Servlet version two code

Remember, the model is just plain old Java, so we call it like we'd call any other Java method—instantiate the model class and call its method!

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class BeerSelect extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws IOException, ServletException {

        String c = request.getParameter("color");
        BeerExpert be = new BeerExpert();
        List result = be.getBrands(c);

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Beer Selection Advice<br>");

        Iterator it = result.iterator();
        while(it.hasNext()) {
           out.print("<br>try: " + it.next());
        }

    }
}
```

Don't forget the import for the package that BeerExpert is in.

We're modifying the original servlet, not making a new class.

Instantiate the BeerExpert class and call getBrands().

Print out the advice (beer brand items in the ArrayList returned from the model). In the final (third) version, the advice will be printed from a JSP instead of the servlet.

The HTML for the initial form page

The HTML is simple—it puts up the heading text, the drop-down list from which the user selects a beer color, and the submit button.

```
<html><body>
<h1 align="center">Beer Selection Page</h1>
<form method="POST"
  action="SelectBeer.do">
  Select beer characteristics<p>
  Color:
  <select name="color" size="1">
    <option value="light"> light </option>
    <option value="amber"> amber </option>
    <option value="brown"> brown </option>
    <option value="dark"> dark </option>
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form></body></html>
```

Why did we choose POST instead of GET?

This is what the HTML thinks the servlet is called. There is NOTHING in your directory structure named "SelectBeer.do"! It's a logical name...

This is how we created the pull-down menu; your options may vary.

(Did you figure out size="1" ?)

Q: **Why is the form submitting to "SelectBeer.do" when there is NO servlet with that name? In the directory structures we looked at earlier, I didn't see anything that had the name "SelectBeer.do". And what's with the ".do" extension anyway?**

A: SelectBeer.do is a logical name, not an actual file name. It's simply the name we want the client to use! In fact the client will NEVER have direct access to the servlet class file, so you won't, for example, create an HTML page with a link or action that includes a path to a servlet class file.

The trick is, we'll use the XML Deployment Descriptor (web.xml) to map from what the client requests ("SelectBeer.do") to an actual servlet class file the Container will use when a request comes in for "SelectBeer.do". For now, think of the ".do" extension as simply part of the logical name (and not a *real* file type). Later in the book, you'll learn about other ways in which you can use extensions (real or made-up/logical) in your servlet mappings.

The first version of the controller servlet

Our plan is to build the servlet in stages, testing the various communication links as we go. In the end, remember, the servlet will accept a parameter from the request, invoke a method on the model, save information in a place the JSP can find, and forward the request to the JSP. But for this first version, our goal is just to make sure that the HTML page can properly invoke the servlet, and that the servlet is receiving the HTML parameter correctly.

Servlet code

Be sure you match the development and deployment structures we created earlier.

```
package com.example.web;

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

public class BeerSelect extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Beer Selection Advice<br>");
        String c = request.getParameter("color");
        out.println("<br>Got beer color " + c);
    }
}
```

HttpServlet extends GenericServlet, which implements the Servlet interface...

We'll use doPost to handle the HTTP request, because the HTML form says:

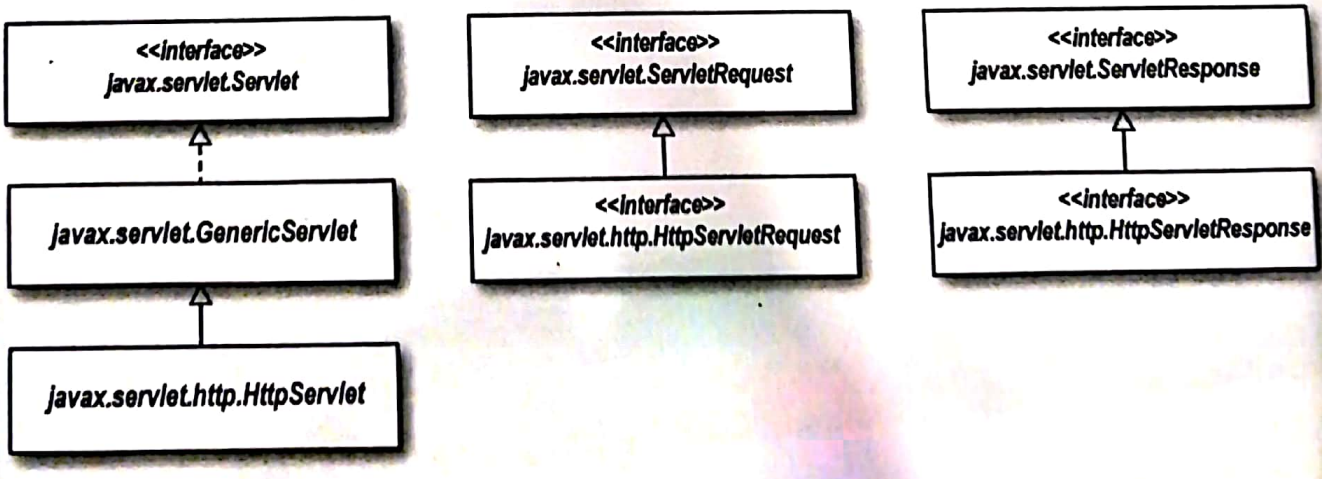
method=POST

This method comes from the ServletResponse interface.

This method comes from the ServletRequest interface. Notice that the argument matches the value of the "name" attribute in the HTML's <select> tag.

We're not giving back advice here, just displaying test information.

Key APIs



Building and testing the model class

In MVC, the model tends to be the "back-end" of the application. It's often the legacy system that's now being exposed to the web. In most cases it's just plain old Java code, with no knowledge of the fact that it might be called by servlets. The model shouldn't be tied down to being used by only a single web app, so it should be in its own utility packages.

The specs for the model

- Its package should be **com.example.model**
- Its directory structure should be /WEB-INF/classes/com/example/model
- It exposes one method, **getBrands()**, that takes a preferred beer color (as a String), and returns an ArrayList of recommended beer brands (also as Strings).

Build the test class for the model

Create the test class for the model (yes, *before* you build the model itself). You're on your own here; we don't have one in this tutorial. Remember, the model will still be in the development environment when you first test it—it's just like any other Java class, and you can test it without Tomcat.

Build and test the model

Models can be extremely complicated. They often involve connections to legacy databases, and calls to complex business logic. Here's our sophisticated, rule-based expert system for the beer advice:

```
package com.example.model;
import java.util.*;

public class BeerExpert {
  public List getBrands(String color) {
    List brands = new ArrayList();
    if (color.equals("amber")) {
       brands.add("Jack Amber");
       brands.add("Red Moose");
    }
    else {
       brands.add("Jail Pale Ale");
       brands.add("Gout Stout");
    }
    return(brands);
  }
}
```

Notice how we've captured complex, expert knowledge of the beer paradigm using advanced conditional expressions

```
File Edit Window Help Skateboard
% cd beerV1
% javac -d classes src/com/example/model/BeerExpert.java
```