]));
    }
}
```

The first call to **average.func()** returns the value 2.5. The second call, which passes a zero-length array, causes an **EmptyArrayException** to be thrown. Remember, the inclusion of the **throws** clause in **func()** is necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with **func()**.

Fully updated for Java SE 8 (JDK 8)

Java

A Beginner's Guide

Sixth Edition

Create, Compile, and Run Java Programs Today

Herbert Schildt



Oracle
Press™



Chapter 4

Introducing Classes, Objects, and Methods

Key Skills & Concepts

- Know the fundamentals of the class
 - Understand how objects are created
 - Understand how reference variables are assigned
 - Create methods, return values, and use parameters
 - Use the **return** keyword
 - Return a value from a method
 - Add parameters to a method
 - Utilize constructors
 - Create parameterized constructors
 - Understand **new**
 - Understand garbage collection and finalizers
 - Use the **this** keyword
-

Before you can go much further in your study of Java, you need to learn about the class. The class is the essence of Java. It is the foundation upon which the entire Java language is built because the class defines the nature of an object. As such, the class forms the basis for object-oriented programming in Java. Within a class are defined data and code that acts upon that data. The code is contained in methods. Because classes, objects, and methods are fundamental to Java, they are introduced in this chapter. Having a basic understanding of these features will allow you to write more sophisticated programs and better understand certain key Java elements described in the following chapter.

Class Fundamentals

Since all Java program activity occurs within a class, we have been using classes since the start of this book. Of course, only extremely simple classes have been used, and we have not taken advantage of the majority of their features. As you will see, classes are substantially more powerful than the limited ones presented so far.

Let's begin by reviewing the basics. A class is a template that defines the form of an object. It specifies both the data and the code that will operate on that data. Java uses a class specification to construct *objects*. Objects are *instances* of a class. Thus, a class is essentially

a set of plans that specify how to build an object. It is important to be clear on one issue: a class is a logical abstraction. It is not until an object of that class has been created that a physical representation of that class exists in memory.

One other point: Recall that the methods and variables that constitute a class are called *members* of the class. The data members are also referred to as *instance variables*.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the instance variables that it contains and the methods that operate on them. Although very simple classes might contain only methods or only instance variables, most real-world classes contain both.

A class is created by using the keyword **class**. A simplified general form of a **class** definition is shown here:

```
class classname {  
    // declare instance variables  
    type var1;  
    type var2;  
    // ...  
    type varN;  
  
    // declare methods  
    type method1(parameters) {  
        // body of method  
    }  
    type method2(parameters) {  
        // body of method  
    }  
    // ...  
    type methodN(parameters) {  
        // body of method  
    }  
}
```

Although there is no syntactic rule that enforces it, a well-designed class should define one and only one logical entity. For example, a class that stores names and telephone numbers will not normally also store information about the stock market, average rainfall, sunspot cycles, or other unrelated information. The point here is that a well-designed class groups logically connected information. Putting unrelated information into the same class will quickly destructure your code!

Up to this point, the classes that we have been using have had only one method: **main()**. Soon you will see how to create others. However, notice that the general form of a class does not specify a **main()** method. A **main()** method is required only if that class is the starting point for your program. Also, some types of Java applications, such as applets, don't require a **main()**.

Defining a Class

To illustrate classes we will develop a class that encapsulates information about vehicles, such as cars, vans, and trucks. This class is called **Vehicle**, and it will store three items of information about a vehicle: the number of passengers that it can carry, its fuel capacity, and its average fuel consumption (in miles per gallon).

The first version of **Vehicle** is shown next. It defines three instance variables: **passengers**, **fuelcap**, and **mpg**. Notice that **Vehicle** does not contain any methods. Thus, it is currently a data-only class. (Subsequent sections will add methods to it.)

```
class Vehicle {
    int passengers; // number of passengers
    int fuelcap;   // fuel capacity in gallons
    int mpg;       // fuel consumption in miles per gallon
}
```

A **class** definition creates a new data type. In this case, the new data type is called **Vehicle**. You will use this name to declare objects of type **Vehicle**. Remember that a **class** declaration is only a type description; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Vehicle** to come into existence.

To actually create a **Vehicle** object, you will use a statement like the following:

```
Vehicle minivan = new Vehicle(); // create a Vehicle object called minivan
```

After this statement executes, **minivan** will be an instance of **Vehicle**. Thus, it will have “physical” reality. For the moment, don’t worry about the details of this statement.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Vehicle** object will contain its own copies of the instance variables **passengers**, **fuelcap**, and **mpg**. To access these variables, you will use the dot (.) operator. The *dot operator* links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Thus, the object is specified on the left, and the member is put on the right. For example, to assign the **fuelcap** variable of **minivan** the value 16, use the following statement:

```
minivan.fuelcap = 16;
```

In general, you can use the dot operator to access both instance variables and methods.

Here is a complete program that uses the **Vehicle** class:

```
/* A program that uses the Vehicle class.

   Call this file VehicleDemo.java
*/
class Vehicle {
    int passengers; // number of passengers
    int fuelcap;   // fuel capacity in gallons
    int mpg;       // fuel consumption in miles per gallon
}
```

© 2014 McGraw-Hill Education

```

}

// This class declares an object of type Vehicle.
class VehicleDemo {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        int range;

        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16; ← Notice the use of the dot
        minivan.mpg = 21;      operator to access a member.

        // compute the range assuming a full tank of gas
        range = minivan.fuelcap * minivan.mpg;
        System.out.println("Minivan can carry " + minivan.passengers +
            " with a range of " + range);
    }
}

```

You should call the file that contains this program **VehicleDemo.java** because the **main()** method is in the class called **VehicleDemo**, not the class called **Vehicle**. When you compile this program, you will find that two **.class** files have been created, one for **Vehicle** and one for **VehicleDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Vehicle** and the **VehicleDemo** class to be in the same source file. You could put each class in its own file, called **Vehicle.java** and **VehicleDemo.java**, respectively.

To run this program, you must execute **VehicleDemo.class**. The following output is displayed:

```
Minivan can carry 7 with a range of 336
```

Before moving on, let's review a fundamental principle: each object has its own copies of the instance variables defined by its class. Thus, the contents of the variables in one object can differ from the contents of the variables in another. There is no connection between the two objects except for the fact that they are both objects of the same type. For example, if you have two **Vehicle** objects, each has its own copy of **passengers**, **fuelcap**, and **mpg**, and the contents of these can differ between the two objects. The following program demonstrates this fact. (Notice that the class with **main()** is now called **TwoVehicles**.)

```

// This program creates two Vehicle objects.

class Vehicle {
    int passengers; // number of passengers
    int fuelcap;    // fuel capacity in gallons
    int mpg;        // fuel consumption in miles per gallon
}

// This class declares an object of type Vehicle.
class TwoVehicles {
    public static void main(String args[]) {

```

```

Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();

int range1, range2;

// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;

// compute the ranges assuming a full tank of gas
range1 = minivan.fuelcap * minivan.mpg;
range2 = sportscar.fuelcap * sportscar.mpg;

System.out.println("Minivan can carry " + minivan.passengers +
    " with a range of " + range1);

System.out.println("Sportscar can carry " + sportscar.passengers +
    " with a range of " + range2);
}
}

```

Remember, **minivan** and **sportscar** refer to separate objects.

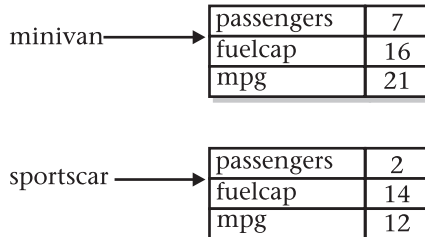
The output produced by this program is shown here:

```

Minivan can carry 7 with a range of 336
Sportscar can carry 2 with a range of 168

```

As you can see, **minivan**'s data is completely separate from the data contained in **sportscar**. The following illustration depicts this situation.



How Objects Are Created

In the preceding programs, the following line was used to declare an object of type **Vehicle**:

```
Vehicle minivan = new Vehicle();
```


This declaration performs two functions. First, it declares a variable called **minivan** of the class type **Vehicle**. This variable does not define an object. Instead, it is simply a variable that can *refer to* an object. Second, the declaration creates a physical copy of the object and assigns to **minivan** a reference to that object. This is done by using the **new** operator.

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in a variable. Thus, in Java, all class objects must be dynamically allocated.

The two steps combined in the preceding statement can be rewritten like this to show each step individually:

```
Vehicle minivan; // declare reference to object
minivan = new Vehicle(); // allocate a Vehicle object
```

The first line declares **minivan** as a reference to an object of type **Vehicle**. Thus, **minivan** is a variable that can refer to an object, but it is not an object itself. At this point, **minivan** does not refer to an object. The next line creates a new **Vehicle** object and assigns a reference to it to **minivan**. Now, **minivan** is linked with an object.

Reference Variables and Assignment

In an assignment operation, object reference variables act differently than do variables of a primitive type, such as **int**. When you assign one primitive-type variable to another, the situation is straightforward. The variable on the left receives a *copy* of the *value* of the variable on the right. When you assign one object reference variable to another, the situation is a bit more complicated because you are changing the object that the reference variable refers to. The effect of this difference can cause some counterintuitive results. For example, consider the following fragment:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

At first glance, it is easy to think that **car1** and **car2** refer to different objects, but this is not the case. Instead, **car1** and **car2** will both refer to the same object. The assignment of **car1** to **car2** simply makes **car2** refer to the same object as does **car1**. Thus, the object can be acted upon by either **car1** or **car2**. For example, after the assignment

```
car1.mpg = 26;
```

executes, both of these **println()** statements

```
System.out.println(car1.mpg);
System.out.println(car2.mpg);
```

display the same value: 26.

Although **car1** and **car2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **car2** simply changes the object to which **car2** refers. For example:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();

car2 = car3; // now car2 and car3 refer to the same object.
```

After this sequence executes, **car2** refers to the same object as **car3**. The object referred to by **car1** is unchanged.

Methods

As explained, instance variables and methods are constituents of classes. So far, the **Vehicle** class contains data, but no methods. Although data-only classes are perfectly valid, most classes will have methods. Methods are subroutines that manipulate the data defined by the class and, in many cases, provide access to that data. In most cases, other parts of your program will interact with a class through its methods.

A method contains one or more statements. In well-written Java code, each method performs only one task. Each method has a name, and it is this name that is used to call the method. In general, you can give a method whatever name you please. However, remember that **main()** is reserved for the method that begins execution of your program. Also, don't use Java's keywords for method names.

When denoting methods in text, this book has used and will continue to use a convention that has become common when writing about Java. A method will have parentheses after its name. For example, if a method's name is **getval**, it will be written **getval()** when its name is used in a sentence. This notation will help you distinguish variable names from method names in this book.

The general form of a method is shown here:

```
ret-type name( parameter-list ) {
    // body of method
}
```

Here, *ret-type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, the parameter list will be empty.

Adding a Method to the Vehicle Class

As just explained, the methods of a class typically manipulate and provide access to the data of the class. With this in mind, recall that **main()** in the preceding examples computed the range of a vehicle by multiplying its fuel consumption rate by its fuel capacity. While technically correct,

This program generates the following output:

```
Minivan can carry 7. Range is 336
Sportscar can carry 2. Range is 168
```

Let's look at the key elements of this program, beginning with the `range()` method itself. The first line of `range()` is

```
void range() {
```

This line declares a method called `range` that has no parameters. Its return type is `void`. Thus, `range()` does not return a value to the caller. The line ends with the opening curly brace of the method body.

The body of `range()` consists solely of this line:

```
System.out.println("Range is " + fuelcap * mpg);
```

This statement displays the range of the vehicle by multiplying `fuelcap` by `mpg`. Since each object of type `Vehicle` has its own copy of `fuelcap` and `mpg`, when `range()` is called, the range computation uses the calling object's copies of those variables.

The `range()` method ends when its closing curly brace is encountered. This causes program control to transfer back to the caller.

Next, look closely at this line of code from inside `main()`:

```
minivan.range();
```

This statement invokes the `range()` method on `minivan`. That is, it calls `range()` relative to the `minivan` object, using the object's name followed by the dot operator. When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call.

In this case, the call to `minivan.range()` displays the range of the vehicle defined by `minivan`. In similar fashion, the call to `sportscar.range()` displays the range of the vehicle defined by `sportscar`. Each time `range()` is invoked, it displays the range for the specified object.

There is something very important to notice inside the `range()` method: the instance variables `fuelcap` and `mpg` are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that `fuelcap` and `mpg` inside `range()` implicitly refer to the copies of those variables found in the object that invokes `range()`.

Returning from a Method

In general, there are two conditions that cause a method to return—first, as the `range()` method in the preceding example shows, when the method's closing curly brace is encountered. The second is when a `return` statement is executed. There are two forms of `return`—one for use in

void methods (those that do not return a value) and one for returning values. The first form is examined here. The next section explains how to return values.

In a **void** method, you can cause the immediate termination of a method by using this form of **return**:

```
return ;
```

When this statement executes, program control returns to the caller, skipping any remaining code in the method. For example, consider this method:

```
void myMeth() {
    int i;

    for(i=0; i<10; i++) {
        if(i == 5) return; // stop at 5
        System.out.println();
    }
}
```

Here, the **for** loop will only run from 0 to 5, because once **i** equals 5, the method returns. It is permissible to have multiple return statements in a method, especially when there are two or more routes out of it. For example:

```
void myMeth() {
    // ...
    if(done) return;
    // ...
    if(error) return;
    // ...
}
```

Here, the method returns if it is done or if an error occurs. Be careful, however, because having too many exit points in a method can destructure your code; so avoid using them casually. A well-designed method has well-defined exit points.

To review: A **void** method can return in one of two ways—its closing curly brace is reached, or a **return** statement is executed.

Returning a Value

Although methods with a return type of **void** are not rare, most methods will return a value. In fact, the ability to return a value is one of the most useful features of a method. You have already seen one example of a return value: when we used the **sqrt()** function to obtain a square root.

Return values are used for a variety of purposes in programming. In some cases, such as with **sqrt()**, the return value contains the outcome of some calculation. In other cases, the return value may simply indicate success or failure. In still others, it may contain a status code. Whatever the purpose, using method return values is an integral part of Java programming.

Methods return a value to the calling routine using this form of **return**:

```
return value;
```

Here, *value* is the value returned. This form of **return** can be used only with methods that have a non-**void** return type. Furthermore, a non-**void** method *must* return a value by using this form of **return**.

You can use a return value to improve the implementation of **range()**. Instead of displaying the range, a better approach is to have **range()** compute the range and return this value. Among the advantages to this approach is that you can use the value for other calculations. The following example modifies **range()** to return the range rather than displaying it.

```
// Use a return value.

class Vehicle {
    int passengers; // number of passengers
    int fuelcap;    // fuel capacity in gallons
    int mpg;        // fuel consumption in miles per gallon

    // Return the range.
    int range() {
        return mpg * fuelcap; ← Return the range for a given vehicle.
    }
}

class RetMeth {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        Vehicle sportscar = new Vehicle();

        int range1, range2;

        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;

        // assign values to fields in sportscar
        sportscar.passengers = 2;
        sportscar.fuelcap = 14;
        sportscar.mpg = 12;

        // get the ranges
        range1 = minivan.range();
        range2 = sportscar.range();
    }
}
```

Assign the value returned to a variable.

```
System.out.println("Minivan can carry " + minivan.passengers +
    " with range of " + range1 + " Miles");

System.out.println("Sportscar can carry " + sportscar.passengers +
    " with range of " + range2 + " miles");

    }
}
```

The output is shown here:

```
Minivan can carry 7 with range of 336 Miles
Sportscar can carry 2 with range of 168 miles
```

In the program, notice that when **range()** is called, it is put on the right side of an assignment statement. On the left is a variable that will receive the value returned by **range()**. Thus, after

```
range1 = minivan.range();
```

executes, the range of the **minivan** object is stored in **range1**.

Notice that **range()** now has a return type of **int**. This means that it will return an integer value to the caller. The return type of a method is important because the type of data returned by a method must be compatible with the return type specified by the method. Thus, if you want a method to return data of type **double**, its return type must be type **double**.

Although the preceding program is correct, it is not written as efficiently as it could be. Specifically, there is no need for the **range1** or **range2** variables. A call to **range()** can be used in the **println()** statement directly, as shown here:

```
System.out.println("Minivan can carry " + minivan.passengers +
    " with range of " + minivan.range() + " Miles");
```

In this case, when **println()** is executed, **minivan.range()** is called automatically and its value will be passed to **println()**. Furthermore, you can use a call to **range()** whenever the range of a **Vehicle** object is needed. For example, this statement compares the ranges of two vehicles:

```
if (v1.range() > v2.range()) System.out.println("v1 has greater range");
```

Using Parameters

It is possible to pass one or more values to a method when the method is called. Recall that a value passed to a method is called an *argument*. Inside the method, the variable that receives the argument is called a *parameter*. Parameters are declared inside the parentheses that follow the method's name. The parameter declaration syntax is the same as that used for variables. A parameter is within the scope of its method, and aside from its special task of receiving an argument, it acts like any other local variable.

Here is a simple example that uses a parameter. Inside the **ChkNum** class, the method **isEven()** returns **true** if the value that it is passed is even. It returns **false** otherwise. Therefore, **isEven()** has a return type of **boolean**.

// A simple example that uses a parameter.

```
class ChkNum {
    // return true if x is even
    boolean isEven(int x) { ← Here, x is an integer parameter of isEven().
        if((x%2) == 0) return true;
        else return false;
    }
}

class ParmDemo {
    public static void main(String args[]) {
        ChkNum e = new ChkNum();
        if(e.isEven(10)) System.out.println("10 is even.");
        if(e.isEven(9)) System.out.println("9 is even.");
        if(e.isEven(8)) System.out.println("8 is even.");
    }
}
```

Pass arguments to **isEven()**.

Here is the output produced by the program:

```
10 is even.
8 is even.
```

In the program, **isEven()** is called three times, and each time a different value is passed. Let's look at this process closely. First, notice how **isEven()** is called. The argument is specified between the parentheses. When **isEven()** is called the first time, it is passed the value 10. Thus, when **isEven()** begins executing, the parameter **x** receives the value 10. In the second call, 9 is the argument, and **x**, then, has the value 9. In the third call, the argument is 8, which is the value that **x** receives. The point is that the value passed as an argument when **isEven()** is called is the value received by its parameter, **x**.

A method can have more than one parameter. Simply declare each parameter, separating one from the next with a comma. For example, the **Factor** class defines a method called **isFactor()** that determines whether the first parameter is a factor of the second.

```
class Factor {
    boolean isFactor(int a, int b) { ← This method has two parameters.
        if( (b % a) == 0) return true;
        else return false;
    }
}
```



```

    }
}
class IsFact {
    public static void main(String args[]) {
        Factor x = new Factor();
        if(x.isFactor(2, 20)) System.out.println("2 is factor");
        if(x.isFactor(3, 20)) System.out.println("this won't be displayed");
    }
}

```

Pass two arguments
to **isFactor()**.

Notice that when **isFactor()** is called, the arguments are also separated by commas.

When using multiple parameters, each parameter specifies its own type, which can differ from the others. For example, this is perfectly valid:

```

int myMeth(int a, double b, float c) {
// ...
}

```

Adding a Parameterized Method to Vehicle

You can use a parameterized method to add a new feature to the **Vehicle** class: the ability to compute the amount of fuel needed for a given distance. This new method is called **fuelneeded()**. This method takes the number of miles that you want to drive and returns the number of gallons of gas required. The **fuelneeded()** method is defined like this:

```

double fuelneeded(int miles) {
    return (double) miles / mpg;
}

```

Notice that this method returns a value of type **double**. This is useful since the amount of fuel needed for a given distance might not be a whole number. The entire **Vehicle** class that includes **fuelneeded()** is shown here:

```

/*
    Add a parameterized method that computes the
    fuel required for a given distance.
*/

class Vehicle {
    int passengers; // number of passengers
    int fuelcap;    // fuel capacity in gallons
    int mpg;       // fuel consumption in miles per gallon
}

```

```
// Return the range.
int range() {
    return mpg * fuelcap;
}

// Compute fuel needed for a given distance.
double fuelneeded(int miles) {
    return (double) miles / mpg;
}
}

class CompFuel {
    public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        Vehicle sportscar = new Vehicle();
        double gallons;
        int dist = 252;

        // assign values to fields in minivan
        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;

        // assign values to fields in sportscar
        sportscar.passengers = 2;
        sportscar.fuelcap = 14;
        sportscar.mpg = 12;

        gallons = minivan.fuelneeded(dist);

        System.out.println("To go " + dist + " miles minivan needs " +
            gallons + " gallons of fuel.");

        gallons = sportscar.fuelneeded(dist);

        System.out.println("To go " + dist + " miles sportscar needs " +
            gallons + " gallons of fuel.");

    }
}
```

The output from the program is shown here:

```
To go 252 miles minivan needs 12.0 gallons of fuel.
To go 252 miles sportscar needs 21.0 gallons of fuel.
```

Try This 4-1 Creating a Help Class

HelpClassDemo.java

If one were to try to summarize the essence of the class in one sentence, it might be this: a class encapsulates functionality. Of course, sometimes the trick is knowing where one “functionality” ends and another begins. As a general rule, you will want your classes to be the building blocks of your larger application. In order to do this, each class must represent a single functional unit that performs clearly delineated actions. Thus, you will want your classes to be as small as possible—but no smaller! That is, classes that contain extraneous functionality confuse and destructure code, but classes that contain too little functionality are fragmented. What is the balance? It is at this point that the science of programming becomes the *art* of programming. Fortunately, most programmers find that this balancing act becomes easier with experience.

To begin to gain that experience you will convert the help system from Try This 3-3 in the preceding chapter into a Help class. Let’s examine why this is a good idea. First, the help system defines one logical unit. It simply displays the syntax for Java’s control statements. Thus, its functionality is compact and well defined. Second, putting help in a class is an esthetically pleasing approach. Whenever you want to offer the help system to a user, simply instantiate a help-system object. Finally, because help is encapsulated, it can be upgraded or changed without causing unwanted side effects in the programs that use it.

1. Create a new file called **HelpClassDemo.java**. To save you some typing, you might want to copy the file from Try This 3-3, **Help3.java**, into **HelpClassDemo.java**.
2. To convert the help system into a class, you must first determine precisely what constitutes the help system. For example, in **Help3.java**, there is code to display a menu, input the user’s choice, check for a valid response, and display information about the item selected. The program also loops until the letter q is pressed. If you think about it, it is clear that the menu, the check for a valid response, and the display of the information are integral to the help system. How user input is obtained, and whether repeated requests should be processed, are not. Thus, you will create a class that displays the help information, the help menu, and checks for a valid selection. Its methods will be called **helpOn()**, **showMenu()**, and **isValid()**, respectively.
3. Create the **helpOn()** method as shown here:

```
void helpOn(int what) {
    switch(what) {
        case '1':
            System.out.println("The if:\n");
            System.out.println("if(condition) statement;");
            System.out.println("else statement;");
            break;
        case '2':
            System.out.println("The switch:\n");
            System.out.println("switch(expression) {");
```

(continued)

```

        System.out.println(" case constant:");
        System.out.println(" statement sequence");
        System.out.println(" break;");
        System.out.println(" // ...");
        System.out.println("}");
        break;
    case '3':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
    case '4':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
    case '5':
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println(" statement;");
        System.out.println("} while (condition);");
        break;
    case '6':
        System.out.println("The break:\n");
        System.out.println("break; or break label;");
        break;
    case '7':
        System.out.println("The continue:\n");
        System.out.println("continue; or continue label;");
        break;
    }
    System.out.println();
}

```

4. Next, create the **showMenu()** method:

```

void showMenu() {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
    System.out.println(" 4. while");
    System.out.println(" 5. do-while");
    System.out.println(" 6. break");
    System.out.println(" 7. continue\n");
    System.out.print("Choose one (q to quit): ");
}

```

5. Create the `isValid()` method, shown here:

```
boolean isValid(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
}
```

6. Assemble the foregoing methods into the `Help` class, shown here:

```
class Help {
    void helpOn(int what) {
        switch(what) {
            case '1':
                System.out.println("The if:\n");
                System.out.println("if(condition) statement;");
                System.out.println("else statement;");
                break;
            case '2':
                System.out.println("The switch:\n");
                System.out.println("switch(expression) {");
                System.out.println("    case constant:");
                System.out.println("        statement sequence");
                System.out.println("        break;");
                System.out.println("    // ...");
                System.out.println("}");
                break;
            case '3':
                System.out.println("The for:\n");
                System.out.print("for(init; condition; iteration)");
                System.out.println(" statement;");
                break;
            case '4':
                System.out.println("The while:\n");
                System.out.println("while(condition) statement;");
                break;
            case '5':
                System.out.println("The do-while:\n");
                System.out.println("do {");
                System.out.println("    statement;");
                System.out.println("} while (condition);");
                break;
            case '6':
                System.out.println("The break:\n");
                System.out.println("break; or break label;");
                break;
        }
    }
}
```

(continued)

```

        case '7':
            System.out.println("The continue:\n");
            System.out.println("continue; or continue label;");
            break;
    }
    System.out.println();
}

void showMenu() {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
    System.out.println(" 4. while");
    System.out.println(" 5. do-while");
    System.out.println(" 6. break");
    System.out.println(" 7. continue\n");
    System.out.print("Choose one (q to quit): ");
}

boolean isValid(int ch) {
    if(ch < '1' | ch > '7' & ch != 'q') return false;
    else return true;
}
}

```

7. Finally, rewrite the `main()` method from Try This 3-3 so that it uses the new **Help** class. Call this class **HelpClassDemo.java**. The entire listing for **HelpClassDemo.java** is shown here:

```

/*
   Try This 4-1

   Convert the help system from Try This 3-3 into
   a Help class.
*/

class Help {
    void helpOn(int what) {
        switch(what) {
            case '1':
                System.out.println("The if:\n");
                System.out.println("if(condition) statement;");
                System.out.println("else statement;");
                break;

```

```
case '2':
    System.out.println("The switch:\n");
    System.out.println("switch(expression) {");
    System.out.println("  case constant:");
    System.out.println("    statement sequence");
    System.out.println("  break;");
    System.out.println("  // ...");
    System.out.println("}");
    break;
case '3':
    System.out.println("The for:\n");
    System.out.print("for(init; condition; iteration)");
    System.out.println(" statement;");
    break;
case '4':
    System.out.println("The while:\n");
    System.out.println("while(condition) statement;");
    break;
case '5':
    System.out.println("The do-while:\n");
    System.out.println("do {");
    System.out.println("  statement;");
    System.out.println("} while (condition);");
    break;
case '6':
    System.out.println("The break:\n");
    System.out.println("break; or break label;");
    break;
case '7':
    System.out.println("The continue:\n");
    System.out.println("continue; or continue label;");
    break;
}
System.out.println();
}

void showMenu() {
    System.out.println("Help on:");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. for");
    System.out.println(" 4. while");
    System.out.println(" 5. do-while");
    System.out.println(" 6. break");
    System.out.println(" 7. continue\n");
    System.out.print("Choose one (q to quit): ");
}
```

(continued)

```

        boolean isValid(int ch) {
            if(ch < '1' | ch > '7' & ch != 'q') return false;
            else return true;
        }
    }

class HelpClassDemo {
    public static void main(String args[])
        throws java.io.IOException {
        char choice, ignore;
        Help hlpobj = new Help();

        for(;;) {
            do {
                hlpobj.showMenu();

                choice = (char) System.in.read();

                do {
                    ignore = (char) System.in.read();
                } while(ignore != '\n');

            } while( !hlpobj.isValid(choice) );

            if(choice == 'q') break;

            System.out.println("\n");

            hlpobj.helpOn(choice);
        }
    }
}

```

When you try the program, you will find that it is functionally the same as before. The advantage to this approach is that you now have a help system component that can be reused whenever it is needed.

Constructors

In the preceding examples, the instance variables of each **Vehicle** object had to be set manually using a sequence of statements, such as:

```

minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;

```


An approach like this would never be used in professionally written Java code. Aside from being error prone (you might forget to set one of the fields), there is simply a better way to accomplish this task: the constructor.

A *constructor* initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **booleans**, respectively. However, once you define your own constructor, the default constructor is no longer used.

Here is a simple example that uses a constructor:

```
// A simple constructor.

class MyClass {
    int x;

    MyClass() { ←———— This is the constructor for MyClass.
        x = 10;
    }
}

class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();

        System.out.println(t1.x + " " + t2.x);
    }
}
```

In this example, the constructor for **MyClass** is

```
MyClass() {
    x = 10;
}
```

This constructor assigns the instance variable **x** of **MyClass** the value 10. This constructor is called by **new** when an object is created. For example, in the line

```
MyClass t1 = new MyClass();
```

the constructor **MyClass()** is called on the **t1** object, giving **t1.x** the value 10. The same is true for **t2**. After construction, **t2.x** has the value 10. Thus, the output from the program is

```
10 10
```

Parameterized Constructors

In the preceding example, a parameter-less constructor was used. Although this is fine for some situations, most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name. For example, here, **MyClass** is given a parameterized constructor:

```
// A parameterized constructor.

class MyClass {
    int x;

    MyClass(int i) { ←————— This constructor has a parameter.
        x = i;
    }
}

class ParmConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass(10);
        MyClass t2 = new MyClass(88);

        System.out.println(t1.x + " " + t2.x);
    }
}
```

The output from this program is shown here:

```
10 88
```

In this version of the program, the **MyClass()** constructor defines one parameter called **i**, which is used to initialize the instance variable, **x**. Thus, when the line

```
MyClass t1 = new MyClass(10);
```

executes, the value 10 is passed to **i**, which is then assigned to **x**.

Adding a Constructor to the Vehicle Class

We can improve the **Vehicle** class by adding a constructor that automatically initializes the **passengers**, **fuelcap**, and **mpg** fields when an object is constructed. Pay special attention to how **Vehicle** objects are created.

```
// Add a constructor.

class Vehicle {
    int passengers; // number of passengers
    int fuelcap;    // fuel capacity in gallons
    int mpg;        // fuel consumption in miles per gallon
}

© 2014 McGraw-Hill Education
```

```
// This is a constructor for Vehicle.
Vehicle(int p, int f, int m) { ← Constructor for Vehicle.
    passengers = p;
    fuelcap = f;
    mpg = m;
}

// Return the range.
int range() {
    return mpg * fuelcap;
}

// Compute fuel needed for a given distance.
double fuelneeded(int miles) {
    return (double) miles / mpg;
}
}

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

        // construct complete vehicles
        Vehicle minivan = new Vehicle(7, 16, 21);
        Vehicle sportscar = new Vehicle(2, 14, 12);
        double gallons;
        int dist = 252;

        gallons = minivan.fuelneeded(dist);

        System.out.println("To go " + dist + " miles minivan needs " +
            gallons + " gallons of fuel.");

        gallons = sportscar.fuelneeded(dist);

        System.out.println("To go " + dist + " miles sportscar needs " +
            gallons + " gallons of fuel.");

    }
}
```

Both **minivan** and **sportscar** are initialized by the **Vehicle()** constructor when they are created. Each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Vehicle minivan = new Vehicle(7, 16, 21);
```

the values 7, 16, and 21 are passed to the **Vehicle()** constructor when **new** creates the object. Thus, **minivan**'s copy of **passengers**, **fuelcap**, and **mpg** will contain the values 7, 16, and 21, respectively. The output from this program is the same as the previous version.

The new Operator Revisited

Now that you know more about classes and their constructors, let's take a closer look at the **new** operator. In the context of an assignment, the **new** operator has this general form:

```
class-var = new class-name(arg-list);
```

Here, *class-var* is a variable of the class type being created. The *class-name* is the name of the class that is being instantiated. The class name followed by a parenthesized argument list (which can be empty) specifies the constructor for the class. If a class does not define its own constructor, **new** will use the default constructor supplied by Java. Thus, **new** can be used to create an object of any class type. The **new** operator returns a reference to the newly created object, which (in this case) is assigned to *class-var*.

Since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn about exceptions in Chapter 9.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Garbage Collection and Finalizers

As you have seen, objects are dynamically allocated from a pool of free memory by using the **new** operator. As explained, memory is not infinite, and the free memory can be exhausted. Thus, it is possible for **new** to fail because there is insufficient free memory to create the desired object. For this reason, a key component of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation. In some programming languages, the release of previously allocated memory is handled manually. However, Java uses a different, more trouble-free approach: *garbage collection*.

Java's garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention. It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released. This recycled memory can then be used for a subsequent allocation.

Ask the Expert

Q: Why don't I need to use **new** for variables of the primitive types, such as `int` or `float`?

A: Java's primitive types are not implemented as objects. Rather, because of efficiency concerns, they are implemented as "normal" variables. A variable of a primitive type actually contains the value that you have given it. As explained, object variables are references to the object. This layer of indirection (and other object features) adds overhead to an object that is avoided by a primitive type.

Garbage collection occurs only sporadically during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. For efficiency, the garbage collector will usually run only when two conditions are met: there are objects to recycle, and there is a need to recycle them. Remember, garbage collection takes time, so the Java run-time system does it only when it is appropriate. Thus, you can't know precisely when garbage collection will take place.

The `finalize()` Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called `finalize()`, and it can be used to ensure that an object terminates cleanly. For example, you might use `finalize()` to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the `finalize()` method. The Java run-time system calls that method whenever it is about to recycle an object of that class. Inside the `finalize()` method you will specify those actions that must be performed before an object is destroyed.

The `finalize()` method has this general form:

```
protected void finalize()  
{  
    // finalization code here  
}
```

Here, the keyword `protected` is a specifier that limits access to `finalize()`. This and the other access specifiers are explained in Chapter 6.

It is important to understand that `finalize()` is called just before garbage collection. It is not called when an object goes out of scope, for example. This means that you cannot know when—or even if—`finalize()` will be executed. For example, if your program ends before garbage collection occurs, `finalize()` will not execute. Therefore, it should be used as a “backup” procedure to ensure the proper handling of some resource, or for special-use applications, not as the means that your program uses in its normal operation. In short, `finalize()` is a specialized method that is seldom needed by most programs.

Ask the Expert

Q: I know that C++ defines things called *destructors*, which are automatically executed when an object is destroyed. Is `finalize()` similar to a destructor?

A: Java does not have destructors. Although it is true that the `finalize()` method approximates the function of a destructor, it is not the same. For example, a C++ destructor is always called just before an object goes out of scope, but you can't know when `finalize()` will be called for any specific object. Frankly, because of Java's use of garbage collection, there is little need for a destructor.

Try This 4-2 Demonstrate Garbage Collection and Finalization

Finalize.java

Because garbage collection runs sporadically in the background, it is not trivial to demonstrate it. However, one way it can be done is through the use of the **finalize()** method. Recall that **finalize()** is called when an object is about to be recycled. As explained, objects are not necessarily recycled as soon as they are no longer needed. Instead, the garbage collector waits until it can perform its collection efficiently, usually when there are many unused objects. Thus, to demonstrate garbage collection via the **finalize()** method, you often need to create and destroy a large number of objects—and this is precisely what you will do in this project.

1. Create a new file called **Finalize.java**.
2. Create the **FDemo** class shown here:

```
class FDemo {
    int x;

    FDemo(int i) {
        x = i;
    }

    // called when object is recycled
    protected void finalize() {
        System.out.println("Finalizing " + x);
    }

    // generates an object that is immediately destroyed
    void generator(int i) {
        FDemo o = new FDemo(i);
    }
}
```

The constructor sets the instance variable **x** to a known value. In this example, **x** is used as an object ID. The **finalize()** method displays the value of **x** when an object is recycled. Of special interest is **generator()**. This method creates and then promptly discards an **FDemo** object. You will see how this is used in the next step.

3. Create the **Finalize** class, shown here:

```
class Finalize {
    public static void main(String args[]) {
        int count;

        FDemo ob = new FDemo(0);
```

```

    /* Now, generate a large number of objects. At
       some point, garbage collection will occur.
       Note: you might need to increase the number
       of objects generated in order to force
       garbage collection. */

    for(count=1; count < 100000; count++)
        ob.generator(count);
    }
}

```

This class creates an initial **FDemo** object called **ob**. Then, using **ob**, it creates 100,000 objects by calling **generator()** on **ob**. This has the net effect of creating and discarding 100,000 objects. At various points in the middle of this process, garbage collection will take place. Precisely how often or when depends upon several factors, such as the initial amount of free memory and the operating system. However, at some point, you will start to see the messages generated by **finalize()**. If you don't see the messages, try increasing the number of objects being generated by raising the count in the **for** loop.

4. Here is the entire **Finalize.java** program:

```

/*
   Try This 4-2

   Demonstrate garbage collection and the finalize() method.
*/

class FDemo {
    int x;

    FDemo(int i) {
        x = i;
    }

    // called when object is recycled
    protected void finalize() {
        System.out.println("Finalizing " + x);
    }

    // generates an object that is immediately destroyed
    void generator(int i) {
        FDemo o = new FDemo(i);
    }
}

class Finalize {
    public static void main(String args[]) {
        int count;

```

(continued)

```
FDemo ob = new FDemo(0);

/* Now, generate a large number of objects. At
   some point, garbage collection will occur.
   Note: you might need to increase the number
   of objects generated in order to force
   garbage collection. */

for(count=1; count < 100000; count++)
    ob.generator(count);
}
}
```

The this Keyword

Before concluding this chapter it is necessary to introduce **this**. When a method is called, it is automatically passed an implicit argument that is a reference to the invoking object (that is, the object on which the method is called). This reference is called **this**. To understand **this**, first consider a program that creates a class called **Pwr** that computes the result of a number raised to some integer power:

```
class Pwr {
    double b;
    int e;
    double val;

    Pwr(double base, int exp) {
        b = base;
        e = exp;

        val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) val = val * base;
    }

    double get_pwr() {
        return val;
    }
}

class DemoPwr {
    public static void main(String args[]) {
        Pwr x = new Pwr(4.0, 2);
        Pwr y = new Pwr(2.5, 1);
        Pwr z = new Pwr(5.7, 0);
    }
}
```



```
System.out.println(x.b + " raised to the " + x.e +
    " power is " + x.get_pwr());
System.out.println(y.b + " raised to the " + y.e +
    " power is " + y.get_pwr());
System.out.println(z.b + " raised to the " + z.e +
    " power is " + z.get_pwr());
}
}
```

As you know, within a method, the other members of a class can be accessed directly, without any object or class qualification. Thus, inside `get_pwr()`, the statement

```
return val;
```

means that the copy of `val` associated with the invoking object will be returned. However, the same statement can also be written like this:

```
return this.val;
```

Here, `this` refers to the object on which `get_pwr()` was called. Thus, `this.val` refers to that object's copy of `val`. For example, if `get_pwr()` had been invoked on `x`, then `this` in the preceding statement would have been referring to `x`. Writing the statement without using `this` is really just shorthand.

Here is the entire `Pwr` class written using the `this` reference:

```
class Pwr {
    double b;
    int e;
    double val;

    Pwr(double base, int exp) {
        this.b = base;
        this.e = exp;

        this.val = 1;
        if(exp==0) return;
        for( ; exp>0; exp--) this.val = this.val * base;
    }

    double get_pwr() {
        return this.val;
    }
}
```

Actually, no Java programmer would write `Pwr` as just shown because nothing is gained, and the standard form is easier. However, `this` has some important uses. For example, the Java syntax permits the name of a parameter or a local variable to be the same as the name of an instance variable. When this happens, the local name *hides* the instance variable. You can

gain access to the hidden instance variable by referring to it through **this**. For example, the following is a syntactically valid way to write the **Pwr()** constructor.

```
Pwr(double b, int e) {
    this.b = b;
    this.e = e;
    val = 1;
    if(e==0) return;
    for( ; e>0; e--) val = val * b;
}
```

This refers to the **b** instance variable, not the parameter.

In this version, the names of the parameters are the same as the names of the instance variables, thus hiding them. However, **this** is used to “uncover” the instance variables.



Chapter 4 Self Test

1. What is the difference between a class and an object?
2. How is a class defined?
3. What does each object have its own copy of?
4. Using two separate statements, show how to declare an object called **counter** of a class called **MyCounter**.
5. Show how a method called **myMeth()** is declared if it has a return type of **double** and has two **int** parameters called **a** and **b**.
6. How must a method return if it returns a value?
7. What name does a constructor have?
8. What does **new** do?
9. What is garbage collection, and how does it work? What is **finalize()**?
10. What is **this**?
11. Can a constructor have one or more parameters?
12. If a method returns no value, what must its return type be?

Mastering Lambdas: **Java** Programming in a Multicore World

Best Practices for Using Lambda Expressions



Maurice Naftalin

Oracle
Press™



CHAPTER 1

Taking Java to the Next Level

Java 8 contains the biggest changes to Java since version 1.0 shipped in 1996, combining coordinated changes to the language, the libraries, and the virtual machine. It promises to alter the way we think about the execution of Java programs and to make the language fit for use in a world, soon to arrive, of massively parallel hardware. Yet for such an important innovation, the actual changes to the language seem quite minor. What is it about these apparently minor modifications that will make such a big difference? And why should we change a programming model that has served us so well throughout the lifetime of Java, and indeed for much longer before that? In this chapter we will explore some of the limitations of that model and see how the lambda-related features of Java 8 will enable Java to evolve to meet the challenges of a new generation of hardware architectures.

The Trouble with Iteration

Let's start with code that simply iterates over a collection of mutable objects, calling a single method on each of them. The following code fragment constructs a collection of `java.awt.Point` objects (`Point` is a conveniently simple library class, consisting only of a pair (x,y) of coordinates). Our code then iterates over the collection, translating (i.e., moving) each `Point` by a distance of 1 on both the x and y axes.

```
List<Point> pointList = Arrays.asList(new Point(1, 2), new Point(2, 3));  
for (Point p : pointList) {  
    p.translate(1, 1);  
}
```

Before Java 5 introduced the for-each loop, we would have written the loop like this:

```
for (Iterator pointItr = pointList.iterator(); pointItr.hasNext(); ) {  
    ((Point) pointItr.next()).translate(1, 1);  
}
```

Here we are asking `pointList` to create an `Iterator` object on our behalf, and we are then using that object to access the elements of `pointList` in turn. This version is still relevant, because today this is the code that the Java compiler generates to implement the for-each loop. Its key aspect for us is that the order of access to the elements of `pointList` is controlled by the `Iterator`—there is nothing that we can do to change it. The `Iterator` for an `ArrayList`, for example, will return the elements of the list in sequential order.

Why is this problematic? After all, when the Java Collections Framework was designed in 1998, it seemed perfectly reasonable to dictate the access order of list elements in this way. What has changed since then?

The answer, of course, lies in how hardware has been evolving. Workstations and servers have been equipped with multiple processors for a long time, but between the design of the Java Collections Framework in 1998 and the appearance of the first dual-core processors in personal computers in 2005, a revolution had taken place in chip design. A 40-year trend of exponentially increasing processor speed had been halted by inescapable physical facts: signal leakage, inadequate heat dissipation, and the hard truth that, even at the speed of light, information cannot travel fast enough across a chip for further processor speed increases.

But clock speed limitations notwithstanding, the density of chip components continued to increase. So, since it wasn't possible to offer a 6 GHz core, the chip vendors instead began to offer dual-core processors, each core running at 3 GHz. This trend has continued, with currently no end in sight; at the time of writing (early 2014) quad-core processors have become mainstream, eight-core processors are appearing in the commodity hardware market, and specialist servers have long been available with dozens of cores per processor. The direction is clear, and any programming model that doesn't adapt to it will fail in the face of competition from models that do adapt. Adaptation would mean providing developers with an accessible way of making use of the processing power of multiple cores by distributing tasks over them to be executed in parallel.¹ Failing to adapt, on the other hand, would mean that Java programs, bound by default to a single core, would run at a progressively greater speed disadvantage compared to programs in languages that had found ways to assist users in easily parallelizing their code.

The need for change is shown by the code at the start of this section, which could only access list elements one at a time in the order dictated by the iterator. Collection processing is not the only processor-intensive task that programs have to carry out, but it is one of the most important. The model of iteration embodied in Java's loop constructs forces collection element processing into a serial straitjacket, and that is a serious problem at a time when the most pressing requirement for runtimes—at least as far as performance is concerned—is precisely the opposite: to distribute processing over multiple cores.

Internal Iteration


The intrusiveness of the serial model of iteration becomes obvious when we imagine imposing it on a real-world situation. If someone were to ask you to mail some letters with the instruction “repeat the following action: if you have any more letters, take the next one in alphabetical order of addressee's surname and put it in the mailbox,” your kindest thought would probably be that they have overspecified the

¹ The distribution of a processing task over multiple processors is often called *parallelization*. Even if we dislike this word, it's a useful shorthand that will sometimes make explanations shorter and more readable.

4 Mastering Lambdas: Java Programming in a Multicore World

task. You would know that ordering doesn't matter in this task, and neither does the mode—sequential or parallel—of execution, yet it would seem you aren't allowed to ignore them. In this situation you might feel some sympathy with a collection forced by external iteration to process elements serially and in a fixed order when much better strategies may be available.

In reality, all you need to know for that real-world task is that every letter in a bundle needs mailing; exactly how to do that should be up to you. And in the same way, we ought to be able to tell collections *what* should be done to each element they contain, rather than specifying *how*, as external iteration does. If we could do that, what would the code look like? Collections would have to expose a method accepting the “what,” namely the task to be executed on each element; an obvious name for this method is `forEach`. With it, we can imagine replacing the iterative code from the start of this section with this:²

```
 pointList.forEach(/*translate the point by (1,1)*/);
```

This is called *internal iteration* because, although the explicit iterative code is no longer obvious, iteration is still taking place internally to the collection itself.

The change from external to internal iteration may seem a small one, simply a matter of moving the work of iteration across the client-library boundary. But the consequences are not small. The parallelization work that we require can now be defined in the collection class instead of repeatedly in every client method that must iterate over the collection. Moreover, the implementation is free to use additional techniques such as laziness and out-of-order execution—and, indeed, others yet to be discovered—to get to the answer faster.

So internal iteration is a necessity for a programming model to allow collection library writers the freedom to choose, for each collection, the best way of implementing bulk processing. But what is to replace the comment in the call of `forEach`—how can the collection's methods be told what task is to be executed on each element?

The Command Pattern

There's no need to go outside traditional Java mechanisms to find an answer to this question. For example, we routinely create `Runnable` instances and pass them as arguments. If you think of a `Runnable` as an object representing a task to be executed when its `run` method is called, you can see that what we now require is very similar. For another example, the Swing framework requires the developer to define the different tasks that will be executed in response to different events on the

²Our imagination will need to be quite powerful, since (as of Java 7) `java.util.List` has no `forEach` method and, as an interface, cannot have one added. In Chapter 5 we'll see how Java 8 overcomes this problem.

user interface. If you are familiar with classical design patterns [GoF], you will recognize this loose description of the Command Pattern.

In the case we're considering, what command is needed? Our starting point was a call to the `translate` method of every `Point` in a `List`. So for this example it appears that `forEach` should accept as its argument an object exposing a method that will call `translate` on each element of the list. If we make this object an instance of a more general interface, `PointAction` say, then we can define different implementations of `PointAction` for different actions that we want to have iteratively executed on `Point` collections:

```
public interface PointAction {
    void doForPoint(Point p);
}
```

Right now, the implementation we want is

```
class TranslateByOne implements PointAction {
    public void doForPoint(Point p) {
        p.translate(1, 1);
    }
}
```

Now we can sketch a naïve implementation of `forEach`:

```
public class PointArrayList extends ArrayList<Point> {
    public void forEach(PointAction t) {
        for (Point p : this) {
            t.doForPoint(p);
        }
    }
}
```

and if we make `pointList` an instance of `PointArrayList`, our goal of internal iteration is achieved with this client code:

```
pointList.forEach(new TranslateByOne());
```

Of course, this toy code is absurdly specialized; we aren't really going to write a new interface for every element type we need to work with. Fortunately, we don't need to; there is nothing special about the names `PointAction` and `doForPoint`; if we simply replace them consistently with other names, nothing changes. In the Java 8 collections library they are called `Consumer` and `accept`. So our `PointAction` interface becomes:

```
public interface Consumer<T> {
    void accept(T t);
}
```


6 Mastering Lambdas: Java Programming in a Multicore World

Parameterizing the type of the interface allows us to dispense with the specialized `ArrayList` subclass and instead add the method `forEach` directly to the class itself, as is done by inheritance in Java 8. This method takes a `java.util.function.Consumer`, which will receive and process each element of the collection.

```
public class ArrayList<E> {  
    ...  
    public void forEach(Consumer c) {  
        for (E e : this) {  
            c.accept(e);  
        }  
    }  
}
```

Applying these changes to the client code, we get

```
class TranslateByOne implements Consumer<Point> {  
    public void accept(Point p) {  
        p.translate(1, 1);  
    }  
}  
...  
pointList.forEach(new TranslateByOne());
```

You may think that this code is still pretty clumsy. But notice that the clumsiness is now concentrated in the representation of each command by an instance of a class. In many cases, this is overkill. In the present case, for example, all that `forEach` really needs is the *behavior* of the single method `accept` of the object that has been supplied to it. State and all the other apparatus that make up the object are included only because method arguments in Java, if not primitives, have to be object references. But we have always needed to specify this apparatus—until now.

Lambda Expressions


The code that concluded the previous section is not idiomatic Java for the command pattern. When, as in this case, a class is both small and unlikely to be reused, a more common usage is to define an *anonymous inner class*:

```
pointList.forEach(new Consumer<Point> {  
    public void accept(Point p) {  
        p.translate(1, 1);  
    }  
});
```


Experienced Java developers are so accustomed to seeing code like this that we have often forgotten how we felt when we first encountered it. Common first reactions

to the syntax for anonymous inner classes used in this way are that it is ugly, verbose, and difficult to understand quickly, even though all it is doing really is saying “do this for each element.” You don’t have to agree completely with these judgements to accept that any attempt to persuade developers to rely on this idiom for every collection operation is unlikely to be very successful. And this, at last, is our cue for the introduction of lambda expressions.³

To reduce the verbosity of this call, we should try to identify those places where we are supplying information that the compiler could instead infer from the context. One such piece of information is the name of the interface being implemented by the anonymous inner class. It’s enough for the compiler to know that the declared type of the parameter to `forEach` is `Consumer<T>`; that is sufficient information to allow the supplied argument to be checked for type compatibility. Let’s de-emphasize the code that the compiler can infer:

```
 pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

Secondly, what about the name of the method being overridden—in this case, `accept`? There’s no way that the compiler can infer that in general. But there is an effective workaround: we can apply a rule that for any object to be used in the abbreviated form that we are developing, it must implement an interface, like `Consumer`, that has only a *single* abstract⁴ method (this is called a *functional interface*, or sometimes a *SAM* interface). That gives the compiler a way to choose the correct method without ambiguity. Again let’s de-emphasize the code that can be inferred in this way:

```
 pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```


³People are often curious about the origin of the name. The idea of lambda expressions comes from a model of computation developed in the 1930s by the American mathematician Alonzo Church, in which the Greek letter λ (lambda) represents functional abstraction. But why that particular letter? Church seems to have liked to tease: asked about his choice, his usual explanation involved accidents of typesetting, but in later years he had an alternative answer: “Eeny, meeny, miny, moe.”

⁴This qualification may seem unnecessary if you believe that interfaces can contain only abstract methods. In fact, all interfaces already contain concrete methods inherited from `Object`, and from Java 8 onward may contain user-defined concrete methods also (see Chapter 5).

8 Mastering Lambdas: Java Programming in a Multicore World


Finally, the instantiated type of `Consumer` can often be inferred from the context, in this case from the fact that when the `forEach` method calls `accept`, it supplies it with an element of `pointList`, previously declared as a `List<Point>`. That identifies the type parameter to `Consumer` as `Point`, allowing us to omit the explicit type declaration of the argument to `accept`.

This is what's left when we de-emphasize this last component of the `forEach` call:

```
 pointList.forEach(new Consumer<Point>() {  
    public void accept(Point p) {  
        p.translate(1, 1);  
    }  
});
```

The argument to `forEach` represents an object, implementing the interface (`Consumer`) required by `forEach`, such that when `accept` (`Consumer`'s only abstract method) is called for a `pointList` element `p`, the effect will be to call `p.translate(1, 1)`.

Some extra syntax ("`->`") is required to separate the *parameter list* from the *expression body*. With that addition, we finally get the simple form for a lambda expression. Here it is, being used in internal iteration:

```
 pointList.forEach(p -> p.translate(1, 1));
```

If you are unused to reading lambda expressions, you may find it helpful for the moment to continue to think of them as an abbreviation for a method declaration, mentally mapping the parameter list of the lambda to that of the imaginary method, and its body (often preceded by an added `return`) to the method body. In the next chapter, we will see that it is going to be necessary to vary the simple syntax in the preceding example for lambda expressions with multiple parameters and with more elaborate bodies, and in cases where the compiler cannot infer parameter types. But if you have followed the reasoning that brought us to this point, you should have a basic understanding of the motivation for the introduction of lambda expressions and of the form that they have taken.

This section has covered a lot of ground. Let's summarize: we began by considering the adaptations that our programming model need to make in order to accommodate the requirements of changing hardware architectures; this brought us to a review of processing of collection elements, which in turn made us aware of the need to have a concise way of defining behavior for collections to execute; finally, paring away the excess text from anonymous inner class definitions brought us to a simple syntax for lambda expressions. In the remaining sections of this chapter, we will look at some of the new idioms that lambda expressions make possible. We will see that bulk processing of collection elements can be written in a much more expressive style, that these changes in idiom make it much easier for library writers to incorporate parallel algorithms to take advantage of new hardware architectures, and finally that

emphasizing functional behavior can improve the design of APIs. It's an impressive list of achievements for such an innocuous-looking change!

From Collections to Streams

Let's extend the example of the previous section a little. In real-life programs, it's common to process collections in a number of stages: a collection is iteratively processed to produce a new collection, which in turn is iteratively processed, and so on. We'll model this in our toy example by starting with a collection of `Integer` instances, then using an arbitrary transformation to produce a collection of `Point` instances, and finally finding the maximum among the distances of each `Point` from the origin.

```
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i))
}
double maxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
```

This is idiomatic Java—most developers have seen many examples of code in this pattern—but if we look at it with fresh eyes, some unpleasant features stand out at once. Firstly, it is very verbose, taking nine lines of code to carry out only three operations. Secondly, the collection `pointList`, required only as intermediate storage, is an overhead on the operation of the program; if the intermediate storage is very large, creating it would at best add to garbage collection overheads, and at worst would exhaust available heap space. Thirdly, there is an implicit assumption, difficult to spot, that the minimum value of an empty list is `Double.MIN_VALUE`. But the worst aspect of all is the gap between the developer's intentions and the way that they are expressed in code. To understand this program you have to work out what it's doing, then guess the developer's intention (or, if you're very fortunate, read the comments), and only then check its correctness by matching the operation of the program to the informal specification you deduced.⁵ All this work is slow and error-prone—indeed, the very purpose of a high-level language is supposed to be to minimize it by supporting code that is as close as possible to the developer's mental model. So how do we close the gap?

⁵The situation is better than it used to be. Some of us are old enough to remember how much of this kind of work was involved in writing big programs in assembler (*really* low-level languages, not far removed from machine code). Programming languages have become much more expressive since then, but there is still plenty of room for progress.

Let's restate the problem specification:

"Apply a transformation to each one of a collection of `Integer` instances to produce a `Point`, then find the greatest distance of any of these `Points` from the origin."

If we de-emphasize the parts of the preceding code that do not correspond to the elements of this informal specification, we see what a poor match there is between code and problem specification. Omitting the first line, in which the list `intList` is initially created, we get:

```

List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 3));
}
double maxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
    
```

This suggests a new, data-oriented way of looking at the program: we can follow the progress of a single value from the source collection, viewing it as being transformed first from an `Integer` to a `Point` and secondly from a `Point` to a `double`. Both of these transformations can take place in isolation, without any reference to the other values being processed—exactly the requirement for parallelization. Only with the third step, finding the greatest distance, is it necessary for the values to interact (and even then, there are techniques for efficiently computing this in parallel).

This data-oriented view can be represented diagrammatically as in Figure 1-1. In this figure it is clear that the rectangular boxes represent operations. The connecting lines represent something new, a way of delivering a sequence of values to an operation. This is different from any kind of collection, because at a given moment the values to be delivered by a connector may not all have been generated yet. These value sequences are called *streams*. Streams differ from collections in that they provide an ordered sequence of values without providing any storage for those values; they are just a means for expressing bulk data operations. In the Java 8 collections API, streams are represented by interfaces—`Stream` for

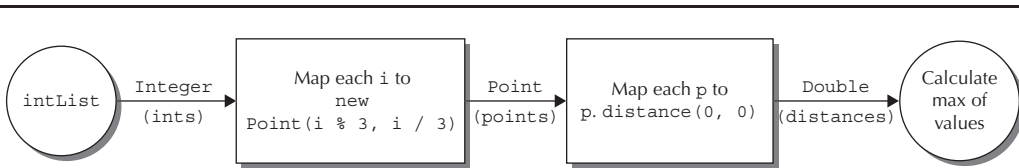




FIGURE 1-1. Composing streams into a data pipeline

reference values, and `IntStream`, `LongStream`, and `DoubleStream` for streams of primitive values—in the package `java.util.stream`.


In this view, the operations represented by the boxes in Figure 1-1 are operations on streams. The boxes in this figure represent two applications of an operation called `map`; it transforms each stream element using a systematic rule. Looking at `map` alone, we might think that we were dealing with operations on individual stream elements. But we will soon meet other stream operations that can reorder, drop, or even insert values; each of these operations can be described as taking a stream and transforming it in some way. Each rectangular box represents an *intermediate operation*, one that is not only defined on a stream but that returns a stream as its output, as well. For example, assuming for a moment that a stream `ints` forms the input to the first operation, the transformations made by the intermediate operations of Figure 1-1 can be represented in code as:

```
 Stream<Point> points = ints.map(i -> new Point(i % 3, i / 3));
DoubleStream distances = points.mapToDouble(p -> p.distance(0, 0));
```


The circle at the end of the pipeline represents the *terminal operation* `max`. Terminal operations take a stream and, like `max`, return a single value (or nothing, represented by an empty `Optional`, if the stream is empty):

```
 OptionalDouble maxDistance = distances.max();
```

Pipelines like that in Figure 1-1 have a beginning, a middle, and an end. We have seen the operations that defined the middle and the end; what about the beginning? The values flowing into streams can be supplied by a variety of sources—collections, arrays, or generating functions. In practice, a common use case will be the transformation of a collection into a stream, as here. Java 8 collections expose a new method `stream()` for this purpose, so the start of the pipeline can be represented as:

```
 Stream<Integer> ints = intList.stream();
```

and the complete code with which this section began has become:

```
 OptionalDouble maxDistance = intList.stream()
    .map(i -> new Point(i % 3, i / 3))
    .mapToDouble(p -> p.distance(0, 0))
    .max();
```

This style, often called *fluent* because “the code flows,” is unfamiliar in the context of collection processing and may seem initially difficult to read in this context, but compared to the successive iterations in the code that introduced this section, it provides a nice balance of conciseness with a close correspondence to the problem statement: “map each integer in the source `intList` to a corresponding `Point`, map each `Point` in the resulting list to its distance from the origin, then find the maximum of the resulting values.” And, as a bonus, the performance overhead of creating and managing intermediate collections has disappeared as well.

From Sequential to Parallel

This chapter began with the assertion that Java now needs to support parallel processing of collections, and that lambdas are an essential step in providing this support. We've come most of the way by seeing how lambdas make it easy for client code developers to make use of internal iteration. The last step is to see how internal iteration of the collection classes actually implements parallelism. It's useful to know the principles of how this will work, although you don't need them for everyday use—the complexity of the implementations is well hidden from developers of client code.

Independent execution on multiple cores is accomplished by assigning a different thread to each core, each thread executing a subtask of the work to be done—in this case a subset of the collection elements to be processed. For example, given a four-core processor and a list of eight elements (in all practical cases there will be many more elements to be processed than cores available to process them), a program might define a `solve` algorithm to break the task down for parallel execution in the following way:

```

if the task list contains more than two elements {
    leftTask = task.getLeftHalf()
    rightTask = task.getRightHalf()
    doInParallel {
        leftResult = leftTask.solve()
        rightResult = rightTask.solve()
    }
    result = combine(leftResult, rightResult)
} else {
    result = task.solveSequentially()
}

```

The preceding pseudocode is a highly simplified description of parallel processing using a list specialization of the pattern of *recursive decomposition*—recursively splitting large tasks into smaller ones, to be executed in parallel, until the subtasks are “small enough” to be executed in serial. Implementing recursive decomposition requires knowing how to split tasks in this way, how to execute sufficiently small ones without further splitting, and how to then combine the results of these smaller executions. The technique for splitting depends on the source of the data; in this case, splitting a list has an obvious implementation. Combining the results of subtasks is often achieved by applying the pipeline terminal operation to them; for the example of this chapter, it involves taking the maximum of two subtask results.

The Java concurrent utility `ForkJoinPool` uses this pattern, allocating threads from its pool to new subtasks rather than creating new ones. Clearly reimplementing this pattern is far more coding work than can realistically be expected of developers every time a collection is to be processed. This is library work—or it certainly should be!

In this case the library class is the collection; from Java 8 onwards the collections library classes will be able to use `ForkJoinPool` in this way, so that

client developers can put parallelization, essentially a performance issue, to the back of their minds and get on with solving business problems. For our current example, the only change necessary to the client code is shown *italicized*:

```
OptionalDouble maxDistance = intList.parallelStream()
    .map(i -> new Point(i % 3, i / 3))
    .mapToDouble(p -> p.distance(0, 0))
    .max();
```

This illustrates what is meant by the slogan for the introduction of parallelism in Java 8: *explicit but unobtrusive*. Parallel execution is achieved by breaking the initial list of `Integer` values down recursively as described in the preceding example until the sublists are small enough, then executing the entire pipeline serially, and finally combining the results with `max`. Figure 1-2 shows the case of four cores and eight

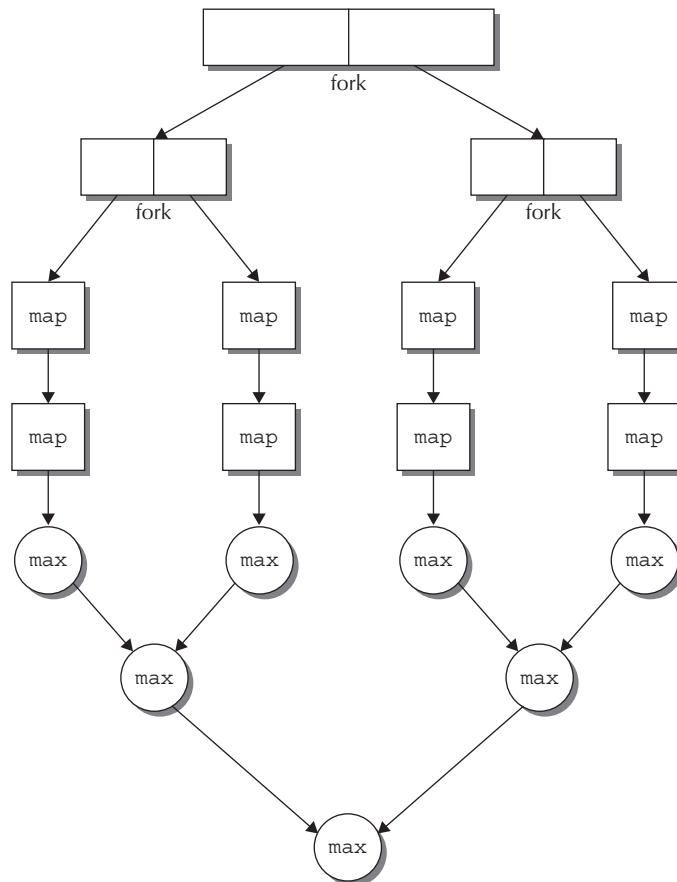


FIGURE 1-2. Recursive decomposition of a list processing task

elements with “small enough” taken to be two elements, but this just for the sake of illustration; in practice the overhead of executing in parallel means that it is only worthwhile for large data sets. We will explore this topic in more detail in Chapter 5.

Unobtrusive parallelism is an example of one of the key themes of Java 8; the API changes that it enables give much greater freedom to library developers. One important way in which they can use it is to explore the many opportunities for performance improvement that are provided by modern—and future—machine architectures.


Composing Behaviors

Earlier in this chapter we saw how functionally similar lambda expressions are to anonymous inner classes. But writing them so differently leads to different ways of thinking about them. Lambda expressions *look* like functions, so it’s natural to ask whether we can make them *behave* like functions. That change of perspective will encourage us to think about working with behaviors rather than objects, and that in turn will lead in the direction of some very different programming idioms and library APIs.

For example, a core operation on functions is *composition*: combining together two functions to make a third, whose effect is the same as applying its two components in succession. Composition is not an idea that arises at all naturally in connection with anonymous inner classes, but in a generalized form it corresponds very well to the construction of traditional object-oriented programs. And just as object-oriented programs are broken down by decomposition, the reverse of composition will work for functions too.

Suppose, for example, that we want to sort a list of `Point` instances in order of their `x` coordinate. The standard Java idiom for a “custom” sort⁶ is to create a `Comparator`:


```

 Comparator<Point> byX = new Comparator<Point>() {
    public int compare(Point p1, Point p2) {
        return Double.compare(p1.getX(), p2.getX());
    }
}

```

Substituting a lambda expression for the anonymous inner class declaration, as described in the previous section, improves the readability of the code:

```

 Comparator<Point> byX = (p1, p2) ->
    Double.compare(p1.getX(), p2.getX());
---(1)

```

⁶Two ways of comparing and sorting objects are standard in the Java platform: a class can have a *natural order*; in this case it implements the interface `Comparable` and so exposes a `compareTo` method that an object can use to compare itself with another. Or a `Comparator` can be created for the purpose, as in this case.

But that doesn't help with another very significant problem: `Comparator` is monolithic. If we wanted to define a `Comparator` that compared on `y` instead of `x` coordinates, we would have to copy the entire declaration, substituting `getY` for `getX` everywhere. Good programming practice should lead us to look for a better solution, and a moment's reflection shows that `Comparator` is actually carrying out two functions—extracting sort keys from its arguments and then comparing those keys. We should be able to improve the code of (1) by building a `Comparator` function parameterized on these two components. We'll now evolve the code to do that. The intermediate stages we'll go through may seem awkward and verbose, but persist: the conclusion will be worthwhile.

To start, let's turn the two concrete component functions that we have into lambda form. We know the type of the functional interface for the key extractor function—`Comparator`—but we also need the type of the functional interface corresponding to the function `p -> p.getX()`. Looking in the package devoted to the declaration of functional interfaces, `java.util.function`, we find the interface `Function`:

```
public interface Function<T, R> {
    public R apply(T t);
}
```

So we can now write the lambda expressions for both key extraction and key comparison:

```
Function<Point, Double> keyExtractor = p -> p.getX();
Comparator<Double> keyComparer = (d1, d2) -> Double.compare(d1, d2);
```

And our version of `Comparator<Point>` can be reassembled from these two smaller functions:

```
Comparator<Point> byX = (p1, p2) ->
    keyComparer.compare(keyExtractor.apply(p1), keyExtractor.apply(p2));
---(2)
```

This matches the form of (1) but represents an important improvement (one that would be much more significant in a larger example): you could plug in any `keyComparer` or `keyExtractor` that had previously been defined. After all, that was the whole purpose of seeking to parameterize the larger function on its smaller components.

But although recasting the `Comparator` in this way has improved its structure, we have lost the conciseness of (1). We can recover that in the special but very common case where `keyComparer` expresses the natural ordering on the extracted keys. Then (2) can be rewritten as:

```
Comparator<Point> byX = (p1, p2) ->
    keyExtractor.apply(p1).compareTo(keyExtractor.apply(p2));
---(3)
```

16 Mastering Lambdas: Java Programming in a Multicore World

And, noticing the importance of this special case, the platform library designers added a static method⁷ `comparing` to the interface `Comparator`; given a key extractor, it creates the corresponding `Comparator`⁸ using natural ordering on the keys. Here is its method declaration (in which generic type parameters have been simplified for this explanation):

```
public static <T,U> Comparator<T> comparing(Function<T,U> keyExtractor) {  
    return (c1, c2) ->  
        keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));  
}
```

Using that method allows us to write instead of (3) (assuming a static import declaration of `Comparators.comparing`):

```
Comparator<Point> byX = comparing(p -> p.getX());
```

 ---(4)

In comparison to (1), (4) is a big improvement: more concise and more immediately understandable because it isolates and lays emphasis on the important element, the key extractor, in a way that is possible only because `comparing` accepts a simple behavior and uses it to build a more complex one from it.

To see the improvement in action, imagine that our problem changes slightly so that instead of finding the single point that is furthest from the origin, we decide to print all the points in ascending order of their distance. It is straightforward to capture the necessary ordering:

```
Comparator<Point> byDistance = comparing(p -> p.distance(0, 0));
```

And to implement the changed problem specification, the stream pipeline needs only a small corresponding change:

```
intList.stream()  
    .map(i -> new Point(i % 3, i / 3))  
    .sorted(comparing(p -> p.distance(0, 0)))  
    .forEach(p -> System.out.printf("(%f, %f)", p.getX(), p.getY()))
```

The change needed to accommodate the new problem statement illustrates some of the advantages that lambdas will bring. Changing the `Comparator` was straightforward because it is being created by composition and we needed to specify only the single component being changed. The use of the new comparator fits smoothly with the existing stream operations, and the new code is again close to the

⁷In Chapter 5 we will describe the detail of how Java 8 allows methods to be added to interfaces.

⁸Other overloads of `comparing` can create `Comparators` for primitive types in the same way, but since natural ordering can't be used, they instead use the `compare` methods exposed by the wrapper classes.

problem statement, with a clear correspondence between the changed part of the problem and the changed part of the code.

It should be clear by now why the introduction of lambda expressions has been so keenly awaited. We saw earlier the possibilities for performance improvement that they will create by allowing library developers to leverage parallelism. Now, in the signature of `comparing`, we see a sign of things to come: as client programmers become comfortable with supplying behaviors like the key extraction function that it accepts, fine-grained library methods like `comparing` will become the norm and, with them, corresponding improvements in the style and ease of client coding.

Quick Start Guide to **JavaFX**



Create Dynamic Enterprise Client Applications

J.F. DiMarzio

*Oracle
Press™*



Chapter 9

Basic Animation

Key Skills & Concepts

- Using Timelines
 - Creating paths
 - Using KeyFrames and KeyValues
-

This chapter introduces you to the world of basic JavaFX animation. Whether you want to create animated text and fly-ins or gain knowledge for creating games, basic animation is the place to begin.

You will need to master three basic topics when tackling basic JavaFX animation:

- Timelines
- KeyFrames and KeyValues
- Paths

To begin this chapter, open NetBeans and create a new, empty JavaFX application named **Chapter9**. Based on previous chapters, after you remove the Hello World code, the contents of the Chapter9.java file should look as follows:

```
package Chapter9;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

/**
 *
 * @author J F DiMarzio
 */
public class Chapter9 extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }
}
```

```
@Override
public void start(Stage primaryStage) {

    StackPane root = new StackPane();
    primaryStage.setScene(new Scene(root, 600, 480));
    primaryStage.show();
}
}
```

The first section of this chapter covers Timelines.

Timelines

All animation, whether it is “traditional” hand-drawn animation or computer-based animation, is controlled by timing. What actions occur and when, the length of time it takes to walk from one side of a room to another, and syncing the dialogue to a character’s mouth movements are all actions that are controlled by some sense of timing. The timing of the animation dictates when each action begins, when it ends, and how long it lasts.

In JavaFX, animation can be moving objects around on the screen, but it can also be something like a highlight being applied to a button on a mouse over or the expansion of a pane in an menu.

Timing is critical to smooth animation. If there is too much time between each frame of animation, it will look slow and jerky to the user. If there is too little time between each frame, the animation will be too fast. This is why timing is so critical.

In JavaFX, the timing of animation is controlled by a Timeline. A Timeline takes a set of KeyFrames and KeyValues to modify properties of your application over time. The class that defines JavaFX Timelines is `javafx.animation.Timeline`.

The purpose of a Timeline is to break down frames of animation into “stops,” by time. This means that if you tell a Timeline where you want an object to be one second from now, and then five seconds from now, the Timeline will modify a value that you can apply to your object. The Timeline takes on the responsibility of producing a smooth increment of values that can be used to represent the movement of your object over the time you specify in your keyframes. This may sound a bit confusing now, but it will make a lot more sense when you see a Timeline in action.

A Timeline is broken down into a collection of keyframes. A *keyframe* is a point at which you want one or more KeyValues to be set to defined values. The Timeline then interpolates between the defined values automatically. For example, you are going to make an animation of a ball moving from the top of the screen to the bottom. Therefore, your keyframes will represent the start of your animation at the top of the screen as well as the end of your animation at bottom of the screen, as shown in Figures 9-1 and 9-2, respectively. The job of the Timeline is to fill in the space in between.



Figure 9-1 The first ball keyframe

You are now going to animate a ball image used from <http://jfdimarzio.com/ball.png>. You will make the ball image move down the scene, along the y-axis. To begin, you need to create a group.

Why a group? A group is a collection of nodes. In this case you are going to need a group to hold your `ImageView` and the ball image. The `StackPane` that you have been using in the previous chapters allows you to stack nodes, but it doesn't allow them to be freely moved around the scene. To animate an image, you need to literally manipulate the x-, y-, or z-coordinate of the image. A `StackPane` simply doesn't allow that natively.

To get around this issue, you are going to create a group that can be freely moved around the scene. You will then add the image to the group.

The following code sets up your group and image calls:

```
package Chapter9;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
```



Figure 9-2 The second ball keyframe

```
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

/**
 *
 * @author J F DiMarzio
 */
public class Chapter9 extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        ImageView imageView = new ImageView();
    }
}
```

```
Image ball = new Image("http://jfdimarzio.com/ball.png");
imageView.setImage(ball);

primaryStage.setHeight(480);
primaryStage.setWidth(600);

Group group = new Group();
Scene scene = new Scene(group);
group.getChildren().add(imageView);

primaryStage.setScene(scene);
primaryStage.show();

    }
}
```

If you have read Chapters 7 and 8, you will recognize most of this code. Therefore, this is a quick overview. The first section contains the image and `ImageView`. Next, you set the size of the Stage and create new `Group` and `Scene` nodes. The `Group` is added to the `Scene`'s constructor, and the `ImageView` is added to the `Group`.

Once your image, group, and scene are created, you can begin to set up your `Timeline`.

To use the `Timeline`, you will need to add one or more `KeyFrames` to it. Each `KeyFrame` has one or more `KeyValues`. The `KeyValue` specifies what property of the `ImageView` (or any node) you want to modify, and the value you want to modify it to. In this example, you are going to modify the `y`-translation property of the image to 370. When the app starts, the image will be translated by zero pixels on the `y`-axis; the `Timeline` will use the `KeyFrame` and its `KeyValue` to interpolate the `y`-translation between 0 and 370.

Now you can create a `KeyFrame` that takes your `KeyValue` and applies a specific amount of time to it. For this example, you position the `KeyFrame` 2 seconds after the `Timeline` starts. Finally, add the `KeyFrame` to the `Timeline`. The `Timeline` will do the rest for you. Here is the code:

```
package Chapter9;

import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
```

```
import javafx.scene.image.ImageView;
import javafx.stage.Stage;
import javafx.util.Duration;

/**
 *
 * @author J F DiMarzio
 */
public class Chapter9 extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        ImageView imageView = new ImageView();
        Image ball = new Image("http://jfdimarzio.com/ball.png");
        imageView.setImage(ball);

        primaryStage.setHeight(480);
        primaryStage.setWidth(600);

        Group group = new Group();
        Scene scene = new Scene(group);
        group.getChildren().add(imageView);

        primaryStage.setScene(scene);
        primaryStage.show();

        Timeline timeLine = new Timeline();

        KeyValue keyValue = new KeyValue(imageView.translateYProperty(), 370);
        KeyFrame frame = new KeyFrame(Duration.seconds(2), keyValue);
        timeLine.getKeyFrames().add(frame);
        timeLine.play();
    }
}
```

Run this app and you will see the ball image move from the top of the screen to the bottom.

This process is good for simple motions, but what if you want to move your ball in a more elaborate way? The next section of this chapter covers animating your images along a path.

Animating Along a Path

If you want to do a lot of math—and some tricky calculations—you can create a lot of movements with the animation style explained in the previous section. However, if you really want to do some complex animation, such as moving an object around the screen in a curving motion, you will want to use path animation, which is another method that JavaFX has for creating animation that allows you to move an object around a predefined path. In this section you learn how to create a path using knowledge you picked up in previous chapters. You then use this path to animate the ball.

The concept here is that you can create a path using lines, arcs, and points. JavaFX will then animate your image moving along this path.

To begin, set up your `Chapter9.java` file as shown here:

```
package Chapter9;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

/**
 *
 * @author J F DiMarzio
 */
public class Chapter9 extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        ImageView imageView = new ImageView();
        Image ball = new Image("http://jfdimarzio.com/ball.png");
        imageView.setImage(ball);

        primaryStage.setHeight(480);
        primaryStage.setWidth(600);

        Group group = new Group();
```

```
Scene scene = new Scene(group);
group.getChildren().add(imageView);

primaryStage.setScene(scene);
primaryStage.show();
    }
}
```

Similar to what you have seen before, this code grabs the same ball image you have been working with and creates a group. In the previous section, you added a Timeline to this code to animate the ball image travelling up and down the y-axis. For this example, you are going to animate the ball moving around an abstract path.

The next step is to create the path you want your ball image to travel along. You will use a Path node to create this path. The node accepts a collection of elements to create a path from. You can easily create a group of elements that makes an abstractly shaped path. The following piece of code defines a small element array with some basic line-based shapes:

```
...
Path path = new Path();
ArcTo arc1 = new ArcTo();
ArcTo arc2 = new ArcTo();

arc1.setX(350);
arc1.setY(350);
arc1.setRadiusX(150);
arc1.setRadiusY(300);

arc2.setX(150);
arc2.setY(150);
arc2.setRadiusX(150);
arc2.setRadiusY(300);

path.getElements().addAll(new MoveTo(150f, 150f), arc1, arc2);
...
```

There is nothing too complex or tricky about what is happening here. You have created a collection of elements and added them to a Path. The elements contained within the Path are MoveTo and two instances of ArcTo. The combination of these elements creates a path your ball can follow.

The MoveTo element does exactly what the name suggests: It moves you to a specific point on the Cartesian grid. In this case, it moves you to x150, y150. You are specifying this as your first element to cleanly move the starting point of your path before you start “drawing.”

The next two elements draw arcs. The first `ArcTo` element draws an arc from the last position of the point (in this case, `x150, y150`, thanks to the `MoveTo` element). The second `ArcTo` draws another arc from the end point of the last arc.

The JavaFX animation package can now take this `Path` node and use it to animate your ball using a `PathTransition`:

```
PathTransition pathTransition = PathTransitionBuilder().create();

pathTransition.duration(Duration.seconds(5))

pathTransition.node(group)
pathTransition.path(path) pathTransition.orientation(OrientationType.
ORTHOAGONAL_TO_TANGENT)
pathTransition.build();

pathTransition.play();
```

To create your animation, you use a `PathTransition` class, which takes in a few familiar parameters. Like `Timeline`, `PathTransition` can accept parameters for `AutoReverse`, `CycleCount`, and an interpolator—if you choose to use them. However, it is the node, path, duration, and orientation that you want to focus on for this animation.

The node is the object you want animated. In this case, the ball image from the image group. The ball is assigned to the node in the `PathTransition` class. The node will be animated along the path you created earlier. Use the `setPath()` method of the `PathTransition` to add.

Finally, the orientation parameter specifies the position of the node as it is animated along the path. If you do not specify an orientation, the image will remain in whatever orientation it is in when it is drawn to the screen. Setting the orientation to `ORTHOAGONAL_TO_TANGENT` tells JavaFX to change the orientation of the node as it moves along the path. This change in orientation gives the animation a more organic feel.

The full path animation code should look as follows:

```
package Chapter9;

import javafx.animation.PathTransition;
import javafx.animation.PathTransition.OrientationType;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;
import javafx.util.Duration;
```

```
/**
 *
 * @author J F DiMarzio
 */
public class Chapter9 extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        ImageView imageView = new ImageView();
        Image ball = new Image("http://jfdimarzio.com/ball.png");
        imageView.setImage(ball);

        primaryStage.setHeight(480);
        primaryStage.setWidth(600);

        Group group = new Group();
        Scene scene = new Scene(group);
        group.getChildren().add(imageView);

        primaryStage.setScene(scene);
        primaryStage.show();

        Path path = new Path();
        ArcTo arc1 = new ArcTo();
        ArcTo arc2 = new ArcTo();

        arc1.setX(350);
        arc1.setY(350);
        arc1.setRadiusX(150);
        arc1.setRadiusY(300);

        arc2.setX(150);
        arc2.setY(150);
        arc2.setRadiusX(150);
        arc2.setRadiusY(300);

        path.getElements().add (new MoveTo (150f, 150f));
        path.getElements().add (arc1);
        path.getElements().add (arc2);

        PathTransition pathTransition = new PathTransition();
        pathTransition.setDuration(Duration.seconds(5));
        pathTransition.setNode(group);
        pathTransition.setPath(path);
        pathTransition.setOrientation(OrientationType.ORTHOGONAL_TO_TANGENT);
    }
}
```



```
        pathTransition.play();
    }
}
```

Compile the preceding code and run it. You will see the image of the ball animated around an elliptical path.

Try This Create a Path Animation

In the previous chapters, the “Try This” sections have focused on added functionality that may not have been directly covered in the chapter. However, the skills covered in this chapter are so important that this section will focus on enhancing these skills.

Create a new project and add an image or shape to it. Then, try creating your own path along which you will animate the image. Experiment with paths of different sizes and lengths. Adjust the speed of your Timelines to change the feel of the animation.

The more you are comfortable with the animation capabilities of JavaFX, the better your applications will be.



Chapter 9 Self Test

1. Why is timing important to animation?
2. What controls the timer in JavaFX animation?
3. True or false? A Timeline contains a child collection of KeyFrames.
4. How do you start the Timeline?
5. True or false? A keyframe is a point at which you want one or more keyvalues to be set to defined values.
6. Which property of Animation sets the number of times a Timeline executes?
7. What is the purpose of ArcTo?
8. A path is created from a group of what?
9. What builder class is used to create an animation from a path?
10. Which PathTransition.OrientationType will change the orientation of the node as it moves along the path?

Mastering **JavaFX 8** Controls



Create Custom JavaFX 8 Controls for Cross-Platform
Applications

Hendrik Ebbers

Oracle
Press



CHAPTER 1

The History of Java UI Toolkits

2 Mastering JavaFX 8 Controls

Almost 20 years have passed since Java was first released in 1995; the eighth major version was released in 2014. During these two decades, the IT world has rapidly evolved. The size, speed, and requirements of computer hardware have changed dramatically, as have the user interfaces of software. In 1995, computers were mainly used in offices, making a decorative user interface (UI) unimportant for most applications. Most dialogs consisted only of labels, text fields, and buttons. More complex graphical user interface (GUI) elements such as tables or tab panes were not supported by most of the UI toolkits. But as computing has evolved from a specialized niche to part of everyday life for millions of people worldwide, the importance of polished, practical, and purposeful UIs has become paramount. It is now normal to have a computer or tablet-based device at home to manage music, photos, or other private documents, and most people using applications today do not have technical backgrounds, which is why applications have to be intuitive and easy to use. A good layout and modern UI controls and effects can help generate a better user experience. By using up-to-date technologies and frameworks, developers can create outstanding web, desktop, and mobile applications, and that's why UI toolkits, including the Java UI toolkits available with the Java Development Kit (JDK), have evolved over the last 20 years.

This chapter will give you an overview of the important Java-based UI toolkits and some rising trends. Today, most applications have their own style, and the views are laid out in a pixel-perfect way. You'll find out how that came to be.

Java SE UI Toolkits

Several generations of UI toolkits have been introduced in the JDK over the years to allow developers to create state-of-the-art applications with Java. JavaFX is the newest framework to provide the ability to create and design desktop applications with Java. Before I discuss the controls of JavaFX in depth, it is important to take a short look at the history of Java-based UI toolkits that are part of Java Standard Edition (Java SE). By doing so, you will get an overview of the fundamental differences and similarities between several generations of UI toolkits, specifically in relation to the JavaFX controls.

AWT

The first version of the Java Abstract Window Toolkit (AWT) was introduced in 1996; AWT is an abstraction of the underlying native user interfaces. Since Java runs on various platforms, AWT supports only the least common denominator of these platforms, so it has only a small number of supported components. Standard controls such as buttons and text fields are available, but more complex components such as tables are not part of the toolkit. By using AWT, developers create GUI components in Java code. Simultaneously, a native graphical component is created as a counterpart by the operating system, and a peer class is used as the link between these two instances. (These kinds of components are called *heavyweight* components.) Developers can define the attributes of a component, such as the visible text of a button, by using the Java class. However, the Java application has no influence on the graphical representation of the components because the operating system (OS) is responsible for rendering the controls.

AWT was improved with Java 1.1; it included new features such as event listeners and new components such as the scroll pane. However, the great advantage of AWT is also its worst weakness: By using the toolkit, each Java-based application takes on the native look and feel of the operating system automatically. On Windows, the typical Windows buttons and combo boxes

will be shown if you create an app by using the framework, for example. On Mac OS, all components are rendered by using the Aqua look (Apple's default UI definition). It's almost impossible to create new components or modify the look of a control to deliver an application with a unique appearance.

Java Foundation Classes and the Emergence of Swing

In parallel with Java 1.1, Netscape developed the Internet Foundation Classes (IFC) library that represents a completely platform-independent UI toolkit for Java. Unlike AWT, IFC does not create a wrapper around native components; it provides controls that are completely managed and rendered by pure Java. This technology was originally designed to display applets in the Netscape browser, and the main objective of IFC was to create browser-independent applications that have the same appearance on any OS. In 1997, Sun Microsystems and Netscape announced the intention to merge IFC into Java.

The Java Foundation Classes (JFC) framework is the result of integrating IFC into Java. The classes in the framework include AWT, Java2D, Swing, and some additional APIs. JFC has been part of Java SE since 1998, which means Swing has been part of Java SE since version 1.2 (Java 2) and has become the main UI toolkit of Java.

Swing

Unlike the base development of IFC, which was written from scratch as a new API, Swing's control classes are based on AWT; however, the internal architecture of the framework is completely different from AWT. This approach was chosen for compatibility purposes. Swing offers a set of so-called *lightweight* components that are completely managed and rendered by Java. Because of this, you can achieve the same graphical representation of components across operation systems. From a technical point of view, the graphical output of Swing is based on Java2D, an API for rendering two-dimensional objects that is also part of JFC. Although the features of Java2D and Swing are not "state of the art" anymore, these were modern APIs with many incredible options when JFC was released. Even today, you can create astonishingly good results by using Swing.

All UI controls in Swing are based on the `JComponent` class, which extends the `AWT Component` class. This ensures that the main concepts for using Swing components are already known by AWT developers, and AWT layout managers, for example, can be used to lay out Swing-based applications without any learning curve. Figure 1-1 shows a general overview of the class hierarchy of AWT and Swing components.

By using the Java2D API, you can change the appearance of Swing-based components at any time or even create new components from scratch. Swing uses a Model–View–Controller (MVC) approach internally, in which the graphical representation of components is separated from the model in a special UI class. The base skin of a Swing button is defined by the `ButtonUI` class, for example. Since the operating system doesn't draw the components in Swing, the controls will have the same look across OSs. To achieve this, Swing includes the LookAndFeel (LAF) API. By using LookAndFeel, you can define your own style for the complete Swing component set. In fact, Swing comprises a set of cross-platform LAFs and system-dependent LAFs. If you want to develop an application that always looks like the underlying operating system, you set the OS-specific look and feel for Swing. A Java version for Mac OS includes the Aqua LAF, for example. This will render all components so that they look like native Mac OS controls. If your application is running on a Windows system, it can use the Windows LAF that is part of Java on every Windows-based PC. New versions of these LAFs have native support for creating controls that you can't

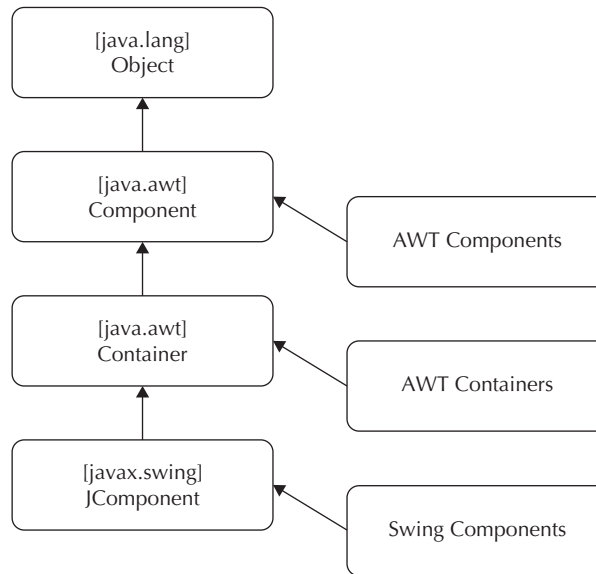


FIGURE 1-1. *Class hierarchy for AWT and Swing*

distinguish from native ones. The framework offers some helper methods, as well. By using them, you can configure Swing to always use the provided system look and feel depending on which platform the application is running.

Another advantage Swing has over AWT is the rich set of components it includes. For example, in Swing, you can find tables, lists, and tree-based controls to represent the application data in the way that best fits your application view. These controls can handle lists of data by using renderers to support large amounts of data and show or process them in the interface without any problems. Above all, these new and flexible components are the reason why Swing is used to develop business applications. With Swing's ability to manage and render controls that support LAFs and its internal use of Java2D, along with many open source libraries and frameworks that can be used to extend functionality, Swing deposed AWT and remained for several years the standard UI toolkit for creating graphical desktop applications in Java.

From today's point of view, Swing also has some weaknesses. One weakness is that many graphical effects that are standard in today's modern applications cannot be implemented by using Swing (or they need a lot of hacks and workarounds). Examples include reflections and blur effects. Animations are also missing from Swing's API, and a Swing-based dialog needs a lot of boilerplate code. Although creating special skins for controls or creating new components from scratch is possible in Swing, it is difficult to do. It requires a lot of training, and there are many pitfalls you can trip over before being ready to develop usable components for Swing. These are crucial reasons why Swing needed to be replaced by a new UI toolkit. Hence, JavaFX emerged and has been the recommended UI toolkit since Java 8.

Before diving into the history and features of JavaFX, I'll briefly cover a few other UI toolkits and place them in the historical context of the default Java SE toolkits.

Additional UI Toolkits

In addition to the default toolkits that are part of Java SE, some other UI-based frameworks have been developed over the years. SWT and Apache Flex are two examples of toolkits developed during the reign of Swing. SWT is based on Java, but Apache Flex has nothing to do with Java and even offers some concepts that JavaFX has picked up.

SWT

Parallel to the release of Java 2 in 1998, IBM decided to implement its next generation of Java development tools in Java. The first generation of IBM's development environment, VisualAge for Java, was based on Smalltalk and used the common widget (CW) framework to create the surface. This API was a thin layer on top of the native components of the operating system and, therefore, resembled AWT. For the developers at IBM, it was important that the new development environment, which today is known as Eclipse, would be based on a native look and feel. Since Swing could not provide these requirements by supporting platform-specific LAFs, the developers decided to create a separate UI toolkit with the same features as CW. The result was the Standard Widget Toolkit (SWT).

Like AWT, SWT provides wrappers on top of native controls. The native controls are provided via the Java Native Interface (JNI), but SWT includes an API to write your own GUI components. Additionally, SWT provides a larger set of default controls than AWT does. All components that are not supported by an underlying OS are emulated in Java. Tables, for example, are supported by the Microsoft Windows platform, and SWT can depend on native components by using JNI. On an OS that doesn't support tables, SWT will use a fallback and manage and render a table control completely in Java. With this functionality, developers can create an application with a native appearance and add controls or change the skin of controls to define a unique look for the app. Compared to Swing, SWT requires fewer system resources because most of the controls are managed by the OS and not by Java. Today, SWT is still the default UI toolkit of Eclipse and is also used in many projects that are based on the Eclipse rich client platform (RCP).

Apache Flex

In recent years, other languages have breathed new life into the field of UI toolkits. Apache Flex is an example of a toolkit developed in the last few years, and it was clearly designed for creating rich clients. It is based on Adobe Flex, which was passed to the Apache Foundation in 2012. Internally, Flex is based on Flash for rendering and offers its own programming language called ActionScript.

Flex offers some interesting techniques and concepts that have been sought after in Java UI toolkits. For example, with its MXML library, Flex provides an XML-based file format to define user interfaces and their layout. In these files, the structure of a view with all embedded controls and their attributes can be defined. Version 4 of Flex introduced Spark as a new architecture to skin and create controls in Flex. In Spark, all controls are split in two files: A skin file that is written in MXML and that defines the look of the component and an ActionScript class that defines the model and the controller. In addition, Flex provides support for effects and transformations.

The Way to JavaFX

As you can see, there are plenty of UI toolkits on the market, both based on Java and other languages. But no toolkit is perfect. Sometimes a cool feature is incompatible to the main architecture of a toolkit and can't be added. Additionally, sometimes different UI toolkits have

6 Mastering JavaFX 8 Controls

different philosophies. Some rely on native controls, while others have extended support for skinning. Another feature that has become more important over the years is the way the metadata of controls, such as the background or border and the layout of views, is described. Most modern toolkits remove this information from the source and add file types such as MXML in Flex or the XML layout in Android to define this information. Old Java-based UI toolkits like Swing can't handle these needed features.

From F3 to JavaFX 8

The JavaFX story started with the F3 API developed by Chris Oliver at SeeBeyond. The company required a modern UI toolkit to create new desktop applications that looked superior to the competition, so Oliver started developing the F3 framework, and it was acquired by Sun Microsystems as part of the SeeBeyond acquisition during the API's development. Oliver continued on at Sun to lead the development of F3, which was renamed and introduced as JavaFX at JavaOne in 2007. The first version of JavaFX was published one year later. However, version 1 of JavaFX (JavaFX Script) had very little to do with the current version; it was a script-based language for the Java platform that could interoperate with Java code.

After Oracle's acquisition of Sun Microsystems, version 2 was announced that would be based completely on the Java API, which would allow any Java developer to use it with any IDE. By doing this, the barrier of entry to using JavaFX was reduced, and the competition for a great UI toolkit was leveled. JavaFX Script was also discontinued with this release. JavaFX supports a lot of effects, transformations, and animations, all of which will be covered in the following chapters.

What Kind of Applications Can Be Built with JavaFX?

So, what kind of applications can you build with JavaFX? As an initial answer, I would say every kind of application. For sure, some types of applications are a better match to a JavaFX-based technology stack than others, such as business applications that use databases or servers as the back end. All the needed components are part of the JDK and the JavaFX library, so you can create an application mostly the same way as you would have with Swing.

But JavaFX can do so much more. I have seen some 2D games that were created by using JavaFX with the great performance and features of the JavaFX scene graph API or the JavaFX canvas API. Additionally, JavaFX offers 3D support to create 3D landscapes. By adding embedded support to Java, JavaFX allows you to create the UI and user interaction for smart embedded devices. Using JavaFX in this way is as easy as developing a desktop application. You can even develop a media center because the API to play media files is part of JavaFX. As you can see, there is a lot of potential when using JavaFX as the UI toolkit to develop applications.

In reality, most of the applications that will be developed with JavaFX will be business applications, so this book will concentrate on the APIs and knowledge that you need to know to develop these kind of applications. But even when developing data-centric applications, you can use the creative power of JavaFX. By using the JavaFX effects, animations, or multitouch input, you can create an application with an outstanding user experience.

JavaFX Compared to HTML5 and Web-Based Technologies

Today a lot of applications that are created are web applications or rich Internet applications (RIAs), also called plain HTML apps, that run in a browser such as Firefox or Chrome. Most of these applications are written in HTML5, CSS, and JavaScript. Other technologies can also be used to create RIAs: Technologies such as Adobe Flash/Flex and Silverlight can be used to create applications that are running inside a browser with a browser plug-in.

These rich Internet applications could also be created with JavaFX. (Although you can integrate a JavaFX application as an applet in a web page, this workflow isn't best practice anymore as it will create some problems; therefore, it won't be discussed in this book.) I discussed the non-HTML technologies earlier in the chapter, so now it's time to take a deeper look at plain HTML RIAs and how they compare to applications created with JavaFX.

First, it's hard to compare HTML with JavaFX because of some big differences: HTML runs inside a browser, and JavaFX applications are desktop applications running directly in the OS. Additionally, HTML is only a markup language, and you can't define application logic with HTML. A developer needs to use a combination of HTML, JavaScript, and CSS to create an interactive application.

Here is the default structure of an HTML-based RIA: By using HTML, you define all components that appear on a web page and structure them. If you need application logic, you can use JavaScript to add the logic to your application. Additionally, in most applications, CSS is used to define special skins for the app and all components that are part of the application. This is a technology stack that is unusual for a desktop application; however, JavaFX provides a comparable set of technologies. Specifically, the layout of all views can be done by using FXML, which is an XML-based markup language for defining JavaFX views. For the skinning of an application, JavaFX supports CSS; it doesn't use the same attributes that are used in HTML web applications, but the CSS syntax is the same. Instead of JavaScript, you can use Java to define the logic and interaction of a JavaFX application.

JavaFX offers all the benefits that a developer might know from HTML application development. For example, the structure of the views isn't created in code; the markup language FXML is used to define the layout of all application dialogs. By doing so, the layout of an application can be done by a designer who doesn't need to understand Java code. Additionally, CSS is used to define custom skins of controls. By using CSS, it is easy to change the font of all buttons that are used in a JavaFX application, for example. There is another big benefit in JavaFX, too: The APIs are ready for extensions. In HTML, you can't use other tags than the defined ones, and CSS provides some default attributes and a set of additional ones that are browser-specific. With FXML, you can easily integrate any custom control, and you can define new CSS attributes with a Java API. By doing so, you can easily add components to an application that are not part of the default framework.

HTML applications do have some advantages over JavaFX ones, however. HTML is always running in a browser, and a normal user doesn't need to install anything to run web applications. By contrast, JavaFX applications mostly run on the desktop, and if they are not packaged as native applications, the user will need the Java runtime on the OS. And if a JavaFX application is running in a browser, the user will need the applet plug-in. JavaFX 8 fixes this issue by offering a tool that can package JavaFX applications as native ones and add the needed Java runtime to the package automatically. By doing so, no additional software is needed on a client computer. Still, HTML

8 Mastering JavaFX 8 Controls

applications are easier to administrate because most users have a browser, but often cross-browser development is a necessity.

You could say that there is no final rule which of these two technologies should be used for application development. Both have benefits and are stronger in some areas. But JavaFX has learned a lot from HTML and has taken some of the best parts of it to offer a great infrastructure and ecosystem for application developers.

Java-Based Web Frameworks

In addition to creating plain HTML web applications, developers can use Java to develop web applications with frameworks such as JSF, Wicket, Play, or GWT. All these frameworks will create applications with views that are rendered as HTML views in a browser. Normally the Java code is running on a server, and HTML views are created that will be sent to the client. In all these frameworks, Java can be used to define the application logic, and sometimes even the views can be created in Java. In the end, all the frameworks will create HTML and JavaScript. Because of this, it is often more complicated to create pixel-perfect applications with these frameworks. Comparing all these frameworks to JavaFX is beyond the scope of this book.

Summary

UI-related technology has become more important in the past few years because developers are creating more impressive UIs than ever before. JavaFX is the newest in a series of UI toolkits, and it supports all modern UI methods and patterns. Without a doubt, JavaFX will become the most important UI toolkit for Java applications in the next few years and will be used on various platforms. Therefore, it is important for every Java application developer to know and understand the core concepts of JavaFX. One of the most important parts of the framework is the controller API, a core focus of this book.

Save 20% and get free U.S. shipping on our site

on all books included in this sampler plus many other authoritative Java books from Oracle Press; also available at online retailers.

