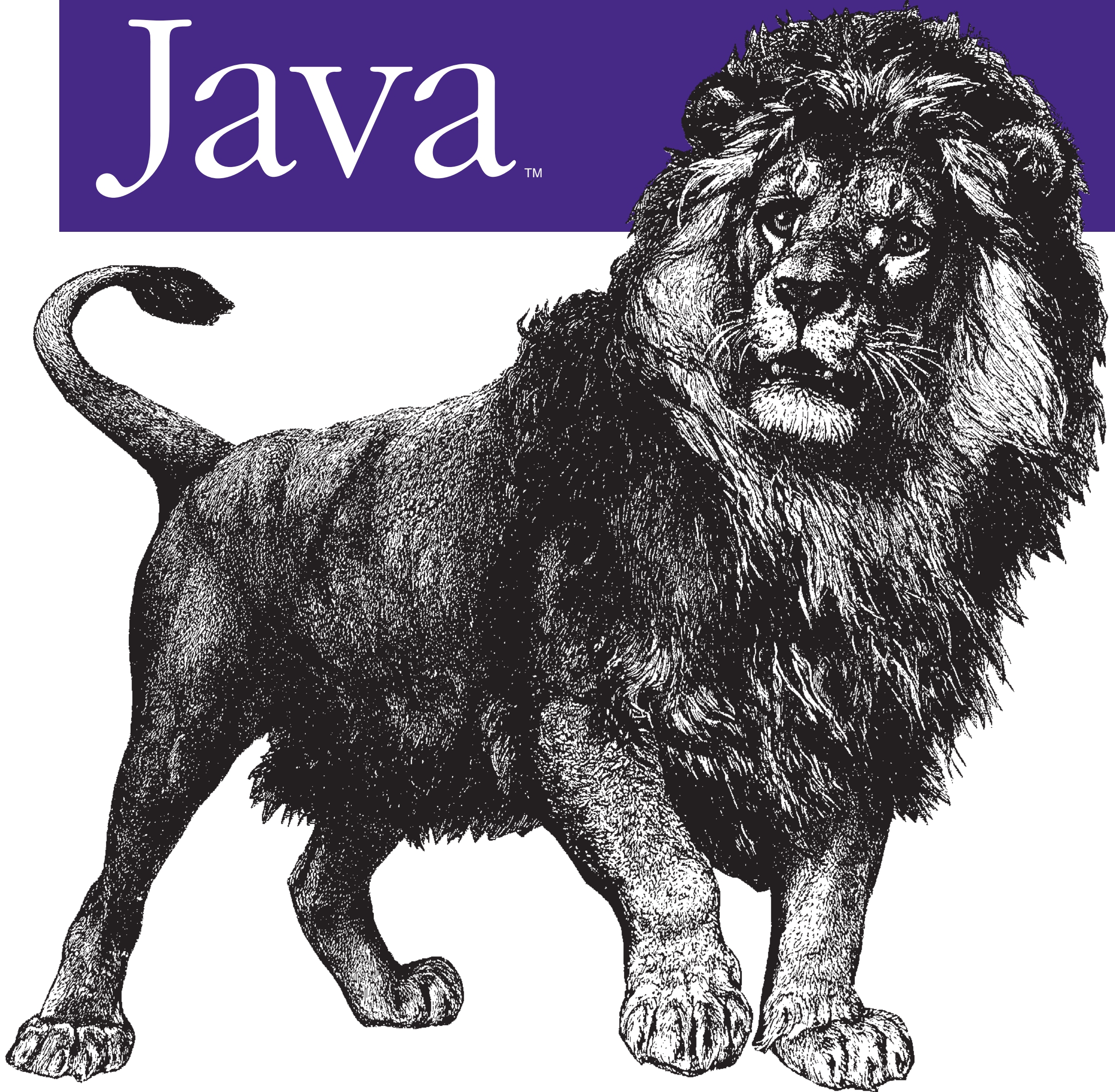
Secrets of the Java Masters

Hardcore Java™



O'REILLY®

Robert Simmons Jr.

CHAPTER 2
The Final Story

One fundamental principle of programming is that, generally, it is best to swap a logic error for a compiler error. Compiler errors tend to be found in seconds and are corrected just as fast. Syntax errors are a good example. A missing semicolon can make things confusing. If the compiler error is something particularly cryptic, the resolution may take as long as a couple of minutes to discover.

Logic errors, on the other hand, are the bane of all programmers. They hide and hate to reveal themselves. Logic errors seem to have minds of their own, constantly evading detection and dodging your efforts to pin down their cause. They can easily take a thousand times more effort to solve than the worst compiler errors. Worst of all, many logic errors are not found at all and occur only intermittently in sensitive places, which causes your customers to scream for a fix. Logic errors often require you to throw thousands of man-hours at them, only to finally discover that they are minor typos.

The Java keyword `final` can be instrumental in turning thousands of logic errors into compiler errors without too much effort. With some training in coding standards and some code retrofitting, you can save an enormous amount of man-hours that are better spent elsewhere. Also, you can save your support departments from having to deal with irate customers.

Final Constants

Final constants are a good place to start, since many of you are already familiar with the concept. Consider the code in Example 2-1.

Example 2-1. A class that doesn't use constants

```
package oreilly.hcj.finalstory;
public class FinalConstants {

  public static class CircleTools {
```

Example 2-1. A class that doesn't use constants (continued)

```
    public double getCircleArea(final double radius) {
      return (Math.pow(radius, 2) * 3.141);
    }

    public double getCircleCircumference(final double radius) {
      return ((radius * 2) * 3.141);
    }

    public double getCircleExtrudedVolume(final double radius,
                                          final double height) {
      return ((radius * 2 * height) * 3.141);
    }
  }
}
```

The problem with this code is that the developer has to change all three instances of the value `3.141`, his estimate for π, in all three methods if he wants to make his calculations more precise. Seasoned developers will see the opportunity for a class-scoped constant, as seen in Example 2-2.

Example 2-2. Simple constants using final

```
package oreilly.hcj.finalstory;
public class FinalConstants {

  public static class CircleToolsBetter {
    /** A value for PI. **/
    public final static double PI = 3.141;

    public double getCircleArea(final double radius) {
      return (Math.pow(radius, 2) * PI);
    }

    public double getCircleCircumference(final double radius) {
      return ((radius * 2) * PI);
    }

    public double getCircleExtrudedVolume(final double radius,
                                          final double height) {
      return ((radius * 2 * height) * PI);
    }
  }
}
```

This code is much better. Now the developer can change the constant, and this one change will propagate throughout the class. The reason I am beating this particular dead horse is because there are some traps involving constants that trip up even experienced developers.

Public Primitives and Substitution

The first of these traps involves public primitive constants that are used by other code. Because primitive `final`s are substituted at compile time with their values, if you change a `final` that is used by other classes, you must remember to recompile those other classes or your change will not take effect. The same rule applies to constants of type `java.lang.String`. Although `String` is a constructed type, it is also substituted at compile time. All constructed types other than `String`, mutable or not, are not substituted at compile time. To understand how this works, look at Example 2-3.

Example 2-3. Various final variables

```
package oreilly.hcj.finalstory;
public class FinalReplacement {
  /** A string constant */
  public final static String A_STRING = "Hardcore Java";

  /** An int constant. */
  public final static int AN_INT = 5;

  /** A double constant. */
  public final static double A_DOUBLE = 102.55d;

  /** An array constant. */
  public final static int[] AN_ARRAY = new int[] {1, 2, 3, 6, 9, 18, 36};

  /** A color constant. */
  public final static Color A_COLOR = new Color(45, 0, 155);

  public void someMethod( ) {
    System.out.println(A_STRING);
    System.out.println(AN_INT);
    System.out.println(A_DOUBLE);
    System.out.println(AN_ARRAY);
    System.out.println(A_COLOR);
  }
}
```

Once the compiler sees code such as this, it starts substituting out the primitives and `String` objects. After the first pass of the compiler, the class will look something like this:

```
package oreilly.hcj.finalstory;
public class FinalReplacement {
  /** A string constant */
  public final static String A_STRING = "Hardcore Java";

  /** An int constant. */
  public final static int AN_INT = 5;

  /** A double constant. */
  public final static double A_DOUBLE = 102.55d;
```

```
  /** An array constant. */
  public final static int[] AN_ARRAY = new int[] {1, 2, 3, 6, 9, 18, 36};

  /** A color constant. */
  public final static Color A_COLOR = new Color(45, 0, 155);

  public void someMethod( ) {
    System.out.println("Hardcore Java");
    System.out.println(5);
    System.out.println(102.55d);
    System.out.println(AN_ARRAY);
    System.out.println(A_COLOR);
  }
}
```

The compiler will concatenate consecutive String literals to form one literal. Therefore, the following two lines are identical from the point of view of the compiler:

```
public final static String A_STRING = "Hardcore Java";
public final static String A_STRING = "Hardcore"+ "Java";
```

Both of these lines would result in an identical declaration that is a string constant. Also, this optimization technique applies to where there are consecutive string literals in your code.

The primitive and String constants were substituted while the other constructed types were left as variables. Since this code is all in one class, if you change a constant, you have to recompile this class anyway. However, if another class (for example, ExternalUser) is using the constant A_STRING and you change it in FinalReplacement, you have a problem. ExternalUser will have to be recompiled to trigger a resubstitution using the new A_STRING value, but the Java compiler will not notice this dependency. Here's a simple version of ExternalUser:

```
package oreilly.hcj.finalstory;
public class ExternalUser {
  public static void main(String[] args) {
    System.out.println("The title of the book is: " +
                       FinalReplacement.A_STRING + ".");
  }
}
```

This extremely simple class uses the A_STRING constant from the FinalReplacement class. If you run the main( ) method, the output will look like the following:

```
>ant -Dexample=oreilly.hcj.finalstory.ExternalUser run_example
run_example:
     [java] The title of the book is: Hardcore Java.
```

Now change the value of A_STRING to "Java Hardcore" in the FinalReplacement class:

```
/** A string constant */
  public final static String A_STRING = "Java Hardcore";
```

Recompile FinalReplacement using the following command:

```
>ant -Dexample=oreilly/hcj/finalstory/FinalReplacement.java compile_example
compile_example:
    [javac] Compiling 1 source file to C:\dev\hcj\bin
```

Now run the ExternalUser example again:

```
>ant -Dexample=oreilly.hcj.finalstory.ExternalUser run_example
run_example:
     [java] The title of the book is: Hardcore Java.
```

There is no change despite the change in the A_STRING constant. To fix this problem, recompile the ExternalUser class:

```
>ant -Dexample=oreilly/hcj/finalstory/ExternalUser.java compile_example
compile_example:
    [javac] Compiling 1 source file to C:\dev\hcj\bin
```

Running the example once more gives you the output you were seeking when you changed A_STRING in FinalReplacement:

```
>ant -Dexample=oreilly.hcj.finalstory.ExternalUser run_example
run_example:
     [java] The title of the book is: Java Hardcore.
```

The Java compiler doesn't automatically notice the dependency between the String and primitive constants and their users. Some build environments recognize this dependency automatically but don't depend on it.

In fact, whenever you change a public primitive constant, it's a good idea to simply rebuild the whole project, just to be safe. If your code is also being used by other projects, make sure you put the change in your release notes so that others will know to recompile their projects.

During a project for the aerospace industry, my consulting company changed a primitive public constant and forgot to change the release notes. We promptly broke a customer's code and only hours later did we figure out that they needed to rebuild all 400 of their classes to get it working.

Excessive Constants

The overuse of class-scoped constants is another common trap that can clutter up otherwise well-built code. For example, consider a situation in which there are many constants in a mathematically oriented class, as shown in Example 2-4.

Example 2-4. Excessive use of private constants

```
package oreilly.hcj.finalstory;
public class FinalConstants {
```

Example 2-4. Excessive use of private constants (continued)

```
  public class SomeClass {
    /** Contains the constant for the first equation. */
    private static final double K1 = 3.141;

    /** Contains the offset for the first equation. */
    private static final double X1 = 15.0;

    /** Contains the constant for the second equation. */
    private static final double K2 = 1.414;

    /** Contains the offset for the second equation. */
    private static final double X2 = 45.0;

    /** Contains a constant for both equations. */
    private static final double M = 9.3;

    public double equation1(final double inputValue) {
      return (((Math.pow(inputValue, 2.0d) / K1) + X1) / M);
    }

    public double equation2(final double inputValue) {
      return (((Math.pow(inputValue, 3.0d) * K2) + X2) * M);
    }
  }
}
```

Although there is nothing technically wrong with this code, it is a rather nasty mess. If your equations become large, with multiple constants and numerous terms, the situation turns into something more appropriate for a horror movie. To avoid this trap, rewrite this disaster:

```
package oreilly.hcj.finalstory;
public class FinalConstants {

  public class SomeClassBetter {
    /** Contains a constant for both equations. */
    private static final double M = 9.3;

    public double equation1(final double inputValue) {
      final double K = 3.141;
      final double X = 15.0;

      return (((Math.pow(inputValue, 2.0d) / K) + X) / M);
    }

    public double equation2(final double inputValue) {
      final double K = 1.414;
      final double X = 45.0;

      return (((Math.pow(inputValue, 3.0d) * K) + X) * M);
    }
  }
}
```

Although the method-scoped `final` variables may look strange, they are quite legal and useful. For the compiler, the semantics for method-scoped constants are the same as those for class-scoped constants. The compiler will replace primitive and `String` constant variables with the value of the variable at compile time.

In fact, since the constants in Example 2-4 are private, you know that they cannot be accessed outside of the class, so there is no reason to leave them in the scope of the class. In the process of making your code easier to read and understand, you removed the need for those silly `1`s and `2`s in your constant names. Since the constants are used only in those methods, it is appropriate to restrict them to the methods in which they are used.

Take Advantage of Warning Options in Good Development Tools

Good tools can really help you to identify private variables and mistakes associates with them. For example, my IDE is configured to warn me if a private member of a class is not used in that class. Not only does this help me find private constants that should have been declared public, it also helps me clean up code that has been developed for a while and may have accumulated some lint throughout the process. Eclipse, for example, has many of these options; you should turn on all of them to warning level. Using these warnings can save you a lot of debugging time down the road.

To summarize, if you have a `private final static` that is used only in one method, you should probably move it into that method. On the other hand, if the constants are being used in more than one method, you should leave them as class-scoped. The constant `M` was not moved because it was being used by two of the methods.

Final Variables

While we are on the subject of scoped `final` variables, you should keep in mind that these variables don't have to be primitives to be useful. Final variables that are scoped and constructed can be used as a powerful tool to solidify code in methods.

Method-Scoped final Variables

Although `final` variables that appear within methods are a little strange to some people at first, they become quite addictive once you get used to reading them. See Example 2-5.

Example 2-5. Catching mistakes with method-scoped final variables

```
package oreilly.hcj.finalstory;
public class FinalVariables {
```

Example 2-5. Catching mistakes with method-scoped final variables (continued)

```
  public static String someMethod(final String environmentKey) {
    final String key = "env." + environmentKey;
    System.out.println("Key is: " + key);
    return (System.getProperty(key));
  }
}
```

In this class, you build a scoped `final` variable that adds a prefix to the parameter `environmentKey`. In this case, the `final` variable is final only within the *execution scope*, which is different at each execution of the method. Each time the method is entered, the `final` is reconstructed. As soon as it is constructed, it cannot be changed during the scope of the method execution. This allows you to fix a variable in a method for the duration of the method. To see how this works, use the test program in Example 2-6.

Example 2-6. Testing final variables

```
package oreilly.hcj.finalstory;
public class FinalVariables {

  public final static void main(final String[] args) {
    System.out.println("Note how the key variable is changed.");
    someMethod("JAVA_HOME");
    someMethod("ANT_HOME");
  }
}
```

Running this test program results in the following:

```
>ant -Dexample=oreilly.hcj.finalstory.FinalVariables run_example
run_example:
     [java] Note how the key variable is changed.
     [java] Key is: env.JAVA_HOME
     [java] Key is: env.ANT_HOME
```

Each time the method is entered, the passed-in `environmentKey` parameter is appended to the constant prefix and then frozen for the duration of the method call. So why make the variable final? Because once this variable is set in the body of the method, it cannot be changed. Consider what would happen if you made a mistake like the one shown in Example 2-7.

Example 2-7. A coding mistake caught by a final variable

```
package oreilly.hcj.finalstory;
public class FinalVariables {

  public static String someBuggedMethod(final String environmentKey) {
    final String key = "env." + environmentKey;
    System.out.println("Key is: " + key);
```

Example 2-7. A coding mistake caught by a final variable (continued)

```
    key = new String("someValue"); // <= compiler error.
    return (System.getProperty(key));
  }
}
```

When you try to compile this code, it will give the following result:

```
>ant -Dexample=oreilly/hcj/finalstory/FinalVariables.java compile_example
compile_example:
    [javac] Compiling 1 source file to C:\dev\hcj\bin
    [javac] C:\dev\hcj\src\oreilly\hcj\finalstory\FinalVariables.java:53: cannot
assign a value to final variable key
    [javac]     key = new String("someValue"); // <= compiler error.
    [javac]     ^
    [javac] 1 error
```

In the example code, I commented out the compiler error; you will have to uncomment it to run this test. I use a similar procedure for all compiler errors throughout the book.

In this example, the mistake was made of trying to reassign `key` to a different value. This type of mistake simply happens; however, since you are a savvy programmer, and you used the `final` keyword, the compiler tells you that an error was made. This is a great example of trading a logic error for a compiler error.

The technique of fixing variables with `final` is extremely handy for long or complicated methods that have many local variables. When alerted by the compiler, repairing this mistake takes a matter of seconds. If you don't use `final` to fix your variables now, you run the risk of spending long hours to find logic bugs, only to discover that someone reset your variable halfway through the method because of a typo.

Deferred Initialization

If you want to, you can defer the initialization of a `final` variable within the method. Hold onto your hat because Example 2-8 is going to look a little weird at first.

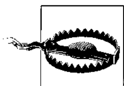
Example 2-8. Method-scoped final variables with deferred initialization

```
package oreilly.hcj.finalstory;
public class FinalVariables {

  public void buildGUIDialog (final String name) {
    final String instanceName;
    if (name == null) {
      // no problem here.
      instanceName = getClass().getName() + hashCode( );
    } else {
```

Example 2-8. Method-scoped final variables with deferred initialization (continued)

```
      // no problem here as well.
      instanceName = getClass().getName( ) + name;
    }

    JDialog dialog = new JDialog( );

    // .. Do a bunch of layout and component building.

    dialog.setTitle(instanceName);

    // .. Do dialog assembly

    instanceName = "hello";  // <= compiler error
  }

}
```

In this case, you declare a `final` variable at the start of the method without giving it a value, since the contents of that variable depend on whether the user passed you `null`. During the `if` statement, check for `null` and then assign the variable appropriately. Once you assign the variable a value, it can't be assigned a value again. However, you could have gone through half the method before assigning the variable its value. This coding technique allows you to make single-shot, assign-and-freeze variables. After assignment, these variables behave like constants for the rest of the method.



Method-scoped `final` variables aren't the same as constants, although they behave like constants at times. The difference is that method-scoped `final` variables are *variable*. Each time the method is entered, their values are changed based on the needs of that particular execution. However, method-scoped constants always have the same values regardless of the circumstances under which the method is run. Also, primitive and `String` method-scoped `final` variables are not substituted at compile time like primitive and `String` method-scoped constants.

In addition to deferring the initialization of method variables, you can defer the initialization of instance-scoped variables and class-scoped variables. Instance-scoped variables must be initialized in a constructor, and class-scoped variables must be initialized in the `static{}` method or you will receive compiler errors stating that the variable has not been initialized.

Chained deferred initialization

One interesting trick you can employ with deferred initialization is to chain the initialization of multiple `final` variables together. For example, consider the following code, which chains instance-scoped `final` variable initialization:

```
package oreilly.hcj.finalstory;
public class ChainingFinals {
  public final String name;
  public final int nameLength = this.name.length;
  // public final String anotherValue = name;  // <== Won't compile

  public ChainingFinals(final String name) {
    this.name = name;
  }
}
```

In this code, the emphasized line will work properly because the `final` variable `name` must be initialized in the constructor to the class. Therefore, the `final` variable `nameLength` can take advantage of `name` when the instance is initialized. However, make sure that you use the `this` keyword in front of the variable name. If you don't, it won't compile.

```
package oreilly.hcj.finalstory;
public class ChainingFinals {
  public final String name;
  public final int nameLength = name.length; // <== Won't compile

  public ChainingFinals(final String name) {
    this.name = name;
  }
}
```

In this slightly revised example, I left off the keyword `this` when using name. As a result, the line won't compile but will instead tell me that `name` is not declared. The compiler requires that you use the `this` reference when chaining `final`s in this manner.

Chaining `final` initialization can be a great tool for precaching data or initializing `final` members that are dependent on other `final` members. It also has the benefit of giving you insight into how object instantiation is managed in the virtual machine. As a result of this process, the constructor to the class is run before the initializers.

Final Parameters

Just when you think it's safe to hit Compile, you can go even further with `final`s. To illustrate, suppose you hire a new developer and, while adding a new feature, he decides to make a little change to the `equation2()` method from Example 2-4. The changes he makes are shown in Example 2-9.

Example 2-9. Danger of nonfinal parameters

```
package oreilly.hcj.finalstory;
public class FinalParameters {

  public double equation2(double inputValue) {
    final double K = 1.414;
    final double X = 45.0;

    double result = (((Math.pow(inputValue, 3.0d) * K) + X) * M);

    double powInputValue = 0;
    if (result > 360) {
      powInputValue = X * Math.sin(result);
    } else {
      inputValue = K * Math.sin(result);
    }

    result = Math.pow(result, powInputValue);
    if (result > 360) {
      result = result / inputValue;
    }

    return result;
  }

}
```

The problem is that the new guy changed the value of the parameter passed in to the method. During the first `if` statement, the developer made one little mistake—he typed `inputValue` instead of `powInputValue`. This caused errors in the subsequent calculations in the method. The user of the function expects certain output and doesn't get it; however, the compiler says that everything in the code is okay. Now it's time to put on another pot of coffee and hope your spouse remembers who you are after you figure out this rather annoying problem.

Little bugs like this are often the most difficult to locate. By Murphy's Law, you can absolutely guarantee that this code will be in the middle of a huge piece of your project, and the error reports won't directly lead you here. What's more, you probably won't notice the impact of the bug until it goes into production and users are screaming for a fix.

You cannot afford to forget that once you write code, the story is not over. People will make changes, additions, and errors in your code. You will have to look through the code and fix everything that was messed up. To prevent this problem from occurring, do the following:

```
package oreilly.hcj.finalstory;
public class FinalParameters {
```

```
public double equation2Better(final double inputValue) {
  final double K = 1.414;
  final double X = 45.0;

  double result = (((Math.pow(inputValue, 3.0d) * K) + X) * M);

  double powInputValue = 0;
  if (result > 360) {
    powInputValue = X * Math.sin(result);
  } else {
    inputValue = K * Math.sin(result);   // <= Compiler error
  }

  result = Math.pow(result, powInputValue);
  if (result > 360) {
    result = result / inputValue;
  }


  return result;
}
```

When you state that the parameter `inputValue` is `final`, the compiler will catch any attempts to assign another value to that parameter and give you an error message with the line number and reason for the problem. The benefit of this little trick (which takes about two seconds to implement) becomes even more obvious when you consider the hypothetical Java bean shown in Example 2-10.

Example 2-10. A bean with a bug

```
public class Person {
  private String name = null;
  public void setName(String name) throws PropertyVetoException {
    String oldName = this.name;
    vetoableChangeSupport.fireVetoableChange("name", oldName, name);
    name = name;
    propertyChangeSupport.firePropertyChange("name", oldName, name);
  }
}
```

On the emphasized line, the programmer forgot to use the prefix `this` on the left-hand side of the assignment; the line should have read:

```
this.name = name;
```

Instead, the assignment does absolutely nothing. In a data model with 212 objects and over 1,000 attributes, bugs like this are extremely difficult to detect. However, if you have a policy to always label method parameters as `final`, such an assignment will cause a compiler error. The `final` parameter version is shown here:

```
public class Person {
  private String name = null;
```

```
    public void setName(final String name) throws PropertyVetoException {
      String oldName = this.name;
      vetoableChangeSupport.fireVetoableChange("name", oldName, name);
      name = name; // <= Compiler error
      propertyChangeSupport.firePropertyChange("name", oldName, name);
    }
  }
```

When compiling this code, the programmer immediately gets a compiler error on the (erroneous) assignment line. The programmer looks back at the code, spots the bug in about two seconds, fixes it, gets another soda, and continues work without even thinking about how much trouble he just avoided. The coding standard here saved him hours of work.

Final Collections

Periodically, while programming, you may want to make constant sets and store them in `final` variables for public use. This desire can lead to all sorts of problems. Consider the code in Example 2-11.

Example 2-11. A collection in a final static member

```
package oreilly.hcj.finalstory;
public class FinalCollections {

  public static class Rainbow {

    public final static Set VALID_COLORS;

    static {
      VALID_COLORS = new HashSet( );
      VALID_COLORS.add(Color.red);
      VALID_COLORS.add(Color.orange);
      VALID_COLORS.add(Color.yellow);
      VALID_COLORS.add(Color.green);
      VALID_COLORS.add(Color.blue);
      VALID_COLORS.add(Color.decode("#4B0082")); // indigo
      VALID_COLORS.add(Color.decode("#8A2BE2")); // violet
    }
  }
}
```

The goal of this code is to declare a class with a `Set` of `final` and `static` `Colors` representing the colors of the rainbow. You want to be able to use this `Set` without concerning yourself with the possibility of accidentally changing it. The problem is that the `Set` isn't `final` at all! Break it with Example 2-12.

Example 2-12. A defect caused by a nonimmutable set

```
package oreilly.hcj.finalstory;
public final static void someMethod( ) {
```

Example 2-12. A defect caused by a nonimmutable set (continued)

```
    Set colors = Rainbow.VALID_COLORS;
    colors.add(Color.black); // <= logic error but allowed by compiler
    System.out.println(colors);
  }
```

The reference to the Set is final, but the Set itself is mutable. In short, your constant variable isn't very constant. The point is that final is *not the same* as immutable.

You can firm up this code in the same way you locked down returned collections from a bean in Chapter 1:

```
package oreilly.hcj.finalstory;
public static class RainbowBetter {

    public final static Set VALID_COLORS;

    static {
      Set temp = new HashSet( );
      temp.add(Color.red);
      temp.add(Color.orange);
      temp.add(Color.yellow);
      temp.add(Color.green);
      temp.add(Color.blue);
      temp.add(Color.decode("#4B0082")); // indigo
      temp.add(Color.decode("#8A2BE2")); // violet
      VALID_COLORS = Collections.unmodifiableSet(temp);
    }
  }
}
```

This version of the class is much better. Your Set of Colors cannot be modified because you have turned it into an immutable object. The reference to the Set is final, and the contents of the collection are locked down.

In the static{} initializer, note how you have to use a temporary set to store the colors. This is because you can set a final variable only once, even in the initializer. If you try to set it more than once or change the variable in the initializer, your compiler will give an error message stating that you cannot change the final variable. Remember that deferred finals are a one-shot deal. Once set (no pun intended), they can't be changed.

Now that you have a strategy to lock down your Set, let's revisit the old logic bug that we discussed in Example 2-12:

```
package oreilly.hcj.finalstory;
public final static void someMethod( ) {
  Set colors = RainbowBetter.VALID_COLORS;
  colors.add(Color.black); // <= exception here
  System.out.println(colors);
}
```

Now that you have the Set locked down, this code results in an exception. Specifically, the method will throw an UnsupportedOperationException whenever the user tries to use any write methods on VALID_COLORS, as it is now immutable. In this case, you haven't been able to trade a logic bug for a compiler bug, but you have been able to trade a logic bug for an exception. Although this trade isn't as good, it's still definitely worthwhile. Always use the java.util.Collections class to get unmodifiable collections and maps when creating final collections and maps.

As far as unmodifiable sets go, the performance hit is negligible. As it turns out, the JDK implements unmodifiable collections in a performance-conscious way. If you look into the JDK source, you will see the static nested classes UnmodifiableSet and UnmodifableCollection. The code in Example 2-13* is pasted directly from the JDK source. All I did was change the spacing to conform to O'Reilly standards and remove the Javadoc for brevity's sake.

Example 2-13. Implementation of unmodifiable collections

```
package oreilly.hcj.finalstory;
public static Collection unmodifiableCollection(Collection c) {
  return new UnmodifiableCollection(c);
}

static class UnmodifiableCollection implements Collection, Serializable {
  // use serialVersionUID from JDK 1.2.2 for interoperability
  private static final long serialVersionUID = 1820017752578914078L;

  Collection c;

  UnmodifiableCollection(Collection c) {
    if (c==null)
      throw new NullPointerException( );
    this.c = c;
  }

  public int size( )     {return c.size( );}
  public boolean isEmpty( )     {return c.isEmpty( );}
  public boolean contains(Object o)   {return c.contains(o);}
  public Object[] toArray( )     {return c.toArray( );}
  public Object[] toArray(Object[] a) {return c.toArray(a);}
  public String toString()           {return c.toString( );}

  public Iterator iterator( ) {
    return new Iterator( ) {
      Iterator i = c.iterator( );

      public boolean hasNext() {return i.hasNext( );}
      public Object next( )  {return i.next( );}
```

* From JDK source Java.util.Collections.

Example 2-13. Implementation of unmodifiable collections (continued)

```
      public void remove( ) {
        throw new UnsupportedOperationException( );
      }
    };
  }

  public boolean add(Object o){
    throw new UnsupportedOperationException( );
  }

  public boolean remove(Object o) {
    throw new UnsupportedOperationException( );
  }

  public boolean containsAll(Collection coll) {
    return c.containsAll(coll);
  }

  public boolean addAll(Collection coll) {
    throw new UnsupportedOperationException( );
  }

  public boolean removeAll(Collection coll) {
    throw new UnsupportedOperationException( );
  }

  public boolean retainAll(Collection coll) {
    throw new UnsupportedOperationException( );
  }

  public void clear( ) {
    throw new UnsupportedOperationException( );
  }
}

public static Set unmodifiableSet(Set s) {
  return new UnmodifiableSet(s);
}
```

When you call `Collections.unmodifiableSet( )`, the class creates a new instance of this static nested class and sets the source collection as the delegate object. As you can see in the example code from the JDK, the class `UnmodifiableSet` implements `java.util.Set` and inherits from `UnmodifiableCollection`, which in turn implements `java.util.Collection`. Together, they form a delegate structure. Any read call to the `UnmodifiableCollection` is forwarded to the delegate collection. However, if the user tries to access a write operation, the class throws an instance of `UnsupportedOperationException`. Therefore, the additional overhead of the `UnmodifiableSet` is only a single method call.

This delegate structure also plugs another big hole: if the `UnmodifiableSet` class inherited from `HashSet`, then the user could just cast the instances back to `HashSet` to gain access to write methods. The delegate structure in the JDK quite elegantly blocks this, ensuring that an `UnmodifiableSet` truly is unmodifiable, even when placed in the hands of a clever and sneaky programmer.

All of the other collection classes work similarly to `UnmodifiableSet`. You should use these heavily in your code. Regrettably, there is no similar way to lock down `final` array objects, so be careful when using them.

Instance-Scoped Variables

Another type of `final` class member that can be very useful is instance-scoped `final` attributes. Consider the code in Example 2-14.

Example 2-14. A creation date property

```
package oreilly.hcj.finalstory;
public class FinalMembers {
  /** Holds the creation date-time of the instance. */
  private Date creationDate =
    Calendar.getInstance(TimeZone.getTimeZone("GMT")).getTime( );

  /**
   * Get the Date-Time when the object was created.
   *
   * @return The creation date of the object.
   */
  public Date getCreationDate( ) {
    return this.creationDate;
  }
}
```

The job of the property `creationDate` is to hold the date and time of the instance's creation. This property represents a read-only property that is set once; after all, an object can be created only once. However, there is a problem with this property: it leaves a massive potential bug lurking in your code. To illustrate this, lets look at another part of the same class in Example 2-15.

Example 2-15. A modification date property

```
package oreilly.hcj.finalstory;
public class FinalMembers {
  /** Holds the modification date-time of the instance. */
  public Date modificationDate = creationDate;

  public void setModificationDate(Date modificationDate) {
    if (modificationDate == null) {
      throw new NullPointerException( );
    }
```

Example 2-15. A modification date property (continued)

```
    this.creationDate = modificationDate;
  }

  public Date getModificationDate( ) {
    return this.modificationDate;
  }
}
```

Here, you have a neat and cryptic little logic bug. If you didn't see the bug instantly, that only reinforces my point. The problem is that the writer of the `setModificationDate( )` method is obviously setting the wrong parameter. Due to a simple typo, instead of setting `modificationDate`, this method sets `creationDate`. No one ever intends to write bugs like this, but it happens. Fortunately, there is a way you can block this problem with a coding standard:

```
package oreilly.hcj.finalstory;
public class FinalMembers {

  private final Date creationDate2 =
    Calendar.getInstance(TimeZone.getTimeZone("GMT")).getTime( );

  public Date getCreationDate2( ) {
    return this.creationDate2;
  }

  public void setModificationDate2(Date modificationDate) {
    if (modificationDate == null) {
      throw new NullPointerException( );
    }
    this.creationDate2 = modificationDate; // <= Compiler error.
  }
}
```

Now that the `creationDate2` is `final`, any attempts to write to it after it is initialized will cause a compiler error.

Instance-scoped `final` variables have a similar morphology to method-scoped `final` variables. They are constructed once at each instantiation of the class and then are frozen for the life of the instance. They are ideal for read-only attributes set only at instantiation.

This technique gives you yet another coding standard that can save your skin. What used to be a difficult-to-find logic bug is magically transformed by the keyword `final` into a minor compile-time bug. Finding the logic bug could potentially take hours of painstaking work with a debugger. The compile-time bug you replaced it with would be fixed by a programmer in only a few seconds. It's unlikely the programmer would even notice the repair!

You can take this concept even further. When writing classes, you can create instance-scoped attributes that are `final` and are not initialized. The class in Example 2-16 shows how this deferred initialization works.

Example 2-16. Deferring final initialization to the constructor

```
package oreilly.hcj.finalstory;
public class FinalMembers {

  /** Holds the creation date-time of the instance. */
  private final Date creationDate3;

  /**
   * Constructor
   *
   * @param creationDate The creation date.
   * @param modificationDate The last modification date.
   */
  public FinalMembers(final Date creationDate,
                      final Date modificationDate) {
    if (modificationDate.compareTo(creationDate) < 0) {
      throw new IllegalArgumentException("modificationDate");
    }
    this.creationDate3 = creationDate;
    // do a bunch of date calculations.
    this.creationDate3 = modificationDate;  // <= compiler error
  }

  /**
   * Second constructor.  Use current date for creation date.
   *
   * @param modificationDate The last modification date.
   */
  public FinalMembers(final Date modificationDate) {
    this.modificationDate = modificationDate;
    // <= compiler error: 'creationDate may not have been initialized'
  }
}
```

In this example, you create a member variable named `creationDate3` and set it as `final`, but don't initialize it. Upon construction, the user passes a date to the class and `creationDate3` is set. As before, once it is set, it cannot be changed. This ability to defer an initialization to a constructor gives you a lot of flexibility and safety in object initialization.

In the first constructor, you accidentally try to set `creationDate3` twice: once with the `creationDate` parameter and once with the `modificationDate` parameter. Obviously, the author of this file mistyped the variable name and meant to type **`this.modificationDate`** in the second instance. However, since you are being vigilant with the `final` keyword, you catch the error at compile time and correct it.

As a bonus, if you forget to set the variable, as in the second constructor in Example 2-16, you get a compiler error telling you that the variable hasn't been initialized:

```
>ant -Dexample=oreilly/hcj/finalstory/FinalMembers.java compile_example
compile_example:
    [javac] Compiling 1 source file to C:\dev\hcj\bin
    [javac] C:\dev\hcj\src\oreilly\hcj\finalstory\FinalMembers.java:69: variable
creationDate3 might not have been initialized
    [javac]   public FinalMembers(final Date modificationDate) {
    [javac]                                                    ^
    [javac] 1 error
```

The reason you get this compiler error is because Java requires that all instance-scoped `final` variables and members must be set before the end of the constructor. In a similar manner, all class-scoped members must be set by the end of the static initialization of the class they are declared in, and all method-scoped `final`s must be set by the end of the method in which they are declared in.

In the case of Example 2-16, your coding standard has given you two layers of security for the price of one. This is a bargain that no programmer could pass up!

Final Classes

A `final` class is a class that does not allow itself to be inherited by another class. Final classes mark endpoints in the inheritance tree.

There are two ways to make a class final. The first is to use the keyword `final` in the class declaration:

```
public final class SomeClass {
  // ...Class contents
}
```

The second way to make a class final is to declare all of its constructors as `private`:

```
public class SomeClass {
  public final static SOME_INSTANCE = new SomeClass(5);

  private SomeClass(final int value) {
  }
}
```

When you give all constructors `private` visibility, you are *implicitly* declaring the class as final; often, this is not the intended result. In fact, it is the omission of the keyword `final` on the class declaration that should alert you to the fact that something is wrong. The class above may very well need to be final, in which case you should always specifically use the keyword `final` in the class declaration. If you don't follow this rule, you could end up causing some devious problems.

To find an example of these problems, you need to look no further than the JDK itself. In the `java.beans` package, you will find a class called `Introspector` (see Chapter 8). Take a look at its single constructor in Example 2-17.*

Example 2-17. The java.beans.Introspector source snippets

```
public class Introspector {
  // ...snip...
  private Introspector(Class beanClass, Class stopClass, int flags)
    throws IntrospectionException {
    // ...snip...
  }
}
```

The constructor for the `Introspector` class is private. I noticed this while studying this class. My goal was to extend the `Introspector` and create a class that is more feature-rich than `Introspector` itself. Unfortunately, since the only constructor of the class is private, it is impossible to extend this class. In the case of the `Introspector` class, there is no reason that the class should be final. The `Introspector` class is a good example of how implicit `final` classes can cause problems.

Most singleton classes shouldn't be declared as `final`. You never know what other features your class's user will dream up. However, since singleton classes need to be protected from external instantiation, you can't make the constructor public. The solution to the problem is a protected constructor, as shown in Example 2-18.

Example 2-18. An extensible singleton

```
package oreilly.hcj.finalstory;
public class Singleton {
  private static final Logger LOGGER = Logger.getLogger(Singleton.class);
  public static Singleton instance = null;
  private final String[] params;

  public static void init(final String[] params) {
    // ...do some initialization...
    instance = new Singleton(params);
  }

  protected Singleton(final String[] params) {
    this.params = params;
    if (LOGGER.isDebugEnabled( )) {
      LOGGER.debug(Arrays.asList(this.params).toString( ));
    }
  }
}
```

* J2SDK source code. © 2002 by Sun Microsystems.

Since this singleton class has a protected constructor instead of a private constructor, you can extend its functionality while protecting it against construction by the user. This will allow you to extend the singleton, as shown in Example 2-19.

Example 2-19. An extension of a singleton

```
package oreilly.hcj.finalstory;
public class ExtendedSingleton extends Singleton {
  private final static int DEFAULT_VALUE = 5;
  private final int value;

  public static void init(final String[] params) {
    instance = new ExtendedSingleton(params, DEFAULT_VALUE);
  }

  public static void init(final String[] params, final int value) {
    instance = new ExtendedSingleton(params, value);
  }

  protected ExtendedSingleton(final String[] params, final int value) {
    super(params);
    this.value = value;
  }
}
```

The protected constructor technique is not limited to classes with instance variables and methods. A protected constructor should be declared for classes that are entirely static in nature. Although these classes have no other instance members, it would be possible, if a bit pointless, to instantiate them. Preventing this instantiation while providing extensibility is definitely an asset to the development process.

There are rare circumstances when a class should be made final. One example is the concept of the constant object class, which we will discuss in Chapter 5. However, making classes *implicitly* final is never a good thing; if you want to make a class final, come right out a declare it with the class declaration.

Final Methods

Final methods are an interesting feature of Java. They allow you to make a class partially final without preventing its inheritance by another class. To make a method final, use the `final` keyword on the declaration, as shown in Example 2-20.

Example 2-20. A final method

```
package oreilly.hcj.finalstory;
public class FinalMethod {
  public final void someMethod( ) {
  }
}
```

This declaration is the antithesis of the `abstract` keyword. Whereas the `abstract` keyword declares that subclasses *must* override the method, the `final` keyword guarantees that the method can *never* be overridden by subclasses. Subclasses can inherit from the `FinalMethod` class and can override any method other than `someMethod( )`.

You should never make a method final unless it *must* be final. When in doubt, leave the `final` keyword off a method. After all, you never know the kinds of variations the users of your class may come up with.

One example of a situation in which making a method final *is* the proper route to take is when a read-only property is used. Example 2-21 shows an example of such a property.

Example 2-21. A final property

```
package oreilly.hcj.finalstory;
public class FinalMethod {
  /** A demo property. */
  private final String name;

  protected FinalMethod(final String name) {
    this.name = name;
  }

  public final String getName( ) {
    return this.name;
  }
}
```

In this example, the `name` property is set at construction time and can never be changed. Also, you have defined that you never want a subclass to hide this property (which it could by declaring its own `name` property if `getName( )` wasn't final). This is a good reason to make a method final. By making `getName( )` a `final` method, you can guarantee that the user of subclasses of this object will always call this method when she executes `getName( )`. In the JDK, the method `getClass( )` in `java.lang.Object` is final for this very reason.

Conditional Compilation

Conditional compilation is a technique in which lines of code are not compiled into the class file based on a particular condition. This can be used to remove tons of debugging code in a production build. To understand the power of conditional compilation, consider Example 2-22, which demonstrates a method that does a complex transaction and logs it using Log4J.

Example 2-22. A method with traces

```
package oreilly.hcj.finalstory;
import org.apache.log4j.Logger;
```

Example 2-22. A method with traces (continued)

```
public class ConditionalCompile {
  private static final Logger LOGGER =
    Logger.getLogger(ConditionalCompile.class);

  public static void someMethod( ) {
    // Do some set up code.
    LOGGER.debug("Set up complete, beginning phases.");
    // do first part.
    LOGGER.debug("phase1 complete");
    // do second part.
    LOGGER.debug("phase2 complete");
    // do third part.
    LOGGER.debug("phase3 complete");
    // do finalization part.
    LOGGER.debug("phase4 complete");
    // Operation Completed
    LOGGER.debug("All phases completed successfully");
  }
}
```

If you assume that there is a lot of code in each phase of the method, the logging shown in this example could be essential to finding business logic errors. However, when you deploy this application in a production environment, you have to go back through the code and eliminate all the logging, or this method will run like a three-legged dog in quicksand. Even if Log4j is set to a higher error level, every logging statement requires a call to another method, a lookup in a configuration table, and so on.

Leaving extensive logging in your program is just not a viable option. I remember going to a new company and working on some code written by biologists. I fired up the GUI, which was one of those "typical slow Java GUIs," and immediately noticed something odd. In my console window, there was so much stuff being written that the word "spam" hardly does it justice. In just initializing the application, the program wrote in the neighborhood of 6,000 lines of tracing information. "No wonder this GUI is slow," I thought. Writing out traces is an extremely CPU-expensive activity, and you should avoid it in a production system whenever possible.

To reduce the overhead, you could try to turn off the logging using a variable:

```
package oreilly.hcj.finalstory;
import org.apache.log4j.Logger;

public class ConditionalCompile {
  private static final Logger LOGGER =
    Logger.getLogger(ConditionalCompile.class);

  private static boolean doLogging = false;
```

```
    public static void someMethodBetter( ) {
      // Do some set up code.
      if (doLogging) {
        LOGGER.debug("Set up complete, beginning phases.");
      }

      // do first part.
      if (doLogging) {
        LOGGER.debug("phase1 complete");
      }

      // do second part.
      if (doLogging) {
        LOGGER.debug("phase2 complete");
      }

      // do third part.
      if (doLogging) {
        LOGGER.debug("phase3 complete");
      }

      // do finalization part.
      if (doLogging) {
        LOGGER.debug("phase4 complete");
      }

      // Operation Completed
      if (doLogging) {
        LOGGER.debug("All phases completed successfully");
      }
    }
  }
```

In this code, you used the `doLogging` variable to shut off the logging information at runtime. Using `if` statements such as this improves your runtime performance and allows you to turn the logging on and off whenever you want. When `doLogging` is `true`, you get all of the logging along with the performance hit. When `doLogging` is `false`, you get no logging, and somewhat less of a performance hit.

This may be a good idea in some cases. However, if your logging is printing out extensive information, this technique will still slow things down. When the variable is `false`, instead of printing out the debugging information, the CPU performs one comparison each time it hits a logging statement.

When using the `isDebugEnabled( )` method in Log4J, the cost is much more than one comparison per `if` statement.

Also, you can't afford to forget that in a method run thousands of times, a few comparisons per method can add up quickly.

You need the ability to compile conditionally, depending on your build paradigm. In development mode, you want all of the debugging code. In deployment mode, you don't. C++ refugees will undoubtedly recognize this ability in the `#ifdef` directive. Although Java doesn't have this directive, the `final` keyword can be used to accomplish similar results:

```
package oreilly.hcj.finalstory;
import org.apache.log4j.Logger;

public class ConditionalCompile {
  private static final Logger LOGGER =
    Logger.getLogger(ConditionalCompile.class);

  private final static boolean doLogging = false;

  public static void someMethodBetter( ) {
    // Do some set up code.
    if (doLogging) {
      LOGGER.debug("Set up complete, beginning phases.");
    }

    // do first part.
    if (doLogging) {
      LOGGER.debug("phase1 complete");
    }

    // do second part.
    if (doLogging) {
      LOGGER.debug("phase2 complete");
    }

    // do third part.
    if (doLogging) {
      LOGGER.debug("phase3 complete");
    }

    // do finalization part.
    if (doLogging) {
      LOGGER.debug("phase4 complete");
    }

    // Operation Completed
    if (doLogging) {
      LOGGER.debug("All phases completed successfully");
    }
  }
}
```

By converting the `doLogging` attribute into a `final` attribute, you have told the compiler that whenever it sees `doLogging`, it should replace it with `false` as per the compile-time

substitution rules from earlier in the chapter. The first pass of the compiler changes the code to something like this:

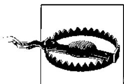
```
package oreilly.hcj.finalstory;
import org.apache.log4j.Logger;

public class ConditionalCompile {
  private static final Logger LOGGER =
    Logger.getLogger(ConditionalCompile.class);

  private static boolean doLogging = false;

  public static void someMethodBetter( ) {
    // Do some set up code.
    if (false) {
      LOGGER.debug("Set up complete, beginning phases.");
    }

    // do first part.
    if (false) {
      LOGGER.debug("phase1 complete");
    }

    // do second part.
    if (false) {
      LOGGER.debug("phase2 complete");
    }

    // do third part.
    if (false) {
      LOGGER.debug("phase3 complete");
    }

    // do finalization part.
    if (false) {
      LOGGER.debug("phase4 complete");
    }

    // Operation Completed
    if (false) {
      LOGGER.debug("All phases completed successfully");
    }
  }
}
```

Once this is done, the compiler takes another look at it and sees that there are unreachable statements in the code. Since you are working with a top-quality compiler, it doesn't like all those unreachable byte codes. So it removes them, and you end up with this:

```
package oreilly.hcj.finalstory;
import org.apache.log4j.Logger;
```

```
public class ConditionalCompile {
  private static final Logger LOGGER =
    Logger.getLogger(ConditionalCompile.class);

  private static boolean doLogging = false;

  public static void someMethodBetter( ) {
    // Do some set up code.
    // do first part.
    // do second part.
    // do third part.
    // do finalization part.
    // Operation Completed
  }
}
```

This is perfect! By setting the value of the doLogging to true or false and recompiling the class, you can turn on and off inclusion of code in your program. When you build for development, you alter the code to set doLogging to true. When you build a production release, set it to false. You now have the best of both worlds.



Remember that if you change a primitive final static logging variable, you'll have to change the class that contains that variable and recompile all the classes that reference it.

Conditional Compilation Variable Location

When implementing conditional compilation, the question of where to put these compilation variables always comes up. There are many potential solutions to the problem, but you need to watch out for a couple of pitfalls.

First of all, putting a variable in the top-level package of your product is probably not a good idea. The problem is that there may be other classes in this package as well. Since all classes will be referencing this package, you could accidentally create circular package dependencies.

Circular dependencies arise when Package A depends on Package B, which in turn depends on Package A. Circular dependencies make it very difficult to separate code into components, and circular references make code fragile by allowing bugs to migrate across the dependencies into other packages.

Your best bet is to create a new package in each of your major products. If you are an employee of a Sun, for example, each major product would be defined by the package directly under com.sun. I like to create a package named _development under the major packages of my clients' products. The leading underscore helps me remember that this package is not part of the product base but is instead a container for things such as these variables. Inside this new package, place a class named DevelopmentMode.

In this class, install one variable for each package in your product. Then you can simply import the class and access the appropriate variable. For the *Hardcore Java* example code, the class would look something like the following.

```
package oreilly.hcj._development;

public final class DevelopmentMode {
  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_bankdata = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_collections = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_constants = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_datamodeling = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_exceptions = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_finalstory = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_immutable = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_nested = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_proxies = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_references = true;

  /** Development mode constant for package oreilly.hcj.bankdata. */
  public static final boolean hcj_review = true;

  private DevelopmentMode( ) {
    assert false: "DevelopmentMode is a Singleton.";
  }
}
```

In this class, you don't specify the `oreilly` package because it would be a bit redundant, since all the code you would be writing for your company would be in that package. However, the variable naming here is a matter of taste. The important fact

is that all packages can be turned from development to deployment mode in one place. Using the constant is easy:

```
package oreilly.hcj.finalstory;

import oreilly.hcj._development.DevelopmentMode;

public class ConditionalCompile {

  public void projectVariables( ) {
    if (DevelopmentMode.hcj_finalstory) {
      // ...do conditional code.
    }
  }
}
```

You merely use the constant in the same way it was used in the last section. This allows you to take advantage of conditional compilation in a maintainable manner. However, be careful not to place any malfunctioning code into the `_development` package (or whatever you named your package), or you may create circular package dependencies. The `_development` package shouldn't depend on anything other than third-party libraries and the JDK itself.

Using final as a Coding Standard

I imagine that many of you never thought you would see an entire chapter written on a single keyword. However, this particular keyword is quite useful. I strongly advise that you spread `final` all over your code. You should use it so much that not seeing it becomes a rare, if not completely unknown, occurrence.

This coding standard may take a little getting used to but it will really pay off in the long term. The best way to get started is to force yourself to use `final` heavily whenever you write or edit code. Also, you should force the junior developers working for you to adopt the use of `final` as a coding standard. They may grumble and balk for a bit, but the coding standard will quickly become so automatic that the developers won't consciously think about it.

Like good Javadoc habits, this one is much easier to implement if you do it while you are coding. Having to go back through old code to implement the standard is a real pain. For this reason, I suggest you make it a coding standard starting today. If you have tools that allow you to edit the code templates, edit them to introduce `final` everywhere you can. Also, when you edit someone else's code, introduce the `final` variable liberally. Doing so helps to guarantee that no one can mess up your code without actually trying to do so.