Chapter 1

Starting with the Basics

In This Chapter

- Getting an overview of Jakarta Struts
- Creating the structure of a Web application
- Understanding the Model-View-Controller paradigm

Suppose that you're a programmer and your job is creating Web applications. You know the basics of Web applications. You use the Java programming language because of its power and flexibility. To make the Web pages interactive, you create Java Servlets and JavaServer Pages (JSP). You're getting pretty good at what you do, so your Web applications are becoming more complex.

You've heard the buzz about Jakarta Struts and how it can help structure leaner, tighter Web applications. You want to know how you can make use of this powerful programming framework to make your application programming more systematic and consistent, while taking less time. In this chapter, we explain what Jakarta Struts is all about and how it fits into the scheme of a Web application.

What Is Jakarta Struts?

Jakarta Struts is incredibly useful in helping you create excellent Web applications. When you use Jakarta Struts, your applications should work more effectively and have fewer bugs. Just as important (because your time is important), Struts should save you hours and hours of programming and debugging.

As we explain more fully later in this chapter, Struts is a *framework* that structures all the components of a Java-based Web application into a unified whole. These components of a Web application are

- **Java Servlets:** Programs written in Java that reside on a Web server and respond to user requests

- **JavaServer Pages:** A technology for generating Web pages with both static and dynamic content
- **JavaBeans:** Components that follow specific rules, such as naming conventions
- **Business logic:** The code that implements the functionality or rules of your specific application

We provide an overview of the first three items in this chapter. (The business logic varies with each application.)

Jakarta Struts uses a specific *paradigm,* or *design pattern,* to structure your application. You simply fill in the pieces of the structure. The design pattern is called Model-View-Controller (MVC). The MVC design pattern helps you organize the various pieces of the application puzzle for maximum efficiency and flexibility. We explain MVC later in this chapter and expand on the Model, View, and Controller concepts in Chapters 4, 5, and 6.

Structuring a Web Application

We define a *Web application* as a program that resides on a Web server and produces static and dynamically created pages in a markup language (most commonly HTML) in response to a user's request. The user makes the request in a browser, usually by clicking a link on the Web page. Figure 1-1 shows a high-level view of Web architecture. We explain the components of this figure subsequently in this chapter.

To build Web applications, you use Java 2 Enterprise Edition (J2EE), which provides support for Servlets, JSP, and Enterprise JavaBeans (EJB), a distributed, multi-tier, scalable component technology.

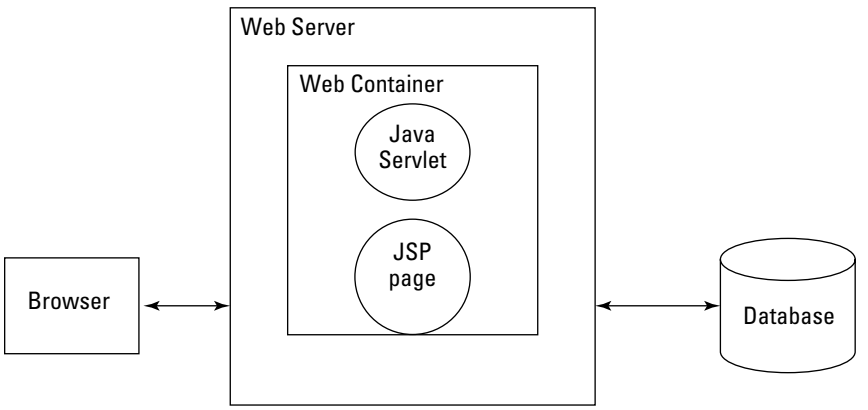

Figure 1-1: High-level view of Web architecture.

Where does Jakarta Struts come from?

To understand what Jakarta Struts is all about, you need to know something about the open-source movement that is its heritage. *Open-source* generally refers to software that the distributor provides at no cost to the user and that includes both the binary (compiled) code and the source code.

You obtain open-source software under a specific license, and the license can vary from one software provider to another. For example, the GNU (`www.gnu.org`) license provides that you must always include the source code if you redistribute the software of the application, whether or not you have made modifications to the original source code. The Apache (`www.apache.org`) license does not require you to provide the source code when you redistribute one of their applications. So open-source software licenses vary — check the license to be sure. For more information on open-source software, take a look at `www.opensource.org`.

Jakarta is one of many projects under the auspices of the Apache Software Foundation (ASF) (`www.apache.org`), formerly known as the Apache Group. The Apache Group was formed in 1995 by a number of individuals who worked together to create one of the most successful examples of an open-source project, the Apache Web Server (used by 64% of the Web sites on the Internet as of October, 2003). In 1999, the Apache Group became the non-profit Apache Software Foundation, to better provide support for its members and a legal presence to protect its resources.

As the popularity of Apache grew, so did ideas for other related open-source applications. Currently 16 software projects are supported by ASF. Actually, *software projects* is a bit of a misnomer because many of these projects have numerous subprojects that are really independent projects in themselves. Creativity is unlimited, so the ideas keep coming!

Jakarta (`jakarta.apache.org`) is one of the principal 16 ASF projects. To quote from their Web site, "Jakarta is a Project of the Apache Software Foundation, charged with the creation and maintenance of commercial-quality, open-source, server-side solutions for the Java Platform, based on software licensed to the Foundation, for distribution at no charge to the public." Struts is one of the 22 subprojects currently listed. Yes, this entire book is about one subproject.

Struts was created by Craig R. McClanahan and donated to ASF in May, 2000. Craig is an employee of Sun Microsystems and is the primary developer of both Struts and Tomcat 4. You can read about Craig and many other Struts contributors at `jakarta.apache.org/struts/volunteers.html`. The Struts 1.0 release had 17 contributors. With release 1.1 that number has jumped to 50. The project was named Struts as a reference to the architectural structures in buildings and homes that provide the internal support. The present version of Struts is 1.1.

A *Web container* is a program that manages the components of a Web application, in particular JSP pages and Java Servlets. A Web container provides a number of services, such as

- **Security:** Restricted access to components, such as password protection
- **Concurrency:** The capability to process more than one action at a time
- **Life-cycle management:** The process of starting up and shutting down a component

Some people use the term *JSP/Servlet container,* which means the same thing as Web container. We favor Web container — it's shorter and easier to type.

Apache Tomcat is an example of a Web container — an open-source *implementation* of the J2EE Java Servlet and JavaServer Pages (JSP) specifications. A *specification* is a document that describes all the details of a technology. The implementation is the actual program that functions according to its specification. In fact, Apache Tomcat is the official reference implementation for the J2EE Java Servlet and JSP specifications. As a result, Apache Tomcat is a popular Web container for Web applications that use JSP and Servlets, including applications that use Struts. We use Tomcat in all the examples in this book. However, many other commercial and open-source Web containers are available.

Typically, a Web container also functions as a Web server, providing basic HTTP (Hypertext Transfer Protocol) support for users who want to access information on the site. When requests are for static content, the Web server handles the request directly, without involving Servlets or JSP pages.

However, you may want your Web pages to adapt in response to a user's request, in which the response is *dynamic.* To generate dynamic responses, the Servlet and JSP portion of the container gets involved. Tomcat has the capability to act as both a Web server and a Web container. However, it also can interact with a standard Web server, such as Apache Web Server, letting it handle all static requests and getting involved only when requests require Servlet and JSP service.

Using Java Servlets

Java Servlets extend the functionality of a Web server and handle requests for something other than a static Web page. They are Java's answer to CGI (Common Gateway Interface) scripts of olden times (5 to 6 years ago). As their name implies, you write Java Servlets in Java and usually extend the `HttpServlet` class, which is the base class from which you create all Servlets. As such, Java Servlets have at their disposal the full functionality of the Java language, which give them a lot of power.

Servlets need to run in a *Web container,* an application that adheres to the Java Servlet Specification. In most cases, the container will support also the JavaServer Pages Specification. You can find a list of products supporting the

Java Servlet and JSP specifications at `java.sun.com/products/servlet/industry.html`. The latest Java Servlet Specification is 2.3, and the latest JavaServer Pages Specification is 1.2.

Creating JavaServer Pages

You use JavaServer Pages to present dynamic information to the user in a Web page. A JSP page has a structure like any static HTML page, but it also includes various JSP tags, or embedded Java *scriptlets* (short Java code fragments), or both. These special tags and scriptlets are executed on the server side to create the dynamic part of the presentation, so that the page can modify its output to reflect the user's request.

What really happens behind the scenes is that the JSP container translates the JSP page into a Java Servlet and then compiles the Servlet source code into runnable byte code. This translation process happens only the first time a user accesses the JSP page. The resulting Servlet is then responsible for generating the Web page to send back to the user.

Each time the JSP page is changed, the Web container translates the JSP page into a Servlet.

Listing 1-1 shows an example of a JSP page, with the JSP-specific tags in **bold.**

Listing 1-1 Sample JSP Page

```
<%@ page  contentType="text/html;charset=UTF-
      8"language="java" %>
<%-- JSTL tag libs --%>
<%@ taglib prefix="fmt" uri="/WEB-INF/fmt.tld" %>
<%-- Struts provided Taglibs --%>
<%@ taglib uri="/WEB-INF/struts-html-el.tld"
      prefix="html" %>
<html:html locale="true"/>
<head>
   <fmt:setBundle basename="ApplicationResources" />
   <title><fmt:message key="loggedin.title"/></title>
</head>
<body>
   <jsp:useBean id="polBean"
      class="com.othenos.purchasing.struts.POListBean"/>
   <H2>
      <fmt:message key="loggedin.msg">
         <fmt:param value='${polBean.userName}' />
      </fmt:message>
   </H2>
</body>
</html>
```

JSP defines six types of tag elements:

- **Action:** Follows the XML (eXtended Markup Language) format and always begins with `<jsp:some action/>`. It provides a way to add more functionality to JSP, such as finding or instantiating (creating) a JavaBean for use later. You see one example of an action tag in line 12 of the code in Listing 1-1.
- **Directive:** A message to the Web container describing page properties, specifying tag libraries, or substituting text or code at translation time. The form is `<%@ the directive %>`. Listing 1-1 has directives on lines 1, 3, and 5.
- **Declaration:** Declares one or more Java variables or methods that you can use later in your page. The tag has this form `<%! declaration %>`.
- **Expression:** Defines a Java expression that is evaluated to a `String`. Its form is `<%= expression %>`.
- **Scriptlet:** Inserts Java code into the page to perform some function not available with the other tag elements. Its form is `<% java code %>`.
- **Comment:** A brief explanation of a line or lines of code by the developer. Comments have the form `<%-- the comment --%>`. Lines 2 and 4 in Listing 1-1 are examples of comments.

Because a JSP file is just a text file, you can create it in just about any kind of text editor. Note that some editors understand JSP syntax and can provide nice features such as formatting and color coding. A few of the bigger ones are Macromedia Dreamweaver (`www.macromedia.com/software/dreamweaver/`), NetBeans (`www.netbeans.org`), and Eclipse (`www.eclipse.org`); the last two are complete Java development environments.

Like Java Servlets, JSP pages must be run in a Web container that provides support for JSP technology, as we explained in the preceding section, "Using Java Servlets."

Using JavaBeans

When you program in Java, you define or use classes that function as a template for objects that you create. A *JavaBean* is a special form of Java class that follows certain rules, including the methods it uses and its naming conventions.

Beans are so useful because they are portable, reusable, and platform independent. Beans are *components* because they function as small, independent programs. JavaBeans *component architecture* defines how Beans are constructed and how they interact with the program in which they are used.

Scope

Scope refers to an area in which an object (such as a Bean or any Java class) can be stored. Scopes differ based on the length of time stored objects are available for reference, as well as where the objects can be referenced from.

In JSP and Struts, scope can be one of four values:

- **Page:** Objects in the page scope are available only while the page is responding to the current request. After control leaves the current page, all objects stored in the page scope are destroyed.
- **Request:** Objects in the request scope are available as long as the current request is being serviced. A request can be serviced from more than one page.
- **Session:** The objects in the session scope last as long as the session exists. This could be until the user logs out and the session is destroyed or until the session times out due to inactivity. Each client using the Web application has a unique session.
- **Application:** The longest lasting scope is the application scope. As long as the application is running, the objects exist. Furthermore, objects in the application scope are available to all clients using the application.

You can call a JavaBean a Bean and everyone will know what you're talking about, as long as you're not discussing coffee.

The JavaBean documentation refers to the rules as *design patterns.* However, this term is more generally used to refer to design patterns such as the Model-View-Controller design pattern. *Naming conventions* is a more appropriate term.

As an example of the special Bean rules, let's look at properties. A Bean's properties that are exposed (public) are available only through the getter and setter methods, because the actual property definition is typically private (available to only the defining class). The properties follow the naming convention that the first letter of the property must be lowercase and any subsequent word in the name should start with a capital letter, such as `mailingAddress`. (We explain getters and setters after Listing 1-2.) Listing 1-2 is an example of a simple Bean.

Listing 1-2 Example of a Simple JavaBean

```
public class SimpleBean implements java.io.Serializable
{
    private String name;

    // public no-parameter constructor
    public SimpleBean()
```

```
    {
    }
    // getter method for name property
    public String getName()
    {
        return name;
    }
    // setter method for name property
    public void setName(String aName)
    {
        name = aName;
    }
}
```

In this example, `String` is the type of property and `name` is the property.

Methods that access or set a property are *public* (available to anyone using the Bean) and also use a certain naming convention. You name these methods as follows:

- To get a property's value, the method must begin with `get` followed by the property name with the first letter capitalized, as in `public String getName()`;.These methods are called *getters*.
- To set a property's value, the method must begin with `set` followed by the property name with the first letter capitalized and the value to set the property to, as in `public void setName(String theName)`;. These methods are called *setters*.

You should also be familiar with special naming conventions for Boolean and indexed properties. Many additional requirements exist, but they are less important for our situation. See `java.sun.com/docs/books/tutorial/javabeans/index.html` for more information on JavaBean requirements.

You should follow the JavaBean conventions when creating Beans to ensure that the user of the Bean knows how to get information in and out of the component. Classes that use the Beans know that if it's really a Bean, it follows the proper conventions; therefore, the class can easily discover the properties, methods, and events that make up the Bean.

In Struts, you commonly use Beans in Web applications and specifically in a more restricted manner than in the component architecture we just described. You use Beans more often as temporary holding containers for data. For example, suppose that a user requests to see a purchase order. The Web application then does the following:

1. Retrieves a copy of the requested purchase order information from the backend database

2. Builds a `PurchaseOrder` Bean
3. Populates the Bean with the retrieved data
4. Uses the Bean in the JSP page to display the data.

Because the Web application has transferred the data from the backend database to the Web page or for access by the business logic, the Bean is called a *Data Transfer Object* (DTO). A DTO is a design pattern.

Understanding the Model-View-Controller Design Pattern

Although Struts is not a complete application, it can be customized through extension to satisfy your programming needs. By using Struts, you can save hundreds, if not thousands, of hours of programming time and be confident that the underlying foundation is efficient, robust, and pretty much bug-free. When implemented properly, Struts is definitely a boon.

An *application framework* is a skeleton of an application that can be customized by the application developer. Struts is an application framework that unifies the interaction of the various components of a J2EE Web application — namely Servlets, JSP pages, JavaBeans, and business logic — into one consistent whole. Struts provides this unification by implementing the Model-View-Controller (MVC) design pattern. Struts provides an implementation of the MVC design pattern for Web applications. To understand why this is so important, you need to see why MVC is such a useful architecture when dealing with user interactions.

The MVC pattern is the grand-daddy of object-orientated design patterns. Originally used to build user interfaces (UI) in Smalltalk-80, an early object-oriented programming system, it has proved useful everywhere UI's are present. The MVC pattern separates responsibilities into three layers of functionality:

- **Model:** The data and business logic
- **View:** The presentation
- **Controller:** The flow control

Each of these layers is loosely coupled to provide maximum flexibility with minimum effect on the other layers.

What is a design pattern?

The expression "Don't reinvent the wheel" means that you shouldn't try to solve a common problem that many bright people have already faced and solved in a clever and elegant way. For many years, other disciplines (for example, architecture) have recognized that repeating patterns of solutions exist for common problems. In 1995, an often-quoted book called *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides (published by Addison-Wesley Publishing Co.) used the same technique to formalize problem-solving patterns in the field of object-orientated design.

A *design pattern* is a blueprint for constructing a time-tested solution to a given problem. It's not a concrete implementation; rather, it's a high-level design of how to solve a problem. Because design patterns are more general than concrete implementations, they are consequently more useful because they have broader applications.

The MVC design pattern

In the MVC design pattern, the Model provides access to the necessary business data as well as the business logic needed to manipulate that data. The Model typically has some means to interact with persistent storage — such as a database — to retrieve, add, and update the data.

The View is responsible for displaying data from the Model to the user. This layer also sends user data to the Controller. In the case of a Web application, this means that both the request and the response are in the domain of the View.

The Controller handles all requests from the user and selects the view to return. When the Controller receives a request, the Controller forwards the request to the appropriate *handler,* which interprets what action to take based on the request. The Controller calls on the Model to perform the desired function. After the Model has performed the function, the Controller selects the View to send back to the user based on the state of the Model's data.

Figure 1-2 shows the relationships among the three layers.

To get an idea of why the MVC pattern is so useful, imagine a Web application without it. Our fictional application consists of just JSP pages, with no Servlets. All the business logic necessary to service a user's request and present the user with the desired results is in those JSP pages. Although this scheme is simpler than an implementation using MVC, it is also difficult to work with for anything but the most trivial application, due to the intermixing of Model, View, and Controller elements.

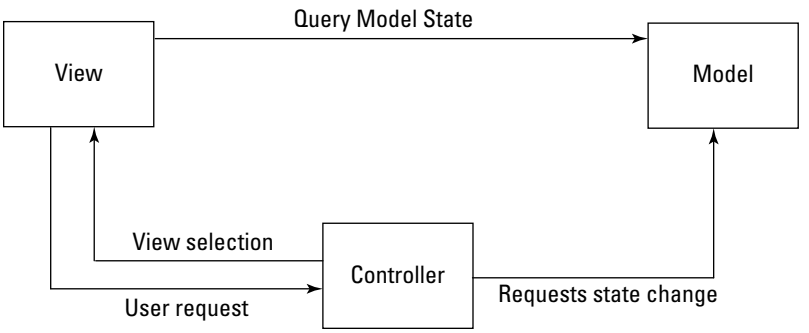

Figure 1-2: The Model-View-Controller pattern.

To illustrate the difference between Web applications that don't use MVC and those that do, think about the difference between Rocky Road and Neapolitan ice cream. Both may be delicious, but if you want to make any changes to Rocky Road, think about how much trouble it would be to switch the almonds for walnuts. The almonds are too deeply embedded in the ice cream to do the switch without affecting everything else. On the other hand, because Neapolitan is cleanly separated into layers, switching one flavor for another is an easy task. Think of Neapolitan as MVC compliant, and Rocky Road as not.

Using the MVC pattern gives you many advantages:

- **Greater flexibility**: It's easy to add different View types (HTML, WML, XML) and interchange varying data stores of the Model because of the clear separation of layers in the pattern.
- **Best use of different skill sets:** Designers can work on the View, programmers more familiar with data access can work on the Model, and others skilled in application development can work on the Controller. Differentiation of work is easier to accomplish because the layers are distinct. Collaboration is through clearly defined interfaces.
- **Ease of maintenance:** The structure and flow of the application are clearly defined, making them easier to understand and modify. Parts are loosely coupled with each other.

How Struts enforces the MVC pattern

The architecture of Struts provides a wonderful mechanism that, when followed, ensures that the MVC pattern remains intact. Although Struts provides a concrete implementation of the Controller part of the pattern, as well as providing the connections between the Controller and Model layers and between the Controller and View layers, it doesn't insist on any particular View paradigm or require that you construct the Model in a particular way.

The Struts Controller

Although Struts does not provide or require any particular Model or View components of the MVC pattern, it does implement the Controller as well as the mechanisms that bind the three layers and allow them to communicate with each other. The primary controller class is a Java Servlet called the `ActionServlet`. This class handles all user requests for Struts-managed URLs. Using information in the configuration files, the `ActionServlet` class then gets the appropriate `RequestProcessor` class that collects the data that is part of the request and puts it into an `ActionForm`, a Bean that contains the data sent from or to the user's form. The final step of the Controller is to delegate control to the specific handler of this request type. This handler is always a subclass of the `Action` class. Figure 1-3 shows how Struts uses the MVC pattern.

The `Action` subclass is the workhorse of the Controller. It looks at the data in the user's request (now residing in an `ActionForm`) and determines what action needs to be taken. It may call on the business logic of the Model to perform the action, or it may forward the request to some other View. The business logic may include interacting with a database or objects across the network or may simply involve extracting some data from an existing JavaBean.

After the necessary action has been performed, the `Action` subclass then chooses the correct View to send back to the user. The View is determined by the current state of the Model's data (the model state) and the specifications you defined in the Struts configuration file. (For an explanation of the configuration file, see the "The Struts configuration file" section later in this chapter). Figure 1-4 shows the principal classes of the Struts Controller.

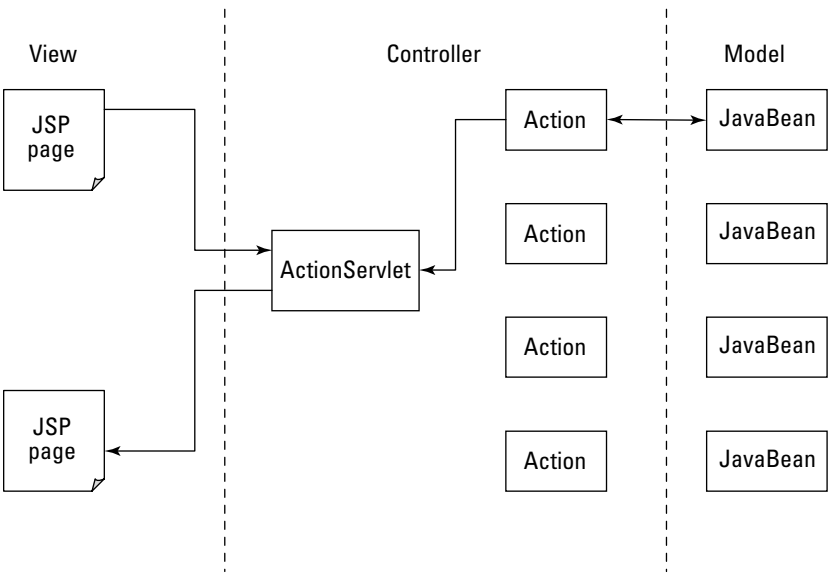

Figure 1-3: The Struts use of the MVC pattern.

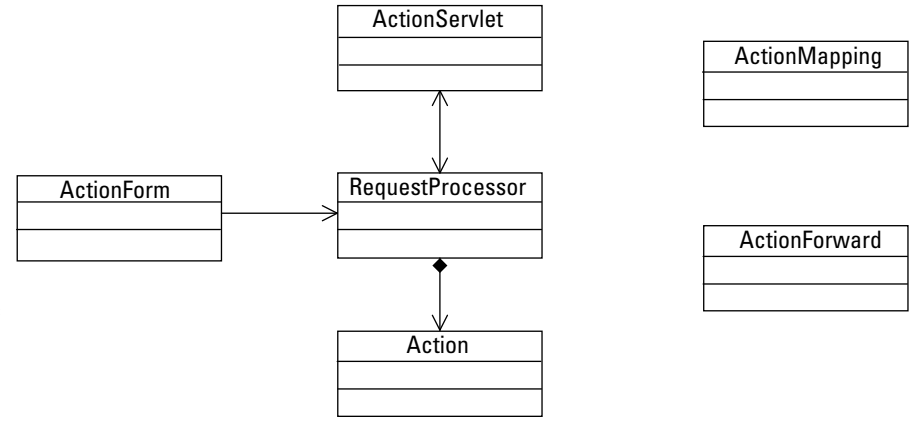

Figure 1-4: Principal Struts classes of the Struts Controller.

The Struts View

As mentioned, Struts does not provide, nor is it dependent on, a specific presentation technology. Many Struts applications use JSP (JavaServer Pages) along with the Struts tag library (Struts and Struts-EL), JSTL (JSP Standard Tag Library), and JSF (Java Server Faces). Some other possibilities are

- Apache Cocoon (`cocoon.apache.org/`)
- Jakarta Velocity templates (`jakarta.apache.org/velocity/index.html`)
- XSLT (eXtensible Stylesheet Language Transformation) (`www.w3.org/TR/xslt`)

The JSP specification provides for the creation of HTML-like tags that extend the functionality of JSP. These custom tags are bundled by their creators into *custom tag libraries* and are accompanied by a descriptor file called a *Tag Library Descriptor* (tld). The Struts and Struts-EL tag libraries are examples of this extended functionality.

Our examples throughout the book use JSP along with Struts-EL, JSTL, and other tag libraries. (For more on tag libraries, see Chapter 10.)

For new projects, the recommendation from the Struts Web site is to use *not* the standard Struts tag libraries, but instead the Struts-EL tag library along with JSTL. The Struts-EL tags library is really a reimplementation of the standard Struts tag library to make it compatible with JSTL's method of evaluating values. However, when a JSTL tag implemented the same functionality, the Struts tag was not reimplemented in the Struts-EL library. See `jakarta.apache.org/struts/faqs/struts-el.html` for full details on the Struts-EL tag library.

The Struts Model

Nothing in Struts dictates how to construct the Model. However, the best practice is to encapsulate the business data and operations on that data into JavaBeans, as we described previously when discussing Data Transfer Objects (in the "Using JavaBeans" section). The data and operations may reside in the same class or in different classes, depending on your application.

The operations represent the business logic that your application is defining. Operations may be the rules that should operate on a particular business entity. For example, if you're writing a purchasing system, part of the business data might be an entity called a Purchase Order. You may encapsulate this data into a class called `PurchaseOrder` as a way of representing the Purchase Order entity. Furthermore, you may choose to place your business rules directly into this class, or you may choose to put the rules into a different class.

The connection between the Controller and Model rests in the code that you write in the `Action` subclasses. The `Action` subclasses contain the analysis of the user's request that determines the interaction (if any) with the Model. Some examples of that interaction are

- Creating a JavaBean (like the `PurchaseOrder` class example above) that in turn accesses a database to populate itself and then makes it available to subsequent Views.
- Referencing a business logic object and asking it to perform some operation based on incoming data from the user.

The `Action` subclass initiates any action required to handle a user's request, thereby creating the connection with the Model.

When formulating a response, the Controller may pass some or all of the Model data to the View through the use of the `ActionForm` Bean. Although this Bean is a data container, it should not be considered part of the Model but rather just a transport mechanism between the Model and the View. Just as often, the View may directly reference the Model's data by referencing one or more of the Beans that belong to the Model.

The standard MVC pattern describes an interaction between the Model and the View so that when the Model's data changes, it can immediately *push* those changes out to the View so the user sees them. However, this is more difficult to achieve in the Web application architecture. Consequently, the View is commonly updated by the user requesting it.

The Struts configuration file

The Struts configuration file performs an important role in structuring your Struts application. Although it is not really part of the Model, View, or Controller, it does affect the functioning of the three layers. The configuration file allows you to define exactly which of your `Action` subclasses should be used under what circumstances and which `ActionForm` should be given to that `Action` subclass. So you specify part of the Controller interaction in the configuration file.

In addition, when the Controller decides which View to return to the user, it chooses the particular View according to specifications in the configuration file. Thus the configuration file actually defines many of the connections between the MVC components. The beauty of the configuration file is that you can change the connections *without* having to modify your code. The configuration file does much more than defining connections, which is why we devote all of Chapter 7 to the configuration file.