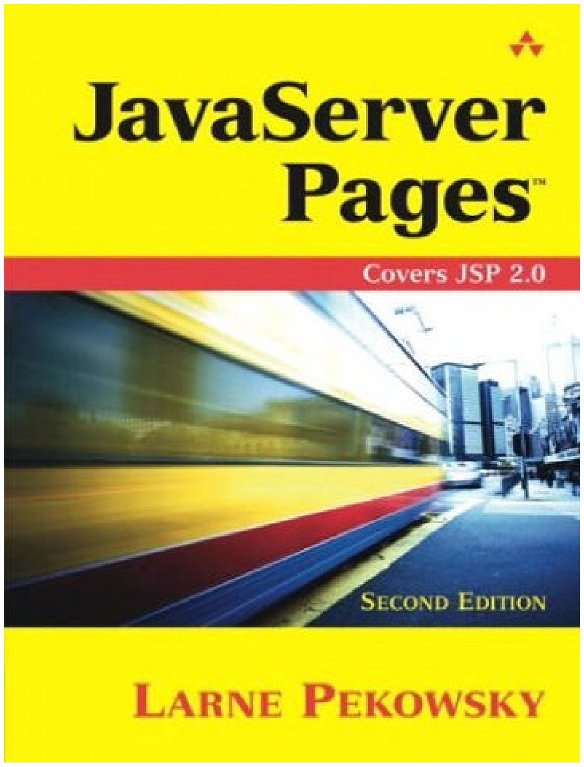


JavaServer Pages™ , Second Edition

By Larne Pekowsky

Publisher : Addison Wesley

Pub Date : August 15, 2003

ISBN : 0-321-15079-1

Pages : 368

Slots : 1

Preface

This is a book about how to use an exciting and powerful technology, JavaServer Pages, (JSP) to create dynamic, interactive Web sites. As the name implies, this technology is based on the Java programming language and inherits many of the language's features and benefits. Most notably, Java makes JSPs available on almost every kind of computer and operating system and certainly all those in common use.

JavaServer Pages are now a mature and stable technology, already in use in thousands of companies. But maturity has certainly not led to stagnation! Recently, a new version of the JSP specification was released, bringing new capabilities and possibilities. In addition, several companion technologies have been developed to augment the fundamental specification. The new specification, as well as the most important of these associated technologies, are all covered in this book. Throughout this book, effort has been made to show the capabilities of all these tools and to discuss how they can best be used.

One of the most important features of JavaServer Pages is how easy they are to use. Anyone who is reasonably comfortable with HTML (Hypertext Markup Language) can learn to write JavaServer Pages by using a few simple tags that may do very sophisticated things behind the scenes, along with small packages of code called JavaBeans. This allows for a very productive working relationship between HTML experts who build pages and Java programmers who build beans and new tags.

Both kinds of developer will find material of interest in this book. Chapter 1 gives a brief history of the Web, setting JSPs in context and clarifying what they are, how they work, and why they work that way. Chapter 2 introduces some of the simpler features of JSPs and shows just how easy the transition from HTML to JSP is.

The next two chapters introduce the two vital technologies that give JSPs their enormous power and flexibility: JavaBeans in Chapter 3 and custom tags in Chapter 4. These tags are presented as page authors will use them: components that hide all the complexities of Java behind simple interfaces that can be combined and used in limitless ways. Chapter 5 uses beans and tags to build a fully functional Web site.

One of the great benefits of JSPs is that they make it possible for pages to interact with complex systems. A very common such system is a database. Chapter 6 introduces database concepts and discusses easy ways in which a page author can access data. Chapter 7 uses this information to expand the utility of the site built in Chapter 5.

XML (Extensible Markup Language) is an increasingly important technology, and JSPs are already well equipped to work with XML. This topic is covered in Chapter 8.

The first eight chapters comprise a logical first half of the book, dealing with the myriad things page authors can do with JSPs without knowing anything about Java. The remainder of the book delves under the hood to explain how all this is accomplished and how Java programmers can extend the capabilities of JSPs. For readers who are not yet familiar with Java, Chapter 9 introduces the language.

Chapter 10 covers the process of creating new beans. Chapter 11 covers a technology, called servlets, that underlies JSPs. This information is then used in Chapter 12 to talk about controllers, Java code that helps pieces of a Web site work together simply and cleanly. Chapter 13 discusses how to use Java to create new tags. Chapter 14 covers a few remaining advanced topics.

Readers who are not interested in programming will get the most out of this book by reading Chapters 1 through 9, which comprise a complete course on how to use JSPs, beans, tags, and related technologies to build just about any Web site imaginable. At that point, such readers may wish to learn a little Java from Chapter 9 and then proceed on through the rest of the book in order to understand better how everything works.

On the other hand, readers who already know Java but who may not be familiar with JSPs, the new features added as part of the latest specification, or related technologies will want to move quickly through Chapter 2 to get a feel for JSP syntax and then go through Chapters 3 and 4 to see how JSPs interface with Java via tags and beans. Programmers may then wish to proceed to Chapter 10 to see how new beans are created, and from there through the second half of the book in order to understand servlets and tags.

Finally, as amazing as it may seem, there is absolutely no cost to developing and deploying JSPs! There is no need to buy a special server or particular hardware or operating system. All the tools needed, and many others, have been released for free by the Apache group. The CD-ROM accompanying this book contains these tools, as well as all the examples from the book. It is my sincere hope that this book, in conjunction with these tools, will help you get the most out of this revolutionary new technology for building exciting, compelling Web sites.

Acknowledgements

This book is very much a group effort, and I am deeply indebted to everyone who helped make it possible.

It has been my great pleasure to work with some of the brightest and most dedicated people in New York at CapitalThinking. Although there are too many to name, I want to thank them all for helping to keep technology fun and exciting enough to write about. Almost every real-world consideration regarding server-side Java that appears in this book came out of projects my colleagues and I worked on.

Many thanks and high praise to everyone at the Apache project behind Tomcat, the standard tag libraries, struts, and so much more. Their decision to make such high-quality tools free and open source deserves a round of applause from every Java and JSP developer.

All the code in this book was developed on a FreeBSD system. I owe a debt of gratitude to everyone behind both the operating system and the Java ports.

I would also like to thank everyone who took the time to read over the manuscript and make suggestions. The final result is profoundly better for their efforts.

This book would be nothing but a collection of unread bits on my hard drive if not for everyone at Addison-Wesley, including Ann Sellers, Jacqui Doucette, Debby Van Dijk, Michael Mullen, Mary O'Brien, and the many others whom I may not have been lucky enough to work with directly.

Finally, I would like to thank the artists who created the music that kept me company while I was writing. Many of their names appear in examples scattered throughout the text.

. . . and I wrote this book for evil gray monkeys who haunt me. . .

Chapter 1. Introduction

Since JavaServer Pages were introduced in June 1999, they have taken the world by storm! Dozens of products are devoted to JSPs, and hundreds of companies are using JSPs to build their Web sites and corporate intranets. The friendly .jsp extension can be seen all over the Web.

The most significant of the many good reasons for this is that it is amazingly easy to develop sophisticated Web sites with JSPs. Anyone who can write HTML can quickly create rich, dynamic, and responsive Web sites that enable users to get the most out of their online time. Through a mechanism called *JavaBeans*, JSPs have made it possible for large teams or individuals working on complex projects to divide the work in such a way as to make each piece simple and manageable, without sacrificing any power. JSPs also provide a great deal of flexibility when generating HTML, through the ability to create HTML-like *custom tags*.

In addition to this fundamental ease of development, high-quality JSP tools are readily available and easy to use. Developers do not need to buy expensive software or commit to a particular operating system in order to use JSPs. The CD-ROM accompanying this book contains everything a JSP author needs to get started, and the tools are powerful enough to serve even a midsized Web site without problems. These free, open-source tools are stable and secure and run on nearly every platform. Of course, high-quality commercial JSP tools are available as well, suitable for serving even the most complex and high-traffic Web sites.

Although JSPs have been useful and powerful since the beginning, this is an especially exciting time to be a JSP developer. The recently released version 2.0 of the JSP specification provides even more features that simplify the process of creating Web sites. In addition, a standard tag library that provides many JSP tags that solve a wide range of common problems has been released. Finally, in the time since they were released, a number of best practices for using JSPs have emerged.

This book covers all the topics: the basic powerful features of the JSP specification, the improvements introduced with version 2.0, as well as the new standard tag library and all the things it does. In addition, this book discusses how best to use these tools, based on real-world experiences.

However, before we get into all the fun, let's take a look back at how the Web has evolved. This will highlight the kinds of problems that Web authors have faced since the beginning. Once this is understood, it will be clear how JSPs solve these problems and make page creation so easy.

1.1 A Brief History of the Web

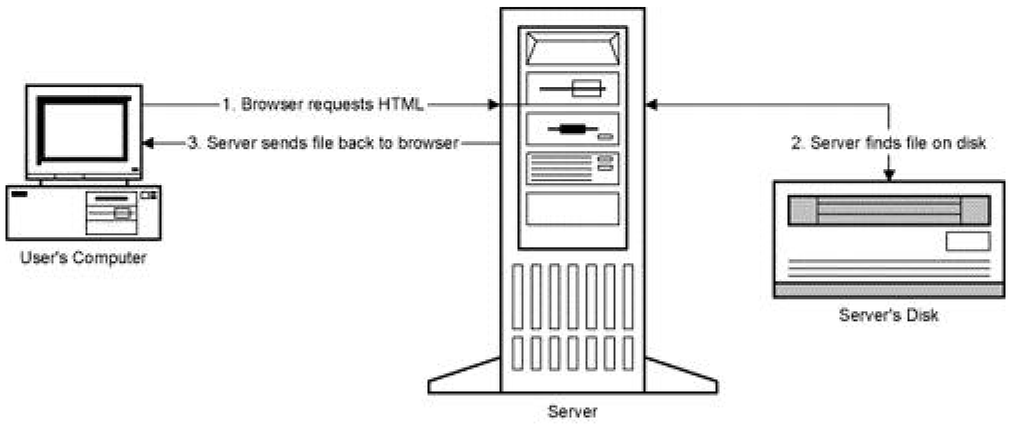
A Web transaction involves two participants: the *browser* and the *server*. As originally conceived, the browser was supposed to be a very simple, lightweight program that would allow users to navigate through data. This data could consist of plain text, HTML, images, and so on, and the browser would render all the data in a way that humans could understand and interact with. Data could be interconnected, and the browser would render references between documents as an image or text that could be clicked or otherwise selected.

Over time, regrettably, rapid development, the race to add new features, and poor adherence to standards have caused browsers to lose the simplicity for which they once strived. This has resulted in a situation best summarized by Internet legend James "Kibo" Parry's description of browsers as "fragile assemblies of bugs, held together with Hello Kitty stickers."

The server is an equally complex program. The server is responsible for finding the data requested by the browser, packaging the data for transmission, and sending it back to the browser.

In the simplest kind of Web transaction, the browser asks for a single document from the server, and the server retrieves this data and sends it back, at which point the browser renders it in an appropriate way for the user. This whole process is shown in Figure 1.1.

Figure 1.1. The relationship between browser and server.


1. Browser requests HTML
3. Server sends file back to browser
2. Server finds file on disk
User's Computer
Server
Server's Disk

This basic activity has many variations. The data being requested by the browser may come from a user typing a URL (universal resource locater) directly, may be in response to the user's clicking a link, or may be automatic, such as an image contained within a page. In each case, the server will receive a properly formatted request for the data, no matter how the request is generated on the client.

How the server fulfills this request also has many variations. In the simplest model, the response may come from a file. Often, there is a simple relationship between URLs and such files. For example, the URL http://somesite.net/lyrics/This_Ascension/forever_shaken.txt might come from a file called C:\Webfiles\This_Ascension\forever_shaken.txt on the computer called somesite.net.

However, just as the server does not care how the request is generated, the client does not care how the response is constructed. Storing data in a file is perfectly adequate for information that never changes, such as the lyrics to a song, or that doesn't change very often, such as a band's tour schedule. When a new date is added to such a schedule, someone can simply edit the file by using a text editor, such as emacs, vi, or notepad, or a full HTML editor, such as Dreamweaver.

However, the file-based model does not work for information that changes very rapidly or that requires input from the user. Following are a few of the ways in which a site might need to take input from a user.

- Many Web sites are dedicated to getting stock quotes, and these sites are used by uncountable numbers of people. If Web servers could do no more than send files around, every site would need to have a separate file for every single stock in existence. The result would be a huge set of files, and it would be difficult, if not impossible, to keep them all updated.
- Although many e-commerce companies have gone out of business, e-commerce itself is thriving, and the electronic shopping cart is now commonplace. This activity would also be completely impossible without the ability to run programs on the server. A site could still put its catalog online as a collection of files, but it takes a *program* to keep track of what items have been ordered, as well as to connect with the shipping and inventory systems to send the merchandise to the user.
- Now that the Web is so big, the only way to find a particular piece of information is with a search engine. Some companies, notably Yahoo, build huge, well-ordered catalogs of Web sites, which could in principle be regular HTML

files. For a user to enter arbitrary text in an entry box and obtain a list of files that contain that word requires a program to look through the files and find ones that match.

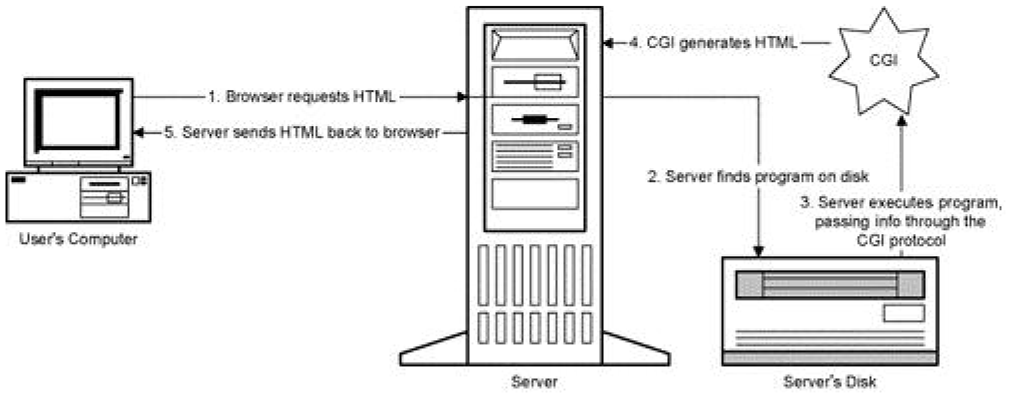
- Users love Web sites where they can vote for their favorite celebrity, manage a virtual stock portfolio, or compete against other users in a match of wits and knowledge.

What all these situations have in common is that the content is now *dynamic;* it needs to change based on time, user input or preferences, or any of hundreds of other attributes.

1.2 Basic Dynamic Page Generation

Fortunately, a solution to the problem of dynamic content has been available since the earliest days of the Web. Rather than reading the data from a file, the server can run a program in order to *generate* the data. This process is illustrated in Figure 1.2.

Figure 1.2. How a server generates dynamic content.


1. Browser requests HTML
5. Server sends HTML back to browser
4. CGI generates HTML
CGI
2. Server finds program on disk
3. Server executes program, passing info through the CGI protocol
User's Computer
Server
Server's Disk

As far as the browser is concerned, this situation is identical to that when the data comes from a file. The browser doesn't care whether the server obtains its data by reading a file or running a program, as long as what it gets back is valid HTML. However, the fact that a program is being run behind the scenes enables the content to be dynamic.
As shown in Figure 1.2, the mechanism by which the server and the HTML-generating program communicate is called CGI (common gateway interface). Over time, the terminology has shifted somewhat, and the programs themselves are now usually referred to as CGIs.

The earliest CGIs were written in a language called C. A sample CGI written in C is shown in Listing 1.1. It generates a Web page that contains the current time and date.

Listing 1.1 A sample CGI in C

```
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv)  {
  time_t now;

  printf("<HTML>\n");
  printf("<BODY>\n");

  time(&now);
  printf("The time is now: %s", ctime(&now));

  printf("</BODY>\n");
  printf("</HTML>\n");

  exit(0);
}
```

This CGI illustrates two important drawbacks to the way the earliest CGIs worked. First, this program is clearly not a good format for HTML developers. Such a CGI could be created only by a programmer, and it looks nothing like the HTML that page developers are used to. In fact, this isn't even a good format for programmers. Each little piece of HTML requires its own piece of C code. This is acceptable, if ugly, for a simple CGI such as this, but for a more realistic example involving lots of tables and colors and JavaScript, this would quickly become overwhelming. It would also make it difficult to fix problems in the HTML or to make changes in order to make the page more attractive or useful.

Owing to the speed at which C programs run, C is still frequently used for CGI portions that are not directly related to the generation of HTML. However, C was rapidly overtaken as the CGI language of choice by another language, called Perl. Perl's main advantage is that it is extremely good at manipulating text, which eliminates some of the overhead involved with C. Listing 1.2 shows the Perl equivalent of the CGI from Listing 1.1.

Listing 1.2 A sample CGI in Perl

```
#!/usr/bin/perl

$now = localtime(time());

print <<"<EOT>"

<HTML>
<BODY>

The time is now: $now

</BODY>
</HTML>

<EOT>
```

This CGI is a little better; all the code is in one place, and all the HTML is in another. They are still in the same file, though, and tightly coupled, so there is no easy way for different people to work on different portions. In addition, it was possible to move all the code to the top only because of the simplicity of this example. In a more complex example, it would likely still be necessary to intermingle the logic with the HTML.

Today, it is possible to write CGIs in almost every language available, even Java. In the end, however, the CGI model itself has a number of intrinsic problems, regardless of any language-specific issues.

The first problem is speed. Just the mere task of having the Web server locate and invoke the CGI program may take up to half a second or so. This may not seem like much, but as any impatient Web surfer can attest, those seconds add up.

Worse, this start-up penalty must be paid for every request, as once it has finished processing one request, a CGI program exits and disappears from the computer's memory. The next time it needs the program, the Web server must be restarted. This is particularly a problem for complex CGIs that need to access a database or other system resource.

These programs need not only to start up fresh each time but also to load up their resources.
The transient nature of CGI programs also limits what they can do, at least without help. The shopping cart is a classic example of this. Clearly, a shopping cart will need to remember which items a user has selected, but it cannot do this alone if it is going to evaporate after each item is added. In more technical terms, CGI programs are *stateless*, meaning that they cannot keep track of any data between requests. Most CGIs get around this problem by saving all necessary information to a database before they exit, but this can be slow and requires that the connection to the database be opened each time the program is started.
Perhaps the most serious problem with CGIs is the way they mesh presentation with logic. As noted, the presentation of a page is expressed as HTML and is typically written by designers and/or expert HTML authors. Program logic, such as what to do on a stock page if the requested ticker symbol does not exist, lives in the program code and is written by programmers. Despite exceptions to this division of labor, both HTML coding and programming are generally such complex and specialized activities that it is rare to find someone skilled at both.
The problem here is that at some point, the HTML must be incorporated into the program because ultimately, the program must generate the output; in order to do this, the program must have all the HTML that will go on the page. This is bad for both the programmers and HTML authors. When the design changes or new pages are designed, the HTML authors cannot change the HTML directly because it is buried in the program. They must present the new designs to the programmers, who must then incorporate the changes into their code without breaking any functionality. The HTML authors must then try out the program to ensure that the HTML that comes out is identical to the HTML that went in, and so on. Hours of company time can be lost this way, and animosity can all too frequently develop between the programming and production groups.

1.3 Solving CGI Problems

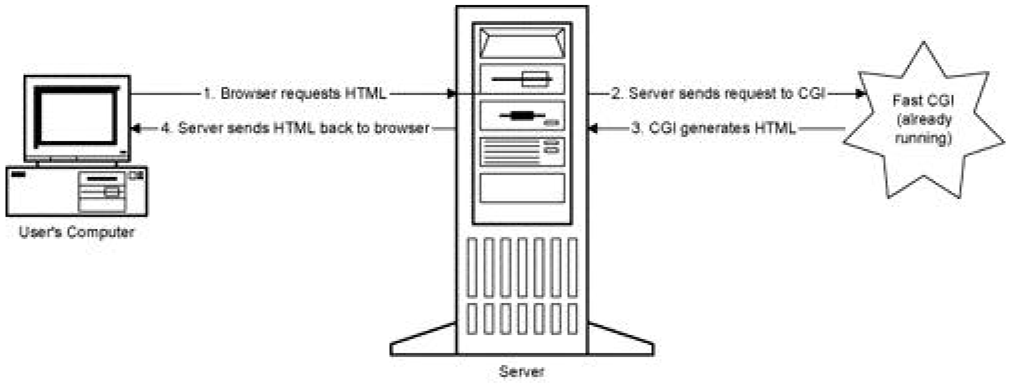
The problems of speed, lack of data persistence, and development have all been addressed in a number of ways.

1.3.1 Speeding up CGI

As noted earlier, one of the biggest problems with CGIs is that a whole new program must be started up for every request. A number of approaches have been taken to eliminate this overhead.

One such approach is called *Fast CGI*. In this model, the CGI remains running instead of restarting each time. The Web server passes the CGI requests to the program over a communication channel called a *socket*, reads the HTML back over the same channel, and then passes the HTML on to the user. This gives the situation illustrated in Figure 1.3.

Figure 1.3. Fast CGI.



1. Browser requests HTML
4. Server sends HTML back to browser
2. Server sends request to CGI
3. CGI generates HTML
Fast CGI (already running)
User's Computer
Server

In addition to solving some of the speed problems, this approach also solves the problem of keeping state. Because the CGI program never exits, it can hold onto information between requests. All that is then needed is a way for the CGI to recognize which user is accessing the page, so it will be able to associate the right data with the right user. Typically, this is accomplished by sending the user a *cookie*, a small marker that the server first sends to the browser and that the browser then includes in any future requests to the same server. Fast CGIs also allow programs to keep connections to a database open, eliminating the need to reopen one for each request. This speeds things up another notch.

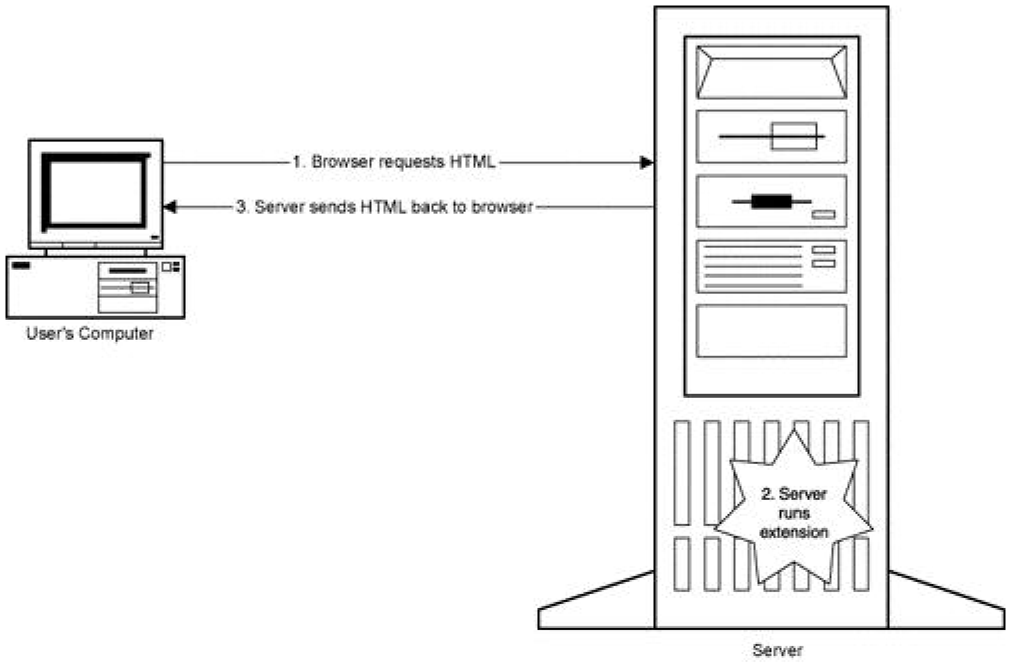
Some problems remain with Fast CGIs, however. Most notably, each CGI program is now a separate process, and each will use up a portion of memory and some of the central processor. This can be alleviated by putting the CGIs on a different computer from the one where the Web server lives, but then the sockets must talk across the network, which will slow things down. This will still be faster than having to start a new program each time, so the situation is not all that bad.

Fast CGIs also introduce a new problem. Updating a regular CGI or adding a new one is a pretty simple matter, simply replacing the old version of the program with the new one. Updating a Fast CGI is a bit more involved, as the old version needs to be shut down and the new one started, and the Web server needs to close down the socket and open a new one. Installing a brand new Fast CGI is even more difficult and will typically require some change to the Web server's configuration describing where the Fast CGI process is running and other information. Most Fast CGI implementations will make this process as automated as possible, but it may still require special system privileges to make all the changes happen.

Fast CGIs can be written in C, Perl, or numerous other languages. Typically, the programs look like regular CGIs, with perhaps some additional code at the beginning. This makes it very easy for programmers to learn how to write Fast CGIs, but it leaves all the same problems regarding the intermingling of program code and HTML.

Since the development of Fast CGIs, a few modifications to address these problems have been made. Most of the popular Web servers can address the problem of too many separate processes by allowing new dynamic functionality to be added to the Web server itself. The idea is that new capabilities can be added to the Web server; when it sees a request that it formerly would have passed off to a CGI, the Web server instead invokes the new routines. This greatly enhances the speed of requests, as everything now stays in one process. This architecture is illustrated in Figure 1.4.

Figure 1.4. Web server extensions.

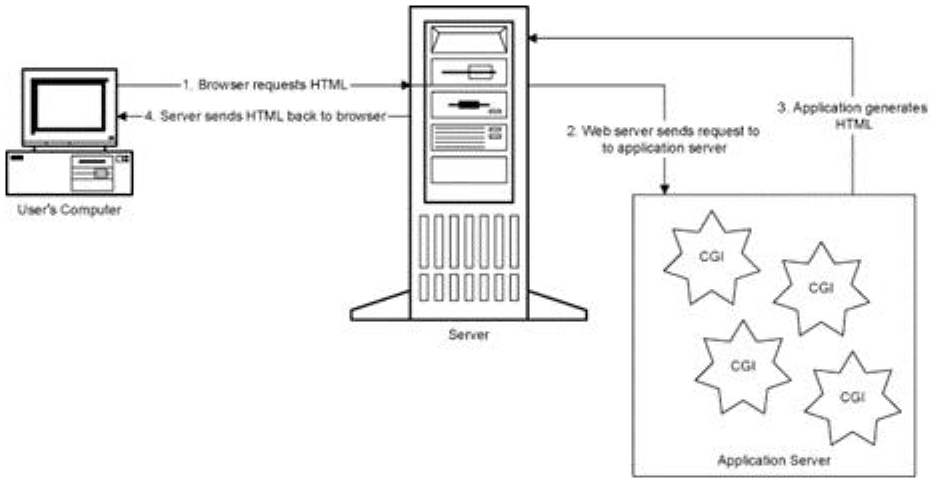


Apache, perhaps the most used and most extensible Web server, took this idea a step further and incorporated the Perl interpreter. This extension, called mod_perl, allows any Perl program, with some minor modifications, to run as part of the Web server. However, extending the Web server this way is not for the faint of heart! It typically requires a lot of knowledge about the inner details of how the Web server works, as well as very careful programming. If an error causes a CGI to exit prematurely, no harm is done, as the next request will simply start a new one. Even Fast CGIs can typically recover after a crash. But if an extension to the Web server crashes, the whole server is likely to go down.

Updating extensions to the Web server is even more difficult than updating a Fast CGI, and only a few system administrators within any given company will typically have the knowledge and permissions to do so. This makes such an extension useful only for adding very fundamental kinds of functions, such as new registration or security features, and not at all well suited to CGI-like applications.

Another approach to improving performance was taken by application servers. Application servers combine the best features of Fast CGIs and server extensions. Like Fast CGIs, an application server runs as a separate process and stays running between requests. This eliminates the cost of starting a new program each time. Like server extensions, application servers are extensible, allowing programmers to add new features as needed. This architecture is illustrated in Figure 1.5.

Figure 1.5. An application server.


1. Browser requests HTML
4. Server sends HTML back to browser
User's Computer
Server
2. Web server sends request to to application server
3. Application generates HTML
CGI
CGI
CGI
CGI
Application Server

In a sense, application servers can be thought of as enhanced Fast CGIs, whereby each extension acts as a separate Fast CGI, but they all sit in the same process. This has numerous benefits. For example, it was mentioned that a Fast CGI can maintain an open connection to a database. Clearly, though, each Fast CGI running individually will need its own connection. An application server can maintain a central "pool" of open connections and hand one off to each component as needed.

Most modern application servers also support some form of load balancing. This allows multiple instances of an application server to run, possibly on different computers. If one application server gets too busy or one of the computers crashes, all requests can go to another server. The users should never even notice the problem.

1.3.2 Separating HTML from Code

In parallel with the developments on the speed front, progress was made in separating HTML from program logic. The motivation behind many of these approaches can be understood by first considering a very simple CGI and building the complexity up from there.

Consider again Listing 1.1, which is just about the simplest possible CGI. It has almost no logic, except for the two lines needed to get the date and time. Mostly, it prints out a bunch of HTML. Therefore, all the HTML could be pulled out of the CGI and put into a separate file. The CGI would open this file, read the contents, and send them back to the server. The HTML author could then edit the file without needing to touch the program code.

Once this mechanism has been built, it is easy to extend it slowly in order to include such things as the date and time. This is a specific instance of a general problem, namely, that frequently a CGI will have to incorporate some data, such as the date or a user's name, into the page.

However, the HTML author need not care where this data comes from. As far as the design of the page is concerned, the important thing is that the date shows up where it belongs. The HTML author could indicate this by using a special tag, perhaps something like `<date/>`. If the CGI is written in Perl, this tag could even be a Perl variable, such as the `$now` variable used in Listing 1.2. Now when it reads the HTML and before sending it to the user, the program can look over the whole file for any occurrences of `<date/>`, do whatever it needs to in order to get the date, replace the tag with the value, and then send the page along.

This idea can be extended by creating more tags to indicate other common data or behaviors. Essentially, the new tags define a new language that both the programmers and the HTML authors agree to speak, and the CGI acts as a translator, converting tags into actions. This sort of system is often called *templating*, as the HTML page with the special tags acts as a template, or blueprint, for all the pages built by the CGI. Unfortunately, this scheme is not quite powerful enough to do all the things dynamic pages need to do. Or rather, by the time it does become sufficiently powerful, the set of tags will be as complicated as any programming language, and we will be back to where we started. Consequently, most systems built around this idea have introduced a few mechanisms that allow this basic scheme to be extended dynamically.

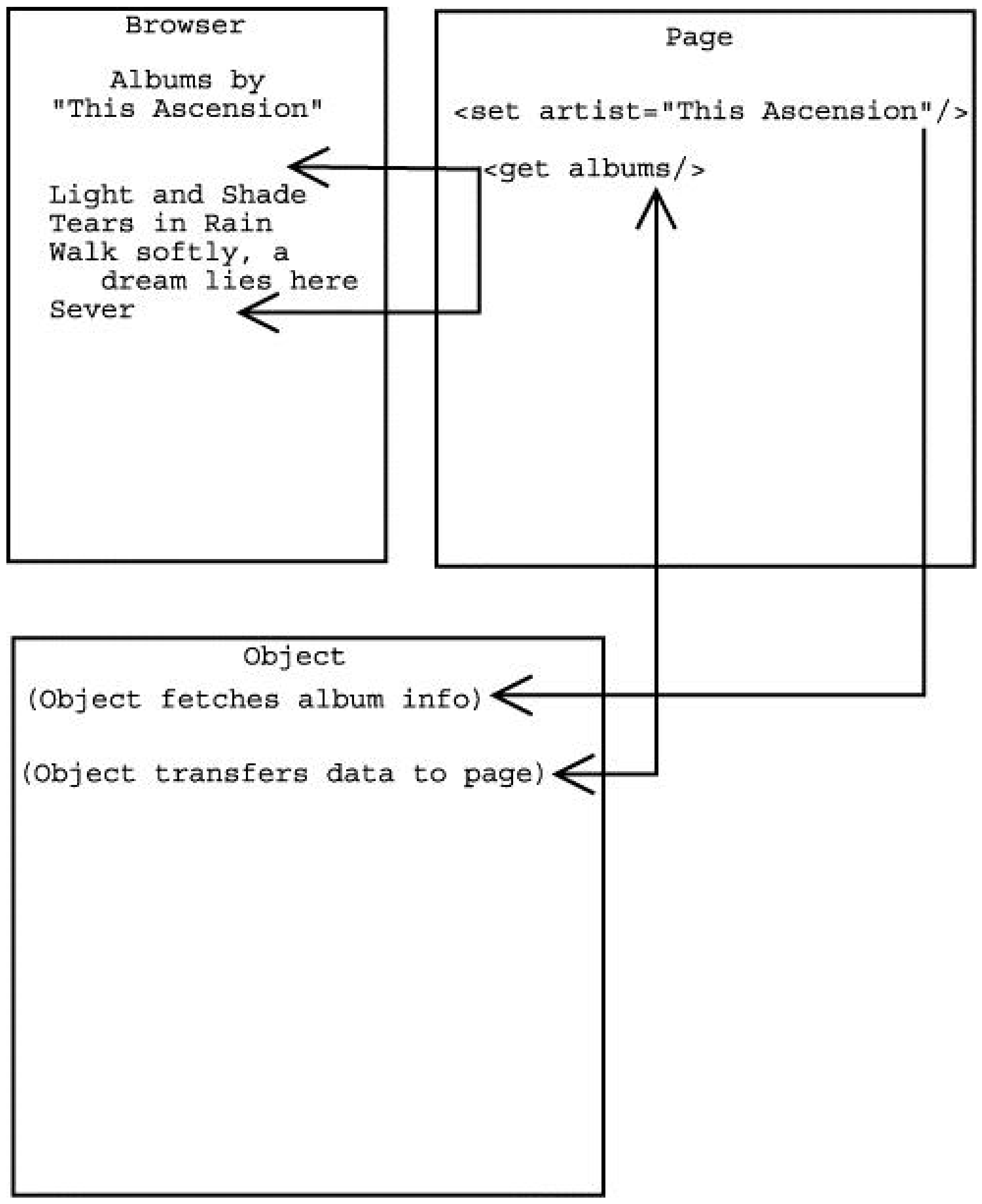
The first is to allow the set of tags to be extensible. An HTML author creating a page for a music catalog, for example, might need a tag to represent the artist's name. In an extensible system, the HTML author could communicate this need to someone in the programming staff, and the two could agree on a new tag to use. The HTML author could then start using this tag while the programmer goes and writes the code that will respond to it.

The system can also be extended in other ways. In addition to creating new tags, the programmers could create new functions, which may be thought of as small magic black boxes, with a slot to drop things into and a ramp where things come out. An HTML author could, metaphorically, drop the name of a musical artist into the slot, and the names of all that artist's albums would come spilling out from the ramp. Then tags could be used to specify where on the page these album names should go.

Best of all, a programmer can extend a templating system like this by providing new *objects*. In programming terms, objects are much like physical objects in the real world. They have properties that can be obtained or changed, numerous complex ways in which they may relate to other objects, and so on. In the previous example, instead of providing a function, the programmer could provide an "artist" object. One property of this object would be a list of albums, and an HTML-like tag could request this list. Each album would also be an object, and some of its properties would be the year it was recorded, the list of track names, and so on. In other words, the artist object would encapsulate all the relevant information in one neat bundle. Again, tags could be created to access the information from this object and the other objects it contains.

This concept is illustrated in Figure 1.6. The box at the top left displays the list of albums as rendered in a browser. Below this is a simplified view of the "artist" object. The box on the upper right shows a hypothetical page that could generate the data for the browser. First is a tag that sets up the "artist" object with the name of the artist in question; then another tag retrieves the set of albums from the object and sends them to the browser. Although highly simplified, the basic concepts are very similar to the way JSPs work.

Figure 1.6. A tag that uses an object.


Browser
Albums by "This Ascension"
Light and Shade
Tears in Rain
Walk softly, a dream lies here
Sever
Page
<set artist="This Ascension"/>
<get albums/>
Object
(Object fetches album info)
(Object transfers data to page)

New tags, functions, and objects can greatly extend the way the templates get data but still do not allow much control over how that information is presented. For example, it is easy to create a tag that means "get all albums by this artist," but it is much more difficult to express the concept "display all the albums where the first track was written by the lead singer." All the necessary information might be present in the object, but combining that information in arbitrary ways may be infeasible.

Consequently, most templating systems allow for some form of scripting. Scripting allows elements of a full programming language usually the language in which the translation CGI is written to be included in the page. All languages have a way to compare two pieces of text and do different things, depending on whether they match. The example in the previous paragraph would require writing some code that checked whether the name of the lead signer matched the name of the author of the first track. This is once again mixing programlike code with HTML, but because most of the hard work is done in the objects or functions, all the HTML authors typically need to know is some relatively simple control structures that do such things as compare two values or loop through a set of values, performing some action on each.

So far in discussing templating, nothing has been said about performance. Not surprisingly, the speed of such a system may be less than ideal. First, the CGI has to be started, then it has to read the template, then it has to look for all the special tags and process them, and so on. All other things being equal, a system like this is likely to be orders of magnitude slower than a CGI written entirely in C.

There is still hope for templating by mixing it with the performance-improvement ideas that were discussed previously. In particular, it is possible to turn the templating CGI into an extension to the Web server, which will eliminate the need to start the CGI up each time a template is needed. This also allows templates to save state information, use shared resources efficiently, and so on. Many such templating systems are alive and well today, from PHP, a well-known hypertext preprocessor that mixes scripting commands with HTML; to WebSQL, a templating system that provides easy access to databases; to osforms, a system built by Object Design to work with its object database.

1.3.3 Servlets and JavaServer Pages

Let's reconsider some of the problems with CGIs. As previously noted, it is time-consuming to have to restart a CGI program for each request. Because the program does not persist between connections, it is difficult to maintain state. Although it is possible to overcome these problems with Fast CGIs or server extensions, these make adding new functionality relatively difficult.

Sun's approach to solving these problems is to use *servlets*. Just as an applet is a small application that extends the functionality of a Web browser, a servlet is a small piece of code that extends the functionality of a server.

Technically, a servlet is an object, written in Java, that is equipped to receive a request and to construct and send back a response. Because servlets are written in Java, they inherit all the language's power and strengths. One of these strengths is speed, as a great deal of effort has been put into making Java perform well as a server language. Equally important, Java is also a cross-platform technology, so it is possible to develop a servlet under Linux, deploy it on NT, and move to Solaris when the site grows, all without needing to change or recompile anything.

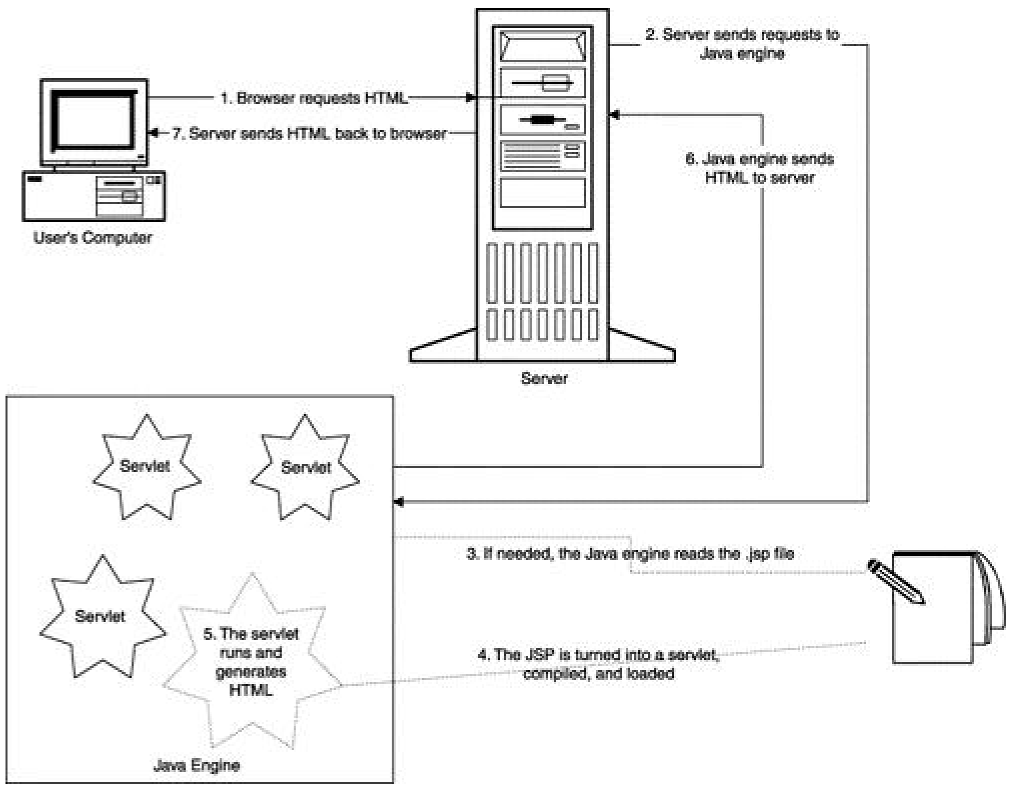
Of special interest to Web developers, Java is an intrinsically dynamic and extensible language. This neatly eliminates the problems inherent in extending a Web server. If it supports Java, the server can be told to load a new servlet with a minimal change to a configuration file and without needing to recompile anything.

The servlet architecture is designed to eliminate the need to reload anything every time a request is made. Instead, the servlet is loaded once, the first time it is needed; after that, it stays active, turning requests into responses as quickly as the Web server can send them. This gives servlets all the speed of Fast CGIs, with none of the hassles. In short, servlets can completely replace CGIs, with no downside.

Servlets also have one additional advantage. Because Java was designed from the ground up as a secure language, servlets can be run in a "secure sandbox," which will prevent them from accessing any potentially sensitive system resources. It is certainly possible to write a CGI that has no security problems, and there are tools to assist in this endeavor. Nonetheless, many security breaches on Web sites happen through insecure CGIs.

The next logical step would be to build a templating system on top of servlets. However, the Art Technology Group (ATG) had an even better idea. Instead of writing a servlet that reads a file, figures out what to do based on special tags, and then does it, why not translate the special tags directly into Java and then compile and run the Java code? This would mean that the first time a page was requested, it would take a little longer to do the conversion, but the second and subsequent requests would be as fast as a servlet. This revolutionary concept, called *page compilation*, was introduced in ATG's application server, called Dynamo. Sun was so impressed by the concept that it licensed the technology for inclusion in its own Java Web Server. The JHTML model is shown in Figure 1.7.

Figure 1.7. The flow of information through a JHTML page.


2. Server sends requests to Java engine
1. Browser requests HTML
7. Server sends HTML back to browser
6. Java engine sends HTML to server
User's Computer
Server
Servlet
Servlet
Servlet
5. The servlet runs and generates HTML
3. If needed, the Java engine reads the .jsp file
4. The JSP is turned into a servlet, compiled, and loaded
Java Engine

No idea is perfect on the first try, and page compilation had problems. Most significantly, there were problems with the set of special tags that ATG had defined, which were somewhat cumbersome, somewhat limited, and completely unlike the tags that other templating systems were using. Over time, Sun has refined these tags to create JavaServer Pages.

JavaServer Pages, or JSPs, combine the best features of the many approaches to dynamic page generation we have discussed. JSPs are implemented in Java, so they are cross platform and inherit all Java's other strengths. Because they are built on top of servlets, JSPs are fast and can be changed easily. They are extensible, and programmers can easily create new objects and functionality using JavaBeans, which page authors can use equally easily.

Sun considers JSPs so important that they are included as a formal part of the Java 2 Enterprise Edition, the standard version of Java for large companies doing complex, performance-critical tasks. Every major vendor of application servers has announced support for JSPs, including ATG, BEA's WebLogic, IBM's Web Sphere, and many more. JSPs have truly become an industry standard.

Best of all, the power of JSPs is not limited to big enterprises or companies that can afford an application server. At the 1999 Java One conference, Sun announced a partnership with the makers of the Apache Web server to provide full support for JSPs under Apache. The resulting project, called Jakarta, has been refined and improved many

times since then and now allows anyone with a computer to develop and deploy JSPs c

ompletely for free.

1.4 Welcome to Java News Today

Throughout this book, we will be following the evolution of a fictional Web site called Java News Today. JNT is a start-up company of Java enthusiasts who want to create a compelling, up-to-the-minute site covering all things Java. Because it wants to attract and maintain an audience, JNT will make its site as dynamic as possible. In addition to updating the content frequently, the site is to have games, polls, search functionality, and other interactive features. JNT also considers it very important to allow users to customize and adjust the site to fit their own needs. The folks at JNT hope that lots of users will make JNT their home page and that no users will move into a home they cannot decorate themselves.

Everyone at JNT will openly admit to being a fan of the Slashdot site, at http://www.slashdot.org, and the Java Lobby, at http://www.javalobby.org. Readers familiar with those sites may notice some similarities in the features that JNT is trying to build. But then, imitation is the sincerest form of flattery.

1.5 Trying the Examples

All the examples in this book have been included on the companion CD-ROM, so readers can see them in action and experiment with changes. The CD-ROM also includes Tomcat, the high-performance JSP engine provided for free by the Apache project, at http://jakarta.apache.org. The version included is 5.0, the first implementation of the JSP 2.0 and servlet 1.3 specifications.

The CD-ROM also includes a number of third-party libraries that provide useful utilities. These libraries include HypersonicSQL, a file-based database written in Java; Jaxen, a set of Java classes for working with XML; and the Canetoad utilities, which provide a number of utilities for working with beans.

Instructions for installing and running the examples can be found in the index.html file on the CD-ROM. Most users can simply double-click this file to get started.

Chapter 2. Simple JSPs

Chapter 1 presented the case for dynamic sites and surveyed a number of techniques for building such sites, focusing on the strengths of JavaServer Pages technology. With these preliminaries out of the way, everything is in place to start creating some pages! This chapter begins by introducing some of the simpler features of JSPs.

It is a time-honored tradition for computer books to start with an example that allows the system being studied to introduce itself. This book is no exception, so without further delay, Listing 2.1 contains our first JSP.

Listing 2.1 A simple JSP

```
<html>
<!-- Our first JavaServer Page -->
<body>
Hello, world!
</body>
</html>
```

This may look like a plain old chunk of HTML, not a very interesting one at that. However, when saved in a file called index.jsp and given to a JSP engine such as Tomcat, this chunk of HTML becomes much more than a static block. In fact, this is a program very similar to the programs in Listings 1.2 and 1.3.

As a program, this file contains a series of instructions that the JSP engine will follow. Written out in English, these instructions are equivalent to the following:[1]

[1] This isn't quite true. For the sake of efficiency, most JSP engines send out contiguous chunks of HTML all at once. However, it is often helpful to think of JSPs as being processed one line at a time.

1. Send the text `<html>` to the user.
2. Send the text `<!-- Our first JavaServer page -->` to the user.
3. Send the text `<body>` to the user.
4. Send the text `Hello, world!` to the user.
5. Send the text `</body>` to the user.
6. Send the text `</html>` to the user.

More technically, a program called the *page compiler* converts the original file into another little Java program, a *servlet* as discussed in Section 1.3. This servlet is what gets run. Servlets are an important technology, and they are covered in more detail in Chapter 11. For the time being, these details are unimportant, and it is perfectly reasonable to think of the JSP file itself as the program.

At this point, three different things are all going by the name index.jsp. One is the original file, sitting in a directory from where it can be edited like any other file. Second is the servlet, which is managed by the JSP engine and is generally not meant to be seen directly. Third is the URL and the corresponding page as seen in a browser. To avoid confusion, the specific meaning will always be indicated when it is not clear from the context.

In this particular case, all the extra work of creating and running a program has not accomplished anything. However, the translation has not been pointless, as it has created a program from HTML. This is why JSP authors generally do not need to do much programming themselves. The JSP engine is quite sophisticated and can turn a few simple tags into very complex code. The servlet that is generated and the environment in which this code runs are also very sophisticated, which removes even more of the programming burden.

This program illustrates two of the basic rules for the JSP programming language:

1. Plain text turns into a command to send that text to the user.
2. Regular HTML tags turn into a command that sends that tag to the user.

2.1 Removing Text from a JSP

All it takes is a small change to Listing 2.1 in order to start exploring some of the things the JSP engine can do. Note that the HTML comment, "Our first JavaServer Page," has turned into a program instruction that sends the comment to the user.

To people building and maintaining pages, these kinds of comments are often useful because they can clarify what a block of otherwise indecipherable HTML is meant to be. However, because it is a regular part of the document, this comment will show up in the "view source" function of a user's browser.

This is typically not a problem, although it is possible for these comments to contain implementation details that might be confidential. Or maybe a page author was having a bad day and used some comment space to rant about his or her boss or relationship or the state of the world. These comments can be quite embarrassing if anyone happens to see them.

So, here is a dilemma. Comments are useful to authors but useless, or worse, for readers. JSP has a solution to this. The preceding HTML comment could be replaced with a JSP comment, like so:

```
<%-- Our first JavaServer Page --%>
```

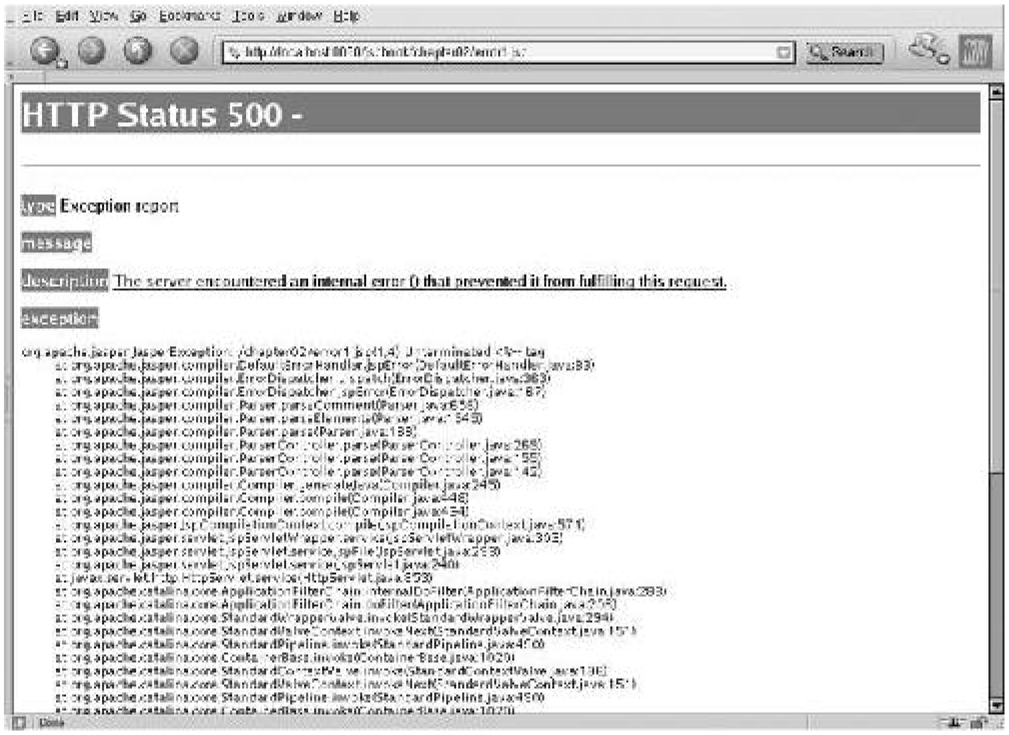
When it sees this tag, the page compiler will recognize it as a comment and will not put it into the servlet that it builds. Hence, this comment will never be sent to the user and will not show up when the user does a view source. Again, this effect is subtle and, frankly, not that exciting. However, it does begin to show that what goes into a JSP file can and will be different from what comes out. Further, this adds a third rule to the JSP programming language: Text enclosed between comment tags (`<%--` and `--%>`) does not turn into an instruction at all but is simply ignored.

2.2 JSP Errors

As smart as the JSP engine is, it is also very literal minded. Like any other program ever written, the best it can manage is to do what we say, which is not always the same thing as to do what we want. When a JSP page does not specify what to do in exactly the right way, the JSP engine sometimes has no alternative but to give up, return an error page, and ask for help.

One common error is leaving out a closing tag. This might happen if a page author tries to close a JSP comment as if it were an HTML comment, as in `<%-- our first JSP -->`. A user who tries to access this page will receive a rather unsettling page giving a great deal of information about the cause and nature of the error. The exact format of this message varies between JSP engines; Tomcat generates a page such as the one shown in Figure 2.1.

Figure 2.1. The Tomcat error page.



HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
org.apache.jasper.JasperException: /chapter02/error1.jsp(1,4) Unterminated <%-- tag
	at org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:83)
	at org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher.java:383)
	at org.apache.jasper.compiler.ErrorDispatcher.jspError(ErrorDispatcher.java:187)
	at org.apache.jasper.compiler.Parser.parseComment(Parser.java:656)
	at org.apache.jasper.compiler.Parser.parseElements(Parser.java:1545)
	at org.apache.jasper.compiler.Parser.parse(Parser.java:133)
	at org.apache.jasper.compiler.ParserController.parse(ParserController.java:268)
	at org.apache.jasper.compiler.ParserController.parse(ParserController.java:155)
	at org.apache.jasper.compiler.ParserController.parse(ParserController.java:142)
	at org.apache.jasper.compiler.Compiler.generateJava(Compiler.java:245)
	at org.apache.jasper.compiler.Compiler.compile(Compiler.java:448)
	at org.apache.jasper.compiler.Compiler.compile(Compiler.java:434)
	at org.apache.jasper.JspCompilationContext.compile(JspCompilationContext.java:571)
	at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:303)
	at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:293)
	at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:240)
	at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
	at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:283)
	at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:255)
	at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:294)
	at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
	at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:450)
	at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:1020)
	at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:135)
	at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
	at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:450)
	at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:1020)
```

The most significant part of this message is the first exception line, which contains the following:

```
/error1.jsp(1.4) Unterminated <%-- tag
```

This line concisely specifies what the problem is, along with the name of the file in which the error occurred. The numbers in parentheses are the line number and the number of characters within the line where the problematic tag starts.

A variation of this problem is even more insidious. Consider the following JSP snippet:

```
<%-- This is a comment -->
Hello, world.
<%-- This is another comment --%>
```

When a browser requests this page, the content will be missing, but no error will be generated. The reason is that this time, the comment tag *is* closed; it just happens to be closed by the second comment tag! This means that the page compiler will consider "Hello, world" as part of the comment and will discard it.

2.3 Including Text in a JSP

Removing text from a page is only slightly useful; it is much more exciting to consider ways in which a JSP can add data to a page. This data may come from any number of places, such as a database, some Java code, or data explicitly provided by the user. Regardless of the source, it will be the JSP's job to inject this data into the page.

The first and simplest place a JSP can get data from is another JSP. Listing 2.2 shows a slightly modified version of Listing 2.1.

Listing 2.2 A JSP that includes another JSP

```
<html>
<body>
Hello again, world!
<jsp:include page="content.jsp"/>
</body>
</html>
```

This turns into a program just as Listing 2.1 did, and once again, all the HTML tags turn into instructions that send those tags to the browser. The `jsp:include` tag turns into an instruction for the JSP engine to run the program called content.jsp, which is shown in Listing 2.3.

Listing 2.3 The included content

```
This is some text from content.jsp.
```

Note that Listing 2.3 contains a complete and valid JSP. A browser could request content.jsp directly, and the response would be the message with no HTML or other tags. However, the intended use is that the browser will request the top-level page, which will render its content, and then the `jsp:include` tag will call content.jsp. Control will then return to the original page, which will send the final closing body and HTML tags. The result, as far as the browser is concerned, will look exactly as if all the HTML was in the original page all along.

Errors to Watch For

Includes can suffer from the same kinds of errors as comments. First, tags can be broken, such as `<jsp:include page="content.jsp">`, which is missing the closing slash. The JSP engine will catch this error and report it as

```
/index.jsp(3,0) Expected "param" tag with "name" and
"value" attributes without the "params" tag.
```

It is also possible to attempt to include a file that does not exist, which will usually happen because of a typo, such as typing headers.jsp instead of header.jsp. In this case, the JSP engine will report "Can't read file headers.jsp," which is easily fixed.

Two files do not make for a very interesting site, but the `jsp:include` tag becomes much more useful when there are many more files. In one common scenario, many files may all want to include some common text. For example, every page on a site might have at the bottom a clever or amusing quote that the site administrators change once a week. If this quote is kept in its own JSP, it is necessary to change only that one file in order to change the whole site.

Conversely, one file may want to include several others. A customized news site might have separate JSPs for top headlines, technology stories, weather, and sports. Many different combinations of content pages could then be easily created by simply choosing which of these pieces to include. In this sense, using JSPs is a lot like building with LEGOS: Whole sites can be constructed by combining simple blocks in different ways. Both of these techniques will be used extensively later in this chapter.

Closely related to the `jsp:include` tag is another, called `jsp:forward`. Whereas `jsp:include` includes the contents of one page within another, `jsp:forward` simply sends the user to another page. This requires that the page issuing the `jsp:forward` have no text other than blank lines either before or after the tag. This may not seem useful yet but later will allow pages to make decisions about what content should be displayed.

2.4 The Phases of a JSP

Each JSP goes through two distinct phases. The first phase, when the translator turns the file into a servlet, is called *translation time*. This translation from JSP to servlet happens whenever the JSP file is modified.

The second phase, when the resulting servlet is run to generate the page, is called *request time*. This happens whenever a browser requests the page. Different things happen in each phase, and the distinction is important.

The handling of JSP comment tags happens at translation time. The translator simply omits any text within comment tags, so the servlet will not even need to deal with it.

Conversely, the `jsp:include` tag is handled at request time. For every request that comes in for the index.jsp URL, the content.jsp servlet will be run when the `jsp:include` tag is encountered. In principle, this could have been done at translation time; the chunk of HTML from the content.jsp file could have been dropped right into the index.jsp servlet. This would be much less powerful, however. By processing includes at request time, the contents of the included file can change independently of the main file. Another advantage to processing includes at request time is that doing so ensures that no JSP is fundamentally special or different. Every JSP is a little program that can be run by requesting the corresponding URL through a browser; there is no distinction between "top-level" and "included" JSPs. As mentioned, this means that a browser can directly request an included file, such as content.jsp, which is often useful when testing page components.

This also guarantees that all JSP elements will work the same way in all pages. This means that JSP comments will be stripped out of included files and that the `jsp:include` tag will work inside included files! Files can include files that can include files, and so on, potentially to infinity.

For the sake of completeness, it is worth mentioning the translation-time version of the `jsp:include` tag, called the include *directive*. This tag looks like this:

```
<%@include file="contents.jsp" %>
```

This tag is called a directive because it directs the page compiler to take some action. In this case, when it sees the directive, the page compiler will embed the contents of the contents.jsp file directly into the servlet that it is building for index.jsp. Subsequently, if the contents.jsp file is edited, index.jsp will not change. Browsers will continue to get the old message until the page compiler is forced to rebuild the index.jsp servlet.

Note that the include directive specifies what is to be included with `file=`, whereas the `jsp:include` tag specifies `page=`. This nicely encapsulates the differences between the two: The directive includes a file as it is building the servlet at request time, and the tag includes another page at request time.

In some obscure instances, the include directive and the `jsp:include` tag are not equivalent. For the most part, though, the two are functionally identical, and as the `jsp:include` tag is more convenient, it will be used in preference to the directive throughout this book.

2.5 Creating Custom Error Pages

Many other directives are available for issuing instructions to the page translator. One of the most useful is called, not surprisingly, `page`.
This directive takes a number of forms, many of which will be encountered throughout this book as needed. One immediately useful option allows a JSP to specify where the user should be directed in the event of an error. Tomcat's default error page, as already shown in Figure 2.1, can be useful to developers but more than a little scary to end users. Ideally, users will never see an error page, but a good site plans for all contingencies and so should include an error page that fits visually with the rest of the site and that allows the user to continue what he or she was doing, to whatever extent possible.
Using a custom error page involves two steps. The first is to create the page, which can be another JSP. However, error pages need to be treated differently from regular pages, and so the page translator must be notified that a JSP will be used as an error page by use of the following page directive at the very top of the file:

```
<%@ page isErrorPage="true" %>
```

Once such an error page has been created and properly identified, it can be used in one of two ways. The first is globally, by telling Tomcat which error page to use for each type of error. This is done through a configuration file and is shown in Appendix B.
In addition to the global approach, each JSP can specify its own error page through another variation of the page directive, as in

```
<%@ page errorPage="error_page_url" %>
```

Here, *`error_page_url`* is the URL of the error page, relative to the current page. Both of these versions of the page directive will be demonstrated in the next section.

2.6 Java News Today

Java News Today, our fictional news site, is ready to begin constructing its site. JNT has decided to start with the new home page and for the moment will not worry about the dynamic elements. The first version is shown in Listing 2.4.

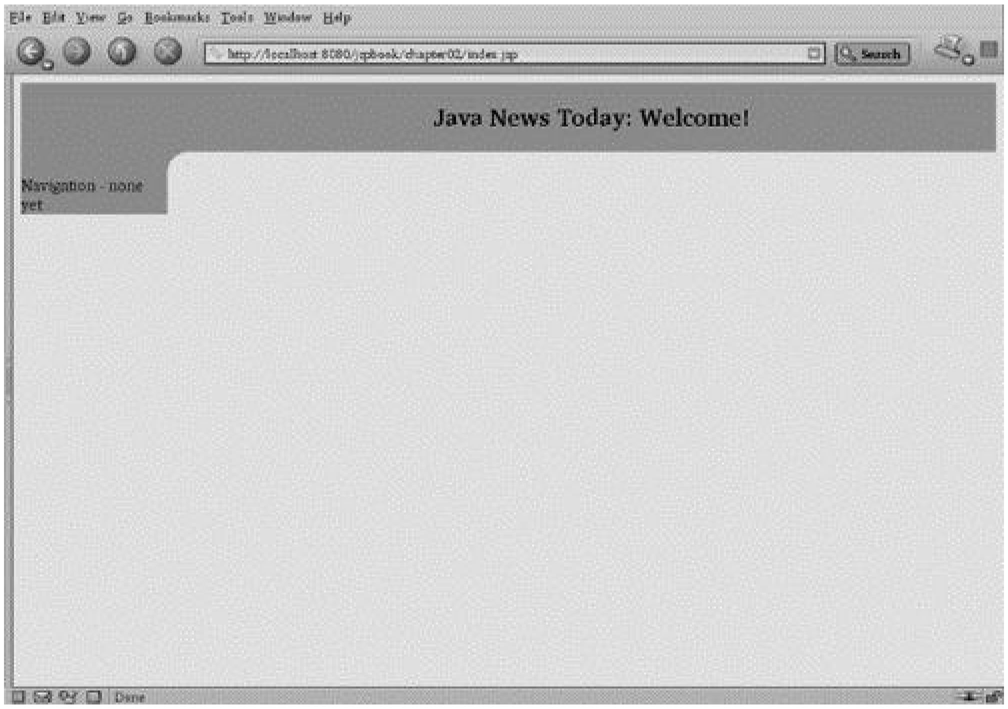
Listing 2.4 The JNT index page

```
<%@ page errorPage="error.jsp" %>

<html>
<head>
```

```
  <link rel="StyleSheet"
        href="style.jsp"
        TYPE="text/css"
        media="screen">
  <title>Java News Today: Welcome!</title>
</head>

<body>

<table width="100%"
       border="0"
       cellspacing="0"
       cellpadding="0">

  <tr>
    <td width="15%" class="borders">
      <%-- Big Empty Corner --%>
    </td>

    <td width="20" class="borders">
      <%-- Little buffer for the curvy bit --%>
      <img src="1x1.gif">
    </td>

    <td class="borders">
      <%-- start header --%>
      <center><h2>Java News Today: Welcome!</h2></center>
      <%-- end header --%>
    </td>
  </tr>

  <tr>
    <td width="15%" class="borders"></td>
    <td width="20" height="20">
      <%-- The curvy bit --%>
      <img src="corner20x20.gif">
    </td>
    <td><img src="1x1.gif"></td>
  </tr>

  <tr>
    <td width="15%" class="borders" valign="top">
      <%-- start navigation --%>
      Navigation - none yet
      <%-- end navigation --%>
    </td>
    <td width="20"><img src="1x1.gif"></td>

    <td valign="top" align="left">
      <%-- All content goes here! --%>
    </td>
  </tr>

</table>
</body>
</html>
```

JNT saves the contents of Listing 2.4 into a file called index.jsp, points its browsers at the corresponding URL, and as expected sees the page in Figure 2.2.

Figure 2.2. The JNT home page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter02/index.jsp
Search
Java News Today: Welcome!
Navigation - none yet
Done

Conceptually, this page consists of four major elements: the header, the navigation, the content, and the HTML that connects it all together. Note the use of JSP comments to delineate these sections. It is generally wise to mark off major functional areas of a page, but as the end user is probably not interested in these fences, they might as well be JSP comments instead of HTML comments.

Different pages will have different content, but it is reasonable to expect that the header and navigation will be repeated all over the site, although doing so can be a major headache. The author of each new page will have to remember to put these pieces in and will have to worry about getting everything right. Worst of all, sooner or later will come the hateful day when a new section is introduced and everyone has to go back and reedit all their pages.

The solution for this nightmare scenario is, of course, the `jsp:include` tag. The header and footer will be split into separate files. The header.jsp file will contain everything in Listing 2.4 from `<%-- start header --%>` to `<%-- end header --%>`. Likewise, navigation.jsp will hold everything from `<%-- start navigation --%>` to `<%-- end navigation --%>`.

This technique of pulling out common chunks of HTML and putting them in separate files is called templating, although the use of the word here is slightly different from that in Chapter 1. Here, a template is merely an HTML page with some "holes" where text should be, along with a way to indicate where this text should be found. The advantage is that many pages can have the same spaces, and all these holes can be filled from the same place. This makes it possible to keep the header in exactly one file and let each page have a space that should be filled by this file.

So far, this might seem like a rather goofy thing to do, as the header and navigation are currently so small. However, rest assured that they will grow in subsequent chapters, and the advantages of removing them from the main page will become increasingly obvious. One such advantage is that it now becomes easy to create alternative versions of the home page. Because all the HTML elements remain in the main file, a simplified version suitable for text-only browsers, such as Lynx, could be created with a page like that shown in Listing 2.5.

Listing 2.5 A version of the home page without tables

```
<%@ page errorPage="error.jsp" %>

<html>

<head>
  <title>Java News Today: Welcome!</title>
</head>

<body>

<jsp:include page="header.jsp"/>
<hr>

<jsp:include page="navigation.jsp"/>
<hr>

<%-- The contents of each page go here --%>
</body>

</html>
```

Note that there are a lot of HTML elements. It would be a real pain to have to rewrite them for every page on the site. Fortunately, the `jsp:include` tag can once again come to the rescue. The idea is that everything above the content can be placed in one file and everything below in another, and then each new page can be created as simply as writing the content and including two files. This final version of the JNT home page is shown in Listing 2.6.

Listing 2.6 The final version of the index page

```
<%@ page errorPage="error.jsp" %>

<jsp:include page="top.jsp"/>

Content goes here!

<jsp:include page="bottom.jsp"/>
```

Now that's what a JSP should look like! For the sake of completeness, top.jsp is shown in Listing 2.7 and bottom.jsp in Listing 2.8.

Listing 2.7 The top part of the page

```
<%@ page errorPage="error.jsp" %>

<html>
<head>
  <link rel="StyleSheet"
       href="style.jsp"
       TYPE="text/css"
       media="screen">
  <title>Java News Today: Welcome!</title>
</head>

<body>

<table width="100%"
      border="0"
      cellspacing="0"
      cellpadding="0">

  <tr>
    <td width="15%" class="borders">
```

```
      <%-- Big Empty Corner --%>
    </td>

    <td width="20" class="borders">
      <%-- Little buffer for the curvy bit --%>
      <img src="1x1.gif">
    </td>

    <td class="borders">
      <%-- start header --%>
      <jsp:include page="header.jsp"/>
      <%-- end header --%>
    </td>
  </tr>

  <tr>
    <td width="15%" class="borders"></td>
    <td width="20" height="20">
      <%-- The curvy bit --%>
      <img src="corner20x20.gif">
    </td>
    <td><img src="1x1.gif"></td>
  </tr>

  <tr>
    <td width="15%" class="borders" valign="top">
      <%-- start navigation --%>
      <jsp:include page="navigation.jsp"/>
      <%-- end navigation --%>
    </td>
    <td width="20"><img src="1x1.gif"></td>

    <td valign="top" align="left">
```

Listing 2.8 The bottom of the page

```
    </td>
  </tr>

</table>
</body>
</html>
```

Top.jsp includes header.jsp and navigation.jsp, but it is perfectly OK for an included JSP to include yet other ones.

One problem with the way this page has been split up is that the title tag is currently hard-coded in top.jsp, and the page banner is likewise hard-coded in header.jsp. This is quite easy to fix but requires a tag that has not been introduced yet and so will have to wait until Chapter 4.

It is also worth pointing out that this page uses a style sheet to set various visual attributes of the page. This in itself is not unusual; most Web sites do the same thing. However, note that the style sheet being used is not a regular .css file but another JSP! This will turn out to be very important when tackling customization, as it will allow the style sheet itself to be generated dynamically! For the moment, this file is static and pretty small but is shown in Listing 2.9 for those interested.

Listing 2.9 The style sheet

```
TABLE.form { border-style: groove;
            border-color: #004400; }

TD.label { border-style: solid;
          border-width: 1px;
          border-color: #00aa00;
          background: #00AA00;
          color: #000000;
          padding-right: 5px }

TD.form { border-style: solid;
         border-width: 1px;
         border-color: #004400;}

TD.borders { background: #66ffff; }
```

```
DIV.bordered { border-style: groove;
             border-color: #004400; }
```

All versions of the index page now use the page directive to link to a custom error page, and this error page is shown in Listing 2.10.

Listing 2.10 The error page

```
<%@ page isErrorPage="true" %>

<html>

<head>
  <title>Java News Today: Error</title>
</head>

<body>

We're sorry, but an error occurred while building
your page. We will try to fix this problem shortly;
in the meantime please return to the
<a href="index.jsp">JNT home page</a>

</body>

</html>
```

Apart from the page directive at the top, this looks just like any other JSP. This is only a very bare-bones example; a more realistic error page would include the same header and navigation elements that the index page used, so as to make it look more like a regular page on the site. In addition, a sophisticated error page could do something like notify the site administrator that an error had occurred. Unfortunately, doing something like this requires many features that have not yet been discussed, and so it will have to wait until Chapter 14.

2.7 Summary and Conclusions

This chapter started down the exciting road of writing JSPs. The two phases of a JSP's existence were discussed: the translation phase, in which JSP code is turned into a Java servlet, and the request phase, in which the servlet is run to produce HTML.
So far, we have not even begun to scratch the surface of what JSPs can do. The next chapter starts looking at some of the things that can be done at request time, which is when the doorway to dynamic content really opens.

2.8 Tags Learned in This Chapter

`<%-- --%>` JSP comment
Parameters: None
Body: None
Anything within the comment tag is removed by the page compiler at translation time.

`jsp:include` Include tag
Parameters:
  `page`: Specifies the page to be included
Body: None
The named page is evaluated and the results included into the output at request time.

`jsp:forward` Forward tag
Parameters:
  `page`: Specifies the page to be included
Body: None
Control passes to the named page. The calling page must not contain any text before or after the tag.

`@include` Include directive
Parameters:
  `file`: Specifies the file to be included
Body: None
The named file is included into the servlet built by the page compiler at translation time.

Chapter 3. Using Beans

The examples in Chapter 2 were quite simple, although they did illustrate a few important points. Notably, these examples demonstrated how a JSP file is converted into a little program and how this program can then perform certain actions when it is run. What is needed now is a mechanism for pages to react to user input and other data.

A page may need to react to such information in any of several ways. For example, a page that displays a stock portfolio typically uses green text to show the stocks that have increased in value and uses red to show those that have decreased in value. Here the page needs to display a value and also change an aspect of its appearance, based on the value.

A page might also want to use a value in a complex calculation or process. In the online portfolio, once a "buy" order is placed, the current status of the user's account must be checked to ensure that it contains sufficient funds to cover the purchase and any transaction fees. If this condition is met, the order must be handled, which entails a number of processes behind the scenes. In either case, the result of the request must then be reported back to the user.

The first kind of reaction, in which the presentation of the page is altered, can be handled either by putting Java code within the page itself or by using special JSP tags. But before diving into the second type of reaction, it is worth discussing the more general problem of software engineering.

3.1 Splitting Big Tasks into Manageable Pieces

As with any kind of engineering, software engineering is inherently difficult. No one person can build a bridge, plane, or building. The same is true of software beyond fairly small projects. Often a project is so big and complex that it simply cannot all fit into a single human brain.

For a large project to be manageable, it must be split into smaller pieces. This is true when many members of a team are working on the same project, so that each knows what part of the whole he or she will create. It is even true when a project is being undertaken by a single person. Splitting big problems into smaller ones allows the developer to concentrate on one thing at a time. Usually, the smaller pieces are easier to design, build, and test; once built, the individual components can be updated or changed without worrying about how that change will affect the rest of the system.

A difficult task can be partitioned in many ways. Java itself provides a natural unit of work in the form of *classes*, which will be discussed further in Chapter 9. Although these classes can help define the way work is done, the more fundamental question remains: what each class should do and which classes each developer should build.

One approach to this question advocates dividing the work into major functional units. This makes perfect sense, as it is the way most physical engineering is done. When building a car, one team is likely to be responsible for the engine, another for the electronics that controls everything, and a third for the exterior. These are natural ways to divide the work, as each piece is somewhat independent of the others and requires very different skills.

There is no set recipe for the way in which a software project should be divided, but over time, an important pattern that advocates dividing the project into three major pieces has emerged. The first piece will be responsible for modeling the problem to be solved and so is called the *model*. This model might be a virtual shopping cart for an online catalog, a database representing a CD collection, or a set of equations representing a complex scientific simulation. In each case, the model contains all the information about how the data is internally stored and the operations that may be performed on that data.

The second piece is responsible for allowing users to interact with this data. This could be a desktop application written in Java using the Swing API, or it could be an applet or even a program that controls a huge electronic billboard. The code for this piece contains everything needed in order to navigate through the data, display the values, modify the display as needed, and, possibly, allow the user to make changes to the model. Ideally, the presentation should also be aesthetically pleasing and intuitive to use. This piece is known as the *view*.

Finally, the third piece acts to mediate between the first two. Although it allows the user to request particular data, the view will not itself load that data into the model. In addition, some data may be restricted to certain users. Such security information should not reside in either the model or the view. So, the third piece, called the *controller*, is responsible for controlling the model, based on instructions from the view, and may also tell the view to hide certain information, based on data in the model.

In addition to identifying these three pieces, it is important to ensure that they all fit together and can interoperate. This is done by providing well-defined interfaces between each pair of components. The view will know how to get data from the model, the controller will know how to configure the view and the model, and so on.

Splitting the work in this way is known as the model/view/controller paradigm, and it is very powerful. Not surprisingly, JSPs will act as the view. The controller piece is not important for simple pages, and so discussion of it is deferred until Chapter 12.

That leaves the model. In the Java world, the model is usually composed of reusable software components called *JavaBeans*. These beans provide the logic behind the stock-purchasing system discussed earlier.

The importance of beans cannot be overstated. Besides being a fundamental Java specification in their own right, beans are *the* means for making JavaServer Pages do interesting and exciting things. Using beans properly allows sites to be built and maintained much more easily than they would otherwise. On the other hand, confusion about what belongs on a page versus what belongs in a bean can be a recipe for disaster. With that in mind, it's time to find out what these bean things actually are.

3.2 Defining Beans

For the moment, forget about Java and consider a real bean. A bean has certain characteristics, or *properties*, such as color, size, shape, species, and so on. Not all beans have the same properties. Coffee beans have a "grams of caffeine" property, which lima beans do not have. [1]

[1] Pedants may point out that lima beans do have a "grams of caffeine" property; it's just that the value is always zero! Such people are welcome to try brewing a cup of decaf from lima beans.

It is always possible to determine the value of these properties, although this sometimes requires a careful chemical analysis. However, imagine if a person could discover the value of one of these properties by asking the bean. Further, imagine if beans could change their properties at will; someone could order a bean to set its size to 3 feet or its color to blue, and the bean would suddenly change. People could then instantly decaffeinate their coffee beans or double the amount of caffeine for those lengthy early morning meetings.

Finally, consider a "bean microscope" that could automatically list all a bean's properties.

Note that none of these activities finding the current state of a property, changing a property, or discovering which properties are available would require cutting the bean open or studying its metabolism or anything similarly complex.

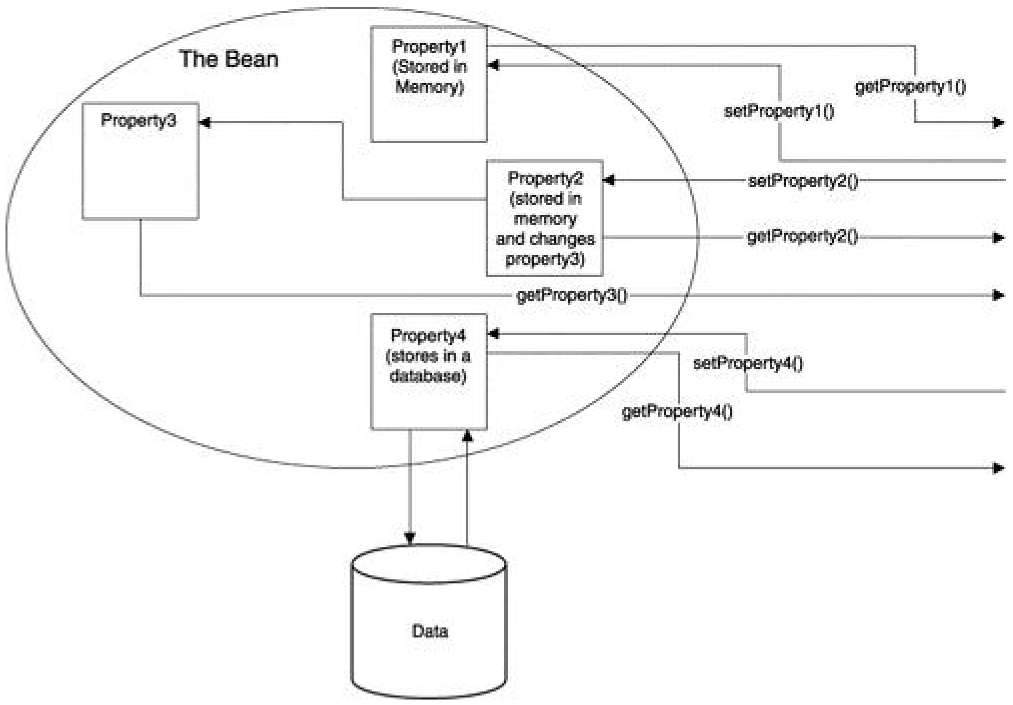
This is now a reasonably good metaphor for a JavaBean. A JavaBean has a set of properties that can be read or changed. It is also possible to find out the names of the properties that a bean has available.

Many objects, processes, and other things in the real world can be modeled as a set of properties. A CD could be described by giving the values of properties representing the year it was released, the record company that produced it, the list of tracks, and so on. Likewise, each track has properties, such as the lyrics, the key it is in, and the length in seconds. Similarly, a stock portfolio can be specified by giving a list of stocks it contains, along with how many shares of each is held and the initial purchase price. The stocks themselves have properties, such as current value, volatility, price-to-earnings ratio, and dozens more. In both of these cases, once the properties of interest have been identified, a JavaBean can be designed to create a model.

Two things make beans especially useful. The first is that neither Java programmers nor JSP programmers need to know anything about a bean in advance in order to use it. A CD bean could be purchased from a bean company, and it would only need to be installed on the local system for JSP authors to start using it immediately.

The second useful thing about beans is that it does not matter how they go about manipulating their properties. It is possible for the request for a value to cause the bean to look up some information in a database. When a bean's property is changed, it could send e-mail to a system administrator with a notification of the new value. In fact, both accessing and changing properties can trigger arbitrarily complex actions, but the JSP author does not need to worry about this. In this sense, beans act as mysterious black boxes with switches and readouts. Page authors can turn the knobs to change properties and view the properties off the readouts without ever knowing what is going on inside the box. This model of a bean is illustrated in Figure 3.1.

Figure 3.1. A bean as a black box.



It is even possible for properties to depend internally on one another. A bean might have the two properties `value1` and `value2` and a third, `sum`, that is always constrained to be `value1` plus `value2`. This would provide all the functionality necessary to build a simple JSP-based calculator, and in fact we will shortly see a bean that does this.
As mentioned previously, for the model/view/controller paradigm to work, the three pieces must fit together easily and naturally. JSPs can interact with beans through three new tags, discussed in the next section.

3.3 JavaBean Tags

The JavaServer Pages specification provides three basic tags for working with beans: one that finds the bean and makes it available to the rest of the page, one that gets a property from the bean, and one that sets one or more properties. Because of the many ways to use a bean, these tags have a number of variations. These tags and their variations are the basic gateways between the JSP view and the bean model.
Note that these tags are very different from the usual HTML tags. Tags such as `<b>` and `</b>` give instructions to the Web browser, saying "present the enclosed text in a bold font." The tags that will shortly be presented give instructions to the JSP engine, telling it

how to build the HTML that will eventually be sent to the browser. The browser itself will never see these tags and wouldn't know what to do with them.

In the most basic form, a bean may be made available to a JSP with the following tag:

```
<jsp:useBean id="bean name"  class="bean class"/>
```

Here `bean name` will be the name that is used later to refer to the bean. This name has only two restrictions. First, it must contain only letters, numbers, and the underscore character.[2] The name must also be unique everywhere it is to be used. This means that two beans cannot have the same `id`, and an `id` cannot be any of the words `request`, `response`, or `out`, which are reserved by the JSP system.

[2] Technically, it must be a "valid Java identifier," which will make more sense after Chapter 9.

The `bean class` will be the name of a Java class that defines the bean. Chapter 9 discusses classes in more detail, but for now, a class can be thought of as the collection of Java code that makes up the bean, just as DNA is the code that makes up a lima bean. This bean class must be available to the JSP engine. The JSP specification includes details on how to make classes available to Web applications, which is discussed in Appendix B.

The trailing slash at the end of the tag is important; it signals that there is no corresponding `</jsp:useBean>` close tag. If this slash is missing, an error will occur at translation time. Examples later in this chapter use a closing tag, but when it is not used, the JSP engine must be told not to expect it.

Errors to Watch For

The most common problem with the `jsp:useBean` tag is specifying a class that the JSP engine cannot find. When the page is tested, this will be reported as a `java.lang.ClassNotFoundException`. This error may be caused by a simple misspelling or typo in the name, or it may be because the file that makes up the class is not in the right directory.

3.3.1 Getting a Property

Once a bean has been obtained with `jsp:useBean`, getting a property is as simple as using the `jsp:getProperty` tag:

```
<jsp:getProperty name="bean name" property="property name"/>
```

`Bean name` will be the same name that was used in the `id` field, and *`property name`* will be the name of the property to get. Listing 3.1 demonstrates the use of these tags.

Listing 3.1 A JSP that gets properties from a bean

```
<jsp:useBean
  id="bean1"
  class="com.awl.jspbook.ch03.Bean1"/>

<p>Here is some data that came from bean1:</p>

<ul>

<li>The name of this bean is:
<jsp:getProperty name="bean1" property="name"/>

<li>The 7th prime number is:
<jsp:getProperty
  name="bean1"
  property="seventhPrimeNumber"/>

<li>The current time is:
<jsp:getProperty name="bean1" property="currentTime"/>

</ul>
```

The first thing this JSP does is make the bean available to the page. Here, the `id` has some relationship to the class name, but that need not be the case. It is also not necessary for the `jsp:useBean` tag to appear right at the top of the page, as long as it appears before any `jsp:getProperty` tags. However, putting all the `jsp:useBean` tags together at the top of a page makes it easy to see all the beans that a page is using.

Once the bean has been loaded, the `jsp:getProperty` tag is used to retrieve data from it. As promised, this hides a number of programming details from the page author. Within the bean, the code that gets the seventh prime number could simply return it from a precomputed list of prime numbers, recompute it each time, or pull it out of a database. In fact, it does the easiest thing and returns the number 17.[3]

[3] If 17 ever ceases to be the seventh prime number, please contact the author, and a replacement bean will be cheerfully provided.

The method the bean uses to get the current date cannot rely on a similar trick, as the date changes every time the page loads. This method must therefore have some code to it, which is shown in Chapter 10.

Finally, note that bean properties can be used anywhere on a page, including within HTML tags, as in:

```
<td bgcolor="<jsp:getProperty name="bean1"
            property="color"/>">
```

Here, the color of a table cell is being pulled out of a bean. For this to work, the color property must be presented in one of the acceptable forms for HTML, such as #FF0000 or "red." This is another place where the person coding the page and the person coding the bean must agree.

Errors to Watch For

The only error that can come from the `jsp:getProperty` tag results from trying to get a property that the bean does not have. This error would be presented on the page as

```
com.sun.jsp.JspException:
getProperty(id): can't find method to read prop
```

Most likely, this error will arise because of a typo in the property name, which can simply be corrected. If the JSP does need a property that the bean does not have, the programmer will have to be asked to add it.

3.3.2 Setting Properties

Of the many ways to set a bean's properties, the simplest looks almost exactly like getting a property:

```
<jsp:setProperty
     name="bean name"
     property= "property name"
     value= "property value"/>
```

Here, `name`, as before, is the `id` from the `jsp:useBean` tag, and `property` is the name of the property to set. `Value` is the new value to assign to the property. The simplest type of value is a string enclosed in quotes, such as `"red"` or `"3"`.
Listing 3.2 shows a JSP that uses a bean with two properties related to the time. The `format` property allows the JSP author to specify the format in which the time should be presented, and the `currentTime` property has the date.

Listing 3.2 A JSP that sets a property

```
<jsp:useBean
  id="date"
  class="com.awl.jspbook.ch03.DateBean"/>

<ul>

<jsp:setProperty name="date" property="format"
 value="EEEE, MMMM dd yyyy 'at' hh:mm"/>

<li><jsp:getProperty name="date" property="currentTime"/>

<jsp:setProperty name="date" property="format"
 value="hh:mm:ss MM/dd/yy"/>

<li><jsp:getProperty name="date" property="currentTime"/>

<jsp:setProperty name="date" property="format"
 value="yyyyy.MMMMM.dd GGG hh:mm aaa"/>

<li><jsp:getProperty name="date" property="currentTime"/>

</ul>
```

Recall from Chapter 2 that a JSP is evaluated by the JSP engine from top to bottom. So, the JSP engine will first see the `jsp:useBean` tag and make the bean available to the page under the name `date`. Then, this bean's `format` property will be set. The exact meaning of the value is unimportant, although details are available in the documentation for the `java.text.SimpleDateFormat` class. The important thing is that when the JSP engine next goes to get the value of the `currentTime` property from the bean, the bean will use the value of the `format` property to render it.

When it encounters the next `jsp:setProperty` tag, the JSP engine will replace the old value of the `format` property with the new one. This is not a problem; that old value has already served its purpose. When it is next asked for `currentTime`, the bean will use the new value and will present the current time differently. The bean might also present a slightly different time, as a few milliseconds will have passed since the last `jsp:getProperty`.

Hard-coded values such as these format specifications are fine for many purposes. But to participate in dynamic pages, beans must be capable of interacting with other dynamic elements.

The Connection Between Forms and Properties

Most interesting dynamic pages are driven at least partially by values that have been provided by users through forms. Because most program logic resides in beans, it seems natural that many JSPs take input values from forms, pass these values into beans via `jsp:setProperty` tags, and then display other properties representing the result of a computation.

Fortunately, the JSP architects realized how common this situation would be and provided another simple tag to accomplish it. If the form is providing a value called `"parameter"` and the bean has a property called `"parameter"`, the value can be set with the tag

```
<jsp:setProperty
     name="bean name"
     property="parameter"/>
```

In this case, the value is implied and is assumed to come from the form. Sometimes, the name of the form parameter and the name of the property will not match. They can be connected through another variation of the `jsp:setProperty` tag:

```
<jsp:setProperty
     name="bean name"
     property="property name"
     param="param name"/>
```

Here, the JSP will use the form parameter called *`param name`* to set the property called *`property name`*.

The most powerful version of the `jsp:setProperty` tag looks through all the parameters provided by the form and all the methods provided by the bean and links them automatically. This version looks as follows:

```
<jsp:setProperty name="bean name" property="*"/>
```

If the form provides values called `param1`, `param2`, and so on and if the bean has properties with the same names, everything will match up perfectly. If the form provides some parameters for which there are no matching properties, these parameters will be ignored, and no error will occur. The JSP could proceed to do something else with those parameters, such as pass them on to the bean manually or send them to a different bean with another `jsp:setProperty` tag. Likewise, if the bean provides properties for which the form does not supply values, these properties will simply not be set. The JSP can call them manually, if needed.

Listing 3.3 shows a page with a simple HTML form. When this form is submitted, the values will be passed to the page in Listing 3.4, which will pass these values to a bean and then use the bean to access the values.

Listing 3.3 A form that sends data to a bean

```
<form action="form_handler.jsp" method="post">

Enter some text: <input type="text" name="textField"><br>

Now select a color: <select name="color">
<option value="red">red
<option value="green">green
<option value="blue">blue
</select><br>

Do you like cheese?
<input type="radio" name="cheese" value="yes">Yes
<input type="radio" name="cheese" value="no">No<br>

<input
   type="submit"
   name="submit"
   value="Send this info to a bean">
```

```
</form>
```

Absolutely nothing is special about this form; you can hardly even tell that it is a JSP! In particular, note that the page with the form knows nothing about the bean that will be receiving the form values. That is handled entirely by the receiving page, which is shown in Listing 3.4.

Listing 3.4 Processing form inputs with a bean

```
<jsp:useBean
  id="values"
  class="com.awl.jspbook.ch03.FormBean"/>
<jsp:setProperty name="values" property="*"/>
Here is your text, in the color you chose:
<font color="<jsp:getProperty
             name="values"
             property="color"/>">
<jsp:getProperty name="values" property="textField"/>
</font><br>

When asked whether you like cheese, you said:
<jsp:getProperty name="values" property="cheese"/>
```

The first line of this example summons a bean into existence and calls it `values`. The second line then passes all the values from the form into this bean by setting properties. This works only because the bean was built to work with this form and so has properties called `text`, `color`, and `cheese`. These properties are then pulled out of the bean in a few ways. The `text` and `cheese` values are simply displayed, but the `color` property is used within a font tag in order to set a color.

The bean in this example is acting like a simple glass box; values are put in with the `jsp:setProperty` tag and can then be viewed with the `jsp:getProperty` tag. However, once a bean has been given values, it can do much more than simply hold them and give them back.

The next example, Listing 3.5, illustrates this. This JSP uses a bean that acts like a simple calculator; it has two properties, called `value1` and `value2`, which may be set from a form. The bean also has a third property, `sum`, which it computes by adding `value1` and `value2`.

Listing 3.5 A bean calculator

```
<jsp:useBean
  id="calc"
  class="com.awl.jspbook.ch03.CalcBean"/>

<jsp:setProperty name="calc" property="*"/>

The sum of your two numbers is
<jsp:getProperty name="calc" property="sum"/>
```

There you have it: a calculator in just four lines of code! The form that calls this page is very simple and so is not shown here, although it may be found on the CD-ROM accompanying this book. As in Listing 3.4, the form knows nothing about the bean but simply provides two text boxes named `value1` and `value2`, and the bean takes care of the rest.

Having conveyed how well beans and forms work together, it's time for a little fine print. For this cooperation to work, it must be an echo of cooperation between the person writing the bean and the person writing the form. They must agree on the names of the form variables, which ones will be multivalued, and so on. This should not be any burden to either person, as the bean interface makes both their jobs easier.

It is important to note that each time a user visits the calculator page, the `jsp:useBean` tag will create a brand new instance of the `CalcBean`. This is a good thing, as it means that if several users access the same page at the same time, they will each get a copy of the bean, with different values for all the properties. This ensures that if one person tries to use the calculator from Listing 3.5 to compute 100 + 156 at the same time that another user is computing 62 + 34, they will get 256 and 96, respectively.

This may not seem like a big deal; in fact, many people would probably not even have thought that this would be an issue. However, in some Web application frameworks, developers must give a great deal of thought to how their systems will behave if many users access the site simultaneously. Fortunately, using beans and JSPs means that developers can write their pages as if only one user will be using them at a time, and they will automatically work properly even when there are many simultaneous users.

Errors to Watch For

Attempting to set a property that bean does not possess will result in the following error:

```
com.sun.jsp.JspException:
  setProperty(id): Can't Find the method for setting prop
```

As in attempting to get a nonexistent property, this error may be a misspelling, or the bean developer must add the needed property. Note that this error will not be generated when setting a bean's properties from a form. If the form names and properties do not match up, the mismatched values will simply be ignored. This can be worse, as it means that a page may not work without an obvious error message to explain why.

Form Values

Unfortunately, simple things are seldom perfect; a bug is lurking in the simple calculator of Listing 3.5. If a user enters on the form a value that is not a number, such as `A`, the page will return a cryptic error message. The bean programmer could avoid this by constructing the bean in such a way that it would simply ignore bad inputs. This would hardly be an improvement, though, as the page would return an incorrect answer instead of an error. At least with the error message, the user knows that something is awry.
What is needed here is a means for the bean to tell the page whether the inputs were valid and for the page to display different text in either case. This is possible, but such interaction between a model and a view is best mediated by a controller, so this will be deferred until Chapter 12.
It would also be possible to use some JavaScript within the page to ensure that the values are legal. This could be done by adding to the form an `onSubmit` value, which would check that the values look like numbers and would pop up an error dialogue if they do not. The advantage to this approach is that the feedback is presented to the user immediately, without having to pass all the data to the server. This also means one less condition the server needs to check and one less submission it needs to process. For very popular applications, reducing the strain on the server in such a way can make a noticeable difference in overall system performance.
Unfortunately, using JavaScript has lots of downsides. First, getting the same JavaScript to work on multiple different browsers can be surprisingly difficult. Browsers' incompatibilities and fragility are especially evident when JavaScript is involved. Many Web programmers decide to target only one browser, making their site unavailable to a portion of their potential audience. Alternatively, it is possible to write many different versions of the JavaScript code and have the server include the appropriate version based

on which browser is being used. JSPs can make it relatively painless to detect the browser being used and include appropriate JavaScript; this will also be demonstrated in Chapter 4. Even using JSPs, all that JavaScript code will still need to be tested and maintained.
In addition to the compatibility issues, some users choose to turn off JavaScript, and some browsers, such as those included in many mobile phones, don't support JavaScript at all. This means that it will be necessary to write the code to handle the form validation on the server anyway.
Finally, it is generally good practice to have all related code in one place. Because the form will ultimately be processed on the server, it makes aesthetic sense to keep the validation code there as well. This way, if and when the business logic changes, it will not be necessary to hunt down the JavaScript code that checks values and the completely separate Java code that uses these values.

3.4 Making Data Available Throughout an Application

So far, all the beans used have been fairly transient entities: They are summoned into existence through a `jsp:useBean` tag at the top of a page and disappear when the page ends. This is usually a good thing; in fact, this is the very property that allows each user to get his or her own version of the bean.
However, in many situations, it is useful to have a bean last longer than a single page. A simple mechanism allows JSP authors to specify any of several lifetimes for their beans. These lifetimes are called *scopes*, and they indicate the period of activity during which the bean is available. If a bean is created and placed in a scope, any changes to that bean made via `jsp:setProperty` tags will be visible to other pages using the same bean.
A scope may be given to a bean by adding `scope=` to the `jsp:useBean` tag:

```
<jsp:useBean
  id="bean name"
  class="bean class"
  scope="scope"/>
```

Legal values for the scope property are `"page"`, `"request"`, `"session"`, and `"application"`. Each may be useful in different situations.

3.4.1 The Page Scope

Beans created in the page scope, the smallest scope, will be available only within the JSP from which they were created. It is reasonable to think of the page scope as spanning a single JSP file.

This means that if one page includes another by using a `jsp:include` tag and that if they each use a `jsp:useBean` tag to obtain the same bean in the page scope, they will each in fact get a different instance of the bean. This is illustrated in Listing 3.6.

Listing 3.6 A page that uses page scope

```
<jsp:useBean
  id="bean1"
  class="com.awl.jspbook.ch03.AnimalBean"
  scope="page"/>

At first, our animal is:
<jsp:getProperty name="bean1" property="animal"/>
<br>
<jsp:setProperty
  name="bean1"
  property="animal"
  value="octopus"/>

After setting the property, the animal is:
<jsp:getProperty name="bean1" property="animal"/>
<br>

<jsp:include page="page_scope2.jsp"/>
```

This example creates a bean, places it in the page scope, and then shows the value of the bean's `animal` property. This bean comes with a default value for this property, `"ferret"`, which will be displayed by the first `jsp:getProperty`. This property is then changed to `"octopus"` and redisplayed, to prove that it changed. Then the page includes another page, which is shown in Listing 3.7.

Listing 3.7 An included page that reuses the bean

```
<jsp:useBean
  id="bean1"
  class="com.awl.jspbook.ch03.AnimalBean"
  scope="page"/>

In the include, the animal is:
<jsp:getProperty name="bean1" property="animal"/>
```

This page obtains the same bean, and once again displays the `animal` property. Because the bean is in the page scope, a completely new instance of the bean will be created when the included page reaches the `jsp:useBean` statement; this instance will still have the original value, `"ferret"`, and so the page will display the following:

```
At first, our animal is: ferret
After setting the property, the animal is: octopus
In the include, the animal is: ferret
```

3.4.2 The Request Scope

The request scope is larger than the page scope; beans created in the request scope will last from the time they are first created until the last of the data is sent to the user. Most significantly, this means that the same instance of the bean will be available to all included pages, which can be demonstrated by changing `scope="page"` to `scope="request"` in Listings 3.6 and 3.7. When this is done, the resulting page will look like this:

```
At first, our animal is: ferret
After setting the property, the animal is: octopus
In the include, the animal is: octopus
```

The request scope is useful when multiple components of a page are scattered among several files that include one another, and all need access to the same data. As this is the most common situation, the request scope is the default. In all the previous examples, in which no scope was specified, we were in fact using the request scope without knowing it.

3.4.3 The Session Scope

Both of the scopes encountered so far associate data with pages or pieces of pages. But because users, not pages, should be the focus of a site, a way is needed to tie data to a particular user so it will be available and adjustable from whichever pages the user visits.

An obvious example of this is a shopping cart. Multiple users can all see the same "checkout" page of a shopping site, but each user will see his or her own selections. Likewise, many users may view a particular item, but when one user elects to buy it, the item goes into that person's shopping cart and no one else's.

In JSP terms, data associated with a user is in the *session scope*. A session does not correspond directly to a user but rather to the period of time the user spends at a site. Typically, this period is defined as extending from the first visit to the site, through the point at which the user has not accessed any pages on the site for more than half an hour. Thereafter, the session will have *expired*, and any beans in the session scope will disappear. Listing 3.8 demonstrates a simple use of the session scope to keep track of how many times a user has accessed the site.

Listing 3.8 A counter bean in the session scope

```
<jsp:useBean
  id="counter"
  class="com.awl.jspbook.ch03.CounterBean"
  scope="session"/>

<jsp:setProperty
  name="counter"
  property="incrementCount"
  value="1"/>

You have visited this page
<jsp:getProperty name="counter" property="count"/>
time(s).
```

This example doesn't look like anything special, but a user who accesses the page and repeatedly clicks the browser's reload button will see the counter steadily climb. Two things make this page work. The first is the way the `incrementCount` property works. When this property is set, it adds its value to the count property, much as Listing 3.5 added the `value1` and `value2` properties to set the `sum` property.

More important, the secret to this page's functionality is the session scope, which keeps the bean around even after the page completes. If the scope were changed to page or request, the counter would stay stuck at 1 perpetually.

The session scope extends across multiple pages, as well as across individual pages multiple times. If the `jsp:useBean` and `jsp:setProperty` tags were copied into another page, the counter would then register the total number of times the user had visited both pages during the current session.

3.4.4 The Application Scope

The application scope is the largest of the available scopes. Beans placed in the application scope will be available to all pages, for all users, until the Web application is shut down or restarted. This scope is useful for displaying global information, such as a "quote of the day" that might be displayed at the bottom of every page across a site. The application scope can also be used to create a counter that displays the total number of times a page has been accessed. A simple modification to Listing 3.8 counts how many times each user has accessed a page: To make this global, change `scope="session"` to `scope="application"`.

3.5 Special Actions When Beans Are Created

Now that beans can live beyond the scope of a single page, any given instance of a `jsp:useBean` tag may or may not create a new instance of a bean. Often a page will need to take a special action when the bean is first created.
This can be done with a simple extension to the `jsp:useBean` tag. So far, this tag has been used without a closing `/jsp:useBean` tag. If there is a matching close tag, anything in the body between the open and close tags will be evaluated when the bean is created and not subsequently.
The content enclosed by these tags can be anything at all. For a start, it can be plain text. Listing 3.9, a slight modification to Listing 3.8, illustrates this ability.

Listing 3.9 Displaying text when a bean is created

```
<jsp:useBean
  id="counter2"
  class="com.awl.jspbook.ch03.CounterBean"
  scope="session">
```

```
Welcome to the site!

</jsp:useBean>
```

With this change, the user will see the message "Welcome to the site" when the bean is created. On the user's subsequent visits to this page, or any other using the counter bean, the message will not be displayed. Note that the identifier `counter2` is used to distinguish this bean from the one used in Listing 3.8. If this were not done, the counter would register how many times the user had been to *either* page, and if counter.jsp were visited before counter2.jsp, the welcome message would not show up.

In addition to text, the initialization block can contain `jsp:setProperty` tags. This can be useful when the bean needs to have certain values before it is used. In some cases, these default values can be placed in the bean itself, such as the initial value of 0 for the number of visits in the counter bean. In other cases, the values with which the bean needs to be initialized may depend on the current user or the page or any of thousands of other possibilities. Rather than putting values with these kinds of dependencies into the bean code, it makes more sense to let the page set up the beans as needed.

Listing 3.10 shows another addition to Listing 3.8. This time, the bean will be told the name of the page on which it was created.

Listing 3.10 Setting a value when a bean is created

```
<jsp:useBean
  id="counter3"
  class="com.awl.jspbook.ch03.CounterBean"
  scope="session">

<jsp:setProperty
  name="counter3"
  property="firstPage"
  value="counter3.jsp"/>

Welcome to the site!

</jsp:useBean>

<jsp:setProperty
```

```
  name="counter3"
  property="incrementCount"
  value="1"/>

The first page you visited was
<jsp:getProperty name="counter3" property="firstPage"/>

You have seen
<jsp:getProperty name="counter3" property="count"/>
page(s) on this site.
```

The bean will now keep track of where it was created. If this bean were then used on multiple pages, it could report on the total number of pages visited, as well as the first page.

At this point, it is natural to wonder whether there are corresponding close tags for `jsp:getProperty` and `jsp:setProperty`. In fact, there are not. Although it is often necessary to perform different actions, depending on whether a property is set, this is done using special JSP tags called conditionals, which will be discussed in the next chapter.

3.6 Making Beans Last Forever

Although it is not a scope, a related technology extends the lifetime of a bean. Beans can be *frozen* by saving their contents into files, and these files can then be transparently turned back into beans when they are needed.

The mechanism to turn beans into files is called *serialization*, a built-in facility of the Java programming language. Serialization is also very useful to JSP authors because it allows beans to be tailored, or customized, before they are used.

For example, a software company might make a bean that computes how quickly money in a bank account will grow. One property of this bean would be the interest rate. Many different banks could purchase this bean, set the `interestRate` property to the appropriate value, and then save the bean.

When they create their Web pages, the JSP authors at these banks can use the standard bean tags to access the bean, and none of the pages they create will need to worry about the interest rate. This also makes these sites easier to maintain. When the interest rate changes, the administrator will simply need to replace the serialized file with one containing the new rate, and all the pages will automatically use the new value. In a sense, serialization can be used to "bake in" values that are appropriate to a site. JSPs can then use these values as if they were intrinsic to the bean.

The details of how serialization is done are beyond the scope of this book, but many programs hide the details and make it easy to create such frozen beans. Sun provides one called the Bean Box, although this is targeted primarily for people using beans to build graphic front ends and is overkill for the kinds of beans used in JSPs. A much simpler editor from Canetoad Software is included on the CD-ROM for this book, along with instructions for its use.

Using a serialized bean is no more complicated than creating a bean from scratch; it simply requires a slight variation to the `jsp:useBean` tag:

```
<jsp:useBean
      id="bean name"
      beanName="bean name"
      type="bean class"/>
```

Here, `id` is the name by which the JSP will use the bean, the same as always, and `beanName` should be the name of a file containing a serialized bean. By convention, such files end with the .ser extension, which should not be included in the name. Finally, `type` is the class or interface for which the bean is an instance. Note that type is used instead of class because the class is implicitly provided by the serialized file. An instance always knows what class it is an instance of, and this is true even when that instance has been stored in a file. The type is still necessary because the JSP still needs to assign a type to the variable that will hold the bean. In practice, the type can usually be thought of as the bean's class.

Listing 3.11 shows a JSP that uses a serialized bean to get information about a record, in this case "Tinderbox" by Siouxsie and the Banshees.

Listing 3.11 Using a serialized bean

```
<jsp:useBean id="album" beanName="tinderbox3"
 type="com.awl.jspbook.ch03.AlbumInfo"/>

<body
  bgcolor="<jsp:getProperty
             name="album"
```

```
            property="bgColor"/>"
  text="<jsp:getProperty
            name="album"
            property="textColor"/>">

<h1><jsp:getProperty name="album" property="name"/></h1>

Artist: <jsp:getProperty name="album"
          property="artist"/><p>
Year: <jsp:getProperty name="album" property="year"/></p>
```

This looks much like the other examples in this chapter, except for the `jsp:useBean` tag and the fact that there is no obvious place where the properties have been set. The reason is that they are not set in the JSP but have already been stored in the serialized file.

It would be nice if this JSP could also display the list of tracks, but there is a problem. In general, the page won't know in advance how many tracks are on a given CD. The bean could have a separate property for each track, and the page could display any *fixed* number of tracks in the obvious way:

```
<li><jsp:getProperty name="album" property="trackName1"/>
<li><jsp:getProperty name="album" property="trackName2"/>
<li><jsp:getProperty name="album" property="trackName3"/>
```

However, if the CD has only two tracks, this will display an extra bullet with no name next to it. If the CD has four or more tracks, some won't get shown at all. What is needed is a way to repeat some region of HTML, once for each track in the bean. This is called *iteration* and is covered in the next chapter.

This use of serialization makes beans behave a little like a database. Perhaps the "tinderbox" bean came as part of the collection of beans for all of the Banshees' albums. To create pages for these others, it would be necessary only to change the `beanName` to `"Hyaena"` or `"Juju"`, or so on. In fact, the deep connection between beans and databases is explored more fully in Chapter 6.

3.7 Java News Today and Beans

The Java News Today staff loves beans and is planning to build the majority of the JNT site around bean technologies. To start with, beans will allow JNT to add to the site a new feature: a daily quiz consisting of a single multiple-choice question. This quiz will appear in the right-hand navigation bar right above the list of sections. Of course, in order to make this new component easy to work with, it will be stored in a separate file called quiz.jsp, which will be included in the navigation bar with a `jsp:include` tag. The new version of the navigation is shown in Listing 3.12.

Listing 3.12 The new navigation bar

```
<div class="bordered">
<jsp:include page="quiz.jsp"/>
</div>
```

Now, onto the quiz itself. Beans have two aspects that will make this feature very easy to add. The first is that the user will respond to the quiz through a form; as we have seen, beans excel at handling form inputs. The second concerns the way the questions will be stored. If the questions were placed directly in the HTML, it would be a pain to update them. Instead, the questions and the correct response will be stored in a serialized bean, which can be updated using any of several available bean editors. Listing 3.13 shows how this bean will be used in quiz.jsp.

Listing 3.13 The daily quiz

```
<jsp:useBean
  id="quiz"
  beanName="todaysQuiz3"
  type="com.awl.jspbook.ch03.QuizBean"/>

<jsp:getProperty name="quiz" property="question"/><P>

<form action="quiz_result.jsp" method="POST">
  <input type="radio" name="userGuess" value="1">
  <jsp:getProperty name="quiz" property="answer1"/><BR>

  <input type="radio" name="userGuess" value="2">
  <jsp:getProperty name="quiz" property="answer2"/><BR>

  <input type="radio" name="userGuess" value="3">
  <jsp:getProperty name="quiz" property="answer3"/><BR>
  <input type="Submit" name="Guess" value="Guess">
</form>
```

There are no new surprises here; the serialized bean is obtained with the `jsp:useBean` tag, and properties are put on the page with the `jsp:getProperty` tag. Recall from Listing 3.4 that it is not necessary for a form page to obtain a bean, even if the form will ultimately be sending data to that bean. This is still true; in this example, the bean is being used not directly with the form but only to obtain blocks of text that are displayed within the form.

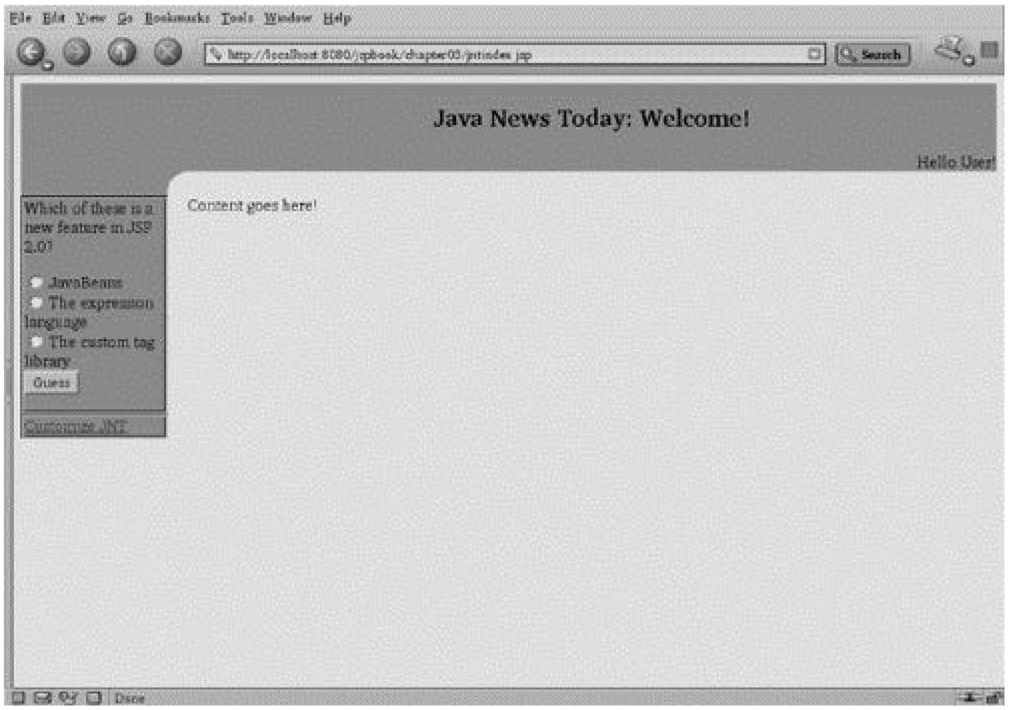
The JNT designers are now ready to start thinking about personalization, at least at a very high abstract level. Even though it is not yet clear how personalization will work, it is safe to assume that every user will be modeled as a bean and that this bean will have various properties describing the user's preferences. This is enough to start laying some groundwork. To start with, the designers will use this bean to customize the header by displaying the user's name, if it has been provided. This is shown in Listing 3.14.

Listing 3.14 The header, with a bean property

```
<jsp:useBean id="user"
 class="com.awl.jspbook.ch03.UserInfoBean"
 scope="session"/>

<center><h2>Java News Today: Welcome!</h2></center>

<div class="left">
  Hello <jsp:getProperty name="user" property="name"/>!
</div>
```

To prepare further for customization, a link will also be added to the left navigation bar. With the addition of the link and the quiz, the front page is starting to look like a real site. It is shown in Figure 3.2.

Figure 3.2. The new JNT home page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter03/jstindex.jsp
Search
Java News Today: Welcome!
Hello User!
Which of these is a new feature in JSP 2.0?
JavaBeans
The expression language
The custom tag library
Guess
Customize JNT
Content goes here!
Done

It is also possible to use custom values in the style sheet: for example, to change the color behind the navigation bar and header. This is illustrated in Listing 3.15.

Listing 3.15 The style sheet, with a bean property

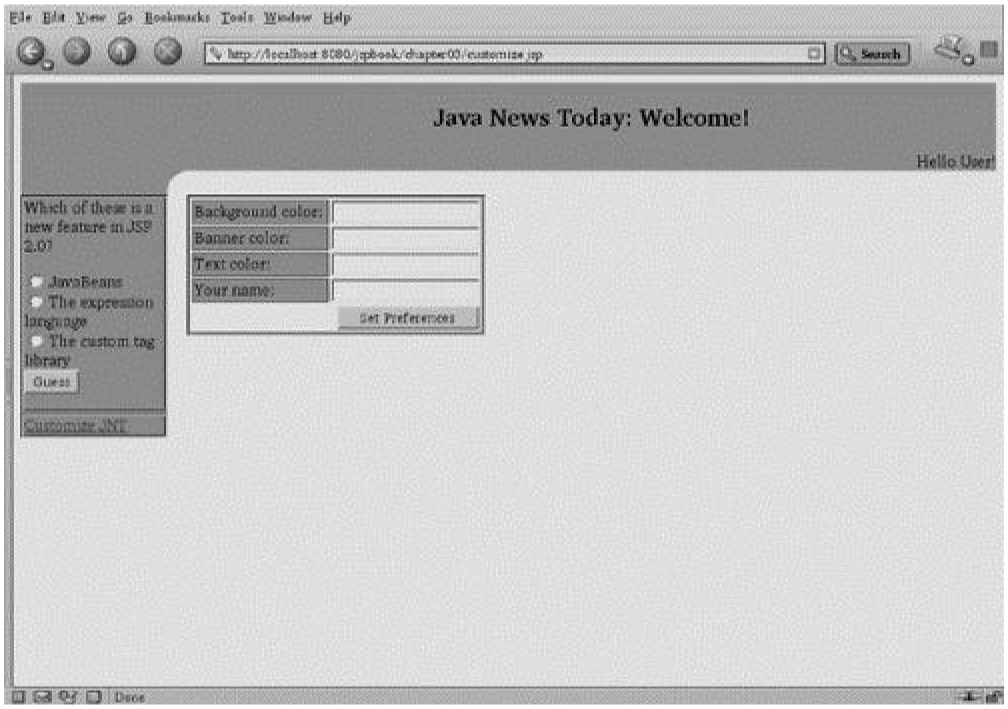
```
<jsp:useBean id="user"
 class="com.awl.jspbook.ch03.UserInfoBean"
 scope="session"/>

TABLE.form { border-style: groove;
           border-color: #004400; }

TD.label { border-style: solid;
         border-width: 1px;
         border-color: #00aa00;
         background: #<jsp:getProperty
             name="user"
             property="bannerColor"/>;
         color: #000000;
         padding-right: 5px }
```

```
TD.form { border-style: solid;
         border-width: 1px;
         border-color: #004400;}

TD.borders { background:
            #<jsp:getProperty
              name="user"
              property="bannerColor"/>; }

DIV.bordered { border-style: groove;
              border-color: #004400; }

DIV.left { margin: 0px 0px 0px 0px;
          padding: 0px 0px 0px 0px;
          text-align: right; }
```

There is one small catch to changing the colors. The little corner bit, which is used to give the site a slightly smoother look, is an image, and its color is not controlled by the style sheet. It will therefore look noticeably out of place if the colors are changed. This could be solved by making part of the image transparent in a clever way, but it is also possible to generate the image dynamically so that its colors match the site. The advantage of the latter technique is that it makes it possible to handle an image with multiple different colors that all need to match colors on the site. The disadvantage is that this requires some advanced techniques, and so it will not be possible to discuss this until Chapter 14.

Difficulties with the image aside, it is already possible for the user to modify these values, thanks to the fact that the bean is in the session scope. Setting new values can be done by providing a simple form that will set the name and color attributes. Because the bean is in the session scope, these values, once set by such a form, will remain set until the session expires or the user changes them. Listing 3.16 shows the customization form, and the page as seen in a browser is shown in Figure 3.3.

Figure 3.3. The JNT customization page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter03/customize.jsp
Search
Done

Java News Today: Welcome!

Hello User!

Which of these is a new feature in JSP 2.0?

- JavaBeans
- The expression language
- The custom tag library

Guess

Customize JNT

Background color:	
Banner color:	
Text color:	
Your name:	
	Set Preferences

Listing 3.16 The customization form

```
<jsp:include page="top.jsp"/>

<table class="form">
<form action="customize_handler.jsp" method="post">
<tr>
  <td class="label">Background color:</td>
  <td><input type="text" name="bgColor"></td>
</tr>

<tr>
  <td class="label">Banner color:</td>
  <td><input type="text" name="bannerColor"></td>
</tr>

<tr>
  <td class="label">Text color:</td>
  <td><input type="text" name="textColor"></td>
</tr>
```

```
<tr>
  <td class="label">Your name:</td>
  <td><input type="text" name="name"></td>
</tr>

<tr>
  <td colspan="2" align="right">
    <input type="submit" name="go" value="Set Preferences">
  </td>
</tr>
</form>
</table>

<jsp:include page="bottom.jsp"/>
```

This is yet another standard HTML form. As expected, the fun happens on the receiving page, which simply sets the bean and notifies the user that his or her preferences have been changed. This is shown in Listing 3.17.

Listing 3.17 The customization handler

```
<jsp:include page="top.jsp"/>

<jsp:useBean id="user"
 class="com.awl.jspbook.ch03.UserInfoBean"
 scope="session"/>

<jsp:setProperty name="user" property="*"/>

Your preferences have been set!

<jsp:include page="bottom.jsp"/>
```

These three examples should demonstrate how easy it is to use beans in order to make sites dynamic. On one page, the bean is simply used to display values. Another page has a very simple form that sends values to the bean in order to change these values. Finally, a third page uses a single `jsp:setProperty` tag to do the setting. From then on, the new values will be available to the first page.

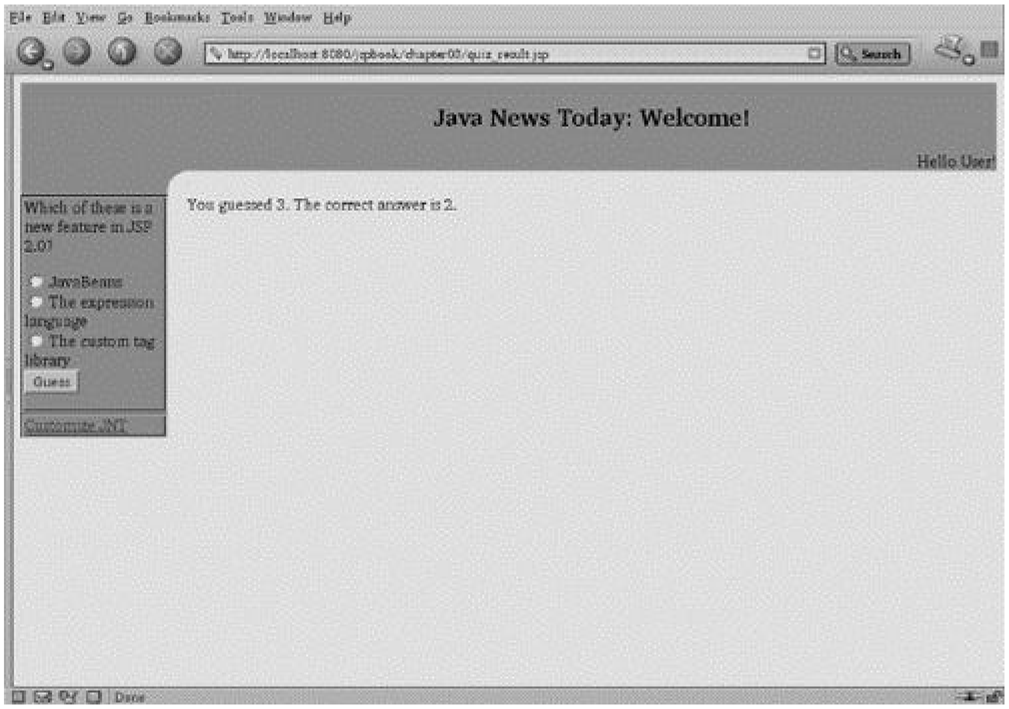
Finally, to tie it all together, a link to the customization page can be added to the navigation bar, as shown in Listing 3.18.

Listing 3.18 Adding a link to the customization page

```
<jsp:include page="top.jsp"/>

<jsp:useBean id="user"
 class="com.awl.jspbook.ch03.UserInfoBean"
 scope="session"/>

<jsp:setProperty name="user" property="*"/>

Your preferences have been set!

<jsp:include page="bottom.jsp"/>
```

Although customization is proceeding nicely, things are not looking as good for the quiz. Unfortunately, it is not yet possible to write the result page, which will state whether the user's guess was correct. This will require one of those conditionals mentioned earlier and so will have to wait for the next chapter. For the moment, however, it is possible to create a page that will at least tell the user what the right answer is, along with the user's guess. This is shown in Listing 3.19 and illustrated in Figure 3.4.

Figure 3.4. The placeholder quiz result page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter03/quiz_result.jsp
Search
Java News Today: Welcome!
Hello User!
Which of these is a new feature in JSP 2.0?
JavaBeans
The expression language
The custom tag library
Guess
Customize JNT
Done

You guessed 3. The correct answer is 2.

Listing 3.19 The quiz result page

```
<jsp:include page="top.jsp"/>

<%-- Start content --%>
<jsp:useBean
  id="quiz"
  beanName="todaysQuiz3"
  type="com.awl.jspbook.ch03.QuizBean"/>

<jsp:setProperty name="quiz" property="*"/>
You guessed
<jsp:getProperty name="quiz" property="userGuess"/>.

The correct answer is
<jsp:getProperty name="quiz" property="correctAnswer"/>.
<%-- End content --%>

<jsp:include page="bottom.jsp"/>
```

Again, this is very straightforward; the guess is sent with a standard `jsp:setProperty` tag to the bean, and then both the guess and the right answer are shown with `jsp:setProperty` tags. Note again how easy it was to create a new JNT page by adding the two appropriate `jsp:include` tags.

3.8 Future Directions

As noted earlier in this chapter, there is more to beans than the names of the properties. One other feature beans provide is the ability to notify one another when certain events have occurred. For example, a bean used on one page could notify a bean used on another that a user had just visited that page, provided some input on a form, or any of a million other things. JSPs have no built-in facility to connect beans in this way, so this ability will not be discussed in any further detail. However, it would not be surprising to see this as a feature in a future version of the JSP spec, so stay tuned. In the meantime, there is no reason interested programmers or page authors cannot use this ability manually.

A number of products that make beans even more powerful and useful are available. Sun provides a set of classes, called the Infobus, that further extends the way beans can communicate with one another. The Java Activation Framework package adds the ability for beans to discover dynamically the type of pieces of data and the available methods related to that type.

Finally, the Java 2 Enterprise Edition makes extensive use of some additional types of beans, collectively known as Enterprise JavaBeans, or EJBs. EJBs provide a suite of methods for managing persistent data, ensuring that beans are kept in a consistent state, and using beans in distributed environments, where different beans may reside on different computers on a network. EJBs are beyond the scope of this book, but it is likely that they will more and more converge with JSPs.

3.9 Summary and Conclusions

Beans are Java's standard component model and integrate well with JSPs. Beans help separate Java code from HTML by providing standard tags that allow the JSP to get data to and from the bean, via the bean's properties. Beans also make writing dynamic pages that use forms easier, by providing easy ways to send form data into beans and get results out. By supporting serialization, beans also help pull changeable data out of pages, which allows a bean to be customized and stored. This customization can tailor a bean for a site or a period of time.

Chapter 10 discusses how to write beans in more detail. In the meantime, the source code for all the beans used in this chapter is included on the CD-ROM for interested readers to explore.

In order to complete the calculator and quiz examples from this chapter, a page must be able to customize itself based on certain criteria. In order to do this, the page will need to use the special JSP tags from the standard library. This is the topic of the next chapter.

3.10 Tags Learned in This Chapter

`jsp:useBean` Makes a bean available to a page

Parameters:

`id`: The name by which this bean will be known to the rest of the page

`class`: The Java class that represents the bean

`beanName`: For serialized beans, indicates the file where the bean is stored

`type`: For serialized beans, indicates the type

`scope`: The scope page, request, session, or application in which the bean is stored.

Body: Optional arbitrary JSP code or text. If present, a body will be evaluated when the bean is created.

`jsp:useBean` Sets a property in a bean

Parameters:

`name`: The name of the bean; should match the `id` in the `useBean` tag

`property`: The name of the property to set, or `"*"` to set all available properties from a form

`value`: If present, specifies the value to set; if not present, the value from the form

Body: None

`jsp:useBean` gets a property from a bean

Parameters:

`name`: The name of the bean; should match the `id` in the `useBean` tag

`property`: The name of the property to get

Body: None

Chapter 4. The Standard Tag Library

Chapter 3 explained how to get values from beans to pages with the `jsp:getProperty` tag, along with a number of limitations in this process. There was no good way to display the tracks on a CD, because the page has no way to know how many tracks a bean will be holding. The quiz was unable to determine whether the user's answer was correct, because the page has no way to compare two values in a bean.

Both of these problems can be solved by a new set of tags: the standard tag library. Although these tags are not technically a portion of the JSP specification, they are closely related and can be used in any application server that supports JSPs. This chapter looks at what these tags can do, after a few words on how tags in JavaServer Pages work in general.

4.1 Tag Libraries

We have already seen tags that deal with things ranging from including other JSPs to manipulating beans. These tags are all useful and perform their specific tasks well, but almost from the beginning, the authors of the JSP specification realized that no set of tags could possibly do everything that everyone would need from JSPs. To address that issue, those authors provided a mechanism for programmers to create new tags that could do anything possible and an easy way for pages to use these custom tags. The topic of the creation of new tags is covered in Chapter 13. Listing 4.1 illustrates how a page loads and uses a tag.

Listing 4.1 A JSP that uses a custom tag

```
<%@ taglib prefix="awl"
   uri="http://jspbook.awl.com/samples" %>
The time, in two different formats:<p>
<awl:date format="EEEE, MMMM dd yyyy 'at' hh:mm"/><br>
<awl:date format="hh:mm:ss MM/dd/yy"/><br>
```

The tag library is loaded with the first line. The URI (Uniform Resource Identifier) specifies the location of the tag library definition, and the prefix specifies the name that will be used to access the tags. Here, the prefix is `awl`, but it could be anything, as long as it is used consistently. One of the tags from this library, `time`, is used twice in the last two lines. The name of the tag is prepended by the prefix specified at the top.[1]

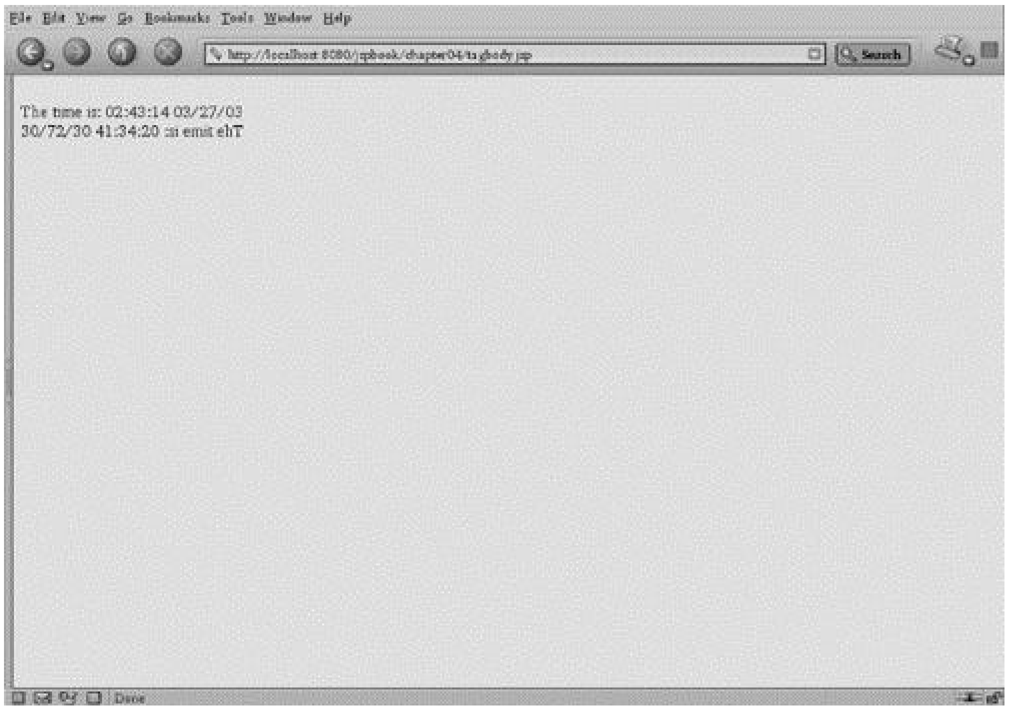
[1] Formally, the tag lives in an XML namespace specified by the prefix. Custom tags can be loaded with any namespace; formally, the portion before the colon is not part of the name. In the text however, this prefix will always be included to avoid possible confusion between tags, such as `c:param` and `sql:param`.

The `awl:time` tag itself simply sends the current time to the page, in a format specified by the `format` property. If this looks familiar, it is because this does essentially the same thing as Listing 3.2. That example used a bean with an input for the format and an output for the time. Using a custom tag, the input is specified as a named property, and the output is implicit in the way the tag works.
Technically, neither example was particularly good. Because they play the part of models in the model/view/controller paradigm, beans should not be concerned with how their data will be presented. Hence, the bean used in Listing 3.2 should not have had to deal with formatting issues. Similarly, tags are intrinsically part of the view portion and so should not deal directly with data, but the `awl:time` tag in Listing 4.1 holds data in the form of the current time. With some effort, the standard tag library can help make such separations of roles between tags and beans easier to manage, as will be seen later in this chapter.

4.2 Tags with Bodies

Custom tags can do more than output data controlled by parameters. A custom tag can have a body, which it can control in arbitrary ways. Recall a similar tag, `jsp:useBean`, which renders its body only when the bean it is accessing is created. Listing 4.2 shows such a custom tag that can be used to display its body, hide it, or even reverse it. The result is shown in Figure 4.1.

Figure 4.1. The result of a custom tag.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter04/tagbody.jsp
Search
Done

```
The time is: 02:43:14 03/27/03
30/72/30 41:34:20 :si emit ehT
```

Listing 4.2 A custom tag with a body

```
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<awl:maybeShow show="no">
You can't see me!
</awl:maybeShow><br>

<awl:maybeShow show="yes">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow><br>

<awl:maybeShow show="reverse">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow><br>
```

This example loads the same tag library used in Listing 4.1 and again specifies that it will be using the `awl` prefix to access the tags. The tag used this time is called `awl:maybeShow`, and it has a parameter, `show`, that controls what the tag should do with its body. This parameter may be set to `no`, in which case the body is hidden from the page; `yes`, in which case the body is displayed; or `reverse`, in which case the body is shown backward.

Note that the body of the `awl:maybeShow` tag may include anything, including other JSP tags. This was also true of the `jsp:useBean` tag and in fact is true of any custom tag that has been properly programmed. This property is described by saying that JSP tags can be *nested*. From here on, it will simply be assumed, unless otherwise noted, that the body of any tag can contain any other tag.

4.3 Dynamic Attributes in Tags

For the standard tag library to be able to do all the wonderful things it claims to do, the tags will need to take parameters that are more complicated than such simple instructions as "yes" and "no." In fact, the parameters to the standard tag library comprise a full language, although one that is significantly simpler than Java itself and much better suited for building pages.

This language is built into the very core of JSPs in the latest version of the JSP specification. This means that programmers creating new tags may use this language for their own purposes; this will also be illustrated in Chapter 13.

Expressions in this language are surrounded by braces and preceded by a dollar sign. The simplest kinds of expressions in the language are constants, such as strings or numbers:

```
${23}
${98.6}
${'hello'}
```

These expressions don't mean anything on their own, but when used as the value of a parameter, they are evaluated by the expression language before they are sent to the tag. Because numbers and strings evaluate to themselves, this means that the following two expressions mean the same thing:

```
<awl:maybeShow show="${'yes'}">

<awl:maybeShow show="yes">
```

Note that within an expressions, literals are surrounded by single quotes and that the whole expression is surrounded by double quotes.

Errors to Watch For

If an expression is written incorrectly, such as leaving off a closing quote or a brace, a JSP page error will report something like

```
An error occurred while parsing custom action
```

Now for the fun part: The scripting language can also refer to beans and properties of beans. Listing 3.1 used a bean to display some static properties, including the seventh prime number. Suppose that bean were loaded into a page with this tag:

```
<jsp:useBean id="bean1" class="com.awl.jspbook.ch03.Bean1"/>
```

In that case, then the scripting language would refer to the seventh prime number property as

```
${bean1.seventhPrimeNumber}
```

Note the pattern: first, the name of the bean as defined in the `jsp:useBean` tag, then a dot, then the name of the property. This is not exactly equivalent to the `jsp:getProperty` tag, as dropping this script fragment into a page will not display the value. In fact, it will not do anything at all. However, this would serve perfectly as a way to send the seventh prime number to a custom tag. Admittedly, there would probably never be any need to do such a thing, but often it will be necessary to send a value from a form to a tag. We now have the means to do this: Send the form inputs into a bean with the `jsp:setProperty` tag and then send the value from the bean to a tag with a scripted parameter.

Errors to Watch For

If an attempt is made to access a property that does not exist, a page error that looks like the following will be generated:

```
Unable to find a value for "property"
in object of class "beanClass"
```

Listing 4.3 shows a simple form that lets the user choose whether to show, hide, or reverse a block of text.

Listing 4.3 A form that will be used by a tag

```
<html>
<body>
```

```
<form action="show_result.jsp" method="post">
Shall I display the tag body?
<select name="shouldShow">
<option>yes
<option>no
<option>reverse
</select><br>

<input type="Submit" name="Go" value="Go">
</form>

</body>
</html>
```

The page that will use this form is shown in Listing 4.4. It combines many of the things that have been discussed so far: a bean, the `awl:maybeShow` tag, and a scripted parameter.

Listing 4.4 Using a bean and a tag together

```
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
  id="form"
  class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>

<awl:maybeShow show="${form.shouldShow}">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow>
```

The first portion of this example should be old hat by now: First, a tag library is loaded, and then a bean is obtained and fed the form values. The second part uses the tag almost exactly as in Listing 4.2. The only difference is that the `show` parameter is not a fixed value but comes from the bean via a script. Using a bean, a custom tag, and the scripting language, we can now dynamically control a whole block of text!

4.4 Displaying Expressions

The ability to use a bean to control a tag is certainly powerful, but often such values must be shown to the user rather than used by a tag. A standard tag, `c:out`, renders values to the page, and the use of this tag is quite straightforward. Listing 4.5 revisits the example from Listing 3.1, which displayed various values from a bean. Listing 4.5 use the same bean but now displays values using the new tag.

Listing 4.5 The `out` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="bean1"
  class="com.awl.jspbook.ch03.Bean1"/>

<p>Here is some data that came from bean1:</p>

<ul>

<li>The name of this bean is:
<c:out value="${bean1.name}"/>

<li>The 7th prime number is:
<c:out value="${bean1.seventhPrimeNumber}"/>

<li>The current time is:
<c:out value="${bean1.currentTime}"/>

</ul>
```

Because this does exactly the same thing as Listing 3.1, it may not be immediately clear why anyone would use the `c:out` tag instead of the `jsp:getProperty` tag. Although `c:out` is somewhat shorter, the real reason to use it is that it has many advantages, all of which are derived from the fact that what is being shown is the result of a script, not a simple property.

The expression language allows page developers to manipulate properties in many ways. For example, it is possible to write an expression that will add two numbers right in the page, without needing to rely on the bean to do it. Listing 4.6 shows another version of our calculator from Listing 3.6, only doing the addition in the page.

Listing 4.6 Addition in the expression language

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="calc"
  class="com.awl.jspbook.ch04.CalcBean"/>
<jsp:setProperty name="calc" property="*"/>

The sum is:
<c:out value="${calc.value1 + calc.value2}"/>
```

It is now possible to extend this easily to do more complex calculations, such as finding the average of the two numbers or raising one to the power of the other, and so on. Note that although this is very powerful, it also breaks the model/view/controller paradigm, as the model is now being manipulated directly from the view. Sometimes, this is worth doing, but as a general rule of thumb, it is better to leave such calculations in the bean.

Another advantage to the `c:out` tag is that it can display things other than beans. Every JSP has available a number of *implicit objects*, that is, objects that the system provides without the developer's needing to load or name them explicitly. One of these is the `pageContext` object, which contains a great deal of information about the action currently being performed, such as the name of the page being generated, the name of the computer from which the request came, and so on. Listing 4.7 uses the `pageContext` object to display some of the available information.

Listing 4.7 The `request` object

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<ul>

<li>Your computer is called
```

```
<c:out value="${pageContext.request.remoteHost}"/>

<li>This page came from server
<c:out value="${pageContext.request.serverName}"/>

<li>This page came from port
<c:out value="${pageContext.request.serverPort}"/>

</ul>
```

This example illustrates a new kind of syntax: expressions with multiple dots. This will make more sense following the discussion of compound data later in this chapter.

Another important implicit variable is `param`, and it holds all the values that have been sent to a page by a form. This variable acts like a special bean in that it does not have a predefined set of properties but instead has a property for every value in the form.[2]

Suppose, for example, that a form has an input like this:

[2] For readers familiar with Java, `param` is in instance of a class that implements the java.util.Map interface. The expression language handles the dot operator following the name of a map by treating the identifier after the dot as a key.

```
<input type="text" name="color">
```

The user's response could be displayed on a page using the following:

```
<c:out value="${param.color}"/>
```

This feature of the expression language provides a fix for a problem with the Java News Today site. Recall that the page title appears in `top.jsp`, which is shared by every page, but this title really should change to identify each page. This can be accomplished as follows:

```
<title>
  Java News Today: <c:out value="${param.title}"/>
</title>
```

Here, the parameter `title` will not come from a form but instead can be passed in through a variation of the `jsp:include` tag. For example, the index page will now include the top portion of the page with

```
<jsp:include page="top.jsp">
  <jsp:param name="title" value="Welcome!"/>
</jsp:include>
```

The availability of `param` also means that the bean isn't needed in Listing 4.4 at all! The whole page can be reduced to

```
<awl:maybeShow show="${param.shouldShow}">
  The time is:
  <awl:date format="hh:mm:ss MM/dd/yy"/>
</awl:maybeShow>
```

Likewise, the calculator could do without its bean, reducing the page to

```
The sum is:
<c:out value="${param.value1 + param.value2}"/>
```

Errors to Watch For

Most of the comments about possible errors when using the `jsp:getProperty` tag also apply to `c:out` and other tags that use expressions. In particular, trying to reference a property that the bean does not possess will result in an error. In addition, trying to reference a bean that does not exist, such as `c:out value="${someBean.someProperty}"` if `someBean` has not been loaded, will not result in an error but simply in nothing being displayed. This can result in problems that may be difficult to find and fix, for example, if the name of a bean is simply misspelled.

Because `c:out` acts like an enhanced version of `jsp:getProperty`, it is not surprising that an equivalent of the `jsp:setProperty` tag is in the standard library. This tag, `c:set`, looks like this:

```
<c:set
    target="bean"
    property="property name"
    value="property value"/>
```

This tag sets the property called ***`property name`*** in the bean identified as `bean name` to `value`. Unlike the `jsp:setProperty` tag, the `c:set` tag can not set all the properties in a bean at once by using the special property `*`. However, each of the parameters to `c:set` may be a script, which allows properties to be set with dynamic values.

Errors to Watch For

When using the `c:set` tag, it is very important to mind the distinction between something like `target="bean"` and `target="${bean}"`. The former is a name that has no properties; the latter is a bean obtained from the name by the expression language. This can be a natural source of confusion, as the `jsp:setProperty` tag does use the name. Even if the reason is not completely clear at this point, remember that the target should always take an expression, not simply a name.

4.5 Formatting Output

Consider once again the calculator from Listing 4.6. If the user enters large numbers say,

1264528 and 9273912 the sum will be 10538440. This is certainly the right answer, but it is not in a particularly readable format. It would be much better if it could be displayed as 10,538,440. The issue of formatting comes up frequently when designing Web pages, as there are often particular rules about how numbers, currencies, and dates should be displayed.

Another portion of the standard library provides tags for displaying formatted values. This library can be imported by using

```
<%@ taglib prefix="fmt"
    uri="http://java.sun.com/jstl/fmt" %>
```

Once loaded, a number of new tags are available, some of which work similarly to the `c:out` tag but allow a format to be specified. For example, the `c:out` tag from Listing 4.6 could be replaced with

```
<fmt:formatNumber
  value="${calc.value1 + calc.value2}"
  pattern="###,###"/>
```

The `pattern` indicates that there should be a comma after every three digits. It would also be legal to provide a decimal point, as in `###,###.##`, which would indicate that a comma should be placed every three digits, with two digits following the decimal point.

A number of examples have contained custom mechanisms for formatting dates. Now a more general solution to this problem is available: the `fmt:formatDate` tag. This tag works very much like `fmt:formatNumber` tag but expects its value to be a date. If the bean from Listing 3.1 were loaded with `jsp:useBean`, the `date` property could be formatted with

```
<fmt:formatDate
  value="${bean1.date}"
  pattern="hh:mm:ss MM/dd/yy"/>
```

The valid expressions for `pattern` can be found in the documentation for the `java.text.SimpleDateFormat` class, but note for the moment that any of the expressions from Listing 3.2 would work.

The formatting tags can do a great deal more than has been shown here. Different countries have different standard ways to express numbers and dates, and the format tags can ensure that data is formatted in an appropriate way for each country, through a mechanism called *internationalization*. The format tags can also be used to parse values, which would allow the calculator to accept inputs with commas and decimal points. These topics are beyond the scope of this book, but now that the basic functionality of these tags is clear, interested readers can see the remaining details in section 9 of the JavaServer Pages Standard Tag Library specification, which is available from http://java.sun.com/products/jsp/.

4.6 Compound Data in the Expression Language

Up until now, all the bean properties have have been simple types: strings of text or numbers. This feature of the examples that have appeared is not a fundamental restriction on beans themselves. Beans can contain *compound values* as well.

Compound values, as the name implies, contain multiple pieces of data. If this sounds familiar, it should; beans themselves hold multiple pieces of data. Indeed, beans can contain other beans, which can contain yet other beans, and so on, indefinitely.

As an example of how this might be useful, consider how a bean would be used to model a home entertainment system, which may contain many individual components, such as an amplifier, CD player, cassette player, and radio tuner. The system as a whole may have certain properties, such as which component is currently playing and the overall color and size of the system. In addition, each individual component has its own set of properties. The CD player has a property representing the name of the disc it currently contains, the tuner has a property indicating the station it is currently tuned to, and so on.

It would in principle be possible to give all the properties of the components to the system as a whole, but this is bad design. It is much better to *encapsulate* logical units as separate beans. This design allows more complex beans to be constructed incrementally by using the individual building blocks, in the same way that using the `jsp:include` tag allows complex pages to be built up from smaller ones.

Given the home entertainment system bean, there is no way in which the `jsp:getProperty` tag could be used to determine the name of the CD in the CD player. The whole CD player bean could be obtained with

```
<jsp:getProperty
  id="homeEntertainment"
  property="cdPlayer"/>
```

However, this will display the whole CD player bean. Beans as a whole have no standard representation; this might display as something cryptic, such as `com.awl.ch04.jspbook.CdBean10b053`, or it might display as a list of all the properties of the bean or anything else that the bean programmer has chosen. In any case, it is unlikely to display only the name of the current disc. What is needed is a way to traverse a set of compound data. Fortunately, the expression language provides a mechanism to do this.

As discussed previously, within the expression language, a single dot between two names indicates that the name on the left should be a bean and the name on the right a property. This extends in a natural way; if a property is itself a bean, it is legal to add another dot followed by the name of a property within that bean, and so on. Getting the name of CD from a CD player within a home entertainment system would therefore look something like

```
${homeEntertainment.cdPlayer.currentDisk}
```

The meaning of the multiple dots in Listing 4.7 should make more sense now. An object called `pageContext` holds information about the page currently being generated. Within this object is another object, called `request`, which holds information pertaining to the request being processed. Finally, the `request` object has such data as the name of the local computer, the remote computer, and so on.

4.6.1 Repeating a Section of a Page

Another important kind of compound data is a collection of an arbitrary number of values. A CD has a number of tracks, but as this number is different for different CDs, a CD bean cannot simply have a different property for each track. Similarly, a shopping cart bean will contain a number of items, but this number will change as the bean is used.

Java has many ways to manage collections of varying size, but the simplest is called an *array*. Arrays are lists of objects of the same type, such as arrays of strings, arrays of numbers, and arrays of CDs. Within these arrays, items are referenced by a number called the *index*, starting with 0.

The expression language makes it possible to pull a particular element out of such an array by placing its index within brackets. Obtaining the first track of a CD could be done with an expression like this:

```
${cd.tracks[0]}
```

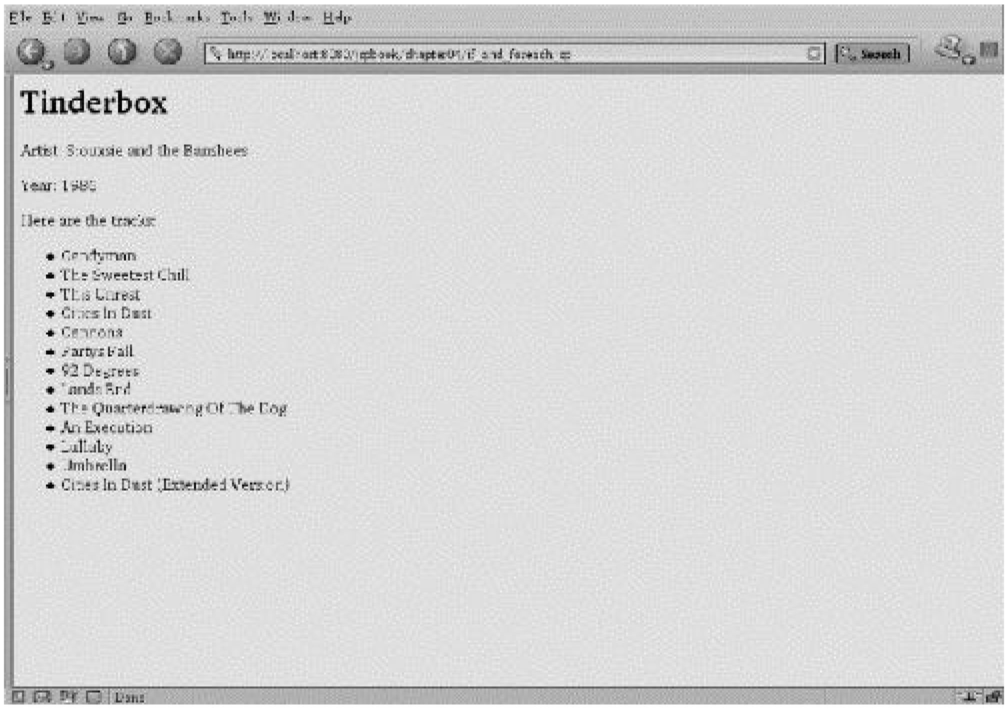
Again, note how this logically follows from the way properties work: `${cd.tracks}` would return the entire array; following this with `[0]` pulls out a particular element from that array.

Errors to Watch For

If a request is made for an index beyond the number of elements in the array, the result will be empty.

It is unusual to need to access a particular element in an array; it is more common to need to repeat some action for every element, regardless of how many there are. This process is known as *iteration*, and it should come as no surprise that a tag in the standard library handles it: `jsp:forEach`. Recall that Listing 3.13 obtained information about a CD from a serialized bean. At that point, however, there was no way to list the tracks, because the page could not know in advance how many there would be. Listing 4.8 uses the `c:forEach` tag to solve this problem, and the resulting page is shown in Figure 4.2.

Figure 4.2. Iteration used to display every element in an array.



Tinderbox

Artist: Siouxsie and the Banshees

Year: 1986

Here are the tracks:

- Candyman
- The Sweetest Chill
- This Unrest
- Cities In Dust
- Cannons
- Partys Fall
- 92 Degrees
- Lands End
- The Quarterdrawing Of The Dog
- An Execution
- Lullaby
- Umbrella
- Cities In Dust (Extended Version)

Listing 4.8 The `forEach` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
 id="album"
 beanName="tinderbox4"
 type="com.awl.jspbook.ch04.AlbumInfo"/>

<h1><c:out value="${album.name}"/></h1>

Artist: <jsp:getProperty name="album"
           property="artist"/><p>
Year: <c:out value="${album.year}"/></p>

Here are the tracks:
<ul>
<c:forEach items="${album.tracks}" var="track">
  <li><c:out value="${track}"/>
</c:forEach>
```

```
</ul>
```

The `c:forEach` tag takes a number of parameters. The first is the items to iterate over, which is specified by a script. The second is a name to use as a variable; within the body of `c:forEach`, this variable will be set to each element in the array in turn. This variable can be accessed by the expression language as a bean, which means, among other things, that the `c:out` tag can be used to display it.

Errors to Watch For

If something other than an array is used as the `items` parameter, the `c:forEach` tag will treat it as if it were an array with one element.

4.6.2 Optionally Including Sections of a Page

Iteration allows a page to do one thing many times. The other major type of control a page may need is determining whether to do something at all. The custom `awl:maybeShow` tag introduced at the beginning of this chapter handled a limited version of that problem, but the standard tag library provides a number of much more general mechanisms, called collectively the *conditional* tags. The most basic of these tags is called `c:if`.

In its most common form, the `c:if` tag takes a single parameter, `test`, whose value will be a script. This script should perform a logical check, such as comparing two values, and facilities are provided to determine whether two values are equal, the first is less than the second, the first is greater than the second, and a number of other possibilities. Listing 4.9 shows how the `c:if` tag can work with a bean to determine whether to show a block of text.

Listing 4.9 The `if` tag

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
  id="form"
  class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>
```

```
<c:if test="${form.shouldShow == 'yes'}">
 The time is:
 <awl:date format="hh:mm:ss MM/dd/yy"/>
</c:if>
```

Note the expression in the script for the `test` parameter. Two equal signs, `==`, are used to check two values for equality. Here, the first value comes from a property and is obtained with the normal dotted notation. The second value, `yes`, is a constant, or *literal*, which is reflected by the single quotes around it in the script. If these quotes were not present, the expression language would look for a bean called `"yes"`; as no such bean exists, the result would be an error.

Listing 4.9 is similar to Listing 4.3; the major difference is that Listing 4.9 uses the standard tag instead of the custom `awl:maybeShow`. The downside is that the `c:if` tag cannot reverse a block of text; all it can do is decide whether to include its body content in the final page.

This may seem like a shortcoming but in fact reflects a good design pattern. Note that `awl:maybeShow` does two completely unrelated things: checks whether a value is `yes`, `no`, or `reverse` and reverses a block of text. Rather than making one tag do two things, it is better to have two different tags. According to the so-called UNIX philosophy of software, each piece of code should do only one thing and do it well, and there should be easy ways to knit these small pieces together. For tags, this means that each tag can be used independently or combined with other tags. In this case, if an `awl:reverse` tag did nothing but reverse its body content, it could be combined with the `c:if` tag to do the same thing as Listing 4.3. This is shown in Listing 4.10.

Listing 4.10 Splitting tags

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="awl"
    uri="http://jspbook.awl.com/samples" %>
<jsp:useBean
   id="form"
   class="com.awl.jspbook.ch04.FormBean"/>
<jsp:setProperty name="form" property="*"/>
```

```
<c:if test="${form.shouldShow == 'yes'}">
 The time is:
 <awl:date format="hh:mm:ss MM/dd/yy"/>
</c:if>

<c:if test="${form.shouldShow == 'reverse'}">
 <awl:reverse>
   The time is:
   <awl:date format="hh:mm:ss MM/dd/yy"/>
 </awl:reverse>
</c:if>
```

Note that two `c:if` tags are used here: one to check whether the value is `yes` and another to check whether it is `reverse`. The body content of both of these tags is the same, which is rather wasteful. It means that if the body ever needs to change, it will need to be modified in two places in order to keep everything consistent. It would be better in this case to put the body in a separate file and then have both of the `if` tags include that file with a `jsp:include` tag. Now that the functionality of `awl:maybeShow` has been divided into two pieces, the `c:if` tag can be used for many other things, and the `awl:reverse` tag can be used to reverse unconditionally a block of text, should such a thing ever be useful.

Listing 4.10 imports two tag libraries: the standard one, which is installed as `c` and provides the `c:if` tag, and the custom one installed as `awl`, which provides the `awl:reverse` tag. This is perfectly valid; often a page will need many different tags from different libraries, and it will then need to import all of them. The only catch is that each tag library must be given a different prefix.

4.7 Browser Detection

Web programmers face many difficult decisions, not the least of which is how to deal with the fairly horrible state of modern browsers. A popular Web site is likely to receive requests from versions of Internet Explorer 3 through 6, Mozilla, Netscape 4.7, Opera, various AOL browsers, and numerous custom browsers now available in consumer devices, such as phones and PDAs (personal digital assistants). Each of these is likely to render HTML slightly differently, support different media types, and handle JavaScript differently, if at all.

One way of dealing with this variability is to use the "lowest common denominator," that is, only those features that are supported and work the same in every browser. This makes things easier for the Web developer but means that the user will be getting a site that looks like something from the early 1990s, which may disappoint many users.

Alternatively, Web developers may design a site for one browser perhaps Mozilla 1.0 a
nd put up a note encouraging other users to switch to this browser. This is likely to infuriate many users who either don't want to or can't change browsers simply to get to one site.

Finally, developers can create parallel versions of all the browser-specific HTML and JavaScript and so on and send out the appropriate version, based on which browser is being used. The browser makes this possible by identifying itself with every request, and JSPs make this possible through the conditional tags. A skeleton of code that accomplishes this is shown in Listing 4.11.

Listing 4.11 Browser detection

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<jsp:useBean
  id="browser"
  class="com.awl.jspbook.ch04.BrowserBean"/>

<c:set
  target="${browser}"
  property="request"
  value="${pageContext.request}"/>

You are using a browser that identifies itself as
<c:out value="${browser.fullName}"/><p>

<c:if test="${browser.type == 'Gecko'}">
... include Mozilla code here ...
</c:if>

<c:if test="${browser.type == 'MSIE'}">
```

```
... include IE code here ...
</c:if>
```

This example uses `BrowserBean`, a utility bean that extracts browser information from the request. In order to obtain this information, `BrowserBean` must have access to the `request` object. This object is obtained from the `pageContext`, as was done in Listing 4.7, and passed with a `c:set` tag to the bean.

A bean such as `BrowserBean` is needed for two reasons: first, because the browser name is not available as a simple property, such as the ones shown in Listing 4.7; second, because the full name of the browser is likely to be something unwieldy, such as Mozilla/5.0 (X11; U; FreeBSD i386; en-US; rv:1.0rc3) Gecko/20020607, which contains information about the specific revision and operating system on which the browser is running. This is generally more information than needed to select the appropriate browser-specific code for a page. This second problem is solved by having the bean recognize major browser types, and it is this type that is used by the `c:if` tags.

4.8 Combining Tags

As mentioned previously, the bodies of JSP tags can contain anything, including other JSP tags. An example is the `c:out` tag within the `c:forEach` tag in Listing 4.8. To demonstrate this further, the `c:if` and `c:forEach` tags work together in the following example.

If given an empty array, a `c:forEach` tag will not render its body content at all. This is fine but can lead to some odd-looking pages. In Listing 4.8, if the CD is empty, the page will display "Here are the tracks" and then stop. This is technically correct but to the user may look as though the page stopped generating halfway through. It would be better to inform the user that the CD is empty rather than to display a list with no elements. This can be accomplished by putting the `c:forEach` tag inside a `c:if` tag, as shown in Listing 4.12.

Listing 4.12 Tags working together

```
<%@ taglib prefix="c"
   uri="http://java.sun.com/jstl/core" %>
<jsp:useBean id="album" beanName="tinderbox4"
 type="com.awl.jspbook.ch04.AlbumInfo"/>
```

```
<h1><jsp:getProperty name="album" property="name"/></h1>

Artist: <jsp:getProperty name="album"
          property="artist"/><p>
Year: <jsp:getProperty name="album" property="year"/></p>

<c:if test="${empty album.tracks}">
There are no tracks! What a boring CD.
</c:if>

<c:if test="${!(empty album.tracks)}">
  Here are the tracks:
  <ul>
  <c:forEach items="${album.tracks}" var="track">
    <li><c:out value="${track}"/>
  </c:forEach>
  </ul>
</c:if>
```

Conceptually, the only new thing about this example is the check that is done in the `c:if` tag. The `empty` in the test checks whether the named property exists,[3] and if it does exist and is an array, whether it has any elements. The exclamation point in the test should be read as "not." It means that if the following test would be true, it returns `false`, and vice versa.

[3] Technically, it tests whether the value equals `null`, as will be discussed in Chapter 9.

4.9 Selecting among Multiple Choices

Once again, the preceding example had to use two `c:if` tags, although the bodies are different in this case. However, this is still somewhat clumsy, as the same check is being performed twice: once to see whether it is true and once to see whether the reverse is true. This double check is needed because the `c:if` tag is capable of deciding only between two alternatives: to include its body or not to include it. Another set of tags allows *multiway branching*, or choosing from among several mutually exclusive possibilities. Unlike the other tags seen so far, three tags work together to obtain the desired result. The outermost tag, `c:choose`, has no parameters; it merely serves as a container for a collection of two other tags: `c:when` and `c:otherwise`. Each individual `c:when` tag acts a lot like a `c:if` tag. Both tags take a parameter called `test`, which should be a script, and render their body content if the condition in the script is `true`. The difference is that multiple `c:if` tags will each be checked in turn, whereas a `c:choose` tag will stop after finding the first `c:when` tag with a `test` that is `true`.

In other words, consider a set of possible values for a bean property, such as the colors red, green, and blue. The following snippet of code would check each of these possibilities regardless of the value:

```
<c:if test="${bean.color == 'red'}">...</c:if>
<c:if test="${bean.color == 'green'}">...</c:if>
<c:if test="${bean.color == 'blue'}">...</c:if>
```

The following snippet will check whether the color is red; if so, it will stop and will not then have to check whether it is green and then blue:

```
<c:choose>
  <c:when test="${bean.color == 'red'}">...</c:when>
  <c:when test="${bean.color == 'green'}">...</c:when>
  <c:when test="${bean.color == 'blue'}">...</c:when>
</c:choose>
```

Clearly, the second option is more efficient. In addition, using the `c:choose` tag groups related code in one place and so makes JSPs easier to read and understand.

The `c:choose` tag works with another tag: `c:otherwise`. This tag also has no parameters; its body will be evaluated if none of the `c:when` tags has a `true` condition.

It is now clear how it would be possible to avoid doing the check twice in Listing 4.11 b

y using one `c:when` and a `c:otherwise` rather than by using two `c:if` tags. This is

shown in Listing 4.13.

Listing 4.13 The `choose` tag

```
<c:choose>
  <c:when test="${empty album.tracks}">
```

```
    There are no tracks! What a boring CD.
  </c:when>

  <c:otherwise>
    Here are the tracks:
    <ul>
    <c:forEach items="${album.tracks}" var="track">
      <li><c:out value="${track}"/>
    </c:forEach>
    </ul>
  </c:otherwise>
</c:choose>
```

This code is a little more verbose than Listing 4.12 but has the advantage of avoiding one redundant `test`. Using the `c:choose` tag also makes it clear that the conditions are mutually exclusive, and hence only one of the bodies will ever be rendered.

Chapter 2 briefly mentions the `jsp:forward` tag, which sends the user from one page to another. This tag can be combined with the `c:choose` tag to provide a type of control called *dispatching*, whereby one page determines where the appropriate content lives and sends the user to that page. This is illustrated in Listing 4.14.

Listing 4.14 Using the `choose` tag as a dispatcher

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<c:choose>
  <c:when test="${param.whichPage == 'red'}">
    <jsp:forward page="red.jsp"/>
  </c:when>

  <c:when test="${param.whichPage == 'green'}">
    <jsp:forward page="blue.jsp"/>
  </c:when>

  <c:when test="${param.whichPage == 'blue'}">
    <jsp:forward page="blue.jsp"/>
  </c:when>

  <c:otherwise>
    <jsp:forward page="select_page.jsp"/>
  </c:otherwise>
</c:choose>
```

This page looks for a form parameter, `whichPage`, which should be `red`, `green`, or `blue`, and, based on this value, sends the user to one of three pages. If no value has been provided, the `otherwise` tag forces the user to `"select_page.jsp"`, which contains the form to be filled out.

4.10 Summary and Conclusions

Now we're cooking! This chapter introduced the concept of custom tags and the standard tag library now part of the JSP specification. These tags give page authors full control over what the user ends up seeing, by providing the means to show arbitrary values, repeat a section of a page as many times as needed, conditionally remove a section of page, or choose from many possible sections. In the next chapter, we'll see how these tags, together with beans, allow Java News Today to build its site.

4.11 Tags Learned in this Chapter

`c:forEach` Repeats a section of the page for every item in an array
Parameters:
  `items`: An expression specifying the array to use, most likely a bean property
  `var`: The name of the variable with which each element in the array will be referred
Body:Arbitrary JSP code

`c:out` Displays a value
Parameters:
  `value`: An expression to be evaluated and displayed

Body: Arbitrary JSP code; the body content will be displayed if `value` is `null`

`c:if` Conditionally include a portion of the page
Parameters:
`test`: An expression that should be a logical test of a property
`var`: If present, names a variable where the result of the expression will be stored
Body: Arbitrary JSP code

`c:choose` Includes one of several portions of a page
Parameters: None
Body: Arbitrary number of `c:when` tags and, optionally, one `c:otherwise` tag

`c:when` One possibility for a `c:choose` tag
Parameters:
`test`: An expression that should be a logical test of a property
Body: Arbitrary JSP code

`c:otherwise` The catch-all possibility for a `c:choose` tag. If none of the expressions in the `c:when` tags evaluates to `true`, the body of the `c:otherwise` will be included.
Parameters: None
Body: Arbitrary JSP code

`c:set` Set a property in a bean
Parameters:
`target`: The name of a bean
`property`: The property within the bean to set
`value`: The value to assign; may be a script
Body: None

`fmt:formatNumber` Format a number for output
Parameters:
`value`: The value to be formatted; may be a script
`pattern`: A pattern specifying how the number should be formatted
Body: None

`fmt:formatDate` Format a date and/or time for output

Parameters:

`value`: The value to be formatted; may be a script

`pattern`: A pattern specifying how the date should be formatted

Body: None

Chapter 5. Java News Today: Part I

Armed with the power of beans, the expression language, and the standard tag library, Java News Today is at last ready to start putting its site together in earnest. In order to do so, JNT will need to decide what functionality the site will offer, design the beans that will represent the entities they will be dealing with, and build pages to provide that functionality. This chapter looks at each of these steps.

5.1 The Beans

Generally the first step of any large project, *data modeling*, consists of deciding what data the system will need to maintain, how this data will be represented, and how it interrelates. Traditionally, such modeling takes place in the context of a database, discussed in Chapter 6. For current purposes, however, it is reasonable to model everything in terms of beans. As there is not yet anywhere to store all the data, the examples in this chapter use beans in which the data has been hard-coded, although this is never a good idea in real-world projects. Even when prototyping a system, it is better to use a small, simple database. However, this little cheat will not significantly change the way the pages work, so it will suffice for now.

Java News Today has already identified a few beans it will need. In Chapter 3, JNT developed the `QuizBean`, which holds the question, options, and correct answer for the daily quiz, and created a `UserInfoBean` to hold users' preferences for colors, as well as a name. At this time, JNT is ready to consider allowing users to register on the site, in order to store their preferences permanently. This will necessitate adding some logic to the `UserInfoBean` in order to handle logging users on the system. The fields added will be `username`, `password`, and `isLoggedIn`, a flag that will be `true` if the user is currently logged in and `false` otherwise.

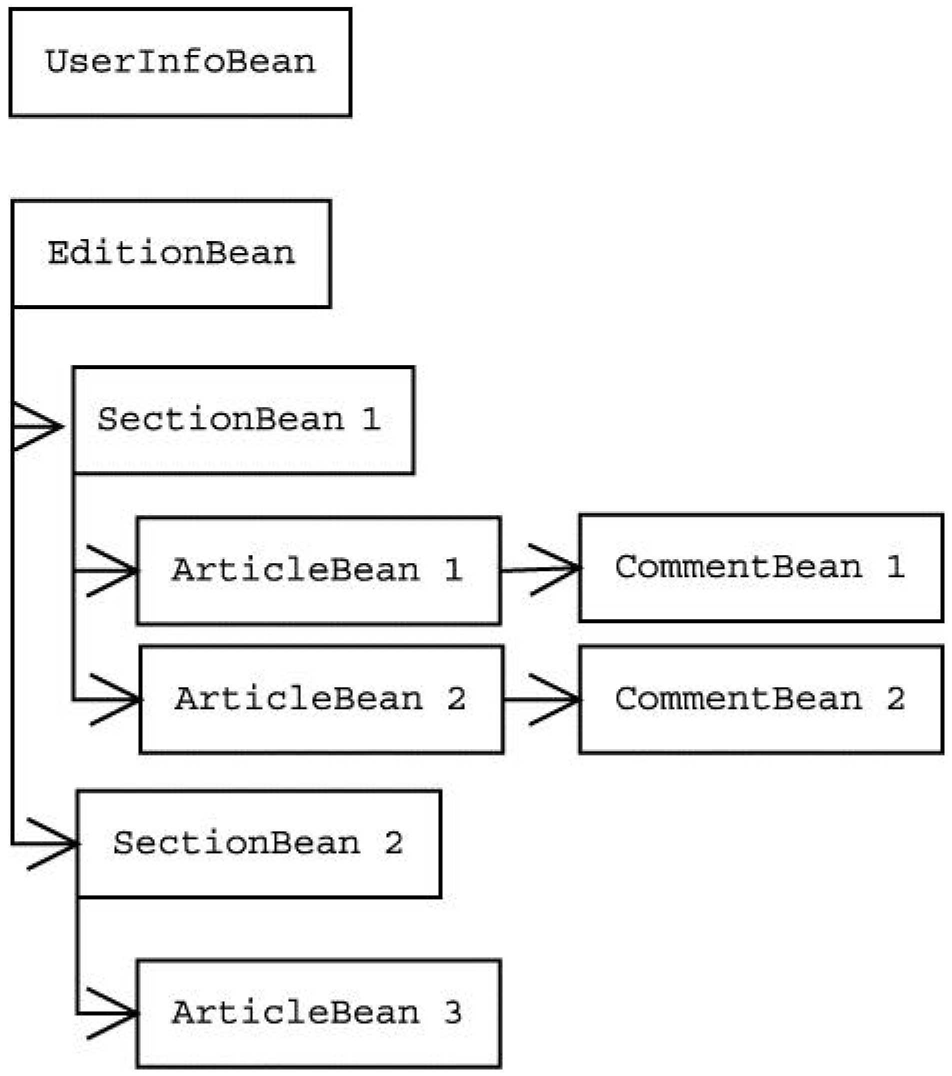
In addition to users and quizzes, the other major entities behind the JNT site are articles. An `ArticleBean` will hold the text of the story, a headline, and a date and time of publication. Each `ArticleBean` will also have a unique numeric identifier to identify and load that story.

As with a physical newspaper, articles will be grouped into major sections covering broad categories, such as J2EE, the Java community, related technologies, and so on. Each section will have a name and a description and will also keep track of all the articles it contains. This containment will be managed by creating a `SectionBean` and giving each instance of `SectionBean` an array of `ArticleBean` instances. This should immediately suggest the use of the `jsp:forEach` tag to display all the stories in a section, and indeed such a page will be on the site. This illustrates how the data-modeling phase of a project can inspire and affect the page-designing phase.

Similarly, sections will be grouped into an edition. At the moment, an edition will have only an array of sections; later, its role will be expanded to manage many of the personalization options Java News Today will offer.

Finally, in order to make this a community site, the staff at Java News Today would like to allow users to add comments to stories. Some sites, notably http://www.slashdot.org, provide very sophisticated commenting systems that can include threaded discussions, moderation of comments, and a whole host of other features. For the moment, JNT will allow only a simple "flat" commenting system, whereby comments simply appear in the reverse order they were added. This suggests the need for a `CommentBean` and an array of such beans held by each `ArticleBean`. Figure 5.1 shows a sample of beans as they might exist in memory and their relationships to one another.

Figure 5.1. The JNT beans.



5.2 The Header

The header is the site's simplest component, as its only job is to display a banner with the title of the page, along with the user's name as added in Chapter 3. In order to clean it up a little, the header will be modified to show only the user's name if the user has logged in. The resulting page is shown in Listing 5.1.

Listing 5.1 The header

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
```

```
<jsp:useBean id="user5"
 class="com.awl.jspbook.ch05.UserInfoBean"
 scope="session"/>

<center>
  <h2>
    Java News Today: <c:out value="${param.title}"/>
  </h2>
</center>

<c:if test="${user5.isLoggedIn}">
  <div class="left">
    Hello <c:out value="${user5.name}"/>!
  </div>
</c:if>
```

This is about as simple a page as one could hope for. It loads a bean, checks a property with the `c:if` tag, and displays a value with the `c:out` tag.

The bean is loaded from the session scope because the user information should remain active as long as the user is active on the site. If this bean were in the request scope, the user's preferences would need to be reloaded —or worse, would be lost completely —on every new page. If the bean were in the application scope, every user would share the same data, which would not allow each user to have different options. Thus, session scope is definitely the right place for this bean.

Note that the test simply checks the value of a property. The test does not need to check whether the `${user.5isLoggedIn == true}isLoggedIn` property itself returns `true` or `false` directly.

5.3 The Left-Hand Navigation

The left side of the page has thus far contained only the daily quiz, which was developed in Listing 3.13, and a link to the customization page. This will now be enhanced by the addition of a login box from which the user can log in. This will also be a simple form, but in the interest of encapsulation, it will be placed in its own file and included with a `jsp:include`. Listing 5.2 shows the new login form.

Listing 5.2 The login form

```
<form action="login_result.jsp" method="post">
Username:<br>
<input type="text" name="username" size="8"><br>
Password:<br>
<input type="text" name="password" size="8"><br>
<input type="submit" name="Login" value="Login">
</form>
```

This file contains no beans, scripts, or special tags, which should come as no surprise. There have already been many examples of forms that provide values to beans, and in all these cases, the forms themselves need not know anything about the beans, as all the action happens on the receiving page, where the form values are loaded into a bean with the `jsp:setProperty` tag. The only requirement for this to work is that the bean's properties must be called `username` and `password`. Because these names were chosen in the data-modeling phase, both the author of this form and the author of the `UserInfoBean` will know to use those names.

The implementation of the `UserInfoBean` used in this chapter knows about one user whose `username` and `password` are both `"test"`, so those are the values to enter into the login form when exploring the examples on the CD-ROM.

The other necessary element of the left-hand navigation is the list of sections available in the current edition. The designer for the site would like an asterisk next to the current section so that the user will always know where in the site he or she currently is. The section list will also be placed in a separate file for easy manipulation; this file is shown in Listing 5.3.

Listing 5.3 The list of sections

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="edition"
  beanName="jnt"
  type="com.awl.jspbook.ch05.EditionBean"/>

<jsp:useBean
  id="currentSection"
  class="com.awl.jspbook.ch05.SectionBean"/>

<jsp:setProperty
  name="currentSection"
  property="sectionId"/>

<c:forEach items="${edition.sections}" var="section">
  <c:if
   test="${currentSection.sectionId == section.sectionId}">
   *
  </c:if>

  <a href="<c:url value="section.jsp">
   <c:param name="sectionId" value="${section.sectionId}"/>
  </c:url>"><c:out value="${section.name}"/></a><br>
</c:forEach>
```

This example is somewhat more complicated, so let's go through it line by line. The first line loads a tag library, and the next two lines load beans. The `EditionBean` will hold the list of sections, as decided in the data-modeling phase. This bean will live in the session scope for the same reasons that the `UserInfoBean` does.

The `SectionBean` will hold the currently selected section, information that will be needed in order to put the asterisk in the right place in the list. In order to know what the current section is, this bean will need to be told, which is done by setting the `sectionId` property in the `jsp:setProperty` tag on the following line.

The next line starts an iteration with the standard `c:forEach` tag. Here, the items to iterate are the sections from the edition; the iteration variable is called `section`. Note that different names are used for the bean and the iteration variable in order to keep everything clear.

Next, a check is performed to determine whether to display the asterisk. This simple test for equality is handled by the `c:if` tag.

Now things get exciting! Displaying the name of the section would be easy enough using the `c:out` tag, as can be seen just before the closing `c:forEach` tag. However, this name needs to be turned into a link so that the user can select a section by clicking the name. This link will need to look something like the following:

```
<a href="???"><c:show value="${section.name}"/></a>
```

It is now necessary to determine what value should fill in the mystery spot in `href`. A page called section.jsp will enable the user to see all the stories in a section, so that page should be the destination. The only remaining question is how to pass along the information about which section was selected. If the section were selected via a menu or drop-down in a form, the `sectionId` would be passed along as a form variable. As it turns out, attaching a name and a value to a URL behaves exactly the same as using a form. In particular, the `jsp:setProperty` tag can load a bean with values passed in such a URL. Thus, the URL should look like this:

```
section.jsp?sectionId=<c:show
   value="${section.sectionId}"/>
```

This should explain how the `jsp:getProperty` tag at the top of this file will work. Clicking one of the links will go to the section page, passing along `sectionId=` with the section ID that was selected. The section page will include the section list page; when the `jsp:setProperty` tag is encountered, the section list page will grab the value from the URL. This may seem slightly weird, as this page is therefore sort of eating its own output. It may indeed be weird, but it is also a very common technique in Web development and is worth getting accustomed to it.

Although specifying the URL would work in most cases, a better approach uses a new tag from the standard library. The `c:url` tag builds a URL using the `value` parameter as the base page and appending the names and values from any `c:param` tags within the body. Using the `c:url` tag instead of manually constructing a URL has a number of advantages. One of the most important advantages is that the `c:url` tag will ensure that the resulting URL is valid. Certain characters, such as spaces and the equal sign, are not valid in names or values within URLs. The `c:url` tag will translate such characters into a legal representation automatically.

Now that the pieces are in place, the left-hand navigation bar itself is a straightforward enhancement of previous versions and is shown in Listing 5.4.

Listing 5.4 The left-hand navigation bar

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
```

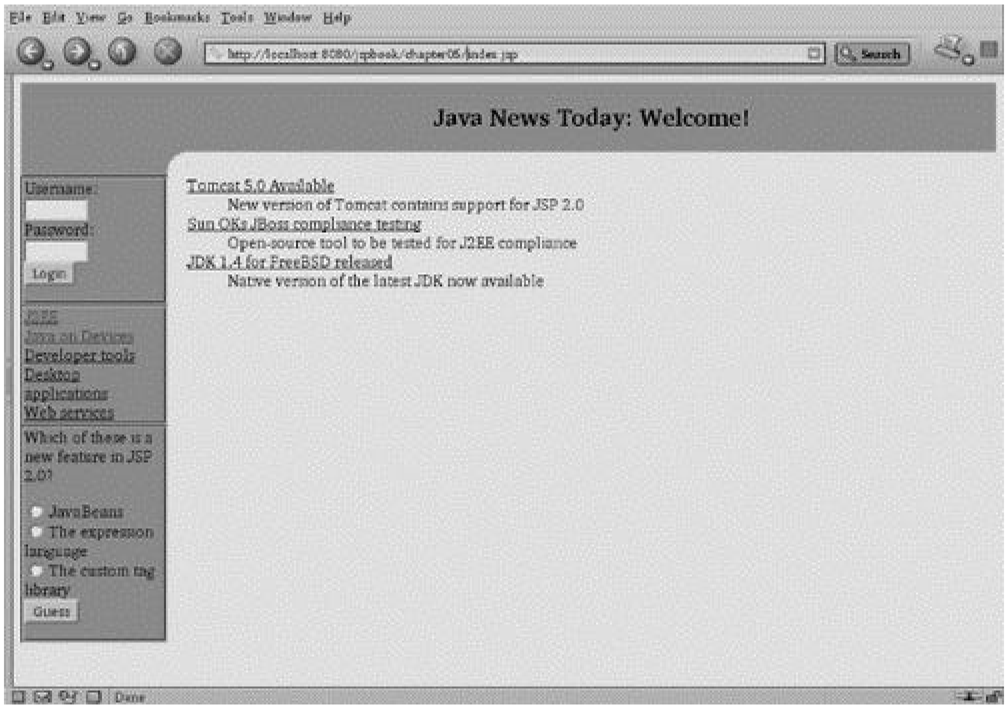
```
<jsp:useBean
  id="user5"
  scope="session"
  class="com.awl.jspbook.ch05.UserInfoBean"/>

<c:if test="${!user5.isLoggedIn}">
  <div class="bordered">
    <jsp:include page="login.jsp" flush="true"/>
  </div>
</c:if>

<div class="bordered">
<jsp:include page="section_list.jsp" flush="true"/>
</div>

<div class="bordered">
<jsp:include page="quiz.jsp"/>
</div>

<c:if test="${user5.isLoggedIn}">
  <div class="bordered">
    <a href="customize.jsp">Customize JNT</a>
  </div>
</c:if>
```

The page imports the standard tag library and loads the `UserInfoBean`. This bean is used to hide the login form if the user is already logged in and, if the user is logged in, to display the customization link.

It is somewhat a matter of personal preference whether the check for the login form should be done here or in login.jsp. The advantage to putting it in login.jsp is that all the login-related logic is in one file. However, doing so would mean that there would be no way to override the decision not to display the login form.

In general, this sort of decision should he handled by the controller layer instead of the view. This issue will be revisited in Chapter 12 when controllers are discussed in more detail. The new home page with all the new navigation elements is shown in Figure 5.2.

Figure 5.2. The new JNT home page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter05/index.jsp
Search
Login
Guess
Done

Java News Today: Welcome!

Username:

Password:

J2EE
Java on Devices
Developer tools
Desktop applications
Web services

Which of these is a new feature in JSP 2.0?

JavaBeans
The expression language
The custom tag library

Tomcat 5.0 Available
New version of Tomcat contains support for JSP 2.0
Sun OKs JBoss compliance testing
Open-source tool to be tested for J2EE compliance
JDK 1.4 for FreeBSD released
Native version of the latest JDK now available

5.4 The Login Page

Now that a form has been provided so users can log themselves in on the system, there needs to be a page that will perform the necessary actions. The page that does this is shown in Listing 5.5.

Listing 5.5 The login handler

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="user5"
  scope="session"
  class="com.awl.jspbook.ch05.UserInfoBean"/>

<jsp:setProperty name="user5" property="*"/>
<jsp:setProperty
  name="user5"
  property="login"
  value="true"/>

<jsp:include page="top.jsp">
  <jsp:param name="title" value="Login"/>
</jsp:include>

<c:choose>
  <c:when test="${user5.isLoggedIn}">
    You have sucessfully logged into Java News Today!<p>
    Click <a href="jntindex.jsp">here</a> to proceed to
    your custom edition.
  </c:when>
  <c:otherwise>
    We're sorry, we were unable to log you in. Perhaps you
    mistyped your username or password; use the form on
    the left to try again.
  </c:otherwise>
</c:choose>

<jsp:include page="bottom.jsp"/>
```

This page begins with the usual things, including loading the `UserInfoBean`. The bean's properties are then set: the `username` and `password` from the login form in Listing 5.2. Part of the `UserInfoBean`'s job as the model of users is to provide a mechanism that logs a user in on the system, given the `username` and `password`. This mechanism is triggered by setting the `login` property of the bean, which will cause the bean to check these values against a list of all users in the system; if a match is found, the `isLoggedIn` property will be set to `true`.

This setting of properties has to be done *before* the page top is included. If it were done afterward, the user's `isLoggedIn` property would still be `false` during processing of the header and navigation, and consequently the name would not be shown and the login form would.

The rest of the page is pretty anticlimactic: another `c:choose` tag used to determine whether the login succeeded and to display an appropriate message in either case.

5.5 The Quiz Result Page

Chapter 3 introduced the daily quiz that Java News Today will use to liven up its site. The quiz consists of a serialized bean that holds the questions and correct answer, along with a form from which the user can guess, as shown in Listing 3.13. At that point, there was no way to check whether the user was correct, but that's easily remedied now that we have the standard tag library at our disposal. The quiz result page, shown in Listing 5.6, it is very straightforward.

Listing 5.6 The quiz result page

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="quiz"
  beanName="todaysQuiz3"
  type="com.awl.jspbook.ch03.QuizBean"/>

<jsp:setProperty name="quiz" property="*"/>

<jsp:include page="top.jsp" flush="true">
  <jsp:param name="title" value="Quiz result"/>
</jsp:include>

<c:choose>
  <c:when test="${quiz.userGuess == quiz.correctAnswer}">
    That's right!<p>
  </c:when>
  <c:otherwise>
    Sorry, that's incorrect; the right answer is
    <c:out value="${quiz.correctAnswer}"/><p>
  </c:otherwise>
</c:choose>
```

```
<jsp:include page="bottom.jsp" flush="true"/>
```

This is another example of setting bean properties from a form and then checking a condition with a `c:choose` tag.

5.6 The Section Page

Listing 5.3 showed how the section page will be called from the list of available sections and how this page will be passed a `sectionId` as if a form had sent it. This means that it will be possible to use a bean and a `jsp:setProperty` to tell that bean which section was selected, just as was done in the section list to place an asterisk in front of the current section.

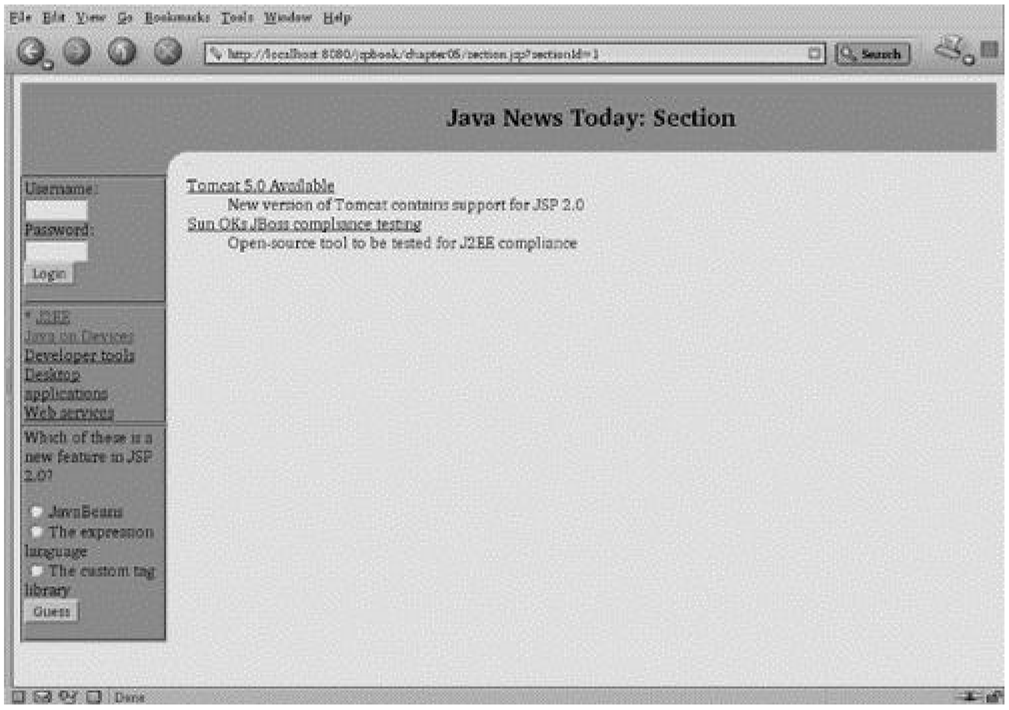
If the section bean is designed to load up all the stories in a section when the `sectionId` property is set, all that is necessary to build the section page is to iterate the available articles with a `c:forEach` tag. That is exactly what Listing 5.7 does.

Listing 5.7 The section page

```
<jsp:useBean
  id="currentSection"
  class="com.awl.jspbook.ch05.SectionBean"/>

<jsp:setProperty
  name="currentSection"
  property="sectionId"/>
<dl>
  <c:forEach items="${currentSection.articles}"
            var="article">
    <dt><a href="<c:url value="article.jsp">
    <c:param name="articleId"
            value="${article.articleId}"/>
    </c:url>"><c:out value="${article.headline}"/></a>
    <dd><c:out value="${article.summary}"/>
  </c:forEach>
</dl>
```

If this looks very similar to the section list from Listing 5.3, it should! They both do essentially the same thing; the only significant difference is that the items in this example are in a definition list instead of an unordered list. In particular, the `c:url` tag is used in both. Figure 5.3 shows how the section page looks in a browser.

Figure 5.3. The JNT section page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter05/section.jsp?sectionId=1
Search
Username:
Password:
Login
Guess
Done

Java News Today: Section

Tomcat 5.0 Available
New version of Tomcat contains support for JSP 2.0
Sun OKs JBoss compliance testing
Open-source tool to be tested for J2EE compliance

* J2EE
Java on Devices
Developer tools
Desktop applications
Web services

Which of these is a new feature in JSP 2.0?

JavaBeans
The expression language
The custom tag library

5.7 The Article Page

The article page consists of two pieces: the contents of the article and the comment region, which allows users to comment on stories and read others' comments. The first portion is even simpler than the section page, as it need only display the contents of a few properties from the `ArticleBean`, as shown in Listing 5.8.

Listing 5.8 The article page

```
<jsp:useBean
  id="article"
  class="com.awl.jspbook.ch05.ArticleBean"/>
```

```
<jsp:useBean
  id="user5"
  scope="session"
  class="com.awl.jspbook.ch05.UserInfoBean"/>

<jsp:setProperty name="article" property="articleId"/>
<h2><c:out value="${article.headline}"/></h2>
<i>Posted by <c:out value="${article.author}"/>
at
<fmt:formatDate
  value="${article.time}"
  pattern="MM/dd/yy hh:mm"/>
</i><p>

<c:out value="${article.contents}"/>
```

Note the use of the `fmt:formatDate` tag from the previous chapter to format the date. The comment portion appears below the article contents and shows the list of available comments, along with a form to add an additional one, as shown in Listing 5.9.

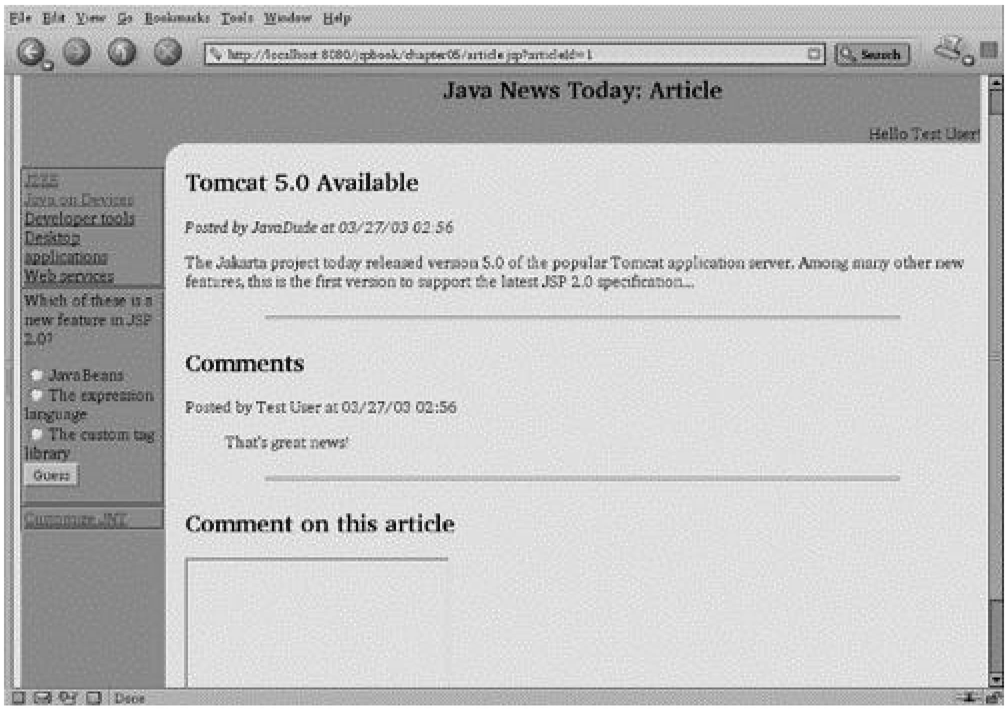
Listing 5.9 The comment section

```
<hr width="80%">
<h2>Comments</h2>

<c:forEach items="${article.comments}" var="comment">
  Posted by <c:out value="${comment.author}"/>
  at <fmt:formatDate
  value="${comment.time}"
  pattern="MM/dd/yy hh:mm"/><br>
  <blockquote>
    <c:out value="${comment.text}"/>
  </blockquote><p>
</c:forEach>

<hr width="80%">
```

```
<c:if test="${user5.isLoggedIn}">
  <h2>Comment on this article</h2>
  <form action="comment_result.jsp" method="post">
    <input
      type="hidden"
      name="author"
      value="<c:out value="${user5.name}"/>">
    <input type="hidden"
      name="articleId"
      value="<c:out value="${article.articleId}"/>">

    <textarea name="text" rows="10" cols="30">
    </textarea><br>
    <input type="submit" name="Submit" value="submit">
  </form>
</c:if>
```

Anyone may read existing comments, so the current set is displayed with a standard `c:forEach` tag. Java News Today has decided that only logged-in users may add comments. This encourages users to sign up with the site and makes it easier to ban users who abuse the system. Consequently, the input form is wrapped in a `c:if` tag. The browser view of this page for a user who has logged in is shown in Figure 5.4. Note that this page recognizes a logged-in user in three ways: The login form is gone, the user's name appears in the header, and the comment section is active.

Figure 5.4. The JNT article page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter05/article.jsp?articleId=1
Search
JSP 2.0
Java on Devices
Developer tools
Desktop applications
Web services
Guess
Customize JNT
Done

Java News Today: Article

Hello Test User!

Tomcat 5.0 Available

Posted by JavaDude at 03/27/03 02:56

The Jakarta project today released version 5.0 of the popular Tomcat application server. Among many other new features, this is the first version to support the latest JSP 2.0 specification...

Comments

Posted by Test User at 03/27/03 02:56

That's great news!

Comment on this article

Which of these is a new feature in JSP 2.0?

- JavaBeans
- The expression language
- The custom tag library

One new feature to the form itself is that the name of the user is passed in a hidden variable, a common trick for transmitting data from one page to another. Although it would certainly have been possible for the receiving page to set manually the user's name in the `CommentBean` from the `UserInfoBean`, providing that information through the form allows the receiving page simply to do one `jsp: setProperty` instead of having to get properties from multiple places.

The comment result page is much like other pages that have already been considered. The heart of this page will simply set the values from the form into the bean in the standard way:

```
<jsp:setProperty name="comment" property="*"/>
```

The `CommentBean` is designed so that once all the fields have been set, the comment is correctly associated with an `ArticleBean`. Once again, putting the complex logic in the model has made it very easy to create the view.

5.8 The Remaining Pages

That pretty much wraps up the set of pages available at Java News Today, at least for now. Two other pages were not mentioned because they do not include anything new, but for the sake of completeness, they will be discussed briefly. All pages are available on the companion CD-ROM.

The front page, index.jsp, shows a list of the ten most recent stories. This looks exactly like the section page except that the list comes from `edition. recentArticles` instead of `currentSection.articles`. Finally, a page is available for the user to change preferences, which was already covered in Listings 3.16 and 3.17.

5.9 Summary and Conclusions

This chapter conveyed how easy it is to put together a site using beans and the standard tag library. Although Java News Today is still quite simple in both design and functionality, the principles used in this example are universal and will scale well in *any* site.

Although the JNT site itself is fairly dynamic, the data behind it is not. There is no way to add new stories, user preferences will be lost when the session expires, and comments will be lost if the system is ever shut down. The solution to all these problems is to have the beans communicate with a database, but before seeing how this is done, it will be necessary to discuss databases in general. This is the topic of the next chapter.

5.10 Tags Learned in this Chapter

`c:param` Passes a parameter to a page or URL
Parameters:
  `name`: The name of the parameter
  `value`: The value of the parameter; may be a script
Body: None

`c:url` Construct a URL suitable for use in an `href`
Parameters:
  `value`: The base page of the URL
Body: `c:param` tags

Chapter 6. Databases

In one sense, all Web sites are about information, or data. The stories on a news site are data, as are the items in a catalog. A great deal of data exists behind the scenes, such as information about users or the types of data they are interested in.

The problem of organizing large amounts of data is not a new one; many companies had to organize inventory or customer data long before the Web. This need to organize data gave rise to a kind of application called a *database*, a repository of structured information optimized to store and retrieve data quickly. Databases also allow multiple users to access or even change the same data simultaneously without corrupting it.

This chapter presents a brief overview of database technology, including standard tag library built-in features that greatly simplify working with databases. This chapter also discusses low-level techniques that allow JavaServer Pages to access databases and then discusses a bean-based approach that is both sophisticated and simple to use.

6.1 A Quick Introduction to Databases

Because any large collection of information is in a sense a database, there are many kinds of databases. The most commonly used kinds of commercial databases are called *relational databases*.

Relational databases store information in conceptually simple structures called *tables*. A table in a database is something like an HTML table or, for that matter, a table in book. For example, Table 6.1 contains some information about a CD collection.

Table 6.1. A Table with CD Information

Artist	Album Name
Black Tape for a Blue Girl	The Scavenger Bride
Mors Syphylitica	Feather and Fate
Voltaire	Boo Hoo

The data in Table 6.1 is organized into *rows*, each of which describes a single CD. Each row has *columns*, or *fields*, each containing a simple attribute of the CD. Each column also has a name, specified in the table header.

A table in a database also has rows containing named columns; the only additional feature is that each column also has a specified type. Most databases handle types that will be familiar to Java developers: integers, characters, strings, dates, floats, and so on. Some fields will be allowed to have a special value, `NULL`, which means "no data is available." The `empty` test as used in Listing 4.12 can be used to check for this special value.

Next, consider the problem of adding track data to the CD table. One possibility would be simply to add fields, such as track title and track length, to Table 6.1, but doing so would mean that every track entry would need to contain the album and artist name as well, which would waste space on the page or on disc, in the case of a real database. It would be much more efficient to use two tables: one for tracks and one for CDs. The two can be linked by giving each CD a unique integer ID and referencing that ID in the track table. This would lead to Tables 6.2 and 6.3.

Using integers to link up tables is a very common technique, especially when mapping *one-to-many* relationships, whereby a row in one table may connect to many rows of another table. Integers are small and so do not take up much space in the database, and because integers are easy to sort and manipulate, looking up information based on an ID is typically very fast. Similarly, because artists typically have many albums, another possible efficiency is to be gained by moving artists into their own tables and using an artist ID to map them to their albums.

Many, many databases are available. Many business sites use products from Oracle or Microsoft, but a number of high-quality, free databases also are available. These databases are perfectly suitable for small to midsized sites or for development and are very attractive to people who cannot afford a large commercial database. MySQL and PostgreSQL are prime examples of this latter type of database. MySQL is available from http://www.mysql.org, and PostgreSQL is available from http://www.postgresql.org.

Table 6.2. The CD Table with a Unique ID

Artist	Album Name	Album ID
Black Tape for a Blue Girl	The Scavenger Bride	1
Mors Syphylitica	Feather and Fate	2

Table 6.2. The CD Table with a Unique ID

Artist	Album Name	Album ID
Voltaire	Boo Hoo	3

Table 6.3. The Track Table

Album ID	Track Name
1	The Scavenger Bride
1	Kinski
2	The Hues of Longing
2	Naturally Cruel
3	Future Ex-Girlfriend
3	I'm Sorry

All the examples in this book use a database called hsqldb, a small, fast, free relational database implemented in 100% Pure Java. In addition to its other features, hsqldb can run on any platform and is completely self-contained, so readers running the examples in this book will not need to worry about setting up or configuring a database. Hsqldb is included on the companion CD-ROM and is also available from http://hsqldb.sourceforge.net/.

6.2 A Language for Databases

For humans and databases to work together, they must speak a common language. Although in principle, every database manufacturer could define its own such language, doing so would cause problems for both users and database vendors. To avoid these problems, a standard called *Structured Query Language* (SQL, pronounced "sequel") that all database vendors support, although frequently with some enhancements specific to their products, has been defined.

Most databases provide a utility program that allows users to enter SQL commands interactively and get results back. That program for hsqldb's can be accessed by running the following:

```
java -cp hsqldb.jar org.hsqldb.util.DatabaseManager
```

One such command might be instructions to create a new table by specifying the names and types. The SQL commands to create the CD and track tables from Tables 6.1 and 6.2 are shown in Listing 6.1.

Listing 6.1 SQL commands to create tables

```
CREATE TABLE artist (
    artist_id int,
    name     char(40)
);

CREATE TABLE cd (
    album_id int,
    artist_id int,
    name     char(40)
);

CREATE TABLE track (
    album_id int,
    name    char(60)
);
```

These commands define the columns in each table by giving each column a name and a type. The semicolons here indicate the end of each SQL command. This is a common convention but is not universal. Some SQL interpreters require the word *go* after each command.

Once the tables have been created, data can be stored in them with SQL's `insert` command, as shown in Listing 6.2.

Listing 6.2 SQL commands to put data into tables

```
INSERT INTO artist VALUES(1,'Mors Syphilitica');

INSERT INTO cd VALUES(1,1,'Primrose');
INSERT INTO cd VALUES(2,1,'Feather and Fate');

INSERT INTO track VALUES(1,'Ungrateful Girl');
INSERT INTO track VALUES(1,'Remidy');

INSERT INTO track VALUES(2,'The Hues of Longing');
INSERT INTO track VALUES(2,'Naturally Cruel');
```

These commands build rows in the database by specifying the value for each column in that row. Astute readers will note that the name of the second track is misspelled; fortunately, there is a way to change data once it has been entered, and this will be shown shortly.

Of course, data is useful only if it can be retrieved, and the SQL command that does this is called `select`. It has a number of variations, but the simplest lists all data from a table. The following command would list all tracks for all albums:

```
SELECT * FROM track;
```

The asterisk indicates that all fields should be retrieved. If only the track name and duration were desired, the asterisk would be replaced by `name`,`length`.

Generally, pulling all the rows from a table is not that interesting. In this example, it would have pulled the tracks from both albums, which is unlikely to be of any particular interest. A SELECT command can be modified by a `where` *clause*, which imposes one or more conditions that must be true in order for the row to be retrieved. To see only the names of the tracks on "Primrose," the SQL command would look like this:

```
SELECT name from track WHERE album_id = 1;
```

This command will obtain the desired data, but in order to construct this query, it is necessary to know the album ID. This ID could be found by looking at the CD table, using the following query:

```
SELECT album_id from cd WHERE name='Primrose';
```

But this is cumbersome. Fortunately, it is unnecessary, as the two queries can be combined into a single command by selecting from the two tables simultaneously and imposing a condition that connects them. This kind of query is called a *join* because it joins two or more tables together. Here is the SQL to accomplish this:

```
SELECT track.name FROM cd, track
WHERE cd.album_id = track.album_id
AND   cd.name     = 'Primrose';
```

The field to select is specified as the table name, a dot, and then the column name. This is necessary because both the CD and track tables have a field called `name`, so it is necessary to clarify which table is intended. Without this clarification, the database would respond with an error about a "field ambiguity." The `SELECT` is done on both the CD and track tables, and they are joined by the condition that the `album_id` fields must match.

An additional requirement is placed on the album name, so that only the tracks from that album will be returned.

The `SELECT` command has many more options. But this is enough to follow the examples throughout the book.

Other SQL commands delete and update rows. The `DELETE` command also takes a `where` clause and will delete all rows that satisfy the condition in the clause. The `UPDATE` command likewise takes a `where` clause, as well as a set of new values. For example, to change one of the track names, a SQL statement like this could be used:

```
UPDATE track
SET name='Remedy'
WHERE name='Remidy';
```

This will find all rows in which the title track is named "Remidy" and will replace the name with the correct spelling.

6.3 Using SQL Directly from JSPs

The standard tag library contains tags that allow SQL commands to be embedded directly in a page. The most basic of these is the `query` tag, which allows a page to perform a `select` and display the results. The tag's use is demonstrated in Listing 6.3, which selects the list of artists from the table defined in Listing 6.1.

Listing 6.3 A page that gets data from a database

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql"
    uri="http://java.sun.com/jstl/sql" %>

<sql:query
  dataSource="jdbc:hsqldb:jspbook,org.hsqldb.jdbcDriver,sa"
  sql="select * from artist"
  var="artists"/>
<ul>
<c:forEach items="${artists.rows}" var="artist">
<li><a href="<c:url value="show_cds.jsp">
```

```
  <c:param name="artist_id" value="${artist.artist_id}"/>
  <c:param name="name" value="${artist.name}"/>
</c:url>"><c:out escapeXml="false"
               value="${artist.name}"/></a>
</c:forEach>
</ul>
```

This example starts by importing the core library and a new SQL library that contains the new tags. Immediately after loading the library, the `query` tag is used to load some data. The `query` tag has many options, but the ones used here are the most common. First, the tag needs to be told how to connect to the database where the information lives, which is specified as the `dataSource` parameter. The exact form of this will make more sense after Chapter 9, but for now, think of it as naming three things: the location of the database, the kind of database, and the user name and password with which to connect to the database. These are all specified on one line, separated by commas.

The `sql` parameter specifies the SQL to execute. The SQL used here is a simple `select` command.

Finally, the `var` parameter names a variable in which the results of the query should be stored. This is somewhat similar to the `var` parameter in the `c:forEach` tag in that both make a value available elsewhere on the page.

Not coincidentally, the next place this variable is seen is in a `c:forEach` tag on the next line. Note that this variable is used as the `items`, because this one variable contains something like an array, each element of which will be one row of data. The `artist` variable, defined in the `c:forEach` tag, will hold each row in turn.

Within the body of the `c:forEach` tag, the `artist` variable acts like the `param` variable in Section 4.6, which has a different property for each value sent by a form. Similarly, `artist` will have one property for each column, which may be obtained by using the normal dot notation used with beans. The artist name, therefore, is obtained with

```
<c:out value="${artist.name}" escapeXml="false"/>
```

The `escapeXml` option to the `c:out` tag is new. Some bands have non-ASCII characters in their names, such as The Crüxshadows or Björk. Such names can be stored in the database by using the HTML that encodes these characters. For example, `&#252;` represents the character ü. However, by default, the `c:out` tag will itself encode any special characters it encounters, including ampersands. If this were allowed to happen, it would turn `&#252` into `&#252`. Setting `escapeXml="false"` turns off this behavior and should be used whenever the `c:out` tag will be displaying data that has already been encoded for display.

The artist name should be a link to a page where all of that artist's albums will be shown. In order to do that, the `url` tag is used to construct a URL that will call the show_cds.jsp page and pass along the `artist_id` of interest. This works just like the Java News Today section list from the previous chapter. The artist's name is also passed along so that it can be displayed on the following page. This is not strictly necessary, as once the artist ID is available, the name could be obtained through another `select`. However, because the name is already available, it may as well be used from here in order to save the effort of doing an extra call to the database.

Listing 6.4 shows the show_cds.jsp page, which will once again use the `sql:query` tag. Whereas in Listing 6.3, the query was always the same, here there must be a way to build a `where` clause that includes the `artist_id`. Fortunately, the tag library allows for this.

Listing 6.4 A parameterized query

```
<%@ taglib prefix="c"
   uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql"
   uri="http://java.sun.com/jstl/sql" %>

<sql:query
  dataSource="jdbc:hsqldb:jspbook,org.hsqldb.jdbcDriver,sa"
  sql="select * from cd where artist_id = ?"
  var="cds">

  <sql:param value="${param.artist_id}"/>
</sql:query>

<h2>Albums by <c:out escapeXml="false"
                  value="${param.name}"/></h2>

<ul>
<c:forEach items="${cds.rows}" var="cd">
<li><a href="<c:url value="show_tracks.jsp">
<c:param name="cd_id" value="${cd.cd_id}"/>
<c:param name="name" value="${cd.name}"/>
</c:url>"><c:out value="${cd.name}"/></a>
```

```
</c:forEach>
</ul>
```

The `sql:query` tag here looks very similar to the one in Listing 6.3; both specify a `dataSource`, `var`, and `sql` to run. In this example, however, the `sql` has a question mark where the `artist_id` passed in from the previous page might be expected. Correspondingly, the `sql:query` tag has a body containing a `sql:param` tag, whose value is the very `artist_id` that was needed.

This is another feature of the `sql:query` tag. Before the query is run, question marks within the `sql` parameter may be filled in with values from `sql:param` tags in the body. Because the values of `sql:param` come from scripts, queries can be dynamically altered as needed.

After the `sql:query`, the rest of the page is straightforward. Another `c:forEach` iterates all the CDs and provides a link to see the tracks on another page.

6.4 Inserting Data from JSPs

To make the little CD application more useful, it can be expanded to allow the user to add new artists, CDs, and tracks. Not surprisingly, the standard tag library provides another tag to facilitate this: `sql:update`. Before jumping into seeing how this tag is used, it is worthwhile to step back and consider what will need to be done in order to add a new artist.

First, the user will specify the name in a form, which will be sent to another JSP, which will use the new tag to perform an `insert`. It would be reasonable to expect that we will use a `sql:param` in order to pass the name to the query. This is all straightforward enough. However, it is important to keep in mind that the artist table has not only a name but also an `artist_id`. Where will this ID come from?

One possibility would be to force the user to provide it along with the name. But this is far from satisfactory; this ID is used only internally by the system to track data and has no intrinsic meaning to the user. Hence the user should never see it. In addition, there is no clear way in which the user would know what value to use.

It therefore seems that the system should keep track of IDs. That is perfectly fine, as such information can easily be added to the database. It is merely necessary to create another table of IDs, which will be called `sequence`, as it will provide sequences of ID values. Its definition is simple:

```
create table sequence (
  name char (60),
  id   int)

insert into sequence values('artist',0);
insert into sequence values('album',0);
insert into sequence values('track',0);
```

With this table in place, creating a new artist would take the following steps:

1. Use a `select` to find the current ID where `name` is `artist`.
2. Use an `update` to increment that ID, so the next artist created will get a new number.
3. Use the obtained ID in an `insert` to create the `artist`.

There is in fact a further complication. If two users try to add an artist at the same time, they might both get the same ID in step 1 before either can get to step 2 to update the current ID. Most modern databases have a way to prevent this, and it is supported by the tag library through the `jsp:transaction` tag, which is beyond the scope of the book. Listing 6.5 shows everything that must be done in a JSP in order to add an artist to the database with a proper ID.

Listing 6.5 Using a JSP to add data to a database

```
<%@ taglib prefix="sql"
    uri="http://java.sun.com/jstl/sql" %>

<sql:query
  dataSource="jdbc:hsqldb:jspbook,org.hsqldb.jdbcDriver,sa"
  sql="select value from sequence where name='Artist'"
  var="ids"/>

<sql:update
  dataSource="jdbc:hsqldb:jspbook,org.hsqldb.jdbcDriver,sa"
  sql="insert into artist(artist_id,name) values(?,?)">
<sql:param value="${ids.rows[0].id}"/>
<sql:param value="${param.name}"/>
</sql:update>
```

```
<sql:update
  dataSource="jdbc:hsqldb:jspbook,org.hsqldb.jdbcDriver,sa"
  sql="update sequence set value=? where name='Artist'">
<sql:param value="${ids.rows[0].id + 1}"/>
</sql:update>

New artist has been added!<p>
<a href="show_artists.jsp">Return to the
artist list</a>
```

The example exactly follows the steps outlined previously. The only noteworthy point is that the ID obtained from the `select` is referred to as `ids.rows[0].id`. Recall that `rows` is an arraylike object, suitable for using in `c:forEach` tags; therefore, element 0 of this object will be the first row.

6.5 SQL and Beans

In Listing 6.5, it is immediately obvious that 99 percent of it is manipulating the model t

he database with only a single tiny line of view information announcing the completion

of the task. This is just plain wrong!

The view layer has too much model, and using the SQL tags as in the previous section is fine for quick-and-dirty database applications. However, problems would soon arise when dealing with a larger, more complex site. If ten pages use some hard-coded SQL and then the structure of the database changes, it can be very difficult to find and fix all the problems. Although it may seem as though a database, once designed, should never change, requirements in the real world commonly shift over the course of a project.

The solution, as always, is to move the model layer, where it belongs, into some Java beans. Fortunately, this is a simple exercise, as beans and databases already have a great deal in common. A database row has a number of named columns, just as a bean has a number of named properties. A table can have many rows, just as an array can have many beans. In Chapter 5, these correspondences were used in a set of hard-coded beans to mimic a database. All that is necessary to complete the picture is to modify those beans so they connect to a real database.

Tools that will automatically build a class or bean that reflects a table are available. A very simple tool, Table2Bean, from Canetoad Software, is included on the accompanying CD-ROM. As its name implies, Table2Bean takes a SQL table definition and builds a bean. This bean can then provide easy mechanisms for interfacing with the underlying table.

To see how this will work, consider CDBean, generated from the table in Listing 6.1.

1. If the `cdId` property is set, the bean will construct a SQL command, such as `SELECT * FROM CD WHERE cdId= the provided id`, execute this statement, and use the result to populate the rest of the properties.
2. After loading the data, any property can be changed by using the normal set methods. The bean will also provide a special property, called `save`. If this property is set after any other properties have been changed, the changes will be saved back to the database with an `UPDATE` command.
3. Similarly, if the `save` property is set before the `cdId` property has been set, the bean will assume that this is new data and will enter it into the database with an `INSERT` command.
4. Finally, another special property, called `beans`, will return an array of beans that match the current properties. If a page sets the `artistId` field to 1, the `beans` property will return an array of all the albums from artist number 1.

The next chapter discusses how Java News Today will use these new beans, but the principles can be examined by seeing how they could be used to simplify the CD application. Listing 6.6 shows the new version of the page that displays all an artist's albums.

Listing 6.6 Retrieving data through a bean

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="cdBean"
  class="com.awl.jspbook.ch06.CdBean"/>
```

```
<jsp:setProperty name="cdBean" property="artistId"/>

<h2>Albums by
<c:out escapeXml="false" value="${param.name}"/></h2>
<ul>
<c:forEach items="${cdBean.beans}" var="cd">
<li><a href="<c:url value="show_tracks.jsp">
<c:param name="cd_id" value="${cd.cdId}"/>
<c:param name="name" value="${cd.name}"/>
</c:url>"><c:out value="${cd.name}"/></a>
</c:forEach>
</ul>
```

The only difference is that the `sql:query` tag has been replaced by a `jsp:useBean` and by `jsp:setProperty` tags; now the iteration goes over `cdBean.beans`. Although this is no shorter than the database version, the conceptual difference is huge. Now this page does not know whether the data is coming from a database or a serialized bean or is connecting to a Web site in order to get its information. The details of the model have therefore been hidden from the view, which is as it should be.
The difference is even more pronounced in the bean version of the page that adds an artist, which is shown in Listing 6.7.

Listing 6.7 Storing data through a bean

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="artistBean"
  class="com.awl.jspbook.ch06.ArtistBean"/>

<jsp:setProperty name="artistBean" property="name"/>

<jsp:setProperty
  name="artistBean"
  property="save"
  value="true"/>
```

```
New artist has been added!<p>
<a href="show_artists_beans.jsp">Return to the
artist list</a>
```

Now that's more like it! All the details of the ID are hidden away in the bean, so all the view needs to do is load the data and then tell the model to save itself.

One small detail has been glossed over in these last two examples: how these beans get the information necessary to connect to the database. This was passed in explicitly when using the SQL tags, but the beans are able to hide this information by using a feature of Java. It is possible for a Java class to load a resource given its name, so a resource called "db" that holds the connection information has been created, and the beans know to load that information when it is first needed.

6.6 Summary and Conclusions

A database is a collection of tables, and tables contain rows of data, organized into columns. Each column contains one attribute of the row. SQL is a common language that allows humans to communicate with databases, and the standard tag libraries make it relatively painless to use SQL from within pages. For many reasons, however, it is better to hide the SQL and other database information within beans.

Up to this point, Java News Today has been a somewhat uninteresting site, as there has been no way to add new stories or make users' preferences permanent. This will change in the next chapter, where JNT will move to a database and add editorial screens.

6.7 Tags Learned in This Chapter

`sql:query` Perform a query against a database

Parameters:

`dataSource`: A string specifying how to connect to the database

`sql`: The SQL to run; may contain parameters to be filled in, indicated by question marks

`var`: The name of the variable in which to store the results

Body:

`sql:param` tags

`sql:update` Update, create, or delete data from a database

Parameters:

`dataSource`: A string specifying how to connect to the database

`sql`: The SQL to run; may contain parameters to be filled in, indicated by question marks

`var`: The name of the variable in which to store the results

Body: `sql:param` tags

`sql:param` Provide a parameter to SQL in a `sql:query` or `sql:update` tag

Parameters:

`value`: The value to use; may be a script

Body: None

Chapter 7. Java News Today: Part 2

Finally, after all the preliminaries and the read-only site of Chapter 5, Java News Today is ready to start providing some content! Doing so has not been possible until now because there was no good place to store this content. It would not make sense to have to write a brand new JavaServer Page or manually update the beans used in Chapter 5 each time a new story was published. What is needed is a JSP that will allow a reporter to write a new story as easily as a user can read one. Databases, as covered in Chapter 6, provide the means to build such functionality.

7.1 Designing the Tables

As a data model was already developed in Chapter 5, the simplest plan of attack would be to turn this model into SQL and create the database. Once that's done, an object-relational mapping tool could turn these tables back into beans, and the job would practically be finished. The reason is that, if all the naming conventions for bean properties and column names are carefully followed, the new beans will have the same property names as the original ones, and none of the pages or forms will need to change at all. This is another big advantage to the model/view/controller paradigm: It makes it possible to change completely the way the model works; as long as the interfaces between the model and the view stay the same, the view will not need to be rewritten.

Although it would be very easy to follow this plan of attack and recreate the existing site on top of a database, doing so would preclude a great deal of possible new functionality that a database could offer. Creating a site based on hard-coded beans leads to necessary restrictions in the ways in which the data could be accessed. Because there is no easy way to filter out a subset, it is necessary to show all the sections within in an edition and all articles within a section. With a database and a set of beans that make it easy to construct SQL `where` clauses, the data can be managed, grouped, and arranged in any way that might be useful. In particular, users now have the option to view only sections in which they are interested; further, it is possible to rank articles within those sections to indicate which ones are likely to be the most interesting.

To support these features, the data model will need to be rethought a little. The basic fields in the old beans will still be needed; for example, the `ArticleBean` will still need the text of the article, the time it was published, a headline, a summary, and the name of the author.

This last item already suggests one major change that should be made. In the CD database from Chapter 6, it was noted that rather than store the artist's name in every CD, it made more sense to have a separate table of artists and to link artists to CDs through the use of a small ID. The same is true for authors and articles; it would be possible to connect an author to an article by storing in the `article` table the `user_id` of the author rather than the author's name. This way, if an author's name changes, it will not be necessary to change every article; the `user_info` table can simply be updated in one place. This is also more efficient, as the name may take up 20 bytes to store, but an ID will take only 4. This process of pulling common data into separate tables is called *normalizing* the database.

More generally, when working with a database, it is important to consider what relationships will exist between otherwise apparently unconnected data items. For example, currently there is no relationship between sections and users, but for users to be able to select the set of sections in their editions, such a relationship must be included in the database. The question then becomes, How this should be modeled?

One possibility would be to add to the `user_info` table some additional columns, such as `wants_section_1`, `wants_section_2`, and so on. But this is not very general; if it creates a new section a year from now, JNT will need not only to update all the users but also to change the very structure of the database and modify all the beans and JSPs that use this table. That is something that no one should have to live through if it can be avoided, and, fortunately in this case, it can be avoided.

Following the examples from Chapter 6, each of the tables will have a unique ID, so each user will have a `user_id`, each section will have a `section_id`, and so on. So, to model the connection between users and sections, another table that will have a `user_id` and a `section_id` can be introduced. If user 50 does *not* want section 3, this new table would have a row where `user_id = 50` and `section_id = 3`. A table like this, which holds only the IDs of other tables and has no data of its own, is known as a *join table*.

It would also be possible, and in some ways simpler, to keep track of which sections a user *does* want. The advantage of storing unwanted sections is that if a new section is created, every user will initially get it by default and can then opt to turn it off. If the database tracked only sections a user did want, the user would need to act explicitly to add new sections and hence might miss out on some good content.

Two more new tables will be used to connect articles to users, although less directly. First, the notion of a *keyword* will be added to the system. A keyword is a single word or short phrase that describes an article. This is more finely grained than sections; whereas a section might deal with a broad category, such as "Java on consumer devices," the keywords might list particular devices or vendors that support Java.

As each article may have many keywords, each connected to many articles, another join table will be used to connect them. In order to do so, this new table will have a `keyword_id` and an `article_id`. This also suggests creating a similar table to connect users to keywords by maintaining a list of `keyword_id` and `user_id` pairs. This table will allow users to indicate the set of keywords in which they are interested. With these two tables, a user can be connected to an article by looking for good matches between article keywords and user keywords.

The database design is almost finished; the only other thing needed is a way to ensure that only Java News Today staff can create new articles. To do this, a new field will be added to the `user_info` table to mark certain users as reporters. With that done, the SQL needed to create the Java News Today database is shown in Listing 7.1.

Listing 7.1 The JNT schema

```
create table user_info (
       usr_id          int,
       username        char(40),
       password        char(40),
       name            char(20),
       bg_color        char(6),
       text_color      char(6),
       banner_color    char(6),
       reporter_ind    char(1)
);

create table section (
       section_id       int,
       name             char(20),
       summary          varchar(1024)
);

create table article (
       article_id       int,
       section_id      int,
       author_id       int,
       created_date    datetime,
       headline        varchar(80),
       summary         varchar(1024),
       text            varchar(4096)
);

create table keyword (
       keyword_id      int,
       name            char(20)
);

create table user_sections (
       user_id         int,
       section_id      int
);

create table user_keywords (
       user_id         int,
       keyword_id      int
);

create table article_keywords (
      article_id       int,
      keyword_id       int
);

create table comment (
       comment_id      int,
       article_id      int,
       author_id       int,
       created_date    datetime,
       text            varchar(4096)
);
```

```
create table quiz (
       question        varchar(80),
       answer1         varchar(80),
       answer2         varchar(80),
       answer3         varchar(80),
       correct_answer   int
);
```

The names for fields and tables follow certain well-accepted conventions. Database names use underscores to separate multiword names; when beans are generated from these tables, the underscores will be removed, and the letter following the underscores will be capitalized. Database fields ending with `_ind` are indicators with the value `Y` or `N`. The equivalent bean property will have values `true` or `false` and will therefore be suitable for use as the tests in `c:if` and `c:when` tags.

7.2 Adding Articles

With the new database and coding conventions established, adding new articles to the database is quite simple. First, a link to the article creation page must be added to the left-hand navigation, and a check must be done to ensure that this link is available only to reporters. This is handled by a simple addition to the navigation, shown in Listing 7.2.

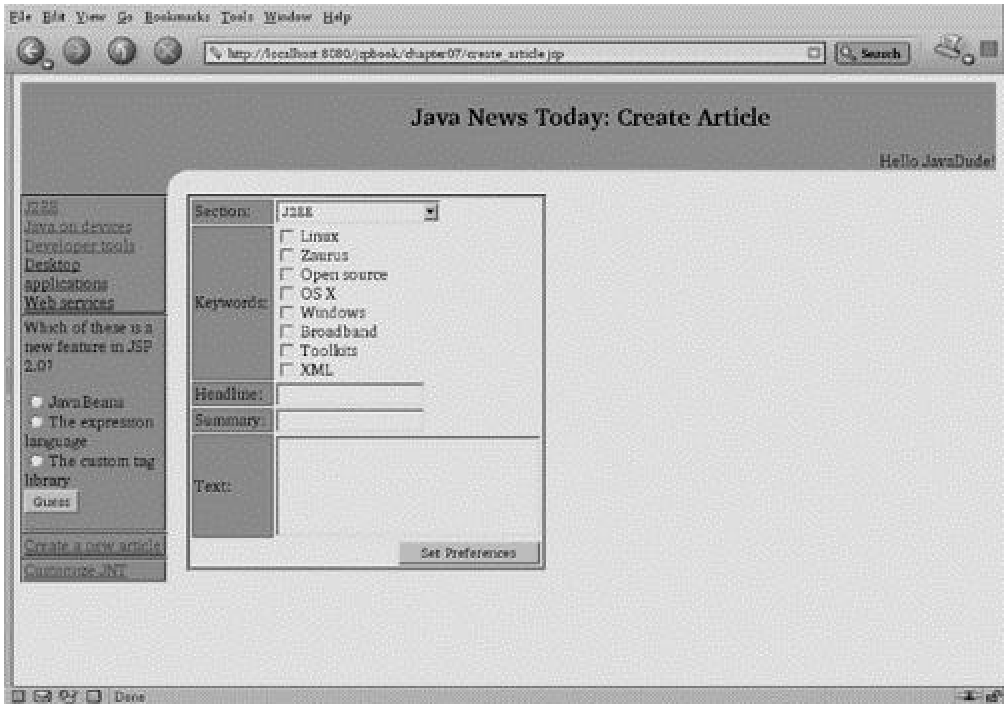
Listing 7.2 New link for reporters

```
<c:if test="${user7.isReporter}">
  <div class="bordered">
    <a href="create_article.jsp">Create a new article</a>
  </div>
</c:if>
```

If the user has not yet logged in, the `isReporter` property will be empty, that is, neither true nor false. In this case, the test will still fail, and the link will not be shown. The check here is technically not sufficient, as it does nothing to prevent a malicious user from going to create_article.jsp directly by entering the URL in his or her browser. To prevent this, a controller is needed to handle site security; one will be built in Chapter 12.
The page to create articles is another standard HTML form. The author will need to be able to select the section to which the new story should be added, a way to mark which

keywords are relevant, and a big text box for the contents. The JSP is shown in Listing 7.3, and the result as viewed in a browser appears in Figure 7.1.

Figure 7.1. The article creation page.



Listing 7.3 The article creation page

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="user7"
  class="com.awl.jspbook.ch07.UserInfoBean"
  scope="session"/>

<jsp:useBean
  id="edition7"
  class="com.awl.jspbook.ch07.EditionBean"/>

<jsp:include page="top.jsp" flush="true">
  <jsp:param name="title" value="Create Article"/>
```

```
</jsp:include>

<table class="form">
<form action="article_handler.jsp" method="post">
<input
   type="hidden"
   name="authorId"
   value="<c:out value="${user7.userInfoId}"/>">

<tr>
  <td class="label">Section:</td>
  <td>
    <select name="sectionId">
      <c:forEach items="${edition7.allSections}"
                      var="section">
      <option value="<c:out
              value="${section.sectionId}"/>">
      <c:out value="${section.name}"/>
      </c:forEach>
    </select>
  </td>
</tr>

<tr>
  <td class="label">Keywords:</td>
  <td>
    <c:forEach items="${edition7.allKeywords}"
                    var="keyword">
      <input
        type="checkbox"
        name="keywordId"
        value=<c:out value="${keyword.keywordId}"/>>
       <c:out value="${keyword.name}"/><br>
     </c:forEach>
  </td>
</tr>
```

```
<tr>
  <td class="label">Headline:</td>
  <td><input type="text" name="headline"></td>
</tr>

<tr>
  <td class="label">Summary:</td>
  <td><input type="text" name="summary"></td>
</tr>

<tr>
  <td class="label">Text:</td>
  <td>
    <textarea name="text" rows="5" cols="30">
    </textarea>
  </td>
</tr>

<tr>
  <td colspan="2" align="right">
    <input type="submit" name="go" value="Set Preferences">
  </td>
</tr>
</form>
</table>

<jsp:include page="bottom.jsp" flush="true"/>
```

Note that the lists of available sections and keywords come from the `EditionBean`. This is appropriate, as this bean acts as the master container for all options on the site, as well as the specific set of those options selected by each user.

Because all the work is done in a bean representing the model, as it should be, the view portion of the article handler is almost trivial:

```
<jsp:setProperty bean="article" property="*">
<jsp:setProperty
  name="article"
  property="useNowAsDate"
  value="true"/>
<jsp:setProperty bean="article" property="save"
                 value="true">
```

The first line sets all the properties of the article, including the section and the text. The first line also sets an array of keyword IDs, which the bean will use internally to set up the proper entries in the `article_keyword` table. The second line sets a special property, called `useNowAsDate`, of the bean. When this property is set, it will set the underlying `createdDate` property to the current time. It would be possible to set `createdDate` directly from the page, but working with date properties can be cumbersome, so the `useNowAsDate` property was provided as a convenience.

Finally, the third line will then perform the save and will write all the data to the database. It might seem that this extra `jsp:setProperty` tag could be avoided by using a hidden input field, as was done in Listing 5.9, to pass the user's name to the comment handler: perhaps something like

```
<input type="hidden" name="save" value="true">
```

However, this is not guaranteed to work. Nothing in the JSP specification says anything about the order in which properties will be set. If the `save` property were sent along with `text` and `sectionId`, it is quite possible that first `sectionId` would be set, then `save`, and finally `text`. The net effect would be that an article would be placed in the database with a `sectionId` but no content!

7.3 User Pages

As promised, very few changes need to be made to the pages from Chapter 5. The section, article, quiz, and navigation can all stay almost exactly the same. The few things that do need to change reflect the new use of normalized tables.

In the article page, it was formerly possible to obtain the author's name with

```
<c:out value="${article.authorName}"/>
```

But because the `authorName` is no longer kept in the `ArticleBean` but only the `authorId`, an additional mechanism must be used to go from the ID to the author before getting the name. Fortunately, the bean provides a means to do this, by providing an `author` property that holds the appropriate `UserInfoBean`. Getting the name is then as simple as

```
<c:out value="${article.author.name}"/>
```

Note the use of a nested property.

The remaining user pages that need to be changed are those that now need to send data to the database. These consist of the page that handles the saving of user preferences (Listing 3.18) and the page that adds a comment to an article (Listing 5.9).

Recall that the user preferences are placed in the bean with the tag `jsp:setProperty`, just as the one used to set the article properties. Therefore, all that is needed to write these values to the database is another `jsp:setProperty` tag that sets the `save` property. This will tell the bean to save its contents to the database, and because it will already have a `userInfoId` from the time when the user logged in, it will know that data is being updated instead of created.

Now that the preferences for an existing user can be saved, it is easy to allow the system to create new users. First, the page should contain a message prompting users to sign up with the site. The easiest way to do this is by adding a small message to the login form in Listing 5.2:

```
Don't have an account yet?
Click <a href="user_preferences.jsp">here</a>
to register with Java News Today!
```

It may seem odd that users would be sent to the user preferences page to sign up, as that page lets existing users change their options. It would certainly be possible to create a separate sign-up page, but consider what such a page would contain. It would need a form that prompted for a user name, password, and real name, which would seem to be the minimal information needed in order to register a new user. However, it would make sense to give new users the option to set their preferences at the time they join, which would mean that the sign-up page would have all the same fields as the user preferences page, in addition to the new ones. It would instead seem to be easier to put all these fields on the same page and use a conditional tag to turn off the ones that aren't always needed. This modifies the user preferences page as shown in Listing 7.4.

Listing 7.4 The new user preferences page

```
<c:if test="${!user7.isLoggedIn}">
  <tr>
    <td class="label">Your name:</td>
    <td><input type="text" name="name"></td>
  </tr>
  <tr>
    <td class="label">User name:</td>
    <td><input type="text" name="username"></td>
  </tr>

  <tr>
    <td class="label">Password:</td>
    <td><input type="password" name="password"></td>
  </tr>
</c:if>

<tr>
  <td class="label">Background color:</td>
  <td><input type="text" name="bgColor"
       value='<c:out value="${user7.bgColor}"/>'></td>
</tr>

<tr>
  <td class="label">Banner color:</td>
  <td><input type="text" name="bannerColor"
       value='<c:out value="${user7.bannerColor}"/>'></td>
</tr>

<tr>
  <td class="label">Text color:</td>
  <td><input type="text" name="textColor"
       value='<c:out value="${user7.textColor}"/>'></td>
</tr>
```

To ensure that this will work, consider what will happen when the user clicks the submit button and goes to preferences_handler.jsp in each of the circumstances this page will need to handle. In both cases, the result will be to set all the form variables and then set the `save` property. This is the right thing to do, regardless of whether the user is signing up or changing preferences. The latter case has already been considered and is known to work. In the former case, the initial `jsp:setProperty` will set the additional user name and password fields; then, when the `save` property is set, the bean will recognize that there is not yet a `userId` specified and hence will do a SQL `insert` instead of an `update`. The upshot of all this is that once again, by putting all the hard work in the model layer, the task of creating the view has been greatly simplified. If we did not have beans at our disposal, we would need separate pages for new users and existing users and then two other pages to handle saving the data in each of these cases. As an exercise, consider how these cases would be handled if all this work needed to be done using only `sql:query` and `sql:update` tags!

7.4 Other User Preferences

So far, the user preferences page has dealt only with simple properties: the ones that do not involve the join tables. The problem with the remaining properties is twofold: figuring out how to (1) display the user's current choices and (2) allow them to be changed. The solutions to these problems will be different for sections and keywords because of the different ways this information will be used.

A good way to figure out how to tackle such problems is to solve first them in raw SQL. Then the SQL can be moved into the bean. Showing the list of sections the user has selected *not* to display is easy:

```
select section.name
from section,user_sections
where section.section_id = user_sections.section_id
and user_section.user_id = ?
```

The question mark would get filled in with a `sql:param` from the current user. Unfortunately, what is needed is the inverse of this: a list of all the sections that the user does want. This could be done in three steps: (1) run the query, (2) run another query to get the list of all sections, and (3) remove the items in the first list from the second list in some Java code. This would work, but as a general rule, it is worth trying to use as few queries as possible and to do as much work with those queries as possible. This is partly for the sake of efficiency, as each query will take some time and impose some overhead on the database and the network. In general, databases will also be able to manipulate data more efficiently than the equivalent Java code.

It turns out that it is possible to cook up a query that will do all the necessary work in one step. This conceptually does the same thing that could be done manually in Java: Select the items from `user_section`, and remove the matching items from `section`:

```
select * from section
where section_id not in
(select section_id from user_section
 where user_info_id = ?)
```

The inner `select`, the one in parentheses, retrieves the list of sections that the user does not want; then the SQL keywords `not in` remove those sections from the outer `select`. Now that the query has been designed, it will be put into the `EditionBean` as a new property: `selectedSections`. This will alter the section list in the navigation as shown in Listing 7.5.

Listing 7.5 The customized section list

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="edition7"
  class="com.awl.jspbook.ch07.EditionBean"
  scope="session"/>

<jsp:useBean
  id="currentSection"
  class="com.awl.jspbook.ch07.SectionBean"/>

<jsp:setProperty
  name="currentSection"
  property="sectionId"/>

<c:forEach items="${edition7.selectedSections}"
          var="section">

  <c:if
   test="${currentSection.sectionId == section.sectionId}">
```

```
    *
  </c:if>

  <a href="<c:url value="section.jsp">
   <c:param name="sectionId" value="${section.sectionId}"/>
  </c:url>"><c:out value="${section.name}"/></a><br>
</c:forEach>
```

One other point needs to be made about this example. The query to get sections requires `user_id` as a parameter. If the user has not yet logged in, no ID will be available, and the query will fail. This condition could be checked in the JSP with a `c:choose` tag. If `user.userId` is empty, the page would then do what it did previously and iterate the sections from `edition.sections`. However, this test has been placed in the bean for all the usual reasons about keeping the view simple. In this case, it would be more correct to put this check in the controller, as the model needs to be controlled based on an external criterion.

Similarly, it will be necessary to notify the `EditionBean` of the user's ID when the user logs in. This can be done with a simple addition to the login handler page:

```
<c:set name="edition7.userId" value="${user7.userId}"/>
```

Now that the navigation can make use of the user's section choices, a means for the user to alter them is needed. The logical user interface for this would be a list of every section, with a check box next to the ones the user would like to see. When the user goes to edit the list, the sections already selected should be checked so the user does not have to reenter the choices whenever adding or removing only one.

This again requires connection between the `section` and `user_info` tables but with an additional complication. The page cannot show only the sections the user has selected, as that would not allow the person to add one. Nor can it show only the ones the user has not selected, as there would then be no way to remove one. This could be handled with two queries, first iterating one set of sections and then the other. A better solution would be to select all the sections in one shot, along with an indicator as to whether the user has selected each.

This can be done with yet another feature of SQL: *outer join*. The idea is that a normal, or *inner*, join between two tables `A` and `B` will have one row for each value common to both tables. An outer join might have one row for every row in `A`. If `B` has matching data, that data will be available; if not, those values will be marked as NULL.

To make these ideas more concrete, consider the two tables defined next.

```
create table character (character_id int,
                     character_name char(10))
create table actor (actor_id int, actor_name char(10))

insert into character values(1,'John Crichton')
insert into character values(2,'Aeryn Sun')
insert into character values(3,'Chiana')

insert into actor values(1,'Ben Browder')
insert into actor values(3,'Gigi Edgley')
```

A regular inner join could be used to get a list of actors and characters:

```
select character_name,actor_name
from actor, character
where character_id = actor_id
```

The result would be in the following table:

John Crichton	Ben Browder
Chiana	Gigi Edgley

However, this table is missing information about characters for whom the corresponding actor is not available. This can be remedied with an outer join:

```
select character_name,actor_name
from character
left join actor
on character_id = actor_id
```

This produces the following table:

John Crichton	Ben Browder
Aeryn Sun	NULL
Chiana	Gigi Edgley

This table contains all the information we have available and might serve to remind someone to insert "Claudia Black" into the actor table at some point. It is now fairly straightforward to use these ideas to construct an equivalent query for users and sections, with an extra field to indicate which ones the user does not want:

```
select section_name,name,user_id from section
left join user_section
on section.section_id = user_section.section_id
```

```
where user_section.user_id = ?
```

The result will have one row for each section. For those that the user does not want, the row will also have the user's ID; sections that the user does want will have a value of NULL for this column.

Hiding this query in the `EditionBean` will require a little more work. The easiest way to do this is to add a new property, `selected`, to the `SectionBean` and let the `EditionBean` set this property based on the results of the query. Pages can then obtain this specially marked list of sections through a new `allSections` property, which can be used in the user preferences page, as shown in Listing 7.6.[1]

[1] Of course, it would also be possible to use this property instead of `selectedSections` in the navigation by using the value of the selected flag to determine whether to show the section. However, doing it that way would have missed out on a perfect opportunity to introduce the concept of nested `selects`, which is well worth knowing.

Listing 7.6 Selecting sections

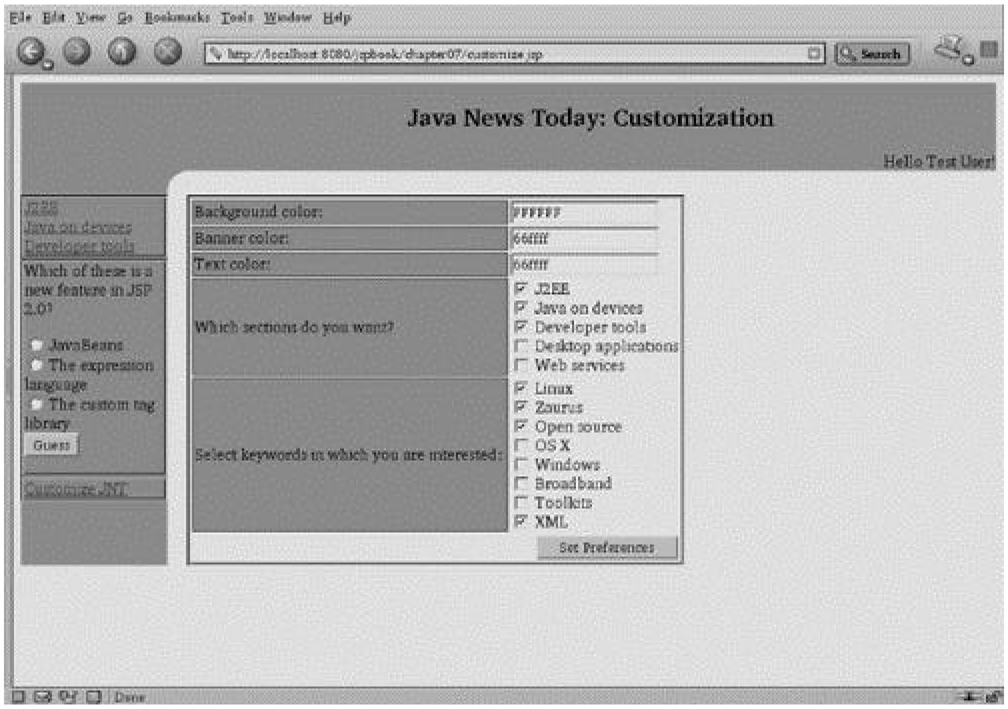
```
<jsp:setProperty
  name="user7"
  property="clearSections"
  value="true"/>

<tr>
  <td class="label">Which sections do you want?</td>
  <td>
  <c:forEach items="${edition7.allSections}" var="section">
    <input
      type="checkbox"
      name="sections"
      value="<c:out value="${section.sectionId}"/>"
      <c:if test="${section.selected}">CHECKED</c:if>>
    <c:out value="${section.name}"/><br>
  </c:forEach>
  </td>
</tr>
```

Note that options are marked as checked if the corresponding field is empty, because the user should be shown the sections wanted, but the table keeps track of those not wanted. When the form is submitted, the `UserInfoBean` will get passed an array of selected section IDs, which it must then use to add or remove entries in the `user_section` table. This requires a bit of data manipulation in the Java layer, which can be found in the code for the `UserInfoBean` on the CD-ROM accompanying this book.
The keywords list, which will work almost exactly the same as the section list, will use an outer join to select all the available keywords and simultaneously flag which ones the user has selected. The result is easily added to the user preferences page and is shown in Listing 7.7.

Listing 7.7 Selecting keywords

```
<jsp:setProperty
  name="user7"
  property="clearKeywords"
  value="true"/>


<tr>
  <td class="label">
    Select keywords in which you are interested:
  </td>
  <td>
    <c:forEach items="${edition7.allKeywords}" var="word">
      <input
        type="checkbox"
        name="keywords"
        value="<c:out value="${word.keywordId}"/>"
      <c:if test="${word.selected}">CHECKED</c:if>>
      <c:out value="${word.name}"/><br>
    </c:forEach>
  </td>
</tr>
```
The new user customization page, with these two options added, is shown in Figure 7.2.

Figure 7.2. The new customization page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter07/customize.jsp
Search
Done

Java News Today: Customization

Hello Test User!

J2EE
Java on devices
Developer tools

Which of these is a new feature in JSP 2.0?

- JavaBeans
- The expression language
- The custom tag library

Guess

Customize JNT

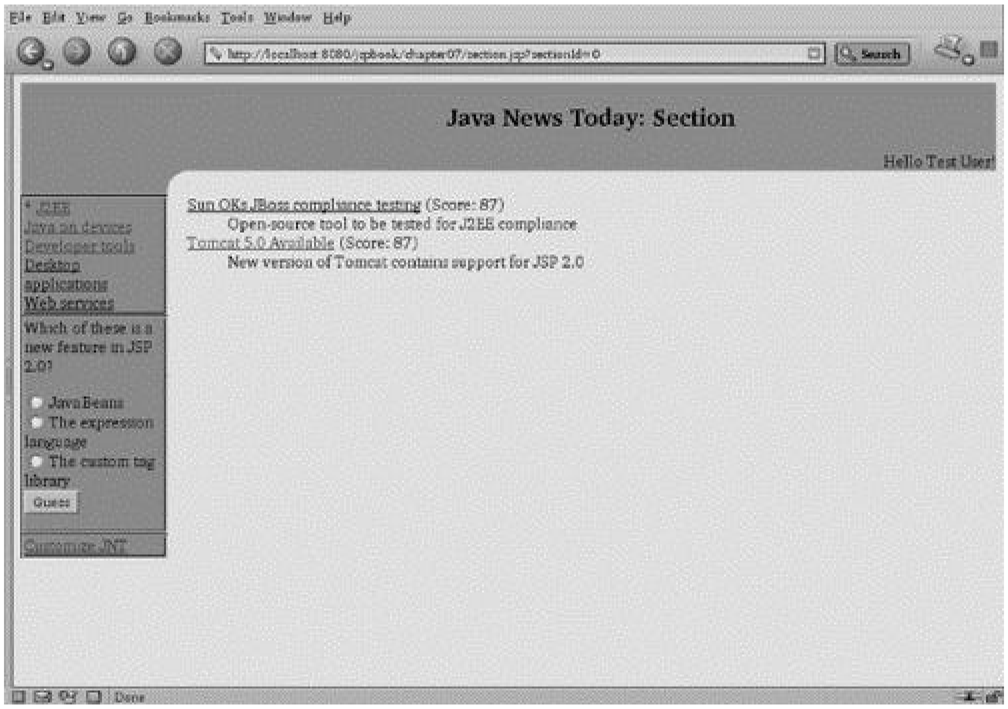
Background color:	FFFFFF
Banner color:	66ffff
Text color:	66ffff
Which sections do you want?	✓ J2EE ✓ Java on devices ✓ Developer tools ☐ Desktop applications ☐ Web services
Select keywords in which you are interested:	✓ Linux ✓ Zaurus ✓ Open source ☐ OS X ☐ Windows ☐ Broadband ☐ Toolkits ✓ XML

Set Preferences

Finally, the user's selected keywords and the set of keywords associated with each article will be used to compute for each article a score that will be displayed on the front page and the section page. Such a score can draw the user's attention to stories he or she is most likely to find interesting.

This score will be computed by examining each keyword; if both the user and the article either have or do not have that keyword, it will count for one point. The final score will then be the total number of points, divided by the total number of keywords and multiplied by 100 to produce a percentage. Of course, such a complex calculation should never be done in the view, so it will be added to the `ArticleBean`. The `ArticleBean` will therefore need to know for which user its score should be computed, but this is easily handled by a `jsp:setProperty` tag. This modifies the section page as shown in Listing 7.8, with the result shown in Figure 7.3.

Figure 7.3. The new section page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter07/section.jsp?sectionId=0
Search
Done

Java News Today: Section

Hello Test User!

* J2EE
Java on devices
Developer tools
Desktop applications
Web services

Which of these is a new feature in JSP 2.0?

JavaBeans
The expression language
The custom tag library

Guess

Customize JNT

Sun OKs JBoss compliance testing (Score: 87)
Open-source tool to be tested for J2EE compliance
Tomcat 5.0 Available (Score: 87)
New version of Tomcat contains support for JSP 2.0

Listing 7.8 The new section page

```
<jsp:useBean
  id="currentSection"
  class="com.awl.jspbook.ch07.SectionBean"/>

<jsp:setProperty
  name="currentSection"
  property="sectionId"/>
<dl>
  <c:forEach items="${currentSection.articles}"
             var="article">
    <dt><a href="<c:url value="article.jsp">
    <c:param name="articleId"
             value="${article.articleId}"/>
    </c:url>"><c:out value="${article.headline}"/></a>
    <c:if test="${user7.isLoggedIn}">
      <c:set target="${article}"
        property="userInfoId"
        value="${user7.userInfoId}"/>
      (Score: <c:out value="${article.score}"/>)
    </c:if>
    <dd><c:out value="${article.summary}"/>
  </c:forEach>
</dl>
```

7.5 Advertising

Money does not really make the world go around; gravity and angular momentum take care of that quite nicely. However, money can keep a Web site running, which at times may seem almost as important. One of the most time-tested ways for a Web site to make money is to sell space on each page to advertisers.

This is not fundamentally at odds with a usercentric site, such as Java News Today. No one enjoys the endless repetition of ads for unwanted items or constant plugs to buy shoddy or uninteresting goods. However, the Web can make shopping very easy and convenient, and an advertisement for an item a user would like but did not know about is a win for the user, the vendor, and the Web site.

The secret here is to show users only items that might appeal to them and to filter out all the advertising "noise" that most people find so irritating. In other words, the key is personalization, just as it is with content. By customizing the ads to the user, users will not be bothered with irrelevant advertising, and advertisers are generally willing to pay much more to ensure that their ads are seen only by people who might buy their products. Again, everybody wins.

Because personalization will be the driving force behind JNT's ads, it should not be surprising that ads will also be stored in the database. Once again, this means that the first step will be to design the tables by considering what information needs to be stored.

The first and most obvious element is the text of each ad. In order to match ads with users, the ads will need to be weighted according to relevant keywords, so an auxiliary table mapping ad IDs to keyword IDs will be needed. This will work in much the same way that keywords were associated with articles. Finally, most ads are sold based on a number of *impressions*; in other words, an advertiser may pay a certain amount to ensure that the ad is seen a certain number of times. The database will thus need to store the number of impressions sold, and the bean will need to decrement this count each time the ad is viewed and remove it from the system when the count reaches 0. The new tables are shown in Listing 7.9.

Listing 7.9 The advertising tables

```
create table ad (
       ad_id          int,
       impressions     int,
       text           varchar(4096)
);

create table ad_keywords (
       ad_id          int,
       keyword_id      int
);
```

Because ads are marked with keywords, just as articles are, it is possible to use the scoring mechanism that was developed for articles to compute a score for each ad. Rather then show this score directly to the user, it can be used by a new `AdManagerBean`. This bean will compute a score for every ad in the system and randomly return an ad from among the ten with the highest score. This ad will be placed in the header, which is shown in Listing 7.10.

Listing 7.10 The header, with an ad

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean id="user7"
 class="com.awl.jspbook.ch07.UserInfoBean"
 scope="session"/>

<jsp:useBean
  id="adManager7"
  class="com.awl.jspbook.ch07.AdManagerBean"
  scope="session"/>

<center>
```

```
  <h2>
    Java News Today: <c:out value="${param.title}"/>
  </h2>
</center>

<c:if test="${user7.isLoggedIn}">
  <div class="left">
    Hello <c:out value="${user7.name}"/>!
  </div>
</c:if>
```

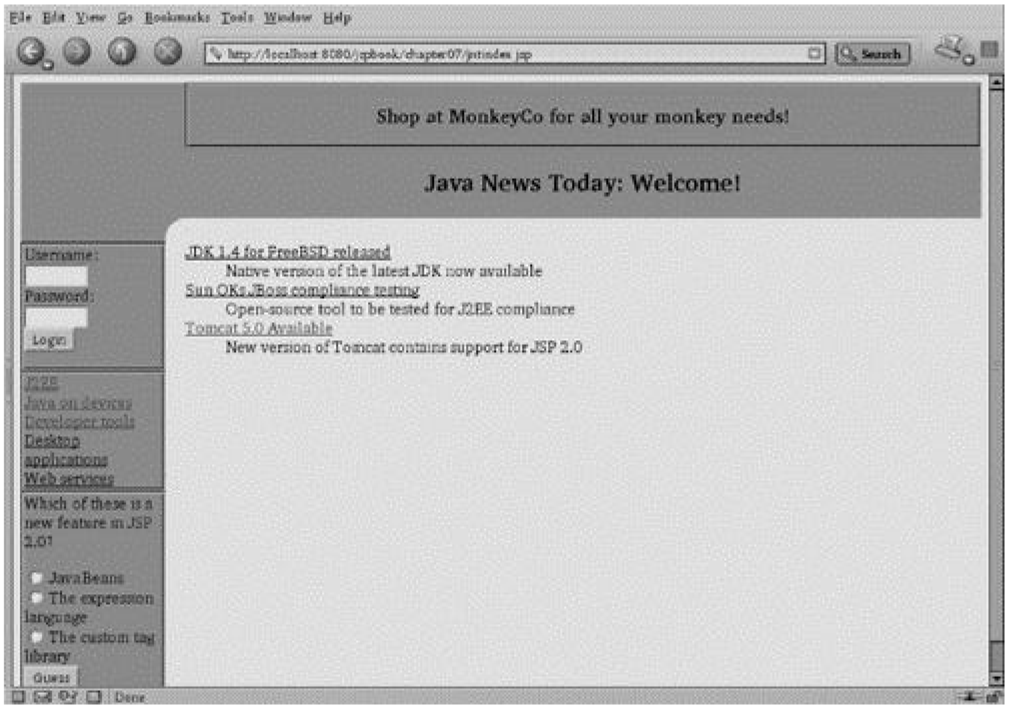
Note that the `AdManagerBean` is stored in the session. The process of computing the score may be somewhat time-consuming, and because the scores will not change much while a user is on the site, the score does not need to be recomputed on every page.

The implication of this is that when the user logs in, the `AdManagerBean` must be told who the user is in order to compute the scores, just as the `EditionBean` needed this information to select the correct sections. This is done with another little addition to the login handler page:

```
<c:set name="ads.userId" value="${user.userId}"/>
```

With such an ad in the header, the new index page will look like the one in Figure 7.4.

Figure 7.4. The new index page.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter07/jnindex.jsp
Search
Done

Shop at MonkeyCo for all your monkey needs!

Java News Today: Welcome!

Username:

Password:

Login

J2EE
Java on devices
Developer tools
Desktop applications
Web services

Which of these is a new feature in JSP 2.0?

JavaBeans
The expression language
The custom tag library

Guess

JDK 1.4 for FreeBSD released
Native version of the latest JDK now available
Sun OKs JBoss compliance testing
Open-source tool to be tested for J2EE compliance
Tomcat 5.0 Available
New version of Tomcat contains support for JSP 2.0

A slight variation to this scheme is worth mentioning. Instead of asking the user to specify manually which keywords are of interest, this information could be collected automatically. Every time a user reads an article, it would be possible to track that article's keywords and so over time build up a record of the user's behavior on the site. This profile could then be used to select advertisements using essentially the same `AdManagerBean`. Although there may be ethical concerns about the collection of information without a user's knowledge or participation, there is no technical barrier to doing so.

7.6 Summary and Conclusions

We now have a full working version of the Java News Today site. As with any site, more could always be done. The keywords could also be used for an internal search engine. To do this, one page would list all available keywords in a form, and these would be used in a `where` clause to select all articles possessing that keyword.
Similarly, more functionality could be added to the editing features. At some point, reporters will probably want to be able to make changes to old articles. This could be easily accomplished by slightly modifying the article creation page to retrieve the article

based on ID, populate the form with the current values, and then send it to a page that does an update. The ability to delete articles could be handled similarly.

There is also no page where a new reporter, section, or keyword can be added. These pages would also be straightforward, but because these things happen infrequently, it is not too much of a burden to require them to be done by issuing SQL commands directly to the database.

No doubt hundreds of other additions could be made to this basic setup, but that will always be true. A Web site should always be considered a work in progress, and JSPs make it easy to add new features or pages continually. Readers are encouraged to experiment with the site code provided on the CD-ROM.

Chapter 8. Working with XML

XML, the Extensible Markup Language, is many things to many people. XML provides a mechanism to store documents in a format that can be read and manipulated as easily by programs as by humans. XML provides the basis for programs running on different computers and operating systems to talk to one another over the Web. XML is also a language on top of which a huge number of industry-specific data formats have been created, describing everything from corporate workflow to warehouse inventories to geographic encyclopedias.

To support these and many more functions, a plethora of toolkits has become available to simplify creating, processing, and manipulating XML documents. In an important sense, XML provides another way to model data, and so great benefits are to be had by pairing XML with a view technology, such as JavaServer Pages. The JSP specification itself, along with a number of tags from the standard tag library, make this pairing possible on a number of levels.

8.1 A Brief Introduction to XML

In its most fundamental sense, XML simply provides a way to add *structure* to documents. Consider the problem that someone might face when e-mailing a list of CDs to a friend. Clearly, this e-mail will need to contain a list of artists, albums, and tracks: the same entities dealt with when constructing a CD database in Chapter 6. One approach might be to use tab stops to group information together, as in Listing 8.1.

Listing 8.1 Structuring a document with tabs

```
The Crüxshadows
    Telemetry of a Fallen Angel (1996)
        Descension
        Monsters
        Jackal-Head
    The Mystery of the Whisper (1999)
        Isis & Osiris (Life/Death)
        Cruelty
        Leave me Alone
    Wishfire (2002)
        Before the Fire
        Return (Coming Home)
        Binary
```

Although this is certainly easy for a human to read, and not even too difficult for a computer, a lot of information is lacking. The numbers in parentheses indicate the year the CD was released, but if someone is unfamiliar with that particular convention, the numbers will appear meaningless. Also, simply looking at any particular word does not indicate what it represents. "Jackal-Head" could be an artist, album, or track or even the name of a store where the CD was purchased, a club where the band played, or a restaurant. If the recipient does not know to expect a list in exactly this precise form, the file becomes meaningless because the *semantics* of the information what each piece means and how the pieces relate to one another are not present in the file.

In addition to that fundamental problem, this format has no standard. Perhaps one person will choose to use tab stops of four spaces, whereas someone else will use eight. Maybe someone will choose to have one new line between each album and two before the start of each new artist. Although none of these changes will greatly impact the ability of a person to read the file, it may complicate the creation of a program to manage such lists. For simple data, such as a CD collection that deals with only three kinds of objects and two relationships, these problems are manageable. But for much more complex systems, these problems quickly become insurmountable. In a system that manages hundreds of relationships, six tab stops might mean one thing one place in a file and another somewhere else, and determining which is appropriate cannot be done without mentally processing the whole document.

XML offers a way out of this nightmare by providing a very simple syntax with which to add semantic information to documents. This syntax looks very much like HTML, which is not surprising, as both XML and HTML have a common ancestor: SGML (Standard Generalized Markup Language).

An HTML tag, such as `<H1>...</H1>`, was originally intended to convey a semantic meaning: that the body of the tag is a level 1 header. Over time, this meaning has become diluted; today, HTML is generally used to specify *how data should be presented* rather than *what the data means*. In the terms that have been used throughout this book, HTML has gone from describing a model to describing a view.

Despite HTML's changing role, the fundamental idea of using such tags to denote meaning is still sound. The only major piece missing is a way to create new tags to describe arbitrary kinds of entities instead of a fixed set of headers, images, and so on. This is where the "extensible" in Extensible Markup Language comes in.
Creating an XML document can be as simple as deciding what tags to use and how they relate. Listing 8.1 could be rewritten in a much better, more structured way using XML, as shown in Listing 8.2.

Listing 8.2 Structuring a document with XML

```
<?xml version='1.0' encoding='iso-8859-1'>

<artist name="The Crüxshadows">

  <album name="Telemetry of a Fallen Angel" year="1996">
    <track>Descension</track>
    <track>Monsters</track>
    <track>Jackal-Head</track>
  </album>
  <album name="The Mystery of the Whisper" year="1999">
    <track>Isis & Osiris (Life/Death)</track>
    <track>Cruelty</track>
    <track>Leave me Alone</track>
  </album>
  <album name="Wishfire" year="2002">
    <track>Before the Fire</track>
    <track>Return (Coming Home)</track>
    <track>Binary</track>
  </album>
</artist>
```

As this listing shows, the rules of XML are very much like those of HTML, despite some important differences in terminology. First, the file starts with a declaration of what kind of document it is and the character set it is using.[1] In XML, the entities in angle brackets, or tags in HTML, are called *nodes*. Every node has a *name*, which is the primary identifier. Listing 8.2 has nodes named `artist`, `album`, and `track`. Nodes are allowed to have *attributes*, as in HTML. The `album` node has the attributes `name` and `year`. The use of the word `name` as an attribute may be a bit misleading but is seen quite often. Here, `name` refers to the name of the album, not the name of the node.

[1] Listing 8.2 uses ISO-8859-1 in order to support the umlaut. Documents that use only ASCII characters will more likely use the UTF-8 character set.

Nodes can be nested arbitrarily, but a document can, and must, have one and only one top-level node, called the *root node*. In Listing 8.2, the `artist` node is the root. It would not be legal to list the CDs from another artist in this same document by simply adding a new `artist` node. Instead, both `artist` nodes would need to be contained within another node, which might be called `collection`. Besides containing other nodes, a node can contain a block of plain text, as the `track` nodes in Listing 8.2 do.

More freedom is possible when deciding on the format of an XML document. For example, the name of each track could be placed in an attribute, such as `<track name="Binary"/>`, instead of in the body of the track node. The choice is completely free, although experience will often suggest one way over another. Note that if a node has no body, it *must* end with a slash?TT>/> to indicate that the file does not have a corresponding close tag.

Listing 8.2 constitutes what is called a *well-formed* XML document, meaning that it follows the rules of XML syntax, such as providing a single root node, properly matching opening and closing tags, and so on. Beyond following these simple rules, an XML document can and should have much more information. Listing 8.2 implies certain things about the nodes that are used, such as the existence of the `artist`, `album`, and `track` nodes; that `artist` may have a `name` attribute; and so on. However, these rules are not explicitly stated; nor does the listing specify any others that may be important to enforce. Placing an `album` node within a `track` node would still result in well-formed XML, but this information would now be meaningless in context.

The mechanism to fix this is called a *document type definition* (DTD). The DTD describes all the nodes that a document will use, their attributes, and their relationships. This information, and more, could also be specified using an *XML schema*; however, schemas are beyond the scope of this book, as are the art and science of creating DTDs. A possible DTD for describing a CD collection is shown in Listing 8.3.

Listing 8.3 The document type definition

```
<!ELEMENT artist (album*)>
```

```
<!ATTLIST artist name CDATA #REQUIRED>

<!ELEMENT album (track*)>
<!ATTLIST album name CDATA #REQUIRED>
<!ATTLIST album year CDATA #REQUIRED>

<!ELEMENT track (#PCDATA)>
```

Once such a DTD is created, the document can reference it with a single line at the top:

```
<!DOCTYPE artist SYSTEM "cd.dtd">
```

With the inclusion of a DTD, like Listing 8.3, an XML document can be not only well formed but also *valid*. Such a document not only is syntactically correct but also follows all the rules and is therefore semantically correct. Flipping tags around in a meaningless way would now render a document invalid. This check can be done very early, when the document is first parsed, avoiding any potential errors that could result from bad data getting farther into the system. In addition, providing a DTD will often allow the data to be parsed and represented more efficiently. Many XML editors are also able to read a DTD and can ensure that the rules are followed while the document is being created or changed.

8.2 Using XML in JSPs

As an XML document is merely a bunch of text, creating one through a JSP is no more difficult than creating an HTML document. Listing 8.4 shows a JSP that retrieves CD information from a database and generates the CD collection from Listing 8.2.

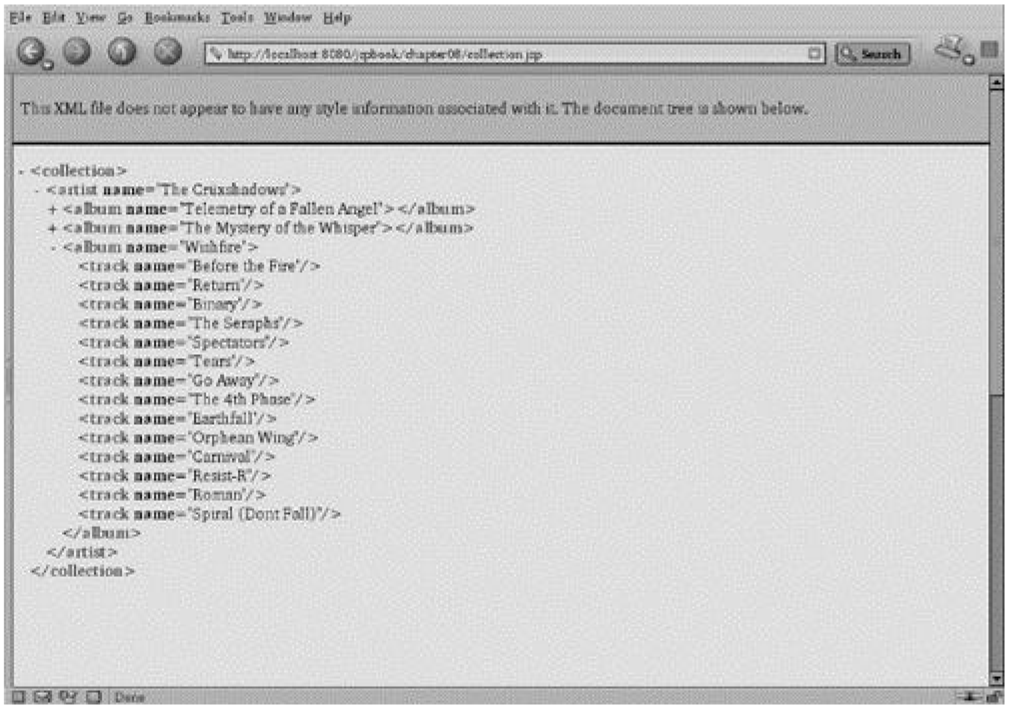
Listing 8.4 Generating XML with a JSP

```
<%@ page contentType="text/xml" %>

<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>

<jsp:useBean
  id="artist"
  class="com.awl.jspbook.ch08.ArtistBean"/>
```

```
<jsp:setProperty
  name="artist"
  property="*"/>

<collection>
  <artist name="<c:out escapeXml="false"
                    value="${artist.name}"/>">
  <c:forEach items="${artist.cds}" var="cd">
    <album name="<c:out value="${cd.name}"/>">
      <c:forEach items="${cd.tracks}" var="track">
      <track name="<c:out value="${track.name}"/>"/>
      </c:forEach>
    </album>
  </c:forEach>
  </artist>
</collection>
```

In almost all respects, this example is identical to Listing 6.6, the major difference being the use of XML tags here instead of HTML. As it will not be returning an HTML document, it is important that this page notify the browser what kind of data to expect. This is accomplished by the use of the `page` directive at the top. Telling the browser that it will be getting an XML document allows the browser to present the data properly. For example, both Mozilla and Internet Explorer have a special mode that allows users to open and close portions of XML documents interactively. In Figure 8.1, which shows Mozilla's view of such data, a `+` in front of a node indicates that it may be expanded by clicking it; conversely, `-` means that the node can be collapsed.

Figure 8.1. The browser view of an XML document.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter08/collection.jsp
Search
Done

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <collection>
  - <artist name="The Cruxshadows">
    + <album name="Telemetry of a Fallen Angel"></album>
    + <album name="The Mystery of the Whisper"></album>
    - <album name="Wishfire">
        <track name="Before the Fire"/>
        <track name="Return"/>
        <track name="Binary"/>
        <track name="The Seraphs"/>
        <track name="Spectators"/>
        <track name="Tears"/>
        <track name="Go Away"/>
        <track name="The 4th Phase"/>
        <track name="Earthfall"/>
        <track name="Orphean Wing"/>
        <track name="Carnival"/>
        <track name="Resist-R"/>
        <track name="Roman"/>
        <track name="Spiral (Dont Fall)"/>
      </album>
    </artist>
  </collection>
```

More interesting is that the output of this page contains all the data from the database, and the DTD contains almost all the information present in the SQL schema from Listing 6.1. This suggests a deep connection between databases and XML, and because there is already a known relationship between databases and JavaBeans, this would suggest that all three are in some sense interchangeable.

To an extent, this is true. Just as tools can create beans from databases, tools can create database schemas from XML DTDs and vice versa. Tools can also convert between DTDs and beans, most notably Sun's JAXB toolkit, available at http://java.sun.com/xml/jaxb/.

All three types of relationships are ways to store and manipulate data. Each one has strengths that make it well suited to particular tasks. Databases are appropriate for storing large quantities of data and retrieving it based on arbitrary criteria. XML is appropriate for storing and transmitting relatively small amounts of data and for data that needs to be translated programmatically into other forms. Beans, as seen numerous times, are well suited for moving data from the underlying model to the view or, more generally, for providing access to the model from other code.

8.3 Selecting Data from an XML Document

If the beans from Chapter 6 were used in a JSP to navigate through a collection of cds, the page might use an expression such as

```
collection.artist[0].album[2].track[5]
```

A similar but more powerful expression language for navigating XML documents is *XPath*, which plays a major role in the way JSPs use XML.

Syntactically, XPath resembles traversing a set of beans except that the separator is a slash (/) instead of a dot (.), and arrays start counting from 1, not 0. Therefore, the XPath expression that does the same thing as the preceding bean expression would be

```
/collection/artist[1]/album[3]/track[6]
```

Note that the expression also starts with a leading slash.

XPath and beans diverge beyond the simple mechanism used to select a specific element. One powerful feature of XPath is its ability to specify only part of an expression, and such a partial expression will retrieve all elements that match. The simplest example of this would be to leave off the last set of square brackets, as in

```
/collection/artist[1]/album[3]/track
```

This specifies all tracks on the third album of the first artist. This idea can be extended by leaving off more array specifiers. The following, for example, would return all tracks on all albums by the first artist:

```
/collection/artist[1]/album/track
```

Indexed and nonindexed elements can be freely mixed. The following would return the second track on each album:

```
/collection/artist[1]/album/track[2]
```

Portions of a path can even be omitted entirely by using two slashes, as in `//track`, which would return all tracks from albums by all artists.

Attributes can be specified by prefacing the name with an at sign (`@`), so in order to get the name of the first artist, the expression would be `/collection/artist[1]/@name`.

Attributes can also be used in brackets to restrict the set of returned data. The expression `//album[@name='Wishfire']/track` would return all tracks from all albums named "Wishfire," of which there happens to be only one.

Much more could be said about XPath, but this will be sufficient for the remainder of this book. Readers interested in the full specification can find it at http://www.w3.org/TR/xpath; a nice tutorial is online at http://www.zvon.org/xxl/XPathTutorial/General/examples.html.

8.4 Processing XML in JSPs

The standard tag library provides a number of tags that make it easy and natural to move through XML documents using XPath. An example of these tags in action is shown in Listing 8.5.

Listing 8.5 Using XPath expressions in a JSP

```
<%@ taglib prefix="x"
    uri="http://java.sun.com/jstl/xml" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>



<c:import
url="http://localhost:8080/jspbook/chapter08/
          collection.jsp"
var="xml"/>

<x:parse xml="${xml}" var="doc"/>

Albums by <x:out select="$doc//artist[1]/@name"/>:<br>
<ul>
  <x:forEach select="$doc//artist[1]/album" var="album">
    <li><x:out select="$album/@name"/>
  </x:forEach>
</ul>
```

First, note that this example loads a new portion of the standard tag library, which is imported with the prefix `x`. The first new tag used in this example, `c:import`, is not technically a part of the XML tags but is often used in conjunction with them. The tag `c:import` works like a superenhanced version of the `jsp:include` tag. Amazingly, `c:import` can grab data from *anywhere*, not only from the site where the page lives. This makes it possible for sites to include content from other sites, although in general this should be done only with the other site's knowledge and permission. This ability works especially well in conjunction with XML, as will soon be demonstrated.

The c:import tag stores the data it has read in a variable rather than automatically sending it to the user. This makes it possible to process this data before the user sees it, which is what will be done here. In this case, the data from the collection page from Listing 8.1 has been put into a variable called xml. This data could then be shown directly to the user with a simple <c:out value="${xml}"/>.

Rather than display this data, it is instead passed to another tag, x:parse, the first of the new XML tags. This tag takes a block of XML and processes it internally into a form that can be used more efficiently. The results of this conversion are stored in yet another variable, which has been called doc.

Next, data is extracted from this internal representation with the x:out tag. This tag works somewhat like c:out but obtains the value to display from a combination of the expression language and an XPath expression. The JSP XML tags allow the beginning of a select expression to start with a number of expression language identifiers, such as the variable doc that was created with the x:parse tag. Immediately following that can be any valid XPath expression, which will be used to pull data from the variable. Here, the pages gets the name of the first artist in the collection.

Next is an x:forEach tag, which is to c:forEach what x:out is to c:out. The x:forEach tag will repeat some action for every element returned by an XPath expression, which in this case is all albums from the first artist. As with c:forEach, each time through the loop, the current value can be assigned to a variable, in this case one called album.

Within the body of the x:forEach tag is another x:out, which displays the value of the name attribute for each album. Because album holds each of the XML album tags, the XPath portion of this second x:out tag does not need the full path starting from the top but instead needs to know only how to get to the name attribute from each album tag.

Note that it would also have been possible to write this loop as

```
<x:forEach select="$doc//artist[1]/album/@name"
          var="name">
  <li><x:out select="$name"/>
</x:forEach>
```

This loop would have the effect of looping over all album names instead of over all albums. This works, as all the page will be showing is the name, but if it had to show both the name and the year the album was released, the page would have had to loop over the albums and then use two x:out tags to display the two different attributes.

The x:if, x:choose, x:when, and x:otherwise tags do essentially the same things as their counterparts from the c library, except that each can take an XPath expression instead of a value from the expression language. This functionality was covered in Chapter 4 and so will not be repeated here.

8.5 Formatting XML

Listing 8.5 does two separate but related things. It pulls out a chunk of an XML document, using the XPath expression `//artist[1]/album`, and then builds some HTML out of the values in the XML in the body of the `x:forEach`. This second part, translating XML into another format, is so common and so important that a whole new language?span class="docEmphasis">XSLT (Xtensible Stylesheet Language

Transformations) was developed to make it easier.

This language uses many of the ideas that have already been discussed. To begin, consider what would be needed in order to find every artist's name from a CD collection in an XML document and output the string "Albums for" followed by the name, enclosed in an `H1` tag. This would pose no challenge: Simply specify the set of nodes to loop over with an `x:forEach` tag, using `//artist` as the set of items. Then, within the `x:forEach` tag, obtain the desired string, using `<x:out select="@name"/>`.
XSLT takes these same concepts but replaces the idea of selecting a set of tags and then iterating them, substituting with the notion of *patterns*. Each clause of an XSLT file specifies a pattern to find in the XML file, such as all artists, all albums with a given name, or any other possibility XPath provides. A provided output template may include elements selected from the XML that matched the input. For example, the XSLT that will format artist names as desired is

```
<xsl:template match="artist">
  <h1>Albums by <xsl:value-of select="@name"/></h1>
</xsl:template>
```

This looks like the corresponding JSP code, with `xsl:template` playing the role of the `x:forEach` and `xsl:value-of` replacing the `x:out`. It is important to note the conceptual difference, however; `xsl:template` is not an iteration operator and does not perform an activity for every element of a set. Instead, it provides a rule saying that whenever and wherever the XPath expression given as `match` is found, the body will be processed.
A similar clause could be added to put album names in level 2 headers:

```
<xsl:template match="album">
  <h2><xsl:value-of select="@name"/></h2>
</xsl:template>
```

However, one more thing must be done to make both of these clauses fit together. The rule given for `artist` specifies that a certain string should result and that no other actions should be taken. To get it to continue examining the rest of the document, XSLT must be told to do so, which can be done by adding the following after the string:

```
<xsl:apply-templates select="album"/>
```

This indicates that XSLT should continue processing the `album` elements within the `artist`. Order is important here; if `xsl:apply-templates` appeared before the string, the result would show *first* the albums and then the artist.

Listing 8.6 rounds out the set of translations by putting track names in a bulleted list.

Listing 8.6 The full XSLT file

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


<xsl:template match="artist">
  <h1>Albums by <xsl:value-of select="@name"/></h1>
  <xsl:apply-templates select="album"/>
</xsl:template>


<xsl:template match="album">
  <h2><xsl:value-of select="@name"/></h2>
  <ul>
    <xsl:apply-templates select="track"/>
  </ul>
</xsl:template>


<xsl:template match="track">
  <li><xsl:value-of select="@name"/></li>
</xsl:template>
```

```
</xsl:stylesheet>
```

Note that the rule for `album` also needs an `xsl:apply-templates` in order to process the tracks.

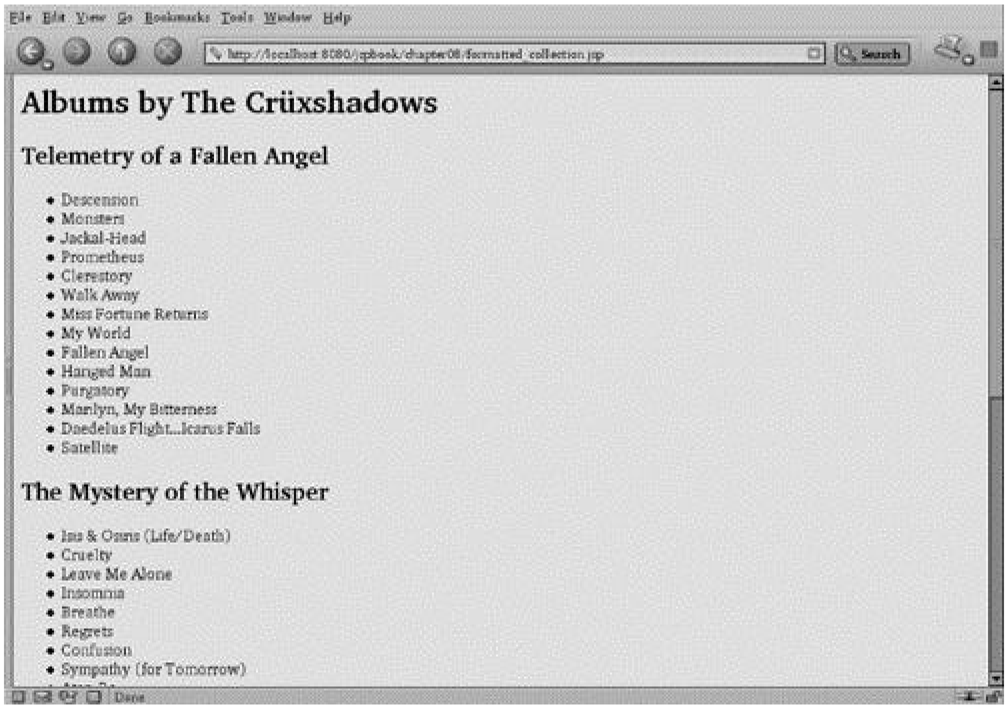
Once an XSLT file has been defined, using it from a JSP is almost ridiculously easy! Such a page is shown in Listing 8.7.

Listing 8.7 Using XSLT from a JSP

```
<%@ taglib prefix="x"
   uri="http://java.sun.com/jstl/xml" %>
<%@ taglib prefix="c"
   uri="http://java.sun.com/jstl/core" %>

<c:import
url="http://localhost:8080/jspbook/chapter08/
          collection.jsp"
var="xml"/>


<c:import
  url="http://localhost:8080/jspbook/chapter08/style.xsl"
  var="xslt"/>


<x:transform xslt="${xslt}" xml="${xml}"/>
```

In this example, the XML and XSLT files are loaded using `c:import` tags. Then the transformation is performed and the result displayed with the new `x:transform` tag. As the final outcome of the transformation is HTML, the content type need not be set, and a browser will be able to render it in the usual way, as shown in Figure 8.2.

Figure 8.2. The result of an XSLT translation.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter08/formatted_collection.jsp
Search
Done

Albums by The Crüxshadows

Telemetry of a Fallen Angel

- Descension
- Monsters
- Jackal-Head
- Prometheus
- Clerestory
- Walk Away
- Miss Fortune Returns
- My World
- Fallen Angel
- Hanged Man
- Purgatory
- Marilyn, My Bitterness
- Daedelus Flight...Icarus Falls
- Satellite

The Mystery of the Whisper

- Isis & Osiris (Life/Death)
- Cruelty
- Leave Me Alone
- Insomnia
- Breathe
- Regrets
- Confusion
- Sympathy (for Tomorrow)

In a sense, this process has split the view layer into two smaller components. One, the XML, provides data from the model to the view. The second, the XSLT, contains all the presentation information. Using pure JSPs, these two actions are typically intertwined, with some bean tags getting data from the model and various iteration and conditional tags munging that data into the desired presentation.

Both of these operations may legitimately be considered part of the view, and so having them in the same JSP is not a bad design. However, splitting them into separate components offers some new possibilities. It is often true that splitting pieces of a complex system into separate modules makes it easy to add new functionality.

In this case, one new piece of functionality is the ability to change the apparence of a page easily without changing any of the underlying implementation. Listing 8.8 shows an alternative XSLT file that uses tables to format a CD collection instead of itemized lists.

Listing 8.8 An alternative style

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


 <xsl:template match="album">
  <table border="1">
```

```
    <tr>
      <td><xsl:value-of select="parent::artist/@name"/></td>
      <td><xsl:value-of select="@name"/></td>
    </tr>
  <xsl:apply-templates select="track"/>
 </table><p></p>
</xsl:template>


<xsl:template match="track">
  <tr><td colspan="2"><xsl:value-of
                select="@name"/></td></tr>
</xsl:template>


</xsl:stylesheet>
```

This XSLT file does nothing for artist nodes. When it encounters an album, it creates a new table, the first row of which will have a column for the artist name and another for the album name. The artist name is obtained with a new kind of XSLT expression: `parent::artist/@name`. The `parent::` portion indicates that the value should be obtained from the parent node, that is, the node that contains the current one. Because album nodes are contained within artist nodes, this will get the artist; from there, getting the name is done as usual.

Now that a second style has been defined, Listing 8.7 can be easily modified to switch between them, based on user preference, as shown in Listing 8.9.

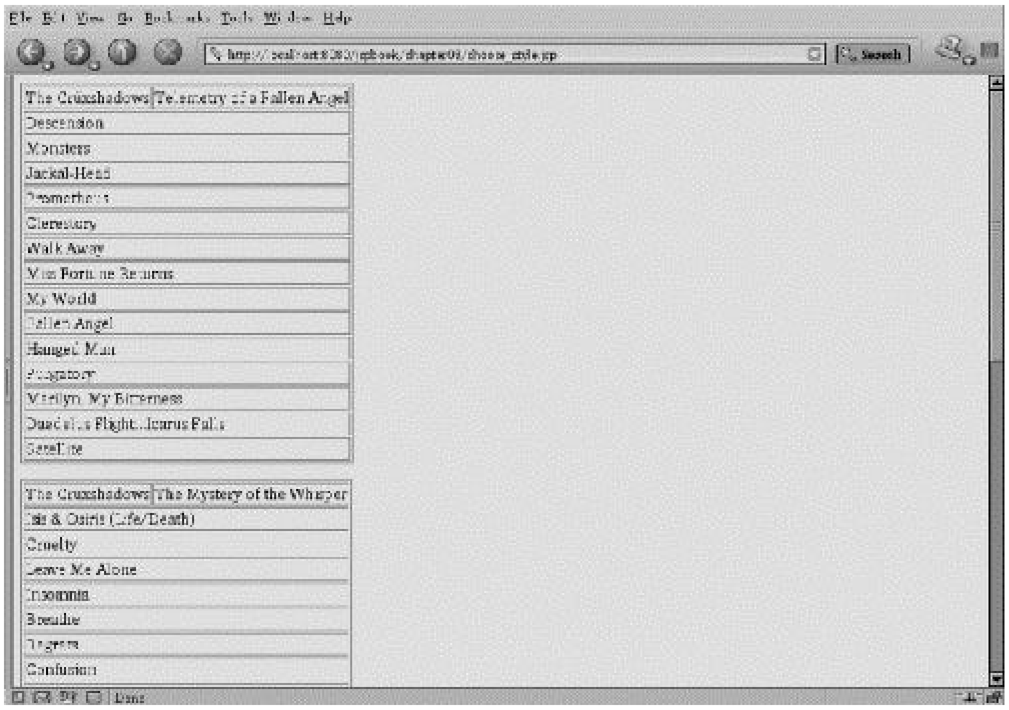
Listing 8.9 Allowing the user to choose a style

```
<%@ taglib prefix="x"
    uri="http://java.sun.com/jstl/xml" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jstl/core" %>


<c:import
url="http://localhost:8080/jspbook/chapter08/
```

```
           collection.jsp"
var="xml"/>



<c:choose>
  <c:when test="${param.style == 'list'}">
    <c:import
    url="http://localhost:8080/jspbook/chapter08/style.xsl"
    var="xslt"/>
  </c:when>
  <c:otherwise>
    <c:import
   url="http://localhost:8080/jspbook/chapter08/
             tablestyle.xsl"
   var="xslt"/>
  </c:otherwise>
</c:choose>



<x:transform xslt="${xslt}" xml="${xml}"/>
```

This example simply uses a `c:choose` tag to load one of two XSLT files into the `xslt` variable, which will then be used by the `x:transform` tag. The result of formatting with the table-based XSLT file is shown in Figure 8.3.

Figure 8.3. An alternative XSLT translation.


File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/jspbook/chapter08/choose_style.jsp
Search
Done

The Cruxshadows	Telemetry of a Fallen Angel
Descension	
Monsters	
Jackal-Head	
Desmorthous	
Clerestory	
Walk Away	
Miss Fortune Returns	
My World	
Fallen Angel	
Hanged Man	
Purgatory	
Marilyn, My Bitterness	
Daedalus Flight...Icarus Falls	
Satellite	

The Cruxshadows	The Mystery of the Whisper
Isis & Osiris (Life/Death)	
Cruelty	
Leave Me Alone	
Insomnia	
Breathe	
Regrets	
Confusion	

8.6 Java News Today and XML

In order to use these new features, Java News Today will be creating XML versions of a few of its pages, notably a stripped-down version of the index and section pages. These pages will not contain some of the dynamic elements, such as the login form or quiz. Both of these elements can be accomplished with a single JSP, as the only difference is whether to obtain the article list from the section or the edition. This JSP is shown in Listing 8.10.

Listing 8.10 The XML version of JNT

```
<%@ page contentType="text/xml" %>
<%@ taglib prefix="c"
   uri="http://java.sun.com/jstl/core" %>


<jsp:useBean
  id="currentSection"
  class="com.awl.jspbook.ch07.SectionBean"/>
```

```
<jsp:useBean
  id="edition8"
  scope="request"
  class="com.awl.jspbook.ch07.EditionBean"/>


<jsp:setProperty
  name="currentSection"
  property="sectionId"/>


<c:choose>
  <c:when test="${empty currentSection.sectionId}">
    <c:set var="articles"
          value="${edition8.recentArticles}"/>
    <c:set var="sectionName" value="Java News Today"/>
  </c:when>
  <c:otherwise>
    <c:set var="articles"
         value="${currentSection.articles}"/>
    <c:set var="sectionName"
          value="${currentSection.name}"/>
  </c:otherwise>
</c:choose>


<javaNewsToday>
  <section><c:out value="${sectionName}"/></section>
  <articles>
    <c:forEach items="${articles}" var="article">
      <article>
        <headline>
          <c:out value="${article.headline}"/>
        </headline>
        <summary>
```

```
          <c:out value="${article.summary}"/>
        </summary>
         <date>
          <c:out value="${article.createdDate}"/>
        </date>
      </article>
    </c:forEach>
  </articles>
</javaNewsToday>
```

The block of code at the top of this example determines whether the user is requesting a section or the index page, based on whether a `sectionId` has been provided. The code block then stores a list of articles in a variable called `articles` and a section name in `sectionName`. This technique has not been seen before, but it works very much like the variables created by the `c:import` tag. The rest of the page is a standard `c:forEach` used to create the XML.

The goal of this XML representation of the site is not to replace the existing pages, which are working well enough as they are. Instead, this XML layer allows Java News Today to offer its content to *other sites*, as well as directly to users.

As mentioned, the `c:import` tag can pull pages from anywhere, not just locally. This means that if another Java site say, javamonkeys.com were interested in giving its users access to Java News Today's articles, it could import the XML file and then format it with its own XSLT file in order to make it mesh seamlessly with the rest of its site.

This ability to provide one site's content to other sites is called *syndication* and is quite popular. When properly done, it can benefit both sites. In this case, javamonkeys.com can offer users additional reasons to visit its site; in exchange, these additional users will learn about Java News Today and may wish to visit the JNT site directly. It is also possible for sites to charge each other for syndicated content or to swap advertising banners.

What makes all this possible is that XML is a standard format. Javamonkeys.com doesn't need to know anything about JNT's database layout, and JNT doesn't need to allow javamonkeys.com to access its database directly, which could be a security risk.

8.7 Summary and Conclusions

XML wraps data in an extensible set of tags so that documents can carry not only the raw data but also information about what the data means and how it interrelates. By providing a standard mechanism to store and transmit data, XML greatly simplifies the process of communication between different systems or different parts of the same system. Creating XML files with JSPs is no more difficult than creating HTML; all the same principles apply. Once a JSP has constructed an XML representation, this data can be searched, tested, or iterated, using the XPath language and XML equivalents of many of the tags in the `c` portion of the standard tag library. In addition, XML data can be transformed from one form into many others, including HTML, through XSLT.

8.8 Tags Learned in this Chapter

`c:import` Imports data from any URL and stores it in a variable
Parameters:
  `url`: The URL to load
  `var`: The name of the variable in which the data should be stored
Body: Optional; if present, may be any number of `c:param` tags, whose values will be sent to the named URL

`x:parse` Transforms an XML document to an internal form that can be used by other tags
Parameters:
  `xml`: An expression specifying where the XML text is stored
  `var`: The name of the variable in which the resulting internal form should be stored
Body: If no `xml` parameter is specified, the XML text may be put in the body.

`x:out` Displays a value
Parameters:
  `select`: An XPath expression to be evaluated and displayed
Body: Arbitrary JSP code

`x:forEach` Repeats a section of the page for every item in an array

Parameters:

`items`: An expression specifying the array to use, most likely a bean property

`var` The name of the variable with which each element in the array will be referred

Body: Arbitrary JSP code

`x:if` Conditionally includes a portion of the page

Parameters:

`test`: An expression that should be a logical test of a property, which may include XPath elements

`var`: if present, names a variable in which the result of the expression will be stored

Body: Arbitrary JSP code

`x:choose` Includes one of several portions of a page

Parameters: None

Body: Arbitrary number of `x:when` tags and, optionally, one `x:otherwise` tag; nothing else is permitted

`x:when` One possibility for an `x:choose` tag

Parameters:

`test`: An expression that should be a logical test of a property, which may include XPath elements

Body: Arbitrary JSP code

`x:otherwise` The catch-all possibility for an `x:choose` tag. If none of the expressions in the `when` tags evaluate to `true`, the body of the `otherwise` will be included.

Parameters: None

Body: Arbitrary JSP code

Chapter 9. A Small Cup of Java

At many points throughout this book, examples could be discussed only so far before running into a boundary. Those boundaries were frequently demarcated by the transition from the view into the model, but the real issue is that on one side of this boundary lives the JSP code and on the other side the unexplored territory of Java code.

In part, this division has been made deliberately; one of the impetuses for the creation of JSPs and for separating working into a model part and a view part was to allow page authors to create interactive, dynamic Web sites without needing to know any Java. But in another sense, all divisions between knowledge are arbitrary, and page authors could benefit from knowing at least some Java.

To those who have never programmed before, programming may seem like a mystical black art, beyond the ken of mere mortals. OK, it may be a *bit* spooky, but there is no reason why everyone cannot learn to program. This chapter will not teach programming; nor will it completely cover the Java language. Many good books will do this, including *Introduction to Programming Using Java: An Object-Oriented Approach, Java 2 Update* by David Arnow and Gerald Weiss and *The Java™ Tutorial, Third Edition: A Short Course on the Basics* by Mary Campione, Kathy Walrath, and Alison Huml (Pearson Education, 2001). A number of colleges and training centers offer courses in Java for programmers and nonprogrammers.

This chapter will also not explain how to use Tomcat or any other development environment. See the documentation for the relevant product for this information or the accompanying CD-ROM for information on setting up Tomcat.

What this chapter *will* do is introduce enough Java basics to follow the code that appears throughout the subsequent chapters. These basics should also be as much Java as most JSP authors will ever need, although, of course, it is never a bad idea to know more. Learning to program in Java will enable JSP authors to create new beans and other utility classes, as well as write servlets instead of JSPs when appropriate.

9.1 Expressions

Conceptually, an *expression* is simply a sequence of characters representing a value. Such expressions have already been encountered via the expression language used in many tags. For example, `article.author.name` is an expression representing the name of an author who wrote an article, as used in the Java News Today site. Expressions in Java are basically the same thing, although their syntax and meaning are different from expressions in the tag expression language.

To start with, here is an incredibly simple Java expression:

```
2
```

Obviously, this expression represents the number 2. Expressions can be more complex:

```
((8 / (2 * 4)) + 3) - (8/4)
```

This expression too represents the number 2. In this expression, `+`, `-`, `*` and `/` are called *operators*, as they perform an operation on two expressions to produce a result. In a numeric context, these operators do the expected things.

9.2 Types

The numbers in all the previous examples have been *integers*: numbers with no decimal part.[1] Division on integers works slightly differently from division on numbers in the real world. For example, the following evaluates to 3, as 3 is the largest integer that, when multiplied by 2, is less than 7:

[1] In computer science, *integer*, *real*, and similar terms typically do not mean exactly the same things as the corresponding terms used in mathematics. For one thing, the set of integers in Java is not infinite.

```
7 / 2
```

If a program wanted this to evaluate to 3.5, the numbers should be Java *doubles*, or double-precision floating-point numbers, instead of integers. This would be expressed by using decimal points, as in

```
7.0 / 2.0
```

This example illustrates an important and fundamental aspect of the Java programming language. Everything in Java has a *value*, such as 2 or 3.1415, and a *type*. This was also true in SQL; when defining a column, it is necessary to assign a name, such as `article_id`, as well as a type, such as `int`. If Java is expecting an expression of a certain type in a particular context, that will restrict the possible values that can be used in that context.

Java supports a number of built-in, or *primitive*, types. We have already seen doubles and integers, which Java calls *ints*. Java can also manipulate text, using the `String` type. Strings are represented by surrounding them with quote marks:

```
"This is a Java string!"
```

It is important to keep in mind that `"2"` and `2` are very different things to Java, even though they may look the same. Do not be fooled by what they look like; the types are different, which is what matters.

String expressions can also be more complex:

```
"This is a " + "Java string!"
```

Here, the `+` operator is used to denote *concatenation*, which simply means appending two strings together. The result of this expression is the same as the previous one. This is the only important instance in which an operator does two different things based on the type of the expression it is operating on.[2] Technically, `+` applied to two doubles does something different from `+` applied to two integers, but for the most part, differences like this can be ignored.

[2] Some languages allow operators to be defined and created by programmers, allowing for *overloading*. Java is not one of these languages.

Under some circumstances, Java automatically converts part of an expression from one type to another. In the following, for example, the 2 will be internally converted to 2.0, and the result of the expression will be a double:

```
7.0 / 2
```

The full set of type-conversion rules is available in any book on Java, but the general rule is that Java never automatically goes from one type to another with less information. The double-precision floating-point number 2.0 has more information than 2, so this conversion can happen automatically.

Java will also automatically convert most types to a `String`, when the result of an expression should be a `String`. The following expression will evaluate to the string `"22"`:

```
"2" + 2
```

The second `2` is first converted to a `String`, yielding `"2"`, which will then be appended to the first `String`.

It is also possible to convert a number of one type explicitly into another type, which is done by specifying the target type in parentheses before the value:

```
(double) 3
```

This example specifies a double number built from 3, which will be equivalent to 3.0.

It is also possible to do conversions that lose data in this way, such as

```
(int) 6.75
```

This will yield the value `6`, the result of simply chopping off the decimal part. Java would never perform such a conversion automatically, but it is perfectly valid for a programmer to do so.

This kind of moving from one type to another is called *casting*. It may help to think of an actor being "type cast," meaning forced into a particular role. Casting will become very important once objects and classes have been introduced.

9.3 Storing Values

All the values seen so far have been *literals*, meaning that they represent themselves. Java also supports *variables*, which can be thought of as boxes that can contain any value and whose value can be changed throughout the course of a program.

Before a variable can be used, it must be *declared*, which will tell Java the name and type of the variable. A typical declaration might look like

```
int aNumber;
```

The semicolon designates the end of a Java *statement*, which is slightly different from an expression. A statement does something, whereas an expression has a value.

The preceding statement creates a new "box" called `aNumber`, which can hold an integer. A value can be placed in this box with an *assignment*, which might look like

```
aNumber = 2 + 4 - 7;
```

This sets `aNumber` to -1. Once it has a value, a variable can be used in expressions just like any other value, such as the following:

```
2 - aNumber
```

This represents the number `2 - (-1)`, which equals 3.

Variables can also be changed as many times as desired:

```
aNumber = 1;
aNumber = aNumber + 1;
```

After these two lines are encountered in a program, `aNumber` will be `2`. Note that this is starting to look like algebra, although the symbols have a subtly different meaning. In algebra, the second statement would be meaningless, as a number can never be equal to itself plus 1. In Java, however, this statement means "compute `aNumber` + 1, which is 2, and then put that value back in the box called `aNumber`."

If a statement or expression tries to mix types in a way that Java cannot automatically resolve, an error will be reported when the programmer tries to convert the program into a form that can be run. Either of these statements will cause an error:

```
aNumber = 2.0;
aNumber = "Hi there";
```

9.4 Method Calls

Java allows programmers to create *methods*, which can be thought of as black boxes with some number of inputs and one output. Values of particular types are dropped into the inputs, and a value of a, possibly different, type comes out of the output. If a method called `max` has been defined, which takes two integers and returns the greater of the two, the following expression will have the value `8`:

```
max(8,3)
```

Method calls can be used in other expressions; the following will set `aNumber` to 10:

```
aNumber = max(8,3) + 2;
```

The values given to a method its *arguments* can also be arbitrary expressions. The following expressions are both valid:

```
max(3+2,12)
max(11,max(13,20))
```

However, the following is not, because `max()` can take only integer arguments:

```
max(2,"some string")
```

Some methods do not return a value. These methods can be used as statements. One very common such method is `System.out.println()`. For the moment, don't worry about the apparently strange name of this method; the important thing is what it does: print its argument to the user's terminal or window. This method might be used in any of the following ways:

```
System.out.println(2);
System.out.println( 7.3 / 2.1 );
System.out.println("Hello, world!");
System.out.println("The current value of aNumber is " +
                  aNumber);
```

The last one will convert `aNumber` to a string, append this string to "`The current value of aNumber is`", and print the result.

9.5 Conditionally Evaluating Code

Frequently, a portion of a page should be shown only under certain circumstances; for example, in the JNT navigation, the link to create new articles should be shown only to reporters. This condition has been handled by the `c:if` and `c:choose` tags. Similar constructs in Java allow code to be run only when appropriate.
A number of operators, such as `+` and `/` have already been encountered; all operate on numbers and produce another number. Another class of operators checks the truth of an expression, such as whether one value is less than another. This would be expressed in the way one might expect:

```
5 < 23
```

The type of this expression is `boolean`; it can have one of the two values `true` or `false`. In this expression, the value is `true`.
Just like any other expression, any value can be replaced by a more complex expression, possibly including variables or method calls:

```
max(aNumber,17) < anotherNumber
```

`boolean` expressions can also be combined with the *and* and *or* operators, expressed as `&&` and `||`, respectively:

```
(aNumber > 12) && (aNumber < 88)
```

This will be `true` if `aNumber` is greater than `12` and `aNumber` is also less than `88`.
It is possible to create `boolean` variables and assign the result of expressions to them, and so on. However, `boolean` expressions are most often used in *conditionals*, Java constructs that can do different things based on an expression. Conditionals consist of the word `if` and a `boolean` expression, followed by a statement, then possibly followed by the word `else` and another statement. If the `boolean` expression is `true`, the statement after the `if` will be executed; if not, the statement following the `else` will be executed:

```
if(aNumber < 0)
    System.out.println("aNumber is negative");
else
    System.out.println("aNumber is not negative");
```

By convention, the statements are indented to make the code more readable. Typically, braces are also placed around the statements, which aids readability. More important, any statements that are between an opening and closing brace are treated as a single statement. If a programmer wanted not only to detect whether `aNumber` were negative but also to change it to positive if it were, the code might look like the following:

```
if(aNumber < 0) {
    System.out.println("aNumber is negative");
    aNumber = -1 * aNumber;
} else {
    System.out.println("aNumber is not negative");
}
```

9.6 Evaluating the Same Code Multiple Times

Conditional statements are *control structures*, in that they can control the flow through a program. Another kind of control structures are *loops*, which can perform the same action multiple times. In other words, Java provides constructs that work like the `c:forEach` tag.

The simplest of these constructs is called a `while` loop, which performs an action as long as a Boolean expression continues to be `true`. The following contains all the code needed for a program that counts from 1 to 10:

```
int count = 1;

while(count < 11) {
  System.out.println("Count is now " + count);
  count = count + 1;
}
```

The first line creates a variable and sets it to 1. Java is a very expressive language, so both variable creation and assignment can be done in one step. Then the `while` loop will execute the statements within the braces until `count` reaches 11. Within the loop, the value is printed and then incremented by 1.

This loop displays a common pattern: A variable is created and initialized, a loop does an action until the variable reaches a certain value, and the value is changed within the loop. Because this combination of steps happens so frequently, another kind of loop, a `for` loop, makes it more convenient by doing something for a certain number of times. The previous code could be rewritten as a `for` loop, which would look like the following:

```
for(int count=0;count<11;count++) {
  System.out.println("Count is now " + count);
}
```

This code does exactly the same thing as the previous example but takes advantage of a number of shortcuts. First, all the code affecting the variable `i` has been moved into the `for` statement. This includes the creation and initialization, the Boolean expression that checks whether `i` has reached 11, and the increment. Another common shortcut is used in the increment; `i++` means exactly the same thing as `i = i + 1`; that is, `i` is set to one more than its current value. The body of the loop is now simply the `println` statement.

9.7 Grouping Code

When writing JSPs, it is often convenient to remove a portion of the page and put it in a separate file, which is then pulled back into the page with a `jsp:include` tag. Doing this has two good reasons. First, the same file may be included by multiple different pages. Second, it may make the site easier to understand and maintain if pages are split into logical units.

The same principles apply to Java code. Often, the same code will be needed in multiple different places, or it may make sense to identify one small portion of the overall task and move the code that performs that portion off to the side. This is accomplished by defining the code as a *method*, such as the `max` method discussed when looking at expressions.

A method definition looks something like a variable definition. The major difference is that a method must declare the types of its arguments, as well as the type of the value it will generate. A method must also define the set of instructions it will perform when it is used, in order to transform the inputs into the output.

A simple method that takes two integers and adds them together could be written as

```
int add(int a, int b) {
  return a + b;
}
```

The declaration states that this method will be called `add`, that it takes two integers, and that it returns another integer. The `return` line states that the two numbers should be added and that the value should then be given back to the code that called the method.

This method can be used in an expression, such as

```
add(12,8)
```

The variables in the method `a` and `b` will be given the values 12 and 8, respectively. The method will then evaluate, add the two numbers, and return 20.

Methods can contain loops, variable declarations, and anything else. The following example shows a method that adds all the numbers between two other numbers and returns the sum:

```
int total(int start, int end) {
   int total = 0;

   for(int i=start;i<=end;i++) {
     total = total + i;
   }

   return total;
}
```

The variable called `total` is created and initialized to zero. Then a `for` loop goes through all the values between `start` and `end`. Note that the test here is `<=`, meaning less than or equal to, which ensures that `i` will reach `end`. Each time through the loop, the value of `i` is added to `total`. At the end, the value of `total` is returned.

Methods that do not return a value, such as `System.out.println()`, have a special return type: `void`. What makes `void` special is that it has no values. An integer can be any whole number, and a Boolean can be `true` or `false`, but nothing can be of type `void`.

9.8 Handling Errors

An *exception* is generated when a Java statement or expression encounters a problem from which it cannot automatically recover. An example might be an integer expression that tried to divide by zero. The result of such a computation cannot be stored in Java, so an exception will be generated; if nothing is done, the program will terminate and report the error.

Sometimes, this behavior is perfectly OK, but more often a program will want to be able to detect exceptions and correct the problem. This can be done with a control structure called `try` and `catch`. A typical use might look like this:

```
try {
   result = value / otherValue;
   System.out.println("The result is " + result);
}catch (ArithmeticException e) {
   System.out.println("The computation could not be done");
}
```

Here, the computation is in the `try` clause. If `otherValue` happens to be zero, an `ArithmeticException` will be generated. This will cause the program to jump to the code in the `catch` clause, after which the program will proceed to whatever follows. If an exception is generated, the variable `e` that appears in this example will hold more information about the exception, including where exactly it happened. The use of this variable is beyond the scope of this chapter but can be very handy for figuring out why a program is not working the way a programmer might expect.

9.9 Modeling a Problem with Objects

Java is often described as an *object-oriented* language. To see what this term means, consider a program to play a CD, perhaps "Elyria" by Faith and the Muse. Traditional programming language would divide this problem into a set of steps that the computer would have to follow:

1. Search through the shelf for "Elyria."
2. Remove the CD in its case from the shelf.
3. Open the jewel case.
4. Remove the CD.
5. Place the CD in the CD player.
6. Push Play.

This is a perfectly valid way of programming, but an alternative treats the "things" in this example as more fundamental than the actions. The "things" here are the shelf, the CD case, the CD itself, and the CD player. Each contains a number of methods. To put it another way, each object "knows" how to do certain things. The CD player can play a CD, the CD case can provide the CD, and the shelf can provide a CD in a case. Using this approach, the program might be rewritten as:

1. `CDInCase = shelf.findCD("Elyria");`
2. `CD = CDInCase.getCD();`
3. `CDplayer.insert(CD);`
4. `CDplayer.play()`

In this version, an expression, such as `shelf.findCD("Elyria")`, means "ask the shelf object to find the CD," and `CDplayer.play()` means "ask the CD player to play whatever disc it currently contains."
Often, thinking in terms of objects can make a program much easier to write and maintain. One advantage of this style of programming is that it often hides many cumbersome details. This example has no code that determines whether the CD case is a traditional plastic jewel box or a cardboard case with the CD in a sleeve. The object itself has the code to support the `getCD()` method, so the programmer who simply needs to use a CD case never needs to think about it. Of course, the programmer who created the CD case object does need to know how this method should work, but once that code is written, it becomes much easier for others to use.

9.10 Objects in Java

Like everything else in Java, each object has an associated type, called its *class*. Classes are created by programmers and contain methods and variables. A very simple Java class might look like this:

```
public class SimpleClass {
    private int value;

    public void setValue(int v) {
      value = v;
    }

    public int getValue() {
      return value;
    }
}
```

In this chapter, this is the first example that is completely valid Java. This code could be saved to a file, compiled, and then used in a JSP page.

This class contains two methods and one variable, although in class terms, variables are more often called *fields*. The methods and fields look much like the examples used previously, with one addition. Everything in a class has associated with it a level of protection, which determines what other code is allowed to use it. Something declared *private*, for example, may be used only in the class in which it is defined. In this example, the field `value` can be accessed only from within the methods that are also defined in this class. A method or field declared *public* may be accessed by anyone. Declaring the field private and the methods public, as the example does, ensures that other classes that use this class must go through the methods if they wish to access the field. This *information hiding* is a very useful technique. In the future, if the `value` were stored in a database instead of in a simple variable, the implementations of `getValue()` and `setValue()` would need to change, but any programs that used this class would not. This is how the CD case class would hide the details of what kind of case it is. As long as the method names and types stay the same, programmers can change the way they work without causing all the code that uses the class to break.

Once a class has been defined, it can be used like any other type. Two variables of type `SimpleClass` could be created by the statement

```
SimpleClass a,b;
```

The variables `a` and `b` are in an interesting state after this statement is executed. Clearly, they both have a type, but they do not as yet have any value.

Before they can be used, they must be given values, which can be done with the `new` operator. Classes may be thought of in some sense as blueprints; `new` takes that blueprint and constructs an *instance* of that class, which is an object of the appropriate type. This is as simple as the following:

```
a = new SimpleClass();
b = new SimpleClass();
```

Now `a` and `b` will have values and may be used in any number of ways:

```
a.setValue(12);
b.setValue(14);
System.out.println(a.getValue() * b.getValue());

if(a.getValue() == b.getValue()) {
   System.out.println("The values are identical");
} else {
   System.out.println("The values are different");
}
```
The conditional at the end illustrates an important point: Each instance has its own copy of each of the fields defined in a class. If a blueprint contains the plans for a penthouse apartment and if several buildings are constructed from the blueprint, each will have its own penthouse.

9.11 Building Objects from Classes

The fact that lines that build new objects look like method calls, complete with parentheses, is not a coincidence. Classes may contain *constructors*, methods that perform special actions when an object is built. If a programmer does not explicitly provide a constructor, a default one that takes no arguments is provided automatically. Writing constructors is as easy as writing any other method. A constructor for the `SimpleClass` class might initialize `value` as
```
public SimpleClass() {
    value = 12;
}
```
This constructor is declared to be public, meaning that any other class or code can construct new instances. Now when new instances are constructed with `new`, `value` will start at 12.

Constructors can also take arguments:
```
public SimpleClass(int startValue) {
    value = startValue;
}
```
This version of the constructor takes an integer and initializes `value` to that number. This constructor could be called with a line of code like
```
SimpleClass a = new SimpleClass(20);
```
Constructors can do almost anything a regular method can, such as contain loops and conditionals, construct other objects, and so on. The only thing constructors cannot do is return a value, as in a sense, the object being constructed is the return value.

9.12 Sometimes Nothing Is Something

Every object variable, regardless of its class, is allowed to have a special value called `null`. In Java, `null` is a special valve that indicates "nothing at all." Note that this is different from the `empty string`, which is a string with no characters, often represented as `""`. This distinction is important. In the real world, this is somewhat analogous to the difference between an empty box and not having a box at all. `Null` is a convenient value to use for an object that has not yet been initialized, as a special "flag" indicating a lack of data, or in many other circumstances. It shows up often in almost all real Java programs. Also note that `null` is not of type `void`.

9.13 Building Classes from Other Classes

Consider a class that represents an animal. Such a class might have fields representing the animal's gender, life expectancy, weight, preferred environment, and so on.
Now consider a class representing a fish. A fish is a particular kind of animal, so it will share many of the characteristics of the general animal class. A fish will also have several fields of its own, such as its swimming speed, preferred water temperature, and so on. Likewise, a class representing mammals would have several fields in common with the animal class but probably none with the fish class.
When building a system to deal with all these animals, it would be inefficient to have to recreate everything in the animal class for each specific kind of animal. Java and other object-oriented languages get around this problem though *inheritance*, which allows one class to inherit the fields and methods of another. Java does this by allowing one class to *extend* another. The outline of the class definitions for the animal program follows:

```
public class Animal {
    public boolean isFemale() {...}
    public double getWeight() {...}
    public void eat(Food someFood) {...}
}

public class Fish extends Animal {
    public double getSpeedInWater() {...}
    public double getWaterTemperature() {...}
}
public class Mammal extends Animal {
    public int getNumberOfLegs() {...}
}

Fish fishy = new Fish();
Mammal aMammal = new Mammal();
```

After this code has executed, `fishy` will have all the methods of the `Fish` class, as well as the methods of the `Animal` class, so an expression like `fishy.isFemale()` will be valid. Likewise, `aMammal` will have all the `Mammal` methods as well as the `Animal` methods. However, trying to call `aMammal.getSpeedInWater()` will result in an error. This idea may be extended further. It would be possible to define a `Cat` class that extends `Mammal`, and a `Housecat` class that extends `Cat`, and so on.

It is possible to cast objects to different types, just as it is possible to cast integers to doubles and vice versa. An object may always be cast to a class from which its class was extended, so the following would be legal:

```
Animal anAnimal = (Animal) fishy;
Animal anotherAnimal = (Animal) aMammal;
```

This operation is called *upcasting* because it moves the object "up" into a more general class. *Downcasting*, whereby an object is cast into a more specific class, is also possible if the thing being cast is of the more restrictive type. The following two casts are legal:

```
(Fish) anAnimal
(Mammal) anotherAnimal
```

But this would cause a `ClassCastException`:

```
(Mammal) anAnimal
```

9.14 Interfaces

In addition to an animal's taxonomy, it is often useful to know whether an animal is kept as a pet. Pets may have special fields or methods that untamed animals might not have, such as a name. This would seem to call for a `Pet` class, but this leads to a problem. The `Housecat` class should extend the `Cat` class, but it should now also extend the `Pet` class.

Some languages do allow this sort of *multiple inheritance*, but Java does not. Java does, however, provide something almost as good: an *interface*.

Interfaces are something like classes but contain no code. Instead, an interface will state only what methods a class will provide. When it provides the methods specified in that interface, a class is said to *implement* the interface. The definitions for the `Pet` interface and the `Cat` class follow:

```
public interface Pet {
    public String getName();
    public void setName(String name);
}

public class Housecat extends Cat implements Pet {
    ...
    public String getName() {...}
    public void setName(String name) {...}
}
```

The `Pet` interface specifies two methods but provides no code. The `Housecat` class implements this interface, and to do so provides code for these methods.

Because interfaces provide no code, it may not be immediately obvious what purpose they serve. The answer lies in the way Java treats types. A class that extends another class and implements an interface, such as `Housecat`, has two types: `Cat` and `Pet`. Thus, a `Housecat` may be cast into either of these two types. This provides an easy way to ensure that code makes sense. For example, a method called `buy` will purchase a pet. This method would obviously take an object of type `Pet` as an argument:

```
void buy(Pet aPet);
```

It would be legal to call this method on a `Housecat` but not on another kind of cat, such as a snow leopard, that does not implement the `Pet` interface. This ensures that only pet animals can ever be purchased.

A class can extend only one other class but may implement any number of interfaces. The makes Java's type system much more powerful than it would be without interfaces.

9.15 Creating Groups of Classes and Interfaces

In a large program, the sheer number of classes may become overwhelming, not to mention the several hundred classes that are part of the Java core libraries! To help manage these classes, Java includes the notion of *packages*, or collections of classes, interfaces, and possibly other packages. One such package, `java.util`, contains a number of generally useful classes that programmers may use to make their lives easier. A commonly used class in this package is called `Vector`, which may be used to store an arbitrary number of objects of arbitrary types. To declare a variable of this type, it is necessary simply to include the package name before the class name:

```
java.util.Vector v = new java.util.Vector();
```

This can be a little cumbersome when many classes are used from one or more packages. To simplify this, a program may *import* one, several, or all the classes from a package, after which the program will need to specify only the class name. So, the preceding example could be rewritten as

```
import java.util.Vector;

Vector v = new Vector();
```

All the import statements must appear at the top of a file containing a Java program. If it wanted to use several of the classes from the `java.util` class, a program could have a separate import line for each, or it could get them all with

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

The asterisk indicates that all classes in the `java.util` package should be imported.

9.16 Using Java in JSPs

As mentioned, a JSP turns into a servlet, and a servlet is simply a Java class. This implies that page authors should be able to embed Java directly in JSPs and that this Java code will then be automatically carried over into the resulting servlet by the page translator. This is indeed the case, and the tag that does this is `jsp:scriptlet`. Any java code between `<jsp:scriptlet>` and the closing `</jsp:scriptlet>` will be put, unchanged, into the servlet. As a convenience, the same effect can be achieved by enclosing code within `<% ... %>` tags.

It is possible to use these tags to write information directly to the page, as in

```
<% out.println("The time is now: " +
             new java.util.Date()); %>
```

This is a complete Java statement, including the closing semicolon. The variable `out`, within a JSP page, refers to a special output mechanism that writes the enclosed data to the user or to any custom tag that may be collecting data. This object is the origin of the name `c:out` in the standard tag library.

Scriptlets can also introduce control structures:

```
<% for(int i=0;i<10;i++) { %>
  Hello!<p>
<% } %>
```

This will send `Hello!<p>` to the output page ten times. When the servlet is built, the first line with the `for` will be injected, complete with the opening brace. Then the hello line will be added and then the second scriptlet with the closing brace. The result will be a complete valid Java block.

A great deal of care must be applied when using scriptlets in this fashion; it is all too easy to miss a closing brace or accidently fail to close the scriptlet in the right place, leading to difficult-to-find problems. In almost every circumstance, it is preferable to use the corresponding control tags from the standard libraries.

It is also possible to get and use beans and access session data from within scriptlets. In fact, it is possible to do anything in a scriplet that can be done from a servlet, precisely because scriplet code goes directly into the servlet constructed by the page translator.

9.17 Database Access from Java

Although it is possible to do a great deal of useful things by manually entering SQL commands, such as those from Chapters 6, into an interpreter, the power of a database increases a thousandfold if its features can be accessed from a programming language. Traditionally, this was provided by a language-specific and database-specific set of functions called an *application programming interface* (API). The API consisted of a set of functions or classes to expose the basic database functionality. For example, a `select` function that would be passed some data structures representing the fields and `where` clause might be provided, and it would return another data structure representing the returned rows.

One of the fundamental goals of Java is to make programming independent of hardware, operating system, and other external aspects of the environment. This approach was extended to databases with the introduction of the *Java Database Connectivity* (JDBC) classes. These classes protect programmers from specific details about which database is being used by allowing queries and commands to be written in standard SQL and data to be retrieved through a standard, unified API. More information about using JDBC, as well as databases in general, is available in *JDBC™ API Tutorial and Reference, Third Edition*, by Maydene Fisher, Jon Ellis, and Jonathan Bruce (Pearson Education, 2003).

The first step in using JDBC is to obtain a JDBC *driver* for the database being used. A driver is a collection of classes that acts as the intermediary between the JDBC classes and the database itself. Of the many kinds of drivers, it is usually preferable, when possible, to use one that is written completely in Java. Before it can be used, the driver must be loaded into the program. Some JSP implementations have a property file in which drivers can be specified, but it is always safe to load the driver manually by explicitly loading its class, which can be done with a call to `Class.forName()`.

Once a driver has been loaded, a connection to a database can be obtained. Databases in JDBC are specified by URLs, which tend to look something like Web page URLs. The URL for HypersonicSQL is jdbc:hsqldb, followed by the name of the database. As all the examples in this book will use a database called jspbook, the URL will be jdbc:hsqldb:jspbook.

Many drivers for different databases can be loaded and available at the same time, which allows a JSP or other Java program to use multiple databases simultaneously. This is possible because each driver will be configured to handle a particular type of database URL.

The connection can next be used to obtain a `Statement` object, the object used to issue queries. Queries come in two basic flavors: those that return results, such as the `select` statement, and those that change the state of the database, such as `create`, `insert`, and `delete`. Corresponding to this are two methods in the `Statement` class. The `executeUpdate()` method returns an integer that indicates how many rows were affected, and the `executeQuery()` method returns a `ResultSet` object, which describes a returned set of rows and columns.

The `ResultSet` class contains methods to iterate through the rows and get the data in each column. Because Java types do not exactly correspond to SQL types, the methods that get column data must specify the type of data that is expected. For example, to get the track length after doing a `select` on the track table, a program would call `getInt("length")`. It would also be possible to use the `getObject()` method, which will return the data as an object. The program could then cast this object to an integer, string, or other appropriate type. The `ResultSet` can also be used to obtain a

ResultSetMetaData object, which will contain information about the field names, types, and other information.

Listing 9.1 shows these ideas in action in a simple command line utility to query the CD database from Chapter 6. This query may be with no arguments:

```
java com.awl.jspbook.ch09.CdExample
```

In this case, it will display the names and IDs of all albums in the system. If it is called with one of these IDs as an argument, it will display all the tracks on the corresponding album.

Listing 9.1 A Java class that uses JDBC

```
package com.awl.jspbook.ch09;

import java.sql.*;

public class CdExample {
   public static void main(String argv[]) throws Exception{
      Class.forName("org.hsqldb.jdbcDriver");

      Connection db =
         DriverManager.getConnection(
                     "jdbc:hsqldb:jspbook",
                     "sa","");

      Statement st = db.createStatement();

      try {
         Integer.parseInt(argv[0]);
      } catch (Exception e) {
         System.err.println("Invalid request");
         System.exit(-1);
      }
     // if the user has asked for detail on an album,
     // provide it
      ResultSet rs;
```

```
        if(argv.length > 0) {
            rs = st.executeQuery(
                     "SELECT name,length from track " +
                     "WHERE albumid = " + argv[0]);

            while (rs.next()) {
                System.out.println(rs.getString("name") +
                                   " -- " +
                                   rs.getInt("length"));

            }
        } else {
           // No album requested, give a list
           rs = st.executeQuery("SELECT name,albumid from cd");

           // go through the results
            while (rs.next()) {
                String name = rs.getString("name");
                System.out.println(rs.getInt("albumid") +
                                   ": " +
                                   rs.getString("name"));
            }
        }

        rs.close();
        st.close();
        db.close();
    }
}
```

This example closely follows the pattern described in the preceding paragraphs. First, the driver class is loaded. The driver is then used to get a `Connection`, and the `Connection` is used to create a `Statement`.

The class then checks whether details on a specific album have been requested and, if so, whether the request is a valid integer. This is an important security consideration; without it, nothing would stop a malicious user from entering a command such as

```
java com.awl.jspbook.ch09.CdExample "1; delete * from tracks;"
```

In this case, the query sent to the database would consist of

```
SELECT name,length from track
WHERE albumid = 1; delete * from tracks;
```

Some databases will stop processing after the first complete statement, but others will proceed to the second, wiping out the database. In general, it is a bad idea to pass input from users directly to the database. Such input should always be verified first.

Assuming that the parameter looks valid, a query is constructed to retrieve the track listing. This is submitted to the `Statement`, and a `ResultSet` is returned. A `while` loop is then used to go though each row and get the data from each column.

If the user did not request information on an album, a different query is used to get a list of all currently available albums. Again, the results are returned as a `ResultSet`, and a `while` loop goes through them.

All the preceding could also be done within a JSP. As noted earlier, however, it is much easier to use the SQL tag library and even easier to use a bean.

9.18 Summary and Conclusions

This chapter is a long way from being a complete description of the Java language but should be enough to follow the use of Java code throughout this book. Although JSPs greatly reduce the need to program in Java, anyone with an interest can learn to program, and most will find it a very useful skill when writing JSPs as well as for many other things. Readers are encouraged to pick up one of the books mentioned in the chapter or to look up a local class on programming in Java.

Chapter 10. Writing Beans

Chapter 9 provided a bridge between the world of JSPs and the world of Java. We can now cross that bridge to explore those most important of Java classes: beans! This chapter describes beans from a programmer's perspective and shows how to create beans and make their properties available to JSPs.

10.1 How Beans Are Implemented

Internally, a bean is an instance of a Java class, although in common terminology, the class itself may also be referred to as a bean. The most basic kind of bean exposes a number of properties by following a few simple rules regarding method names. In general, a bean provides two methods for each property: a method to *get* the property and one to *set* the property, corresponding directly to the `jsp:getProperty` and `jsp:setProperty` tags. Together, these methods are known as *accessors*. Listing 10.1 shows a very simple bean with two properties.

Listing 10.1 A simple bean

```
package com.awl.jspbook.ch10;

public class SimpleBean {
  private int age;
  private String name;

  public int getAge() {return age;}
  public void setAge(int age) {this.age = age;}

  public String getName() {return name;}
  public void setName(String name) {this.name = name;}
}
```

This bean could be used in a JSP just like any other that has appeared throughout this book:

```
<jsp:useBean
```

```
  id="myBean"
  class="com.awl.jspbook.ch10.SimpleBean"/>

<c:out value="${myBean.name}"/>
```

In general, for a property named `foo` of type `type`, the `get` method will return an element of `type` and will be called `getFoo()`. The one exception is that if the type is boolean, the `get` method may be called `isFoo()`. For example, if it needs to keep track of whether it is ready to perform an action, a bean might have a `ready` property, and the method could be called `isReady()`. Whether `getFoo()` or `isFoo()` is used, it is this method that is called by the `jsp:getProperty` tag, as well as any tag, such as `c:out` or `c:if`, that obtains a value from a bean. Similarly, the `set` method will accept an argument of `type`, will be called `setFoo()`, and will be called by `jsp:setProperty` and `c:set` tags.

There is no restriction on the type; it may be something simple, such as an integer or `String`, or it may be a class or interface type. The type may also be an array of another type, in which case the property is called an *indexed property*. In this case, the accessor methods operate on the whole array, and the bean may wish to provide methods to operate on the individual elements, as in Listing 10.2.

Listing 10.2 A bean with an array property

```
package com.awl.jspbook.ch10;

public class ArrayBean {
    private String things[];

    public String[] getThings() {return things;}
    public void setThings(String things[]) {

        this.things = things;
    }

    public String getThings(int i) {return things[i];}
    public void setThings(int i, String thing) {
        things[i] = thing;
    }
}
```

If an attempt is made to set or get an element with an index larger than the size of the array, the method will throw an `ArrayOutOfBounds` exception. This could, and probably should, be made explicit in the definition of the methods. In either case, any calling class should be prepared for this exception and should catch and recover appropriately. Technically, there is no reason why trying to get or set an element outside the array could not be trapped and handled by the bean, perhaps by creating a larger array and copying all the existing elements to it. The JavaBean specification states that the only way to change the size of an array is to use the array version of the `set` method, passing in a larger array. Programmers can weigh the value of adhering strictly to the standards against the need to catch exceptions elsewhere in their programs.

Although it is customary to provide both a `set` and a `get` method, doing so is not required. A property that cannot be set is called a *read-only* property, and one with no `get` method is called *write-only*. Read-only properties are fairly common; write-only ones are less so.

Nothing that has been said so far places any restrictions on what these accessor methods do. The preceding examples simply held their properties in private variables, but as long as the naming conventions are maintained, any other Java class, including a servlet or JSP, will be able to discover and access the properties. To illustrate this point, Listing 10.3 shows the `DateBean` that was used in Listing 3.2.

Listing 10.3 More complex accessor methods

```
package com.awl.jspbook.ch03;

import java.text.*;
import java.util.*;
import java.io.Serializable;

public class DateBean implements Serializable {

    public DateBean() {}

    SimpleDateFormat sdf;

    public void setFormat(String format) {
        sdf = new SimpleDateFormat(format);
    }

    public String getCurrentTime() {
        return sdf.format(new Date());
    }
}
```

This example has both a read-only and a write-only property, although there is no fundamental reason why a `getFormat()` method could not be provided. On the other hand, a `setDate()` method would presumably need to alter time, which will not be possible until Sun comes out with a "Java 2 time traveler's edition."

This example also contains an explicit constructor, even though that constructor doesn't do anything. A bean is allowed to provide as many different constructors as the programmer wants, but it must have one constructor that takes no arguments.

10.2 Automatic Type Conversion

Bean properties can be any type; yet for the most part, JSPs deal with strings. This is certainly true of form parameters, which are the entities that are most often passed to beans' `set` methods. If a bean's `set` method is expecting an integer and is passed a `String`, a runtime exception will occur. In most common cases, this potential problem is transparently resolved by the `jsp:setProperty` and `c:set` tags, which will try to convert the string to an appropriate type. If the method is expecting an integer, the JSP system will call `Integer.parseInt()` to obtain an integer value. If this conversion fails, perhaps because the user has entered a string that cannot be turned into an integer, the `set` method will simply not be called. This can be a problem if some later code expects that all the parameters have been set successfully.

There are a few ways to handle this. The first and most obvious is for all `set` methods to accept strings and do the conversion themselves. However, a more elegant approach would be to use a controller to mediate between the form and the bean. This will be done in Chapter 12.

10.3 How Beans Work

All the JSP/bean functionality is built on the ability of one Java class to discover and invoke methods on another class at runtime. The mechanism that supports this, *introspection*, has been built into Java since version 1.1. This extremely powerful capability is missing from many other object-oriented languages, in which everything must be known in advance; once a program is built, it may have to be changed significantly to extend it with new functionality.

Introspection is possible because a lot of information about method names and signatures is stored in .class files, and certain methods can access and organize this information. An easy way to see the kinds of information that introspection provides is to use the `javap` utility, which is included in the JDK (Java Development Kit). `Javap` is run from the command line and is invoked with the name of a class. If it is given `SimpleBean` from Listing 10.1, `Javap` will generate the following output:

```
Compiled from SimpleBean.java
public synchronized class SimpleBean extends java.lang.Object
    /* ACC_SUPER bit set */
{
    public int getAge();
    public void setAge(int);
    public java.lang.String getName();
    public void setName(java.lang.String);
    public SimpleBean();
}
```

Although `javap` does not use introspection to generate this output, the principle is the same. The utility is able to pull out the names of all the methods and the type of their arguments and to return values. From this, a person or a program could infer that there is a property called `name` that is a string, and so on.

Introspection also provides a mechanism to create a new instance of an object once its class has been loaded. This mechanism will construct this instance by looking for a constructor that takes no arguments, which is why the programmer must provide one. As a convenience, if a class contains no constructors at all, Java will automatically provide one that takes no arguments and doesn't do anything. However, it is always better to make such things explicit.

The classes related to introspection are all in the `java.beans` and `java.lang. reflect` packages, and the whole process starts with the `java.beans. Introspector` class. The use of these classes is beyond the scope of this book, but readers are encouraged to peruse the JDK documentation to see how all this is accomplished.

10.4 Bean Serialization

As discussed in Chapter 3, one of the remarkable features of beans is their ability to store an instance of a bean, perhaps containing some local configuration data, in a file. As mentioned, this requires no special code in the bean; the class must simply implement the `java.io.Serializable` interface. Listing 10.4 shows a bean with a `main` method that allows instances to be created, saved, and loaded.

Listing 10.4 A bean that uses serialization

```
package com.awl.jspbook.ch10;

import java.io.*;
import java.util.*;
import java.text.*;

public class SaveableBean implements Serializable {
  private Date createTime;
  private String message;

  public SaveableBean() {
    setDate(new Date());
  }

  public void setDate(Date createTime) {
    this.createTime = createTime;
  }

  public Date getDate() {return createTime;}

  public void setMessage(String message) {
    this.message = message;
  }
  public String getMessage() {return message;}

  public static void main(String argv[])
    throws Exception
  {
    if (argv[0].equals("-create")) {
      SaveableBean sb = new SaveableBean();
      sb.setMessage(argv[2]);

      ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream(argv[1]));
      out.writeObject(sb);
      out.close();
      System.out.println("Bean created and saved!");
    } else if (argv[0].equals("-load")) {
      ObjectInputStream in = new ObjectInputStream(
   new FileInputStream(argv[1]));
      SaveableBean sb = (SaveableBean) in.readObject();
      in.close();

      SimpleDateFormat sdf =
new SimpleDateFormat("hh:mm:ss dd/MM/yy");

      System.out.println("This bean was created at: " +
 sdf.format(sb.getDate()));

      System.out.println("This bean says: " +
 sb.getMessage());
    }
  }
}
```

The code that does the saving and loading does not need to be in the `main()` method of this class, as any class can read or write saved instances of any other class. This is what makes general bean editors, as well as JSPs, possible.

The `Serializable` interface does not have any methods; it is enough for a class to declare itself serializable. However, such classes must adhere to one restriction: All the members of a serializable class must themselves be serializable, as must their members, and so on. Most of the classes from the core Java libraries that a bean would contain are serializable, so this is not a concern very often.

When a bean does need to have a nonserializable member, this member can be declared `transient`, and the serialization methods will simply ignore this member when saving or loading. This may be desirable even when it is not necessary. For example, if a bean will be used to show the current date, saving its `Date` object in a file may not make sense.

The obvious downside of transient members is that they will be in some uninitialized state, probably `null`, after the bean loads. The serialization mechanism allows a bean to "know" when it is being unserialized, so that it can take the opportunity to put its transient members into some consistent state. To do this, it is necessary only for the bean to have a `readObject()` method, which might look as follows:

```
private void readObject(java.io.ObjectInputStream stream)
    throws java.io.IOException, ClassNotFoundException
{
    stream.defaultReadObject();
    ... initialize transient members here ...
}
```

Likewise, a bean can know when it is being serialized, in case it needs to do some special processing before it is written. This is done by providing a `writeObject()` method. See the page for `java.io.Serializable` in the JDK documentation for more details.

10.5 Events

So far, beans have been fairly self-contained. When a property is obtained or changed or when an instance is saved or loaded, the only objects that know about it are the object that performed the action and the bean itself. Often, it is desirable for beans to communicate with one another. For example, a JSP might have a bean that is used as a shopping cart and another bean that handles inventory. When a product is placed in the cart, the inventory bean should be told that one less item of this product is available for other shoppers. The JSP could handle this manually, by calling the appropriate methods on both beans. Besides being inconvenient, this would risk potential problems with programmers forgetting to call the right methods in the right order. Instead, beans support numerous mechanisms to communicate directly with one another.

Beans were originally designed as graphic components, such as buttons or menus. In this role, a bean would be driven by *events*, such as a user clicking a button. Other beans would need to *listen* for a set of events and react appropriately. This leads to an event-based communication mechanism being incorporated into the bean specification, and this mechanism turns out to be useful for server-side programs as well. The shopping cart might generate, or *fire*, an event when an item is put into it, and the inventory bean might listen for this event and react by decrementing its supply.

Event programming is almost as easy as property programming and once again is expressed mostly as a set of naming conventions. First, it is necessary to define a class to represent the event. Listing 10.5 shows an event that represents putting an item in a shopping cart.

Listing 10.5 An event

```
package com.awl.jspbook.ch10;

public class PurchaseEvent extends java.util.EventObject {
  private String itemName;

  public PurchaseEvent(Object source,String itemName)
  {
    super(source);

    this.itemName = itemName;
  }

  public String getItemName() {return itemName;}
}
```

Once the event has been defined, it is necessary to define an interface that will listen for events of that type, such as the one in Listing 10.6. Listeners are defined as interfaces, which allows any class to declare that it will listen for any set of events.

Listing 10.6 A listener interface

```
package com.awl.jspbook.ch10;

public interface PurchaseListener
```

```
  extends java.util.EventListener
{
  public void purchaseMade(PurchaseEvent e);
}
```

This interface has only one method, but it is allowable for a listener interface to have an arbitrary number. In a real e-commerce system, the listener might need a second method to handle a user removing an item from his or her shopping cart. This could go in the same `PurchaseListener` interface or in a separate one. In the latter case, the inventory bean would need to implement both interfaces.

Once the event and listener have been defined, one or more beans can set themselves up as event sources. This is done by providing two methods, one of which will add a listener; the other will remove a listener. The method names will include the type of event, which will allow introspection to figure out automatically what kinds of events a bean may generate. Listing 10.7 shows a shopping cart bean that generates purchase events.

Listing 10.7 A bean representing a shopping cart

```
package com.awl.jspbook.ch10;

import java.io.*;
import java.util.*;

public class ShoppingCartBean implements Serializable {
  public String items[];
  public int numItems;

  public Vector purchaseListeners;

  public ShoppingCartBean() {
    purchaseListeners = new Vector();
    items             = new String[50];
    numItems          = 0;
  }

  public void setItem(String item) {
    items[numItems++] = item;
    firePurchaseEvent(item);
  }

  private void firePurchaseEvent(String item) {
    PurchaseEvent pe = new PurchaseEvent(this,item);
    Enumeration e = purchaseListeners.elements();
    while(e.hasMoreElements()) {
     ((PurchaseListener) e.nextElement()).purchaseMade(pe);
    }
  }

  public void addPurchaseListener(PurchaseListener p) {
    purchaseListeners.addElement(p);
  }

  public void removePurchaseListener(PurchaseListener p) {
    purchaseListeners.removeElement(p);
  }
}
```

The names `addPurchaseListener` and `removePurchaseListener` are enough for the system to figure out that there must be a `PurchaseEvent` class and a `PurchaseListener` interface, and this information could then be used in a graphic bean builder to hook two or more beans together.

Finally, Listing 10.8 shows the inventory bean, which can handle purchase events.

Listing 10.8 A bean that represents an inventory and handles purchase events

```
package com.awl.jspbook.ch10;

import java.io.*;
import java.util.*;

public class InventoryBean
  implements Serializable,PurchaseListener
{
  private static final Integer ONE = new Integer(1);
  private Hashtable inventory = new Hashtable();

  public void addInventory(String name) {
    Integer count = (Integer) inventory.get(name);
    if(count == null) {
      inventory.put(name,ONE);
    } else {

      inventory.put(name,
    new Integer(count.intValue() + 1));
    }
  }

  public void removeInventory(String name)
  {
    Integer count = (Integer) inventory.get(name);
    if(count == null) {
      return;
    } else if(count.equals(ONE)) {
      inventory.remove(name);
    } else {
      inventory.put(name,
    new Integer(count.intValue() - 1));
    }
  }

  public void purchaseMade(PurchaseEvent pe) {
    removeInventory(pe.getItemName());
  }
}
```

As expected, this class implements the `PurchaseListener` interface and does the obvious thing when it receives a purchase event.

Even more would need to be done in order to hook the shopping cart to the inventory. In particular, the shopping cart will presumably live in one or more session scopes, and the inventory will reside in the application scope. These beans must be able to "discover" each other. Numerous other details remain to be filled in, such as how the inventory should respond if it receives a request for an item that is out of stock.

10.6 Special Events

In the bean sense, special events refer to a couple of event types that are of particular interest to bean authors working with JSPs. The first is called `PropertyChange Event`. A bean may fire one of these any time one of its properties changes, in order to alert other beans to the change. A property that generates a `PropertyChangeEvent` when it is modified is called a *bound* property.

A bean can also refuse to set a property to a new value, by generating a `VetoEvent`. This is typically thrown when another object tries to set a property to an unacceptable value. The inventory bean might throw this exception if someone tried to change the number of items it is holding to ?. A property that can generate a `VetoEvent` is known as a *constrained* property.

These two events are defined in the bean specification. The JSP specification defines an additional event, `HttpSessionBindingEvent`, which can be used to notify a bean that it has been added to or removed from a session scope. Recall that a session will end after a user has not come back to the site after a certain length of time. When this happens, the session will be deleted to make room in memory for other sessions, and any data in the session will be lost. However, before the session is killed, all data objects connected to it will be sent an `HttpSessionBindingEvent`, which gives these data objects a chance to save data to a database or file or to do any other cleanup. Listing 10.9 shows a bean that saves itself when the session shuts down.

Listing 10.9 A bean that listens for session binding events

```
package com.awl.jspbook.ch10;

import java.io.*;
import java.util.*;
import javax.servlet.http.*;

public class SessionBean
  implements Serializable, HttpSessionBindingListener
```

```
{
  private String fileName;
  private String message;

  public void setFileName(String fileName) {
    this.fileName = fileName;
  }

  public String getFileName() {return fileName;}

  public void setMessage(String message) {
    this.message = message;
  }

  public String getMessage() {return message;}

  public void valueBound(HttpSessionBindingEvent b) {
  }

  public void valueUnbound(HttpSessionBindingEvent b) {
    save();
  }

  public void save() {
    try {
      ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream(fileName));
      out.writeObject(this);
      out.close();
    } catch (Exception e) {}
  }

  public static void main(String argv[]) {
    SessionBean sb = new SessionBean();
    sb.setFileName(argv[0]);
    sb.setMessage(argv[1]);
    sb.save();
  }
}
```

An instance of this bean may be created and saved in a file with its `main()` method. A JSP may then use this serialized bean and change its message property. Some time later, when the session has expired, the bean will save itself, including the current message string, back into the file. Note that there is no guarantee when this will happen. The JSP engine may expire the session after a fixed timeout period, when it needs more memory, or perhaps not until the Web server is shut down.

10.7 Bean Errors

The most common problem when using a bean within a JSP is that introspection tends to mask programmer exceptions, making it difficult to see where the problem really is. If a `set` method throws an exception, the JSP engine will likely print out something cryptic:

```
java.lang.reflect.InvocationTargetException
  at com.sun.jsp.runtime.JspRuntimeLibrary.introspecthelper
  at com.sun.jsp.runtime.JspRuntimeLibrary.introspect
  ... etc ...
```

The easiest way to discover the real problem is to put the bodies of all the `set` and `get` methods in `try/catch` blocks and have the `catch` clause dump the exception to `System.err`.

Although this will cause useful debugging information to be generated, it will leave the bean in an inconsistent state. There is no hard-and-fast rule about what to do in such a situation. The `set` method could leave the property in its last known state, or it could be reset to a sensible default. Another possibility is to throw the original exception. The user will get an error page, but this might be preferable to getting weird results. Perhaps in a future version of the JSP specification, the JSP engine will listen for `VetoEvents`, in which case a method could fire such an event on receiving an exception.

Another potential problem concerns serialized instances. Consider what would happen if a class contained a member of type `int`, a serialized instance of this class were created, and then the programmer rewrote the class to make the member a `String`. Even if the deserialization process were able to build something from this, the result would likely not be meaningful.

To prevent this problem, all classes and serialized instances have an ID called the `serialVersionUID`. When an object is deserialized, the ID of the instance is checked against that of the class; if they do not match, an exception will be thrown. The output from the JSP engine in that case would look something like this:

```
java.io.InvalidClassException: SaveableBean;
Local class not compatible:
stream classdesc serialVersionUID=8221280906864288240
local class serialVersionUID=-8806858158408665433
```

If a field has changed types, not much can be done about an error like this, and the only option is to recreate all the serialized instances with the new class. However, some changes are more benign. For example, adding a new field or method should not affect the ability to load old data, as long as it is OK to leave the new fields in an uninitialized state after loading.

In most classes, the serial version (UID) value is not implicit but rather is computed based on properties of the class. When the class structure changes, so will this value. However, if old serialized instances should still work with a new class, an explicit form of the ID can be provided to make sure the IDs match. In the preceding case, it would be necessary simply to tell the class to use the same ID as the stream found, which could be done by adding the following line to the class:

```
private static final long
    serialVersionUID=8221280906864288240L;
```

If an ID has changed because new members were added to the class, the new version of the bean could be given a `readObject()` method to initialize the new fields after loading.

10.8 Summary and Conclusions

Beans are nothing more than Java classes that adhere to certain naming conventions. Beans make properties available by providing `get` and `set` methods, which obtain and modify the property, respectively. Beans may also be serializable, meaning that they can write their data out to disk and restore it later. Finally, beans may generate or listen to events, and such events can be used to tie beans together. Of particular interest is the `HttpSessionBindingListener` interface, which a bean can use to get notified when a bean in a session scope is about to be retired.

Anyone who can write a Java class can write beans that can be used in JSPs. Correspondingly, almost any Java class can be turned into a bean by thinking about what the class does in terms of properties and exporting those properties with appropriately named methods.

Chapter 11. Servlets

This book has made many references to *servlets*. In particular, it has been repeatedly noted that a JSP file is turned into a servlet at translation time and that the resulting servlet is run at request time. This means that JSPs and servlets are the same thing.

As discussed in Chapter 1, a servlet is a small class that may be thought of as a dynamic extension to a Web server or application server. CGIs, by contrast, are external programs started by the Web server. This change from external to internal extensions has a number of advantages, chief of which is performance. Because a servlet is loaded only once, the first time it is needed and subsequently, it resides in the same Java virtual machine (JVM) as the Web server or application server; thus, the large overhead of starting a new program for each request is avoided.

This chapter is not meant to be a comprehensive study of servlets, a topic that could fill a book itself. See, for example, *Enterprise Java™ Servlets* by Jeff M. Genender (Pearson Education, 2002), which offers a much more detailed look at servlets and the servlet API. As JSPs ultimately are servlets, it makes sense for JSP authors to know at least a little about what is going on behind the scenes, if for no other reason than to appreciate how much easier it is to write JSPs!

11.1 The Servlet Life Cycle

A CGI has a pretty simple life. When a user makes a request, the Web server runs the CGI program. For a Perl program, this means starting up the Perl interpreter, which then reads the file comprising the program and starts executing instructions, beginning at the top of the file and moving down. For a CGI written in C or C++, the program starts at its `main()` method, which then may call other procedures, create classes, and so on. In either case, the Web server communicates all the information about the request through a set of variables and data sent to the program's input. The program generates a response by printing some headers containing the result's attributes, such as type and length, followed by the result itself. The CGI program then exits, vanishing from system memory as if it had never existed. If the same CGI has many requests, either all at once or one after the other, a new instance of the CGI will be created each time.

In principle, servlets could follow the same pattern. A servlet could consist of nothing more than a class with a `service()` method to handle a request. Each time a request came in, a new instance of the servlet would be created, and its `service()` method would be called. Inside this method, the class could make calls to get request information, and it could return results by printing the headers and the resulting HTML page, just as a CGI does.

This process would save the overhead of loading the class each time but is still very inefficient. The servlet API can best be understood by starting from this model and seeing what improvements could be made.

First, it is unnecessary to create a new instance for every request. The Web server needs to create only a single instance and can then call this instance's `service()` method for each request. For this to be possible, the `service()` method could not use any global data or write to a common output stream. If global data were used to hold the request, input would be jumbled if two or more requests came in at the same time. Likewise, if a single output stream were used, the output of multiple simultaneous requests would be intermingled.

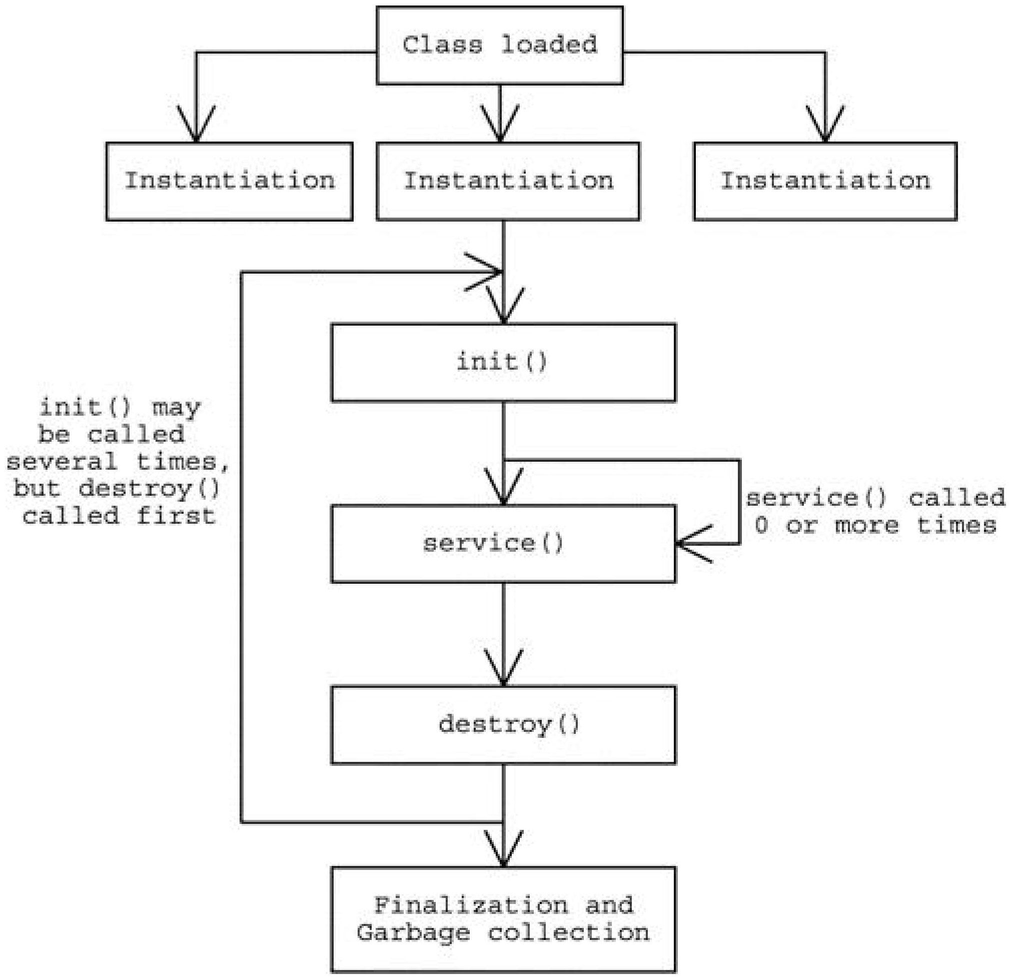
The solution to this problem is to have the Web server pass in unique instances of objects representing the request and response each time it calls the `service()` method. This might seem even worse than constructing a new servlet, but in fact doing it this way has advantages. For one thing, the Web server would likely need to do this work anyway, as different requests must be kept isolated from one another. Second, the request and response objects will typically be much simpler than the servlet object, so it will be easier to build them. By the way, if the notion of a request object sounds familiar, it should. This is the same object from which information about the request was obtained in Listing 4.7 and which contains data that is in the request scope.

Now that the servlet will be constructed only once, a further optimization can be made. Consider a CGI that uses a database. Each time it is started, it will need to reestablish a connection to the database, because a CGI has no way to hold onto a connection between the time it is shut down and the time it starts up again. However, as a servlet never exits, it needs to open this connection only once. The same is true for many other kinds of initializations, such as building some auxiliary classes or setting some variables to known defaults. This means that all the initialization code can be taken out of the `service()` method and put into a separate method: `init()`. The Web server will call the `init()` method once when the servlet is first loaded, and after that it may call the `service()` method multiple times.

If servlets lasted forever, those two methods would be the only ones needed. However, a servlet may be retired in a number of ways. Sooner or later, the Web server will need to shut down; when it does, it should give all its servlets a chance to clean up after themselves, close database connections, and so on. A servlet might also be replaced by a newer version, in which case the old version should also be given the opportunity to close any resources it has opened. Servlets handle this possibility by supporting a `destroy()` method, which will be called by the Web server when it knows that the servlet will not be asked to service any more requests. The servlet then has the chance to undo anything it did in the `init()` method.

These three methods define the servlet life cycle, which is illustrated in Figure 11.1.

Figure 11.1. The servlet life cycle.


Class loaded
Instantiation
Instantiation
Instantiation
init()
init() may be called several times, but destroy() called first
service()
service() called 0 or more times
destroy()
Finalization and Garbage collection

11.2 The Servlet Class Hierarchy

The most basic servlet definitions live in the `javax.servlet` package and consist of a number of interfaces.

- The `ServletContext` interface provides a means for servlets to communicate with the surrounding Web server or application server. This communication can take the form of requests for system resources, reports written by the servlet to a log file, and so on. Indirectly, the `ServletContext` also allows servlets to communicate with one another, primarily by sending requests to other pages. This is how the `jsp:forward` and `jsp:include` tags are implemented, as will be seen shortly. `ServletContext` is implemented by people writing the Web or application server; servlet authors seldom need to use it directly and never need to extend it.
- The `ServletConfig` interface provides a mechanism for the web Server to pass initialization information to the servlet's `init()` method. This information takes the form of pairs of names and values, which are stored in a configuration file called web.xml, which is examined more closely in Appendix B. If a servlet is going to open a connection to a database, it would not make sense to hard-code the name of the driver class and the database URL in the servlet's code. Doing so would make the servlet more difficult to change if a new database were ever installed. Instead, this information could be sent to the servlet as parameters, and the servlet would use the `ServletConfig` to retrieve these values and act accordingly. Like `ServletContext`, this interface is implemented by the authors of the Web server.
- The `Servlet` interface defines the three life-cycle methods?TT>init(), service(),

  and `destory()` as well as a handful of others.

- The `ServletRequest` and `ServletResponse` interfaces encapsulate a request to the servlet and a response from the servlet, respectively. Objects that implement these interfaces will be passed to the servlet's `service()` method. Code within this method can then use the `ServletRequest` to determine information about the request, such as its origin, the exact data being requested, and so on. Similarly code in the `service()` method can then use the `ServletResponse` to return information about the response, as well as the data, such as an HTML page, that comprises the response itself.

The `javax.servlet` package also defines three other classes. Two are the `ServletInputStream` and `ServletOutputStream` classes, which servlets use to read and write data, respectively. The third class, `GenericServlet`, implements both the `Servlet` and `ServletConfig` interfaces and forms the basis for most real servlets.

Note that so far, none of this has been specific to the Web or HTTP (HyperText Transfer Protocol). This is deliberate. Many kinds of servers are on the Web, including FTP (file transfer protocol), mail, chat servers, games, and so on. Many of these servers will want the same kind of dynamic extensibility that Web servers have, so it makes sense for each of these servers to have a corresponding servlet. A multiuser dungeon game might have a "character" servlet and a "weapon" servlet; as new kinds of characters and weapons are introduced, they could be written as servlets and loaded as needed.

The Web-specific versions of the servlet classes are included in a package called `javax.servlet.http`. The heart of this package is the `HttpServlet` class, which extends `GenericServlet`. This class's `service()` method takes `HttpServletRequest` and `HttpServletResponse` objects, instead of the generic versions from the `javax.servlet` package. These variations contain a great deal of HTTP-specific information, such as cookies, remote user names, authentication schemes, and so on.

The `HttpServlet` class also has provided a built-in `service()` method, which looks at what kind of request has been received and calls an appropriate method to handle it. For example, if the request were an HTTP `GET`, the `doGet()` method will be called. There is also a `doPost()` method, `doDelete()`, and so on. This frees a servlet writer from worrying about handing specific requests properly. A servlet writer will simply override the appropriate `do` method or methods; if any other kind of request comes in, the servlet will report that it does not handle that kind of request.

Listing 11.1 shows a simple servlet that shows these methods in use.

Listing 11.1 A simple servlet

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class HelloServlet extends HttpServlet {
  private String message;

  public void init(ServletConfig sc)
    throws ServletException
  {
    super.init(sc);
    message = sc.getInitParameter("message");
    if (message == null)
      message = "Hello, world!";
  }

  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void doPost(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void handle(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    res.setStatus(res.SC_OK);
    res.setContentType("text/html");

    PrintWriter out = res.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>A servlet</TITLE></HEAD>");
    out.println("<BODY>");
    out.println(message);
    out.println("</BODY>");
    out.println("</HTML>");

    out.close();
  }

  public void destroy() {
    return;
  }
}
```

After the servlet is loaded, its `init()` method will be called. This method will look for a parameter to display as the response; if no such parameter is provided, a default will be used. Note that the `init()` method starts by calling `super.init()`, which will ensure that all the behind-the-scenes setup is done properly.

This servlet does not override the `service()` method, so the default one will be used. Consequently, any `GET` or `POST` methods will be handled by the corresponding method, and any other kind of request will result in an error. In this case, the response to both `GET` and `POST` requests will be identical, so both of these methods call `handle()`, which is responsible for handing the request.

First, `handle()` sets some information about the response it is returning. In particular, it specifies that the request succeeded by setting the status code to `SC_OK`, and it specifies that HTML data will be returned. Then `handle()` obtains a `PrintWriter` object, which it uses to print the page. Even on a page as simple as this one, all those `print` statements can be a burden and a hassle, which is one reason it is so much easier to write JSPs.

The fact that `out` was used as the name of the `PrintWriter` is no mere accident! The name was chosen because this object is basically the same kind as used by the `c:out` tag to send the value of an expression to the page, although this will have to be clarified a little when discussing tags in more detail in the next chapter.

This servlet has nothing to clean up when it is decommissioned, so the `destroy()` method simply returns. In fact, this method is not needed here at all but is included for the sake of completeness.

Two exceptions are declared by the methods in this servlet. The `IOException` is a general exception thrown whenever there is a problem with the input or output of data, which could happen if the user clicks the browser stop button before receiving the full page or if a network problem occurs.

The other exception, `ServletException`, provides a means for servlet authors to indicate that the servlet has run into a problem while processing, although this servlet never actually throws one. However, if the servlet author decided that failing to provide a message was a critical error, `init()` could throw a `ServletException` instead of setting `message` to a default value. This exception could be constructed with a message describing the problem, and this message would end up in a log file.

A servlet can provide additional information by throwing a subclass of `ServletException: UnavailableException`. The `UnavailableException` comes in two flavors: permanent and temporary. The permanent version tells the Web server that this servlet cannot continue, and the Web server will consequently never call the servlet's `init()` or `service()` methods again. On the other hand, if a servlet throws the temporary version, it may include a time interval in seconds. The Web server will wait for the specified period of time and will then call the method that threw the exception. This is useful if a servlet discovers that a resource it needs, such as a database, has become unavailable. The chances are good that the database will be restarted shortly, at which point the servlet can reconnect and continue working. In the meantime, however, the servlet need not try to handle requests, which will only put a lot of unnecessary burden on the network.

11.2.1 More about Requests

In the broadest sense, the `HttpServletRequest` interface defines everything there is to know about the request. Some of this information is fairly straightforward, such as the name of the machine from which the request was issued or the browser that is being used. However, a few kinds of information warrant further discussion.

One important kind of data provided by the request object is the list of *cookies* sent by the browser. Cookies are small pieces of data that a Web site can send to a browser, which the browser is then expected to send back to the Web site on subsequent requests. Cookies have a number of properties, including their *domain*, which indicates to which systems the cookie should be sent; their *path*, which indicates for which URLs within that domain the cookie should be sent; and a *maximum age*, which indicates how long the cookie has to live. Once the time has passed, the cookie will no longer be sent back to the Web site and will probably be deleted from the browser's cookie repository.

In the servlet APIs, cookies are represented by the javax.servlet.http class, which contains all the preceding information and possibly additional fields. The list of cookies may be obtained from the request object by calling its `getCookies()` method, which will return an array of all the cookies the browser sent with the current request.

Internally, JSPs and servlets also use a special cookie to keep track of sessions. HTTP is by nature a *stateless* protocol, meaning that no information is preserved between requests. In order to implement sessions, all the information about each session is stored somewhere in memory, and a special key is used to look up the data relevant to a particular user's session. This key is passed in a cookie, so each time the user makes a request, the key will be sent along, and the JSP engine can then retrieve this cookie and use it to access the relevant session data.

Some users distrust cookies, so the servlet API provides an alternative way to pass the key back and forth. This is done by rewriting each URL to include the key. For example, a user without cookies who tries to access http://somesite.com/apage.jsp might be redirected to something like
http://somesite.com/To1010mC0673157862957708/apage.jsp, where the additional information contains the session key. Any links on the page should be relative, such as .../something/another page.jsp, so that when the user follows the link, the portion of the URL that contains the session key will be preserved.

The request object can inform a servlet or JSP whether the user's session is stored in a cookie or a URL, by using two methods called `isRequestedSessionIdFrom-Cookie()` and `isRequestedSessionIdFromURL()` methods. The request can also provide access to the session itself, which we will discuss further when we look at using scopes from servlets.

Listing 11.2 shows a servlet that prints all its cookies, as well as some information about the current session.

Listing 11.2 A servlet that gets cookie and session information

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class CookieServlet extends HttpServlet {
  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
{
    handle(req,res);
}


public void doPost(HttpServletRequest req,
   HttpServletResponse res)
  throws IOException,ServletException
{
  handle(req,res);
}


public void handle(HttpServletRequest req,
   HttpServletResponse res)
  throws IOException,ServletException
{
  res.setStatus(res.SC_OK);
  res.setContentType("text/html");

  PrintWriter out = res.getWriter();

  out.println("<HTML>");
  out.println("<HEAD><TITLE>Cookies</TITLE></HEAD>");
  out.println("<BODY>");

  Cookie cookies[] = req.getCookies();

  if(cookies == null || cookies.length == 0) {
    out.println("You have no cookies");
  } else {
    out.println("<TABLE>");

    out.println("<TR>");
    out.println("<TH>Name</TH>");
    out.println("<TH>Domain</TH>");
    out.println("<TH>Path</TH>");
      out.println("<TH>Value</TH>");
      out.println("<TH>Max Age</TH>");
      out.println("</TR>");

      for(int i=0;i<cookies.length;i++) {
out.println("<TR>");
    out.println("<TD>" + cookies[i].getName() + "</TD>");
    out.println("<TD>" + cookies[i].getDomain() + "</TD>");
    out.println("<TD>" + cookies[i].getPath() + "</TD>");
    out.println("<TD>" + cookies[i].getValue() + "</TD>");
    out.println("<TD>" + cookies[i].getMaxAge() + "</TD>");
out.println("</TR>");
      }
    }

    out.println("</TABLE>");

    if(req.isRequestedSessionIdValid()) {
      if(req.isRequestedSessionIdFromCookie()) {
out.println("This session is from a cookie");
      }

      if(req.isRequestedSessionIdFromURL()) {
out.println("This session is from the URL");
      }
    } else {
      out.println("There is no session associated ");
      out.println("with this request");
    }
    out.println("</BODY>");
    out.println("</HTML>");

    out.close();
  }
}
```

Structurally, this servlet closely resembles the one from Listing 11.1; the major difference is that here, the servlet has no `init()` or `destroy()` methods. Inside the `handle()` method, the cookies are obtained from the request object and printed. The servlet then checks for a current valid session and, if so, determines whether it came from a cookie or was written into the URL.

When this example is first run, the user will not have any cookies, so the output will be rather sparse. The output can be made more interesting by creating a servlet that will issue cookies, which will be done in the next section.

11.2.2 More about Responses

The `HttpServletResponse` interface is in some ways the mirror image of the request interface. The request interface has information about the request that the servlet is meant to read but cannot change. The response object is where the servlet can write information about the data it will send back. Whereas the majority of the methods in the request interface start with `get`, indicating that they get a value, many of the important methods in the response start with `set`, indicating that they change a property. Note how even these interfaces adhere to the usual naming conventions for beans.

Two of these `set` methods have already been encountered: one that sets the status code and one that sets the content type. The status code indicates the status of the response. Ideally, this will usually be `SC_OK`, indicating that the request succeeded normally and that the page data will follow. Other codes include `SC_INTERNAL_SERVER_ERROR`, which indicates a problem internal to the Web server. This is the code that results in "error 500" messages such as the one in Figure 2.1. A complete list of codes supported by HTTP is included in the `HttpServletResponse` interface.

The content type indicates what kind of data will be sent back to the user. The `page` directive uses this same mechanism, as was done in Chapter 8 to specify that some examples were returning data of type `text/xml`. It is now clear how this directive works; it simply generates a `setContentType()` call in the resulting servlet, just as has been

done in Listing 11.2. A servlet could also send out plain text by setting this type to `text/plain` or even generate an image by setting this to something like `image/bmp`.
In addition to these two methods, a servlet can indicate how much data is coming back by using the `setContentLength()` method. This attribute is set in relatively few pages these days, but whenever possible, its use is encouraged, as it can help the browser know when it can close the connection to the server and stop showing the throbbing "N" or spinning "e."
The request object can also add new cookies by calling the `addCookie()` method. This method can take as an argument a brand new cookie, or an existing cookie can be obtained from the request, modified, and then sent back to the user.
The other important method in the request interface is `getWriter()`, which returns a `PrintWriter`, which the servlet then uses to send its data. Normally, this is the preferred way to send data back to the user, as the `PrintWriter` class is convenient to use, as well as automatically handling some internationalization issues. If it is going to be sending back binary data, the servlet may instead wish to use the `ServletOutputStream` class, which can be obtained by calling `getOutputStream()`. This class contains a `write(byte[])` method, which is ideal for sending bytes of data that should not be interpreted or altered in any way by either the browser or the server.
It is important to note that all the header information, such as the content type and length, must reach the browser before any of the data. This makes sense, as the browser will not know what to do with this data until it receives all the header information. Under most circumstances, therefore, all the response's `set` methods must be called before anything is printed to either `ServletOutputStream` or `PrintWriter`.
To be completely accurate, the output from the servlet is *buffered*, meaning that it is held in memory until a certain amount has accumulated, and then it is all sent to the user at once. This is done for the sake of efficiency, as sending data across the network has a lot of overhead and is best done as infrequently as possible. This means that new headers can be set until the buffered content is transmitted, which can be determined with the `isCommitted()` method, which returns `true` if any data has been sent to the user. To be on the safe side, though, it is strongly recommended that all headers be sent before starting to send any data.
Listing 11.3 shows how a servlet can use the `HttpServletResponse` to add a new cookie and set a variety of other information.

Listing 11.3 A servlet that sets cookies

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;


public class ResponseServlet extends HttpServlet {
  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }

  public void doPost(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }

  public void handle(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    StringBuffer text = new StringBuffer();

    String name = req.getParameter("name");
    String value = req.getParameter("value");

    if(name == null) {
      name = "Acookie";
    }

    if(value == null) {
      value = "AcookieValue";
    }

    text.append("<HTML>");
    text.append("<HEAD><TITLE>A servlet</TITLE></HEAD>");
    text.append("<BODY>");
    text.append("<P>Added a cookie whose name is: ");
    text.append(name);
    text.append(" with a value of: ");
    text.append(value);
    text.append("</P>");
    text.append("<FORM ACTION=\"response\" METHOD=\"GET\">");
    text.append("<P>Name: ");
    text.append("<INPUT TYPE=\"TEXT\" NAME=\"name\"></P>");
    text.append("<P>Value: ");
    text.append("<INPUT TYPE=\"TEXT\" NAME=\"value\"></P>");
    text.append("<P><INPUT TYPE=\"SUBMIT\"></P>");
    text.append("</FORM>");
    text.append("</BODY>");
    text.append("</HTML>");

    String html = text.toString();

    res.setStatus(res.SC_OK);
    res.setContentType("text/html");
    res.setContentLength(html.length());

    Cookie cookie = new Cookie(name,value);
    cookie.setMaxAge((int) System.currentTimeMillis() +
       1000 * 60 * 10);
    res.addCookie(cookie);

    PrintWriter out = res.getWriter();
    out.print(html);
    out.close();
}
```

```
}
```

The cookie creation itself is straightforward; the cookie is simply constructed and sent to the user with the `addCookie()` method. The cookie is set to last for 10 minutes by setting its maximum age to the current time plus 10 minutes, expressed in milliseconds.

This servlet also sets the content length. In order to do this, the servlet must buffer the output internally, which it does by using the `StringBuffer`. The code would have been a little cleaner if it had used a `String` and the + operator, but the `StringBuffer` class is generally much more efficient.[1]

[1] Buffering data internally is the only way to determine the content length reliably, but it is grossly inefficient even when a `StringBuffer` is used. This is why so few pages bother with the content length at all.

This servlet also illustrates the use of the `getParamterValues()` method, which contains information about all the data that a form has sent to the servlet. This method is also used behind the scenes by JSPs, as this is what populates the `param` map often used in expressions.

11.2.3 Convenience Methods

Most of the time, servlets will send out formatted HTML, in which case they will set the status to `SC_OK`. However, a servlet might generate two other common kinds of responses: errors and redirects. Error pages are all too common; they contain a numeric code indicating the kind of error and, usually, a short, cryptic message telling the user that something went wrong. Rather then setting the status to the appropriate code and printing the error text, a servlet can simply use the `sendError()` method. This message takes an integer representing the error code and a string to use as the error text. This text will be enclosed by `<body></body>` tags, which makes things a little easier for the servlet programmer. A typical use might look something like

```
res.sendError(res.SC_INTERNAL_SERVER_ERROR,
  "Yikes, something went awry! Please check back later.");
```

Redirects tell a browser that the page for which it has asked has moved to a new URL; the browser will respond by asking the appropriate server for the new URL. This technique is common for preventing links from getting stale, but it is also used to send a user to a different page, based on a certain condition. For example, a user who tries to access a page without having the permission to read it can be sent to a page explaining what to do in order to obtain that permission.

`HttpServletResponse` provides a method, `sendRedirect()`, to make redirects easier to use. It takes a single string as an argument, which should contain the URL to which the user should be sent. Unlike the `sendError()` method `sendRedirect()` does not takes a string describing why the user is being sent elsewhere, as in practice, the user will never have a chance to read this message before the browser loads the new page. Further, `sendRedirect()` does not need a status argument, as the status is assumed to be `SC_MOVED_TEMPORARILY`. A typical use for this method would look like this:

```
res.sendRedirect("http://www.brunching.com");
```

The URL should be complete and cannot be relative, unlike an `HREF`. Most browsers will correctly handle a relative redirect, but this is not part of the official HTTP specification.

11.3 Servlet Events

Often, developers may wish to take certain special actions at various points during the life cycle of either a servlet or the application as a whole. Some of these actions can be handled by simply adding code to the appropriate life-cycle method, but other methods are not available to the general programmer. One common case is session handling. If an application could determine when a session was being expired, it would be possible to move data from the session into a database before it disappeared forever. Likewise, knowledge of when a session was being created would make it possible to move data from the database into the session. The result would be sessions that are effectively immortal, providing a seamless experience for the user.

This could be achieved in many ways, but a natural mechanism is already provided by the JavaBean specification by way of events and listeners, as discussed in the preceding chapter. The idea is that the servlet engine will fire off an event representing various activities, and programmers can build listeners to capture and act on these events. Conceptually, this works just like any other bean event; it is simply a matter of implementing the right listener interfaces and working with the corresponding events. An outline of the session backup listener is shown in Listing 11.4.

Listing 11.4 The outline of a session listener

```
package com.awl.jspbook.ch11;
```

```
import javax.servlet.http.*;

public class BackupListener
    implements HttpSessionAttributeListener
{
    public void
    sessionCreated(HttpSessionEvent event)
    {
        HttpSession session = event.getSession();

        // ... load the session from the database ...
    }

    public void
    sessionDestroyed(HttpSessionEvent event)
    {
        HttpSession session = event.getSession();

       // ... back up the session to the database ...
    }

    public void
    attributeAdded(HttpSessionBindingEvent event)
    {
        // ... It may not be necessary to do anything
              for this
        // event, but the interface requires it be provided
    }

    public void
    attributeRemoved(HttpSessionBindingEvent event)
    {
        // ... It may not be necessary to do anything
              for this
        // event, but the interface requires it be provided
    }
    public void
    attributeReplaced(HttpSessionBindingEvent event)
    {
        // ... It may not be necessary to do anything
              for this
        // event, but the interface requires it be provided
    }
}
```

Corresponding interfaces can capture the creation and destruction of a `servletContext`, as well as the addition and removal of attributes to a `servletContext` or session. See the servlet specification for details.

11.4 Forwarding and Including Requests

Many examples throughout this book have sent a user to another page, using a technique very different from the redirects. The `jsp:forward` tag is sort of a "server-side" redirect, as it instructs the server to generate a different page rather than tells the browser to load a new URL. This is related to the `jsp:include` tag, which includes the body of one JSP, HTML page, or servlet in another. There is no corresponding way to do this on the client side, at least not one that works on all browsers.

Both of these tags are handled internally by the `RequestDispatcher` class, which, as the name implies, can dispatch a request to another resource in the system. It does this through two methods called, appropriately enough, `forward()` and `include()`. Both of these methods take as arguments the request and response objects that the calling servlet was passed. As might be expected, these objects will end up getting passed to the target servlet's `service()` method. Listing 4.14 showed a JSP that used a `jsp:forward` tag and input from the user to send the user one of three pages. Listing 11.5 shows how a servlet would accomplish the same thing.

Listing 11.5 A servlet that forwards requests

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class DispatchServlet extends HttpServlet {
  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    ServletContext sc = getServletContext();
    RequestDispatcher rd;

    String which = req.getParameter("which");

    if(which != null) {
      if(which.equals("red")) {
rd = sc.getRequestDispatcher("/chapter11/red.jsp");
rd.forward(req,res);
      } else if(which.equals("green")) {
rd = sc.getRequestDispatcher("/chapter11/green.jsp");
rd.forward(req,res);
      } else if(which.equals("blue")) {
rd = sc.getRequestDispatcher("/chapter11/blue.jsp");
rd.forward(req,res);
      } else {
res.sendError(res.SC_INTERNAL_SERVER_ERROR,
      "A page was requested that does exist!");
      }
    } else {
      res.sendError(res.SC_INTERNAL_SERVER_ERROR,
    "No destination page was specified!");
    }
  }
}
```
The `which` parameter must be provided by a form or within the URL. The easiest way to see this page in action would be to direct a browser to `"/chapter11/dispatch?which=red"`.

It is important to realize that once the target page finishes, the `forward()` method will return, and the calling servlet will regain control. However, the output stream will have been closed, so the servlet should not try to set new headers or send new data. It can clean up any global resources, if necessary. The calling servlet also cannot print any data before calling `forward()`, because the target page will likely set one or more headers, and as observed previously, this may not work if data has already been sent. A similar restriction applies to the `include()` method. When including a page, the included page cannot set any headers, because the servlet calling `include()` may have already printed data.

11.5 Using Scopes from Servlets

Chapter 3 discussed the various scopes in which beans can live. Although these scopes are usually accessed by setting the `scope` field in a `jsp:useBean` tag, they can also be accessed through scriptlets or through code in a servlet. Listing 11.6 shows a servlet with a counter, which works much like the JSP in Listing 3.8 did.

Listing 11.6 A servlet with a page counter

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class Counter1 extends HttpServlet {
  private int count;

  public void init(ServletConfig sc)
    throws ServletException
  {
    super.init(sc);
    count = 0;
  }

  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void doPost(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void handle(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    res.setStatus(res.SC_OK);
    res.setContentType("text/html");

    PrintWriter out = res.getWriter();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>A Counter</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("This page has been accessed ");
    out.println(count);
    out.println(" times");
    out.println("</BODY>");
    out.println("</HTML>");

    count++;
    out.close();
  }
}
```

The `count` variable here is not technically in a scope. But because the value will persist as long as the servlet is active, the variable behaves as if it were in the application scope in some sense. However, any value that was truly in the application scope could be obtained from any JSP or other servlet, but `count` is available to only this one servlet. Listing 11.7 shows how a similar counter can be used from the session scope, so the page will count how often each user has visited it.

Listing 11.7 A servlet with a session-based counter

```
package com.awl.jspbook.ch11;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Counter2 extends HttpServlet {
  private static final Integer ONE = new Integer(1);

  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void doPost(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }


  public void handle(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    HttpSession theSession = req.getSession();
    Integer count =
      (Integer) theSession.getAttribute("count");

    res.setStatus(res.SC_OK);
    res.setContentType("text/html");

    PrintWriter out = res.getWriter();

    out.println("<HTML>");
    out.println("<HEAD><TITLE>A Counter</TITLE></HEAD>");
    out.println("<BODY>");

    if(count == null) {
      out.println("This is your first visit to this page!");
      count = ONE;
    } else {
      out.println("You have seen this page ");
      out.println(count);
      out.println(" times before");
    }

    theSession.setAttribute("count",
    new Integer(count.intValue() + 1));

    out.println("</BODY>");
    out.println("</HTML>");
    out.close();
  }
}
```

The call to `getSession()` will create a new session if one has not already been given to the current user. This will cause a new cookie to be sent out along with this page. Once the session has been obtained, the `count` variable is accessed with the call to `getValue()`.

Sessions can hold only objects, not primitive types, such as integers, which is why the value is stored as an `Integer`.

The first time `count` is requested, it will not be in the session, and so `null` will be returned. This allows the servlet to know that the session is new, so it can print a different message. This corresponds closely to putting code or text in the body of a `jsp:useBean` tag to do something special when the bean is created.

The variable `ONE` provides a very slight performance improvement, as it saves having to construct a new `Integer` for every new request, which is both faster and uses less memory. Technically, this means that every session will be sharing the same object, at least for each user's first visit. Because the value of `ONE` is never changed, this does not present a problem. However, this does hint at some interesting ways different users could share changeable data. For example, two users could have the same `HashMap` in their sessions, and any values placed in this map by one user could be seen by the other.

The application and request scopes work almost exactly the same way as the session scope; the only difference is the methods used to store and retrieve objects. For the request scope, objects are retrieved from the request object by calling `request.getAtribute(name)`, and objects are stored using the corresponding `request.setAtribute(name,value)`.

The application scope is stored in the `ServletContext` object, as only one of these is in any given server. A servlet can store data in the application scope by calling

```
ServletContext sc = getServletContext();
sc.setAttribute(name,value);
```

Likewise, once the `ServletContext` has been obtained, objects can be retrieved from it with `sc.getAttribute(name)`.

11.6 Using Beans from Servlets

Beans are as useful for servlet authors as they are for JSP authors but are not quite as easy to use from a servlet. Obtaining the bean is pretty straightforward, as shown in Listing 11.8, which uses the bean containing album information for Siouxsie and the Banshee's "Tinderbox" from Chapter 3.

Listing 11.8 A servlet that uses a bean

```
package com.awl.jspbook.ch11;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.beans.*;
import com.awl.jspbook.ch03.AlbumInfo;

public class CDInfo extends HttpServlet {
  public void doGet(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }

  public void doPost(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    handle(req,res);
  }

  public void handle(HttpServletRequest req,
     HttpServletResponse res)
    throws IOException,ServletException
  {
    AlbumInfo tinderbox;

    res.setStatus(res.SC_OK);
    res.setContentType("text/html");

    PrintWriter out = res.getWriter();

    try {
      tinderbox = (AlbumInfo) Beans.instantiate(
    getClass().getClassLoader(),
    "tinderbox3");
    } catch (Exception e) {
      tinderbox = null;
    }

    out.println("<HTML>");
    out.println("<HEAD><TITLE>Album Info</TITLE></HEAD>");
    out.println("<BODY>");

    if(tinderbox == null) {
      out.println("The bean could not be found or loaded");
    } else {
      out.println("<P>Album name: "
  + tinderbox.getName() + "</P>");

      String tracks[] = tinderbox.getTracks();

      out.println("<P>Tracks:</P>");
      out.println("<OL>");
      for(int i=0;i<tracks.length;i++) {
out.println("<LI>" + tracks[i]);
      }
      out.println("</OL>");
    }

    out.println("</BODY>");
    out.println("</HTML>");

    out.close();
  }
}
```

The call to `instantiate()` is what performs the real magic in this example. Here, it is used to load a serialized bean, but if it were given the name of a class instead of a file name, it would have loaded the class, called its constructor, and returned a new instance. The first argument to `instantiate()` is a `ClassLoader`, which, as the name implies, is a class that loads other classes. Every Java class can get access to the class loader that loaded it by calling `getClassLoader()`, and any object can get its class by calling `getClass()`.

Once the bean is loaded, it is treated like any other class. In particular, the bean is first cast into the appropriate type, and then the methods of this class are called directly. This is not using the full power of bean introspection, which can dynamically determine the properties and methods of a bean at runtime. Introspection enables important JSP abilities, such as the automatic setting of properties from form parameters.

Servlets can do introspection, but it is beyond the scope of this book, so an example will not be provided here. More information can be found in a good book on beans or the Java documentation, starting with the `getBeanInfo()` method of the `java.beans.Introspector` class, which can be found at http://java.sun.com/beans/javadoc/java.beans.Introspector.html.

Once it has been obtained by a call to `instantiate()`, a bean may be stored in any of the four scopes. If a bean is placed in a scope by a servlet, a JSP can later retrieve the bean from the scope through the normal `jsp:useBean` tag. The reverse is also true; any bean placed in a scope by a JSP can be obtained and used by a servlet. For example, suppose that Listing 11.7 had included the following lines:

```
ServletContext sc = getServletContext();
sc.setAttribute("tinderbox",tinderbox);
```

In that case, a JSP could access this bean with the following tag:

```
<jsp:useBean id="tinderbox"
    class="com.awl.jspbook.ch05.AlbumInfo"
    scope="application"/>
```

Note that when trying to load a serialized bean, a servlet or JSP will look for a file with the .ser suffix. In this case, the file that is loaded will be tinderbox.ser.

The ability to store beans in the various scopes provides an easy and convenient way for servlets and JSPs to share data and make the transition between them completely transparent to users. In fact, this feature allows servlets to act as controllers in the model/view/controller sense.

Typically, a request will first go to a servlet, which will do some complex processing needed to set up a model, which will be represented as a bean. Once the processing is complete, the bean will be placed in the page or request scope. The servlet will then forward the request to one of several JSPs, based on various rules. The final JSP, in true view fashion, will be concerned only with presenting the data in the bean. A simple use of this design is illustrated over the next several listings, which demonstrate an application that takes a list of numbers and computes their sum and average. Listing 11.9 begins the process with the servlet.

Listing 11.9 A servlet that passes a bean to a JSP

```
package com.awl.jspbook.ch11;

import java.util.StringTokenizer;
import java.io.IOException;

import javax.servlet.*;
import javax.servlet.http.*;

public class SumAvgServlet extends HttpServlet {
    public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
        throws IOException,ServletException
    {
        RequestDispatcher rd;
        ServletContext sc = getServletContext();
        String values     = req.getParameter("values");
        SumAvgBean data   = new SumAvgBean();
        int num          = 0;
        int sum          = 0;
        double avg       = 0;
        int count        = 0;

        /* Add the bean to the request scope */
        req.setAttribute("data",data);

        data.setValues(values);

        if(values == null) {
            rd = sc.getRequestDispatcher(
                "/chapter11/sumavgform.jsp");
            rd.forward(req,res);
        } else {
            StringTokenizer st =
              new StringTokenizer(values,",");
        String token;

        while(st.hasMoreTokens()) {
            token = st.nextToken();
            try {
                num = Integer.parseInt(token);
            } catch (NumberFormatException e) {
                data.setBad(token);
                rd = sc.getRequestDispatcher(
                     "/chapter11/sumavgerror.jsp");
                rd.forward(req,res);
                return;
            }

            count++;
            sum = sum + num;
        }
     }

     data.setSum(sum);
     data.setAvg(sum/count);
     rd = sc.getRequestDispatcher(
          "/chapter11/sumavgresults.jsp");
     rd.forward(req,res);
  }
}
```

This servlet creates a bean called `data` and puts it in the request scope. The bean is shown in Listing 11.10.

Listing 11.10 A bean that contains sum and average data

```
package com.awl.jspbook.ch11;
```

```
public class SumAvgBean {
    private int sum;
    private double avg;
    private String bad;
    private String values;

    public void setBad(String bad) {this.bad = bad;}
    public String getBad() {return bad;}

    public void setValues(String values) {
      this.values = values;
    }
    public String getValues() {return values;}

    public void setSum(int sum) {this.sum = sum;}
    public int getSum() {return sum;}

    public void setAvg(double avg) {this.avg = avg;}
    public double getAvg() {return avg;}
}
```

The servlet then checks whether it has been given any values. If not, it sends the request onto sumavgform.jsp, which is shown in Listing 11.11.

Listing 11.11 A form in which numbers may be entered

```
<html>
<body>

Enter a list of numbers, seperated by commas. I will
compute their sum and average.

<form action="sumAvg" method="post">
<input type="text" name="values">
<input type="submit">
</form>
```

```
</body>
</html>
```

All the numbers provided by the user are passed to the servlet in a single string. The servlet then breaks this string into individual numbers and starts to add them. If a non-number is encountered while processing, the servlet places the offending text in the bean's `bad` property and passes the request to sumavgerror.jsp, which is shown in Listing 11.12.

Listing 11.12 A JSP that displays an error value from a bean

```
<jsp:useBean
  class="com.awl.jspbook.ch11.SumAvgBean"
  id="data"
  scope="request"/>

<html>
<head><title>Error</title></head>
<body>

<p>
I was unable to complete your request, because
<jsp:getProperty name="data" property="bad"/>
is not a number.
</p>

<form action="sumAvg" method="post">
<input type="text" NAME="values"
  value="<jsp:getProperty
          name="data"
          property="values"/>">
<input type="submit">
</form>

</body>
</html>
```

Finally, if all goes well, the resulting sum and average are placed in a bean, and the values are sent to sumavgresult.jsp, which is shown in Listing 11.13.

Listing 11.13 A JSP that displays results from a bean

```
<jsp:useBean
  class="com.awl.jspbook.ch11.SumAvgBean"
  id="data"
  scope="request"/>

<html>
<head><title>Results</title></head>
<body>

The sum of your numbers is
<jsp:getProperty name="data" property="sum"/>.
<p>

The average of your numbers is
<jsp:getProperty name="data" property="avg"/>.

</body>
</html>
```

If this example were to be written solely as a JSP and a bean, the JSP would need to handle differentiating between the cases in which input is or is not provided. The JSP would also need to handle the error conditions. Both of these situations would need to be done either in Java or with some messy conditional tags. However, the servlet can handle both the *application logic* and what might be called the *page-flow* logic. This leaves the JSPs to do what they do best: handle the presentation.

11.7 The JSP Classes

As discussed in Chapter 1, a .jsp file is translated to a Java file by the page compiler, and this file is then compiled and run to produce the page output. Now that servlets have been examined in some depth, it should be clearer what this translation entails. For example, consider a simple JSP:

```
Hello!

<jsp:useBean
  id="aBean"
  class="com.awl.jspbook.ch11.SomeBean"/>

<jsp:getProperty
  name="aBean"
  property="aProperty"/>
```

This could turn into a servlet with the following `service()` method:

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
{
     response.setStatus(res.SC_OK);
     response.setContentType("text/html");

     PrintWriter out = response.getWriter();
     out.println("Hello!");

     com.awl.jspbook.ch11.SomeBean aBean =
       (com.awl.jspbook.ch11.SomeBean)
         Beans.instantiate(getClass().getClassLoader(),
                           "com.awl.jspbook.ch11.SomeBean");

     out.println(aBean.getAPropety());
}
```

This service() method above is not precisely what is generated, but it gives a sense of the kind of translations that take place. In fact the generated file does not even implement the `Servlet` interface directly, nor does it extend `HttpServlet`. Instead it implements an interface called `HttpJspPage` from the `javax.servlet.jsp` package. `HttpJspPage` extends another interface called `JspPage`, and `JspPage` extends `Servlet`. In other words, there is a whole hierarchy of JSP-related classes that closely mirrors the servlet hierarchy. `JspPage` adds two additional methods to the `Servlet` interface: `jspInit()` and `jspDestroy()`, which act much like the `init()` and `destroy()` methods in the `Servlet` class. The only difference is that `jspInit()` is not passed a `ServletConfig` object when it is called; however, the `ServletConfig` can be obtained via the `getServletConfig()` method.

`HttpJspPage` adds one additional method, `_jspService()`. This method is passed an `HttpServletRequest` and `HttpServletResponse`, just like the `service()` method. It is worth noting at this point that humans never write a `_jspService()`. This method is built by the JSP engine, based on the original JSP file. If a programmer also provides a method with this name, there would be a conflict. In practice, this is not a problem, as any code that could be put in a service method can be put in a scriptlet in the JSP page.

The `javax.servlet.jsp` package also provides a number of classes that provide additional information or make life easier for developers. Most of these classes will be used only by the JSP engine, but page authors may well want to use the `PageContext` class. An instance of this class is always available in a JSP as an implicit object called `pageContext`.

The `PageContext` class provides a number of utility methods for handling scoped data and hides the details of how various scopes are implemented. This means that instead of having to know that the request scope is implemented by the `HttpServletRequest` class, the application scope is in the `ServletContext`, and a single method can be used to get or set data from any scope. These methods follow the naming conventions already discussed and are called `getAttribute()` and `setAttribute()`. They work much like the identically named functions from `HttpServletRequest` and `ServletContext` but take an additional parameter specifying which scope to use. Listing 11.14 shows a JSP that uses these methods to create a per session counter, just as Listing 11.6 did in a servlet.

Listing 11.14 A JSP that uses the `PageContext` class

```
<HTML>
<HEAD><TITLE>Another counter</TITLE></HEAD>

<BODY>

<% Integer count = (Integer)
   pageContext.getAttribute("count",
                            PageContext.SESSION_SCOPE); %>
```

```
<% if (count == null) { %>
   <P>This is your first visit to this page!</P>
   <% count = new Integer(1); %>
<% } else {% >
   <P>You have seen this page
   <%= count %> times before </P>

   pageContext.setAttribute("count",
                        new Integer(count.intValue()+1),
                        PageContext.SESSION_SCOPE); %>
<% } %>

</BODY>
</HTML>
```

`SESSION_SCOPE` is a final integer indicating that the methods should use the session scope. The other scopes have similar definitions. This code will turn into a Java class that is almost identical to Listing 11.6 but is a little easier to write and maintain, if only because all the calls to `out.println()` are avoided.

11.8 Intercepting Requests

In fulfilling their role as controllers, servlets often need to access a request before it goes to a JSP, in order to set up some beans or make a decision about which JSP should be invoked. The pattern of using a servlet to do some preprocessing before passing control to a JSP is so common that it has been formally introduced into the servlet specification by way of the `Filter` class.

The idea is that every request is allowed to pass through a *filter chain*, whereby each element in the chain is a class that may manipulate arbitrary data. Often, the last element in a chain is a JSP.

Normally, once it has finished its task, a particular filter will pass the request to the rest of the chain, but it is also possible for a filter to "hijack" a request and handle it on its own by generating its own output, issuing a redirect, or disallowing access. This makes filters well suited to handling security, which will also be discussed in the next chapter.

Before tackling the complex issues of security, here is a simpler example that illustrates yet another way in which pages can display the current date and time. Instead of using a custom tag, as was done previously, this version uses a filter that adds the data to the request, as shown in Listing 11.15.

Listing 11.15 A filter

```
package com.awl.jspbook.ch11;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.text.*;

public class DateFilter implements Filter {
    private DateFormat df = null;
    public void init(FilterConfig conf)
        throws ServletException
    {
        df = new SimpleDateFormat(
                 conf.getInitParameter("format"));
    }

    public void doFilter(ServletRequest req,
                         ServletResponse res,
                         FilterChain chain)
        throws ServletException,IOException
    {
        HttpServletRequest hreq = (HttpServletRequest) req;

        hreq.setAttribute("date",
                          df.format(new java.util.Date()));

        chain.doFilter(req,res);
    }
    public void destroy() {}
}
```

Like servlets, filters are created when the system starts up; at that point, they can be initialized through the `init()` method. Here, a configuration parameter is used to determine how to format the date.

When a request comes in, the `doFilter()` method is called with a `ServletRequest` and `ServletResponse` and a new object, called `FilterChain`, representing the rest of the chain. The filter may then do anything it likes with the request and response and then should call `doFilter()` on the `FilterChain` object to pass the request to the next filter along the chain or the final JSP. Note that the filter has no knowledge about what the next object in the chain will be, which allows filters to be connected together as needed. The order in which filters will be invoked and the set of URLs that will be filtered are controlled by the configuration file for the Web application, which is discussed in Appendix B.

If the filter from Listing 11.15 is installed, a JSP can display the current time as simply as

```
<c:out value="${date}"/>
```

Using filters to set up data in this way can avoid a lot of the overhead of doing so in JSPs or having to learn the tags in an extra custom tag library.

11.9 Summary and Conclusions

The servlet API provides the foundation on which JSPs are built, and understanding this API can come in handy for page authors. The servlet API defines a life cycle for servlets, starting with an `init()` method that is called when the servlet first loads, a `service()` method that is called for each request, and a `destroy()` method that is called before the servlet is retired. The `init()` method may allocate resources that requests will later need, and `destroy()` can free these resources. The `service()` method is passed a request and a response object, which it uses to get information about the request, set information about the response, and send the data.

Servlets can use all the scopes discussed in Chapter 3. Servlets can also interact with JSPs, using beans as an intermediary. Typically, the servlet will do the computation, build a bean with the results, and send the bean on to the JSP for formatting, using the `forward()` method. This provides the cleanest separation between logic and presentation.

JSPs are ultimately servlets. Thus, for pages with any significant amount of HTML, a JSP will almost always be the preferred means of creating pages, as it is easier to read and maintain and it avoids all the `print` statements. On the other hand, pages that are dominated mostly by code expressing page logic may be better off as a servlet, as this will avoid having to put everything in scriptlets.

Chapter 12. The Controller

So far, little has been said about the controller side of the model/view/controller paradigm. One reason is that a great deal can be done without a formal controller. Without a model, there would be nothing to show; without a view, there would be no way to show it. But so far, it has been possible to muddle along by putting controller functionality into one of the other layers. After all, the whole Java News Today site was built without a controller. The site has been able to get away with this only because the models and views have been pretty closely matched. Most of Java News Today's pages have had a one-to-one correspondence among page elements, form fields, bean properties, and database fields. The second, and more relevant, reason that controllers have not yet been discussed is that it would have been impossible to do so without a thorough knowledge of Java. No special JSP tags or similar building blocks can be used to build a controller; they must be hand built in Java. Fortunately, an excellent framework simplifies the task of building such controllers.

It was also necessary to understand bean implementations and servlets, as controllers will mediate between user actions controlled by servlets and JSPs which are themselves servlets and beans. Therefore, the Java code that comprises the controller must be able to interface with both of these APIs.

12.1 Some Common Controller Tasks

Before building a controller, it is necessary to identify what it should do. This can be determined by examining what has been put but that may not belong in the model and view. Many JSPs throughout this book have followed a similar pattern; a form part of the view has a number of fields for a user to fill in; when the form is submitted, the values are loaded into a bean the model via `jsp:setProperty` tags. Then another `jsp:setProperty` may set a pseudoproperty, such as `save`, which causes the bean to write the values to a database.

In this system, the beans are doing two unrelated things: modeling the conceptual entity being manipulated, which is good, and talking to forms, which is bad. The latter requires that the model and view must look pretty similar. At the very least, form names must match property names, but more generally, developers must think of these two very different things as connected in some way.

To separate the model from the view more cleanly, it would therefore make sense to begin by splitting the bean into two: one that will truly model the system and the other that will talk to the form. Doing this allows a cleaner delineation between the view elements, consisting of the JSP containing the form and the form bean, and the model, consisting of another bean that holds and manages the data to be maintained or modeled.

This distinction between form data and model data has already been present in a few situations. Recall Listing 5.9, which allows a user to add a comment to a JNT article, and Listing 7.3, which allows a reporter to create a new article. In both of these cases, the underlying model needs to keep track of the user performing the action. This information was provided by adding hidden fields to the form. In other words, the view was modified to accommodate the needs of the model, although it would have been cleaner to introduce a controller that would have added the user information without having to impact the view.

Looking at the boundary between model and view in this way provides an opportunity to start thinking about error conditions. So far, all the examples have been pretty lax about the form inputs and have allowed users to enter into fields any data, even if it did not make sense. The discussion on Listing 3.5, for examples, mentions that an error would be displayed if a user tried to add something that was not a number, such as the string `A`. This error would arise even if the user provided something that looks like a number to humans but not to Java, such as `8,442.23`; without extra work, Java cannot recognize an expression with a comma as a number. Worst of all, the error displayed is useful to JSP developers but will be totally unfriendly to any end users.

To address this issue, it is now time to start considering the problem of *form validation:* ensuring that the user-provided values are both legal and sensible for the type of data they are meant to represent. Also, a means to report problems back to users in a useful and friendly way will be needed. The question then becomes whether this validation should be done in the beans making up the model or the new form beans that are part of the view.

Because it is the model's job to store and act on the data, the model should usually be responsible for all validation as well. Certainly, some kinds of validation can happen *only* in the model; for example, in an online catalog, the *model* must check whether an item is in stock when the user tries to purchase it. Likewise, a bean modeling a calculator that can do division should be responsible for ensuring that the denominator is not zero.
However, a few kinds of validation are not intrinsic to the model but arise as part of the way the model and view communicate. Again consider a calculator model, which may have a method called `add` that takes two integers as arguments. When used directly by a Java program, this method could not be invoked with the letter `a` as an argument. In essence, the Java compiler would do the validation before the model was ever used.
The dynamic nature of JSPs bypasses this check by the compiler. This check could be put into the calculator bean by adding to the `add` method a version that takes strings as arguments and ensures that they look like numbers before proceeding. However, it has been repeatedly stressed that a view should not need to know the details of how the model works, yet here the model would be changed, based on the details of the view. One reasonable compromise is to note that all *semantic* validation must be done in the model, which is the only part of the system that knows what the data means, but that simple *syntactic* validation can be done by the view, which in this case means by the new form beans.
The controller's role in all this should now start becoming clear. The controller will take values from the form and provide them to the form bean and will then ask that bean to validate them. If the validation fails, the controller will send the user back to the original form, providing the validation errors. The form can then display these errors and ask the user to correct them. Once the validation succeeds, the controller will pass data from the form bean to the model bean, along with any additional information, such as the current user. The controller will then perform the desired action on the model, such as invoking a `save()` method, and then send the user to the appropriate page from which to continue.
In addition to moving data from forms to the back-end model, controllers can prepare beans that are used to move data from the model to JSPs. For example, the JNT article page expected to be called with an `articleId`, which it would then use to load an `ArticleBean`. The controller can detect that a user is going to the article page and can prepare the appropriate `ArticleBean` on the page's behalf. This means that the view will no longer need to deal with loading or initializing elements of the model. This will be moved to the controller, where it belongs.
Finally, the controller can enforce security policies. For example, it can ensure that only reporters are allowed to access the article creation page.

12.2 Support for Controllers: Struts

It is clear that a great deal of new infrastructure is needed to support controllers. Means are needed to associate form beans and controller actions with forms. These controller actions must know where to send users after successfully completing an action. A way is needed to send validation errors to users. Of course, the Java classes to implement the controller actions must also be written.

This seems like a lot of work, but most of it has already been done by a toolkit called *struts*, a free, open-source framework from the Jakarta projects, the same fine folks who built Tomcat. Struts is much more than a way to build controllers; it is a complete application framework containing view elements in the form of custom tags, a controller framework, and much more. Although this book can cover only a small portion of what struts can do, readers are encouraged to find out more at http://jakarta.apache.org/struts/.

Among the many other services it provides, struts adds another layer between data and presentation. Up until now, content on a page either could be hard-coded in the page or come from a bean. A typical example is the CD database from Chapter 6; the name of the artist was provided by a bean, but the preceding string, `Albums by`:, was in the page itself. Struts takes the approach that only structural elements should be part of a JSP, that is, table cells, paragraph breaks, and so on. All other text, such as messages to users, labels for form elements, and so on, should live in a common file separate from all JSPs. Separating content from structure ensures some level of consistency, as a message used on several pages is defined in one place. It also makes it easier to make changes, as there is no question about where to find a particular message.

Most important, isolating all a site's text in one file makes it possible to support multiple languages and locales easily. A site might have multiple versions of such a file. One for English might contain:

```
message.entry=Welcome
message.departure=Goodbye
```

One for German might contain

```
message.entry=Willkommen
message.departure=Auf Wiedersehen
```

Using tags from the struts library or a utility class, a developer can refer to `message.entry`, and the appropriate text will be retrieved, depending on whether the location has been set to an English-speaking or a German-speaking locale. Note that the dots in the names do not necessarily imply any sort of hierarchy, as the dots in bean properties do. Here, the dots are simply a convenient way to group messages mentally into convenient units.

12.2.1 Using Struts

The servlet `org.apache.struts.action.ActionServlet`, is the entry point to struts. This servlet is typically installed such that it will handle all URLs ending in `.do`. The servlet reads a configuration file to determine what to do with each URL.
To make this more concrete, let's use struts to rebuild the calculator from Chapter 3. The full set of messages used by this little calculator is shown in Listing 12.1.

Listing 12.1 The application messages

```
prompt.number1=First number
prompt.number2=Second number


message.result=The sum is:


button.save=Add
button.reset=Reset
button.cancel=Cancel


error.calculator.missing1=\
<li>Please provide a value for the first number</li>


error.calculator.missing2=\
<li>Please provide a value for the second number</li>


error.calculator.bad1=\
<li>The first value does not look like a number</li>
```

```
error.calculator.bad2=\
<li>The second value does not look like a number</li>


errors.header=\
Please correct the following problem(s) and try again:<ul>


errors.footer=\
</ul><hr>
```

Next, the model needs to be defined, which is quite simple and is shown in Listing 12.2.

Listing 12.2 The calculator model

```
package com.awl.jspbook.ch12;


public class Calculator {
    private double number1;
    public double getNumber1() {return number1;}
    public void setNumber1(double number1) {
        this.number1 = number1;
    }

    private double number2;
    public double getNumber2() {return number2;}
    public void setNumber2(double number2) {
        this.number2 = number2;
    }


    private double sum;
    public double getSum() {return sum;}
    public void setSum(double sum) {this.sum = sum;}

    public void computeSum() {
        sum = number1 + number2;
    }


}
```

This class has simple properties for the two inputs and the resulting sum, as well as a method, `computeSum()`, that will perform the computation. In this case, it would be easy enough to have the controller compute the sum and store it by calling `setSum()`, but that would be inappropriate, as the model should be responsible for managing all its data. Note that nothing in this bean knows anything about taking values from a form or parsing numbers with commas or anything else.

The next thing to build is the bean that will directly interface with the HTML form. This is shown in Listing 12.3.

Listing 12.3 The calculator form

```
package com.awl.jspbook.ch12;



import java.text.DecimalFormat;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;



public class CalculatorForm extends ActionForm {
    private String number1;
    public String getNumber1() {return number1;}
    public void setNumber1(String number1) {
        this.number1 = number1;
    }



    private String number2;
    public String getNumber2() {return number2;}
public void setNumber2(String number2) {
    this.number2 = number2;
}


public ActionErrors validate(ActionMapping mapping,
                          HttpServletRequest request)
{
    ActionErrors errors = new ActionErrors();
    DecimalFormat f     =
        new DecimalFormat("###,###.##");

    if(empty(number1)) {
        errors.add("number1",
                   new ActionError(
                       "error.calculator.missing1"));
    } else {
        try {
            f.parse(number1);
        } catch (Exception e) {
            errors.add("number1",
                       new ActionError(
                           "error.calculator.bad1"));
        }
    }


    if(empty(number2)) {
        errors.add("number2",
                   new ActionError(
                       "error.calculator.missing2"));
    } else {
        try {
            f.parse(number2);
        } catch (Exception e) {
            errors.add("number2",
                        new ActionError(
                            "error.calculator.bad2"));
                }
            }


            return errors;
        }


        private boolean empty(String s) {
            return s == null || s.trim().length() == 0;
        }

    }
```
Note that this class extends a struts class called `ActionForm`. In struts terms, each thing the controller does is considered an `Action`, and data is made available to an `Action` via an `ActionForm`.

The `CalculatorForm` contains two simple properties to hold the inputs from the form. These properties are `Strings`, whereas those in the model are `doubles`. This makes sense, as a calculator can add numbers, but the form should allow the user to enter arbitrary text, including representations of numbers with commas.

The `CalculatorForm` allows the inputs to be validated through the `validate()` method; this method is defined in the `ActionForm` base class and will be called automatically by struts when the form is submitted. The method is passed an `ActionMapping`, a struts class containing information about the application, along with an `HttpServletRequest`, which contains the usual request information. Neither of these is used in this example, but both are available for more sophisticated kinds of validation.

The `validate()` method simply checks that values have been provided for both inputs and that Java is able to turn the inputs into numbers. This latter test is done by attempting to parse the data by using the `java.text.DecimalFormat` class, which here has been told to allow numbers with commas. For more information about this class and how it is used, consult the JDK documentation.

If a value is missing or malformed, a new `ActionError`, called `errors`, is added to the set maintained by the `ActionErrors` object. The exact text of these error messages comes from the file in Listing 12.1, which means that these errors could be reported in any language for which a file had been built.

At the end of the method, the `errors` are returned. Internally, struts will check this value. If it is empty, there were no problems, and the form can be processed; otherwise, the user must be informed of the errors and given the opportunity to fix them.

Now that the form bean is completed, it is time to write the class that will implement the action. This is shown in Listing 12.4.

Listing 12.4 The `Action` handler

```
package com.awl.jspbook.ch12;



import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.text.DecimalFormat;
import java.util.Locale;



import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.util.*;



public final class CalculatorAction extends Action {
    public ActionForward perform(ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException
    {
         // Populate the input form
         if (form == null) {
             form = new CalculatorForm();
```

```
          request.setAttribute(mapping.getAttribute(),
                               form);
      }

      CalculatorForm calcForm = (CalculatorForm) form;

      // Build the model
      Calculator calc        = new Calculator();
      calc.setNumber1(getNumber(calcForm.getNumber1()));
      calc.setNumber2(getNumber(calcForm.getNumber2()));
      calc.computeSum();

      // Store the model in the request so the result
      // page can get to it
      request.setAttribute("calc",calc);

      return (mapping.findForward("success"));
  }

  private double getNumber(String s) {
      DecimalFormat d = new DecimalFormat("###,###.##");
      try {
          Number n = d.parse(s);
          return n.doubleValue();
      } catch (Exception e) {
          // No need to worry about parse errors, the
          // check in the form bean assures us of that!
      }

      return 0.0;
```

```
    }
}
```

This class extends another struts class, `Action`, whose `perform()` method will be called after the form bean successfully validates the inputs. This method is invoked with the form bean, the same `ActionHandler` that was passed to the `validate()` method, and the request and response. The method ensures that there is a valid form bean, constructs an instance of the `Calculator` model bean, populates it, and then finishes the process by calling `computeSum()`. In a more complicated example, the model bean might come from a database or other repository rather than being constructed within the `Action`. Finally, the calculator is stored in the request, which sets everything up for the result page to display the sum, and the name of the result page is returned. This name is not hard-coded but rather is kept in the `ActionMapping` under a key called `success`.

That completes the set of classes. It may seem at this point that a lot of overhead is needed to do something as simple as adding two numbers. However, as the task gets more complicated, the amount of overhead diminishes proportionally. A real system needs form validation and a way to perform the required actions, and so writing some amount of Java code is inescapable. The more complex these tasks become, the lighter the struts framework will seem in comparison, and the advantages of using such a framework will rapidly become obvious.

Now that the classes are completed, struts needs to be told how to use them. This is accomplished by providing a configuration file that the `ActionServlet` reads when it starts up. A minimal version of this file is shown in Listing 12.5.

Listing 12.5 The struts configuration file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>



<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Config 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config.dtd">



<struts-config>
  <form-beans>
    <!-- Calculator form bean -->
    <form-bean name="calculatorForm"
               type="com.awl.jspbook.ch12.CalculatorForm"/>
  </form-beans>


  <action-mappings>
    <action path="/calculator"
            type="com.awl.jspbook.ch12.CalculatorAction"
            name="calculatorForm"
            scope="request"
            validate="true"
            input="/chapter12/calculator.jsp">
      <forward name="success"
               path="/chapter12/calc_result.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

The first section defines all the form beans the application will use, which here is the `CalculatorForm` defined earlier. It is given a name, `calculatorForm`, which will be used to reference it from JSP pages and elsewhere in the file.

The next section defines the actions the application will perform. The `path` attribute defines the URL for which this action will be taken. Recall that by convention, the `ActionServlet` is configured to handle all URLs ending in `.do`, so this clause of the configuration file indicates that `CalculatorAction` will be invoked when the user accesses the URL /jspbook/chapter12/calculator.do. The `name` parameter indicates the name of the form class to use, which here is the name given to the `CalculatorForm` in the previous section. The `scope` parameter names the scope in which the form bean should be stored. Using the session scope would allow one bean to collect inputs from a number of forms spread across many JSP pages. This is useful for applications that must collect a great deal of information before they can perform their actions. The `validate` flags indicates whether the `validate()` method of the form bean should be called before calling the `perform()` method of the handler. Finally, the `input` parameter indicates the JSP file that contains the form. This value is used if there are any errors validating the form and the user must be sent back to correct them.

The `action` tag may contain any number of `forward` tags that give symbolic names to pages. This example has only one, called `success`, which matches the name used in the

`CalculatorAction`. Using symbolic names like this makes it much easier to modify the way sites behave. If it was ever decided that after successfully computing a sum the calculator should send the user somewhere other than calc_result.jsp, it would simply be necessary to change the configuration file. The Java file would not need to be changed or recompiled.

Here, struts has been configured with two pages: calculator.jsp, which is marked as the input, and calc result.jsp, which is marked as the "success" URL. Struts will determine which of these pages is appropriate, based on the input it has received, and will then use a `RequestDispatcher` to include it. This means that regardless of whether the user is looking at the input or result pages, the URL will be calculator.do. This one URL thus controls access to these two pages, further justifying the use of the term *controller*.

That completes construction of the model and controller, and struts will even simplify the task of completing the view. The input page is shown in Listing 12.6.

Listing 12.6 The input page

```
<%@ taglib prefix="bean"
    uri="http://jakarta.apache.org/struts/bean" %>
<%@ taglib prefix="html"
    uri="http://jakarta.apache.org/struts/html" %>


<html:html>
<head>
  <title>Calculator</title>
  <html:base/>
</head>


<body>


<html:errors/>


<html:form action="/calculator">
```

```
<bean:message key="prompt.number1"/>
<html:text property="number1"/><br>


<bean:message key="prompt.number2"/>
<html:text property="number2"/><br>


<html:submit>
  <bean:message key="button.save"/>
</html:submit>


<html:reset>
  <bean:message key="button.reset"/>
</html:reset>
<html:cancel>
  <bean:message key="button.cancel"/>
</html:cancel>


</html:form>




</body>
</html:html>
```

This example imports two tag libraries from struts, installed as `bean` and `html`. The `bean` tag library supports a number of tags that simplify working with beans, especially for connecting form beans to forms. The `html` tag library provides a number of tags that simplify the creation of html; of particular interest is a set of tags that simplifies the construction of forms and adds some useful functionality.

The first use of these tags is encountered at the top, with the `html:html` tag. This tag doesn't render any output beyond a standard html tag but does set up a context that other struts tags will use internally. This is also true of the `html:base` tag a few lines down, which establishes the current URL from which URLs to the action and result pages can be built.

The `html:errors` tag displays all the messages that have been added to the `ActionErrors` object by the `CalculatorForm` in Listing 12.2. The first time a user accesses this page, it will not yet have been through the `CalculatorForm`, so the `html:errors` will not render any output. If there are errors, `html:errors` will first display the value of the `errors.header` property from Listing 12.1, then each of the errors, then `errors.footer`. This makes it as easy to change the format of the errors as it is to change their text.

The form itself starts a little lower and begins with another new tag, `html:form`. This tag renders as a regular HTML form tag but ensures that the `action` points to the right place. In particular, this tag will ensure that the form gets sent to the `ActionSeverlet` by pointing the URL at jspbook/chapter12/calculator.do. This in turn will allow the servlet to use the name `calculator` to look up the correct form bean and action handler in the configuration file.

A number of `bean:message` and `html:text` tags follow. The `bean:message` tag simply looks up a message in the resource file from Listing 12.1, once again allowing the messages to be configured, changed, or localized. The `html:text` tags render a standard HTML input of type text; in addition, struts can use the name of the provided property and what it knows about the form bean to provide values for these fields as the form is rendered.

This is tremendously useful. Consider what will happen if a user provides a value for the first number but leaves the second one blank. As previously noted, the `validate()` method will fail, and the user will be returned to this input form. The `html:errors` tag will display the appropriate error message informing the user to provide a value for the second number. However, as the user already filled in the first number, it would not be friendly to make the user fill it out again! The `html:text` tag will be able to get the value for the first number back out of the form bean and make it the default value, so the user will not need to reenter it. Conceptually, this is similar to writing

```
<input
  type="text"
  name="number1"
  value="<c:out value="${calculatorForm.number1}"/>">
```

The `html:text` tag hides all the details of which bean and property are used and is therefore much easier to work with. Struts provides similar tags that handle check boxes, text areas, and all the rest. Struts even provides tags to handle form submit and reset buttons, as shown at the bottom of Listing 12.6.

This is made possible by the fact that the JSP is included by the servlet. The servlet sets up the `CalculatorForm` bean and places it in the request, so when validation fails and the `RequestDispatcher` includes calculator.jsp, the bean and hence the user's original inputs are still available.

The only remaining piece of the calculator is the result page, which is shown in Listing 12.7.

Listing 12.7 The result page

```
<%@ taglib prefix="bean"
    uri="http://jakarta.apache.org/struts/bean" %>
<%@ taglib prefix="html"
    uri="http://jakarta.apache.org/struts/html" %>



<html:html>
<head>
  <title>Calculator</title>
  <html:base/>
</head>
<body>



<html:errors/>



<html:form action="/calculator">



<bean:message key="prompt.number1"/>
<html:text property="number1"/><br>
```

```
<bean:message key="prompt.number2"/>
<html:text property="number2"/><br>


<html:submit>
  <bean:message key="button.save"/>
</html:submit>


<html:reset>
  <bean:message key="button.reset"/>
</html:reset>


<html:cancel>
  <bean:message key="button.cancel"/>
</html:cancel>


</html:form>


</body>
</html:html>
```

That's it! The page does not need to load the `Calculator`, which was already done by the `CalculatorAction`. The page is thus reduced to pure view, with no controller elements at all, which is how it should be.

12.2.2 Providing Security

In addition to controlling the interaction between the view and the model, a controller also controls access to the site and individual pages. The general issues of security and protecting pages and resources are complicated, and they cannot be addressed in depth here. Numerous books discuss the topic, and the Java platform itself provides many libraries that handle security issues. For now, however, the discussion is limited to a relatively simple problem: ensuring that only reporters are allowed to create new articles for Java News Today.

Recall that the `user_info` table in Chapter 7 contains a `reporter_ind` that is `Y` if the user is a reporter. This flag, through the `UserInfoBean`, is used to determine whether to show the link to the article creation page. However, if a malicious nonreporter knows to type create_article.jsp into the browser URL window, nothing stops the person from creating as many stories as desired. The goal, then, is to find a way to protect this page so that nonreporters will be unable to access it.

The simplest way of doing this would be to use the `c:if` tag to wrap the sensitive parts of the page:

```
<c:if test="${user.isReporter}">
... contents of the article creation form ...
</c:if>
```

In this case, a nonreporter can still access the page but will not be able to use the form. This approach is a bit unsatisfying, though. If there were many reporter-only pages, it would be necessary to replicate this code on all of them. The closing tag may also be far from the opening tag, which might obscure the page logic to someone trying to maintain the page months after it was written.

This latter problem can be fixed by reversing the test and sending a nonreporter away from the page. This can be done with the `jsp:forward` we saw in Chapter 11:

```
<c:if test="${!user.isReporter}">
  <jsp:forward page="non_reporter.jsp"/>
</c:if>
```

This code could even be made a little smaller by creating a new custom tag that takes a security check and a page to which nonauthorized users should be sent:

```
<awl:secure
  test="${!user.isReporter}"
  page="non_reporter.jsp"/>
```

Although either of these solutions has the advantage of being more self-contained than the first version, it still needs to be added to every page.

Alternatively, struts could be used to protect pages. So far, an "action" has been thought of as the submission of a form, but it is perfectly valid to treat the simple clicking of a link as an action as well and to send it to an action handler. A `ProtectAction` class that would check whether the user in the session is a reporter could be created, and if so, send that person to the "success" URL. Each protected page would then have an entry in the configuration file:

```
<action path="/create_article"
       type="com.awl.jspbook.ch12.ProtectAction"
       scope="request"
       input="/chapter12/non_reporter.jsp">
  <forward name="success"
          path="/chapter12/create_article.jsp"/>
</action>
```

A reporter accessing create_article.do will then get the contents of create_article.jsp and non_reporter.jsp otherwise. This latter page might have some stern words for users trying to hack the system or a kinder message suggesting that the user log in as a reporter or request a reporter account from the editorial staff.

The only problem with this approach, of course, is that it won't work; create_article.jsp will still exist as an independent page. If a user goes there instead of to create_article.do, the check will not be performed.

It is said that every failure carries the seeds of success; this approach has some valuable ideas that would be useful to carry over. Specifically, it is a good idea to keep all security information in a configuration file rather than in each JSP. Further, the security system should sit "in front of" each page; put another way, the security system should get invoked before the user even gets to the page. Both of these facts are yet another restatement of the advantages of using a controller that is separate from the view.

So, we want something that gets access to requests before they reach the JSP and that can be configured from one place. That sounds a lot like the `Filter` class from Chapter 11! A `Filter` can be configured to intercept any set of JSP requests and take all their configuration information from a single file called web.xml, which is discussed in more detail in Appendix B.

Conceptually, the `Filter` that is needed is quite simple. It will be configured with a set of pages that should be protected and a page to which nonauthorized users should be sent. For each request, it then needs to check only whether the request is for one of the protected pages, and if so, whether the user in the session is a reporter. This filter is shown in Listing 12.8.

Listing 12.8 The security filter

```
package com.awl.jspbook.ch12;
```

```
import java.util.StringTokenizer;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.HashMap;


import com.awl.jspbook.ch07.UserInfoBean;


public class ProtectFilter implements Filter {
    private HashMap protectedPages = null;
    private String loginPage = null;


    public void init(FilterConfig conf)
       throws ServletException
    {
       loginPage = conf.getInitParameter("loginPage");
       protectedPages = new HashMap();
       String pages =
          conf.getInitParameter("protectedPages");


       StringTokenizer st =
          new StringTokenizer(pages,",");


       while(st.hasMoreElements()) {
          protectedPages.put(st.nextElement(),
                           Boolean.TRUE);
      }
    }


   public void doFilter(ServletRequest req,
                   ServletResponse res,
                   FilterChain chain)
    throws ServletException,IOException
{
    HttpServletRequest hreq  =
       (HttpServletRequest) req;
    String page             = hreq.getRequestURI();


    if(protectedPages.get(page) == Boolean.TRUE) {
       HttpSession ses = hreq.getSession(true);
       if(ses != null) {
          UserInfoBean inf =
             (UserInfoBean)
               ses.getAttribute("userInfo");
          if(inf != null && inf.getIsReporter()) {
             chain.doFilter(req,res);
             return;
          }

       }

    }


    HttpServletResponse hres =
       (HttpServletResponse) res;

    try {
       hres.sendRedirect(loginPage);
    } catch(IOException e) {}

}
```

```
   public void destroy() {}
}
```

The `doFilter()` method does essentially what was described earlier. The `init()` method sets up the necessary values. The `loginPage` is straightforward and should contain only the name of the page previously referred to as non_reporter. jsp. The `protectedPages` parameter should be a comma-separated list of pages to protect, which the `init()` method will break into individual strings, which get used as keys in the `protectedPages HashMap`, all of whose values are `TRUE`. This common trick is used to check whether a given item is in a set, as checking whether an item is being used as a key in a `HashMap` can be done very quickly and easily, as shown in the `doFilter` method.
It is also worth discussing one other problem that illustrates just how insidious security problems can be. The handler for changing user preferences sets all properties with a call to `<jsp:setProperty property="*";/>`. Normally, this will set all the fields from the form, but what would happen if a knowledgeable but unscrupulous user were to access `user_prefs_handler.jsp?reporterInd=Y` directly? The `jsp:setProperty` tag would set the `reporterId` value to `Y` and would then dutifully call the `setSave()` method. The result would be that the user would have become a reporter! One way to fix this is to list every property explicitly rather than relying on the asterisk version of the `jsp:setProperty` tag or to use the controller to disallow setting the `reporterInd` property. In general, though, the lesson is that securing a Web site is difficult, and a great deal of attention must be paid to the details.

12.2.3 Struts and JNT

At this point, it would be possible to rewrite JNT almost completely by using the struts framework. Action handlers could be introduced at the section and article levels, as an action can consist of clicking a link and submitting a form. A simple handler for the article page is shown in Listing 12.9.

Listing 12.9 The article handler

```
package com.awl.jspbook.ch12;



import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.text.DecimalFormat;
import java.util.Locale;


import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.util.*;


import com.awl.jspbook.ch07.ArticleBean;


public final class ArticleAction extends Action {
    public ActionForward perform(ActionMapping mapping,
                         ActionForm form,
                         HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {

        // Make sure we were given a valid articleId
        String articleIdString =
           request.getParameter("articleId");
        if(articleIdString == null) {
           return mapping.findForward("noSuchArticle");
        }


        Integer articleId = null;

        try {
           articleId = new Integer(articleIdString);
        } catch (NumberFormatException nfe) {
           return mapping.findForward("noSuchArticle");
        }
        // Get the model
        ArticleBean article = new ArticleBean();


        // Load the data
        article.setArticleId(articleId);


        // Make sure that articleId actually exists in the
        // database


        if(article.getHeadline() == null) {
            return mapping.findForward("noSuchArticle");
        }


        // Everything's ok - Store the model in the
        // request so the result page can get to it
        request.setAttribute("article",article);


        return mapping.findForward("success");
    }
}
```

With this handler installed as article.do, the article JSP page could drop the `jsp:useBean` and `jsp:setProperty` tags and simply use the `ArticleBean` that has been placed in the request. The action handler also performs a number of checks to ensure that the requested article exists and can send the user to an error page if it does not.

On the other hand, using a controller in this case requires 54 lines of Java code instead of 2 lines of JSP code. Further, the error conditions for which it checks are things that may not be worth worrying much about. The only two ways in which an `articleId` can be sent to the article page are clicking a link built on the section or index pages or the user's typing one directly into the browser. In the first case, it is ensured that the `articleId` is valid, because the site itself provided it. In the second case, the user is using the site in a way not intended. This should be cause for concern if doing so could in any way damage the system or impact other users; in this case, the worst that could happen is that the user who is misusing the system will get a page with some garbage data or an error message. In the end, it's up to each site developer to weigh the complexity of protecting every page from every possible input against the consequences to the user and the site as a whole if they are not checked.

One place where it clearly does make sense to use a controller is in form handling, notably the login handler. This would work much like the previous examples in this chapter, so it will not be presented here, but the necessary steps should be pretty clear. The index page would be identified as the "success" page, and a new page with only the login form would be identified as the "input." It would be necessary to ensure that a user name and password were provided to the system, as well as that the user exists and has that password. Checking that the fields were filled in could be done by the form bean's `validate()` method, whereas the more semantic checks for user existence and correctness would be done in the model via the `UserInfoBean`.

12.3 Summary and Conclusions

This completes construction of the last of the three pillars on which good Web applications stand. Although the role of the controller may be a little more elusive than that of the model or view, it is certainly no less important. One measure by which to judge how badly a controller is needed is to count the JSP lines that do not directly turn into data for the user. For example, `jsp:useBean` and `jsp:setProperty` tags are important, but the user will never see them directly. If a page has many such tags, it may be an indication that a controller should be loading these beans.

Unlike beans, servlets, and JSPs, struts is not part of a formal specification, and it is certainly possible to build standards-compliant Web applications without it. However, struts is a first-rate tool and is free, so it should be considered when building a controller.

Chapter 13. Creating New Tag Libraries

Since Chapter 4, we have seen that custom tag libraries are an invaluable asset. The time has now come to learn how to create new ones. Fundamentally, tags are not much more complicated than servlets; in fact, servlets could be used to construct a very limited form of custom tag. If it rendered the current time to a page, a servlet could be used almost as a "tag":

```
<jsp:include page="/dateServlet">
  <jsp:param name="format" value="HH:MM:SS"/>
</jsp:include>
```

Alternatively, if tags were implemented as servlets and held to the bean naming conventions, the page compiler could take a simple JSP:

```
Here is the date:
<awl:date format="HH:MM:SS"/>
<p>
```

and turn it into the following code:

```
public void service(HttpServletRequest request,
                  HttpServletResponse response)
{
   response.setStatus(res.SC_OK);
   response.setContentType("text/html");


   PrintWriter out = response.getWriter();
   out.println("Here is the date:");


   DateServlet tag = new DateServlet();
   tag.setFormat("HH:MM:SS");
   tag.service(request,response);


   out.println("<p>");
}
```

Including the contents of a servlet within a page using either of these approaches is not quite enough to do everything that a tag does. However, this concept will serve as a convenient jumping-off point in exploring how tag libraries are constructed.

13.1 The Tag Life Cycle

The first step in being able to write new tags is to understand how pages will use them. Consider a standard usage of a tag, such as the `awl:date` tag from Chapter 4.

```
<awl:date format="HH:MM:SS"/>
```

Clearly, this request must be handled by a class. The name of this class will be associated with the name `awl:date` through a configuration file that will be described shortly. For now, the class is called `com.awl.jspbook.ch04.DateTag` and must implement an interface called `javax.servlet.jsp.tagext.Tag`.

A logical question at this point is whether the lookup of this class should happen at request time or translation time.[1] Doing it at request time would be more dynamic and might allow for some additional functionality, such as changing tag definitions on the fly. However, the introspection mechanisms that allow for this kind of dynamic behavior can be slow, and as tags are so ubiquitous, it is worth doing everything possible to make them fast.

[1] If the terms *translation time* and *request time* are unclear, refer to Chapter 2.

Therefore, the resolution from tag names to class names happens at translation time, and code to build the tag class will be placed in the resulting servlet. Likewise, the tag configuration file can specify all the parameters the tag will accept, so there is no need to look them up dynamically as is done to obtain bean properties. However, if tag classes stick to the bean naming conventions, the page translator will, when it sees a tag attribute called `format`, know to construct a call to `setFormat()` in the `DateTag` class.

In addition to any parameters that the tag accepts, it will need some other information in order to do its job. At the very least, the `DateTag` will need access to `out`, the output stream to which it should send the formatted date. It is reasonable to expect that in general, tags will need access to the full `HttpServeltRequest` and `HttpServletResponse` objects. Both of these objects, as well as a great deal of additional information, is handily contained in the class introduced in Chapter 12. The tag class must therefore provide a `setPageContext()` method to receive this information. Some tags may also need to know whether they have been nested within another tag. The `c:when` and `c:otherwise` tags need a way to access the `c:choose` tag that surrounds them. The `c:choose` tag can keep track of whether a matching condition has been found yet, and each `c:when` tag can then ask the `c:choose` tag whether it should bother to check its test condition. The outer tag is called the *parent*, and so the tag class must have a `setParent()` method.

Next, the tag will need to provide something akin to the servlet `service()` method to do the work. Unlike a servlet, however, a tag consists of two parts: the opening and closing tags. The preceding example has only an opening tag, and a `/>` is used to indicate the absence of a closing tag, but this is really just shorthand for `<awl:time ...></awl:time>`. In general, there may also be body content between these open and close tags. Therefore, rather than having a single `service()` method, tags must provide `doStartTag()` and `doEndTag()` methods.

Finally, once it has completed its task, a tag may need to clean up some resources, as a servlet does in its `destroy()` method. The equivalent for tags is called `release()`.

A few modifications to this basic scheme need to be considered before it will be possible to write `DateTag`. To allow maximum flexibility, a tag may wish to specify whether its body content should be evaluated, an obvious example of which is the `c:if` tag. This is accomplished by allowing `doStartTag()` to return a code indicating how the tag's body should be treated. Possible values are `EVAL_BODY_INCLUDE` and `SKIP_BODY`.

Similarly, `doEndTag()` may decide that the rest of the page should not be evaluated, such as in a custom security tag that wishes to hide the contents of a page from unauthorized users. Therefore, the `doEndTag()` will also return a status code, which may be `EVAL_PAGE` or `SKIP_PAGE`.

Given all this, the page translator will, when it encounters the `awl:date` tag, inject something like Listing 13.1 into the servlet.

Listing 13.1 Tag code generated by the page translator

```
com.awl.jspbook.ch13.DateTag t =
 new com.awl.jspbook.ch13.DateTag();
t.setFormat("HH:MM:SS");
t.setPageContext(... the page context ...);
t.setParent(... the tag's parent ...);
```

```
if(t.doStartTag() == EVAL_BODY) {
   ... code built for the contents of the tag body ...
}


if(t.doEndTag() == EVAL_PAGE) {
   ... code built for the rest of the page ...

}


t.release();
```

The exact code generated will depend on a number of factors. Some JSP engines will attempt to reuse tags when possible to avoid the overhead of the constructor. Some may exit the page immediately if `doEndTag()` returns `SKIP_BODY` rather than wrapping the page in a conditional. Tag authors should not rely on the specifics of the translation; nor should they need to. The exact details of how tags behave is spelled out in the JSP specification, and all JSP engines will adhere to those rules, regardless of the code they generate.

13.2 Tags without Bodies

We now know everything we need to know in order to write a custom tag. Listing 13.2 shows the much-discussed `awl:date` tag:

Listing 13.2 A custom tag

```
package com.awl.jspbook.ch04;


import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;


public class DateTag implements Tag {
    private String format;
    public String getFormat() {return format;}
    public void setFormat(String format)
        {this.format = format;}


    private PageContext pageContext;
    public PageContext getPageContext()
        {return pageContext;}
    public void setPageContext(PageContext pageContext) {
        this.pageContext = pageContext;
    }


    private Tag parent;
    public Tag getParent() {return parent;}
    public void setParent(Tag parent)
        {this.parent = parent;}


    public int doStartTag() throws JspException {
        SimpleDateFormat df = new SimpleDateFormat(format);


        try {
            pageContext.getOut().print(df.format(
                new Date()));
        } catch (IOException e) {}
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() throws JspException {
        return EVAL_PAGE;
    }



    public void release() {
        pageContext = null;
        parent      = null;
    }
}
```

This listing has all the elements that were deemed to be necessary by the preceding discussion. It has `set` methods for the custom `format` attribute, as well as the `pageContext` and `parent`. Corresponding `get` methods for these properties have also been provided in order to make the tag class more beanlike, although in this case, no one is likely ever to use those methods.

The `doStartTag()` uses the `pageContext` to get `out`, to which it sends the formatted date before returning `EVAL_BODY_INCLUDE`. Both of these actions would seem to be correct, but if a page author decides to use a closing `</awl:date>` tag, the date should probably replace the opening tag instead of the closing one, and the body content should be included rather than mysteriously vanishing, as would happen if `SKIP_BODY` were returned.

Although the `doStartTag()` sends only a bit of data to the page, keep in mind that this method can do anything a servlet can do, including accessing beans or setting their properties in the various scopes. This is what makes tags so useful: They expose the full power of servlets in neat little packages easily used from JSPs.

Regardless of whether the page author uses a close tag, nothing is to be done when the tag ends, so `doEndTag()` simply returns `EVAL_PAGE`. Also, no special cleanup needs to be performed in the `release()` method. However, the method sets the `parent` and `pageContext` to `null`, which may allow Java to reclaim the memory allocated to those objects sooner rather than later.

Now that the tag code has been written, it needs to be added to the configuration file mentioned earlier. Two files are involved here. The first, `web.xml`, is used to configure the whole application, including the set of servlets and filters and many other things. More details about this file may be found in Appendix B, but the portion relevant to tag libraries looks like this:

```
<taglib>
  <taglib-uri>
    http://awl.com/jspbook/samples
  </taglib-uri>
  <taglib-location>
    /WEB-INF/taglibs/awl.tld
  </taglib-location>
</taglib>
```

The `taglib-uri` portion specifies the URI that will be used in the `taglib` directive to load the tag library. The `taglib-location` specifies the location of the *tag library description* (TLD) file, which is specified relative to the top-level directory for the Web application. This file contains an entry for each tag named in the class, the attributes, and so on. For a library containing only the `date` tag, the TLD file would contain the following:

```
<?xml version="1.0" ?>


<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.
    //DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">


<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>samples</short-name>
  <uri>http://awl.com/jspbook/samples</uri>
  <display-name>Samples for JSP book</display-name>
  <description>Samples for JSP book</description>


  <tag>
```

```
    <name>date</name>
    <tag-class>com.awl.jspbook.ch04.DateTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>format</name>
      <required>true</required>
    </attribute>
 </tag>
</taglib>
```

The version information at the top specifies the minimal requirements for this tag library. Here, it is indicated that JSP version 2 is required, although this particular tag would work with anything as far back as 1.1. Shortly, however, tags that use the expression language will be introduced, and these tags will require 2.0.

The `display-name`, `short-name`, and `description` convey some information to anyone reading the file but are meant primarily for development environments, such as NetBeans, that provide a rich workspace and tools to simplify the development and testing of JSPs.

The list of tags follows the opening section, which applies to the whole library. Each tag has a `name` that the page will use and a `tag-class` specifying the implementing class. Each attribute that the tag accepts will have an entry; here, there is only one, for the `format` attribute. Attributes may be marked as `required`, in which case the page translator will report an error at translation time if the attribute is missing.

Normally, every attribute that the tag can accept should have an entry in the `attribute` section. If a tag implements the `DynamicAttributes` interface and provides a method called `setDynamicAttribute()`, however, it is possible to send it arbitrary attributes at request time. These dynamic attributes are passed to the tag by using the `jsp:attribute` tag in the body of the tag in question. For each of these attributes, the tag's `setDynamicAttribute()` method will be called with the name of value of the attribute.

Sometimes, it is not sufficient simply to check whether `required` tags are present in each tag usage. Sometimes, a tag will need one of several attributes to be set but will not care which one. A tag that retrieves information about an album might have an attribute to specify the name and another to specify a unique album ID. Neither one of these will be required, but it is required that one or the other be given.

This situation can be handled by creating an auxiliary class that the page translator will use to perform additional checks on the attributes. Such classes extend the `TagExtraInfo` class and perform their checks in methods called `validate()` and `doValidate()`. The page translator is told of the existence of a `TagExtraInfo` class by providing it in the

TLD along with the name of the class, using the `tei-class` tag. `TagExtraInfo` classes are also able to notify the page translator that the tag will be creating new special variables called *scripting variables*, although this technique has been largely superseded by the practice of adding attributes to the `pageContext`, using the `setAttribute()` method.

The use of the `TagExtraInfo` class is beyond the scope of this book, and it is used relatively infrequently. However, readers who explore the TLDs for the standard tag library will see a few references to that class.

13.3 Tags with Bodies

Often, tags will need access to their bodies beyond being able to specify whether to evaluate them. Iteration tags, such as `c:forEach`, will need to be able to evaluate their bodies several times and each time through will need to do some additional processing to manage the array over which it is iterating. The `awl:reverse` tag from Chapter 4 also needs access to the contents of its body so that it can reverse the text before sending it to the page. Neither of these things can be done from classes that simply implement the `Tag` interface.

Therefore, an extension of `Tag`, `BodyTag`, has some additional methods for dealing with bodies. The first of these methods is `doInitBody()`, which is called before the body is evaluated. This method can be used to initialize iteration variables. The `doAfterBody()` method is called after the body has been processed but before `doEndTag()`. This method may return `EVAL_BODY_AGAIN` in order to repeat the body or `SKIP_BODY` to end processing. Finally, `BodyTag` provides the `setBodyContent()` method, which is passed a `BodyContent` object containing the contents. `BodyTag` also introduces a new value, `EVAL_BODY_BUFFERED`, which `doStartTag()` may return to indicate that the tag will want to intercept the contents of the body for processing.

Because of the numerous methods often many of them doing standard things to write

when building a `BodyTag`, a convenience class, `BodyTagSupport`, is provided that by default runs through the body once and simply sends the contents of the body directly to the page. The common approach to building body tags is to extend this class rather than to implement `BodyTag` directly. This is what the `ReverseTag` from Chapter 4 does, as shown in Listing 13.3.

Listing 13.3 A tag with a body

```
package com.awl.jspbook.ch04;



import java.io.IOException;
import java.io.StringWriter;
import java.text.SimpleDateFormat;
import java.util.Date;



import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;



public class ReverseTag extends BodyTagSupport {
   private PageContext pageContext;
   public PageContext getPageContext()
      {return pageContext;}
   public void setPageContext(PageContext pageContext) {
      this.pageContext = pageContext;
   }



   private Tag parent;
   public Tag getParent() {return parent;}
   public void setParent(Tag parent)
      {this.parent = parent;}



   public int doStartTag() throws JspException {
      return EVAL_BODY_BUFFERED;
   }
```

```
public int doEndTag() throws JspException {
    String output = "";


    if(bodyContent != null) {
        StringWriter sw = new StringWriter();
        try {
            bodyContent.writeOut(sw);
            output = sw.toString();
        } catch(java.io.IOException e) {}
    }


    output = doReverse(output);


    try {
        pageContext.getOut().print(output);
    } catch(java.io.IOException e) {}


    return EVAL_PAGE;
}


public void release() {
    pageContext = null;
    parent      = null;
}

private String doReverse(String output) {
    int len     = output.length();
    char out2[]  = new char[len];
    for(int i=0;i<len;i++) {
        out2[i] = output.charAt(len-1-i);
        }

        return new String(out2);
    }
}
```

This listing has many of the same methods as the previous example, which is to be expected, as body tags will need to be initialized with a `pageContext` and `parent`, just as any other tag. The first difference is that `doStartTag()` returns `EVAL_BODY_BUFFERED`, indicating that it will be manipulating the contents of the body. This manipulation is accomplished in the `doEndTag()` method, which first checks whether the body content is available by checking whether the special variable `bodyContent` is `null`. This variable is defined in the base class.

If the `bodyContent` is available, the tag obtains the contents by writing them to a `StringWriter`, using the `writeOut()` method. This happens *after* the body has been processed, so if the body has any `c:out` or other JSP tags, as well as any other JSP tags, they will already have been replaced by the specified values.

Once the content of the body has been obtained, reversing it is fairly simple and is done by the `doReverse()` method in this class. Sending the contents to the page is then as simple as writing it to `out`, just as was done for the date in Listing 13.2.

It is now possible to clarify exactly what `out` is. It is an instance of another class, called `JspWriter` that at the top level will send any data written to it on to the user. However, within body tags, the `JspWriter` will store, or *buffer*, the data that is written to it, so that the `bodyContent` object can pass this data to the body tag. So in this case, if the `awl:reverse` tag is being used from within another body tag, `out` will hold onto the reversed text and pass it along to the other tag. This is all managed transparently by the JSP engine, so tag authors almost never need to worry about what exactly `out` is when they are writing data to it.

13.4 Using the Expression Language

The JSP engine does not automatically handle attribute values that use the expression language. If a page were to call `awl:date` with a value of `${param. myFormat}` as the value of format, the `DateFormat` object would attempt to use the literal string `${param.myFormat}` when formatting the number and would not automatically look up the value of the parameter called `myFormat`.

It is the tag's responsibility to interpret any expression language variables that it wishes to make dynamic. Fortunately, a few classes make this much easier. The primary one is `ExpressionEvaluator`, which does the evaluation. `ExpressionEvaluator` uses a couple of additional classes: `VariableResolver`, which is responsible for looking up the values of any variables used within an expression, and `FunctionMapper`, which can handle more complex kinds of expressions involving calls to functions. The functionality is split into these pieces in order to make it easier to customize their behavior. It would be possible for a programmer to replace the default `VariableResolver` with one that obtained values from a database. It would even be possible to create an `ExpressionEvaluator` that handled a different kind of expression language entirely, making it possible to write expressions in other languages, such as Scheme or Perl, if such a thing were ever desired.

Leaving aside such exotic thoughts, let's look at the basic use of these classes to enable dynamic attributes. Listing 13.4 shows the class that implements the `awl:maybeShow` tag from Chapter 4. Recall that this tag has one attribute, `show`, which may be `yes`, `no`, or `reverse`.

Listing 13.4 A tag that uses the expression language

```
package com.awl.jspbook.ch04;



import java.io.IOException;
import java.io.StringWriter;
import java.text.SimpleDateFormat;
import java.util.Date;



import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.el.*;


public class MaybeShowTag extends BodyTagSupport {
    private String show;
    public String getShow() {return show;}
    public void setShow(String show) {this.show = show;}

    private PageContext pageContext;
    public PageContext getPageContext() {
        return pageContext;
    }
    public void setPageContext(PageContext pageContext) {
        this.pageContext = pageContext;
    }


    private Tag parent;
    public Tag getParent() {return parent;}
    public void setParent(Tag parent) {
        this.parent = parent;
    }


    public void release() {
        pageContext = null;
        parent      = null;
    }


    public int doStartTag() throws JspException {
        return EVAL_BODY_BUFFERED;
    }
```

```
public int doEndTag() throws JspException {
   // If we've been through the body, grab
   // the contents


   String output = "";

   if(bodyContent != null) {
      StringWriter sw = new StringWriter();
      try {
         bodyContent.writeOut(sw);
         output = sw.toString();
      } catch(java.io.IOException e) {}
   }


   // Resolve the actual command by assuming
   // the provided value is a script in the

   // expression language
   ExpressionEvaluator ee =
      pageContext.getExpressionEvaluator();
   VariableResolver vr    =
      pageContext.getVariableResolver();


   // No default function mapper is provided or
   // needed
   FunctionMapper fm     = null;
   Expression expr       = null;

   try {
      expr = ee.parseExpression(show,
                               String.class,
                               fm,
                                 null);
        } catch (ELParseException e) {
            throw new JspException(
              "Unable to parse expression for show");
        } catch (ELException e2) {
            throw new JspException(
              "Unable to evaluate expression for show");
        }


        try {
            show = (String) expr.evaluate(vr);
        } catch (ELException e) {
            throw new JspException(
              "Unable to evaluate expression for show");
        }


        if("reverse".equals(show)) {
            output = doReverse(output);
        } else if("no".equals(show)) {
            output = "";
        }


        try {
            pageContext.getOut().print(output);
        } catch(java.io.IOException e) {}

        return EVAL_PAGE;
    }


    private String doReverse(String output) {
        int len     = output.length();
        char out2[] = new char[len];
        for(int i=0;i<len;i++) {
            out2[i] = output.charAt(len-1-i);
        }
        return new String(out2);
    }
}
```

To a large extent, this looks like the tags already discussed in this chapter; in particular, nothing is special about the `show` attribute. The magic happens in `doStartTag()`, which first uses the `pageContext` in order to obtain the default `ExpressionEvaluator` and `VariableResolver`. The `doStartTag()` then uses these objects to build an `Expression` object, which is an internal representation of the expression that can be evaluated quickly and efficiently. The call to `parseExpression()` also takes a class as an argument, in this case `String.class`, which represents the type of object that should be returned from the expression.

If the expression parses correctly, it will then be evaluated by passing the `VariableResolver` to the `evaluate()` method. If all goes well, this will return the result of the expression. To be rigorous, the code should now check the value to ensure that it is one of the three acceptable possibilities, but in the interest of keeping the code simple, this check has been omitted. The use of the value itself is pretty straightforward; it is simply, used to decide whether to leave the output as is, reverse it, or delete it.

13.5 JSPs as Custom Tags

As of version 2.0, it is possible to create new tags in JSP, as well as in Java. This is supported by a few new tags, such as `jsp:doBody`, and some new directives, such as `variable`. The idea is straightforward, as the page translator can turn a JSP into a servlet, which is a Java file, as well as turn a JSP into tag, which is another kind of Java file. The details are beyond the scope of this book, but interested readers can find all the details starting in Section 8.4 of the JSP 2.0 specification.

13.6 Summary and Conclusions

In Chapter 2, we were using JSP comments simply to remove chunks of text from a page. We've now reached the point at which new tags can be created, putting the full power of the Java language and libraries into simple boxes that are no more difficult to use than old-fashioned HTML tags.

Of course, writing tags can be a complicated business, and it would not be possible to cover all the intricacies in one chapter. The material here, in conjunction with everything that was said about servlets in Chapter 11, should allow the creation of rich and varied tags that meet most needs. Users interested in more of the details should read Sections 12 and 13 in the JSP 2.0 specification. Fortunately, a large collection of sample tags is available to study: the standard tag library itself. Thanks to the wonders of open-source development, all the code for the tags in the standard library is available for download from the Jakarta site at http://jakarta.apache.org/builds/jakartataglibs/releases/standard/src/. The sheer number of files may seem overwhelming, but looking through some of the classes, such as the one that implements org.apache.taglibs.standard.tag.el.core.OutTag, will quickly reveal some familiar things from this chapter.

Chapter 14. Advanced Topics

The preceding chapters have included more than enough information to build almost any conceivable Web site by using JSPs, beans, databases, and servlets. In fact, however, a great deal in the JSP, servlet, JDBC, and bean specifications could not possibly be included here. This book has concentrated on those features that are most common: the ones that will be used 90 percent of the time. This chapter surveys a few remaining topics that should cover another 5 percent.

14.1 Declaring Variables and Methods

As JSPs are servlets and class variables and new methods can be added to servlets, it should be possible to do the same to a JSP. This can be done by placing the declaration between `<jsp:delcare>` and `</jsp:declare>` tags or, equivalently, by placing the declaration between `<%!` and `%>`. Compare this to the `jsp:scriplet` tag seen in Chapter 9. That tag causes the page translator to embed code within the JSP equivalent of the `service()` method; `jsp:declare` causes code to be embedded in the generated Java class. In other words, the declaration tags can define new methods as well as variables. Listing 14.1 shows a JSP page that computes the the *n*th prime number, using a declared method.

Listing 14.1 A JSP with a user-defined method

```
<%!

  public int primes(int n) {

    if(n < 2) return 2;
    if(n == 2) return 3;

    int primes[] = new int[n];
    primes[0]    = 2;
    primes[1]    = 3;
    int candidate = 5;
    int numSoFar  = 2;
    boolean maybePrime;

    while (numSoFar < n) {
      maybePrime = true;
      for(int i=0;i<numSoFar && maybePrime;i++) {
        maybePrime = (candidate % primes[i]) != 0;
      }

      if(maybePrime) {
        primes[numSoFar++] = candidate;
      }
      candidate++;
    }

    return primes[n-1];
  }

%>

<HTML>
<HEAD><TITLE>Primes</TITLE></HEAD>

<BODY>

<P>Here are the first 5 prime numbers:</P>

<UL>
<LI><%= primes(1) %>
<LI><%= primes(2) %>
<LI><%= primes(3) %>
<LI><%= primes(4) %>
<LI><%= primes(5) %>
</UL>
```

```
</BODY>
</HTML>
```

This example uses another tag to display the values. This expression tag may be written `<%= ... %>`, as was done here or, equivalently, as `<jsp:expression> ... </jsp:expression>`. In either case, the result is identical to writing `<% out.print(...) %>`, with the dot replaced by the expression.

The `primes()` method itself computes primes in a simple way. It starts with the first two primes and computes each prime after that by checking every number against the list of primes it has already computed. When checking 9, it will first check 9/2, which will not divide evenly. It then checks 9/3, which will divide evenly, ruling 9 out as a prime number.

Although this works, it has several inefficiencies. Every time the method is called, it will recompute the whole array to get to the number it wants, even if it has already computed most or all of that array. Listing 14.1 will compute the first four primes when asked to evaluate `primes(4)`, and it will then recompute all four in the next step, when it asked for the fifth.

The solution is to take the `primes` array out of the method and into a separate field in the class. Then each time it is asked for a prime number, the method can check the list it has already built and return the number if it has already been computed. If not, the method will then need to compute only the values between the last one it found and the one for which it has just been asked.

It will also make sense to start the array with more than two values, giving the method a bit more of a jump start. The best place to do this is when the JSP is first loaded, which can be done by placing the initialization code in a `jspInit()` method. Although this method will be treated specially by the JSP engine, it can be declared just like any other method, as shown in Listing 14.2.

Listing 14.2 A JSP with a `jspInit()` method

```
<%!
  int primes[];
%>

<%!
  public void jspInit() {
    /* Pre-populate the first 100 primes */
    primes(100);
  }
%>

<%!
  public int primes(int n) {
    if(primes != null && n < primes.length) {
      return primes[n-1];
    }

    int oldPrimes[] = primes;
    primes         = new int[n+1];

    int candidate;
    int numSoFar;

    if(oldPrimes != null) {
      System.arraycopy(oldPrimes,0,
                       primes,0,oldPrimes.length-1);
      candidate = oldPrimes[oldPrimes.length-1];
      numSoFar  = oldPrimes.length;
    } else {
      primes[0] = 2;
      candidate = 3;
      numSoFar  = 1;
    }

    boolean maybePrime;

    while (numSoFar < n) {
      maybePrime = true;
      for(int i=0;i<numSoFar && maybePrime;i++) {
        maybePrime = (candidate % primes[i]) != 0;
      }
      if(maybePrime) {
        primes[numSoFar++] = candidate;
      }
      candidate++;
    }

    return primes[n-1];
  }
%>

<HTML>
<HEAD><TITLE>Primes</TITLE></HEAD>

<BODY>

<P>Here are the first 5 prime numbers:</P>

<UL>
<LI><%= primes(1) %>
<LI><%= primes(2) %>
<LI><%= primes(3) %>
<LI><%= primes(4) %>
<LI><%= primes(5) %>
</UL>

</BODY>
</HTML>
```

14.2 Extending Different Classes

Considering how much Java code is in Listings 14.1 and 14.2, it might as well be a servlet. But if it were, it would have the same old problem of being difficult to change the appearance or other aspects, such as the number of primes to generate. The code could also be placed in a bean or custom tag, which might initialize the array in its constructor.

This is normally the recommended approach, but another alternative may be preferable in some instances.

In Chapter 11, it was mentioned that all JSPs implement the `HttpJspPage` interface. Tomcat does this by making JSPs extend the `HttpJspBase` class, which in turn implements `HttpJspPage`. In principle, a JSP could extend a different class, so long as that class also implemented `HttpJspPage`. This class could define the `primes()` and `jspInit()` methods. The JSP engine will still call `jspInit()` when it loads the JSP, and the `primes()` method will then be available to the page without the need to define any code in the page itself. Listing 14.3 shows the class containing the prime code.

Listing 14.3 A base class with the prime methods

```
package com.awl.jspbook.ch14;

import org.apache.jasper.runtime.*;

public abstract class Primes extends HttpJspBase {
  int primes[];

  public void jspInit() {
    /* Pre-populate the first 100 primes */
    primes(100);
  }

  /**
   * We don't need to do anything when the JSP
   * is destroyed, but we still need to provide
   * this method to satisfy the interface.
   */
  public void jspDestroy() {
    return;
  }

  public int primes(int n) {
    if(primes != null && n < primes.length) {
      return primes[n-1];
    }

    int oldPrimes[] = primes;
    primes         = new int[n];

    int candidate;
    int numSoFar;

    if(oldPrimes != null) {
      System.arraycopy(oldPrimes,0,primes,0,
      oldPrimes.length-1);
      candidate = oldPrimes[oldPrimes.length-1];
      numSoFar  = oldPrimes.length;
    } else {
      primes[0] = 2;
      candidate = 3;
      numSoFar  = 1;
    }

    boolean maybePrime;

    while (numSoFar < n) {
      maybePrime = true;
      for(int i=0;i<numSoFar && maybePrime;i++) {
        maybePrime = (candidate % primes[i]) != 0;
      }

      if(maybePrime) {
        primes[numSoFar++] = candidate;
      }
      candidate++;
    }

    return primes[n-1];
  }
}
```

Once this class has been defined, using it is quite simple, as shown in Listing 14.4.

Listing 14.4 A JSP that extends a different base class

```
<%@ page extends="com.awl.jspbook.ch14.Primes" %>

<HTML>
<HEAD><TITLE>Primes</TITLE></HEAD>

<BODY>

<P>Here are the first 5 prime numbers:</P>

<UL>
<LI><%= primes(1) %>
<LI><%= primes(2) %>
<LI><%= primes(3) %>
<LI><%= primes(4) %>
<LI><%= primes(5) %>
</UL>

</BODY>
</HTML>
```

The JSP is told to use a different base class with another use of the `page` directive. Apart from this directive, the rest of the page is straightforward and much cleaner than the previous versions.

When faced with the need to add some functionality to a JSP, four choices are now available: Use a bean, use a new tag, define the methods in the JSP, or put the methods in a separate class. Putting the code in the JSP is ugly and cumbersome, which leaves the other three possibilities. The decisions will almost always fall on the side of beans or custom tags. The JSP specification states, in section JSP.1.10: "[The extends attribute] should not be used without careful consideration as it restricts the ability of the JSP engine to provide specialized superclasses that may improve on the quality of rendered service."

14.3 Returning Other Kinds of Data

Several examples in Chapter 8 use the `page` directive to change the content type, and Chapter 11 notes that this is accomplished by calling the `setContentType()` method in the `HttpServletResponse` class. One of the exciting possibilities that this ability offers is for a JSP or servlet to generate binary data, such as an image, as well as various kinds of text. This will at long last make it possible to fix the one remaining problem with the Java News Today site. Recall that users are permitted to change the color of the top and side navigation areas by setting a value in a style sheet. However, a rounded corner is used between these two pieces in order to give the page a smoother apparence, and this rounded corner is an image that cannot be controlled by a style sheet.
Manipulating binary data would be difficult to do directly in a JSP, so a bean will be used to do the data preparation. Listing 14.5 shows a bean that generates the data for a GIF file containing the corner.

Listing 14.5 A bean that generates GIF data

```
package com.awl.jspbook.ch14;

import java.io.*;

public class CornerBean {
  private final static byte cornerBytes[] = {
      (byte)0x49, (byte)0x47, (byte)0x38, (byte)0x46,
      (byte)0x61, (byte)0x37, (byte)0x00, (byte)0x14,
      (byte)0x00, (byte)0x14, (byte)0x00, (byte)0x80,
      (byte)0x66, (byte)0x00, (byte)0xff, (byte)0xff,
      (byte)0xff, (byte)0xff, (byte)0x2c, (byte)0xff,
      (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
      (byte)0x00, (byte)0x14, (byte)0x00, (byte)0x14,
      (byte)0x02, (byte)0x00, (byte)0x84, (byte)0x27,
      (byte)0x69, (byte)0x8f, (byte)0xea, (byte)0xc1,
      (byte)0x9b, (byte)0x0c, (byte)0x31, (byte)0x43,
      (byte)0x55, (byte)0xce, (byte)0xcb, (byte)0xed,
      (byte)0x73, (byte)0x75, (byte)0x7d, (byte)0x4d,
      (byte)0x28, (byte)0x5d, (byte)0x40, (byte)0x82,
      (byte)0x1d, (byte)0x99, (byte)0x8a, (byte)0x68,
      (byte)0x50, (byte)0xad, (byte)0x4a, (byte)0xeb,
      (byte)0x35, (byte)0xb1, (byte)0x75, (byte)0xd3,
      (byte)0xe3, (byte)0x73, (byte)0xce, (byte)0xf9,
      (byte)0x05, (byte)0xf7,(byte)0x3b, (byte)0x00};

    private String fgColor = "7f7f7f";
    private String bgColor = "ffffff";

    public String getFgColor() {return fgColor;}
    public String getBgColor() {return bgColor;}

    public void setFgColor(String fgColor) {
        this.fgColor = fgColor;
        byte tmp[] = toHex(fgColor);
        cornerBytes[13] = (byte) (tmp[0] * 16 + tmp[1]);
        cornerBytes[14] = (byte) (tmp[2] * 16 + tmp[3]);
        cornerBytes[15] = (byte) (tmp[4] * 16 + tmp[5]);
    }

    public void setBgColor(String bgColor) {
        this.bgColor = bgColor;
        byte tmp[] = toHex(bgColor);

        cornerBytes[16] = (byte) (tmp[0] * 16 + tmp[1]);
        cornerBytes[17] = (byte) (tmp[2] * 16 + tmp[3]);
        cornerBytes[18] = (byte) (tmp[4] * 16 + tmp[5]);
    }

    public String getCorner() {
        return new String(cornerBytes);
    }

    public byte[] toHex(String s) {
        byte tmp[] = s.toUpperCase().getBytes();
```

```
        for(int i=0;i<tmp.length;i++) {
            if(tmp[i] >= 'A' && tmp[i] <= 'F') {
                tmp[i] = (byte) (tmp[i] - 'A' + 10);
            } else {
                tmp[i] = (byte) (tmp[i] - '0');
            }
        }

        return tmp;
    }
}
```

The GIF data, with a gray foreground and white background, is held in the `cornerBytes` array. GIFs store their colors in a well-defined location: the *colormap*. The `setFgColor()` and `setBgColor()` methods change the values in this colormap, and the `getCorner` method simply returns the data as a new string. This bean can now be used in a JSP, as shown in Listing 14.6.

Listing 14.6 A JSP page that generates a GIF

```
<%@
taglib
prefix="c"
uri="http://java.sun.com/jstl/core" %><%@
page
contentType="image/gif" %><jsp:useBean
id="corner"
class="com.awl.jspbook.ch14.CornerBean"/><jsp:useBean
id="user7"
class="com.awl.jspbook.ch07.UserInfoBean"
scope="session"/><jsp:setProperty
name="corner"
property="fgColor"
property="${user7.bannerColor}"/><jsp:getProperty
name="corner" property="corner"/>
```

The formatting of this example is a little strange, as all the tags are directly adjacent and all the line breaks are inside the tags. This was done to ensure that no whitespace shows up intermixed with the image data, which would cause a browser to be unable to render it. For this and numerous other reasons, this kind of thing is unlikely ever to be done in the real world. Servlets are much better at manipulating binary data than JSPs, even with the help of beans. However, this does show how a JSP can generate things other than text.
Nothing new is in the code itself. The content type is set through the `page` directive, the bean is loaded, the user's chosen `bannerColor` property is assigned to the corner's `fgColor` property with a `jsp:setProperty` tag, and the `corner` property is then obtained.

14.4 Threads

Threads, an integral and powerful feature of Java, allow a single program to do many things simultaneously. An obvious example is a Web server that is written in Java and that may handle hundreds of user requests at the same time. The earliest Web servers handled multiple requests by essentially creating a copy of themselves for each, which can be slow and use a lot of memory. Under Java, it is necessary only to start a new thread, and it will use the same code and same memory as all the other threads.
To describe what threads are, consider the way someone might have read this book. A person might have started at page 1 and read straight through to this point. Alternatively, the reader may have skipped around a bit, perhaps checking Chapter 9 to read more about a particular Java construct. In any case, each reader defines his or her own path through the book.
Now consider two or more people reading this book simultaneously. In real-world terms, this might mean that the pages would need to be torn out and passed around; conceptually, however, several people could be reading at the same time. Each will define a path through the material, based on personal interests and familiarity with some of the topics. Sometimes, two or more people might find themselves reading the same words at the same time; at other points, everyone will be reading at different places.
Java threads work much like this, except that they are reading Java instructions instead of words. A new reader can start by specifying the chapter at which to start. A new thread can be created at any time and given a method to start with, usually the `run()` method of

a class that implements the `Runnable` interface. Once a thread has started, it may take a different path through the code, based on input from users, time of day, contents of a database, or anything else that can be expressed in a conditional.

So far, there is no problem; multiple readers can go through a book without interfering with one another, and multiple threads can move through a Java program without ever knowing that another thread exists. However, consider what would happen if this book had a quiz at the end of each chapter. One reader might start working on a quiz, starting with the first question, and another reader might start the same quiz a minute later, lagging behind the first reader by a few questions. By the time the first reader finished, most of the answers would have been over-written by the second reader, and the score for the quiz would be a meaningless combination.

Threads have an analogous problem, which can be demonstrated by the simple JSP shown in Listing 14.7.

Listing 14.7 A JSP with a potential thread problem

```
<%! String machineName; %>

<% machineName = request.getRemoteHost(); %>

<HTML>
<BODY>

<P> You are using a computer called <%= machineName %>.</P>

</BODY>
</HTML>
```

In this example, `machineName` is an instance variable, meaning that there will be one shared among all the users of this JSP. Now consider what would happen if two users, Daria and Jane, access this page more or less simultaneously. For the sake of discussion, assume that they are using machines called orwell and van_gogh, respectively.

If Daria's request is received first, `machineName` will be set to Orwell. If Jane's request is then received before Daria's thread gets to the expression, `machineName` will then be set to van_gogh. Then when Daria's request gets to the expression, the JSP will state that she is using van_gogh, which is incorrect.

The chances of this happening are pretty slim if a page is small and simple or if it is accessed infrequently. However, as a page gets more complex and takes longer to generate, or as it is used by more people, potential thread problems become much more likely.

The JSP specification provides an easy way to avoid thread problems, but it is not without its costs. A JSP can declare that it is not *thread safe*, meaning that is is not able to handle multiple threads simultaneously. This can be done by using another variant of the `page` directive; simply add the following line at the top of the JSP:

```
<%@ page isThreadSafe="false"%>
```

The same thing can be done in a servlet by having it implement the `javax.servlet.SingleThreadModel` interface. When it sees that a JSP or servlet is not thread safe, the JSP engine will force all requests to go through sequentially. This means that if Daria gets to the page first, either Jane will have to wait until Daria has the full response back, or the Web server will need to create a second instance of the servlet for Jane.

This does indeed avoid the thread problem, but the result is that either users may have to wait for their turn, which will make the site seem slower, or multiple versions of servlets will need to be created, using up memory. This may eventually cause users to give up on a site in frustration, which is not acceptable for a site that wishes to build and hold an audience. However, the single-threading technique is useful for tracking down problems. If a page or a whole site is exhibiting strange bugs that appear irregularly and are impossible to recreate or track down, it may be worth making all pages single threaded for a while. If the problems go away, it is a safe bet that some threading issue is the cause.

14.4.1 Avoiding Thread Problems

In the general case, avoiding any thread problems may be quite difficult and is a science unto itself. *Concurrent Programming in Java™ Second Edition: Design Principles and Patterns* by Doug Lea (Addison-Wesley, 2000) provides much more information for those who want to understand the issues fully. Fortunately, it is not difficult to avoid the most common kinds of thread problems in a JSP. The first step is knowing what may potentially cause problems.

As Listing 14.7 showed, instance variables can definitely cause bugs, but local variables cannot. When a thread calls a method, a private copy of all that method's variables is created, so each thread is working with its own copy. The problem would vanish if the declaration in Listing 14.7 were replaced with a scriptlet creating a local variable:

```
<% String machineName; %>
```

As each request and response object are also private to each thread, the problem would never have arisen if Listing 14.7 had skipped the intermediate variable altogether and just called

```
<%= request.getRemoteHost() %>
```

This also applies to objects in the request scope. As each thread has its own request, each request scope is separate from all others, which means that beans or other objects placed in this scope will not normally be available to any other thread. Of course, if an object is already available to multiple threads, simply placing it in the request scope will not protect it. If several threads were all to put the `machineName` instance variable into their own request scopes, it will still be the same variable, and any change made by one thread would still be visible to all others.

Objects in the session scope are also safe in general, as only one user, and hence one thread, will typically be accessing a given session at any moment. This, after all, is the whole point of the session scope. It is possible for a user to open multiple browser windows and hit different pages simultaneously, which can in principle cause problems. Normally, this should not be a concern. The same is true of the page scope.

That leaves the application scope, which is clearly not thread safe. Anything in the application scope may be used by several pages and several users simultaneously. This offers a powerful mechanism for sharing data across pages, but it also means that thread safety may need to be considered carefully.

Usually, the application scope will contain beans created from pages by the `jsp:useBean` tag. The issue of thread safety then moves into the Java code within the bean. The easiest way to ensure that a bean is thread safe is for each of its methods to be written as

```
public int someMethod(...) {
    synchronize(this) {
        ... method code ...
    }
}
```

The call to `synchronize()` puts a lock in place, which ensures that only one thread at a time will be able to execute any method in that bean. This goes somewhat back to the situation when JSP is declared not thread safe, but it is much more granular. If Daria and Jane are accessing pages using the same bean in an application scope, one of them will have to wait for the other only if they both happen to call a method in that bean at the same time. If they are in the section of the page that uses the bean at different times, they will both be able to continue using the other portions of the page without needing to wait for each other.

Typically, in fact, not all the methods of a bean will need to be synchronized, which will further decrease the chances that any user will have to wait for another. This is where the science of threading comes in, and interested readers are referred to *Concurrent Programming in Java™ Second Edition: Design Principles and Patterns* for the details.

14.4.2 Using Threads

So far, they have appeared only as potential sources of bugs, but threads can be powerful allies as well. Any time a user's request requires an action on the server side but the user does not need to wait for the action to be completed, the action can be handled by creating a new thread. Examples are such things as placing an order at an e-commerce site. The user doesn't have to wait for the order to reach the shipping center; it is enough for the user to know that the order has been entered into the system. Once that has been done, the user can be shown a page indicating that the order is being processed, and a separate thread that will handle the back-end processing, including contacting the shipping center can start.

Threads can also be used to ensure that data is updated or that data in memory and in a database is synchronized. For example, recall the bean that contained advertising information for Java News Today, as used in Chapter 7. This bean would load data from the database when it was first constructed. This data would include the number of impressions the advertiser had purchased and the number delivered so far, and each time the ad was shown to a user, the count would be incremented. However, it would be extremely inefficient to have this bean update the count in the database each time an ad was shown, as that would mean a constant stream of writes to the database, which would slow down the whole site.

A better solution is to have the ad bean keep the counts in memory, and once every 10 minutes or so update all the counts in the database. The outline of the code that does this is shown in Listing 14.8; the full code is included on the CD-ROM.

Listing 14.8 A bean that periodically saves itself to a database

```
package com.awl.jspbook.ch07;

import java.sql.*;
import com.canetoad.util.PersistentConnection;

public class AdManagerBean implements Runnable {
    public final static Integer ZERO = new Integer(0);

    private AdBean ads[];
    private Thread runner;

    public AdManagerBean() {
        // Load all the ads in the system!
        AdBean tmp = new AdBean();
        ads = tmp.getBeans();

        // Start a thread to keep things synchronized with
        // the database
        runner = new Thread(this);
    }

    public void run() {
        while(runner != null) {
            try {
                // sleep 10 minutes
                Thread.sleep(1000 * 60 * 10);
                // Now update the database by subtracting
                // the number of impressions seen from the
                // number sold
                Connection tmp =
                    PersistentConnection.getConnection();
                Statement s    =
                    tmp.createStatement();

                for(int i=0;i<ads.length;i++) {
                    if(ZERO.equals(
                            ads[i].getImpressions()))
                    {
                        s.executeUpdate(
                "update ad set impressions = impressions-" +
                ads[i].getImpressions() +
                " where ad_id = " +
                ads[i].getAdId());

                        // If the impression count has
                        // reached 0, we should remove it
                        // from the system, but we'll
                        // assume that happens off-line
                        // as part of some nightly
                        // processing

                        // Reset the counter
                        ads[i].setImpressions(ZERO);
                    }
                }

                s.close();
            } catch (Exception e) {
                System.err.println(
                    "Unable to decrement ad counter");
                e.printStackTrace(System.err);
            }
        }
    }
}
```

When the bean is constructed, it will create a new thread, which it then starts. The `Thread` class will call the `run()` method of the object it is created with, which in this case is the same object. The `run()` method simply sleeps for 10 minutes and then calls the bean's `update()` method, which saves any new data to the database.

14.5 Advanced Error Handling

Chapter 2 mentioned in passing that it is possible to send users to a custom JSP when an error occurs. This is a two-step process. First, the page that may generate the error should specify the destination with one use of the `page` directive:

```
<%@ page errorPage="error.jsp" %>
```

Second, the page that is to act as the error handler should declare this fact with another use of the `page` directive:

```
<%@ page isErrorPage="true" %>
```

The error page can do anything that any other JSP can do, but in addition it will probably want to report the error to the user and/or a site administrator. To make this possible, the error itself is contained in an `Exception` object, as described in Chapter 9, and this object is called, not surprisingly, `exception`. Listing 14.9 uses this variable to report the error to the user, using the JNT style instead of the default Tomcat one.

Listing 14.9 A simple error page

```
<%@ page isErrorPage="true" %>

<jsp:include page="top.jsp" flush="true">
  <jsp:param name="title" value="Error"/>
</jsp:include>

<% exception.printStackTrace(out); %>

<jsp:include page="bottom.jsp" flush="true"/>
```

The details of the error are unlikely to be of use to the end user, so it is more likely that an error page will simply offer an apology to the user and behind the scenes report it to the administrator. Listing 14.10 writes the error to a file called "errors."

Listing 14.10 A slightly more sophisticated error page

```
<%@ page isErrorPage="true"
        import="java.io.*" %>

<jsp:include page="top.jsp" flush="true">
  <jsp:param name="title" value="Error"/>
</jsp:include>
```

```
We're sorry, but an error occurred while building
your page. We will try to fix this problem shortly;
in the meantime please return to the
<a href="index.jsp">JNT home page</a>

<%
FileWriter w = new FileWriter("errors",true);
PrintWriter fileOut = new PrintWriter(w);

exception.printStackTrace(fileOut);
w.close();
%>

<jsp:include page="bottom.jsp" flush="true"/>
```

This example uses another new feature of the `page` directive, `import`, which makes a package available to a JSP. Here, it is used to import the `java.io` package, which contains classes to manage files and printing.

In fact, writing an error to a file like this is not terribly productive, as all errors are already saved in the Tomcat error logs. However, it is possible to use this basic technique to do more elaborate things, such as send an e-mail alert to the site maintainer every time an error crops up. A standard Java extension, the Java Mail API, would handle all the hard work in this case.

14.6 Summary and Conclusions

The topics covered in this chapter will not be everyday concerns but are included here on the theory that no knowledge is ever wasted. At times, it may be easier for a JSP to define a utility method than to use a bean, and this method might be used so often that it makes sense to put it in a base class. Sooner or later, the highest-volume sites will have to start worrying about thread issues, and this chapter has enough information to avoid most of the common problems that may be encountered in a multithreaded environment.

That concludes our introduction to the wonderful world of JavaServer Pages, but perhaps it is just the beginning of your use of this exciting and powerful technology. The

CD-ROM contains the complete set of Java News Today pages; a good place to start experimenting might be to take the site as it is and turn it into something with a real design, one that might attract users. Or try creating new sections, authors, or keywords, or perhaps even drop all the existing ones and create a brand new set. Or start from scratch and create a news site about Linux, music, or anything else. For every dynamic Web site that exists today, an infinite number of ideas have yet to be explored. Let JavaServer Pages be the technology that turns your great idea into a dynamic and compelling Web site that might just be the Internet's next Big Thing.

Appendix A. Summary of Tags

A.1 Built-in Tags

Note: Parameters marked as "dynamic" may be expression language scripts.

`jsp:scriptlet`: Embeds Java code directly in the servlet constructed at translation time; equivalently, `<% ... %>`
Body: Fully formed Java code

`jsp:declaration`: Embeds Java declarations directly in the top level of the servlet constructed at translation time; equivalently, `<%! ... %>`
Body: Fully formed Java declarations

`jsp:expression`: Obtains the value of a Java expression, which may contain method calls, declared variables, or beans loaded with `jsp:useBean`; equivalently, `<%= ... %>`. This latter form may be used as the value attribute in a `jsp:param` or `jsp:setProperty` and as the `page` attribute in `jsp:forward` and `jsp:include`.
Body: A fully formed Java expression

`jsp:include`: Includes the contents of one JSP, servlet, or HTML page at request time			
Parameter	**Dynamic?**	**Required?**	**Description**
`page`	No	Yes	The page to include
`flush`	No	No	Flag indicating whether to send all data to the user after the included page completes or to continue to

`jsp:include`: Includes the contents of one JSP, servlet, or HTML page at request time

Parameter	Dynamic?	Required?	Description
			buffer it
Body: Any number of `jsp:param` tags			

`jsp:include`: Transfers control to another JSP, servlet, or HTML page. No data can be written either before or after this tag.

Parameter	Dynamic?	Required?	Description
`page`	No	Yes	The page to which the request should be transferred
Body: Any number of `jsp:param` tags			

`jsp:param`: A named parameter for a request to a `jsp:include` or `jsp:forward`. Such parameters are treated by the receiving page exactly as if they had come from a form.

Parameter	Dynamic?	Required?	Description
`name`	No	Yes	The name of the parameter
`value`	No	Yes	The value of the parameter
Body: None			

`jsp:useBean`: Makes a bean available to the rest of a page

Parameter	Dynamic?	Required?	Description
`id`	No	Yes	The name by which this bean will be known to the rest of the page
`scope`	No	No	The scope in which the bean lives
`class`	No	No	The class that implements the bean. The class and/or type attribute must be provided.
`type`	No	No	The class the bean should be treated as. The

`jsp:useBean`: Makes a bean available to the rest of a page			
Parameter	**Dynamic?**	**Required?**	**Description**
			implementing class must extend or implement this type.
`beanName`	No	No	The name of a serialized bean to load. When this attribute is used, the type must also be specified.
Body: Arbitrary JSP code, which will be executed if the bean has been created as a result of this tag			

`jsp:getProperty`: Displays a property from a bean			
Parameter	**Dynamic?**	**Required?**	**Description**
`name`	No	Yes	The name of the bean, as specified in the `jsp:useBean` tag
`property`	No	Yes	The name of the property to obtain
Body: None			

`jsp:setProperty`: Sets one or more properties in a bean			
Parameter	**Dynamic?**	**Required?**	**Description**
`name`	No	Yes	The name of the bean, as specified in the `jsp:useBean` tag
`property`	No	Yes	The name of the property to set. May be `"*"`, indicating to set all properties whose name matches the name of a parameter from a form
`value`	No	No	The value of the property; if not present, a value will be looked for in the set of parameters
Body: None			

A.2 Core Tags

`c:out`: **Sends the result of an expression to the page**			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	Yes	The expression to be evaluated and displayed
`escapeXml`	Yes	No	Defaults to `true`; if `true`, special characters will be converted to special codes that can be rendered within an HTML page
`default`	Yes	No	If the value is `null` and there is a default, that default will be displayed.
Body: Optional; if present, acts the same as the default attribute			

`c:set`: **Sets a variable or bean property**			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The value to which the variable target will be set
`var`	No	No	Name of the variable to set
`scope`	No	No	Scope in which the variable lives
`target`	Yes	No	Expression evaluating to a bean
`property`	Yes	No	The property in the target to be set
Body: Optional; if present, acts the same as the `value` attribute			

`c:remove`: **Removes a variable from a scope**			
Parameter	**Dynamic?**	**Required?**	**Description**
`var`	No	Yes	The variable to be removed
`scope`	No	Yes	The scope in which the variable lives
Body: None			

`c:catch`: Catches any exception that occurs during JSP processing

Parameter	Dynamic?	Required?	Description
`var`	No	Yes	The name of a variable in the page scope in which the `java.lang.Exception` object will be stored

Body: The content of the body will be processed as usual. If an exception occurs during this processing, it will be stored in the named `var`, and the JSP code following the closing tag will be processed.

`c:if`: Conditionally evaluates a block of JSP code

Parameter	Dynamic?	Required?	Description
`test`	Yes	Yes	An expression that should evaluate to a Boolean
`var`	No	No	The name of a variable in which the result of the test will be stored
`scope`	No	No	The scope in which the variable lives

Body: Anything in the body will be evaluated if and only if `test` evaluates to `true`.

`c:choose`: Conditional evaluates one of several blocks of JSP code

Body: Any number of `c:when` tags; zero or one `c:otherwise` tags

`c:when`: A single alternative within a `c:choose`

Parameter	Dynamic?	Required?	Description
`test`	Yes	Yes	An expression that should evaluate to a Boolean

Body: Anything in the body will be evaluated if and only if `test` evaluates to `true`. Once one `test` within a set of `c:where` tags succeeds, no other options will be tested.

`c:otherwise`: Default option within a `c:choose`

Body: If none of the tests within a set of `c:where` tags succeeds, the body of a `c:otherwise` tag will be evaluated, if provided.

`c:forEach`: Repeats a block of JSP code once for each element in an array or other collection			
Parameter	**Dynamic?**	**Required?**	**Description**
`var`	No	Yes	The name of the variable in which each item of the collection will be stored
`items`	Yes	No	An expression evaluating to an array, list, or map. If `items` is not provided, `begin` and `end` must be.
`varStatus`	No	No	The name of a variable in which the status of the iteration will be stored
`begin`	Yes	No	An expression evaluating to an integer from which the iteration will start counting
`end`	Yes	No	An expression evaluating to an integer at which the iteration will stop
`step`	Yes	No	An expression evaluating to an integer representing the increment to the count; defaults to 1
Body: The contents of the body will be repeated once for each element in `items`, or (end-count)/step times.			

`c:forTokens`: Repeats a block of JSP code once for each token			
Parameter	**Dynamic?**	**Required?**	**Description**
`var`	No	Yes	The name of the variable in which each item of the collection will be stored
`varStatus`	No	No	The name of a variable in which the status of the iteration will be stored
`items`	Yes	Yes	An expression evaluating to a single `String` containing multiple substrings, separated by delimiters
`delims`	Yes	Yes	An expression evaluating to a `String` containing

`c:forTokens`: **Repeats a block of JSP code once for each token**			
Parameter	**Dynamic?**	**Required?**	**Description**
			the set of delimiters with which to split the `items` string
`begin`	Yes	No	An expression evaluating to an integer representing the index of the first token of interest
`end`	Yes	No	An expression evaluating to an integer representing the index of the last token of interest
`step`	Yes	No	The number of intervening tokens to skip
Body: The contents of the body will be repeated once for each token.			

`c:input`: **Loads the content of any URL, and puts it in either the page or a variable**			
Parameter	**Dynamic?**	**Required?**	**Description**
`url`	Yes	Yes	The URL to retrieve
`context`	Yes	No	The name of the context when loading a URL from the same machine but a different Web application
`var`	No	No	The name of a variable in which to store the resulting text, rather than sending it to the page
`scope`	No	No	Scope in which the variable lives
`charEncoding`	Yes	No	The character encoding to use for the data
`varReader`	No	No	The name of the variable holding a `Reader` that will be used to load the data
Body: Any number of `c:param` tags			

`c:url`: **Constructs a URL resolving relative references and properly URL-encoding query parameters**			
Parameter	**Dynamic?**	**Required?**	**Description**

`c:url`: Constructs a URL resolving relative references and properly URL-encoding query parameters

Parameter	Dynamic?	Required?	Description
`value`	Yes	Yes	The base URL to be processed
`context`	Yes	No	The name of a foreign context if the resulting URL is not in the same Web application as the current page
`var`	No	No	A variable in which to store the resulting URL
`scope`	No	No	The scope of the variable
Body: Any number of `c:param` tags			

`c:redirect`: Constructs a URL and issues an HTTP redirect to the browser

Parameter	Dynamic?	Required?	Description
`url`	Yes	Yes	The base URL to be processed
`context`	Yes	No	The name of a foreign context if the resulting URL is not in the same Web application as the current page
Body: Any number of `c:param` tags			

`c:param`: Provides a parameter to a URL in a `c:import`, `c:url`, or `c:redirect`

Parameter	Dynamic?	Required?	Description
`name`	Yes	Yes	The name of the parameter
`value`	Yes	Yes	The value of the parameter
Body: None			

A.3 Format, Parsing, and Internationalization Tags

`fmt:setLocale`: **Sets the current locale for use in all subsequent format tags**			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	Yes	An expression evaluating to the name of the locale to use
`variant`	Yes	No	Minor variant of the locale
`scope`	No	No	Scope in which the locale should be set
Body: None			

`fmt:bundle`: **Loads a resource bundle that will be used in the body content**			
Parameter	**Dynamic?**	**Required?**	**Description**
`basename`	Yes	Yes	The base name of the bundle to load
`prefix`	Yes	No	A `String` that will be prepended to the value `key` of any `fmt:message` tags within the body
Body: Arbitrary JSP code; any `fmt:message` tags within the body will be resolved using the specified bundle			

`fmt:setBundle`: **Loads a resource bundle into a scope, to be used by all** `fmt:message` **tags within that scope**			
Parameter	**Dynamic?**	**Required?**	**Description**
`basename`	Yes	Yes	The base name of the bundle to load
`var`	No	No	The variable in which to store the bundle
`scope`	No	No	The scope of the variable
Body: None			

`fmt:message`: **Looks up a message in the current resource bundle**

Parameter	Dynamic?	Required?	Description
`key`	Yes	No	The name of the message to look up; if not provided as an attribute, this should be in the body
`bundle`	Yes	No	The bundle to use
`var`	No	No	Name of a variable holding a bundle, set up by an `fmt:setBundle` tag
`scope`	No	No	The scope of the variable
Body: A key if not provided as an attribute, along with any number of `fmt:param` tags			

`fmt:param`: Provides a parameter to a message in a bundle			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The value of the parameter; if not provided as an attribute, the body content will be used
Body: Arbitrary JSP code, which will be evaluated and used as the value if one is not provided as an attribute			

`fmt:requestEncoding`: Specifies the request's character encoding			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	Yes	The name of the encoding to use when processing request parameters
Body: None			

`fmt:timeZone`: Specifies a time zone to use when formatting dates within the body content			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	Yes	The time zone to use
Body: Arbitrary JSP code; any `fmt:formatTime` tags will use the given time zone			

`fmt:setTimeZone`: Sets the time zone globally or in a scoped variable

Parameter	Dynamic?	Required?	Description
`value`	Yes	Yes	The time zone to use
`var`	No	No	The variable in which to store the time zone
`scope`	No	No	The scope of the variable
Body: None			

`fmt:formatNumber`: Formats a number appropriately for the current locale			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The number to format; if not provided as an attribute, the body content will be used
`type`	Yes	No	Indicates whether the number should be treated as a number, currency, or percentage
`pattern`	Yes	No	The formatting pattern to use
`currencyCode`	Yes	No	When formatting as a currency, specifies the currency code
`currencySymbol`	Yes	No	When formatting as a currency, specifies the currency symbol
`groupingUsed`	Yes	No	Indicates whether `value` has any grouping symbols
`maxIntegerDigits`	Yes	No	Maximum number of digits comprising the integer portion of the result
`minIntegerDigits`	Yes	No	Minimum number of digits comprising the integer portion of the result
`maxFractionDigits`	Yes	No	Maximum number of digits comprising the noninteger portion of the result
`minFractionDigits`	Yes	No	Minimum number of digits comprising

`fmt:formatNumber`: Formats a number appropriately for the current locale

Parameter	Dynamic?	Required?	Description
			the noninteger portion of the result
`var`	No	No	The name of a variable in which to store the formatted result. If not provided, output goes to the page
`scope`	No	No	The scope of the variable
Body: If value is not provided as an attribute, the body content is used as the value.			

`fmt:parseNumber`: Parses a number, storing the result in a scope variable

Parameter	Dynamic?	Required?	Description
`value`	Yes	No	The string to be parsed; if not provided as an attribute, the body content will be used
`type`	Yes	No	Indicates whether the value should be treated as a number, currency, or percentage
`parseLocale`	Yes	No	The locale to use
`integerOnly`	Yes	No	Indicates whether to ignore any fractional part of the resulting number
`var`	No	Yes	The variable in which the resulting number should be stored
`scope`	No	No	The scope of `var`
Body: If the value is not provided as an attribute, the body content is used as the value.			

`fmt:formatDate`: Formats a date and/or time appropriately for a specified locale

Parameter	Dynamic?	Required?	Description
`value`	Yes	Yes	The value to format
`type`	Yes	No	Specifies whether the time, date, or both are to be formatted

`fmt:formatDate`: **Formats a date and/or time appropriately for a specified locale**			
Parameter	**Dynamic?**	**Required?**	**Description**
`dateStyle`	Yes	No	A predefined pattern from `java.text.DateFormat` to use when formatting
`timeStyle`	Yes	No	A predefined pattern from `java.text.DateFormat` to use when formatting
`pattern`	Yes	Yes	A pattern to use when formatting
`timeZone`	Yes	No	The time zone to use when formatting
`var`	No	No	The name of a variable in which the formatted value should be stored
`scope`	No	No	The scope of `var`
Body: None			

`fmt:parseDate`: **Parses a string representation of a date and/or time**			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The string to be parsed; if not provided as an attribute, the body content will be used
`type`	Yes	No	Specifies whether the time, date, or both are to be formatted
`dateStyle`	Yes	No	A predefined pattern from `java.text.DateFormat` to use when formatting
`timeStyle`	Yes	No	A predefined pattern from `java.text.DateFormat` to use when formatting
`pattern`	No	No	A pattern to use when formatting
`timeZone`	Yes	No	The time zone to use when formatting
`var`	No	No	The name of a variable in which the formatted value should be stored

`fmt:parseDate`: Parses a string representation of a date and/or time			
Parameter	**Dynamic?**	**Required?**	**Description**
`scope`	No	No	The scope of `var`
Body: If the value is not provided as an attribute, the body content is used as the value.			

A.4 SQL Tags

`sql:query`: Issues a query to a database, storing the results in a variable suitable for iterating with `c:forEach`			
Parameter	**Dynamic?**	**Required?**	**Description**
`sql`	Yes	No	The query to be evaluated; if not provided as an attribute, the body content will be used
`dataSource`	Yes	No	The JDBC data source to use to talk to the database; may specify the arguments to the driver/manager class or a `JNDI` resource
`startRow`	Yes	No	The first row to include in the result set
`maxRows`	Yes	No	The maximum number of rows to include in the result set
`var`	No	Yes	The variable in which the result of the query should be stored
`scope`	No	No	The scope of `var`
Body: The query, if not provided as an attribute, along with any number of `sql:param` tags			

`sql:update`:			
Parameter	**Dynamic?**	**Required?**	**Description**
`sql`	Yes	No	The update to be evaluated; if not provided as an attribute, the body content will be used
`dataSource`	Yes	No	The JDBC data source to use to talk to the database; may specify the arguments to the driver/manager class or a JNDI resource
`startRow`	Yes	No	The first row to include in the result set
`var`	No	Yes	The variable in which the result of the update

`sql:update`:

Parameter	Dynamic?	Required?	Description
			should be stored
`scope`	No	No	The scope of `var`
Body: The query, if not provided as an attribute, along with any number of `sql:param` tags			

`sql:transaction`: Provides a transaction context for a set of `sql:query` and `sql:update` tags

Parameter	Dynamic?	Required?	Description
`dataSource`	Yes	No	The JDBC data source to use to talk to the database; may specify the arguments to the driver/manager class or a JNDI resource
`isolation`	Yes	No	The isolation level of the transaction
Body: Arbitrary JSP code. All `sql:query` and `sql:update` tags within the body will occur in the same transaction.			

`sql:setDataSource`: Sets a SQL data source either in a variable or globally

Parameter	Dynamic?	Required?	Description
`dataSource`	Yes	No	The JDBC data source to use to talk to the database; may specify the arguments to the driver/manager class or a JNDI resource
`driver`	Yes	No	The JDBC driver to use
`url`	Yes	No	The URL to use
`user`	Yes	No	The name of the user with which to connect to the database
`password`	Yes	No	The password with which to connect to the database

`sql:setDataSource`: **Sets a SQL data source either in a variable or globally**			
Parameter	**Dynamic?**	**Required?**	**Description**
`var`	No	No	The variable in which to store the data source
`scope`	No	No	The scope of `var`
Body: None			

`sql:param`: **Provides a parameter to a SQL statement used in a** `sql:query` **or** `sql:update`			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The value of the parameter; if not provided as an attribute, the body content will be used
Body: If `value` is not provided as an attribute, the body content will be evaluated and used as the value.			

A.5 XML Tags

`x:parse`: **Parses XML into an internal form suitable for use in subsequent XML tags**			
Parameter	**Dynamic?**	**Required?**	**Description**
`xml`	Yes	No	The XML to parse; if not provided as an attribute, the body content will be used
`systemId`	Yes	No	The URI to use when parsing the XML
`filter`	Yes	No	A filter to apply to the XML source
`var`	No	Yes	The name of a variable in which to store the parsed representation
`scope`	No	No	The scope of `var`
`varDom`	No	No	The name of a variable in which to store the parsed representation as an instance of `org.w3c.dom.Document`
`scopeDom`	No	No	The scope of `varDom`
Body: If the XML is not provided as an attribute, the body content will be used.			

`x:out`: **Sends a segment of XML selected by an XPath expression to the page**			
Parameter	**Dynamic?**	**Required?**	**Description**
`select`	No	Yes	The XPath expression
`escapeXml`	Yes	No	Defaults to `true`; if `true`, special characters will be converted to special codes that can be rendered within an HTML page
Body: None			

`x:set`: **Stores a segment of XML selected by an XPath expression in a scoped variable**

Parameter	**Dynamic?**	**Required?**	**Description**
`select`	No	Yes	The XPath expression
`var`	No	Yes	The name of a variable in which to store the result
`scope`	No	No	The scope of `var`
Body: None			

`x:if`: Evaluates an XPath expression and evaluates the body content if the result is `true`			
Parameter	**Dynamic?**	**Required?**	**Description**
`select`	No	Yes	The XPath expression
`var`	No	No	A variable in which to store the result
`scope`	No	No	The scope of `var`
Body: Optional; if present, will be evaluated if the result of the select is `true`			

`x:choose`: Conditional; evaluates one of several blocks of JSP code
Body: Any number of `x:when` tags; zero or one `x:otherwise` tags

`x:when`: A single alternative within an `x:choose`			
Parameter	**Dynamic?**	**Required?**	**Description**
`select`	No	Yes	An XPath expression
Body: Anything in the body will be evaluated if and only if `test` evaluates to `true`. Once one test within a set of `c:where` tags succeeds, no other options will be tested.			

`x:otherwise`: Default option within an `x:choose`
Body: If none of the tests within a set of `x:where` tags succeeds, the body of an `x:otherwise` tag will be evaluated, if provided.

`x:forEach`: Repeats a block of JSP code once for each element selected by an XPath expression

Parameter	Dynamic?	Required?	Description
`var`	No	Yes	The name of the variable in which each item will be stored
`select`	No	No	The XPath expression
Body: Arbitrary JSP code			

`x:transform`: Applies an XSLT stylesheet to an XML document			
Parameter	**Dynamic?**	**Required?**	**Description**
`xml`	Yes	No	XML document to be transformed
`xslt`	Yes	No	The XSLT specification
`xmlSystemId`	Yes	No	The URI for parsing the XML document
`xsltSystemId`	Yes	No	The URI for parsing the XSLT style sheet
`var`	No	No	The variable in which the transformed document should be stored
`scope`	No	No	The scope of `var`
`result`	Yes	No	A variable in which to store the transformation result
Body: If the `xml` attribute is not provided, the body may hold the XML. If the `xslt` attribute is not provided, the body may contain the XSLT. At least one must be provided as an attribute. Body may also contain any number of `x:param` tags.			

`x:param`: Provides a parameter to the transformation performed by an `x:transform` tag			
Parameter	**Dynamic?**	**Required?**	**Description**
`name`	Yes	Yes	The name of the parameter

`x:param`: **Provides a parameter to the transformation performed by an** `x:transform` **tag**			
Parameter	**Dynamic?**	**Required?**	**Description**
`value`	Yes	No	The value of the parameter; if not provided as an attribute, the body content will be used
Body: If the value is not provided as an attribute, the body content will be used as the value.			

Appendix B. Configuring a Web Application

JavaServer Pages do not exist in a vacuum. At the very least, they will need access to numerous Java classes representing beans and the implementations of tag libraries. In addition, in any site of realistic complexity, JSPs will coexist with a set of servlets and filters and will need access to various other resources. Of course, some files will be needed to configure all this.

All these pieces together comprise a *Web application*, and the exact layout of such applications is defined as part of the J2EE specification. Standardizing on such a format has numerous advantages. It makes it possible to develop under one application server, such as Tomcat, and to deploy under something commercially supported. It also makes it possible to package Web applications as single files called war (Web application resource) files that can be sold or otherwise distributed without needing to support hundreds of deployment scenarios.

B.1 Layout of the Directories

All the JSP files live in the top level of the Web application. It is also possible to create arbitrary subdirectories for JSPs, and these are accessed as URLs in the obvious way.

All the other elements of the application are in a special directory: WEB-INF. Within this directory is the master configuration file, web.xml, which will be examined in the next section.

Java code libraries JAR (Java Archive) files are placed in the WEB-INF/lib directory.

This automatically adds these JAR files to the CLASSPATH for the application. There are no subdirectories under lib.

Code specific to the Web application may be placed in a JAR file, which is then installed in WEB-INF/lib, or in WEB-INF/classes. This directory is added to the effective CLASSPATH; within it, code is laid out according to the usual Java rules. For example, the class `com.awl.jsp.ch08.CdBean` would be found in

`WEB-INF/classes/com/awl/jspbook/ch08/CdBean.class`. It is up to the developer whether to leave the Java source files within these directories or to compile them elsewhere and move the resulting .class files.

It is also common to store resources, such as property files, resource bundles, and serialized beans, in the CLASSPATH. These resources may therefore also be placed in JAR files or under WEB-INF/classes.

Finally, by convention, is a special directory for tag library descriptors: WEB-INF/taglibs. This convention is not enforced, as the location of TLD files is specified in web.xml; in general, it is good practice to put them all in one place.

Once the application has been laid out according to these rules, it can be packaged into a .war file, with the following command:

```
jar -c0f ../application_name.war .
```

Here, `application_name` can be replaced with the chosen name for the application. In many application servers, such as Tomcat, it is possible to deploy an application by simply placing the .war file in the proper directory. For Tomcat, the directory is called webapps under the Tomcat home directory.

B.2 The Web.xml File

The web.xml file controls everything specific to the current Web applications. Typically, one or more files configure the application server as a whole; this vendor-specific file is not defined as part of the J2EE specification.

As with any good XML document (see Chapter 8), web.xml starts with a declaration, DTD reference, and root node. The top level looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
</web-app>
```

The real meat is found within the `web-app` tags and consists of the following elements in order.

1. An optional icon to be used by interactive configuration or maintenance applications. This can specify small (16 x 16) and large (32 x 32) images, which should live within the main directory of the Web application. Example:

```
<icon>
  <large-icon>images/jspbook_large.jpg</large-icon>
  <small-icon>images/jspbook_small.jpg</small-icon>
</icon>
```

2. An optional display name, which can also be used by administration tools. Example:

```
<display-name>JSP book examples</display-name>
```

3. An optional description, for the benefit of people reading the web.xml file or application tools. Example:

```
<description>
Example code from "JavaServer Pages, second edition"
</description>
```

4. An optional flag indicating that this Web application can be run within an application server that runs across multiple computers:

```
<distributable/>
```

5. Any number of context parameters consisting of a name, a value, and an optional description. These parameters will be used by the `servletContext` when it starts up. Example:

```
<context-param>
  <param-name>defaultColor</param-name>
  <param-value>black</param-value>
  <description>Standard background color</description>
</context-param>
```

6. Any number of filter definitions (see Chapter 9). These definitions must define the name of the filter and the implementing class and may also provide initialization parameters. In addition, an icon, a display name, and a description may be provided. Example:

```
<filter>
  <filter-name>authFilter</filter-name>
  <display-name>Auth filter</display-name>
  <description>
    Filter that prevents non-reports from creating articles
  </description>
  <filter-class>com.awl.jspbook.ch13.AuthFilter</filter-class>
  <init-param>
    <param-name>protectedPages</param-name>
    <param-value>create_article.jsp</param-value>
  </init-param>
</filter>
```

7. Any number of filter mappings, which tell the Web application which requests should be passed through each filter. Any given filter may appear in multiple filter-mappings, and multiple filter-mappings may refer to the same URL pattern. In this case, the chain is constructed in the order of the filter-mapping definitions. Example:

```
<filter-mapping>
  <filter-name>authFilter</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

8. Any number of listeners (see Chapter 11). Example:

```
<listener>
  <listener-class>
    com.awl.jspbook.ch11.SampleListener
  </listener-class>
</listener>
```

9. Any number of servlet declarations, specifying the name of the servlet, the implementing class, and any initialization parameters. An icon, display name, and description may also be provided. Additional configuration information related to security may be provided, but such configuration is beyond the scope of this book. Example:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>application</param-name>
    <param-value>
      config.ApplicationResources
    </param-value>
  </init-param>
  <init-param>
```

```
    <param-name>config</param-name>
    <param-value>
      /WEB-INF/classes/config/struts-config.xml
    </param-value>
  </init-param>
</servlet>
```

10. Any number of servlet mappings, associating a servlet name with a class of URLs. Like filters, one servlet may be configured to handle multiple sets of URLs; unlike filters, only one servlet can handle any given URL. Example:

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

11. An optional session config, which specifies how long, in seconds, sessions should last. Example:

```
<session-config>
  <session-timeout>3600</session-timeout>
</session-config>
```

12. Any number of MIME (multipurpose Internet mail extensions) mappings. These associate file name extensions with MIME types, which tell the browser how to handle the data. Example:

```
<mime-mapping>
  <extension>ogg</extension>
  <mime-type>audio/ogg</mime-type>
</mime-mapping>
```

13. An optional list of "welcome files." These files specify a set of files to look for if a user tries to access a directory name. For example, it would be possible to send users going to "/directory" to "/directory/index.jsp" if that file exists, to "/directory/index.html" if it doesn't, and so on. Example:

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

14. Any number of error pages, each of which associates a page with a kind of error. Using this, it is possible to send requests for a nonexistent page to one place, an application error to another, and so on. Example:

```
<error-page>
  <error-code>404</error-code>
  <location>no_such_page.jsp</location>
</error-page>
```

15. Any number of tag library declarations that associate a URI as used in a JSP with a tag library descriptor (see Chapter 13). Example:

```
<taglib>
  <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
  <taglib-location>/WEB-INF/taglibs/c.tld</taglib-location>
</taglib>
```

Then come places to specify various security and resource parameters, including the use of Enterprise JavaBeans. These are also beyond the scope of this book; see the relevant J2EE specifications for details.