
CE212

**Web Application
Programming
Part 3**

Servlets 1

A servlet is a Java program running in a server engine containing methods that respond to requests from browsers by generating content (usually, but not necessarily, in the form of an HTML page) to be sent to the browser.

The example on the next slide shows a `doGet` method from a servlet, which responds to requests of the form

`url?User=fred&Pass=xyz`

It performs very simple password validation and sends a plain text response.

The mapping of the URL to the servlet is performed by the server engine.

Servlets 2

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException
{ response.setContentType("text/plain");
  PrintWriter out = response.getWriter();
  String user = request.getParameter("User");
  String passwd = request.getParameter("Pass")
  if (passwd.length() > 0 &&
      passwd.charAt(0) != ('x'))
    out.println(user + " logged in");
  else
    out.println("Login Failed");
}
```


Java Server Pages

Java Server Pages involves the embedding of Java code inside HTML tags; unlike JavaScript this code is run on the server so the browser does not see any of the JSP.

When used well JSP provides a simple way to generate dynamic content web-pages; if misused with complex embedded Java it can become terribly messy and may violate the once-and-only once principle

Hence the embedded Java should be kept simple and any complex processing should be done by external helper classes or JavaBeans.

The JSP Lifecycle

A JSP page is translated into a servlet and then compiled.

When using the Tomcat server engine the compilation happens the first time a page is requested; hence the first request can be very slow!

For subsequent requests the performance just as fast as a servlet (because the requests are handled by a servlet which is already running in the server engine).

A Simple Example 1

Here is a simple JSP example.

```
<html>
<head>
<title> Hello JSP </title>
</head>
<body>
<p> Hello World:
    <%= new java.util.Date() %>
</p>
</body>
</html>
```

Observe that the embedded Java code appears in a tag between the symbols `<%` and `%>`.

A Simple Example 2

If the page on the previous slide was in a called **Date.jsp** it would be converted into a servlet called **Date_jsp.java**.

This file would contain (as a member of the **Date_jsp** class) a service method which would include the following (which continues on the next slide).

```
out = pageContext.getOut();
_jspx_out = out;
out.write("<html>\r\n");
out.write("<head>\r\n");
out.write("<title> Hello JSP ");
out.write("</title>\r\n");
out.write("</head>\r\n");
```

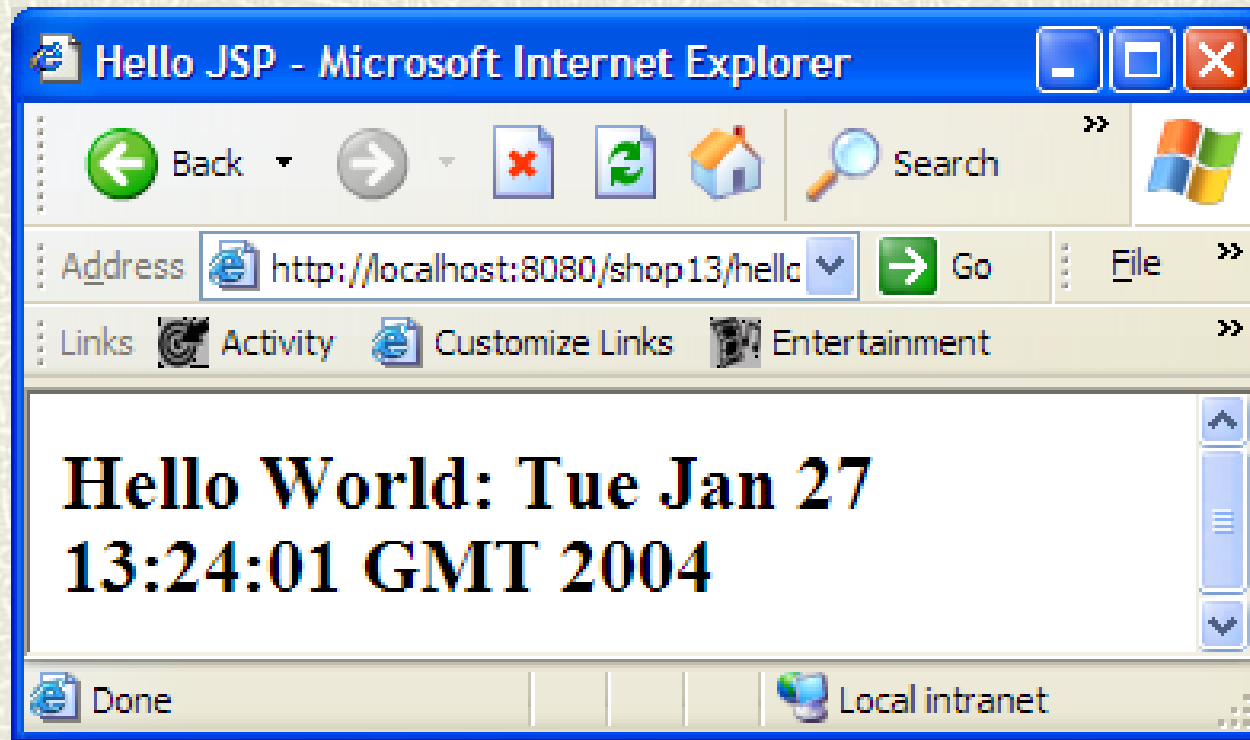
A Simple Example 3

```
// continued from previous slide
out.write("<body>\r\n");
out.write("<p> Hello World:\r\n");
out.print(new java.util.Date());
out.write("\r\n");
out.write("</p>\r\n");
out.write("</body>\r\n");
out.write("</html>\r\n");
```

The generated service method would be called by both the **doGet** and **doPost** methods of the servlet.

The following slide shows the HTML page that is generated by the servlet being displayed on a browser.

A Simple Example 4



Basic JSP Constructs

In the simple example we saw the use of *literal* HTML which is copied directly to the browser, e.g.

```
<head>
```

and embedded Java *expressions*, i.e.

```
<%= new java.util.Date() %>
```

The embedded expressions are supplied as arguments to a call to the **print** method at the appropriate point in the generation of the HTML (as seen in the servlet code). Hence they must not be terminated with semicolons.

JSP pages can also include *directives*, *declarations* and *scriptlets*.

JSP Directives

A directive contains instructions to the compiler – they are converted into appropriate Java code which is placed in the source for the servlet. Directives are placed in tags between `<%@` and `%>`.

Examples include the inclusion of another page, e.g.

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

and the importing of Java packages, e.g.

```
<%@ page import="java.util.Collection" %>
```

If more than one package is to be imported, the strings containing their names should be separated by commas.

JSP Declarations

JSP declarations are used to declare variables and methods. The code is copied into the servlet as class member declarations to allow the declared items to be used later in the JSP page

Declarations are placed in tags between `<%!` and `%>`. They should not be terminated with semicolons.

Examples:

```
<%! int count = 0 %>
<%! double sqr(double x)
    { return x * x;
    }
%>
```

JSP Scriptlets

A scriptlet is a fragment of Java code that is to be placed in the service method of the servlet (i.e. the method that is called when the page is requested by a browser).

Unlike expressions these scripts do not generate a value to be output as part of the HTML, but they may generate output by using `response.getWriter` to obtain a reference to the writer object for the HTML page being generated by the scriptlet.

Note that since the script is included in the service method it will be run every time a page is requested.

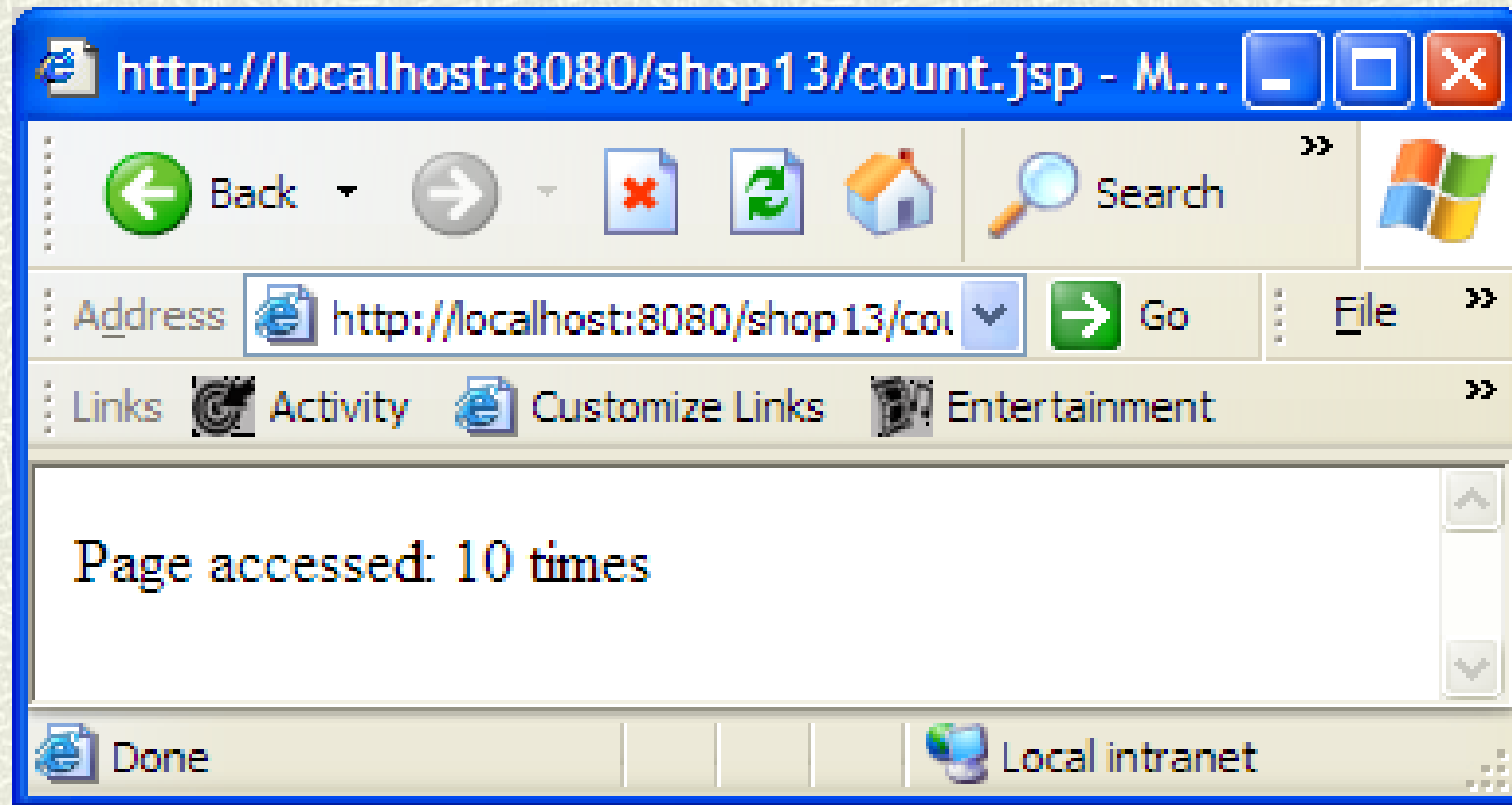
Access Count Example 1

The following example contains a declaration, an expression and a scriptlet.

```
<%! int n = 0; %>  
Page accessed: <%= ++n %> times  
<% if (n%1000 == 0 )  
{ n = 0;  
}  
%>
```

Sample browser output can be seen on the next slide.

Access Count Example 2



JSP Forward and Include 1

The JSP tags `<jsp:forward>` and `<jsp:include>` can be used to pass control from a JSP page to another JSP page.

A tag such as `<jsp:forward page = "Account.jsp">` terminates execution of current page and transfers control to the forwarder.

On the other hand `<jsp:include page = "Account.jsp">` returns control to the current page after executing the specified page.

JSP Forward and Include 2

Circumstances in which the forward tag might be used include situations where common activities are to be performed as a result of actions on different pages that send requests with different formats (e.g. it might be possible to add to a shopping basket from more than one place on a site). The pages handling the requests would extract the request data and then pass control to a shared page that updates and displays the basket.

Inclusion is likely to be used where several different pages contain common dynamic data – the included page would generate this data.

[Examples will be presented later.]

JSP Forward and Include 3

When a forward or include request is made the URL of the original page is shown on the browser, so the user is not aware of the forwarding and if the page is refreshed a request is made to the original page, not to the target page.

request and response

Each JSP page has access to the two objects seen on slide 3; they will be passed as arguments to the service method

The **request** object carries information passed by the HTTP request (e.g. made by the browser).

This includes any submitted form data.

The **response** object is used to pass information back to the client (browser). For example **response.getWriter()** returns an output stream for direct writing to the client.

Redirection

Control can be passed to another JSP page by applying the **sendRedirect** method to the **response** object, e.g.

```
response.sendRedirect("page2.jsp");  
return;
```

The **return** statement is necessary to prevent further processing of the current page.

In contrast to the use of **jsp:forward**, when redirection takes place the URL of the target page is displayed in the browser, so any page refresh would result in a request to the target page.

Form-Handling with JSP 1

HTML forms allow user to supply input using

- text fields – single line
- password fields – single line, blanked-out text
- text Areas – multi-line
- choices (pop-up menu)
- radio-buttons (1 from n)
- check-boxes (m from n)
- *browse* buttons
- *submit* buttons

See the file FormElements.html on Moodle for some examples of HTML form elements.

Form-Handling with JSP 2

Assuming that the contents of the form are to be sent to a server, the form should have an **action** attribute whose value is the URL to which the request containing the form contents should be sent. It may also have a **method** attribute whose value may be either **GET** or **POST**; if it does not the default is to use a POST request. The request will be submitted automatically when the user clicks on the submit button unless the form has an **onsubmit** attribute. The value of this attribute should be a call to a JavaScript function that performs validation; the request will then not be submitted if the function returns **False**. (Note that if the function fails, or cannot be run since JavaScript has been disabled, the request will be submitted.)

Form-Handling with JSP 3

JSP makes the handling of forms very simple. It is possible to use `request.getParameter()` to get submitted values, or a JavaBean can be defined to grab the values semi-automatically.

To generate forms using JSP we need to generate the HTML to send to the client. The forms should be well presented (e.g. aligned in a table). The input fields need to be named so that we can use the names as arguments in calls to `request.getParameter()` in order to extract the data from the submitted form

Form-Processing Architectures

Several different approaches can be taken to implement form processing. These include:

- helper classes for servlets
 - to generate forms (and other HTML)
 - to process the form input
- JSP + JavaBeans
- JSP + Tag Library (not covered in this course)
- XForms – not covered in this module

Hotel Booking Form Example 1

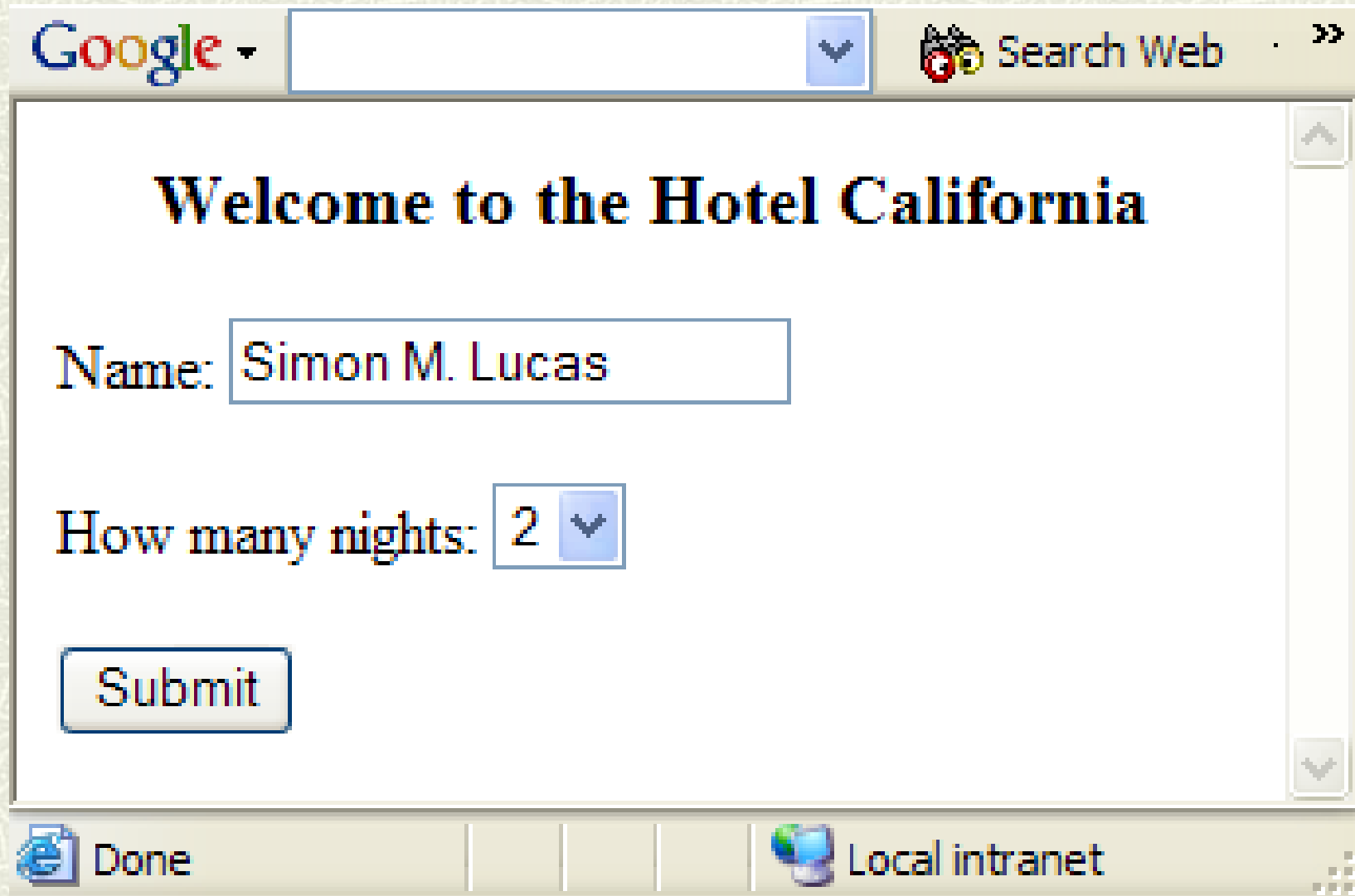
We illustrate the use of forms with a hotel booking example. The HTML for the form is on the next slide; this would normally be generated using JSP.

A screen shot is shown on the following slide.

Hotel Booking Form Example 2

```
<body>
<h3 align="center"> Welcome to the Hotel California </h3>
<form method="POST" action="BookHotel.jsp">
<p> Name: <input type="text" name="name" size="20"> </p>
<p> How many nights:
<select size="1" name="nNights">
  <option selected="">1</option>
  <option>2</option>
  <option>3</option>
</select></p>
<p>
<input type="submit" value="Submit" name="B1">
</p>
</form>
</body>
```


Hotel Booking Form Example 3



The screenshot shows a web browser window with a search bar at the top containing the Google logo and a search button. Below the search bar, the page title is "Welcome to the Hotel California". The form contains two input fields: "Name:" with the value "Simon M. Lucas" and "How many nights:" with a dropdown menu set to "2". A "Submit" button is located below the "How many nights:" field. The browser's status bar at the bottom shows "Done" and "Local intranet".

Google Search Web

Welcome to the Hotel California

Name:

How many nights:

Done Local intranet

Hotel Booking Form Example 4

The JSP code for the form-handling in the file `BookHotel.jsp` referenced in the HTML for the form could be

```
<html>
<head><title>Confirmation</title></head>
<body>
<h2>
  <%= request.getParameter("name") %>
  to stay for
  <%= request.getParameter("nNights") %>
  nights.
</h2>
</body>
</html>
```

JavaBeans 1

A JavaBean is a reusable component that follows certain conventions for class design and can be manipulated visually using a development environment. The conventions ensure that the class may be accessed automatically from JSP pages.

JavaBean classes that allow reading and writing of their properties (instance variables) must provide appropriate **get** and **set** methods to do this – these must use the variable names with the first letter capitalised (e.g. the method to get the value of **name** must be called **getName**).

JavaBeans come in two forms: simple and enterprise (EJB); in this module we will be using only simple beans.

JavaBeans 2

A version of the hotel-booking form handler JSP page using JavaBeans is presented on the next slide; the JavaBeans class used by this version can be seen on the following slide.

Note the use of the `jsp:setProperty` tag with the value `*` for the `property` attribute; this sets all of the properties of the bean to the values of the corresponding fields from the form that were submitted in the POST request.

Hotel Booking Form Example using JavaBeans 1

```
<jsp:useBean id='roomBooking'  
            scope='page'  
            class='beans.HotelBean' />  
  
<jsp:setProperty name='roomBooking' property='*' />  
  
<html>  
<head>  
<title>Bean test</title>  
</head>  
<body>  
<h2> <%= roomBooking.getName() %>  
      to stay for  
      <%= roomBooking.getNNights() %> nights. </h2>  
</body>  
</html>
```

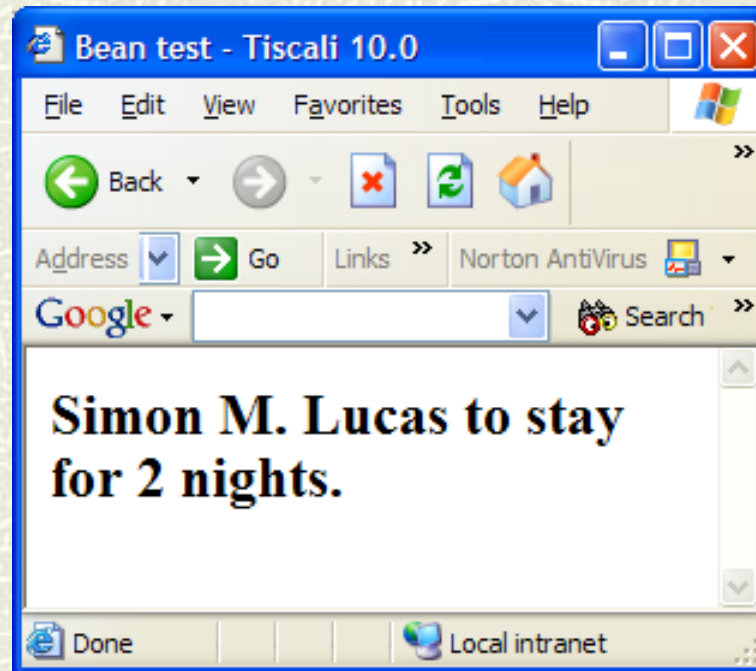
Hotel Booking Form Example using JavaBeans 2

```
package beans;

public class HotelBean
{ String name;
  int nNights;
  public String getName()
  { return name;
  }
  public void setName(String name)
  { this.name = name;
  }
  public int getNNights()
  { return nNights;
  }
  public void setNNights(int nNights)
  { this.nNights = nNights;
  }
}
```


Hotel Booking Form Example using JavaBeans 3

Here is screen shot of the output that was sent to the browser by the JavaBeans version of the form handler. (The manually-written JSP would produce similar output.)



JavaBeans Scope 1

In the `jsp:useBean` tag in the JSP code the `scope` attribute was given the value `page`.

Other possible values are `request`, `session` and `application`.

These values determine where a bean can be accessed and hence how long it will exist.

When *page scope* is used the bean exists only for the execution of the JSP page.

Request scope is similar, but the bean will survive any `forward` or `include` requests made in the JSP page.

JavaBeans Scope 2

When *session scope* is used the Bean exists for multiple requests within the web application, from a particular web browser instance. Hence this scope would be used for shopping baskets.

When *application scope* is used the Bean exists for all requests from all users, for all pages that use it. It survives until the web application server is restarted. This scope would be used for database connections (or connection pools).

JavaBeans v Manual Coding

In the hotel booking example the JavaBeans JSP page was more complex than the manually-coded version and also required an external class definition. However in real-world applications the JSP page is likely to be significantly more complex, justifying the use of JavaBeans.

The possibility of writing JavaBeans with different scopes also introduces benefits, such as the ability to provide a shopping basket.