Chapter 1

A Crash Course from C++ to Java

CHAPTER TOPICS

- "Hello, World" in Java
- Documentation Comments
- Primitive Types
- Control Flow Statements
-
-
-
-
-
-
-
- ods
- yle

The purpose of this chapter is to teach you the elements of Java—or to give you an opportunity to review them—assuming that you know an object-oriented programming language. In particular, you should be familiar with the concepts of classes and objects. If you know C++ and

understand classes, member functions, and constructors, then you will find that it is easy to make the switch to Java.

1.1 "Hello, World!" in Java

Classes are the building blocks of Java programs. Let's start our crash course by looking at a simple but typical class:

Greeter.java

```
1  public class Greeter
2  {
3     public Greeter(String aName)
4     {
5        name = aName;
6     }
7
8     public String sayHello()
9     {
10       return "Hello, " + name + "!";
11    }
12
13    private String name;
14 }
```

This class has three features:

- A *constructor* `Greeter(String aName)` that is used to construct new objects of this class.
- A *method* `sayHello( )` that you can apply to objects of this class. (Java uses the term "method" for a function defined in a class.)
- A *field* `name` that is present in every object of this class

Each feature is tagged as `public` or `private`. Implementation details (such as the `name` field) are private. Features that are intended for the class user (such as the constructor and `sayHello` method) are public. The class itself is declared as public as well. You will see the reason in the section on packages.

To construct an object, you use the `new` operator, followed by a call to the constructor.

```
new Greeter("World")
```

The `new` operator returns the constructed object, or, more precisely, a reference to that object—we will discuss that distinction in detail in the section on object references.

You can invoke a method on that object. The call

```
new Greeter("World").sayHello()
```

returns the string `"Hello, World!"`, the concatenation of the strings `"Hello, "`, `name`, and `"!"`.

More commonly, you store the value that the `new` operator returns in an object variable

```
Greeter worldGreeter = new Greeter("World");
```

Then you invoke a method as

```
String greeting = worldGreeter.sayHello();
```

Now that you have seen how to define a class, let's build our first Java program, the traditional program that displays the words "Hello, World!" on the screen.

We will define a second class, `GreeterTest`, to produce the output.

GreeterTest.java

```
1 public class GreeterTest
2 {
3    public static void main(String[] args)
4    {
5       Greeter worldGreeter = new Greeter("World");
6       String greeting = worldGreeter.sayHello();
7       System.out.println(greeting);
8    }
9 }
```

This class has a `main` method, which is required to start a Java application. The `main` method is *static*, which means that it doesn't operate on an object. After all, when the application first starts, there aren't any objects yet. It is the job of the `main` method to construct the objects that are needed to run the program.

The `args` parameter of the `main` method holds the *command line arguments*. We will discuss it in the section on arrays.

You have already seen the first two statements inside the `main` method. They construct a `Greeter` object, store it in an object variable, invoke the `sayHello` method and capture the result in a string variable. The final statement uses the `println` method of the `System.out` object to print the message and a newline to the standard output stream.

To build and execute the program, put the `Greeter` class inside a file `Greeter.java` and the `GreeterTest` class inside a separate file `GreeterTest.java`. The directions for compiling and running the program depend on your development environment.

The Java Software Development Kit (SDK) from Sun Microsystems is a set of command-line programs for compiling, running, and documenting Java programs. Versions for several platforms are available at `http://java.sun.com/j2se`. If you use the Java SDK, then follow these instructions:

1. Create a new directory of your choice to hold the program files.
2. Use a text editor of your choice to prepare the files `Greeter.java` and `GreeterTest.java`. Place them inside the directory you just created.
3. Open a shell window.
4. Use the `cd` command to change to the directory you just created
5. Run the compiler with the command

   ```
   javac GreeterTest.java
   ```

   If the Java compiler is not on the search path, then you need to use the full path (such as `/usr/local/j2sdk1.4/bin/javac` or `c:\j2sdk1.4\bin\javac`) instead of just `javac`. Note that the `Greeter.java` file is automatically compiled as well since

the `GreeterTest` class requires the `Greeter` class. If any compilation errors are reported, then make a note of the file and line numbers and fix them.

6. Have a look at the files in the current directory. Verify that the compiler has generated two *class files*, `Greeter.class` and `GreeterTest.class`.
7. Start the Java interpreter with the command

   ```
   java GreeterTest
   ```

Now you will see a message "Hello, World!" in the shell window (see Figure 1).

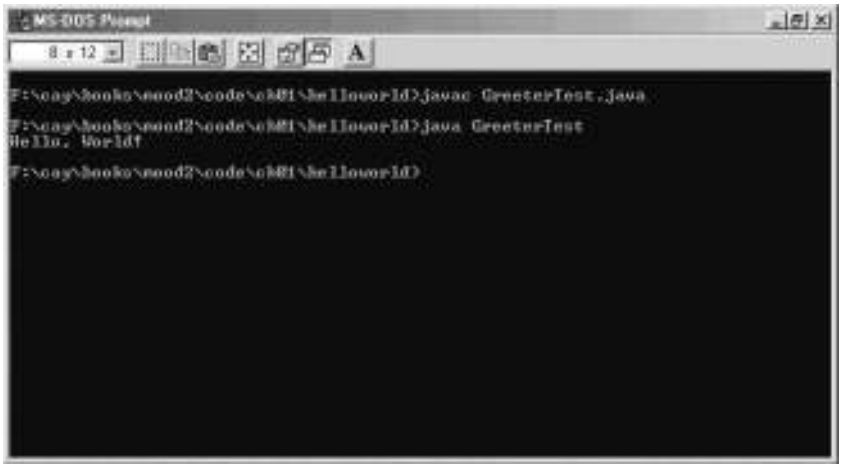

Figure 1

Running the "Hello, World!" Program in a Shell Window

The structure of this program is typical for a Java application. The program consists of a collection of classes. One class has a `main` method. You run the program by launching the Java interpreter with the name of the class whose `main` method contains the instructions for starting the program activities.

The BlueJ development environment, developed at Monash University, lets you test classes without having to write a new program for every test. BlueJ supplies an interactive environment for constructing objects and invoking methods on the objects. You can download BlueJ from `http://www.bluej.org`.

With BlueJ, you don't need a `GreeterTest` class to test the `Greeter` class. Instead, just follow these steps.

1. Select "Project -> New…" from the menu, point the file dialog to a directory of your choice and type in the name of the subdirectory that should hold your classes. This must be the name of a new directory. BlueJ will create it.
2. Click on the "New Class…" button and type in the `Greeter` class.
3. Click on the "Compile" button to compile the class. Click on the "Close" button.
4. The class is symbolized as a rectangle. Right-click on the class rectangle, and select "new Greeter(aName)" to construct a new object. Call the object

`worldGreeter` and supply the constructor parameter `"Hello"` (including the quotation marks).

5. The object appears in the object workbench. Right-click on the object rectangle and select "String sayHello( )" to execute the `sayHello` method.
6. A dialog box appears to display the result (see Figure 2).

Figure 2

Testing a Class with BlueJ

As you can see, BlueJ lets you think about objects and classes without fussing with `public static void main`.

1.2 Documentation Comments

Java has a standard form for documenting comments that describe methods and classes. The Java SDK contains a tool, called `javadoc`, that automatically generates a neat set of HTML pages that document your classes.

Documentation comments are delimited by `/**` and `*/`. Both class and method comments start with freeform text. The `javadoc` utility copies the *first* sentence of each comment to a summary table. Therefore, it is best to write that first sentence with some care. It should start with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Method comments contain additional information. For each method parameter, you supply a line that starts with `@param`, followed by the parameter name and a short explanation. Supply a line that starts with `@return`, describing the return value. You omit the `@param` tag for methods that have no parameters, and you omit the `@return` tag for methods whose return type is `void`.

Here is the `Greeter` class with documentation comments for the class and its public interface.

```
/**
   A class for producing simple greetings.
*/
class Greeter
{
   /**
      Constructs a Greeter object that can greet a person or entity.
      @param aName the name of the person or entity who should
      be addressed in the greetings.
   */
   public Greeter(String aName)
   {
      name = aName;
   }

   /**
      Greet with a "Hello" message.
      @return a message containing "Hello" and the name of
      the greeted person or entity.
   */
   public String sayHello()
   {
      return "Hello, " + name + "!";
   }

   private String name;
}
```

Your first reaction may well be "Whoa! I am supposed to write all this stuff?" These comments do seem pretty repetitive. But you should still take the time to write them, even if it feels silly at times. There are three reasons.

First, the `javadoc` utility will format your comments into a neat set of HTML documents. It makes good use of the seemingly repetitive phrases. The first sentence of each method comment is used for a *summary table* of all methods of your class (see Figure 3). The `@param` and `@return` comments are neatly formatted in the detail description of each method (see Figure 4). If you omit any of the comments, then `javadoc` generates documents that look strangely empty.

Next, it is actually easy to spend more time pondering whether a comment is too trivial to write than it takes just to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment can cause real grief to future maintenance programmers. According to the standard Java documentation style, *every* class, *every* method, *every* parameter, and *every* return value should have a comment.