Chapter 5

JAVA

Brief Summary of Java

Java programs are compiled into an intermediate format, known as bytecode, and then run through an interpreter that executes in a Java Virtual Machine (JVM).

The basic syntax of Java is similar to C and C++. All white space is treated equally, indent level does not matter, statements end in a semicolon, and blocks of code are enclosed between `{` and `}`.

Comments are enclosed between `/*` and `*/`, or else begin with `//`, in which case the rest of the line is a comment.

Data Types and Variables

The integer data types are `byte`, `short`, `int`, and `long`, which correspond to numbers of 8, 16, 32, and 64 bits. The types `float` and `double` store floating-point numbers; `char` stores a 16-bit Unicode character, and `boolean` can hold one of two values, `true` or `false`.

Variables are declared with a type and name, as in the following:

```
int myint;
```

They can be initialized at the same time:

```
char delimeter = '/';
boolean finished = false;
```

Variables can be declared anywhere they are used. The scope of a variable usually extends to the end of the code block it was declared in.

Java allows variables to be converted between different numeric types by casting, as in the following:

```
int a;
double d = (double)a;
```

You can also cast between objects, but that is beyond the scope of this book.

Variables can be declared as `final`, which means that their value cannot be changed after it is initialized:

```
final int MAX_LEN = 128;
```

Arithmetic expressions in Java are straightforward, with `%` used for modulo:

```
k = a + b;
remainder = tot % users;
```

The `++` and `--` operators exist. If they are used in prefix notation, the expression is evaluated after the operation is done. In postfix notation, the expression is evaluated before the operation is complete. So, with the following code

```
d = 4;
e = ++d;
f = e--;
```

`e` and `f` are both set to `5`.

Strings (and Objects)

Beyond the basic data types, everything in Java is declared as a class. A *class* is a grouping of variables and methods (functions that operate

on those variables). The word *object* is often used to refer to a class, but technically, a class is a description of an object and an instance is an actual object.

You can define your own classes; Java includes many predefined ones. One such class is `String` (or more precisely, `java.lang.String`), which is used to store a constant string. Strings in Java are not just arrays of characters—they are a class that has defined methods for accessing and modifying the characters.

The `String` can serve as an example of how Java objects are used. A `String` can be created from an array of characters, as follows:

```
char[] myArray = { 'a', 'b', 'c' };
String myString = new String(myArray);
```

The expression new `String(myArray)` invokes what is called a *constructor* for the class `String`. *Constructors* create a new instance of an object, optionally taking parameters. How many parameters a constructor takes, and the type and order of those parameters, are part of the constructor's *signature*. Multiple constructors can exist for a given class as long as they have different signatures. For example, another constructor for `String` is called as follows:

```
String myString = new String(myArray, 2, 1);
```

That is, specifying an offset and count within `myArray`. You can also call

```
String myString = new String();
```

This creates an empty string. (A `String` cannot be changed after it's initialized, so it would stay empty.) The `String` class actually has nine constructors, plus two more obsolete ones.

When Java sees a literal string in double quotes, it automatically creates a `String` object, so you can write the following:

```
String newString = "text";
```

This is actually an assignment of one `String` to another. This automatic creation of an object from a literal is unique to the `String` class (all other literals, such as numbers, become primitive types), but it sure is convenient.

No destructors exist in Java; objects are destroyed by the *garbage collector* at some point after the last reference to them is removed (often because the variables holding that reference go out of scope). A variable can be

assigned a keyword `null` to force a reference it is holding to be removed:

```
anotherString = null;
```

However, the garbage collector makes no guarantees about how soon an object will be destroyed once there are no references to it.

Java does not have explicit pointers; in a sense, all variables that refer to objects are pointers. When you assign between two objects of the same type, you actually assign a reference to the object on the right-hand side. To create a new instance of an object, you need to call one of its constructors:

```
myObject a, b;
a = b;                  // reference
a = new myObject(b);    // create a new object
```

Classes define methods that can be called on an instance of that class. For example, the `String` class has a method `length()` that returns the length of the string:

```
String j = "abc123";
x = j.length();
```

As previously mentioned, a `String` cannot change after it's initialized. Java has another class, `StringBuffer`, which holds strings that can change. A `StringBuffer` can be constructed from a `String`, or from a length, which specifies how many characters of capacity it should start with:

```
StringBuffer sb1 = new StringBuffer("howdy");
StringBuffer sb2 = new StringBuffer(100);
```

`StringBuffer` has a variety of methods on it:

```
sb.append("more data");
char c = sb.charAt(12);
sb.reverse();
```

In Java, the `+` operator can concatenate strings together. A sequence such as the following

```
String greeting = "Hello";
greeting = greeting + " there";
```

is legal. Because the original `String` that `greeting` points to cannot be modified, the concatenation actually involves the creation of a new

`String`, which `greeting` is then set to point to. Therefore, the reference to the original `"Hello"` string is removed, which eventually causes it to be destroyed.

Note

The concatenation statement also involves some more behind-the-scenes magic by the compiler. It creates a temporary `StringBuffer`, ***then calls the*** `StringBuffer.append()` ***method for each expression separated by a*** `+` ***sign, then calls*** `StringBuffer.toString()` ***to convert it back to the result*** `String`. ***As with the automatic creation of*** `String` ***objects from constant strings, this is a special case on the part of Java, but is there because string concatenation is so useful.***

`StringBuffer.append()` is overloaded, so it can be passed any primitive type. Thus, you can call the following

```
int j = 4;
String b = "Value is" + j;
```

and `b` will equal `"Value is 4"`. In fact, `StringBuffer.append()` works for any object by appending the result of the object's `toString()` method, which can be overridden as needed by the author of the object's class.

Arrays

Arrays in Java are declared with square brackets:

```
int[] intArray;
```

The array then has to be created:

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

`intArray` would then be indexed from `0` to `9`.

Arrays can also be created at declaration time, if values are specified using an array initializer:

```
int[] array2 = { 5, 4, 3, 2, 1 };
```

You can't explicitly specify the length in that case because it's determined from how many values are provided.

You can get the number of elements in an array:

```
k = array2.length;
```

Note that this is not a method, so no parentheses appear after `length`.
Arrays can also hold objects, so you can declare the following:

```
MyObject[] objarray;
```

This would then be created as follows (this could be combined with the declaration):

```
objarray = new MyObject[5];
```

It is important to note that this creates only the array. You still need to create the five objects:

```
for (k = 0; k < 5; k++) {
    objarray[k] = new MyObject();
}
```

To create subarrays, create an array where each element is an array. The first array can be declared and created in one step

```
int[][] bigArray = new int[6][];
```

and then each subarray needs to be created (each one can be different lengths, in fact):

```
for (m = 0; m < 6; m++) {
    bigArray[m] = new int[20];
}
```

You can initialize arrays when they are declared:

```
short[][] shortArray = { { 1, 2, 3 },  { 4 },  { 5 , 6 } };
```

After that, `shortArray[0]` would be an array of three elements, `shortArray[1]` would be an array of one element, and `shortArray[2]` would be an array of two elements.
Finally, if the entries in the arrays are objects, they also have to be constructed, as shown here:

```
final int XDIM = 6;
final int YDIM = 10;
SomeObj[][] oa;
```

```
oa = new SomeObj[XDIM][];
for (int i = 0; i < XDIM; i++) {
    oa[i] = new SomeObj[YDIM];
    for (int j = 0; j < YDIM; j++) {
        oa[i][j] = new SomeObj();
    }
}
```

Conditionals

Java *conditionals* use the same `if/else` syntax as C:

```
if (j == 5) {
    // do something
} else {
    // do something else
}
```

The `switch` statement is also the same, with explicit `break` statements required, and a `default` case:

```
switch (newChar) {
    case "@":
        process_at();
        break;
    case ".":
        process_dot();
        break;
    default:
        ignore();
}
```

Loops

Looping is done with `for`, `while`, and `do/while`:

```
while (k > 8) {
    do_processing();
}

do {
    eof = get_line();
} while (eof != true);
```

`break` breaks out of a loop, and `continue` jumps to the next iteration. A label can be added to `break` or `continue` to specify which loop it refers to:

```
outerloop:
for (x = 0; x < 20; x++) {
    for (y = x; y < 20; y++) {
        if (something) {
            break outerloop;
        }
    }
}
```

`outerloop:` is a label for the loop and the statement `break outerloop;` breaks out of the labeled loop. It does *not* jump to the point where the `outerloop:` label exists in the code.

Classes

A *class* is defined as follows:

```
class MyClass {
    private int a;
    public StringBuffer b;
    public MyClass(int j) {
        a = j;
        b = new StringBuffer(j);
    }
    public MyClass(String s) {
        a = s.length();
        b = new StringBuffer(s);
    }
    public int getLength() {
        return a;
    }
}
```

`a` and `b` are member variables in the class. `a` is defined with an *access specifier* of `private`, which means that it is hidden from the view of external code. `b` is `public`, which means that anyone can access it if he has an instance of `MyClass`. For example

```
MyClass mc = new MyClass("hello");
String abc = mc.b;  // this is allowed, b is public
int def = mc.a;     // this is NOT allowed, a is private
```

We'll get back to access specifiers within the next few paragraphs. For now, note that `MyClass` has two constructors, one of which takes an `int` as a parameter, and the other takes a `String` (the second one is the one called in the previous code sample). Both constructors initialize `a` and `b`. Variables can also be initialized when they are declared, so `b` could have been declared as follows:

```
public StringBuffer b = new StringBuffer();
```

Although, for this class, that would not be necessary because every constructor initializes `b`.

Classes can also inherit from another class. A subclass inherits all the state and behavior of its superclass (but *not* the constructors), although it can override methods by providing new ones with the same name (unless those methods were declared with the `final` keyword).

Inheritance is indicated by the `extends` keyword:

```
abstract class Polygon {
    Point[] points;
    abstract int getcount();
}

class Triangle extends Polygon {
    public Triangle() {
        points = new Point[3];
    }
    int getcount() { return 3 };
}
```

The access specifier of a class variable can be `public`, `private`, `protected`, or `package` (the default). `public` means that any code can access it; `private` means that only methods in the class itself can access it; `package` means that any code in the same "package" (which is a way to group classes) can access it.

A variable marked `protected` can be accessed by the class, subclasses, and all classes in the same package. Actually, to be more precise, subclasses can only access a `protected` member inherited from a superclass when the object is an instance of the subclass (which it usually will be). They can't modify an instance of the superclass itself. (If you didn't catch all that, don't worry too much about it.)

Members of a class (variables or methods) can be declared with the keyword `static`, which makes them "class members," as opposed to

"instance members," which is the case that's been described so far. Class variables and class methods exist just once, as opposed to once per instance. For example, a class could assign unique identifiers to each instance it creates, as shown here:

```
class ImportantObject {
    private static int nextcounter = 0;
    private int counter;
    public ImportantObject() {
        counter = nextcounter++;
    }
    // continues...
}
```

Each instance of the class has its own `counter` member, but there is only one global `nextcounter`.

A method on a class can be declared `abstract`, which means that it defines the parameters and return value, but has no actual implementation. A class can also be declared `abstract`; this is required if it defines at least one abstract method. (It is also required if a class does not provide implementation for any abstract methods declared in its superclasses.) An abstract class cannot itself be instantiated—it exists to ensure that subclasses follow the "contract" that it defines.

Closely related to classes are interfaces. The main difference between an interface and an abstract class is that *all* the methods on an interface must be abstract:

```
public interface identify {
    String getName();
}
```

Other classes can now support an interface using the `implements` keyword. Unlike inheritance, where a class can only inherit from one class, classes can implement as many interfaces as they like, as long as they provide implementations of all the interfaces' methods (or are declared `abstract`):

```
class SomeClass implements identify {
    final String name = "SomeClass";
    String getName() { return name };
    // rest of class follows...
}
```

A class with only public member variables—and no methods—can be used to group variables by name, similar to C structures:

```
class Record {
    public String name;
    public int id;
    public int privilege;
}

Record r = new Record();
r.name = "Joe";
r.id = 12;
r.privilege = 3;
```

Java likely has a class for almost any standard operation you want to do; the documentation lists constructors and methods. For example, classes exist that wrap all the primitive types, such as this one that wraps the short primitive in a class called `Short` (note the capital "S" on the class name), and provides various useful methods:

```
Short s = new Short(12);
String str = s.toString();
```

I won't go into more details about specific classes, except as needed in the examples.

Exceptions

Java supports exceptions, which are objects that can be caught:

```
try {
    file = new FileInputStream("data.tmp");
} catch (FileNotFoundException e) {
    System.err.println("Exception " + e.getMessage());
} finally {
    // cleanup code
}
```

A `try` can have multiple `catch` blocks, each catching a different exception. (There is a hierarchy of exception classes, leading back to a class called `Throwable`. A `catch` block that catches a particular exception also catches any exceptions that are subclasses of that exception.)

If an exception happens and is caught, the `catch` block executes. The `finally` block always executes, whether or not an exception happens, and is usually used for cleanup code.

You can create and throw exceptions:

```
if (bytesleft == 0) {
    throw new EOFException();
}
```

Java requires that methods that can throw an exception specify it in the declaration of the method, using the `throws` keyword:

```
public void read_file(File file)
            throws IOException {
    if (!check_valid(file)) {
        throw new IOException("check_valid() failed");
    }
}
```

Method declarations must also list any exceptions that can be thrown by methods they call, unless they catch the exception. Thus, a method that called `read_file()` (as defined above) would need to either put it in a `try` block with an associated `catch` block that caught `IOException`, or specify in its own declaration that it throws `IOException`. (This "catch or specify" rule does not apply to a class of exceptions known as runtime exceptions, which inherit from the class `RuntimeException`. This rule is detailed in the Java documentation.)

Importing Other Code

To use a class, you must import the package that defines it. This is specified in the documentation of the class. For example, to use the `Timer` class, include the following in the code:

```
import java.util.Timer;
```

This can include a wildcard:

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

Command-Line Applications and Applets

The examples used in this chapter are split between command-line applications and applets designed to run in a web browser. A command-line application has to contain a class that implements a `main()` method, which must be defined as `public static`, `return type void`, and receive the command-line parameters as an array of `String objects` called `args` (the first element in `args` is the first parameter, etc.):

```
public class MyApplication {
    public static void main(String[] args) {
        for (int j = 0; j < args.length; j++) {
            System.out.println(args[j]);
        }
    }
}
```

An applet inherits from a class called `Applet`:

```
public class MyApplet extends Applet {
    public void paint(Graphics g) {
       g.drawString("Testing 123", 10, 10);
    }
}
```

The `paint()` method is overridden from a superclass a few levels up from `Applet`, and is used to display on the screen. The `Graphics` class has many methods used to draw lines and shapes, display text, change color, and so on.

❶ Is a Year a Leap Year?

This program determines if the first argument passed to it is a leap year, and prints the result.

A year is a leap year if it is divisible by 4, unless it is divisible by 100. However, years divisible by 400 *are* leap years.

The program internally uses a method that throws one of two exceptions: one if the year is a leap year, and one if it isn't. Because these exception classes don't do anything different from the built-in `Exception` class, they don't need to override any methods; they can simply be declared and used.

To convert the command-line parameter from a string to a number, the program uses the `static` method `parseLong()` from the class `Long`, which is a class that wraps the primitive type `long`. Because `parseLong()` is a `static` method, it is not called on an instance of the class.

`parseLong()` is defined to throw `NumberFormatException` if the input string cannot be converted to a number. Because `checkLeapYear()` throws the two user-defined exceptions and, thus, typically is called inside a `try` block, it is not too much work to also catch `NumberFormatException`.

`NumberFormatException` is a runtime exception and, therefore, does not have to be listed in the `throws` clause of the declaration of `checkLeapYear()`, but it is included because throwing that exception is the designated way to handle an invalid input.

Source Code

```
1.      public class IsLeapYear {
2.
3.          public static class LeapYearException
4.                  extends Exception {}
5.          public static class NotLeapYearException
6.                  extends Exception {}
7.
8.          static void checkLeapYear(String year)
9.              throws LeapYearException, NotLeapYearException,
10.                    NumberFormatException {
11.
12.             long yearAsLong = Long.parseLong(year);
13.
14.             //
15.             // A leap year is a multiple of 4, unless it is
16.             // a multiple of 100, unless it is a multiple of
17.             // 400.
18.             //
19.             // We calculate the three values, then make a
20.             // 3-bit binary value out of them and look it up
21.             // in results.
22.             //
23.
24.             final boolean results[] =
25.                     { true, false, false, true,
26.                       false, false, false, false };
```

```
27.
28.             if (results[
29.                 ((((yearAsLong % 4) == 0) ? 1 : 0) << 2) +
30.                 ((((yearAsLong % 100) == 0) ? 1 : 0) << 1) +
31.                 ((((yearAsLong % 400) == 0) ? 1 : 0) << 0)]) {
32.                 throw new LeapYearException();
33.             } else {
34.                 throw new NotLeapYearException();
35.             }
36.         }
37.
38.         public static void main(String[] args) {
39.
40.             if (args.length > 0) {
41.
42.                 try {
43.                     checkLeapYear(args[0]);
44.                 } catch ( NumberFormatException nfe ) {
45.                     System.out.println(
46.                         "Invalid argument: " +
47.                         nfe.getMessage());
48.                 } catch ( LeapYearException lye ) {
49.                     System.out.println(
50.                         args[0] + " is a leap year");
51.                 } catch ( NotLeapYearException nlye ) {
52.                     System.out.println(
53.                         args[0] + " is not a leap year");
54.                 }
55.             }
56.         }
57.     }
```

Suggestions

1. What exactly does it mean if a bit is on in `results`?
2. Because the value computed on lines 29–31 is immediately used to index into `results`, an array of size 8, is it guaranteed that this value will be properly restricted so as to not produce an invalid array index?
3. How many possible kinds of years are there, and given that it is less than the size of `results`, are there certain values for the index into `results` that will never occur?

Hints

Walk through `main()` with the following values for `args[0]`:

1. Multiple of 100 is not a leap year: `"1900"`
2. Multiple of 4 is a leap year: `"1904"`
3. Multiple of 400 is a leap year: `"2000"`
4. Any other year is not a leap year: `"2001"`

Explanation of the Bug

Not surprisingly, the problem is in the declaration of the `results` array on lines 24–26. It is reversed; that is, it is declared as if the calculation of the index on lines 29–31 had every bit flipped, which is an **F.init** error. The proper declaration should be as follows:

```
private static final boolean results[] =
    { false, false, false, false,
      true, false, false, true };
```

Alternately, the assignment of the bits on lines 29–31 could be flipped.

With the bits assigned as-is, if a year is divisible by 400, the low bit in the index will be on, which means that the index in binary is in the form xx1. Because such years are also divisible by 4 and 100, the next two bits are also on. Thus, indices 1, 3, and 5 in `results` won't ever be used. Years divisible by 400 always wind up with an index of 7 (binary 111).

If year is not divisible by 400, but is divisible by 100, the low bit is off, but the second bit is on, so the index is in the form x10. Because such a year is also divisible by 4, the index is always 6 (binary 110), and index 2 is never used.

For the rest of the years, those not divisible by 100 (or 400), the index is in the form x00. Years divisible by 4 result in an index of 4 (binary 100), and those not divisible by 4 result in an index of 0.

Therefore, the only indices that matter are 0, 4, 6, and 7. Of those, 0 and 6 should be false (not leap years), and 4 and 7 should be true (leap years). Because of the bug, those indices had their bits flipped, so 7 and 1 were false, and 3 and 0 were true. The "unused" indices (2, 4, 5, and 6) were also set to false.

Thus, the program as written would, by chance, work if a year was divisible by 100 but not by 400 (a year such as 1900), correctly reporting,

based on the value of index 6, that such a year was not a leap year. It would misrepresent the years 2000 (index 7) and 2004 (index 4) as not being leap years, and the year 2001 (index 0) as being a leap year.

❷ Convert a Number to Text

This class takes a number and converts it to the equivalent text in English.

For example, the input 1 should return the string "one" and 123,456 should return the string "one hundred twenty three thousand four hundred fifty six." (The program does not try to insert the word "and" between any of the numbers.)

The class has one constructor, which takes the integer to convert as a parameter, and has a single method—`getString()`—that returns the string. This is not necessarily the ideal interface for such a class, but it works for these purposes.

The constructor uses the `pow()` method from the `Math package`, which raises a number to a power. It also uses the `substring()` method of the `String` class, which when called with one parameter, creates a new `String` starting at the specified offset in the original string. (The offset is zero-based.)

Source Code

```
1.      class EnglishNumber {
2.
3.          private static final String[] ones = {
4.              " one", " two", " three", " four", " five",
5.              " six", " seven", " eight", " nine", " ten",
6.              " eleven", " twelve", " thirteen", " fourteen",
7.              " fifteen", " sixteen", " seventeen",
8.              " eighteen", " nineteen"
9.          };
10.         private static final String[] tens = {
11.             " twenty", " thirty", " forty", " fifty",
12.             " sixty", " seventy", " eighty", " ninety"
13.         };
14.         //
15.         // A Java long can only go up to 2^63 - 1,
16.         // so quintillions is as big as it gets. The
```

```
17.         // program would automatically handle larger
18.         // numbers if this array were extended.
19.         //
20.         private static final String[] groups = {
21.             "",
22.             " thousand",
23.             " million",
24.             " billion",
25.             " trillion",
26.             " quadrillion",
27.             " quintillion"
28.         };
29.
30.         private String string = new String();
31.
32.         public String getString() { return string; }
33.
34.         public EnglishNumber ( long n ) {
35.
36.             // Go through the number one group at a time.
37.
38.             for (int i = groups.length-1; i >= 0; i--) {
39.
40.                 // Is the number as big as this group?
41.
42.                 long cutoff =
43.                     (long)Math.pow((double)10,
44.                                    (double)(i*3));
45.
46.                 if ( n >= cutoff ) {
47.
48.                     int thisPart = (int)(n / cutoff);
49.
50.                     // Use the ones[] array for both the
51.                     // hundreds and the ones digit. Note
52.                     // that tens[] starts at "twenty".
53.
54.                     if (thisPart >= 100) {
55.                         string +=
56.                           ones[thisPart/100] +
57.                             " hundred";
58.                         thisPart = thisPart % 100;
59.                     }
60.                     if (thisPart >= 20) {
61.                         string += tens[(thisPart/10)-1];
62.                         thisPart = thisPart % 10;
63.                     }
```

```
64.                         if (thisPart >= 1) {
65.                             string += ones[thisPart];
66.                         }
67.
68.                         string += groups[i];
69.
70.                         n = n % cutoff;
71.
72.                     }
73.                 }
74.
75.                 if (string.length() == 0) {
76.                     string = "zero";
77.                 } else {
78.                     // remove initial space
79.                     string = string.substring(1);
80.                 }
81.             }
82.         }
```

Suggestions

1. Look at the main `for` loop, running from lines 38–73. What is the goal of one iteration of this loop?
2. What is the meaning of the variable `thisPart`?
3. The functionality is split because the return string is computed in the constructor, but not returned until `getString()` is called. What variable is returned? Where is it modified?
4. What is the trivial input for this program? How is it handled in the code?

Hints

Walk through the constructor with the following inputs to the constructor, and determine what value `getString()` would return:

1. The trivial case: `n == 0`.
2. Test one iteration of the loop, including one case where a digit is 0: `n == 102`.
3. Test several iterations of the loop: `n == 1234567`.

Explanation of the Bug

The code indexes into the `ones` and `tens` array incorrectly. Because arrays are zero-based, the number 1 corresponds to `ones[0]`, not `ones[1]`. Thus, the various accesses need to be adjusted for this. Lines 55–57 should change from

```
string +=
  ones[thisPart/100] +
    " hundred";
```

to

```
string +=
  ones[(thisPart/100)-1] +
    " hundred";
```

Line 61 should change from

```
string += tens[(thisPart/10)-1];
```

to

```
string += tens[(thisPart/10)-2];
```

Line 65 should change from

```
string += ones[thisPart];
```

to

```
string += ones[thisPart-1];
```

This is an **A.off-by-one** error that becomes a **D.index** error. It can actually lead to an `ArrayIndexOutOfBoundsException` being thrown in certain cases. (Can you determine which ones?)

❸ Draw a Triangle on the Screen, Part I

This function draws a triangle on the screen. It becomes the core of an applet that allows the user to pick the three endpoints by clicking three times on the screen, which will be completed in the next example.

The algorithm assumes that the three points are ordered by x coordinate. It fills the triangle by drawing a series of vertical lines, 1 pixel wide. To do this, it splits the triangle into a "left" and "right" half; that is, the part from the x coordinate of the first point to the x coordinate of the second point, and the part from the x coordinate of the second point to the x coordinate of the third point. This algorithm won't work well with triangles that are extremely tall and thin, so to cover those cases, the function also draws a line between each pair of endpoints.

Applets draw to the screen by overriding a member method called `paint()`. This method is passed a `Graphics` class, which supports two methods that are used here: `fillOval()` (used to draw a circle) and `drawLine()`. The meaning of the parameters can be inferred from their use (assume that they are passed in the correct order).

In the declaration of the `Triangle` class, it specifies that it implements the `MouseListener` interface. This is explained in the next program.

Source Code

```
1.  import java.awt.event.*;
2.  import java.awt.*;
3.
4.  public class Triangle extends java.applet.Applet
5.      implements MouseListener {
6.
7.      // The rest of the applet will be in the next
8.      // example.
9.
10.     Point[] pt = new Point[3];
11.     int ptCount = 0;
12.
13.     public void paint(Graphics g) {
14.
15.         int i;
16.
17.         // Draw the points that have been selected
18.
19.         for (i = 0; i < ptCount; i++) {
20.             g.fillOval(pt[i].x - 10, pt[i].y - 10,
21.                        20, 20);
22.         }
23.
```

```
24.             if (ptCount == 3) {
25.
26.                 // Connect the endpoints to handle
27.                 // tall thin triangles.
28.
29.                 g.drawLine(pt[0].x, pt[0].y,
30.                            pt[1].x, pt[1].y);
31.                 g.drawLine(pt[1].x, pt[1].y,
32.                            pt[2].x, pt[2].y);
33.                 g.drawLine(pt[0].x, pt[0].y,
34.                            pt[2].x, pt[2].y);
35.
36.                 // Calculate x and y diffs between points.
37.
38.                 int x0to1 = pt[1].x - pt[0].x;
39.                 int x0to2 = pt[2].x - pt[0].x;
40.                 int x1to2 = pt[2].x - pt[1].x;
41.                 int y0to1 = pt[1].y - pt[0].y;
42.                 int y0to2 = pt[2].y - pt[0].y;
43.                 int y1to2 = pt[2].y - pt[1].y;
44.
45.                 // Left part of the triangle.
46.
47.                 if (x0to1 > 0) {
48.                     for (i = pt[0].x; i <= pt[1].x; i++) {
49.                        g.drawLine(
50.                          i,
51.                          pt[0].y +
52.                            ((y0to1 * (i - pt[0].x)) / x0to1),
53.                          i,
54.                          pt[0].y +
55.                            ((y0to2 * (i - pt[0].x)) / x0to2)
56.                        );
57.                     }
58.                 }
59.
60.                 // Right part of the triangle.
61.
62.                 for (i = pt[1].x+1; i <= pt[2].x; i++) {
63.                     g.drawLine(
64.                         i,
65.                         pt[1].y +
66.                           ((y1to2 * (i - pt[1].x)) / x1to2),
67.                         i,
68.                         pt[1].y +
69.                           ((y0to2 * (i - pt[0].x)) / x0to2)
```

```
70.                             );
71.                         }
72.                     }
73.                 }
74.         }
```

Suggestions

1. There are several places with repetitive statements, such as lines 29–34 and 38–43. Check these lines carefully to ensure that they are correct.
2. Although the points are ordered by x coordinate, it's possible that two or three of them will have the same x coordinate. As a result, `x0to1`, `x0to2`, or `x1to2` could be `0`. Examine the code to ensure that the division operations on lines 52, 55, 66, and 69 would never result in an `ArithmeticException` due to divide by zero.
3. Look at the loops on lines 48–57 and lines 62–71. Determine what values will be passed to `g.drawLine()` on the first and last iteration of each of these loops to make sure that they seem reasonable. Remember how the values are related; for example, the expression `pt[0].y + y0to1` is equal to `pt[1].y`.

Hints

Walk through the function with `ptCount == 3` and the points as follows:

1. A triangle with nothing unusual:
```
pt[0].x = 0;
pt[0].y = 20;
pt[1].x = 2;
pt[1].y = 18;
pt[2].x = 4;
pt[2].y = 28;
```
2. A triangle with points that are the same in the x or y coordinate:
```
pt[0].x = 0;
pt[0].y = 10;
pt[1].x = 4;
pt[1].y = 10;
pt[2].x = 4;
pt[2].y = 0;
```

Explanation of the Bug

A **B.variable** error exists in the calculation of the second y coordinate in the call to `g.drawLine()` in the second loop. Lines 68–69, which read as follows

```
pt[1].y +
    ((y0to2 * (i - pt[0].x)) / x0to2)
```

should be

```
pt[0].y +
    ((y0to2 * (i - pt[0].x)) / x0to2)
```

The problem can be spotted by considering the last iteration of the loop, when `i` is equal to `pt[2].x`. In this situation, the expression as initially written becomes

```
pt[1].y +
    ((y0to2 * (pt[2].x - pt[0].x)) / x0to2)
```

which, because `pt[2].x - pt[0].x` is equal to `x0to2`, becomes

```
pt[1].y + y0to2
```

This does not make any particular sense because `pt[1].y` and `y0to2` are not related. With the fix, the expression is instead

```
pt[0].y + y0to2
```

This equals `pt[2].y`, a reasonable y coordinate for the second endpoint of the last vertical line (in fact, the y coordinate of the first endpoint of the line also evaluates to `pt[2].y`, so the "line" is actually just a single pixel drawn at the point `pt[2]`).

❹ Draw a Triangle on the Screen, Part II

This function draws a triangle on the screen, with the points selected by the user clicking three times on the screen. It uses the `paint()` method from the previous example. Because the `paint()` routine expects the points to be sorted by x coordinate, this function takes care of that.

Calling the `repaint()` method of the `Applet` class (which is actually a method of the `Component` class, the great-grandparent of `Applet`) eventually causes the `paint()` method to be called.

In addition to extending the `java.applet.Applet` class, as all applets do, the function also implements the `MouseListener` interface to receive mouse clicks. The only method in this interface that matters here is `mousePressed()`. This calls the method `getPoint()` (whose functionality is obvious) on the `MouseEvent` passed as a parameter. If an applet consumes a `MouseEvent` (or any event derived from its superclass `InputEvent`), it notes this by calling the `consume()` method.

The class calls `addMouseListener()` during initialization of the applet, and `removeMouseListener()` during destruction. This is how it registers to receive mouse events. These methods take, as a parameter, an object that implements `MouseListener`. Because the `Triangle` class extends `MouseListener`, it can pass `this`, that is, a pointer to the instance of the applet class itself, as a parameter.

The class should implement several methods that are not shown to save space. Applets normally implement a method `getAppletInfo()`, which returns the title and author of the applet. In addition, the `MouseListener` interface has four other methods: `mouseReleased()`, `mouseClicked()`, `mouseEntered()`, and `mouseExited()`. All these methods take a `MouseEvent` as a parameter, but don't need to do anything in this example.

Source Code

```
1.  import java.awt.event.*;
2.  import java.awt.*;
3.
4.  public class Triangle extends java.applet.Applet
5.      implements MouseListener {
6.
7.      Point[] pt = new Point[3];
8.      int ptCount = 0;
9.
10.     public void init() {
11.         addMouseListener(this);
12.     }
13.
14.     public void paint(Graphics g) {
15.         // See previous example for implementation
```

```
16.         }
17.
18.         public void mousePressed(MouseEvent e) {
19.
20.             if (ptCount < 3) {
21.                 pt[ptCount] = new Point(e.getPoint());
22.                 if ((ptCount++) == 3) {
23.                     Point p;
24.
25.                     // Order the points by x value, so
26.                     // pt[0] has the lowest x and pt[2]
27.                     // has the highest.
28.
29.                     if ((pt[1].x < pt[2].x) &&
30.                         (pt[1].x < pt[0].x)) {
31.                         p = pt[0]; pt[0] = pt[1]; pt[1] = p;
32.                     } else if ((pt[2].x < pt[1].x) &&
33.                                (pt[2].x < pt[0].x)) {
34.                         p = pt[0]; pt[0] = pt[2]; pt[2] = p;
35.                     }
36.                     if (pt[1].x > pt[2].x) {
37.                         p = pt[1]; pt[1] = pt[2]; pt[2] = p;
38.                     }
39.                 }
40.             }
41.             e.consume();
42.             repaint();
43.         }
44.
45.         public void destroy() {
46.             removeMouseListener(this);
47.         }
```

Suggestions

1. Look at the code on lines 29–38. The comment on lines 25–27 state that the goal is to order the points. Is it correct? How would you describe the goal after line 35?
2. `mousePressed()` calls `repaint()` even if this is not the third point selected. Is it correct to assume that `paint()` is ready to be called in this situation?
3. Examine the code to swap points on lines 31, 34, and 37. Is it done correctly? How many different inputs would be needed to ensure that all these code lines were covered?

Hints

Walk through the `mousePressed()` method, passing in the third point equal to (20, 50) and with the following values for member variables:

```
ptCount == 2
pt[0].x == 0
pt[0].y == 100
pt[1].x == 10
pt[1].y == 75
```

Explanation of the Bug

The bug is on line 22, which reads as follows:

```
if ((ptCount++) == 3) {
```

When using the postfix notation for `++`, the expression is evaluated before the addition is done. Therefore, this expression is true only when `ptCount` is already 3 before it is incremented. However, the `if()` on line 20 will prevent that entire block of code on lines 21–39 from executing if `ptCount` is 3 or greater. Therefore, the entire block of code from lines 23–38 will never execute, and the variables won't ever be sorted. This leads to `paint()` being called with unordered points (unless the user happens to click them in sorted x order) which causes the algorithm to malfunction.

The code should instead read as follows:

```
if ((++ptCount) == 3) {
```

Because the increment is done at the incorrect time, you could consider this an **F.location** error, or you could describe it as **B.expression**.

❺ Reverse a Linked List

This function reverses a singly linked list by walking the list and changing pointers.

Each element in the list is an instance of a class `ListNode`. The list itself is an instance of a class `List`. `ListNode` has a `next` member that

points to the next element on the list. The final element on the list has a `next` pointer equal to `null`.

List has a method called `Reverse()`, which is the method to reverse the linked list.

Source Code

```
1.      class ListNode {
2.
3.          private int value;
4.          protected ListNode next;
5.
6.          public ListNode(int v) {
7.              value = v;
8.              next = null;
9.          }
10.
11.         public ListNode(int v, ListNode n) {
12.             value = v;
13.             next = n;
14.         }
15.
16.         public int getValue() { return value; }
17.
18.     }
19.
20.     class List {
21.
22.         private ListNode head;
23.
24.         public List() {
25.             head = null;
26.         }
27.
28.         public List(ListNode ln) {
29.             head = ln;
30.         }
31.
32.         public void Reverse() {
33.
34.             // Walk the list, reversing the direction of
35.             // the next pointers.
36.
37.             ListNode ln1, ln2, ln3, ln4;
```

```
38.
39.             if (head == null)
40.                 return;
41.
42.             ln1 = head;
43.             ln2 = head.next;
44.             ln3 = null;
45.
46.             while (ln2 != null) {
47.                 ln4 = ln2.next;
48.                 ln1.next = ln3;
49.                 ln3 = ln1;
50.                 ln1 = ln2;
51.                 ln2 = ln4;
52.             }
53.
54.             //
55.             // When we get to the end of the list, the last
56.             // element we looked at is the new head.
57.             //
58.
59.             head = ln1;
60.         }
61.     }
```

Suggestions

1. What are the empty and trivial cases for the `Reverse()` method? How will the code handle them?
2. What is the purpose of the variable `ln4`?
3. Describe the meaning of `ln1`, `ln2`, and `ln3` after the `while` loop ends. Is the comment on lines 54–57 correct?

Hints

Walk through `Reverse()` in the following cases:

1. The list has only one element, so `head.next == null`.
2. The list has three elements, so `head` points to `Node1`, `Node1.next` points to `Node2`, `Node2.next` points to `Node3`, and `Node3.next` is `null`.

Explanation of the Bug

The code on line 59 is correct. The value of `ln1` after the last iteration of the `while()` loop is the new head of the list. However, the `next` pointer of that element is still `null` because it used to be the end of the list.

In the somewhat-confusing nomenclature of this function, `ln3` is the element that used to be before `ln1` in the list, so to finish off the reversal, there needs to be a line added after line 59 that reads as follows:

```
ln1.next = ln3;
```

This is a **D.limit** error because the code works correctly except when handling the last element of the old list. The effect of the bug is that the last element of the old list keeps its `next` pointer as `null`. Since this element becomes the first element of the new list, this truncates the list to a single element.

❻ Check if a List Has a Loop

This function checks if a singly linked list has a loop in it.

It uses the same `ListNode` and `List` classes from the previous examples, but implements a new member method, `HasLoop()`. A list has a loop if there is some `ListNode` node in it for which `node.next` is equal to `head`.

Source Code

```
1.      class List {
2.
3.          private ListNode head;
4.
5.          public List() {
6.              head = null;
7.          }
8.
9.          public List(ListNode ln) {
10.             head = ln;
11.         }
```

```
12.
13.         public boolean HasLoop() {
14.
15.             //
16.             // The algorithm is to start two pointers
17.             // at the head of the list; as the first pointer
18.             // advances one element in the list, the second
19.             // advances by two elements. If the second
20.             // pointer hits a null next pointer, then the
21.             // list does not have a loop; if the second
22.             // pointer hits the first pointer, then the list
23.             // has a loop.
24.             //
25.
26.             ListNode ln1, ln2;
27.
28.             if ((head == null) || (head.next == null))
29.                 return false;
30.
31.             ln1 = head;
32.             ln2 = head.next;
33.
34.             while (true) {
35.
36.                 if (ln1 == ln2)
37.                     return true;
38.
39.                 if (ln1.next == null)
40.                     return false;
41.                 else
42.                     ln1 = ln1.next;
43.
44.                 if (ln1 == ln2)
45.                     return true;
46.
47.                 if (ln2.next == null)
48.                     return false;
49.                 else
50.                     ln2 = ln2.next;
51.
52.                 if (ln1 == ln2)
53.                     return true;
54.
55.                 if (ln2.next == null)
56.                     return false;
```

```
57.                     else
58.                         ln2 = ln2.next;
59.
60.                 }
61.             }
```

Suggestions

1. What are the empty and trivial cases for this function?
2. Because the main loop in the code is `while(true)`, why is the function guaranteed to eventually exit?
3. How many different inputs are necessary to guarantee complete code coverage?

Hints

Walk through `HasLoop()` in the following cases:

1. The list has only one element, so `head.next == null`.
2. The list has three elements, so `head` points to `Node1`, `Node1.next` points to `Node2`, `Node2.next` points to `Node3`, and `Node3.next` is `null`.
3. The list has a loop, where `head` points to `Node1`, `Node1.next` points to `Node2`, and `Node2.next` points to `head`.

Explanation of the Bug

The code returns `true`, which indicates that it has found a loop, on any list with more than one element. The reason is that the check on lines 44–45, immediately after advancing `ln1`

```
if (ln1 == ln2)
    return true;
```

is `true` when `ln1` is advanced from the first to the second element in the list. This is because `ln2` is initialized before the loop to point to the second element, and it has not moved yet.

In fact, the check is unnecessary because the code is concerned with `ln2` looping around and catching up to `ln1`, so there is no need to check

for equality after `ln1` advances. This is an **F.location** error because the two lines should not exist at all.

❼ Quicksort

This function implements the quicksort algorithm.

Quicksort works by choosing an arbitrary element in the array and then dividing the array into two parts: The first part contains all elements less than or equal to the chosen element, and the second part contains all elements greater than the chosen element. The chosen element is then swapped into the spot between the two parts (known as the pivot point), which is its proper spot in the ultimately sorted array. The function is then called recursively twice—once on each part—to complete the sort.

Assume that the stack is deep enough that recursion will not cause a stack overflow when properly processing any array that is passed to the function.

The function declares an interface `quickcompare`, which has a single method `compare()`. This method is passed two instances of the class `Object` (which, in Java, is the root of the class hierarchy, and thus a superclass of any object), and returns a negative, zero, or positive number if the first parameter is less than, equal to, or greater than the second parameter, respectively. This is how the equivalent of function pointers can be supported in Java. To use the `Quicksort` class, you first declare a class that implements the `quickcompare` interface in an appropriate way for the data you want to sort, such as this one for `String` objects

```
private static class StringComp implements quickcompare {
    public int compare(Object a, Object b) {
        return ((String)a).compareTo((String)b);
    }
}
```

and then pass an instance of that class to `quicksort()`

```
  public static void main(String[] args) {
      quicksort(
          args, 0, args.length-1, new StringComp());
  }
```

(In this example, `StringComp` is declared as `static` so it can be called from `main()`, which is also `static`.)

Note that to make recursion easier, `quicksort()` defines the `end` parameter inclusively, thus the need to pass `args.length-1` in the previous call.

Source Code

```
1.      public class QuickSort {
2.
3.          public interface quickcompare {
4.              public int compare(Object a, Object b);
5.          }
6.
7.          // Declare it static since it does not operate
8.          // on class member variables (there aren't any).
9.
10.         public static void quicksort(
11.                 Object[] array,
12.                 int start,
13.                 int end,
14.                 quickcompare qc) {
15.
16.             if (start < end) {
17.
18.                 Object temp;
19.                 int pivot, low, high;
20.
21.                 //
22.                 // Partition the array.
23.                 //
24.
25.                 pivot = start;
26.                 low = start+1;
27.                 high = end;
28.                 while (true) {
29.                     while ((low < high) &&
30.                            (qc.compare(array[low],
31.                                        array[pivot]) <= 0)) {
32.                         ++low;
33.                     }
34.                     while ((high >= low) &&
35.                            (qc.compare(array[high],
36.                                        array[pivot]) > 0)) {
37.                         --high;
```

```
38.                         }
39.                         if (low < high) {
40.                             temp = array[low];
41.                             array[low] = array[high];
42.                             array[high] = temp;
43.                         } else {
44.                             break;
45.                         }
46.                     }
47.                     temp = array[pivot];
48.                     array[pivot] = array[high];
49.                     array[high] = temp;
50.
51.                     // Now sort before and after the pivot.
52.
53.                     quicksort(array, start, high, qc);
54.                     quicksort(array, high+1, end, qc);
55.                 }
56.             }
57.         }
```

Suggestions

1. What can you say about the relationship between `low` and `high` during the main `while()` loop? Can `low` ever be greater than `high`?
2. What is the goal at line 33? What is the goal at line 38?
3. At the end of the loop, how are `low` and `high` related? What types of inputs would cause different situations at the end of the loop?
4. Think of the empty, trivial, and already solved inputs for this code.
5. Because the code is called recursively, how can you be sure that it will ever terminate?

Hints

Assume an implementation of `quickcompare` that compares objects of type `Integer`. (Recall that `Integer` wraps the primitive `int` type. The array has to be of type `Integer` because `quickcompare` needs to compare a subclass of `Object`.) Walk through the code with the following inputs:

1. Array is unsorted, no duplicates:
   ```
   array = [ Integer(3), Integer(1), Integer(4),
   ➥Integer(5), Integer(2) ];
   ```