4.7 Automatically Populating Java Objects from Request Parameters: Form Beans

The getParameter method makes it easy to read incoming request parameters: you use the same method from doGet as from doPost, and the value returned is automatically URL-decoded (i.e., in the format that the end user typed it in, not necessarily in the format in which it was sent over the network). Since the return value of getParameter is String, however, you have to parse the value yourself (checking for missing or malformed data, of course) if you want other types of values. For example, if you expect int or double values, you have to pass the result of getParameter to Integer.parseInt or Double.parseDouble and enclose the code inside a try/catch block that looks for NumberFormatException. If you have many request parameters, this procedure can be quite tedious.

For example, suppose that you have a data object with three String fields, two int fields, two double fields, and a boolean. Filling in the object based on a form submission would require eight separate calls to getParameter, two calls each to Integer.parseInt and Double.parseDouble, and some special-purpose code to set the boolean flag. It would be nice to do this work automatically.

Now, in JSP, you can use the JavaBeans component architecture to greatly simplify the process of reading request parameters, parsing the values, and storing the results in Java objects. This process is discussed in detail in Chapter 14 (Using JavaBeans Components in JSP Documents). If you are unfamiliar with the idea of beans, refer to that chapter for details. The gist, though, is that an ordinary Java object is considered to be a *bean* if the class uses private fields and has methods that follow the get/set naming convention. The names of the methods (minus the word "get" or "set" and with the first character in lower case) are called *properties*. For example, an arbitrary Java class with a getName and setName method is said to define a bean that has a property called name.

As discussed in Chapter 14, there is special JSP syntax (property="*" in a jsp:setProperty call) that you can use to populate a bean in one fell swoop. Specifically, this setting indicates that the system should examine all incoming request parameters and pass them to bean properties that match the request parameter name. In particular, if the request parameter is named param1, the parameter is passed to the setParam1 method of the object. Furthermore, simple type conversions are performed automatically. For instance, if there is a request parameter called numOrdered and the object has a method called setNumOrdered that expects an int (i.e., the bean has a numOrdered property of type int), the numOrdered request parameter is automatically converted to an int and the resulting value is automatically passed to the setNumOrdered method.

Now, if you can do this in JSP, you would think you could do it in servlets as well. After all, as discussed in Chapter 10, JSP pages are really servlets in disguise: each JSP page gets translated into a servlet, and it is the servlet that runs at request time. Furthermore, as we see in Chapter 15 (Integrating Servlets and JSP: The Model View Controller (MVC) Architecture), in complicated scenarios it is often best to combine servlets and JSP pages in such a way that the servlets do the programming work and the JSP pages do the presentation work. So, it is really more important for servlets to be able to read request parameters easily than it is for JSP pages to do so. Surprisingly, however, the servlet specification provides no such capability: the code behind the property="*" JSP process is not exposed through a standard API.

Fortunately, the widely used Jakarta Commons package (see http://jakarta.apache.org/commons/) from The Apache Software Foundation contains classes that make it easy to build a utility to automatically associate request parameters with bean properties (i.e., with set*Xxx* methods). The next subsection provides information on obtaining the Commons packages, but the important point here is that a static populateBean method takes a bean (i.e., a Java object with at least some methods that follow the get/set naming convention) and a Map as input and passes all Map values to the bean property that matches the associated Map key name. This utility also does type conversion automatically, using default values (e.g., 0 for numeric values) instead of throwing exceptions when the corresponding request parameter is malformed. If the bean has no property matching the name, the Map entry is ignored; again, no exception is thrown.

Listing 4.12 presents a utility that uses the Jakarta Commons utility to automatically populate a bean according to incoming request parameters. To use it, simply pass the bean and the request object to BeanUtilities.populateBean. That's it! You want to put two request parameters into a data object? No problem: one method call is all that's needed. Fifteen request parameters plus type conversion? Same one method call.

Listing 4.12 BeanUtilities.java

```
package coreservlets.beans;

import java.util.*;
import javax.servlet.http.*;
import org.apache.commons.beanutils.BeanUtils;

/** Some utilities to populate beans, usually based on
 *  incoming request parameters. Requires three packages
 *  from the Apache Commons library: beanutils, collections,
 *  and logging. To obtain these packages, see
 *  http://jakarta.apache.org/commons/. Also, the book's
 *  source code archive (see http://www.coreservlets.com/)
 *  contains links to all URLs mentioned in the book, including
 *  to the specific sections of the Jakarta Commons package.
 *  <P>
 *  Note that this class is in the coreservlets.beans package,
 *  so must be installed in .../coreservlets/beans/.
 */

public class BeanUtilities {
  /** Examines all of the request parameters to see if
   *  any match a bean property (i.e., a setXxx method)
   *  in the object. If so, the request parameter value
   *  is passed to that method. If the method expects
   *  an int, Integer, double, Double, or any of the other
   *  primitive or wrapper types, parsing and conversion
   *  is done automatically. If the request parameter value
   *  is malformed (cannot be converted into the expected
   *  type), numeric properties are assigned zero and boolean
   *  properties are assigned false: no exception is thrown.
   */

  public static void populateBean(Object formBean,
                                  HttpServletRequest request) {
    populateBean(formBean, request.getParameterMap());
  }

  /** Populates a bean based on a Map: Map keys are the
   *  bean property names; Map values are the bean property
   *  values. Type conversion is performed automatically as
   *  described above.
   */

  public static void populateBean(Object bean,
                                  Map propertyMap) {
    try {
      BeanUtils.populate(bean, propertyMap);
    } catch(Exception e) {
      // Empty catch. The two possible exceptions are
      // java.lang.IllegalAccessException and
      // java.lang.reflect.InvocationTargetException.
      // In both cases, just skip the bean operation.
    }
  }
}
```

Putting BeanUtilities to Work

Listing 4.13 shows a servlet that gathers insurance information about an employee, presumably to use it to determine available insurance plans and associated costs. To perform this task, the servlet needs to fill in an insurance information data object (`InsuranceInfo.java`, Listing 4.14) with information on the employee's name and ID (both of type `String`), number of children (`int`), and whether or not the employee is married (`boolean`). Since this object is represented as a bean, `BeanUtilities.populateBean` can be used to fill in the required information with a single method call. Listing 4.15 shows the HTML form that gathers the data; Figures 4-12 and 4-13 show typical results.

Listing 4.13 SubmitInsuranceInfo.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import coreservlets.beans.*;

/** Example of simplified form processing. Illustrates the
 *  use of BeanUtilities.populateBean to automatically fill
 *  in a bean (Java object with methods that follow the
 *  get/set naming convention) from request parameters.
 */

public class SubmitInsuranceInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    InsuranceInfo info = new InsuranceInfo();
    BeanUtilities.populateBean(info, request);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Insurance Info for " + info.getName();
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI>Employee ID: " +
                   info.getEmployeeID() + "\n" +
                "  <LI>Number of children: " +
                   info.getNumChildren() + "\n" +
                "  <LI>Married?: " +
                   info.isMarried() + "\n" +
                "</UL></CENTER></BODY></HTML>");
  }
}
```

Listing 4.14 InsuranceInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Simple bean that represents information needed to
 *  calculate an employee's insurance costs. Has String,
 *  int, and boolean properties. Used to demonstrate
 *  automatically filling in bean properties from request
 *  parameters.
 */

public class InsuranceInfo {
  private String name = "No name specified";
  private String employeeID = "No ID specified";
  private int numChildren = 0;
  private boolean isMarried = false;

  public String getName() {
    return(name);
  }

  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setName(String name) {
    this.name = ServletUtilities.filter(name);
  }

  public String getEmployeeID() {
    return(employeeID);
  }

  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setEmployeeID(String employeeID) {
    this.employeeID = ServletUtilities.filter(employeeID);
  }

  public int getNumChildren() {
    return(numChildren);
  }

  public void setNumChildren(int numChildren) {
    this.numChildren = numChildren;
  }

  /** Bean convention: name getter method "isXxx" instead
   *  of "getXxx" for boolean methods.
   */

  public boolean isMarried() {
    return(isMarried);
  }

  public void setMarried(boolean isMarried) {
    this.isMarried = isMarried;
  }
}
```

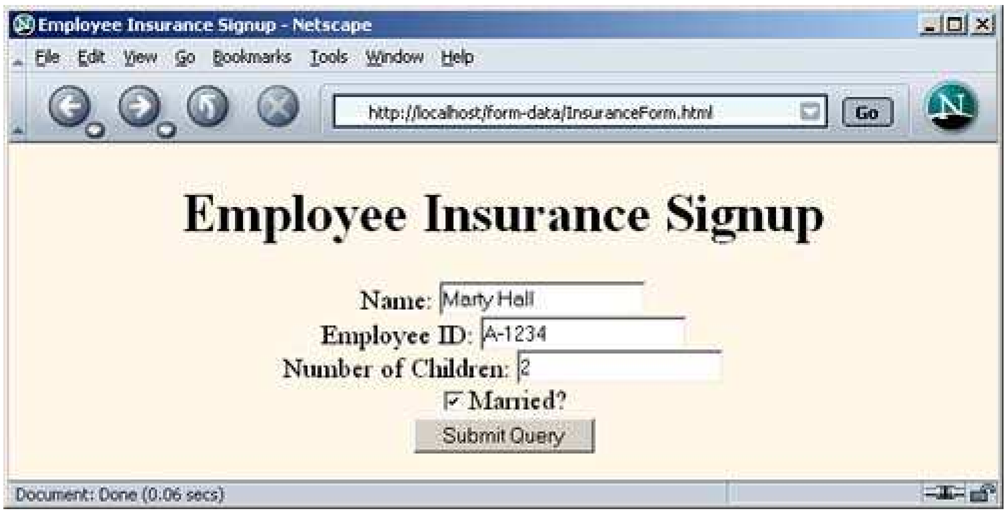
Listing 4.15 InsuranceForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Employee Insurance Signup</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Employee Insurance Signup</H1>

<FORM ACTION="/servlet/coreservlets.SubmitInsuranceInfo">
  Name:  <INPUT TYPE="TEXT" NAME="name"><BR>
  Employee ID: <INPUT TYPE="TEXT" NAME="employeeID"><BR>
  Number of Children:  <INPUT TYPE="TEXT" NAME="numChildren"><BR>
  <INPUT TYPE="CHECKBOX" NAME="married" VALUE="true">Married?<BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</CENTER></BODY></HTML>
```

Figure 4-12. Front end to insurance-processing servlet.



Employee Insurance Signup - Netscape

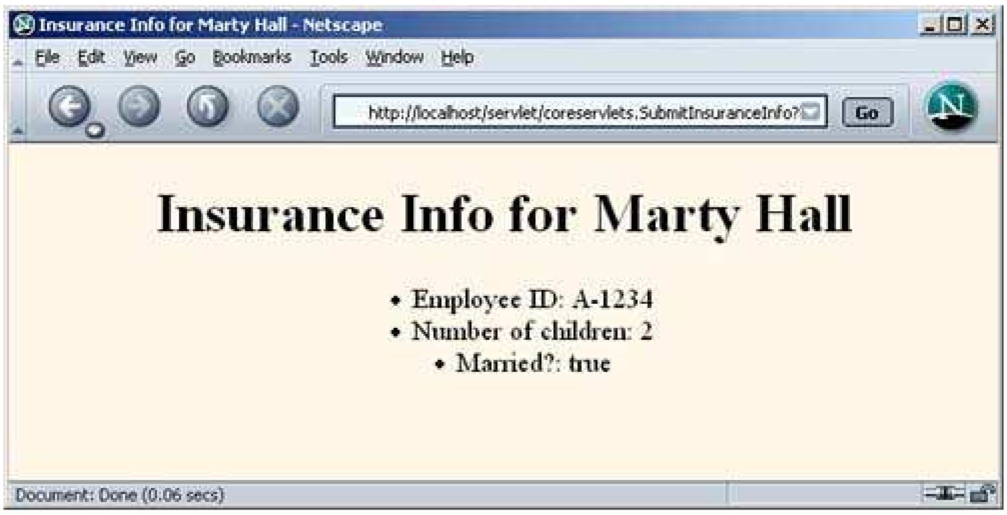
http://localhost/form-data/InsuranceForm.html

Employee Insurance Signup

Name: Marty Hall
Employee ID: A-1234
Number of Children: 2
☑ Married?
Submit Query

Document: Done (0.06 secs)

Figure 4-13. Insurance-processing servlet: the gathering of request data is greatly simplified by use of `BeanUtilities.populateBean`.



Insurance Info for Marty Hall - Netscape

http://localhost/servlet/coreservlets.SubmitInsuranceInfo?

Insurance Info for Marty Hall

- Employee ID: A-1234
- Number of children: 2
- Married?: true

Document: Done (0.06 secs)

Obtaining and Installing the Jakarta Commons Packages

Most of the work of our `BeanUtilities` class is done by the Jakarta Commons `BeanUtils` component. This component performs the reflection (determination of what writable bean properties—`set`*`Xxx`* methods—the object has) and the type conversion (parsing a String as an `int`, `double`, `boolean`, or other primitive or wrapper type). So, `BeanUtilities` will not work unless you install the Jakarta Commons `BeanUtils`. However, since `BeanUtils` depends on two other Jakarta Commons components—`Collections` and `Logging`—you have to download and install all three.

To download these components, start at http://jakarta.apache.org/commons/, look for the "Components Repository" heading in the left column, and, for each of the three components, download the JAR file for the latest version. (Our code is based on version 1.5 of the `BeanUtils`, but it is likely that any recent version will work identically.) Perhaps the easiest way to download the components is to go to http://www.coreservlets.com/, go to Chapter 4 of the source code archive, and look for the direct links to the three JAR files.

The most portable way to install the components is to follow the standard approach:

- For development, list the three JAR files in your `CLASSPATH`.
- For deployment, put the three JAR files in the `WEB-INF/lib` directory of your Web application.

When dealing with JAR files like these—used in multiple Web applications—many developers use server-specific features that support sharing of JAR files across Web applications. For example, Tomcat permits common JAR files to be placed in *`tomcat_install_dir`*`/common/lib`. Another shortcut that many people use on their development machines is to drop the three JAR files into *`sdk_install_dir`*`/jre/lib/ext`. Doing so makes the JAR files automatically accessible both to the development environment and to the locally installed server. These are both useful tricks as long as you remember that *`your-web-app`*`/WEB-INF/lib` is the only standardized location on the deployment server.

[Team LiB]   PREVIOUS NEXT