PART One

Language Syntax

Programmers are always surrounded by complexity . . . If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.

—C.A.R. HOARE, "THE EMPEROR'S OLD CLOTHES"

The Java language is a derivative of the C++ language with some features borrowed from Objective C, Eiffel, Smalltalk, Mesa, and Lisp. While programmers migrating from other languages to Java quickly recognize some features of and similarities to their former favorite language, those same programmers often assume that a similar feature behaves exactly the same way that it does in the older language, which is rarely the case. This phenomenon is especially true for C++ programmers. This part highlights some pitfalls and language behavior that often trip up new Java programmers.

This part contains the following 10 items:

Item 1, "When Is an "Overriden" Method Not Really Overridden?," explains the subtle difference between the dispatching of static methods and instance methods for subclasses.

Item 2, "Usage of String equals( ) Method versus the "==" Operator," examines the different methods for comparing Strings and how the constant pool confuses the issue.

3396 P-01 3/06/2000 9:04 AM Page 2

Item 3, "Java Is Strongly Typed," examines the rules concerning primitive type conversions and promotions. This Item is especially important for C and C++ programmers switching to Java.

Item 4, "Is That a Constructor?," presents a classic, yet simple, pitfall. When training new Java students, this Item always causes an astonished outburst like "the compiler doesn't catch that?" You will chuckle only if you have never been bitten by this one.

Item 5, "Cannot Access Overridden Methods," takes another look at method dispatching in Java. After reading this Item, you will fully understand the issues involved.

Item 6, "Avoid the "Hidden Field" Pitfall," discusses the most common pitfall that should be covered in every introductory Java course and is usually followed by a discussion of the *this* reference.

Item 7, "Forward References," is a short Item demonstrating what forward references are and why you should avoid them.

Item 8, "Design Constructors for Extension," is based on many hours of hard-won experience. This Item is a must read for every programmer interested in developing reusable Java classes.

Item 9, "Passing Primitives by Reference," is especially useful for C and C++ programmers transitioning to Java. This Item tackles the issue of pass by reference in Java.

Item 10, "Boolean Logic and Short-Circuit Operators," covers another common pitfall using logical operators. This Item also demonstrates a clear case of when to use short-circuit operators.

Item 1: When Is an "Overridden" Method Not Really Overridden?

OK, so I admit it. The question posed in the title of this item is really a trick question. The goal of this item, though, is not to trick you, but rather to help you understand the concept of overriding methods. I'm sure that you have picked up a book or two that began explaining Java by pointing out the three main concepts of object-oriented programming: encapsulation, inheritance, and polymorphism. Understanding these three concepts is crucial to understanding the Java language. Understanding method overriding is crucial because it is a key part of inheritance.

Overriding an instance method is covered in Item 5. This item covers overriding static methods. If you didn't know there was a difference, then this item (as well as Item 5) is for you. If you have started breaking out into a sweat and

3396 P-01 3/06/2000 9:04 AM Page 3

find yourself screaming "You can't override a static method!," then you may want to relax a bit and move on to the next item. But first, see if you can figure out what the output of the following example will be.

This example is taken from the Java Language Specification, section 8.4.8.5:

```
01:  class Super
02:  {
03:      static String greeting()
04:      {
05:         return "Goodnight";
06:      }
07:
08:      String name()
09:      {
10:         return "Richard";
11:      }
12:     }

01:  class Sub extends Super
02:  {
03:      static String greeting()
04:      {
05:         return "Hello";
06:      }
07:
08:      String name()
09:      {
10:         return "Dick";
11:      }
12:  }

01:  class Test
02:  {
03:      public static void main(String[] args)
04:      {
05:         Super s = new Sub();
06:         System.out.println(s.greeting() + ", " + s.name());
07:      }
08:  }
```

The output from running class Test is:

```
Goodnight, Dick
```

If you came up with the same output, then you probably have a good understanding of method overriding. If you didn't, let's figure out why. We will start by examining each class. Class *Super* consists of the methods `greeting` and `name`. Class *Sub* extends class *Super* and also has the `greeting` and `name` methods. Class *Test* simply has a `main` method.

In line 5 of class *Test*, we create an instance of class *Sub*. Make sure that you understand that even though variable *s* has a data type of class *Super* it is still an instance of class *Sub*. If that's a bit confusing, then you can think of it this way: Variable *s* is an instance of class *Sub* that is cast to type *Super*. The next line (6) displays the value of the returned by `s.greeting()`, followed by the string "," and the value returned by `s.name()`. The key question is "Are we calling the methods of class *Super*, or are we calling the methods of class *Sub*?" Let's first figure out if we are calling class *Super*'s `name` method or class *Sub*'s `name` method. The `name` method in both the *Super* class and the *Sub* class is an instance method, not a static method. Because class *Sub* extends class *Super* and has a `name` method with the same signature as its parent class, the `name` method in class *Sub* overrides the `name` method in class *Super*. Because variable *s* is an instance of class *Sub* and class *Sub*'s `name` method overrides class *Super*'s `name` method, the value of `s.name()` is "Dick."

We are half way there. Now we need to figure out if the `greeting` method is being called by class *Super* or by class *Sub*. Notice that the `greeting` method in both the *Super* class and the *Sub* class is a static method, also known as a "class" method. Despite the fact that the `greeting` method of class *Sub* has the same return type, the same method name, and the same method parameters, it does not override the `greeting` method in class *Super*. Because variable *s* is cast as type *Super* and the `greeting` method of class *Sub* does not override the `greeting` method of class *Super*, the value of `s.greeting()` is "Goodnight." Still confused? Follow this rule: "Instance methods are overridden, static methods are hidden." If you were the reader who was screaming "You can't override a static method!," you are absolutely right.

Now you may be asking this: "What is the difference between hiding and overriding?" You may have not recognized it, but we actually just covered the difference in the *Super/Sub* class example. Using a qualified name can access hidden methods. Even though variable *s* is an instance of class *Sub* and the `greeting` method of class *Sub* hides the `greeting` method of class *Super*, we can still access the hidden `greeting` method by casting variable *s* to class *Super*. Overridden methods differ because they cannot be accessed outside of the class that overrides them. That is why variable *s* calls class *Sub*'s `name` method and not class *Super*'s `name` method.

This item is a short explanation of a sometimes confusing aspect of the Java language. Perhaps the best way for you to understand the difference between hiding a static method and overriding an instance method is for you to create a few classes similar to class *Sub* and class *Super*. Remember, instance methods are overridden and static methods are hidden. Overridden methods cannot be accessed outside of the class that overrides them. Hidden methods can be accessed by providing the fully qualified name of the hidden method.

Now that we understand that the answer to the question posed by the title of this item is "never," I have a few more tidbits for you to keep in mind:

3396 P-01 3/06/2000 9:04 AM Page 5

- It is illegal for a subclass to hide an instance method of its super class with its own static method of the same signature. This will result in a compiler error.
- It is illegal for a subclass to override a static method of its super class with its own instance method of the same signature. This will also result in a compiler error.
- Static methods and final methods cannot be overridden.
- Instance methods can be overridden.
- Abstract methods must be overridden.

Item 2: Usage of String equals() Method versus the "==" Operator

Those of you who come from a C++ background will no doubt be confused when it comes to the *equal to* operator, ==, when used with class *String*. The main area of confusion centers around the fact that the `String.equals(...)` method and the == operator are not the same even though they sometimes produce the same result. Consider the following example:

```
01: public class StringExample
02: {
03:     public static void main (String args[])
04:     {
05:         String s0 = "Programming";
06:         String s1 = new String ("Programming");
07:         String s2 = "Program" + "ming";
08:
09:         System.out.println("s0.equals(s1): " + (s0.equals(s1)));
10:         System.out.println("s0.equals(s2): " + (s0.equals(s2)));
11:         System.out.println("s0 == s1: " + (s0 == s1));
12:         System.out.println("s0 == s2: " + (s0 == s2));
13:     }
14: }
```

The example contains three variables of type *String*, two of which are assigned the constant expression "Programming" and one of which is assigned an instance of a new *String* whose value is "Programming". Performing the comparisons using the `equals(...)` method and the "==" operator yields the following output:

```
s0.equals(s1): true
s0.equals(s2): true
s0 == s1: false
s0 == s2: true
```

3396 P-01 3/06/2000 9:04 AM Page 6

Using the `equals(…)` method to compare strings will perform a character-by-character comparison on the strings and return a true if the strings are equal. In this case, all three strings are equal so when comparing the string *s0* to either *s1* or *s2* we get a *true* return value. When the "==" operator is used, the references to the string instances are compared. In this case *s0* is not the same instance as *s1*, but *s0* and *s2* are the same object. How can *s0* and *s2* be the same object? The answer to this question comes from the Java Language Specification in the section on String Literals. In this example "Programming," "Program," and "ming" are all string literals[1] and are computed at compile time. When a string is formed by concatenating many string literals, such as *s2*, the result is also computed at compile time to be a string literal. Java ensures that there is only one copy of a string literal so when "Programming" and "Program" + "ming" are determined to have the same value, Java sets both variable references to the same literal reference. Java tracks string literals in the "constant pool."

The "constant pool" is something that is computed at compile time and stored with the compiled *.class* file. It contains information on methods, classes, interfaces, ..., and string literals. When the JVM loads the *.class* file and the *s0* and *s2* variables are resolved, the JVM does something called *constant pool resolution*. The process of constant pool resolution for string follows these steps, as taken from the *Java Virtual Machine Specification (5.4)*:

- If another constant pool entry tagged `CONSTANT_String`[2] and representing the identical sequence of Unicode characters has already been resolved, then the result of resolution is a reference to the instance of class `String` created for that earlier constant pool entry.
- Otherwise, if the method `intern` has previously been called on an instance of class *String* containing a sequence of Unicode characters identical to that represented by the constant pool entry, then the result of resolution is a reference to that same instance of class *String*.
- Otherwise, a new instance of class *String* is created containing the sequence of Unicode characters represented by the *CONSTANT_String* entry; that class instance is the result of resolution.

What this says is that the first time a string is resolved from the constant pool, an instance of the string is created on the Java heap. Any other subsequent references to the same literal in the constant pool always returns the reference of the string instance that was already created. When the JVM processes line 6 it creates a copy of the string literal "Programming" into another instance

[1]From the Java Language Specification: "A *string literal* consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence."

[2]This is used internal to the .class file to identify string literals.

3396 P-01 3/06/2000 9:04 AM Page 7

of class *String*. So when the references to *s0* and *s1* are compared the result is false because they are not the same object. This is why the behavior *s0 == s1* will sometimes be different from the behavior of *s0.equals(s1)*. The first compares the object reference values; the second actually does a character-by-character comparison.

The "constant pool" that exists in the *.class* file is loaded into memory by the JVM and can be extended at runtime. The method `intern()` mentioned previously serves this purpose for instances of class *String*. When the `intern()` method is called on an instance of *String* it will follow the same rules previously outlined for constant pool resolution, except for step 3. Because the instance already exists there is no need to create another one, so the existing instance reference is added to the constant pool. Let's look at another example.

```
01: import java.io.*;
02:
03: public class StringExample2
04: {
05:     public static void main (String args[])
06:     {
07:         String sFileName = "test.txt";
08:         String s0 = readStringFromFile(sFileName);
09:         String s1 = readStringFromFile(sFileName);
10:
11:         System.out.println("s0 == s1: " + (s0 == s1));
12:         System.out.println("s0.equals(s1): " + (s0.equals(s1)));
13:
14:         s0.intern();
15:         s1.intern();
16:
17:         System.out.println("s0 == s1: " + (s0 == s1));
18:         System.out.println("s0 == s1.intern(): " +
19:                                            (s0 == s1.intern()));
20:     }
21:
22:     private static String readStringFromFile (String sFileName)
23:     {
24:         //...read string from file...
25:     }
26: }
```

This example does not set the values of *s0* and *s1* to string literal values. Instead it reads strings from a file at runtime and assigns the instances created from the method `readStringFromFile(...)` to the variables. After line 9 is processed two new string instances will have been created that have identical character values. When you look at the output that results from line 11 and 12 you will notice once again that the objects are not the same, but the contents of the objects are. Here is the output:

```
s0 == s1: false
s0.equals(s1): true
s0 == s1: false
s0 == s1.intern(): true
```

What line 14 does is add the reference of the *String* instance stored by *s0* into the constant pool. When line 15 is processed, the call to `intern()` simply brings back a reference to *s0*. So the output from lines 17 and 18 is what we expected—there are still two distinct instances of class *String*, so *s0 == s1* is false, and because a call to `s1.intern()` brings back the value from the constant pool (which is *s0*), the expression *s0 == s1.intern()* is true. If we wanted *s1's* instance to be in the constant pool, we would have to first set *s0* to null, then run the garbage collector to reclaim the string instance that was pointed to be *s0*. After *s0* was reclaimed a call to `s1.intern()` would add it to the constant pool.

In summary, you should always use the `String.equals(…)` method for doing equality comparisons, not the == operator. If your heart is set on using the == operator you can do so with the help of the `intern()` method. The `n.equals(m)` method returns the same result as the *n.intern() == m.intern()* statement, where *n* and *m* are references to instances of class *String*. The `intern()` method is at your disposal if you determine that you will benefit from using the constant pool.

Item 3: Java Is Strongly Typed

Every Java developer needs a good understanding of the primitive types Java supports. What are the pitfalls? How are they different from the language you used to use? Like many languages, Java is strongly typed, supporting eight primitive data types. These primitives are the building blocks from which objects are constructed. By strictly checking the usage of these types, the Java compiler is able to catch many simple errors early in the development process.

Most developers are familiar with data types and the values and operations associated with them. There are a few subtleties in Java you should be aware of. Unlike when using other languages, because Java's primitive types are always represented consistently in the JVM, you can write code that relies on that representation without affecting portability. This makes bit-manipulation safer to perform.

Also, *boolean* types are not convertible. Unlike C or C++, Java does not let you write code that converts between *boolean* and non-*boolean* types. If you've used either of those languages, you've probably written some "elegant" code hacks that relied on *boolean* false being equal to zero or true being nonzero.

3396 P-01 3/06/2000 9:04 AM Page 9

In C, you could write code that checks the return value of a function like this:

```
value = get_value();
if (value) do_something;
```

Similar code will fail to compile in Java. The conditional statement is expecting a *boolean*, so you must give it one:

```
value = getValue();
if (value != null) doSomething;
```

Type Conversion

Because conversion of primitive types can happen implicitly in Java, you need to understand when and how it works. Conversion of non-*boolean* types is logical, and generally the compiler will warn you if your code could result in a loss of precision.

Arithmetic operations in Java are subject to the same potential problems as other languages. Most developers have written code that results in the accidental truncation of data. For example, line 10 in the Truncation class will print out "2.0" instead of the "2.4" output produced by lines 11 and 12.

```
01: public class Truncation
02: {
03:    static void printFloat (float f)
04:    {
05:       System.out.println ("f = " + f);
06:    }
07:
08:    public static void main (String[] args)
09:    {
10:       printFloat (12 / 5);      // data lost!
11:       printFloat ((float) 12 / 5);
12:       printFloat (12 / 5.0f);
13:    }
14: }
```

Because 12 and 5 are both integers, the result of the expression in line 10 is an integer; therefore the fraction is lost. The fact that the `printFloat` method is expecting a *float* does not matter; the truncation has already been done. The fix is simple: As long as either value in the expression is a *float*, the other will be promoted to a *float* also. So, lines 11 and 12 both work fine.

Widening

As long as your values can be converted without loss of magnitude, this conversion will happen automatically. In these cases, the conversion is referred to

3396 P-01 3/06/2000 9:04 AM Page 10

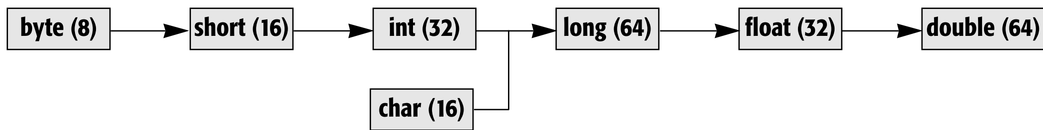

Figure 3.1 Widening conversions.

as *widening* because the types are being converted into a type capable of storing larger values. The automatic conversion is called *promotion*.

For example, you can assign a byte value into an *int* variable because this cannot result in a loss of magnitude or precision. The widening conversions are shown in Figure 3.1. The number of bits used to store each type is shown in parentheses. As long as you convert to a type with more bits, you will not lose any information.

Note that if you convert an *int* or *long* into a *float*, or a *long* into a *double*, you might lose some precision. In other words, some of the least significant bits may be lost. This kind of promotion can happen implicitly. The output from this example (–46) shows a loss of precision that occurs without any compiler warning.

```
public class LostPrecision
{
   public static void main (String[] args)
   {
      int orig = 1234567890;
      float approx = orig;
      int rounded = (int) approx; // lost precision
      System.out.println (orig - rounded);
   }
}
```

Widening is described in more detail in section 5.1.2 of the JLS: "The resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode. Despite the fact that loss of precision may occur, widening conversions among primitive types never result in a runtime exception."

Narrowing

Narrowing conversions (any conversion other than a left-to-right trace on Figure 3.1) can result in a loss of information. For example, if you try to convert a floating-point type to an integer, or when you risk overflow by converting a *long* to a *short*, you'll get a compiler error. You can avoid this error by including an explicit cast, which essentially tells the compiler "I know what I'm doing and accept the risk."

3396 P-01 3/06/2000 9:04 AM Page 11

Implicit Type Conversion

Implicit conversion means a conversion that happens automatically, without requiring an explicit cast operator. This happens only in the case of widening conversions, with one exception. For convenience, narrowing conversions may be implicit if the variable is a *byte*, *short*, or *char*, and the expression is a constant *int* value that will fit into the variable.

For example, the assignment in line 7 of the TypeConversion program will compile because 127 can be stored as a byte (range –128 to +127), but line 8 will not because 128 is too large.

```
01: public class TypeConversion
02: {
03:    static void convertArg (byte b) { }
04:
05:    public static void main (String[] args)
06:    {
07:       byte b1 = 127;
08:       // byte b2 = 128;       // won't compile
09:
10:       // convertArg (127);    // won't compile
11:       convertArg ((byte)127);
12:
13:       byte c1 = b1;
14:       // byte c2 = -b1;       // won't compile
15:       int i = ++b1;           // overflow
16:       System.out.println ("i = " + i);
17:    }
18: }
```

Implicit type conversion can happen in three situations: assignments, method calls, and arithmetic operations. Assignment statements store the value from the right-hand expression into a variable, and conversion is necessary if the types are different.

Similarly, when you make a method call, your arguments may need to be converted. For example, the `Math.pow()` method requires its arguments to be *double*s, and you may wish to use integer values. Because that's a widening conversion, you don't have to explicitly cast your arguments. Note that unlike assignment statements, implicit narrowing conversions are not supported for method calls. So, line 10 will not compile, but line 11, which is an explicit cast, will compile fine.

The third case is called *arithmetic promotion*, and it happens whenever you perform arithmetic operations using values with different types—for example, if you want to add an *int* and a *float*, or when you want to compare a *short* and a *double*. In these cases, the narrower type is converted to the wider type.

3396 P-01 3/06/2000 9:04 AM Page 12

Also, all *byte*, *short*, and *char* values are always promoted into *int* values (at least). This is true for most (but not all) unary operators, such as the unary minus operator (–) in line 14. If you uncomment line 14 and try to compile the program, you'll get this error message:

```
TypeConversion.java:14: possible loss of precision
found   : int
required: byte
```

The compiler promoted *b1* to an *int*, and then it warned you that the *int* value might not fit into the *byte* variable. Based on that, you might expect (and hope) that the *byte* value in line 15 would be automatically promoted to an integer as it was incremented and that line 16 would print out "i = 128." Instead, an overflow occurs, and the output shows a negative number: "i = –128."

Item 4: Is That a Constructor?

Did you ever have a bug that was caused by a simple mistake, yet it took you all day to figure out? The nature of this type of bug is such that you may discover it in a minute or you may anguish over it for hours. The following code contains one of these bugs. Let's see how long it takes you to figure it out.

```
01:  public class IntAdder
02:  {
03:        private int x;
04:        private int y;
05:        private int z;
06:
07:      public void IntAdder()
08:      {
09:      x = 39;
10:      y = 54;
11:      z = x + y;
12:      }
13:
14:      public void printResults()
15:      {
16:         System.out.println("The value of 'z' is '" + z + "'");
17:      }
18:
19:      public static void main (String[] args)
20:      {
21:          IntAdder ia = new IntAdder();
22:          ia.printResults();
23:      }
24:  }
```

3396 P-01 3/06/2000 9:04 AM Page 13

The *IntAdder* class is rather simple. It has three private field members named *x*, *y*, and *z*, a constructor, an instance method named `printResults`, and a static `main` method. Let's step through the code. On line 21, we instantiate an *IntAdder* object named *ia*. The *IntAdder* constructor on line 7 sets the value of field members *x* and *y*, and then it adds them together, storing the results in field member *z*. On line 22, we call the `printResults` method, which prints the value of *z* to the screen. The output from running the *IntAdder* class should be:

```
The value of 'z' is '93'
```

Do you agree with the expected output? If so, you missed the bug. The actual output is:

```
The value of 'z' is '0'
```

Take another look and see if you can find the problem. Still stumped? Let's take a closer look at the code. On line 21, we instantiate an *IntAdder* object name *ia*. The *IntAdder* constructor on line 7 sets the value of field members *x* and *y*, and then it adds them together, storing the results in field member *z*. Or does it? If we take a closer look at line 7 we see that *IntAdder* is actually a method, not a constructor. I mistakenly added a return type of "void" to what I intended to be my constructor. The "void" return type turned my constructor into a method. If the `IntAdder` method on line 7 is not my constructor, then what constructor is getting called when I instantiate my object? Because there is no constructor, Java provides the *IntAdder* class with a default no-argument constructor. The default no-argument constructor has no implementation. The default constructor acts as if I typed in the following constructor:

```
public IntAdder() { }
```

Therefore, the object that gets created on line 21 still has *x*, *y*, and *z* values of 0. When the `printResults` method gets called on line 22, we see the following output:

```
The value of 'z' is '0'.
```

There are a couple of key issues with this "simple" little bug. First, we notice that we did not get a compiler error or a runtime error even though we had a method with the same name as our class name. It is legal to have a method named the same as your class, but it certainly is not advised. Constructors must have the same name as the class. This makes them easy to find. Therefore, only constructors should have the same name as the class. If you have methods with the same name as the class you are likely to confuse your fellow programmers. Also, naming methods the same as the class contradicts standard conventions.

3396 P-01 3/06/2000 9:04 AM Page 14

Method names are generally verbs or verb phrases and have the first character of the first word in lowercase and the first character of subsequent words in uppercase. Class names are generally nouns or noun phrases with the first character of every word in uppercase. See section 6.8 of the Java Language Specification for further information on naming conventions. The second issue we notice in this example is that if we do not have a constructor in our class, Java will provide us with a default constructor with no implementation. This happens only if we do not include a constructor. If we have a constructor with any number of arguments then Java does not provide a default no-argument constructor.

Item 5: Cannot Access Overridden Methods

Imagine that you are maintaining an application that contains a third-party Java text editor. The text editor in your application currently supports RTF and has a spelling and grammar checker. In the program code of the application itself, you can create new documents or open new documents by accessing the editor's *DocumentManager* object. Each time one of these methods is called, a *Document* object is returned, which provides instance methods for spell checking, grammar checking, and so on. One day, you get a call from your client, who tells you that the text editor in the application needs to support HTML display and editing in addition to RTF. You feel confident you can fulfill this request because you know the third-party vendor that supplies your text editor just upgraded it to support HTML display and editing. Furthermore, the vendor guaranteed backward compatibility because all new functionality was placed in a subclass of *Document*, called *HTMLDocument*, and the Document code did not change. The new version could be plugged in immediately with no code changes, and the new features of *HTMLDocument* could be accessed by casting the *Document* object returned from the *DocumentManager* to an *HTMLDocument*. Wanting to please your client and being a little overzealous, you get the new version, add some code to take advantage of the new HTML features, and tell your client that the new functionality will be available in a month (just enough time for QA).

Feeling good about yourself, you are surprised to see a report from QA indicating a problem with the spell checker. You confirm that the spell checker is broken in the *HTMLDocument*, but thinking you're a smart Java programmer, you believe you can trick the *HTMLDocument* into using the `spellCheck()` method from the *Document* object instead. After all, the vendor stated that the *Document* code did not change. You try everything to invoke the `spellCheck()` method from *Document*—not casting *Document* to *HTMLDocument*, declaring a local *Document* variable and setting it to the *Document* object passed in, using reflection to access the `spellCheck()` method in

3396 P-01 3/06/2000 9:04 AM Page 15

Document—but nothing works. After many frustrated calls to the vendor support line, which claims that the *Document* object code did not change, you finally consult the *Java Language Specification (8.4.6.1)* and see this:

> An overridden method can be accessed by using a method invocation expression that contains the keyword *super*. Note that a qualified name or a cast to a superclass type is not effective in attempting to access an overridden method; . . .

Finally, it all makes sense! The third-party Java text editor library always creates an object of type *HTMLDocument* when you get a document from the *DocumentManager*. No matter what you try to do to the *HTMLDocument* (casting, reflection, and so forth), you will always call the `spellCheck()` method of *HTMLDocument* and not the `spellCheck()` method of *Document*. In fact, when you have a subclass that overrides instance methods of a superclass, the only way to access those overridden methods is by using *super* from within the subclass. Any external classes utilizing the subclass can never get to the overridden instance methods of the superclass. The source code that follows demonstrates this concept.

```
01: class DocumentManager
02: {
03:     public Document newDocument()
04:     {
05:         return (new HTMLDocument());
06:     }
07: }
08:
09: class Document
10: {
11:     public boolean spellCheck()
12:     {
13:         return (true);
14:     }
15: }
16:
17: class HTMLDocument extends Document
18: {
19:     public boolean spellCheck()
20:     {
21:         System.out.println("Trouble checking these darn hyperlinks!");
22:         return (false);
23:     }
24: }
25:
26: public class OverridingInstanceApp
27: {
28:     public static void main (String args[])
29:     {
30:         DocumentManager dm = new DocumentManager();
```

```
31:         Document d = dm.newDocument();
32:         boolean spellCheckSuccessful = d.spellCheck();
33:         if (spellCheckSuccessful)
34:             System.out.println("No spelling errors where found.");
35:         else
36:             System.out.println("Document has spelling errors.");
37:     }
38: }
```

Line 32 is where we attempt to call the `spellCheck()` method on what we think is a plain *Document* object. It, however, is really an instance of *HTML-Document* so the `spellCheck()` method call we make calls the one defined in the *HTMLDocument* class. The output produced from this example is this:

```
Trouble checking these darn hyperlinks!
Document has spelling errors.
```

Note that the inability to access overridden superclass methods applies only to instance methods—those methods in a class that are not static. Methods in a class that are *static* can be accessed even if they are overridden by a subclass. This can be accomplished by casting to the superclass. By changing line 11 and 19 to the following line, we change the `spellCheck()` method call to be static.

```
public static boolean spellCheck(Document d)
```

And by changing line 32 to the following line we can make the call to the static method.

```
boolean spellCheckSuccessful = d.spellCheck(d);
```

Running the modified example will produce the following output:

```
No spelling errors where found.
```

You may be wondering if the example described here has actually occurred. This example is fictitious, but cases do exist where an upgrade of a class has occurred in this manner. Any classes that currently support the old object (superclass) still think they are getting an instance of the old class, when in fact they are getting an instance of the new class (subclass with new functionality). A good example of this is Javasoft's introduction of the *Graphics2D* object when Java 2D was introduced into the JDK. All AWT methods that were passed *Graphics* objects (such as `paint(...)` and `update(...)`) in the older version of the JDK were actually being passed *Graphics2D* objects in the new version of the JDK. And yes, *Graphics2D* overrides instance methods in *Graphics*

3396 P-01 3/06/2000 9:04 AM Page 17

(example: `draw3DRect())`. If `draw3DRect()` contained a bug in the *Graphics2D* class but worked fine in the *Graphics* class, guess what—you were stuck with the method provided by *Graphics2D*.

Even though nothing can be done to access an overridden superclass instance method from outside of a subclass, it is good to know that this can be a source for errors. If you suspect that an object you are using is actually an instance of a subclass you can always call `getClass().getName()` on the object to determine its true identity. And if you are the programmer adding new functionality by subclassing, make sure that extensive compatibility tests are performed or ensure that any new functionality is performed by adding new methods and not by overriding superclass methods.

Item 6: Avoid the "Hidden Field" Pitfall

Understanding how field members are hidden in the Java language is just as important as understanding how methods get overridden. If you think you understand how field members get hidden because you understand how methods get overridden, then you better read the rest of this item. Unintentionally hiding a field member or mistakenly thinking that you have "overridden" a field member can cause undesirable results in your program.

```
01:  public class Wealthy
02:  {
03:      public String answer = "Yes!";
04:      public void wantMoney()
05:      {
06:         System.out.println("Would you like $1,000,000? > "+ answer);
07:      }
08:      public static void main(String[] args)
09:      {
10:         Wealthy w = new Wealthy();
11:         w.wantMoney();
12:      }
13:  }
```

Output:

```
Would you like $1,000,000? > Yes!
```

In this example, class *Wealthy* has an instance variable named *answer*, a `wantMoney` method, and a `main` method. The `main` method creates instance *w* of class *Wealthy*. Instance *w* calls its `wantMoney` method, which prints a question and responds with the value of the instance variable *answer*. The

previous example answers the question correctly; now let's take a look at an example that does not answer the question correctly.

```
01:  public class Poor
02:  {
03:      public String answer = "Yes!";
04:      public void wantMoney()
05:        {
06:          String answer = "No!"; // hides instance variable answer
07:          System.out.println("Would you like $1,000,000? > " + answer);
08:      }
09:      public static void main(String[] args)
10:      {
11:          Poor p = new Poor();
12:          p.wantMoney();
13:      }
14:  }
```

Output:

```
Would you like $1,000,000? > No!
```

Notice in the output of this example the response to the question has changed to "No!". The local variable *answer* hides the instance variable *answer*; therefore, the response is the value of the local variable. In an example this simple it is obvious that the local variable *answer* hides the instance variable *answer*, producing an undesired result. In larger, more complex programs, though, it can be difficult to find a problem caused by a field member that is hidden by accident. To avoid problems with data hiding it is important to understand the following:

- The different kinds of Java variables
- The scope of a variable
- Which kinds of variables can be hidden
- How those kinds of variables get hidden
- How to access a hidden variable
- How hidden variables differ from overridden methods

Kinds of Java Variables

The six kinds of variables are class variables, instance variables, method parameters, constructor parameters, exception-handler parameters, and local variables. Class variables are static data fields declared in a class declaration as

3396 P-01 3/06/2000 9:04 AM Page 19

well as static or nonstatic data fields declared in an interface declaration. Instance variables are nonstatic variables declared in a class declaration. The term "field members" refers to both class variables and instance variables. Method parameters are arguments passed to a method. Constructor parameters are arguments passed to a constructor. Exception-handler parameters are arguments passed to the catch block of a try statement. Finally, local variables are variables declared in a block of code or in a "for" statement.

This example declares a variable of each type:

```
01:  public class Types
02:  {
03:      int x;             // instance variable
04:      static int y;         // class variable
05:      public Types(String s)   // s is a constructor parameter
06:      {
07:         // constructor code ƒ.
08:      }
09:      public createURL(String urlString) //urlString is a method parameter
10:      {
11:         String name = "example";   // name is a local variable
12:         try
13:         {
14:            URL url = new URL(urlString);
15:         }
16:         catch(Exception e)     // e is a exception-handler parameter
17:         {
18:            // handle exception
19:         }
20:      }
21:  }
```

Variable Scope

Variable scope is defined as the block of code in which the variable can be referred to by its simple name. A simple name is a single identifier for a variable. The instance variable x on line 3 has a simple name of "x." Instance variables and class variables have a scope of the entire the class or interface in which the variable was declared. The scope of field members x and y is the entire body of the Types class. Method parameters have a scope of the entire body of the method. Constructor parameters have a scope of the entire body of the constructor. Exception-handler parameters have a scope of the entire body of the catch statement. The scope of a local variable is the entire block of code in which it was declared. The local variable *name*, which is declared in the `createURL` method on line 11, has a scope of the entire body of the `createURL` method.

3396 P-01 3/06/2000 9:04 AM Page 20

Which Kinds of Variables Can Be Hidden?

Instance variables and class variables can be hidden. Local variables and parameters can never be hidden. Attempting to hide a parameter with a local variable of the same name results in a compiler error. Similarly, attempting to hide a local variable with another local variable of the same name results in a compiler error.

```
01:  class Hidden
02:  {
03:      public static void main(String[] args)
04:      {
05:         int args = 0;       // illegal - results in a compiler error
06:         String s = "string";
07:         int s = 10;         // illegal - results in a compiler error
08:      }
09:  }
```

In this example, the local variable *args* cannot be named the same as the method parameter *args*. The local variable *s* on line 7 will also cause a compiler error because it cannot have the same name as another local variable in the same scope.

How Instance Variables and Class Variables Get Hidden

Field members can be hidden in part of their scope by local variables or parameters of the same name. Field members can also be hidden by a subclass's field member of the same name or through multiple inheritance. A local variable with the same name as a field member will hide that field member in the scope in which the local variable was declared. A method parameter with the same name as a field member will hide that field member in the scope of the body of the method. A constructor parameter with the same name as field member will hide that field member in the body of the constructor. And an exception-handler parameter with the same name as a field member will hide that field member in the scope of the catch block.

```
01:  public class Bike
02:  {
03:      String type;
04:      public Bike(String type)
05:      {
06:         System.out.println("type =" + type);
07:      }
08:  }
```

3396 P-01 3/06/2000 9:04 AM Page 21

In this example, the constructor parameter *type* hides the instance variable *type*. The value of the *type* variable displayed by the `System.out.println` method will be the value of the constructor parameter *type*, not the instance variable *type*.

A subclass's field member will hide a parent class's field member of the same name.

```
01:  public class Bike
02:  {
03:      String type = "generic";
04:  }
```

```
01:  public class MountainBike extends Bike
02:  {
03:      String type = "All terrain";
04:  }
```

In this example, the instance variable *type* in class *Bike* is hidden by its subclass's instance variable *type*. A subclass's class variable will hide its superclass's class variable or instance variable of the same name. Similarly, a subclass's instance variable will hide its superclass's instance variable or class variable of the same name.

Multiply inherited field members will, in effect, hide each other. This in itself does not cause a compiler error; however, any reference by simple name to the hidden field members will cause a compiler error. Fields are considered to be "multiply inherited" if two or more fields with the same name are inherited from two or more interfaces, or from an interface and superclass.

```
01:  public interface Stretchable
02:  {
03:      int y;
04:  }
```

```
01:  public class Line
02:  {
03:      int x;
04:  }
```

```
01:  public class MultiLine extends Line implements Stretchable
02:  {
03:      public MultiLine()
04:      {
05:         System.out.println("x = " + x);
06:      }
07:  }
```

This example will compile error free. If a variable named x was added to the *Stretchable* interface, then class *MultiLine* would fail to compile because it attempts to refer to the multi-inherited variable x by its simple name.

How to Access a Hidden Field Member

Most field variables can be accessed using the variable's qualified name as opposed to its simple name. The "this" keyword will qualify an instance variable that is being hidden by a local variable. The "super" keyword will qualify an instance variable that is being hidden by its subclass. A class variable can also be qualified by placing the class name and a "." before the class variable's simple name.

```
01:  public class Wealthy
02:  {
03:      public String answer = "Yes!";
04:      public void wantMoney()
05:      {
06:         String answer = "No!";
07:         System.out.println("Do you want to give me $1,000,000? > " +
08:         answer);
09:         System.out.println("Would you like $1,000,000? > " +
10:         this.answer);
11:      }
12:      public static void main(String[] args)
13:      {
14:         Wealthy w = new Wealthy();
15:         w.wantMoney();
16:      }
17:  }
```

Output:

```
Do you want to give me $1,000,000 > No!
Would you like $1,000,000? > Yes!
```

In this example, class *Wealthy* has an instance variable *answer*. The `wantMoney` method declares a local variable named *answer* that hides the instance variable *answer*. In order to give the correct response to each of the questions in the `wantMoney` method we need to access the local variable *answer* as well as the instance variable *answer*. By using the "this" keyword to qualify the instance variable we can tell the compiler that we want the instance variable and not the local variable. As is shown in the output, the response to the first question is given by the value of the local variable *answer*. The response to the second question is given by the value of the instance variable *answer*, which is qualified by the "this" keyword.

This example shows how to qualify a hidden instance variable of a parent class:

```
01:  public class StillWealthy extends Wealthy
02:  {
03:     public String answer = "No!";
```

3396 P-01 3/06/2000 9:04 AM Page 23


```
04:      public void wantMoney()
05:      {
06:         String answer = "maybe?";
07:         System.out.println("Did you see that henway? > " + answer);
08:         System.out.println("Do you want to give me $1,000,000? > " +
09:         this.answer);
10:         System.out.println("Would you like $1,000,000? > " + super.answer);
11:      }
12:      public static void main(String[] args)
13:      {
14:         Wealthy w = new Wealthy();
15:         w.wantMoney();
16:      }
17:  }
```

Output:

```
Did you see that henway? > maybe?
Do you want to give me $1,000,000 > No!
Would you like $1,000,000? > Yes!
```

Notice from the output that the response to the question on line 7 is given by the value of the local variable. The response to the question on line 8 is given by the value of the *StillWealthy* subclass's instance variable, which is qualified by using the "this" keyword. The response to the question on line 10 is given by the value of the superclass's instance variable, which is qualified by using the "super" keyword.

How Hidden Variables Differ from an Overridden Method

Hidden variables differ from overridden methods in several ways. Perhaps the most important difference is that an instance of a class cannot access its superclass's overridden method by using a qualified name or by casting the instance to that of its superclass.

```
01:  public class Wealthier extends Wealthy
02:  {
03:      public void wantMoney()
04:      {
05:         System.out.println("Would you like $2,000,000? > " + answer);
06:      }
07:      public static void main(String[] args)
08:      {
09:         Wealthier w = new Wealthier();
10:         w.wantMoney();
11:         ((Wealthy)w).wantMoney();
12:      }
13:  }
```

3396 P-01 3/06/2000 9:04 AM Page 24

Output:

```
Would you like $2,000,000? > Yes!
Would you like $2,000,000? > Yes!
```

In this example, class *Wealthier* extends class *Wealthy* and overrides the method wantMoney. The main method creates an instance *w* of class *Wealthier* and calls w.wantMoney(). Notice by the first line in the output that the $2,000,000 question is asked. The main method then casts the instance variable *w* to its parent class and once again calls the wantMoney method. Notice by the second line in the output that the $2,000,000 question is still asked. The previous example shows that from an instance of a subclass, the superclass's overridden method cannot be accessed by casting the instance to the superclass.

This example shows that a hidden variable differs from an overridden method because it can be accessed by casting an instance of the subclass to its superclass.

```
01:  public class Poorer extends Wealthier
02:  {
03:      String answer = "No!";
04:      public void wantMoney()
05:      {
06:         System.out.println("Would you like $3,000,000? > " + answer);
07:      }
08:      public static void main(String[] args)
09:      {
10:         Poorer p = new Poorer();
11:         ((Wealthier)p).wantMoney();
12:         System.out.println("Are you sure? > " + ((Wealthier)p).answer);
13:      }
14:  }
```

Output:

```
Do you want $3,000,000? ? No!
Are you sure? > Yes!
```

Class *Poorer* extends class *Wealthier*; the main method creates an instance *p* of class *Poorer*. The main method then casts instance *p* to its superclass. As was explained in the previous example, because the wantMoney method is overridden, the superclass's wantMoney method cannot be accessed by casting to the superclass. Therefore, the wantMoney method of class *Poorer* gets called, which responds with the answer “No!”. The main method then asks the question “Are you sure? >”. The answer is the value of the superclass's variable, not the value of the subclass's variable. This occurs because the subclass just hides the superclass's field member so casting the instance to its superclass allows access to its superclass's field members. Another difference between

3396 P-01 3/06/2000 9:04 AM Page 25

data hiding and method overriding is that a static method cannot override a superclass's instance method. A static variable, however, can hide a superclass's instance variable of the same name. Similarly, an instance method cannot override a superclass's method of the same name but different signature. A field member can hide a superclass's field member of the same name even if it is of a different type.

Avoiding the "hidden field" pitfall by understanding the points discussed in this item will help you achieve the desired results of your application as well as save you countless hours debugging complex programs.

Item 7: Forward References

Class variables and static initializers are executed when a class is loaded into the JVM. Section 8.5 of the Java Language Specification notes that "static initializers and class variable initializers are executed in textual order and may not refer to class variables declared in the class whose declarations appear textually after the use...." In other words, these statements are processed in the order in which they appear in the code. Normally, the compiler will catch any forward references. Consider the following code:

```
1: public class ForwardReference
2: {
3:    int first = second;  // this will fail to compile
4:    int second = 2;
5: }
```

Attempting to compile this class will result in an error:

```
ForwardReference.java:3: Can't make forward reference to second in class
ForwardReference.
```

So, even though both *first* and *second* are in the same scope, the language specification disallows this kind of invalid initialization, and the compiler will catch this.

It is possible, though, to circumvent this protection. Java allows method calls to be used to initialize class variables, and accesses of class variables by methods are not checked in this way. The program that follows will compile cleanly.

```
01: public class ForwardReferenceViaMethod
02: {
03:    static int first = accessTooSoon();
04:    static int second = 1;
05:
06:    static int accessTooSoon()
07:    {
```

```
08:       return (second);
09:    }
10:
11:    public static void main (String[] args)
12:    {
13:       System.out.println ("first = " + first);
14:    }
15: }
```

Executing it, however, results in accessing the default value of *second* (which is 0) before it gets initialized to 1. So, *first* is assigned the value of 0 instead of 1.

There's no simple solution to this problem. If you use method calls to initialize static variables, you have to ensure that those methods don't depend on other static variables that are declared later in the file.

Item 8: Design Constructors for Extension

Perhaps the most significant advantage of object-oriented languages in general, and Java in particular, is that code can be easily reused. One of the most common ways to reuse a class is to extend it, and then add or change the functionality to meet your requirements. Unfortunately, many inexperienced developers do not write code that is easily extensible.

There are many situations in the process of software development in which you must make trade-offs between multiple goals. For example, optimizing often results in code that is more complex, harder to maintain, and less portable. When it comes to extensibility, however, you don't usually lose anything when you make your code more extensible.

When you develop a class, you can encounter many pitfalls that discourage, or even prevent, extensibility. Avoiding these pitfalls as you design or implement your code is often easy, if you're aware of them.

One of the most common pitfalls I've seen is in the implementation of constructors. No matter how well designed your methods are, if you don't provide the right constructors, other developers will have trouble extending your class. Because you can't override constructors, you have to work with whatever the base class provides.

If your constructor tries to do too much, it may require any subclass to do things that are not possible for it to do. This is especially true if the developer writing the subclass doesn't have access to the source code. For example, consider the following classes that provide a simple multiple-choice menu object.

```
01: import java.awt.*;
02: import java.awt.event.*;
03: import java.io.*;
04: import java.util.*;
```

3396 P-01 3/06/2000 9:04 AM Page 27


```
05:
06: import javax.swing.*;
07: import javax.swing.event.*;
08:
09: public class ListDialog extends JDialog
10: implements ActionListener, ListSelectionListener
11: {
12:    JList model;
13:    JButton selectButton;
14:    LDListener listener;
15:    Object[] selections;
16:
17:    public ListDialog (String title,
18:                       String[] items,
19:                       LDListener listener)
20:    {
21:       super ((Frame)null, title);
22:
23:       JPanel buttonPane = new JPanel ();
24:       selectButton = new JButton ("SELECT");
25:       selectButton.addActionListener (this);
26:       selectButton.setEnabled (false); // nothing selected yet
27:       buttonPane.add (selectButton);
28:
29:       JButton cancelButton = new JButton ("CANCEL");
30:       cancelButton.addActionListener (this);
31:       buttonPane.add (cancelButton);
32:
33:       this.getContentPane().add (buttonPane, BorderLayout.SOUTH);
34:
35:       this.listener = listener;
36:       setModel (items);
37:    }
38:
39:    void setModel (String[] items)
40:    {
41:       if (this.model != null)
42:          this.model.removeListSelectionListener (this);
43:       this.model = new JList (items);
44:       model.addListSelectionListener (this);
45:
46:       JScrollPane scroll = new JScrollPane (model);
47:       this.getContentPane().add (scroll, BorderLayout.CENTER);
48:       this.pack();
49:    }
50:
51:    /** Implement ListSelectionListener. Track user selections. */
52:
53:    public void valueChanged (ListSelectionEvent e)
54:    {
55:       selections = model.getSelectedValues();
```

3396 P-01 3/06/2000 9:04 AM Page 28


```
56:       if (selections.length > 0)
57:          selectButton.setEnabled (true);
58:    }
59:
60:    /** Implement ActionListener.  Called when the user picks the
61:     * SELECT or CANCEL button. Generates the LDEvent. */
62:
63:    public void actionPerformed (ActionEvent e)
64:    {
65:       this.setVisible (false);
66:       String buttonLabel = e.getActionCommand();
67:       if (buttonLabel.equals ("CANCEL"))
68:          selections = null;
69:       if (listener != null)
70:       {
71:          LDEvent lde = new LDEvent (this, selections);
72:          listener.listDialogSelection (lde);
73:       }
74:    }
75:
76:    public static void main (String[] args) // self-testing code
77:    {
78:       String[] items = (new String[]
79:          {"Forest", "Island", "Mountain", "Plains", "Swamp"});
80:       LDListener listener =
81:          new LDListener()
82:          {
83:             public void listDialogSelection (LDEvent e)
84:             {
85:                Object[] selected = e.getSelection();
86:                if (selected != null) // null if user cancels
87:                   for (int i = 0; i < selected.length; i++)
88:                      System.out.println (selected[i].toString());
89:                System.exit (0);
90:             }
91:          };
92:
93:       ListDialog dialog =
94:          new ListDialog ("ListDialog", items, listener);
95:       dialog.show();
96:    }
97: }
```

For completeness, here are the *LDListener* and *LDEvent* classes:

```
01: public interface LDListener
02: {
03:    public void listDialogSelection (LDEvent e);
04: }
```

```
01: import java.util.EventObject;
02:
```

3396 P-01 3/06/2000 9:04 AM Page 29


```
03: public class LDEvent extends java.util.EventObject
04: {
05:    Object source;
06:    Object[] selections;
07:
08:    public LDEvent (Object source, Object[] selections)
09:    {
10:       super (source);
11:       this.selections = selections;
12:    }
13:
14:    public Object[] getSelection()
15:    {
16:       return (selections);
17:    }
18: }
```

The *ListDialog* class appears to be fairly well written, but it will turn out to be rather difficult to extend. Let's try. Suppose you have a requirement to develop a menu that will present a list of audio files to the user. You want the user to be able to hear each sound as she clicks on the audio file in your interface. Because you don't want to force your clients to specify all of the audio files, you decide to provide a simple API that accepts a directory name and determines the list of audio files in that directory.

This would seem to be a simple extension of the *ListDialog* class. You know you'll need to listen for *ListSelectionEvents*, so you can play the selected sound. This is easy because you have access to the model, and you can simply call `addListSelectionListener(…)`, as shown in line 7 in the code that follows. If the model had been *private* with no accessor method, you would have been unable to add your listener, and you would have had to start from scratch. So far, so good.

When you try to extend *ListDialog*, however, its only constructor (line 17 in the first listing) requires that you send an array of *String* items—which you don't yet have. And, because the call to `super(…)` must be the first thing your constructor does, there's no way for you to get the list of items and then create the *ListDialog*.

But don't give up yet; there's got to be a workaround. You look at the *javadoc* documentation and notice that there's a `setModel(…)` method that sounds promising. So, you first try to implement your constructor by instantiating the *ListDialog* with a *null* value for the *items* argument, and then you call `setModel(…)` after you've determined your list of files (lines 4–6).

```
01: public SoundDialog (String title, LDListener listener,
02:                     String path)
03: {
04:    super (title, null, listener);
```

3396 P-01 3/06/2000 9:04 AM Page 30


```
05:    String[] items = getItems (path);
06:    setModel (items);
07:    model.addListSelectionListener (this);
08:    model.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);
09: }
```

This seems reasonable, but when you try to run your class, you get the following error:

```
Exception occurred during event dispatching:
java.lang.NullPointerException
   . . .
   at java.awt.Window.pack(Window.java:259)
   at ListDialog.setModel(ListDialog.java:48)
   at ListDialog.<init>(ListDialog.java:36)
   at SoundDialog.<init>(SoundDialog.java:18)
   at SoundDialog.main(SoundDialog.java:89)
```

Examining this stack trace, you realize the *ListDialog* is calling its `setModel(...)` method from its constructor, and that is producing these undesirable results. After some debugging, you determine that the *NullPointerException* is caused by the empty model (trying to call `pack()` on a *JList* with no items).

Now what? Well, lucky for you, *ListDialog*'s `setModel(...)` method isn't *private*, so you can override it with your own, safer version. Notice that you had to move your *SoundDialog*'s calls, which change the model into your version of `setModel` (lines 27–28) because it's possible for one of your clients to use this *public* method, too. Here's the working version:

```
01: import java.applet.*;
02: import java.awt.*;
03: import java.io.*;
04: import java.net.*;
05:
06: import javax.swing.*;
07: import javax.swing.event.*;
08:
09: public class SoundDialog extends ListDialog
10: implements FilenameFilter, ListSelectionListener
11: {
12:    String selection;
13:
14:    public SoundDialog (String title, LDListener ldl, String path)
15:    {
16:       super (title, null, ldl);
17:       String[] items = getItems (path);
18:       setModel (items);
19:    }
```

3396 P-01 3/06/2000 9:04 AM Page 31


```
20:
21:    public void setModel (String[] items)
22:    {
23:       if (items != null)
24:       {
25:          super.setModel (items);
26:          model.addListSelectionListener (this);
27:          model.setSelectionMode
28:            (ListSelectionModel.SINGLE_SELECTION);
29:       }
30:    }
31:
32:    public String[] getItems (String path)
33:    {
34:       File file = new File (path);
35:       File soundFiles[] = file.listFiles (this);
36:       String[] items = new String [soundFiles.length];
37:       for (int i = 0; i < soundFiles.length; i++)
38:          items[i] = soundFiles[i].getName();
39:       return (items);
40:    }
41:
42:    // implement FilenameFilter
43:    public boolean accept (File dir, String name)
44:    {
45:       return (name.endsWith (".aiff") ||
46:               name.endsWith (".au")   ||
47:               name.endsWith (".midi") ||
48:               name.endsWith (".rmf")  ||
49:               name.endsWith (".wav"));
50:    }
51:
52:    // implement ListSelectionListener
53:    public void valueChanged (ListSelectionEvent e)
54:    {
55:       super.valueChanged (e);
56:       JList items = (JList) e.getSource();
57:       String fileName = items.getSelectedValue().toString();
58:       if (!fileName.equals (selection))
59:       {
60:          selection = fileName;
61:          play (selection);
62:       }
63:    }
64:
65:    private void play (String fileName)
66:    {
67:       try
68:       {
69:          File file = new File (fileName);
70:          URL url = new URL ("file://" + file.getAbsolutePath());
```

```
71:          AudioClip audioClip = Applet.newAudioClip (url);
72:          if (audioClip != null)
73:             audioClip.play();
74:       }
75:       catch (MalformedURLException e)
76:       {
77:          System.err.println (e + ": " + e.getMessage());
78:       }
79:    }
80:
81:    public static void main (String[] args) // self-test
82:    {
83:       LDListener listener =
84:          new LDListener()
85:          {
86:             public void listDialogSelection (LDEvent e)
87:             {
88:                Object[] selected = e.getSelection();
89:                if (selected != null) // null if user cancels
90:                   for (int i = 0; i < selected.length; i++)
91:                      System.out.println (selected[i].toString());
92:                System.exit (0);
93:             }
94:          };
95:       SoundDialog dialog =
96:          new SoundDialog ("SoundDialog", listener, ".");
97:       dialog.show();
98:    }
99: }
```

All of these workarounds would have been unnecessary if the original version of *ListDialog* had only provided the appropriate constructors. If you find yourself implementing a constructor that calls a `private` method or requires numerous arguments, make sure you consider the implications. Constructors with lots of arguments can be appropriate, especially if you're providing them as a convenience in addition to other, less burdensome constructors. But if it's your only constructor, consider adding additional versions that accept variations.

If you provide a no-argument constructor and the right additional methods to instantiate your class, you'll be fine. This does require some extra work, in that you have to be careful in your other methods not to use variables that haven't been initialized yet.

If your bare-bones constructor doesn't have enough information to fully instantiate your class, consider restricting access to it. You can do this by leaving off the "public" keyword. This will still allow any subclasses to call it, but you won't have to worry as much about other developers using it to create invalid objects.

3396 P-01 3/06/2000 9:04 AM Page 33

Notice that (like most Swing components) the *JList* class we've been using in this example does provide a no-argument constructor, even though an empty *JList* is not really valid. If you use this constructor, you need to call `setListData(...)` or `setModel(...)` before using the *JList*. Obviously, the Swing components were designed to be easy to extend.

Item 9: Passing Primitives by Reference

If you are a C or C++ programmer you may have been a little disappointed to learn that Java exposes no concept of a pointer to a programmer. The lack of pointers in Java prevents at least two things you may be accustomed to doing with pointers: performing pointer arithmetic and returning multiple values from a function. As it turns out, only the first one is not allowed in Java. The second, returning multiple values from a Java method, is possible by passing arguments by reference instead of by value. All instantiated objects in Java are accessed by using a reference. Variable types that refer to classes, interfaces, arrays, and objects are all classified as *reference* types. Java also provides another type called a *primitive*. Primitives are used to store a specific type of information, such as a number or character. Java provides the following primitive types: *boolean*, *byte*, *short*, *int*, *long*, *char*, *float*, and *double*. An important concept to get across is that primitive types are not objects and therefore cannot be passed by reference.

Did you just read what you thought you did? Then what is the point of this section if primitives cannot be passed by reference? And why would you want to pass a primitive by reference anyway?

Even though primitives cannot be passed by references directly, they can be passed indirectly. What this implies is that you must use a reference type to wrap a primitive if you want to pass a primitive by reference. I can think of at least two cases where passing a primitive by reference is desired:

- Returning multiple primitive values from a function
- Passing primitive values to methods that accept objects only as arguments (for example, `Hashtable`)

Let's look at an example of doing these two cases the wrong way. Take a look at the *PassPrimitiveByReference1* source listed in the code that follows. This class tries to accomplish both of the desired objectives previously listed, but you can see that there are two major problems. The first problem you see is that the program will not compile because we are trying to pass primitive types to the `Hashtable.put()` method. This particular method accepts only objects as arguments. The second problem you see is code in the `getPersonInfo()` method. The intent of the method is to be able to return multiple values to the

3396 P-01 3/06/2000 9:04 AM Page 34

method caller by assigning values to the method arguments. The problem here is that primitive types are passed only by value, which means when they are passed to a method call, a copy of the primitive type is made for the exclusive use of the method. When the `getPersonInfo()` method assigns new values to the method arguments, it changes only the copies of the variables available to the method; it does not change the original variables.

```
01: import java.util.Hashtable;
02:
03: public class PassPrimitiveByReference1
04: {
05:     public static void main (String args[])
06:     {
07:         String  name = null;
08:         int     age = 0;
09:         float   weight = 0f;
10:         boolean isMarried = false;
11:
12:         getPersonInfo(name, age, weight, isMarried);
13:
14:         System.out.println("Name: " + name +
15:                            "\nAge: " + age +
16:                            "\nWeight: " + weight +
17:                            "\nIs Married: " + isMarried);
18:
19:         storePersonInfo(name, age, weight, isMarried);
20:     }
21:
22:     private static void getPersonInfo (String name, int age,
23:                             float weight, boolean isMarried)
24:     {
25:         name = "Robert Smith";
26:         age = 26;
27:         weight = 182.7f;
28:         isMarried = true;
29:     }
30:
31:     private static void storePersonInfo (String name, int age,
32:                               float weight, boolean isMarried)
33:     {
34:         Hashtable h = new Hashtable();
35:         h.put("name", name);
36:         h.put("age", age);              // produces compile time error
37:         h.put("weight", weight);        // produces compile time error
38:         h.put("isMarried", isMarried);  // produces compile time error
39:     }
40: }
```

Now let's look at the solution to the problems. The *PassPrimitiveByReference2* class listed in the code that follows solves both problems presented in

the *PassPrimitiveByReference1* example. The problem with the *Hashtable* has been corrected by using the Java class equivalents of the primitive types. In the java.lang package, you will find corresponding classes to all the primitive types. In this example, we used *Integer* for *int*, *Float* for *float*, and *Boolean* for *boolean*. The Java class equivalent of a given primitive type serves to encapsulate the primitive type and provides various utility methods. This could have been the solution to our second problem as well, but the Java primitive class equivalents are immutable (they contain no `set(...)` method). Instead, we solve the second problem by creating a one-dimensional array of a given type and passing the array to the method. Remember that earlier I mentioned that arrays are reference types. In this particular case, it comes in handy because now I can set the values of primitive types in the `getPersonInfo()` method, and they are available to the calling method.

```
01: import java.util.Hashtable;
02:
03: public class PassPrimitiveByReference2
04: {
05:     public static void main (String args[])
06:     {
07:         String[]  name = new String[1];
08:         int[]     age = new int[1];
09:         float[]   weight = new float[1];
10:         boolean[] isMarried = new boolean[1];
11:
12:         getPersonInfo(name, age, weight, isMarried);
13:
14:         System.out.println("Name: " + name[0] +
15:                            "\nAge: " + age[0] +
16:                            "\nWeight: " + weight[0] +
17:                            "\nIs Married: " + isMarried[0]);
18:
19:         String  name2 = name[0];
20:         Integer age2 = new Integer(age[0]);
21:         Float   weight2 = new Float(weight[0]);
22:         Boolean isMarried2 = new Boolean(isMarried[0]);
23:
24:         storePersonInfo(name2, age2, weight2, isMarried2);
25:     }
26:
27:     private static void getPersonInfo (String[] name, int[] age,
28:                             float[] weight, boolean[] isMarried)
29:     {
30:         name[0] = "Robert Smith";
31:         age[0] = 26;
32:         weight[0] = 182.7f;
33:         isMarried[0] = true;
34:     }
```

```
35:
36:     private static void storePersonInfo (String name, Integer age,
37:                               Float weight, Boolean isMarried)
38:     {
39:         Hashtable h = new Hashtable();
40:         h.put("name", name);
41:         h.put("age", age);
42:         h.put("weight", weight);
43:         h.put("isMarried", isMarried);
44:     }
45: }
```

Running *PassPrimitiveByReference2* will produce the following output:

```
Name: Robert Smith
Age: 26
Weight: 182.7
Is Married: true
```

This demonstration of passing primitives by an array reference by no means advocates its use. I could have easily created a class that contained the primitives I wanted as instance variables and passed the object to the `getPersonInfo()` method to have the values set. Encapsulating things in an object is usually the better way to do it, but you may run into situations where passing primitives by an array reference is the only option.

Item 10: Boolean Logic and Short-Circuit Operators

Like C++ (and C), Java supports *bitwise operators* (& and |) for bit-masking operations. Unlike C++, Java supports both *boolean logical operators* (& and |) and *conditional and/or operators* (&& and ||). This can lead to some problems if you're not careful.

If you were to program a line like this in C++, many compilers would warn you.

```
if (ptr != null & ptr->count > 1)  // wrong operator!
```

For example, the Gnu C compiler produces the warning:

```
warning: suggest parentheses around comparison in operand of &
```

Although this is legal code, it's probably not what you want. If the *ptr* variable is null, you'll attempt to dereference a null pointer in the right-hand comparison

3396 P-01 3/06/2000 9:04 AM Page 37

(and the program will crash). The compiler can make the assumption that you probably did not want to do a bitwise AND there.

In Java, however, the compiler can't make that assumption. If both sides of the expression are *boolean* values, then the "&" operator will be treated as a *boolean* logical operator, not a bitwise AND (see JLS 15.21.2). Let's say you want to check if a *Vector* object has any elements before you use it. You might unintentionally code it this way:

```
if ((v != null) & (v.size() > 0))  // wrong operator!
```

The compiler won't produce a warning because that may very well be what you intended. The code will still throw a *NullPointerException* if *v* is null.

What you want, of course, is the *short-circuit* operator. The *boolean* logical and conditional operators provide similar functionality, with one significant difference. The conditional operators (&& and ||) will short-circuit. That is, if the result of the first (left) expression is enough to determine the result of the conditional operation, then the second (right) expression will not be evaluated. Here's the correct line:

```
if ((v != null) && (v.size() > 0))
```

The same behavior holds true for the OR operators, though these are much less problematic than the common *null*-check described previously. The "|" operator with *boolean* operands will be treated as a logical operator, returning true if either operand is true. Both operands will be evaluated, even if the first one is true. The "||" conditional operator will short-circuit instead. You're not likely to get in trouble using the "|" operator, unless your operands produce some side-effect you weren't counting on.

The simple solution is to always use the conditional operators && and ||. They're safer and more efficient because fewer operands need to be evaluated. If you need to ensure both operands are evaluated, make sure you comment your use of the logical & or | operator.

3396 P-01 3/06/2000 9:04 AM Page 38