# Chapter 1

# A Crash Course
# from C++ to Java

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
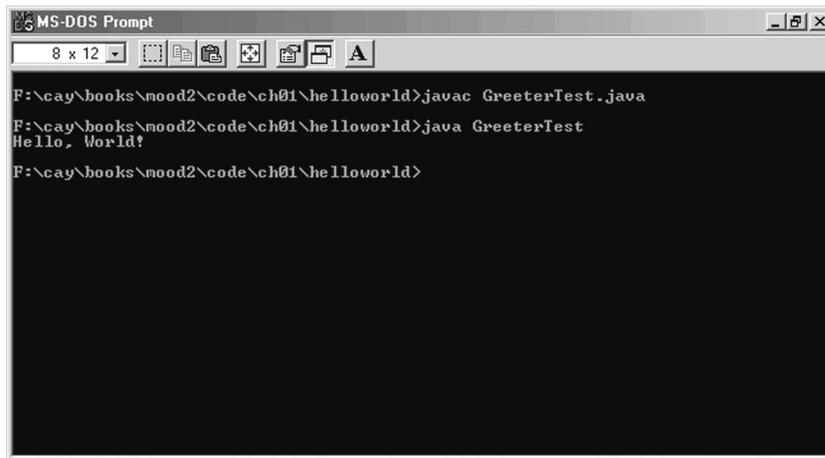
   ```
   javac GreeterTest.java
   ```

   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).

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 quota-
tion marks).

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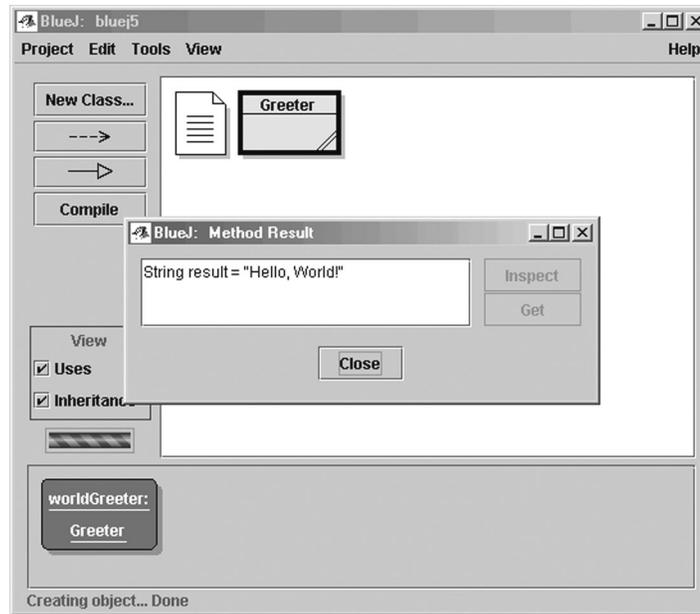6. A dialog box appears to display the result (see Figure 2).



**Figure 2**

Testing a Class with BlueJ

As you can see, BlueJ lets you think about objects and classes without fussing with pub-
lic 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 com-
ments start with freeform text. The javadoc utility copies the *first* sentence of each com-
ment 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.
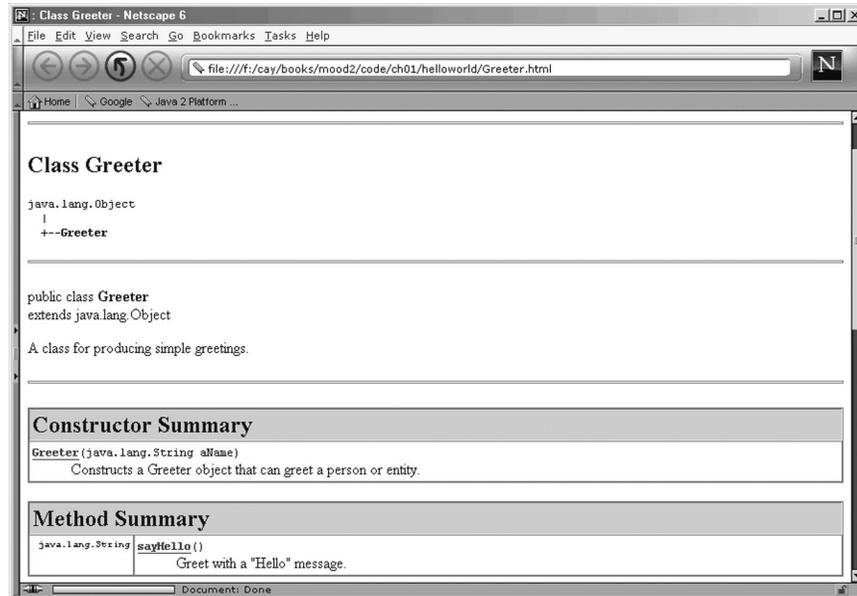
Figure 3

A javadoc Class Summary



Figure 4

Parameter and Return Value Documentation in javadoc

Finally, it is always a good idea to write the method comment *first*, before writing the method code. This is an excellent test to see that you firmly understand what you need to

program. If you can't explain what a class or method does, you aren't ready to implement it.

Once you have written the documentation comments, invoke the `javadoc` utility.

1. Open a shell window.
2. Use the `cd` command to change to the directory you just created
3. Run the `javadoc` utility

   `javadoc *.java`

   If the SDK tools are not on the search path, then you need to use the full path (such as `/usr/local/j2sdk1.4/bin/javadoc` or `c:\j2sdk1.4\bin\javadoc`) instead of just `javadoc`.

The `javadoc` utility then produces one HTML file for each class (such as `Greeter.html` and `GreeterTest.html`) as well as a file `index.html` and a number of other summary files. The `index.html` file contains links to all classes.

The `javadoc` tool is wonderful because it does one thing right: it lets you put the documentation *together with your code*. That way, when you update your programs, you can see immediately which documentation needs to be updated. Hopefully, you will then update it right then and there. Afterwards, run `javadoc` again and get a set of nicely formatted HTML pages with the updated comments.

---

**INTERNET** The `DocCheck` program reports any missing `javadoc` comments. Download it from `http://java.sun.com/j2se/javadoc/doccheck/`.

---

The Java SDK contains the documentation for *all* classes in the Java library, also called the application programming interface or API. Figure  shows the documentation of the `String` class. This documentation is directly extracted from the library source code. The programmers who wrote the Java library documented every class and method and then simply ran `javadoc` to extract the HTML documentation.

Download the SDK documentation from `http://java.sun.com/j2se`. Install the documentation into the same location as the Java SDK.  Point your browser to the `docs/api/index.html` file inside your Java SDK directory, and make a bookmark. Do it now! You will need to access this information very frequently.

Figure 5

The Java API
Documentation
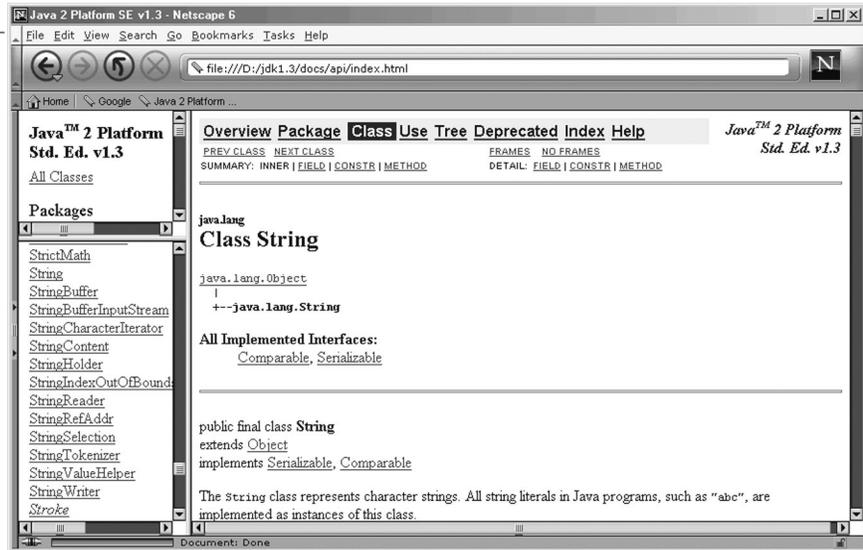


## 1.3    Primitive Types

In Java, numbers, characters, and Boolean values are not objects but values of a primitive type. Table 1 shows the 8 primitive types of the Java language.

| Type | Size | Range |
|------|------|-------|
| `int` | 4 bytes | −2,147,483,648 . . . 2,147,483,647 |
| `long` | 8 bytes | −9,223,372,036,854,775,808L . . . 9,223,372,036,854,775,807L |
| `short` | 2 bytes | -32768 . . . 32767 |
| `byte` | 1 byte | -128 . . . 127 |
| `char` | 2 bytes | `'\u0000' - '\uFFFF'` |
| `boolean` | 1 byte | `false, true` |
| `double` | 8 bytes | approximately ±1.79769313486231570E+308 |
| `float` | 4 bytes | approximately ±3.40282347E+38F |

Table 1

The Primitive Types of the Java Language

Note that `long` constants have a suffix `L` and `float` constants have a suffix `F`, such as `10000000000L` or `3.1415927F`.

Characters are encoded in Unicode, an uniform encoding scheme for characters in many languages around the world. Character constants are enclosed in single quotes, such as `'a'`. Several characters, such as a newline `'\n'`, are represented as 2-character escape sequences. Table 2 shows all permitted escape sequences. Arbitrary Unicode characters are denoted by a `\u`, followed by four hexadecimal digits, enclosed in quotes. For example, `'\u2122'` is the trademark symbol (TM).

| Escape Sequence | Meaning |
|:---:|:---:|
| \b | backspace (\u0008) |
| \f | form feed (\u000C) |
| \n | newline (\u000A) |
| \r | return (\u000D) |
| \t | tab (\u0009) |
| \\ | backslash |
| \' | single quote |
| \" | double quote |
| \u$n_1 n_2 n_3 n_4$ | Unicode encoding |

**Table 2**

**Character Escape Sequences**

**INTERNET**  You can find the encodings of tens of thousands of letters in many alphabets at `http://www.unicode.org`.

Conversions that don't incur information loss (such as `short` to `int` or `float` to `double`) are always legal. Values of type `char` can be converted to `int`. All integer types can be converted to `float` or `double`, even if it means a loss of precision. All other conversions require a *cast* such as

```
double x = 10.0 / 3.0; // sets x to 3.3333333333333335
int n = (int)x; // sets n to 3
float f = (float)x; // sets f to 3.3333333
```

It is not possible to convert between the `boolean` type and number types.

The `Math` class implements useful mathematical functions—see Table 3. The methods of the `Math` class are static methods. That is, they do not operate on objects. (Recall that numbers are not objects in Java.) For example, here is how you call the `sqrt` method:

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

Since the method doesn't operate on an object, you must supply the class name to tell the compiler that the sqrt method is in the Math class. In Java, every method must belong to some class.

| Method | Description |
|---|---|
| Math.sqrt(x) | square root of $x$, $\sqrt{x}$ |
| Math.pow(x, y) | $x^y \left(x > 0, \text{ or } x = 0 \text{ and } y > 0, \text{ or } x < 0 \text{ and } y \text{ is an integer}\right)$ |
| Math.sin(x) | sine of $x$ $\left(x \text{ in radians}\right)$ |
| Math.cos(x) | cosine of $x$ |
| Math.tan(x) | tangent of $x$ |
| Math.asin(x) | $\left(\text{arc sine}\right) \sin^{-1} x \in \left[-\pi/2,\ \pi/2\right], \ x \in \left[-1,\ 1\right]$ |
| Math.acos(x) | $\left(\text{arc cosine}\right) \arccos^{-1} x \in \left[0,\ \pi\right], \ x \in \left[-1,\ 1\right]$ |
| Math.atan(x) | $\left(\text{arc tangent}\right) \tan^{-1} x \in \left(-\pi/2,\ \pi/2\right)$ |
| Math.atan2(y,x) | $\left(\text{arc tangent}\right) \tan^{-1}\left(y/x\right) \in \left[-\pi/2,\ \pi/2\right], \ x \text{ may be } 0$ |
| Math.toRadians(x) | converts $x$ degrees to radians $\left(\text{i.e. returns } x \cdot p/180\right)$ |
| Math.toDegrees(x) | converts $x$ radians to degrees $\left(\text{i.e. returns } x \cdot 180/p\right)$ |
| Math.exp(x) | $e^x$ |
| Math.log(x) | $\left(\text{natural log}\right) \ln\left(x\right), \ x > 0$ |
| Math.round(x) | the closest integer to $x$ $\left(\text{as a long}\right)$ |
| Math.ceil(x) | smallest integer $\geq x$ |
| Math.floor(x) | largest integer $\leq x$ |
| Math.abs(x) | absolute value $\left|x\right|$ |

**Table 3**

Mathematical Methods

## 1.4    Control Flow Statements

The if statement is used for conditional execution. The else branch is optional.

```
if (x >= 0) y = Math.sqrt(x); else y = 0;
```

The while and do statements are used for loops. The body of a do loop is executed at least once.

```
while (x < target)
{
   x = x * a;
   n++;
}

do
{
   x = x * a;
   n++;
}
while (x < target);
```

The for statement is used for loops that are controlled by a loop counter.

```
for (i = 1; i <= n; i++)
{
   x = x * a;
   sum = sum + x;
}
```

You can define a variable in a for loop. Its scope extends to the end of the loop.

```
for (int i = 1; i <= n; i++)
{
   x = x * a;
   sum = sum + x;
}
// i no longer defined here
```

## 1.5   Object References

In Java, an object value is always a *reference* to an object, or, in other words, a value that describes the location of the object. For example, consider the statement

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

The value of the new expression is the location of the newly constructed object. The variable worldGreeter can hold the location of any Greeter object, and it is being filled with the location of the new object (see Figure 6.)
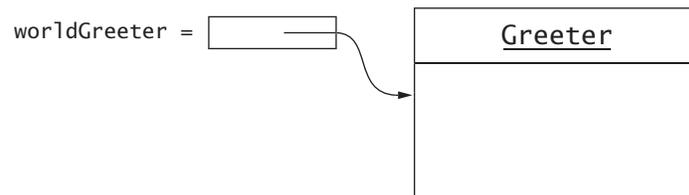


**Figure 6**

An Object Reference

You can have multiple references to the same object. For example, after the assignment

```
Greeter anotherGreeter = worldGreeter;
```

the two object variables both share a single object (see Figure 7).
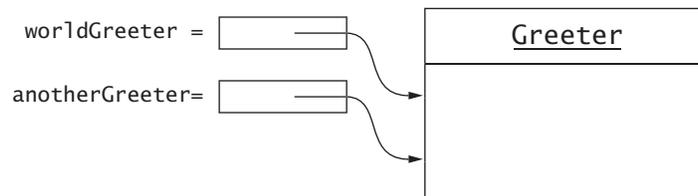


worldGreeter =

anotherGreeter=

Greeter

### A Shared Object

If the `Greeter` class has a method that allows modification of the object (such as a `setName` method), and if that method is invoked on the object reference, then all shared references access the modified object.

```
anotherGreeter.setName("Dave");
// now worldGreeter also refers to the changed object
```

The special reference `null` refers to no object. You can set an object variable to `null`:

```
worldGreeter = null;
```

You can test if an object reference is currently `null`:

```
if (worldGreeter == null) ...
```

If you invoke a method on a `null` reference, an *exception* is thrown. Unless you supply a handler for the exception, the program terminates.

It can happen that an object has no references pointing to it, namely when all object variables that are referred to are filled with other values or have been recycled. In that case, the object memory will be automatically reclaimed by the garbage collector. In Java, you never need to manually recycle memory.

## 1.6    Parameter Passing

In Java, a method can modify the state of an object parameter because the corresponding parameter variable is set to a copy of the passed object reference. Consider this contrived method of the `Greeter` class:

```
/**
    Sets another greeter's name to this greeter's name.
    @param other a reference to the other Greeter
*/
public void setName(Greeter other)
{
    other.name = name;
}
```

Now consider this call:

```
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
worldGreeter.setName(daveGreeter);
```

Figure 8 shows how the `other` parameter variable is initialized with a copy of the `daveGreeter` reference. The `setName` method changes `other.name`, and after the method returns, `daveGreeter.name` has been changed.
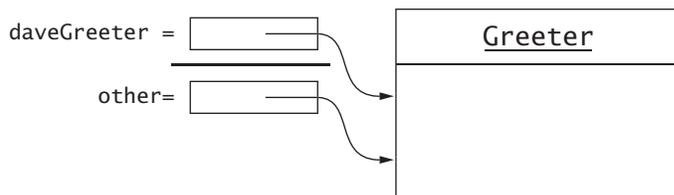


**Figure 8**

Accessing an Object through a Parameter Variable

While a method can change the *state* of an object that is passed as a parameter, it can never update the *contents* of any variable. If you call

```
obj.f(var);
```

where `var` is a variable, then the contents of `var` is the same number, `Boolean` value, or object location before and after the call. In other words, it is impossible to write a method that sets the current value of `var` with to another primitive type value or object location.

Consider yet another set of contrived methods:

```
/**
   Tries to copy the length of this greeter's name into an integer variable.
   @param n  the variable into which the method tries to copy the length
*/
public void setLength(int n)
{
   // this assignment has no effect outside the method
   n = name.length();
}

/**
   Tries to set another Greeter object to a copy of this object.
   @param other  the Greeter object to initialize
*/
public void setGreeter(Greeter other)
{
   // this assignment has no effect outside the method
   other = new Greeter(name);
}
```

Let's call these two methods:

```
int length = 0;
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
worldGreeter.setLength(length); // has no effect on the contents of length
worldGreeter.setGreeter(daveGreeter);
    // has no effect on the contents of daveGreeter
```

Neither call has any effect. Changing the value of the parameter variable does not affect the variable supplied in the method call. Thus, Java has no call by reference. Both primitive types and object references are passed by value.

## 1.7    Packages

Java classes can be grouped into packages. Package names are dot-separated sequences of identifiers, such as

```
java.util
javax.swing
com.sun.misc
edu.sjsu.cs.cs151.alice
```

To guarantee uniqueness of package names, Sun recommends that you start a package name with a domain name in reverse (such as com.sun or edu.sjsu.cs), since domain names are guaranteed to be unique. Then use some other mechanism within your organization to ensure that the remainder of the package name is unique as well.

You place a class inside a package by adding a package statement to the top of the source file:

```
package edu.sjsu.cs.cs151.alice;
public class Greeter
{
    ...
}
```

Any class without a package statement is in the "default package" with no package name.

The *full* name of a class consists of the package name followed by the class name, such as edu.cs.cs151.alice.Greeter. Examples from the Java library are java.util.ArrayList and javax.swing.JOptionPane.

It is tedious to use these full names in your code. Use the import statement to use the shorter class name instead. For example, after you place a statement

```
import java.util.ArrayList;
```

into your source file, then you can refer to the class simply as ArrayList. If you simultaneously use two classes with the same short name (such as java.util.Date and java.sql.Date), then you are stuck—you must use the full name.

You can also import all classes from a package:

```
import java.util.*;
```

However, you never need to import the classes in the java.lang package, such as String or Math.

Large programs consist of many classes in multiple packages. The class files must be located in subdirectories that match the package name. For example, the class file `Greeter.class` for the class

    edu.sjsu.cs.cs151.alice.Greeter

must be in a subdirectory

    edu/sjsu/cs/cs151/alice

of the project's *base directory*. The base directory is the directory that contains all package directories as well as any classes that are contained in the default package (that is, the package without a name).
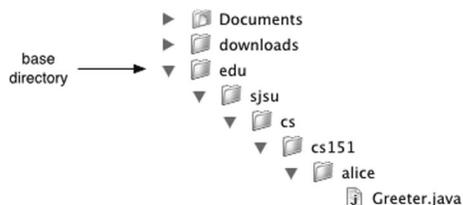


**Figure 9**

Package Name Must Match the Directory Path

Always compile from the base directory, for example

    javac edu/sjsu/cs/cs151/alice/Greeter.java

or

    javac edu\sjsu\cs\cs151\alice\Greeter.java

Then the class file is automatically placed in the correct location.

To run a program, you must start the Java interpreter in the base directory and specify the full class name of the class that contains the main method:

    java edu.sjsu.cs.cs151.alice.Greeter.java

## 1.8   Basic Exception Handling

When a program carries out an illegal action, an *exception* is generated. Here is a common example. Suppose you initialize a variable with the `null` reference, intending to assign an actual object reference later. But then you accidentally use the variable when it is still `null`.

```
String name = null;
int n = name.toUpperCase(); // ERROR
```

This is an error. You cannot apply a method call to `null`. The virtual machine now throws a `NullPointerException`. Unless your program handles this exception, it will terminate after displaying a *stack trace* such as this one:

```
Exception in thread "main" java.lang.NullPointerException
     at Greeter.sayHello(Greeter.java:25)
     at GreeterTest.main(GreeterTest.java:6)
```

Different programming errors lead to different exceptions. For example, trying to open a file with an illegal file name causes an `IOException`.

The Java programming language makes an important distinction between two kinds of exceptions, called *checked exceptions* and *unchecked exceptions*. The `NullPointerException` is an example of an unchecked exception. That is, the compiler does not check that your code handles the exception. If the exception occurs, it is detected at runtime and may terminate your program. The `IOException`, on the other hand, is a checked exception. If you call a method that might generate this exception, you must also specify how you want the program to deal with this failure.

In general, a checked exception is caused by an external cause beyond the programmer's control. Exceptions that occur during input and output are generally checked because the file system or network may spontaneously cause problems that the programmer cannot control. Therefore, the compiler insists that the programmer provides code to handle these situations.

On the other hand, unchecked exceptions are generally the programmer's fault. You should never get a `NullPointerException`. Therefore, the compiler doesn't tell you to provide a handler for a `NullPointerException`. Instead, you should spend your energy on making sure that the error doesn't occur in the first place. Either initialize your variables properly, or test that they aren't `null` before making a method call.

Whenever you write code that might cause a checked exception, you must take one of two actions:

1. Declare the exception in the method header (preferred)
2. Catch the exception

Consider this example. You want to read data from a file.

```
public void read(String filename)
{
   FileReader r = new FileReader(filename);
   ...
}
```

If there is no file with the given name, the `FileReader` constructor throws an `IOException`. Because it is a checked exception, the compiler insists that you handle it. The preferred remedy is to let the exception *propagate to its caller*. That means that the `read` method terminates, and that the exception is thrown to the method that called it.

Whenever a method propagates a checked exception, you must declare the exception in the method header, like this:

```
public void read(String filename) throws IOException
{
   FileReader r = new FileReader(filename);
   ...
}
```

There is no shame associated with acknowledging that your method might throw a checked exception—it is just "truth in advertising".

If a method can throw multiple exceptions, you list them all, separated by commas:

```
public void read(String filename)
    throws IOException, ClassNotFoundException
```

Whoever calls this method is now put on notice that there is the possibility that the checked exception in the `throws` clause may occur. Of course, those calling methods also need to deal with these exceptions. Generally, the calling methods also add `throws` declarations. When you carry this process out for the entire program, the `main` method ends up being tagged as well:

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException
{
    ...
}
```

If an exception actually occurs, the `main` method is terminated, a stack trace is displayed, and the program exits.

However, if you write a professional program, you do not want the program to terminate whenever a user supplies an invalid file name. In that case, you want to *catch* the exception. Use the following syntax:

```
try
{
    ...
    code that might throw an IOException
    ...
}
catch (IOException exception)
{
    take corrective action
}
```

An appropriate corrective action might be to display an error message and to inform the user that the attempt to read the file has failed.

In most programs, the lower-level methods simply propagate exceptions to their callers. Some higher-level method, such as `main` or a part of the user interface, catches exceptions and informs the user.

Occasionally, you need to catch an exception in a place where you can't take corrective action. For now, simply print a stack trace and exit the program. For example,

```
try
{
    ...
    code that might throw an IOException
    ...
}
catch (IOException exception)
{
```

```
        // can't take corrective action
        exception.printStackTrace();
        System.exit(0);
    }
```

To do a better job, you need to know more about exception handling. We will cover the details in Chapter 6.

## 1.9   Array Lists and Arrays

The `ArrayList` class of the `java.util` package lets you collect a sequence of objects of any type. The `add` method adds an object to the end of the array list.

```
ArrayList countries = new ArrayList();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

The `size` method returns the number of objects in the array list. The `get` method returns the object at a given position; legal positions range from 0 to `size() - 1`. However, the return type of the `get` method is `Object`, the common superclass of all classes in Java. Therefore, you need to remember the type of the objects that you put into a particular array list and *cast* the returned value to that type. For example,

```
for (int i = 0; i < countries.size(); i++)
{
    String country = (String)countries.get(i);
    ...
}
```

The `set` method lets you overwrite an existing element with another:

```
countries.set(1, "France");
```

If you access a nonexistent position (< 0 or >= size()), then an `IndexOutOfBoundsException` is thrown.

Finally, you can insert and remove objects in the middle of the array list.

```
countries.insert(1, "Germany");
countries.remove(0);
```

These operations move the remaining elements up or down. The name "array list" signifies that the public interface allows both array operations (`get`/`set`) and list operations (`insert`/`remove`).

Array lists have one drawback—they can only hold objects, not values of primitive types. Arrays, on the other hand, can hold sequences of arbitrary values. You construct an array as

```
new T[n]
```

where `T` is any type and `n` any integer-valued expression. The returned array has type `T[]`. For example,

```
int[] numbers = new int[10];
```

Now `numbers` is a reference to an array of 10 integers—see Figure 10. When an array is constructed, its elements are set to 0, `false` or `null`.
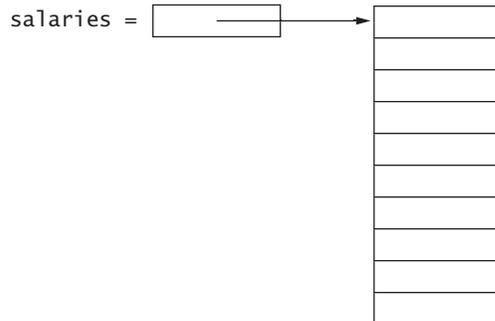
```
salaries =
```

**Figure 10**

**An Array Reference**

The length of an array is stored in the `length` field.

```
int length = numbers.length
```

Note that an empty array of length 0

```
new int[0]
```

is different from `null`—a reference to no array at all.

You access an array element by enclosing the index in brackets, such as

```
int number = numbers[i];
```

If you access a nonexistent position (`< 0` or `>= length`), then an `ArrayIndexOutOf-BoundsException` is thrown.

After an array has been constructed, you cannot change its length. If you want a larger array, you have to construct a new array and move the elements from the old array to the new array. That's a tremendous hassle and most programmers use an `ArrayList` to store objects.

You can obtain a two-dimensional array with a constructor call such as

```
int[][] table = new int[10][20];
```

Access the elements as `table[row][column]`.

The `args` parameter of the `main` method is an array of strings, the strings specified in the command line. `args[0]` is the first string after the class name. For example, if you invoke a program as

```
java GreeterTest Mars
```

then `args.length` is 1 and `args[0]` is `"Mars"` and not `"java"` or `"GreetingTest"`.

## 1.10    Strings

Java strings are sequences of Unicode characters. The `charAt` method yields the charac-
ters of a string. String positions start at 0.

```
String greeting = "Hello";
char ch = greeting.charAt(1); // sets ch to 'e'
```

Java strings are *immutable*. Once created, a string cannot be changed. Thus, there is no
`setCharAt` method. That sounds like a severe restriction, but in practice it isn't. For
example, suppose you initialized `greeting` to `"Hello"`. You can still change your mind:

```
greeting = "Goodbye";
```

The string object `"Hello"` hasn't changed, but `greeting` now refers to a different string
object.

The `length` method yields the length of a string. For example, `"Hello".length()` is 5.

Note that the empty string `""` of length 0 is different from `null`—a reference to no string
at all.

The `substring` method computes substrings of a string. You need to specify the positions
of the first character that you want to include in the substring and the first character that
you no longer want to include. For example, `"Hello".substring(1, 3)` is the string `"el"`
(see Figure 11). Note that the difference between the two positions equals the length of
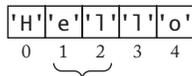the substring.



**Figure 11**

Extracting a Substring

Since strings are objects, you need to use the `equals` method to compare whether two
strings have the same contents.

```
if (greeting.equals("Hello")) ... // OK
```

If you use the `==` operator, you only test whether two string references have the identical
*location*. For example, the following test fails:

```
if ("Hello".substring(1, 3) == "el") ... // NO
```

The substring is not at the same location as the constant string `"el"`, even though it has
the same contents.

If the substrings of a string are separated by a delimiter such as commas or white space,
you can use the `StringTokenizer` class from the `java.util` package to enumerate the
substrings. For example,

```
String countries = "Germany,France,Italy";
StringTokenizer tokenizer = new StringTokenizer(countries, ",");
while (tokenizer.hasMoreTokens())
{
```

```
    String country = tokenizer.nextToken();
    ...
}
```

If you don't supply a delimiter in the constructor, then tokens are delimited by white space.

You have already seen the string concatenation operator: `"Hello, " + name` is the concatenation of the string `"Hello,"` and the string object to which `name` refers.

If either argument of the + operator is a string, then the other is *converted to a string*. For example,

```
int n = 7;
String greeting = "Hello, " + n;
```

constructs the string `"Hello, 7"`.

If a string and an object are concatenated, then the object is converted by the `toString` method. Every class inherits a default implementation of the `toString` method from the `Object` superclass, and it is free to override it to return a more appropriate string. For example, the `toString` method of the `Date` class in the `java.util` package returns a string containing the date and time that is encapsulated in the `Date` object. Here is what happens when you concatenate a string and a `Date` object:

```
// default Date constructor sets current date/time
Date now = new Date();
String greeting = "Hello, " + now;
// greeting is a string such as "Hello, Wed Jan 17 16:57:18 PST 2001"
```

Sometimes, you have a string that contains a number, for example the string `"7"`. To convert the string to its number value, use the `Integer.parseInt` and `Double.parseDouble` methods. For example,

```
String input = "7";
n = Integer.parseInt(input); // sets n to 7
```

If the string doesn't contain a number, or contains additional characters besides a number, a `NumberFormatException` is thrown.

## 1.11   Reading Input

The simplest way to read input in a Java program is to use the static `showInputDialog` method in the `JOptionPane` class. You supply a prompt string as a parameter. The method displays a dialog (see Figure 12) and returns the string that the user provided, or `null` if the user canceled the dialog. To read a number, you need to convert the string to a number with `Integer.parseInt` or `Double.parseDouble`. For example,

```
String input = JOptionPane.showInputDialog("How old are you?");
if (input != null) age = Integer.parseInt(input);
```

Admittedly, it is a bit strange to have dialogs pop up for every input. Also, you can't use input redirection to supply input from a text file. To read from standard input, you have to work a little harder.

An Input Dialog

Console input reads from the System.in object. However, unlike System.out, which was ready-made for printing numbers and strings, System.in can only read *bytes*. Keyboard input consists of characters. To get a reader for characters, you have to turn System.in into an InputStreamReader object, like this:

```
InputStreamReader reader = new InputStreamReader(System.in);
```

An input stream reader can read characters, but it can't read a whole string at a time. That makes it pretty inconvenient—you wouldn't want to piece together every input line from its individual characters. To overcome this limitation, turn an input stream reader into a BufferedReader object:

```
BufferedReader console = new BufferedReader(
    new InputStreamReader(System.in));
```

Now you use the readLine method to read an input line, like this:

```
System.out.println("How old are you?");
String input = console.readLine();
int count = Integer.parseInt(input);
```

The readLine method may throw an IOException if there is an input error. When the end of the input is reached, the readLine method returns null.

## 1.12    Static Fields and Methods

Occasionally, you would like to share a variable among all objects of a class. Here is a typical example. The Random class in the java.util package implements a random number generator. It has methods such as nextInt, nextDouble, and nextBoolean, that return random integers, floating-point numbers, and Boolean values. For example, here is how you print 10 random integers:

```
Random generator = new Random();
for (int i = 1; i <= 10; i++)
    System.out.println(generator.nextInt());
```

Let's use a random number generator in the Greeter class:

```
public String saySomething()
{
```

```
      if (generator.nextBoolean())
         return "Hello, " + name + "!";
      else
         return "Goodbye, " + name + "!";
   }
```

It would be wasteful to supply each `Greeter` object with its own random number generator. To share one generator among all `Greeter` objects, declare the field as `static`:

```
   public class Greeter
   {
      ...
      private static Random generator = new Random();
   }
```

Shared variables such as this one are relatively rare. A more common use for the `static` keyword is to define constants. For example, the `Math` class contains the following definitions:

```
   public class Math
   {
      ...
      public static final double E = 2.7182818284590452354;
      public static final double PI = 3.14159265358979323846;
   }
```

The keyword `final` denotes a constant value. After a `final` variable has been initialized, you cannot change its value.

These constants are public. You refer to them as `Math.PI` and `Math.E`.

A static method is a method that does not operate on objects. You have already encountered static methods such as `Math.sqrt`. Another use for static methods are *factory methods*, methods that return an object, similar to a constructor. Here is a factory method for the `Greeter` class that returns a greeter object with a random name:

```
   public class Greeter
   {
      public static Greeter getRandomInstance()
      {
         if (generator.nextBoolean())
            return new Greeter("World");
         else
            return new Greeter("Mars");
      }
      ...
   }
```

You invoke this method as `Greeter.getRandomInstance()`. Note that static methods can access static fields but not instance fields—they don't operate on an object.

Static fields and methods have their place, but they are quite rare in object-oriented programs. If your programs contain many static fields and methods, then this may mean that you have missed an opportunity to discover sufficient classes to implement your program in an object-oriented manner. Here is a bad example that shows how you can write very poor non-object-oriented programs with static fields and methods:

```
public class BadGreeter
{
   public static void main(String[] args)
   {
      name = "World";
      printHello();
   }

   public static void printHello() // bad style
   {
      System.out.println("Hello, " + name + "!");
   }

   private static String name; // bad style
}
```

## 1.13   Programming Style

Class names should always start with an uppercase letter and use mixed case, such as `String`, `StringTokenizer`, and so on. Names of packages should always be lowercase, such as `edu.sjsu.cs.cs151.alice`. Fields and methods should always start with a lowercase letter and use mixed case, such as `name` and `sayHello`. Underscores are not commonly used in class or method names. Constants should be in all uppercase with an occasional underscore, such as `PI` or `MAX_VALUE`. This is not a requirement of the Java language but a convention that is followed by essentially all Java programmers. Your programs would look very strange to other Java programmers if you used classes that started with a lowercase letter or methods that started with an uppercase letter. It is not considered good style by most Java programmers to use prefixes for fields (such as `_name` or `m_Name`)

It is very common to use `get` and `set` prefixes for methods that get or set a property of an object, such as

```
public String getName()
public void setName(String aName)
```

However, a Boolean property has prefixes `is` and `set`, such as

```
public boolean isPolite()
public void setPolite(boolean b)
```

There are two common brace styles: The "Allmann" style in which braces line up, and the compact but less clear "Kernighan and Ritchie" style. Here is the `Greeter` class, formatted in the "Kernighan and Ritchie" style.

```
public class Greeter {
   public Greeter(String aName) {
      name = aName;
   }

   public String sayHello() {
      return "Hello, " + name + "!";
   }
```

```
    private String name;
}
```

We use the "Allmann" style in this book.

Some programmers list fields before methods in a class:

```
public class Greeter
{
    private String name;
        // listing private features first is not a good idea

    public Greeter(String aName)
    {
        ...
    }
}
```

However, from an object-oriented programming point of view, it makes more sense to list the public interface first. That is the approach we use in this book.

Except for `public static final` fields, all fields should be declared `private`. If you omit the access specifier, the field has *package visibility*—all methods of classes in the same package can access it. That is an unsafe practice that you should avoid.

It is technically legal—as a sop to C++ programmers—to declare array variables as

```
    int numbers[]
```

You should avoid that style and use

```
    int[] numbers
```

That style clearly shows the type `int[]` of the variable.

All classes, methods, parameters, and return values should have documentation comments.

Some programmers like to format documentation comments with a column of asterisks, like this:

```
    /**
    *   Greet with a "Hello" message.
    *   @return a message containing "Hello" and the name of
    *   the greeted person or entity.
    */
```

It looks pretty, but it makes it harder to edit the comments. We don't recommend that practice.

You should put spaces *around* binary operators and after keywords, but not after method names or casts.

| Good | Bad |
|:---:|:---:|
| `x > y` | `x>y` |
| `if (x > y)` | `if(x > y)` |
| `Math.sqrt(x)` | `Math.sqrt (x)` |
| `(int)x` | `(int) x` |

You should not use *magic numbers*. Use named constants (`final` variables) instead. For example, don't use

```
h = 31 * h + val[off]; // Bad--what's 31?
```

What is 31? The number of days in January? The position of the highest bit in an integer? No, it's the hash multiplier.

Instead, declare a local constant in the method

```
final int HASH_MULTIPLIER = 31
```

or a static constant in the class (if it is used by more than one method)

```
private static final int HASH_MULTIPLIER = 31
```

and then use `HASH_MULTIPLIER` instead of 31.


## EXERCISES

**Exercise 1.1.** Add a `sayGoodbye` method to the `Greeter` class and add a call to test the method in the `GreeterTest` class (or test it in BlueJ).

**Exercise 1.2.** What happens when you run the Java interpreter on the `Greeter` class instead of the `GreeterTest` class? Try it out and explain.

**Exercise 1.3.** Add comments to the `GreeterTest` class and the `main` method. Document `args` as "unused". Use `javadoc` to generate a file `GreeterTest.html`. Inspect the file in your browser.

**Exercise 1.4.** Bookmark `docs/api/index.html` in your browser. Find the documentation of the `String` class. How many methods does the `String` class have?

**Exercise 1.5.** Write a program that prints "Hello, San José". Use a \u escape sequence to denote the letter é.

**Exercise 1.6.** What is the Unicode character of the Greek letter "pi" ($\pi$)? For the Chinese character "bu" (不)?

Exercise 1.7. Run the `javadoc` utility on the `Greeter` class. What output do you get? How does the output change when you remove some of the documentation comments?

Exercise 1.8. Download and install the `DocCheck` utility. What output do you get when you remove some of the documentation comments of the `Greeter` class?

Exercise 1.9. Write a program that computes and prints the square root of 1000, rounded to the nearest integer.

Exercise 1.10. Write a program that computes and prints the sum of integers from 1 to 100 and the sum of integers from 100 to 1000. Create an appropriate class `Summer` that has no `main` method for this purpose. If you don't use BlueJ, create a second class with a `main` method to instantiate two objects of the `Summer`class.

Exercise 1.11. Add a `setName` method to the `Greeter` class. Write a program with two `Greeter` variables that refer to the same `Greeter` object. Invoke `setName` on one of the references and `sayHello` on the other. Print the return value. Explain.

Exercise 1.12. Write a program that sets a `Greeter` variable to `null` and then calls `say-Hello` on that variable. Explain the resulting output. What does the number behind the file name mean?

Exercise 1.13. Write a test program that tests the three `set` methods of the example, printing out the parameter variables before and after the method call.

Exercise 1.14. Write a method `void swapNames(Greeter other)` of the `Greeter` class that swaps the names of this greeter and another.

Exercise 1.15. Write a program in which `Greeter` is in the package `edu.sjsu.cs.yourcourse.yourname` and `GreeterTest` is in the default package. Into which directories do you put the source files and the class files?

Exercise 1.16. What is wrong with the following code snippet?

```
ArrayList strings;
strings.add("France");
```

Exercise 1.17. Write a `GreeterTest` program that constructs `Greeter` objects for all command line arguments and prints out the results of calling `sayHello`. For example, if your program is invoked as

```
java GreeterTest Mars Venus
```

then the program should print

```
Hello, Mars!
Hello, Venus!
```

Exercise 1.18. What is the value of the following?

(a)  `2 + 2 + "2"`

(b)  `"" + countries`, where `countries` is an `ArrayList` filled with several strings

(c)  `"Hello" + new Greeter("World")`

Write a small sample program to find out, then explain the answers.

Exercise 1.19. Write a program that prints the sum of its command-line arguments (assuming they are numbers). For example,

```
java Adder 3 2.5 -4.1
```

should print The sum is 1.4

Exercise 1.20. Write a GreeterTest program that asks the user "What is your name?" and then prints out "Hello, *username*". Use an input dialog.

Exercise 1.21. Write a GreeterTest program that asks the user "What is your name?" and then prints out "Hello, *username*". Use a buffered reader.

Exercise 1.22. Write a class that can generate random strings with characters in a given set. For example,

```
RandomStringGenerator generator = new RandomStringGenerator();
generator.addRange('a', 'z');
generator.addRange('A', 'Z');
String s = generator.nextString(10);
    // a random string consisting of ten lowercase
    // or uppercase English characters
```

Your class should keep an ArrayList of Range objects.

Exercise 1.23. Write a program that plays TicTacToe with a human user. Use a class Tic-TacToeBoard that stores a $3 \times 3$ array of char values (filled with 'x', 'o', or space characters). Your program should use a random number generator to choose who begins. When it's the computer's turn, randomly generate a legal move. When it's the human's turn, read the move and check that it is legal.

Exercise 1.24. Write a program that reads input data from the user and prints the minimum, maximum, and average value of the input data. Use a class DataAnalyzer and a separate class DataAnalyzerTest. Use a sequence of JOptionPane input dialogs to read the input.

Exercise 1.25. Repeat the preceding exercise, but read the data from a file. To read the data, create a BufferedReader like this:

```
BufferedReader reader = new BufferedReader(new FileReader(filename));
```

Then use the BufferedReader.readLine and Double.parseDouble methods.

Exercise 1.26. Improve the performance of the getRandomInstance factory method by returning one of two fixed Greeter objects (stored in static fields) rather than constructing a new object with every call.

Exercise 1.27. Use any ZIP utility or the jar program from the Java SDK to uncompress the src.jar file that is part of the Java SDK. Then look at the source code of the String class in java/lang/String.java. How many style rules do the programmers violate? Look at the hashCode method. How can you rewrite it in a less muddleheaded way?