Everything is an object

Monday, April 1, 2024

4:00 PM

**Creating new data types: classes**

The **class** keyword is followed by the name of the new type.

*class ATypeName { /\* Class body goes here \*/ }*

You can create an object of this type using **new**:

*ATypeName a = new ATypeName();*

**Fields and methods**

When you define a class (and all you do in Java is define classes, make objects of those classes, and send messages to those objects), you can put two types of elements in your class: ***fields*** (sometimes called data members), and ***methods*** (sometimes called member functions).

A field is an object of any type that you can talk to via its reference, or a primitive type. If it is a reference to an object, you must initialize that reference to connect it to an actual object (using new, as seen earlier).

*class DataOnly {*

*int i;*

*double d;*

*boolean b;*

*}*

*DataOnly data = new DataOnly();*

You can assign values to the fields, but you must first know how to refer to a member of an object. This is accomplished by stating the name of the object reference, followed by a period (dot), followed by the name of the member inside the object: objectReference.member.

*data.i = 47;*

*data.d = 1.1;*

*data.b = false;*

**Methods, arguments and return values**

Methods in Java determine the messages an object can receive. The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

*ReturnType methodName( /\* Argument list \*/ ) { /\* Method body \*/ }*

Methods in Java can be created only as part of a class. A method can be called only for an object, and that object must be able to perform that method call.

You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this:

*objectName.methodName(arg1, arg2, arg3);*

**Building a Java program**

**Name visibility**

To produce an unambiguous name for a library, the Java creators want you to use your Internet domain name in reverse since domain names are guaranteed to be unique. Since my (original) domain name was MindView.net, my utility Everything Is an Object 73 library of foibles would be named net.mindview.utility.foibles. After your reversed domain name, the dots are intended to represent subdirectories.

**Using other components**

You can tell the Java compiler exactly what classes you want by using the **import** keyword. **import** tells the compiler to bring in a package, which is a library of classes.

**The static keyword**

When you say something is **static**, it means that particular field or method is not tied to any particular object instance of that class. So even if you’ve never created an object of that class you can call a method or access a static field.

To make a field or method static, you simply place the keyword before the definition. For example, the following produces a static field and initializes it:

*class StaticTest {*

*static int i = 47;*

*}*

*StaticTest st1 = new StaticTest();*

There are two ways to refer to a static variable. As the preceding example indicates, you can name it via an object, by saying, for example, **st1.i**. You can also refer to it directly through its class name, something you cannot do with a non-static member:

*StaticTest.i++;*

***Using the class name is the preferred way to refer to a static variable***. Not only does it emphasize that variable’s static nature, but in some cases it gives the compiler better opportunities for optimization.

Similar logic applies to static methods. You can refer to a static method either through an object as you can with any method, or with the special additional syntax **ClassName.method( )**. You define a static method in a similar way:

*class Incrementable {*

*static void increment() { StaticTest.i++; }*

*}*

You can see that the Incrementable method increment( ) increments the static data i using the ++ operator. You can call increment( ) in the typical way, through an object:

*Incrementable sf = new Incrementable();*

*sf.increment();*

Or, because increment( ) is a static method, you can call it directly through its class:

*Incrementable.increment();*

**First Java program**

*// HelloDate.java*

*import java.util.\*;*

*public class HelloDate {*

*public static void main(String[] args) {*

*System.out.println("Hello, it's: ");*

*System.out.println(new Date());*

*}*

*}*

At the beginning of each program file, you must place any necessary **import** statements to bring in extra classes.

There’s a certain library of classes that are automatically brought into every Java file: **java.lang**.

Since **java.lang** is implicitly included in every Java code file, these classes are automatically available.

There’s no Date class listed in **java.lang**, which means you must import another library to use that. If you don’t know the library where a particular class is, or if you want to see all of the classes, you can select “Tree” in the Java documentation.

Use the browser’s “find” function to find Date. When you do you’ll see it listed as **java.util.Date**, which lets you know that it’s in the **util** library and that you must import **java.util.\*** in order to use Date.

If you go back to the beginning, select **java.lang** and then **System**, you’ll see that the System class has several fields, and if you select **out**, you’ll discover that it’s a static **PrintStream** object. Since it’s static, you don’t need to create anything with new. The **out** object is always there, and you can just use it. What you can do with this out object is determined by its type: **PrintStream**. Conveniently, **PrintStream** is shown in the description as a hyperlink, so if you click on that, you’ll see a list of all the methods you can call for **PrintStream**.

For now all we’re interested in is **println( )**, which in effect means “Print what I’m giving you out to the console and end with a newline.”

The name of the class is the same as the name of the file. When you’re creating a standalone program such as this one, one of the classes in the file must have the same name as the file. (The compiler complains if you don’t do this.) That class must contain a method called **main( )** with this signature and return type:

*public static void main(String[] args) {*

The **public** keyword means that the method is available to the outside world.

The argument to **main( )** is an array of String objects. The **args** won’t be used in this program, but the Java compiler insists that they be there because they hold the arguments from the command line.

*System.out.println(new Date());*

The argument is a **Date** object that is being created just to send its value (which is automatically converted to a String) to **println( )**.

When the statement is finished, that **Date** is unnecessary, and the garbage collector can come along and get it anytime.

**Compiling and running**

*javac HelloDate.java*

This command should produce no response.

*java HelloDate*

You’ll get the message and the date as output.

**Comments and embedded documentation**

There are two types of comments in Java. The first is the traditional C-style comment that was inherited by C++. These comments begin with a /\* and continue, possibly across many lines, until a \*/. Note that many programmers will begin each line of a continued comment with a \*:

/\* This is a comment

\* that continues

\* across lines \*/

The second form of comment comes from C++. It is the single-line comment, which starts with a // and continues until the end of the line.

The tool to extract the comments is called **Javadoc**, and it is part of the JDK installation.

It uses some of the technology from the Java compiler to look for special comment tags that you put in your programs. It not only extracts the information marked by these tags, but it also pulls out the class name or method name that adjoins the comment. This way you can get away with the minimal amount of work to generate decent program documentation.

The output of Javadoc is an HTML file that you can view with your Web browser.

In addition, you can write your own Javadoc handlers, called **doclets**, if you want to perform special operations on the information processed by Javadoc.

All of the Javadoc commands occur only within /\*\* comments. The comments end with \*/ as usual.

There are two primary ways to use Javadoc: Embed HTML or use “doc tags.”

* **Standalone doc tags** are commands that start with an ‘@’ and are placed at the beginning of a comment line.
* **Inline doc tags can appear anywhere within a Javadoc comment** and also start with an ‘@’ but are surrounded by curly braces.

There are three “types” of comment documentation, which correspond to the element the comment precedes: class, field or method.

*//: object/Documentation1.java*

*/\*\* A class comment \*/*

*public class Documentation1 {*

*/\*\* A field comment \*/*

*public int i; /*

*\*\* A method comment \*/*

*public void f() {}*

*}*

Note that Javadoc will process comment documentation ***for only public and protected members***. Comments for ***private and package-access members*** (see the Access Control chapter) are ignored, and you’ll see no output.

**Coding style**

The style described in the Code Conventions for the Java Programming Language6 is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don’t use underscores to separate the names), and the first letter of each embedded word is capitalized (camel-casing).

*class AllTheColorsOfTheRainbow { // …*

For almost everything else— methods, fields (member variables), and object reference names—the accepted style is just as it is for classes ***except that the first letter of the identifier is lowercase.***