Objective Viewpoint

A Practical Crash Course in Java 1.1+ Programming and Technology: Part II

by *George Crawford III*

Java Language (Continued)

In the first part of this crash course, we had an overview of Java, traced some of the history of Java, and discussed some of the language features. We also reviewed how to use the JDK, and began examining syntax and semantics issues about the Java language. In this follow-up column, we continue to look at details about Java syntax and semantics. Enjoy!

Classes And Objects

A class is a template for an object. It describes the attributes and general behavior of an object. For example, the Sun (not to be confused with the company), an object, is an instance of a class called Star. There are certain characteristics that are shared by all stars, such as temperature, size, and color. A star represents a generic set of suns. Each instance of the Star class, such as our own sun or Alpha Centauri, will have different temperature, size, and color values. Thus, we can think of a class as a blueprint for an object which contains placeholders for values (e.g., size, color, temperature). An object is a concrete example of that object with specifiable values for its attributes (e.g., size = 13, color = yellow, temperature = 8000K.).

Java has several keywords for class specification, encapsulation, and manipulation, as shown in Table 3:

Table 3: Class Related Keywords

abstract	**implements**	**interface**	**private**	**return**
class	**import**	**new**	**protected**	**super**
extends	**instanceof**	**package**	**public**	**this**

These will be discussed further in the column.

Building Classes in Java

The Star class can be specified in Java with the following simple line of code:

```
class Star {
}
```

To give the class more character, add the following lines of code between the braces ({ }) as demonstrated below:

```
class Star {
  int size;
  int temperature;
  Color color;
}
```

Now we have a class with three attributes: an integer value that indicates the size and temperature of a particular Star object. The color of a specific Star instance is also an instance of a class, the *Color* class, which is part of the standard Java Abstract Window Toolkit (AWT) package.

Declaring a class instance is easy. It is the same as with primitive types. To declare an object of class Star called *myStar*, do the following:

Star myStar;

Declaring a Star, however, does not create an instance of a *Star* . In other words, no memory space has been requested for the *Star* variable. A *Star* is instantiated by using the *new* keyword:

```
  Star myStar = new Star();
```

or

```
  Star myStar;
  myStar = new Star();
```

Java puts new objects in a memory area called the heap. Each time you create an instance of a class, heap space is allocated for that object.

Packages

Java uses *packages* to group related classes, interfaces, and exceptions together in a common repository. For example, the String class is part of the *java.lang* package, and the Color class is part of the *java.awt* package. A list of all packages and their specifications is available at http://www.javasoft.com/products/

If a class exists inside of another package, then you must *import* the class before you can manipulate it. The exception to this is the *java.lang* package, which is a default package for all Java class files. The *import* statement has the following format:

```
  import packagename.<packagename/classname/wildcard>;
```

Where *packagename* is the name of a java package. The name following the first package name may be another package name, a class name, or a wildcard that signifies all classes in the specified package should be imported. All imports must appear *before* a class declaration. The following is an example of class importation using various import techniques:

```
import personal.MyClass;      // imports MyClass from the personal package
import java.util.Hashtable;   // imports the Hashtable class from
                              // the java.util package
import java.awt.*;           // imports all classes in the java.awt
                             // package (wildcard)
```

The first and second import examples are the preferred method of class importation. While using wildcards is acceptable, it does prevent developers from readily viewing class dependencies.

You can also reference packages explicitly instead of importing the package. This technique is most appropriate when you only need to use a class one or two times or two seperate packages both have a class with the same name. The following illustrates the use of *explicit package referencing* :

```
class MyClass extends java.util.Hashtable {
   ...
}
```

Importing classes is more suitable than explicit package referencing for most situations and provides for better code readability.

If you try to compile the Star class with the Color attribute without importing *java.awt.Color* or *java.awt.** you will get an error message indicating that the *Color* class cannot be found. Add the following line of code before the class declaration in the *Star.java* file:

```
import java.awt.Color;
```

Imports must appear somewhere *before* class declaration.

Access Control

If you create an instance of the Star class as it is currently written, you will be able to modify the attribute values. Total access to the class attributes is granted to your application. This is in opposition to the idea of information hiding with objects. The primary purpose of object-oriented design is to prevent promiscuous alteration of encapsulated data. Java provides four methods of controlling access to class members, as shown in table 4:

Table 4: Class Access Control

Access Specifier	**Description**
Private	**Class member is accessible only to other class members.**
Protected	**Class member is accessible to derived classes and other classes in the same package.**
Public	**Class member is available to all classes inside and outside the package and all derived classes.**
Default	**Class member is accessible to other classes in the same package and derived classes in the same package.**

Private members of a class cannot be accessed outside of the class. Only members inside the class can manipulate the values of private data or call private methods. Protected members will be discussed in more detail when we look at *inheritance*. Briefly, protected members of a class are accessible from other classes inside the same package and also from classes that inherit from the protected members' associated class. Public members are available to *all* classes. Normally, the *public* accessor method is applied to constant data or methods. The default access control mechanism restricts class data visibility to only classes in the same package or derived classes. Since the Star class data members do not have any explicit access specifiers, the default specifier is used.

An access specifier precedes the declaration of a variable, constant, class, or method. Make the data members of the Star class private by preceding each variable declaration with the *private* keyword as follows:

```
  private int size;
  private int temperature;
  private Color color;
```

Static Data Members

A class can have *static* data members. A static data member is a class attribute that is shared among all instances of a class. A commonly used *static* data member is a *count* variable that keeps a count of the number of instances of a class. Such a data member would be declared just like any other variable, and can be *private*, *protected*, or *public* . Just for fun, add the following line of code to the end of the Star class:

```
private static int count;
```

Methods

Class member access specifiers prevent programmers from changing encapsulated data values directly. So, how does a programmer change the value of an inaccessible member? She does so through the use of *methods*. Methods are the communication mechanisms for objects. They allow a developer to issue commands to an object that change the object's *state* or *behavior*. Here is the *Star* class with *accessor* methods added:

```
class Star {
  public Color getColor() { return color; }
  public int getSize() {return size; }
  public int getTemperature { return temperature; }
  public void setColor(Color newColor) { color = newColor; }
  public void setSize(int newSize) {size = newSize; }
  public void setTemperature(int newTemp) { temperature = newTemp; }

  private int size;
  private int temperature;
  private Color color;
}
```

Accessor methods provide a means of indirectly altering the values of an object's private members.

The format of a class method is as follows:

```
[AccessSpecifier] [Modifier]  ([Parameters])
{
  [Code]
}
```

AccessSpecifier can be *private*, *protected*, or *public*. If ommited, the default access control is used.

Modifiers are optional, but may include any of the following keywords: *final*, *native, static,* and *synchronized* . The *final* and *static* keywords will be explained later.

All methods must have a *return type*. This is the primitive datatype or class returned by the method after a method completes a call. The return type *void* indicates that the method does not return any value.

The name of a method can be any legal Java name as previously discussed. Use appropriate Java style for naming as described in *Naming Conventions* , which was in Part 1 of this column.

Parameters are enclosed within parentheses. The are declared just like any other variable (e.g., int newValue, Star star, etc.). Multiple parameters are seperated by a comma (","). A left brace ("{") follows the parameter list. This starts the method code block. A right brace ("}") ends the code block and appears on a seperate line immediately following the last line of code in the method.

Listed below are a few examples of class methods:

```
private void initialize() {
 // a private method with no return type.
}
public final Star getStar(String name) {
 // a public method with a modifier (final) and class
 // Star as a return type. The method parameter is a String
 // that represents the desired star's name.
}
public void send(Object[] message, int count) {
 // Another public method with no return type passed
 // an array of objects and an integer variable)
}
```

You may be wondering why we use methods to change the values of an object. Why not just change the values directly? As previously mentioned, information hiding is an important aspect of object-oriented programming. The programmer should not be concerned about the data *inside* an object. She should be primarily interested only in the behavior of the object, i.e., what can she do with the object? Removing the unessential details also allows the object's creator more flexibility in the internal attributes of the object. For example, the designer of a Car class might later decide that a float variable rather than an integer variable would be a better format for the Car's speed attribute. Altering the representation of the Car's speed does not change the appearance of the Car class to a developer. Instances of Car still have *getSpeed()* and *setSpeed()* methods. Thus, the outward behavior of the class remains the same. Consequently, *get* and *set* methods are very commonly used.

Static methods are used to access static data members. To declare a static method, just add the *static* keyword after the access specifier in a method declaration. To access the *count* variable in the *Star* class, add the following method:

```
public static int getCount() { return count; }
```

Method Parameters

Method parameters are used to pass information to an object from the outside world. The *setSize()* method, for example, passes an integer value from outside the class into the class method. The class method can now manipulate the *newSize* variable.

In Java, primitive types are passed by *value* while objects are passed by *reference*. Variables passed by value can have their contents changed without any noticeable difference outside the class. However, variables passed by reference can not only have their contents changed, but the change is effective outside the class method.

Return Types

Methods can return values, either primitive types or classes. You return a value from a method by specifying the return type in the class declaration and using the *return* keyword in the method code block. The *get* methods in class Star demonstrate how to return values from methods.

Constructors

Constructors are methods that allow you to customize an object at the moment of its instantiation. Without constructors, you'd have to explicitly set each attribute of an object using multiple accessor methods. For example:

```
  Star sun = new Star();
  sun.setSize(1390000); sun.setColor(Color.yellow);
  sun.setTemperature(5800);
```

If you have to create many objects, this repetitiveness can become very tiresome. A better approach is possible through the use of constructor methods. Constructor methods permit the passing of parameters to an object during its creation. Using a constructor, the previous code can be reduced to a single line:

```
Star sun = new Star(1390000, Color.yellow, 5800);
```

The format for declaring a constructor is as follows:

```
[AccessSpecifier]  ([Parameters]) {
  [Code]
}
```

Constructors are optional. If you do not include constructors in your code, Java generates a *default* constructor that initializes all primitive data members to zero and all object members to *null*. A null reference means that a particular object does not reference anything (i.e., has not been instantiated or assigned to another object).

Add the following lines of code to the *Star* class just before the getter and setter methods:

```
Star() { this(0, 0, Color.yellow); }
Star(int newSize) { this(newSize, 0, Color.yellow); }
Star(int newSize, int newTemp) { this(newSize, newTemp, Color.yellow); }
Star(int newSize, newTemp, newColor) {
    size=newSize; temperature = newTemp; color = newColor;
}
```

The Star class now has four constructors: the first constructor is a default constructor and the rest are constructors that accept a certain number of parameters. When a class has multiple constructors, the default constructor must be explicitly defined; otherwise, you cannot allocate objects using the default constructor (e. g., *new Star()*).

this

Notice the use of the keyword *this*. *This* is another way to refer to the current class from within a method. In the Star class definition, *this* is used to call the class constructor that expects three parameters. So:

```
this(newSize, newTemp, Color.yellow);
```

calls the fourth constructor in the Star class. The *this* keyword can also be used to refer to data members within its enclosing class. For example, inside of a Star method you can write:

```
this.size = 10;
```

This is commonly used to resolve name conflicts in methods. For example:

```
public void setSize(int size) {
    this.size = size;
}
```

It is preferable, however, to make the names of parameters distinct from class data member names, as is done in the Star class.

Inheritance

Inheritance is a technique whereby a class can inherit the attributes and methods of another class. The technical term for this is *specialization* . As the name implies, specialization defines more specific behavior for a class. It represents the *"is a"* relationship between classes. For example, a *Person* class has general attributes that apply to all people. A more specialized version of a *Person*, such as *Employee* , would have the same attributes as a *Person*, plus other data members and methods that are applicable to all employees (salary, etc.). Simply put, an employee *is a* person.

Inheritance in Java is specified using the *extends* keyword. You would create an Employee class which is an extension of the Person class as follows:

```
class Employee extends Person {
  // methods and attributes here
}
```

Protected members of a superclass are available to the subclass. They can be accessed without the use of getter and setter methods. This is illustrated below:

In Person.java:

```
public class Person {
  public String getName() { return name; }
  public void setName(String newName) { name = newName; }
  protected String name;
}
```

In Employee.java:

```
public class Employee extends Person {
  public void changeSuperName(String newName) {
    name = newName;
  }
}
```

Abstract Classes and Methods

Some classes are so general that they should not be allowed to have specific instances. These classes are called *abstract classes*. For example, consider a Shape class. In the real-world, you would not draw a shape. Instead, you'd draw a square, rectangle, or circle. However, these are all specializations of a shape. A Shape class would never be used to create an object, but would instead provide a framework for more specific shapes. An abstract Shape can be constructed as follows:

```
abstract class Shape {
  // shape methods here.
}
```

An abstract class may have methods just like an ordinary class, but additionally might have *abstract methods*. An abstract method is used for methods that are too general for a specific implementation but may be used to perform a more specific function in a derived, concrete class. As an illustration, consider the following Shape class declaration:

```
abstract class Shape {
  public abstract void draw(Graphics g);
}
```

Since many different representations can be used for a shape (squares, rectangles, octagons, etc.), it is impossible to implement a generic *draw()* method that can work for all types of shapes. However, a class derived from Shape, such as Square, can implement the generic *draw()* method to display itself.

Certain restrictions apply to abstract method declaration:

- Abstract methods cannot be private (otherwise, no subclass could implement the method).
- Constructors cannot be declared abstract (subclasses cannot implement superclass constructors).
- Static methods cannot be abstract (static methods refer to a class as a whole and thus cannot be implemented differently by various subclases).

Interfaces

Interfaces are similar to abstract classes, except that, unlike abstract classes which *can* have some methods which are fully implemented, *none* of the methods in an interface can have an implementation. Furthermore, interfaces can only have *final*, *static* data members. At first, these restrictions may appear to demote the usefulness of an interface. However, interfaces are powerful abstractions. Consider the Shape class discussed earlier. In the most recent JDK, the Shape class (in the `java.awt` package) is declared as an interface with a single method, *getBounds*, which returns a Rectangle object. This rectangle object provides access to its *x*, *y*, *width*, and *height* attributes (while this may seem to violate the rule of encapsulation, it is actually justifiable in this case). A class which *implements* this interface can specify how these fields are set and what they mean. For example, the Polygon (also in the java.awt package) class implements the Shape interface.

An interface is declared as follows:

```
interface  [extends [InterfaceName]] {
  // final, static attributes and
  // methods that must be implemented by a class or will
  // be inherited by another interface that extends this interface
}
```

Notice that an interface can inherit from another interface. A class cannot inherit from an interface or vice versa.

A class implements an interface by using the *implements* keyword, as illustrated below:

```
public class Polygon extends Object implements Shape {
  // See java.awt.Polygon for class attributes and methods.
}
```

The *extends* clause can be omitted from the above, since all classes inherit from the class *Object* by default. It was explicitly written to illustrate that a class can inherit from another class and simultaneously implement an interface.

A class can also implement multiple interfaces. The Polygon class actually implements two interfaces:

```
public class Polygon extends Object implements Shape, Serializable;
```

The serializable interface allows an object to write itself to a stream. We will discuss this mechanism in a later column.

Instanceof

You can find out if an object belongs to a particular class by using the *instanceof* keyword.

```
if(myObject instanceof MyClass) {
  // Perform some action if true.
}
```

As illustrated above, this keyword tests if the object on the left is an instance of the class specified on the right. The expression will return *true* or *false* depending on the outcome of the test.

A good example of when you'd want to use *instanceof* is inside the methods of an employee list that is a linked list which can contain only instances of class Employee, such as:

```
public class EmployeeList implements List {
  // some list methods
  public void add(Object data) {
    if(data instanceof Employee)
      super.add(data);
  }
}
```

Conclusion

This ends part 2 of a two part introduction to Java. In the next issue, we'll further explore the Java language, applets, and the Java class libraries.

References

1

Arnold, Ken and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.

2

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, Reading, Mass., 1996.

3

Lindholm, Tim and Frank Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, Reading, Mass., 1996.

4

Morrison, Michael, et al. *Java 1.1 Unleashed*, Sams.net, Indianapolis, In., 1997.

5

O'Connel, Michael. *Java: The Inside Story*, Web Publishing, Inc. July, 1995.

6

Stevens, W. Richard. *UNIX Network Programming*, Englewood Cliffs, New Jersey, 1990.

Biography

George Crawford III is a software engineer at MPI Software Technology, Inc. and completing his M.S. degree in computer science at Mississippi State University. His technical areas of interest include software design, software process management, distributed object technology, and games programming. He has been using Java since its alpha release in 1995.