11.3 Limiting the Amount of Java Code in JSP Pages

You have 25 lines of Java code that you need to invoke. You have two options: (1) put all 25 lines directly in the JSP page, or (2) put the 25 lines of code in a separate Java class, put the Java class in `WEB-INF/classes/`*`directoryMatchingPackageName`*, and use one or two lines of JSP-based Java code to invoke it. Which is better? The second. The second. The second! And all the more so if you have 50, 100, 500, or 1000 lines of code. Here's why:

- **Development.** You generally write regular classes in a Java-oriented environment (e.g., an IDE like JBuilder or Eclipse or a code editor like UltraEdit or emacs). You generally write JSP in an HTML-oriented environment like Dreamweaver. The Java-oriented environment is typically better at balancing parentheses, providing tooltips, checking the syntax, colorizing the code, and so forth.

- **Compilation.** To compile a regular Java class, you press the Build button in your IDE or invoke `javac`. To compile a JSP page, you have to drop it in the right directory, start the server, open a browser, and enter the appropriate URL.

- **Debugging.** We know this never happens to you, but when *we* write Java classes or JSP pages, we occasionally make syntax errors. If there is a syntax error in a regular class definition, the compiler tells you right away and it also tells you what line of code contains the error. If there is a syntax error in a JSP page, the server typically tells you what line *of the servlet* (i.e., the servlet into which the JSP page was translated) contains the error. For tracing output at runtime, with regular classes you can use simple `System.out.println` statements if your IDE provides nothing better. In JSP, you can sometimes use print statements, but where those print statements are displayed varies from server to server.

- **Division of labor.** Many large development teams are composed of some people who are experts in the Java language and others who are experts in HTML but know little or no Java. The more Java code that is directly in the page, the harder it is for the Web developers (the HTML experts) to manipulate it.

- **Testing.** Suppose you want to make a JSP page that outputs random integers between designated 1 and some bound (inclusive). You use `Math.random`, multiply by the range, cast the result to an `int`, and add 1. Hmm, that sounds right. But are you sure? If you do this directly in the JSP page, you have to invoke the page over and over to see if you get all the numbers in the designated range but no numbers outside the range. After hitting the Reload button a few dozen times, you will get tired of testing. But, if you do this in a static method in a regular Java class, you can write a test routine that invokes the method inside a loop (see Listing 11.13), and then you can run hundreds or thousands of test cases with no trouble. For more complicated methods, you can save the output, and, whenever you modify the method, compare the new output to the previously stored results.

- **Reuse.** You put some code in a JSP page. Later, you discover that you need to do the same thing in a different JSP page. What do you do? Cut and paste? Boo! Repeating code in this manner is a cardinal sin because if (when!) you change your approach, you have to change many different pieces of code. Solving the code reuse problem is what object-oriented programming is all about. Don't forget all your good OOP principles just because you are using JSP to simplify the generation of HTML.

"But wait!" you say, "I have an IDE that makes it easier to develop, debug, and and compile JSP pages." OK, good point. There is no hard and fast rule for exactly how much Java code is too much to go directly in the page. But no IDE solves the testing and reuse problems, and your general design strategy should be centered around putting the complex code in regular Java classes and keeping the JSP pages relatively simple.

Core Approach

Limit the amount of Java code that is in JSP pages. At the very least, use helper classes that are invoked from the JSP pages. Once you gain more experience, consider beans, MVC, and custom tags as well.

Almost all experienced developers have seen gross excesses: JSP pages that consist of many lines of Java code followed by tiny snippets of HTML. That is obviously bad: it is harder to develop, compile, debug, divvy up among team members, test, and reuse. A servlet would have been far better. However, some of these developers have overreacted by flatly stating that it is *always* wrong to have *any* Java code directly in the JSP page. Certainly, on some projects it is worth the effort to keep a strict separation between the content and the presentation and to enforce a style where there is no Java syntax in any of the JSP pages. But this is not always necessary (or even beneficial).

A few people go even further by saying that *all* pages in *all* applications should use the Model-View-Controller (MVC) architecture, preferably with the Apache Struts framework. This, in our opinion, is also an overreaction. Yes, MVC (Chapter 15) is a great idea, and we use it all the time on real projects. And, yes, Struts (Volume 2) is a nice framework; we are using it on a large project as the book is going to press. The approaches are great when the situation gets moderately (MVC in general) or highly (Struts) complicated.

But simple situations call for simple solutions. In our opinion, all the approaches of Figure 11-1 have a legitimate place; it depends mostly on the complexity of the application and the size of the development team. Still, be warned: beginners are much more likely to err by making hard-to-manage JSP pages chock-full of Java code than they are to err by using unnecessarily large and elaborate frameworks.

The Importance of Using Packages

Whenever you write Java classes, the class files are deployed in `WEB-INF/classes/`*`directoryMatchingPackageName`* (or inside a JAR file that is placed in `WEB-INF/lib`). This is true regardless of whether the class is a servlet, a regular helper class, a bean, a custom tag handler, or anything else. All code goes in the same place.

With regular servlets, however, it is sometimes reasonable to use the default package, since you can use separate Web applications (see Section 2.11) to avoid name conflicts with servlets from other projects. However, with code called from JSP, you should always use packages. And, since when you write a utility for use from a servlet, you do not know if you will later use it from a JSP page as well, this strategy means that you should *always* use packages for *all* classes used by either servlets or JSP pages.

Core Approach

Put all your classes in packages.

Why? To answer that question, consider the following code. The code may or may not contain a `package` declaration but does not contain `import` statements.

```
...
public class SomeClass {
  public String someMethod(...) {
    SomeHelperClass test = new SomeHelperClass(...);
    String someString = SomeUtilityClass.someStaticMethod(...);
    ...
  }
}
```

Now, the question is, what package will the system think that `SomeHelperClass` and `SomeUtilityClass` are in? The answer is, whatever package `SomeClass` is in. What package is that? Whatever is given in the `package` declaration. OK, fine. Elementary Java syntax. No problem. OK, then, consider the following JSP code:

```
...
<%
  SomeHelperClass test = new SomeHelperClass(...);
  String someString = SomeUtilityClass.someStaticMethod(...);
%>
```

Now, same question: what package will the system think that `SomeHelperClass` and `SomeUtilityClass` are in? Same answer: whatever package the current class (the servlet that the JSP page is translated into) is in. What package is that? Hmm, good question. Nobody knows! The package is not standardized by the JSP spec. So, packageless helper classes, when used in this manner, will only work if the system builds a packageless servlet. But they don't always do that, so JSP code like this example can fail. To make matters worse, servers sometimes *do* build packageless servlets out of JSP pages. For example, most Tomcat versions build packageless servlets for JSP pages that are in the top-level directory of the Web application. The problem is that there is absolutely no standard to guide when they do this and when they don't. It would be far better if the JSP code just shown always failed. Instead, it sometimes works and sometimes fails, depending on the server or even depending on what directory the JSP page is in. Boo!

Be safe, be portable, plan ahead. Always use packages!

[Team LiB]

◀ PREVIOUS NEXT ▶