Organization of Java

With a firm grounding in the elements of the Java language and in designing programs that include choices and loops, we are well equipped to put Java to work in solving interesting problems. In this section, we will move further along in our study of Java, as we learn to use additional classes in Java's Application Programming Interface (API). At this stage, we only want to put classes to work in interesting programs. Many advanced topics, such as inheritance and polymorphism, are deferred to later chapters.

The Java API is a collection of *packages*. Each package contains a collection of *classes*, and each class contains a collection of *methods*. Through methods, *objects* are created and acted upon. In the following sections, we will describe the organization of Java, as seen through its packages, classes, methods, and objects.

What is a Package?

A *package* is a collection of classes. Packages are simply a convenient place for developers to put classes of a common purpose; however, they play no specific role in the Java language. The Java developers at Sun Microsystems have organized the many hundreds of classes in the Java API into just over 50 core packages. Within these, hundreds of classes and thousands of methods are defined. We can define our own classes and put them into a package as well, and we'll learn how to do that later.

The packages from the Java API encountered in these notes are summarized in Table 1.

Table 1. Packages in the Java API

Package	Description
`java.applet`	provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context
`java.awt`	contains all of the classes for creating user interfaces and for painting graphics and images ("awt" is an acronym for “advanced windowing toolkit”)
`java.io`	provides for system input and output through data streams, serialization, and the file system
`java.lang`	provides classes that are fundamental to the design of the Java programming language
`java.util`	contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)
`javax.swing`	provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms

The import Statement

To use classes from a package in the Java API, the `import` statement is required at the top of the program. For example, the `BufferedReader` class appears in the `java.io` package. Any program that uses this class must include the following statement at the top of the source file:

```
import java.io.BufferedReader;
```

The `import` statement informs the compiler of the whereabouts of classes used in a program that are not defined in the program. It is common and convenient to simplify the statement above as follows:

```
import java.io.*;
```

The asterisk (*) is a wildcard character. With this statement, any class from the `java.io` package can be used in the program.

The `java.lang` package is special. It provides classes that are fundamental to the design of the Java language. The most important classes are `Object`, which is the root of the class hierarchy, and `Class`, instances of which represent classes at run time. In most cases, we are not aware of the presence or use of these classes, as they operate "behind the scenes" in a program. As well, the `java.lang` package provides the wrapper classes, the `String` class, and the `System` class. We will say more about these later.

Since the `java.lang` package is so fundamental to the language, the following statement is implied for all Java programs:

```
import java.lang.*;  // Implied!  Not necessary
```

What is a Class?

A class is like a data type. Just as we create and operate on variables of a certain data type, we create and operate on objects of a certain class. However, a class is not a primitive data type, like `int` or `double`, because the data it encapsulates are more sophisticated. Central to this sophistication are the methods of each class. So, for example, there is a class named "`String`", and we can use methods of the `String` class to perform useful operations on `String` objects.

However, not all classes fit the description above. Each of our demo programs contained a class bearing the same name as the filename. Clearly, these classes are in no way a "data type". As well, Java's `Math` class is unlike a data type. One cannot create a `Math` object (like we create a `String` object). The `Math` class is simply a convenient holding place for useful methods (to perform mathematical operations) and useful data fields (like the constant `PI`).

One feature that is common to all classes is that they contain methods. From the above discussion, it appears there are two broad categories of classes: those for which objects can be created, and those for which objects cannot be created. The `String` class is an example of the former, the `Math` class, the latter. This is a useful distinction, and we will draw on it throughout this discussion.

It's hard to talk about classes without mentioning objects and methods. Let's examine each of these.

What is an Object?

An object is an instance of a class. The act of "instantiating an object" is the act of creating an object of a class and initializing its data fields with values. So, a `String` object is an instance of the `String` class. Once an object is instantiated, we can perform operations on it. However, unlike a variable representing a primitive data type, we cannot, in general, directly access the data fields of an object. Our only means to do so is through the methods of that object's class.

What is a Method?

A method is a set of instructions to do something. So, in a programming sense, methods do the work, and objects are the things methods work on. We can categorize all Java methods as being *constructor methods*, *instance methods*, or *class methods*. Let's look at each of these.

Constructor Methods

The purpose of a constructor method is to construct an object. This is often expressed in more formal terms: The purpose of a constructor method is to *instantiate an object*. It seems reasonable from our earlier discussion that the `Math` class does not have a constructor method (One cannot instantiate a `Math` object!), whereas the `String` class does. In fact, this is the case. There is no method called `Math()`. However, there is a method called `String()`. It is a rule of the Java language that a constructor method has the same name as its class. We have used a few constructor methods already, such as `String()`, `BufferedReader()`, and `InputStreamReader()`. These are constructor methods for the classes `String`, `BufferedReader`, and `InputStreamReader`, respectively.

A characteristic of Java and other object-oriented programming languages is the ability to have more than one method with the same name. We saw examples earlier of two methods called `String()`. One was called without an argument inside the parentheses, the other was called with one argument — a string constant. These are two different methods. The compiler determines which to use by context. If the compiler finds a `String()` constructor with no argument, the default constructor is used. If one argument appears and it is a string constant, then the method with one string constant is used. When two methods have the same name, the methods are said to be *overloaded*. A more fancy term for this is *polymorphism*, which is discussed later.

Instance Methods and Class Methods

Distinguishing between instance methods and class methods is a bit tricky. Most of the methods used in demo programs thus far were instance methods. Consider the following two statements:

```
String s = new String("hello");
int len = s.length();
```

There are two methods above: a constructor method named `String()` and an instance method named `length()`. Both are methods of the `String` class. The `length()` method is an instance method because it is called through an "instance" of the `String` class (in this case, `s`). Dot notation connects the implicit variable's name to the method's name. So, if an object variable precedes the method's name, the method is an instance method: It is called through an instance of the class.

Now consider the following two statements:

```
double x = 99.0;
double y = Math.sqrt(x);
```

The method `sqrt()` is a class method. (Class methods are also called *static methods*.) Preceding the method's name is "`Math.`" which is the name of a class. So, what precedes the method's name helps distinguish a class method from an instance method.

In some cases, class methods are called without dot notation. In fact, the only reason the prefix "`Math.`" is required, is to identify to the compiler where the `sqrt()` method is defined. (It is

defined in the `Math` class in the `java.lang` package.) If a method is defined in the same file where it is called, then prefixing the method's name with the class name is unnecessary. We will see examples of this later.

Class Hierarchy

Although we'll say a great deal about class hierarchies later, this important topic deserves a quick tour now to alert you to an essential characteristic of Java and other object-oriented programming languages. We noted earlier that the relationship between a class and its package is mostly one of convenience. (A package is a convenient place to put common classes.) Of far more importance is the relationship of a class to other classes. All classes in Java exist in a hierarchy. Those "further down" the hierarchy inherit characteristics from those "further up" the hierarchy.

At the top of the hierarchy is the `Object` class. All Java classes inherit characteristics from the `Object` class. An example of a class "just below" the `Object` class is the `String` class. In this relationship, the `Object` class in the *superclass*, and the `String` class is a *subclass*. Figure 1a illustrates this. By convention, the arrow points to the superclass. It is common to show class hierarchies horizontally, as in Figure 1b, since more classes can be placed in a single illustration. We'll use this format in these notes.

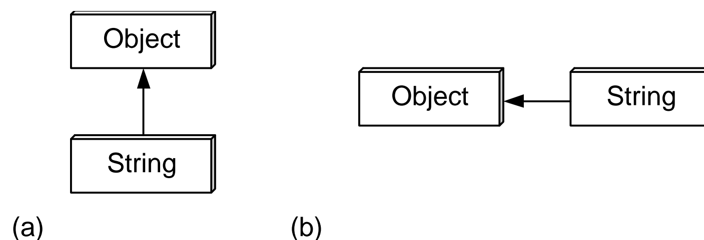

Figure 1. Class hierarchy (a) vertical illustration (b) horizontal illustration

The documentation provided by Sun Microsystems for the Java API illustrates this relationship as shown in Figure 2.

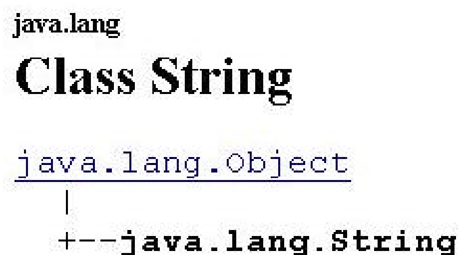

Figure 2. Class hierarchy, as shown in the Java API documentation

Since the `String` class is a subclass of the `Object` class, `String` objects inherit characteristics of `Object` objects. An important benefit of this relationship is that methods defined in the `Object` class are automatically methods of the `String` class too. The `String` class inherits these as part and parcel of the inheritance relationship. We'll say more about this later.

With this introduction to classes, objects, methods, packages, and class hierarchy, we are well prepared to explore some of the many interesting classes in the Java language.