The client operation is:

```
void  Op() { handler.handleRequest(); }
```

where $handler : C1Handler$.

Part 5: Distributed and Internet Applications

This part describes design techniques for systems which use internet to support distributed processing, such as online banking applications, e-commerce, web-based teaching applications, etc.

- Architectures for Internet Applications
- Internet Application Programming Languages
- JavaScript
- JSP
- Servlets

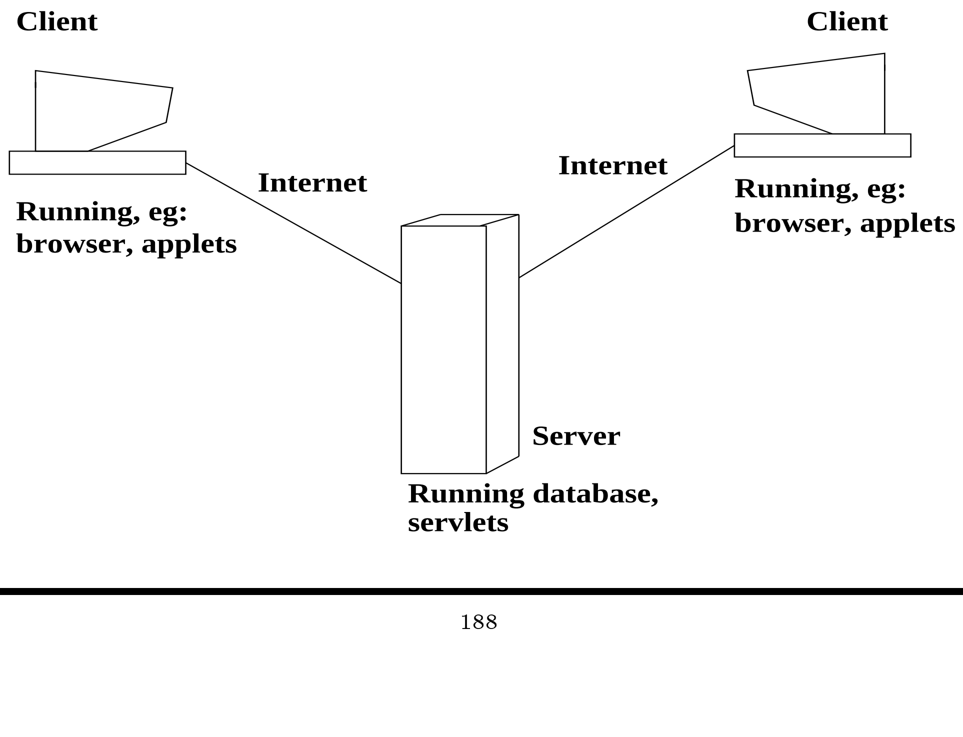
Architectures for Internet Applications

In order to provide wide, sometimes global, access to a particular set of information or resources, many applications are designed to operate in a *distributed* manner – do processing on two or more physically separate hosts. When internet is used to support communication between the hosts, termed an *internet application*.

Key aspect is separation between *client*: the process requesting a service, eg, a web browser on a customers PC, and *server*: the process providing a service, eg, a web server on a remote host where the web pages being browsed are stored.

Internet applications can use mixture of *client-side* and *server-side* processing to carry out functionality.

Client/Server Internet Architecture

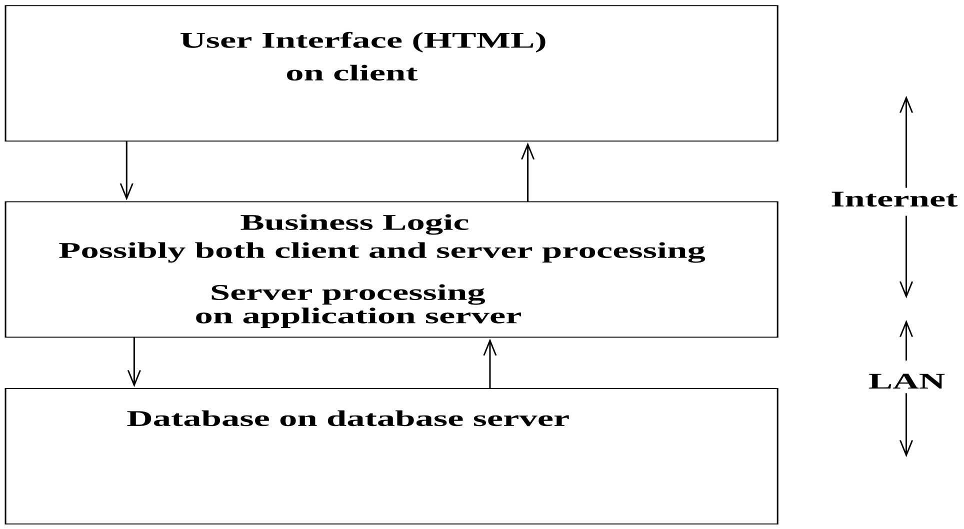


Distributed Three-tier Applications

In many cases processing at server side is further distributed into an *applications* server which does the substantial server-side processing, and a *database* server, which manages a repository for data.

This architecture is called a *distributed three tier* structure.

Distributed Three Tier Architecture



Most general architecture for internet applications in current use is a five-tier architecture consisting of levels:

Client tier: displays information to user and receives information from user. Can be either *thin client* with minimal processing apart from its visual interface functionality, or *fat client* doing more substantial computation. Thin clients typically use web browsers to achieve maximum portability: any machine with browser and internet connection can provide an interface to application. Such clients are called *web clients*.

Presentation tier: This manages presentation of information to client: which pages to send to client, eg, an error report page versus a confirmation page for an on-line restaurant booking, and what sequence of interaction to follow.

Also relays user requests to business tier.

Business tier: This contains business rules of application. Eg,

rules concerning what days bookings will be accepted for, what maximum size of party can be accepted, etc.

Integration tier: This mediates between business tier and resource tier. Manages detail of data retrieval, using interfaces such as JDBC. Insulates business and higher tiers from direct knowledge of how data is stored and retrieved.

Resource tier: Stores data and interacts with external resources such as credit card authorisation services or business-to-business services. Usually involves external commercial software such as a relational database.

In J2EE architecture presentation tier is called the *web tier*, and integration and resource tiers are bundled together as *EIS tier*.

Internet Application Components

- Web pages, consisting of HTML, often combined with some scripting language code which provides dynamic processing of page data and modification of its visual appearance.
- Program files, eg, in Java, which perform main computations and decision-making activities of application. Input data may come from HTML forms completed and submitted by client, output consists of web pages to be returned to client for viewing on browser.
- Configuration files, specifying system parameters, such as location and name of a database and database driver. These often written in structured text format such as XML. Enable application to be quickly deployed on a different server by separating out data which varies between servers from main application code.

Program components are subdivided further into:

- **Controller** components, which do substantial processing, such as selecting what set of web pages to return to a client based on their login id.
- **Session bean** components, which encapsulate information and operations specific to a particular client session (a connection by one browser process to server). Their data does not persist beyond session. Session beans are further divided into:
  - *Stateless session beans*: session beans without data. They typically provide utility functions.
  - *Statefull session beans*: session beans with stored data, which persists for duration of session. An example could be a shopping cart.

- **Entity bean** components, which provide access to persistent data, such as that stored in a database. Typically they provide an object-oriented view of tables of relational database: each entity bean class corresponds to a database table, and it has *getAtt* and *setAtt* methods for each of its attributes which correspond to columns of that table. Each object of bean corresponds to a row of table.

Following class is example of a statefull session bean. Could be used on e-commerce web site to keep track of items a customer intends to purchase:

```
import java.util.HashMap;
import java.util.Set;
import java.util.Iterator;

public class BasketBean
{ private HashMap contents = new HashMap();

  public void addItem(Long itemid)
  { updateItem(itemid,1); }

  public void updateItem(Long itemid, int quantity)
  { if (quantity > 0)
    { contents.put(itemid,new Integer(quantity)); }
    else
```

```
    { deleteItem(itemid); }
  }

  public Set getItems()
  { return contents.keySet(); }

  public int getQuantity(Long itemid)
  { return ((Integer) contents.get(itemid)).intValue(); }

  public void deleteItem(Long itemid)
  { contents.remove(itemid); }

  public void emptyBasket()
  { contents.clear(); }

  public int numberOfItems()
  { return contents.size(); }
```



```
  public static void main(String[] args)
  { BasketBean basket = new BasketBean();
    basket.addItem(new Long(10014));
    basket.updateItem(new Long(20332),4);
    Set contents = basket.getItems();
    Iterator iter = contents.iterator();
    Long itemid;
    int quantity;
    while (iter.hasNext())
    { itemid = (Long) iter.next();
      quantity = basket.getQuantity(itemid);
      System.out.println("Item number " + itemid.toString() +
                         " is in the basket " + quantity +
                         " times.");
    }
  }
}
```

Internet Application Programming Languages

Wide range of technologies exist for programming internet applications, many of which are Java-based. Some client-side examples are:

- *Applets* – Java code which is invoked when a browser views a particular web page held on server. Code executes on client, usually cannot modify state of client machine (eg, deleting a file) but may use its display to present or receive information, and can connect, using Java sockets or RMI, back to server host to transmit/retrieve data from this host.
- *JavaScript* – a Java-like scripting language (a simplified programming language) which executes on client when client browser views web pages containing embedded JavaScript.

Advantage of client-side processing – reduces processing load on server. However server-side processing increases flexibility and

efficiency (when application needs to frequently refer to database). Also more secure, because client may view or decompile code used for client-side processing.

Client side processing, such as form validation, also may not occur because client browser may have disabled scripts/applets or be unable to execute client-side language used.

Some server-side technologies are:

- *ASP* – server-side scripting for Windows servers, scripts to be executed on server are embedded in web pages, can access and update server data directly. Results of processing returned to client browser as HTML.
- *Perl* – server-side text processing and scripting language for generating web pages.
- *JSP* – JavaServer Pages. These embed Java statements in HTML, executed on server when page is viewed by a client browser. Results of processing are returned as web pages.
- *Java Servlets* – a pure Java server-side technology: methods of servlet classes generate web pages for display on client.

JavaScript

JavaScript originally developed by Netscape to add dynamic functionality such as animations, context-sensitive help, etc, to web pages.

Notation is close to Java, many of standard Java statements and control structures may be used in JavaScript.

Here is a simple JavaScript program which checks that two numbers entered by a user represent a valid date:

```
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Date Check Program</TITLE>

<SCRIPT LANGUAGE = "JavaScript">
  var firstNumber,  // first string entered
      secondNumber, // second string
```

```
    number1,      // 1st string as number
    number2;      // 2nd string as number

  // read in numbers as strings:
  firstNumber = window.prompt("Enter month (1--12)", "1");
  secondNumber = window.prompt("Enter day (1--31)", "1");
  // convert from strings to numbers:
  number1 = parseInt(firstNumber);
  number2 = parseInt(secondNumber);
  if (1 <= number2 && number2 <= 31 &&
      (number1 == 1 || number1 == 3 || number1 == 5 ||
       number1 == 7 || number1 == 8 || number1 == 10 ||
       number1 == 12))
  { document.writeln("<H1>Date is correct</H1>"); }
  else if (1 <= number2 && number2 <= 30 &&
           (number1 == 6 || number1 == 4 ||
            number1 == 9 || number1 == 11))
  { document.writeln("<H1>Date is correct</H1>"); }
```



```
  else if (1 <= number2 && number2 <= 29 && number1 == 2)
  { document.writeln("<H1>Date is correct</H1>"); }
  else
  { document.writeln("<H1>Date is not correct!</H1>"); }
</SCRIPT>
</HEAD>

<BODY>
Click Refresh or Reload to run script again
</BODY>
</HTML>
```

This script pops up simple dialog entry boxes for two numbers (very similar to *JOptionPane* input dialog of Java Swing), checks if date entered is valid, then prints message to current web page.

Features of JavaScript illustrated are:

- Variable declarations, written as:

  `var identifier;`

  No explicit type for variable is given. Instead any value (string, array, object, number or boolean) can be stored in a JavaScript variable.

- *if* statement, which has identical syntax to *if* in Java.
- Reading values from a dialog, using *prompt*. As in Java, these values are strings, need to be converted to numbers, using *parseInt*, before numeric processing can be used on them.
- $document.writeln(s);$ This writes string s back to current web

page, s is then interpreted by browser. HTML tags can be used in s to format output.

  Other output techniques are to write to browser's information bar by assignment $window.status = s;$ where s is message string, or open an *alert* dialog using $window.alert(s);$

- JavaScript program is written within a <SCRIPT> tag pair in head of document, with *JavaScript* identified as script language being used.

Evaluation of arithmetic expressions follows same rules as in Java, and same comparison operators $==$, $!=$, $<$, $>$, $<=$ and $>=$ are used.

Array data structure in JavaScript is similar to that of Java. To declare an array of 10 elements, write:

```
var arr1 = new Array(10);
```

Numbering of elements starts from 0, as in Java, size of an array is $arr1.length$, and individual elements are referred to by $arr1[index]$.

However, arrays can grow dynamically: if assign a value to $arr1[10]$ then value will be stored and $arr1$ extended to have length 11.

JavaScript adopts *while*, *do* and *for* loops of Java, written as their Java counterparts except that a *for* loop has form:

```
for (var x = 0; x < val; x++)
{ ... statements ... }
```

Loops are often used to write out an HTML table, with each row being generated by one iteration of loop.

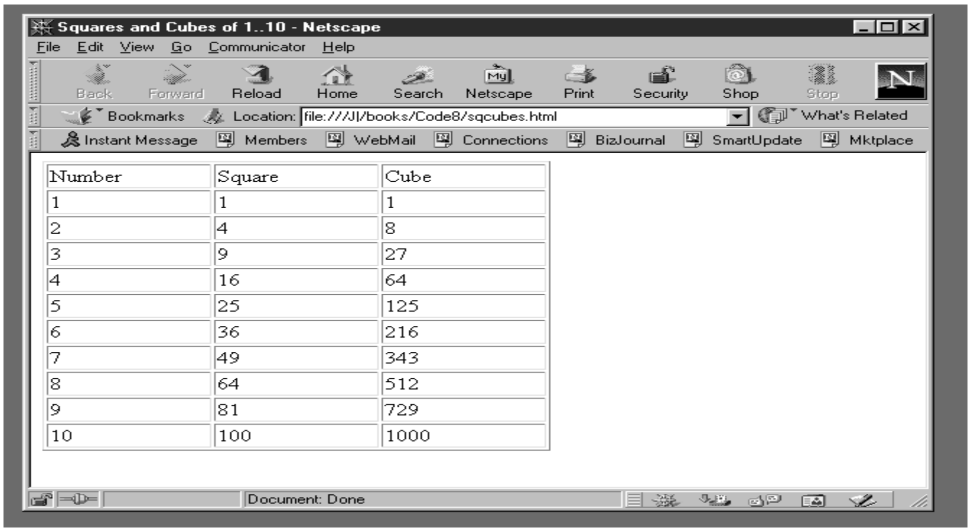
JavaScript programs are structured by being broken up into *functions*, corresponding to methods of a Java class.

Here is an example illustrating *for* loops, and function definition and invocation:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <HEAD>
<TITLE>Squares and Cubes of 1..10</TITLE>

<SCRIPT LANGUAGE = "JavaScript">
  document.writeln("<TABLE BORDER = '1' WIDTH = '60%'>");
  document.writeln("<TR><TD WIDTH = '20%'>Number</TD> " +
    "<TD WIDTH = '20%'>Square</TD> " +
    "<TD WIDTH = '20%'>Cube</TD> </TR>");
  for (var i = 1; i <= 10; i++)
  { document.writeln("<TR> <TD> " + i +
                     " </TD> <TD> " + square(i) +
                     " </TD> <TD> " + cube(i) +
                     " </TD> </TR>");
  }
```

```
  document.writeln("</TABLE>");
  // end of main function

  function square(x)  // function definition
  { return x*x; }

  function cube(x)
  { return x*square(x); }
</SCRIPT>
</HEAD>  <BODY>  </BODY>
</HTML>
```


Output of Squares/Cubes script



Number	Square	Cube
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Much of power of JavaScript as a client-side interface language comes from ability to detect user events such as mouse clicks or other actions on browser interface, and to invoke functions in response to these events.

Common events to which JavaScript event handlers can be attached are:

Event	*HTML Syntax*
Page is loaded	$< BODY\ \ ONLOAD = \text{“}function()\text{”} >$
Button is pressed	$< INPUT\ \ TYPE = \text{“}button\text{”}$ $VALUE = \text{“}Command\text{”}$ $ONCLICK = \text{“}function()\text{”} >$
Mouse is moved	$< BODY\ \ ONMOUSEMOVE = \text{“}function()\text{”} >$
Mouse over element	$< ...\ ONMOUSEOVER = \text{“}function()\text{”} >$
Mouse leaves element	$< ...\ ONMOUSEOUT = \text{“}function()\text{”} >$
Form entry active	$< INPUT \ldots ONFOCUS = \text{“}function()\text{”} >$
Form entry inactive	$< INPUT \ldots ONBLUR = \text{“}function()\text{”} >$

Example of event handling:

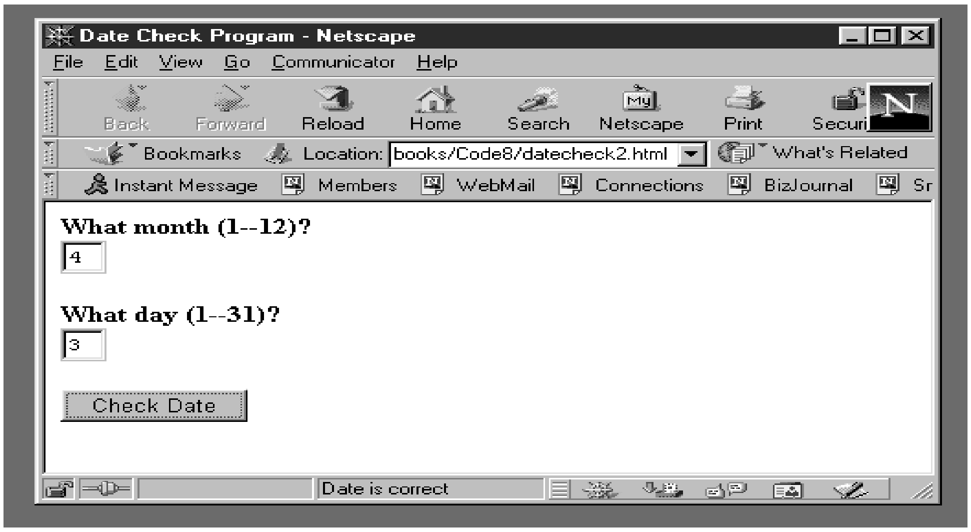
```
<HTML>
<HEAD>
<TITLE>Date Check Program</TITLE>

<SCRIPT LANGUAGE = "JavaScript">
// An array which holds number of days in each month:
var maxDaysInMonth =
  [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

function checkDate()
{ var smonth,  // month as a string
      sday,    // day as a string
      nmonth,  // month as a number
      nday;    // day as a number

  // Get values from form fields:
```



```
  smonth = document.date.month.value;
  sday = document.date.day.value;
  // convert from strings to numbers:
  nmonth = parseInt(smonth);
  nday = parseInt(sday);
  if (1 <= nmonth && nmonth <= 12 && 1 <= nday &&
      nday <= maxDaysInMonth[nmonth-1])
  { window.status = "Date is correct"; }
  else
  { window.status = "Date is not correct!"; }
}
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="date">
<STRONG>What month (1--12)?</STRONG><br>
<INPUT NAME = "month" TYPE = "text" SIZE = "2"
```

```
 MAXLENGTH = "2"></P>

<STRONG>What day (1--31)?</STRONG><br>
<INPUT NAME = "day" TYPE = "text" SIZE = "2"
 MAXLENGTH = "2"></P>

<INPUT TYPE="button"
 VALUE="Check Date" ONCLICK="checkDate()">
</FORM>
</BODY>
</HTML>
```


Interface of Date Check Program



Software Engineering Guidelines for JavaScript

Writing JavaScript program is similar to defining single Java class, so guidelines for simple Java programs apply to JavaScript:

- Structure program by dividing it into functions with clear specifications, and single unambiguous purpose.
- Limit length of functions by using helper functions.
- Factor sections of code that occur in several different places in program into helper functions.
- Name functions, variables and constants using Java conventions, and with meaningful names.
- Divide GUI into set of HTML files, each file should have clear simple purpose and name that reflects that purpose.

Site Maps

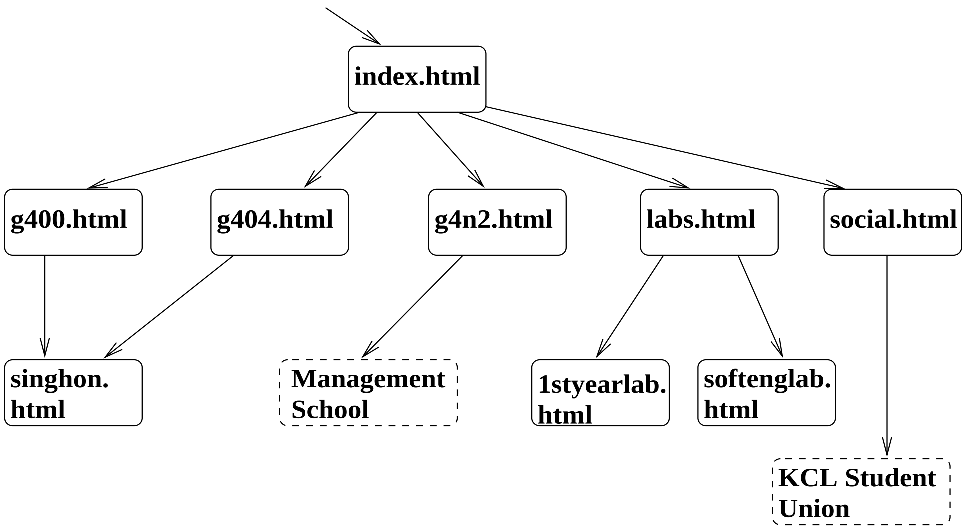
For any website can draw a graph, similar to state machine, which shows web pages as states and jumps from one page to another, via links or explicit page loading, as transitions between states.

Following shows structure of web site giving information to prospective students. States with dashed outlines denote pages on remote sites.

In conventional website such a graph shows how easy or difficult it is to navigate from one page to another.

In a web site programmed using JavaScript it shows sequence of processing states or modes that web-based GUI can go through.

Site Map of BSc Website



HTTP GET and POST Requests

Two main ways in which client side of an internet application sends information to server side using *Hypertext transfer protocol (HTTP)*:

- GET requests send information to server as part of request URL. Data values appear after ? sign in URL, for example: http://www.serv.org?param1=val1&param2=val2.
- POST requests send information to server in separate message body (not URL). Therefore more secure – data being sent not visible in browser URL display. POST requests are used for updating server data, eg, entering details from form into a database.

Another disadvantage of GET is that some browsers may place limit on length of a URL, restricting amount of data that can be appended in GET request. Limit may be as low as 255 bytes.

GET and POST

Method is specified in FORM tag in HTML. Eg, *booktable.html* file uses GET to submit restaurant booking information:

```
<HTML>
<BODY>
<FORM NAME = "tableform"
 METHOD="GET" ACTION = "http://localhost:8080/test.jsp">
....
</FORM>
</BODY>
</HTML>
```

Data is sent using GET to server-side program, a JSP file, specified in ACTION clause.

JavaServer Pages (JSP)

JSP plays same role for server side of a web application that JavaScript does for client side: adding processing capabilities to HTML pages. JSP code will generally be more complex than JavaScript – may implement middle-tier business rules.

JavaScript code is usually restricted to input validation, help facilities, page animation and visual effects, etc.

Simple example of JSP page, *test.jsp*, which echoes back parameters it receives from an HTTP request (GET or POST), is:

```
<HTML>
<HEAD><TITLE>Echo Parameters</TITLE></HEAD>
<BODY>
Thank you, <%= request.getParameter("name") %> for
your booking. You requested a table for
<%= request.getParameter("number") %> people, in a
<%= request.getParameter("smoke") %> area,
```

```
on day <%= request.getParameter("day") %>
of month <%= request.getParameter("month") %>.
</BODY>
</HTML>
```

Code within <% and %> brackets is executed on server, in contrast to JavaScript code written between <SCRIPT> and </SCRIPT> tags, which execute on client.

Directive <%= *exp*%> evaluates Java expression *exp*, placing result, as string, into web page which is returned to client as response to HTTP query. *request* is Java object representing HTTP request, *getParameter*(*att*) returns value of *att* parameter in request, as a string. Java code fragments written in a JSP page are often called *scriptlets*.

Mixing Java code fragments with HTML can lead to an incomprehensible mess, extremely difficult to maintain or debug. Usual problem of mixing GUI and back-end code.

Server side of application can use request parameters in query or update to a database on server. Typically databases are wrapped in entity beans – Java objects with *get* and *set* methods for attributes to be queried/stored by JSP code. Bean may use JDBC to access database, or files or other techniques to make data persistent.

Version, *test2.jsp*, of above JSP file using beans is:

```
<jsp:useBean id="booker" class="BookingBean"/>
<jsp:setProperty
 name="booker" property="month" param="month"/>
<jsp:setProperty
 name="booker" property="day" param="day"/>
<jsp:setProperty
 name="booker" property="name" param="name"/>
<jsp:setProperty
 name="booker" property="number" param="number"/>
<jsp:setProperty
 name="booker" property="smoking" param="smoke"/>
```

```
<HTML>
<HEAD><TITLE>Echo Parameters</TITLE></HEAD>
<BODY>
Thank you, <jsp:getProperty name="booker" property="name" /> for
your booking. You requested a table for
<jsp:getProperty name="booker" property="number" />
people, in a
<jsp:getProperty name="booker" property="smoking" /> area,
on day <jsp:getProperty name="booker" property="day" />
of month <jsp:getProperty name="booker" property="month" />.
</BODY>
</HTML>
```


This page writes and reads data to an instance, *booker*, of bean class *BookingBean*:

```
public class BookingBean
{ private int month;
  private int day;
  private String name;
  private String number;
  private boolean smoking;

  // All bean classes must provide a
  // no-argument constructor:
  public BookingBean() { }

  public void setMonth(String mon)
  { month = Integer.parseInt(mon); }

  public String getMonth()
```

```
  { return "" + month; }

  public void setDay(String d)
  { day = Integer.parseInt(d); }

  public String getDay()
  { return "" + day; }

  public void setName(String nme)
  { name = nme; }

  public String getName()
  { return name; }

  public void setNumber(String num)
  { number = num; }

  public String getNumber()
```



```
  { return number; }

  public void setSmoking(String smoke)
  { if (smoke.equals("smoking"))
    { smoking = true; }
    else
    { smoking = false; }
  }

  public boolean isSmoking()
  { return smoking; }
}
```

JSP statements in *test2.jsp* create, update and read particular object, *booker*, of *BookingBean* class:

1. $< jsp : useBean\ id = \text{“}booker\text{”}\ class = \text{“}BookingBean\text{”} / >$
   This corresponds to Java statement:

```
BookingBean booker = new BookingBean();
```

if *booker* does not already exist.

2. $< jsp : setProperty\ name = “booker”\ property = “month”\ param = “month”/ >$

   This corresponds to Java statement:

   ```
   booker.setMonth(request.getParameter("month"));
   ```

   where *request* is Java object representing HTTP request, and *getParameter* extracts value of *month* parameter of request.

3. $< jsp : getProperty\ name = “booker”\ property = “name”/ >$

   This corresponds to Java statement:

   ```
   booker.getName();
   ```

   In case of boolean properties such as *smoking*, *booker.isSmoking*() is called instead.

Beans can have four different levels of scope or persistence:

Scope	*Persistence of bean object*	*Purpose*
page	Only exists while page is loaded	local page data/ processing
request	Exists while particular request is processed. Valid in declaring page & pages it forwards request to	data related to a given request
session	Exists for duration of a browsing session, ie, connection by particular execution of browser to server	holds session data such as a customer number
application	Exists for duration of server application	application level data, such as database connection pool.

Can use normal Java conditional and loop statements within a JSP page, eg, in script, *testdate.jsp*, which checks date entered in one form before allowing user to proceed to another:

```
<jsp:useBean id="booker"
 scope="session" class="BookingBean"/>
<jsp:setProperty
 name="booker" property="month" param="month"/>
<jsp:setProperty
 name="booker" property="day" param="day"/>
<HTML>
<% if (booker.dateError())
{ out.println("<H2>Error in data: Press Back to re-enter</H2>"); }
else  {
%>
<BODY>
<FORM NAME = "tableform"
 METHOD="GET" ACTION = "http://localhost:8080/test3.jsp">
```



```
<STRONG>What is your name?</STRONG><br>
<INPUT NAME = "name" TYPE = "text" SIZE = "30"></P>

<STRONG>How many people is your booking for?</STRONG><br>
1 or 2
<INPUT NAME="number" TYPE="radio" VALUE="1 or 2">
3 to 6
<INPUT NAME="number" TYPE="radio" VALUE="3 to 6" CHECKED>
More than 6
<INPUT NAME="number" TYPE="radio" VALUE="more than 6">
<P>

<STRONG>Smoking or non-smoking?</STRONG><BR>
Non-smoking
<INPUT NAME="smoke" TYPE="radio" VALUE="non smoking" CHECKED>
Smoking
<INPUT NAME="smoke" TYPE="radio" VALUE="smoking">
```

```
<P>

<INPUT TYPE="submit" VALUE="Submit data">
</FORM>
</BODY>
<% } %>
</HTML>
```

In this JSP file switch back and forth between Java (inside <% and %> brackets) and HTML with JSP tags.

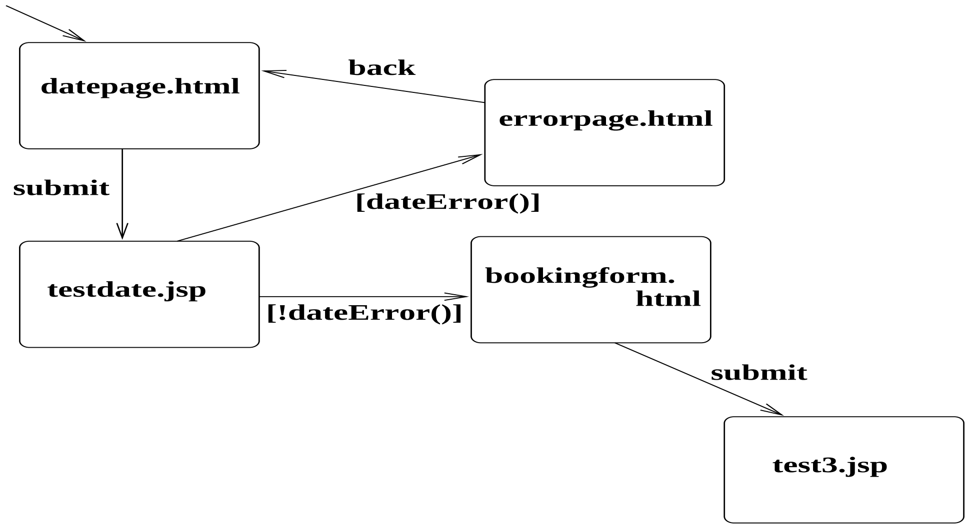
In above script *day* and *month* values are copied from HTTP request into session bean, which provides method $dateError() : boolean$ to determine if date is valid or not. If invalid, message "Error in data: Press Back to re-enter" is returned to client browser, otherwise a further HTML form, *bookingform.html*, is returned. This calls following JSP page, *test3.jsp*, when submitted:

```
<jsp:useBean id="booker"
 scope="session" class="BookingBean"/>
<jsp:setProperty
 name="booker" property="name" param="name"/>
<jsp:setProperty
 name="booker" property="number" param="number"/>
<jsp:setProperty
 name="booker" property="smoking" param="smoke"/>

<HTML>
```

```
<HEAD><TITLE>Echo Parameters</TITLE></HEAD>
<BODY>
Thank you, <jsp:getProperty name="booker" property="name" /> for
your booking. You requested a table for
<jsp:getProperty name="booker" property="number" />
people, in a
<jsp:getProperty name="booker" property="smoking"/> area,
on day <jsp:getProperty name="booker" property="day"/> of
month <jsp:getProperty name="booker" property="month"/>.
</BODY>
</HTML>
```

Bean object will not be created by this script, since it was previously created by *testdate.jsp* within same session. Instead additional parameter values are copied into it.

Control Flow of Booking Program in JSP



Installing JSP on a Web Server

Many web servers provide capability to execute JSP files, for example Apache Tomcat (http://www.apache.org), Jigsaw (http://www.w3.org/Jigsaw) and Resin (http://www.caucho.com/products/resin).

Resin is a commercial web server, which is free for educational uses such as student projects. Downloads for Windows and Unix systems are available at `http://www.caucho.com/products/resin`. Having downloaded Resin for Unix, uncompress and untar (from Unix command line) download:

```
uncompress resin-2.0.4.tar.Z
tar -xvf resin-2.0.4.tar
```

Result is a directory *resin*-2.0.4 which contains all source code, libraries, binaries and documentation. To run, Resin needs *tools.jar*

Java archive to be in class path. Resin can be started by typing *./httpd.sh* in *resin-ver/bin/* directory:

```
kcl@pc779 ~/resin-2.0.4/bin $ ./httpd.sh
Resin 2.0.4 (built Thu Nov 15 17:56:24 PST 2001)
Copyright(c) 1998-2001 Caucho Technology.  All rights reserved.

Starting Resin on Thu, 07 Feb 2002 08:09:09 +0000 (GMT+00:00)
http listening to *:8080
srun listening to 127.0.0.1:6802
```

To use JSP, place JSP files in *resin-ver/doc/* directory. *class* files of beans should be placed in *resin-ver/doc/WEB-INF/classes* directory. URL to use to connect to a JSP file *test.jsp* on current machine is `http://localhost:8080/test.jsp`, or `http://machine:8080/test.jsp` on a remote host *machine* given as an IP address or fully qualified host name.

Java Servlets

Servlets are a pure Java technique for defining server-side processing in internet applications, in contrast to hybrid HTML/Java notation of JSP. Therefore usually easier to debug and understand.

Following Servlet corresponds to *test.jsp* program. Represents controller component – servlets usually more appropriate for controllers than JSPs.

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

public class BookingServlet extends HttpServlet
{ public BookingServlet() {}
```



```
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
  throws ServletException, IOException
  { res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    String name = req.getParameter("name");
    String number = req.getParameter("number");
    String smoke = req.getParameter("likes");
    String day = req.getParameter("day");

    pw.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD " +
               "HTML 4.0 Transitional//EN\">");
    pw.println();
    pw.println("<head>");
    pw.println("<meta http-equiv=\"Content-Type\" " +
               "content=\"text/html; charset=ISO-8859-1\">");
    pw.println();
    pw.println("<title>Booking Details</title>");
```

```
    pw.println("</head>");
    pw.println();
    pw.println("<body bgcolor=#FF69CC>");
    pw.println();
    pw.println("<h1>Thank you, " + name + " for booking. </h1>");
    pw.println("<h1>You booked for: " + number + " people </h1>");
    pw.println("<h1>In a " + smoke + " area </h1>");
    pw.println("<h1>On " + day + ".</h1>");
    pw.println();
    pw.println("</body>");
    pw.close();
  }

  public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
  throws ServletException, IOException
  { doGet(req,res); }
}
```


doGet handles HTTP *GET* requests from client. *req* holds data which has been sent, and has parameters for each named input item in source form. *res* has output print writer, *pw* in above example, to which HTML text can be sent, which is servlet's response to request. *doPost* calls *doGet* as there is no need for specific processing of *POST* requests in this case.

Servlet responds to *GET* requests from following form:

```
<HTML>
<BODY>
<FORM METHOD="GET"
 ACTION="http://localhost:8080/servlet/BookingServlet">

<STRONG>What is your name?</STRONG><br>
<INPUT NAME = "name" TYPE = "text" SIZE = "30"></P>

<STRONG>How many people is your booking for?</STRONG><br>
1 or 2
```

```
<INPUT NAME="number" TYPE="radio" VALUE="1 or 2">
3 to 6
<INPUT NAME="number" TYPE="radio" VALUE="3 to 6" CHECKED>
More than 6
<INPUT NAME="number" TYPE="radio" VALUE="more than 6"> <P>

<STRONG>Smoking or non-smoking?</STRONG><BR>
Non-smoking
<INPUT NAME="likes" TYPE="radio" VALUE="non smoking" CHECKED>
Smoking
<INPUT NAME="likes" TYPE="radio" VALUE="smoking"> <P>

<STRONG>Day:</STRONG>
<SELECT NAME="day">
<OPTION SELECTED>Monday
<OPTION>Tuesday
<OPTION>Wednesday
<OPTION>Thursday
```



```
<OPTION>Friday
<OPTION>Saturday
<OPTION>Sunday
</SELECT> <P>

<INPUT TYPE="submit" VALUE="Submit data">
</FORM>
</BODY>
</HTML>
```

Directly writing HTML text within a servlet is very primitive way of dealing with requests. For more complex processing can use Java classes such as *HtmlPage* and *HtmlForm* which represent corresponding HTML constructs.

Eg, here is a class (a statefull session bean) which represents an HTML page that is shown if there is error in booking data.

```
public class ErrorPage extends HtmlPage
{ private String month;
  private String day;
  private String name;
  private String number;
  private String smoke;

  public ErrorPage(String mon, String d, String nme,
                   String numb, String sm)
  { super();
    month = mon;
```



```
    day = d;
    name = nme;
    number = numb;
    smoke = sm;
  }

  public boolean isError()
  { // checks that all fields are non-null and valid.
    if (month == null || day == null || name == null ||
        name.equals(""))
    { return true; }
    int monthint;
    int dayint;
    try
    { monthint = Integer.parseInt(month);
      dayint = Integer.parseInt(day);
    }
    catch (Exception e) { return true; }
```

```
    return dateError(monthint,dayint);
  }

  private boolean dateError(int mon, int d)
  { /* true if mon, d invalid date, false otherwise */ }

  public void buildPage()  // if an error
  { HtmlHead head = new HtmlHead("Errors in Booking");
    HtmlBody body = new HtmlBody();
    HtmlText title = new HtmlText("Errors in Booking", "H2");
    HtmlText mess =
      new HtmlText("Press Back and re-enter data in form","P");
    body.add(title);
    if (month == null)
    { HtmlText monthrep =
        new HtmlText("Month not specified!", "P");
      body.add(monthrep);
    }
```



```
    if (day == null)
    { HtmlText dayrep =
        new HtmlText("Day not specified!", "P");
      body.add(dayrep);
    }
    if (name == null || name.equals(""))
    { HtmlText namerep =
        new HtmlText("Name not specified!", "P");
      body.add(namerep);
    }
    body.add(mess);
    setHead(head);
    setBody(body);
  }
}
```

This is used in following version of booking servlet:

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

public class BookingServlet2 extends HttpServlet
{ public BookingServlet2() {}

  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
  throws ServletException, IOException
  { res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    String month = req.getParameter("month");
    String day = req.getParameter("day");
    String name = req.getParameter("name");
```



```
    String number = req.getParameter("number");
    String smoke = req.getParameter("likes");

    pw.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD " +
               "HTML 4.0 Transitional//EN\">");
    pw.println();
    ErrorPage ep =
      new ErrorPage(month,day,name,number,smoke);
    if (ep.isError())
    { ep.buildPage();
      pw.println(ep.getHtml());
    }
    else
    { pw.println("<head>");
      pw.println("<meta http-equiv=\"Content-Type\" " +
                 "content=\"text/html; charset=ISO-8859-1\">");
      pw.println();
      pw.println("<title>Booking Details</title>");
```

```
      pw.println("</head>");
      pw.println("<body bgcolor=#FF69CC>");
      pw.println();
      pw.println("<h1>Thank you, " + name +
                 " for your booking. </h1>");
      pw.println("<h1>You booked for: " + number +" people</h1>");
      pw.println("<h1>In a " + smoke + " area </h1>");
      pw.println("<h1>On day " + day + ".</h1>");
      pw.println();
      pw.println("</body>");
    }
    pw.close();
  }
}
```

This servlet displays error report if there is some mistake in data, or displays a confirmation page.

HtmlPage is defined in outline as:

```
public class HtmlPage implements HtmlComponent
{ private HtmlHead head = new HtmlHead(" ");
  private HtmlBody body = new HtmlBody();

  public HtmlPage() { }

  public void setHead(HtmlHead h) { head = h; }

  public void setBody(HtmlBody b) { body = b; }

  public String getHtml()
  { return "<HTML> \n" + head.getHtml() + "\n" +
           body.getHtml() + "\n </HTML>";
  }
}
```

```
public interface HtmlComponent
{ public String getHtml(); }
```

is interface declaring *getHtml* method common to all HTML components. *HtmlHead* represents header of an HTML document:

```
public class HtmlHead implements HtmlComponent
{ private String title = "";

  public HtmlHead(String t)
  { title = t; }

  public String getHtml()
  { return "<HEAD> <TITLE> " + title +
           " </TITLE> </HEAD>";
  }
}
```

HtmlBody represents body of an HTML page as sequence of HTML components.


```
import java.util.Vector;

public class HtmlBody implements HtmlComponent
{ private Vector contents = new Vector();
  /* Of HtmlComponent */

  public void add(HtmlComponent c)
  { contents.add(c); }

  public String getHtml()
  { String res = "";
    for (int i = 0; i < contents.size(); i++)
    { HtmlComponent c = (HtmlComponent) contents.get(i);
      res = res + "\n " + c.getHtml();
    }
    return res;
  }
}
```

Individual HTML elements such as anchors or text can be defined in same way:

```
public class HtmlText implements HtmlComponent
{ private String text = "";
  private String format = "P";

  public HtmlText(String s, String f)
  { text = s;
    format = f;
  }

  public String getHtml()
  { return "<" + format + "> " + text + " </" +
              format + "> ";
  }
}
```


Managing Sessions

A servlet can record and read information concerned with particular client session, ie, interaction between servlet and particular client, by creating an *HttpSession* object. Eg, to record someone's login id from opening login prompt of a website, do:

```
String userid = request.getParameter("userid");
HttpSession session = request.getSession(true);
session.putValue("login",userid);
```

in *doPost* method that receives request from login prompt. Parameter *true* in *getSession* indicates that session object should be created for client (for whom *doPost* is being executed) if one does not already exist for client. If a session already exists, it is returned, otherwise new session is returned.

Session can be referred to in another method, and its values extracted:

```
HttpSession session = request.getSession(false);
if (session != null)
{ String userid = (String) session.getValue("login");
  ...
}
```

If this code is executed within same browsing session, eg, by client submitting data from another form, then session object exists and login value stored by initial request is returned. Can be used to check authorisation for taking an action or giving information.

false parameter in *getSession* indicates that new session should not be created if one does not already exist, instead *null* is returned.

Three Tier Applications

Database connections can be shared between several requests, so can be placed in servlet instance variables. Connection could be created in *init* method of servlet, executed when servlet is initialised by server, before any client requests are handled. Each client request can use shared database connection to read or write data, although only one request should use connection at a time, ie, database update and enquiry methods should be declared as *synchronized*.

Code for establishing and operating on database connection should be encapsulated in separate class to servlet, to separate database tier from business logic tier.

Example Three Tier Application

Example of construction of three tier application is restaurant booking, uses database table *Booking* which stores relevant attributes:

bookingid	*name*	*month*	*day*	*number*	*smoke*
1	"Felix Ali"	12	18	1	Y
2	"Linda Pel"	11	22	3	N

Use *PostgreSQL* database management system to construct database to hold table, and use JDBC to link middle tier servlet classes to database.

PostgreSQL is open source, free DBMS which provides similar capabilities to commercial databases such as Oracle or Access. Runs on Linux.

Following steps are needed to set up a PostgreSQL database installation for first time:

1. Create directory *pgdata* in your home directory to hold database files, then *cd* to this directory.
2. Run *initdb*.

Once this settup is complete, database server process can be started to deal with database creation, queries and updates:

```
postmaster -i > server.log &
```

To create a database called *bookingdb*, run:

```
createdb bookingdb
```

Tables of database can be defined using interactive *psql* program:

```
kcl@pc779 ~/pgdata $ psql bookingdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

bookingdb=# CREATE TABLE booking
bookingdb-# (bookingid INT2,
bookingdb(#  name CHAR(30),
bookingdb(#  month CHAR(3),
bookingdb(#  day CHAR(2),
bookingdb(#  number CHAR(10),
bookingdb(#  smoke CHAR(1),
```



```
bookingdb(# PRIMARY KEY (bookingid));
```

This SQL statement creates table called *booking* in *bookingdb* database. Can view its structure by the psql *d* command:

```
bookingdb=# \d booking
         Table "booking"
 Attribute |   Type   | Modifier
-----------+----------+----------
 bookingid | smallint | not null
 name      | char(30) |
 month     | char(3)  |
 day       | char(2)  |
 number    | char(10) |
 smoke     | char(1)  |
Index: booking_pkey
```

Rows can be added to *booking* using SQL INSERT command:

```
bookingdb=#  INSERT INTO booking
bookingdb-# VALUES(001,'Felix Ali','12','18','1','Y');
INSERT 18801 1
```

Table can be viewed by using *SELECT* $*$ *FROM booking*; as usual. All commands are standard SQL commands, as we used with Access in Part 3.

To quit from *psql* type \\q. To shut down database server, identify its Unix process ID by using *ps*, then type *kill id* where *id* is ID of *postmaster* process.

To connect to PostgreSQL database from Java programs, archive *postgresql.jar* needs to be in CLASSPATH. Provides database driver for PostgreSQL. An interface class (an entity bean) which uses driver to connect to above database can be defined as:

```
import java.io.*;
import java.sql.*;

public class BookingDBI
{ private Connection conn;
  private Statement stat;

  /** Construct a database connection and
      statement for "bookingdb" with username "kcl" */
  public BookingDBI()
```

```
{ try
  { Class.forName("org.postgresql.Driver");
    conn =
       DriverManager.getConnection(
             "jdbc:postgresql:bookingdb", "kcl", "");
    stat = conn.createStatement();
  }
  catch (Exception e)
  { e.printStackTrace(); }
}

/** Add a new booking to the database. */
public synchronized void addBooking(String name,
                           String month, String day,
                           String number, String smoke)
{ int maxId = getMaxId();
  String numb;
  String sm;
```



```
  maxId++;

  if (number.equals("1 or 2"))
  { numb = "1"; }
  else if (number.equals("3 to 6"))
  { numb = "3"; }
  else
  { numb = "6"; }

  if (smoke.equals("non smoking"))
  { sm = "N"; }
  else
  { sm = "Y"; }

  try
  { stat.executeUpdate("INSERT INTO booking " +
              "VALUES (" + maxId + ",'" + name + "','" +
               month + "','" + day + "','" +
```

```
                 numb + "','" + sm + "')");
      conn.commit();
    }
    catch (Exception e)
    { e.printStackTrace(); }
  }

  private int getMaxId()  // Returns max primary key
  { int maxId = 0;        // value used in database.
    try
    {
      String sQuery = "SELECT MAX(bookingid) " +
                      "AS MaxId FROM booking";
      ResultSet res = stat.executeQuery(sQuery);
      while (res.next())
      { maxId = res.getInt("MaxId"); }
    }
    catch(SQLException e)
```



```
    { System.err.println("Error in getting max id");
      e.printStackTrace();
    }
    return maxId;
  }

  /** Close connection to database. */
  public synchronized void logoff()
  { try
    { stat.close();
      conn.close();
    }
    catch (Exception e)
    { e.printStackTrace(); }
  }
}
```

In a professional application, name of database, table and database driver should not be hard-coded into database interface class, but instead defined in a separate *configuration file*.

Above interface can be used by booking servlet to record bookings:

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

public class BookingServlet extends HttpServlet
{ private BookingDBI dbi; // Interface to database

  public BookingServlet() {}

  public void init(ServletConfig cfg) throws ServletException
  { super.init(cfg);
    dbi = new BookingDBI();
```



```
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  { res.setContentType("text/html");
    PrintWriter pw = res.getWriter();
    String name = req.getParameter("name");
    String month = req.getParameter("month");
    String number = req.getParameter("number");
    String smoke = req.getParameter("smoke");
    String day = req.getParameter("day");
    pw.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0" +
               " Transitional//EN\">");
    pw.println("<head>");
    pw.println("<meta http-equiv=\"Content-Type\" " +
               "content=\"text/html; charset=ISO-8859-1\">");
    ErrorPage ep = new ErrorPage(month,day,name,number,smoke);
    if (ep.isError())
```

```
    { ep.buildPage();
      pw.println(ep.getHtml());
      pw.close();
      return;
    }
    pw.println("<title>Booking Details</title>");
    pw.println("</head>");
    pw.println("<body bgcolor=#FF69CC>");
    pw.println();
    pw.println("<h1>Thank you, " + name +
               " for your booking. </h1>");
    pw.println("<h1>You booked for: " + number + " people </h1>");
    pw.println("<h1>In a " + smoke + " area </h1>");
    pw.println("<h1>On day " + day + " of month " +
               month + ".</h1>");
    pw.println();
    pw.println("<HR>");
    if (dbi != null)
```



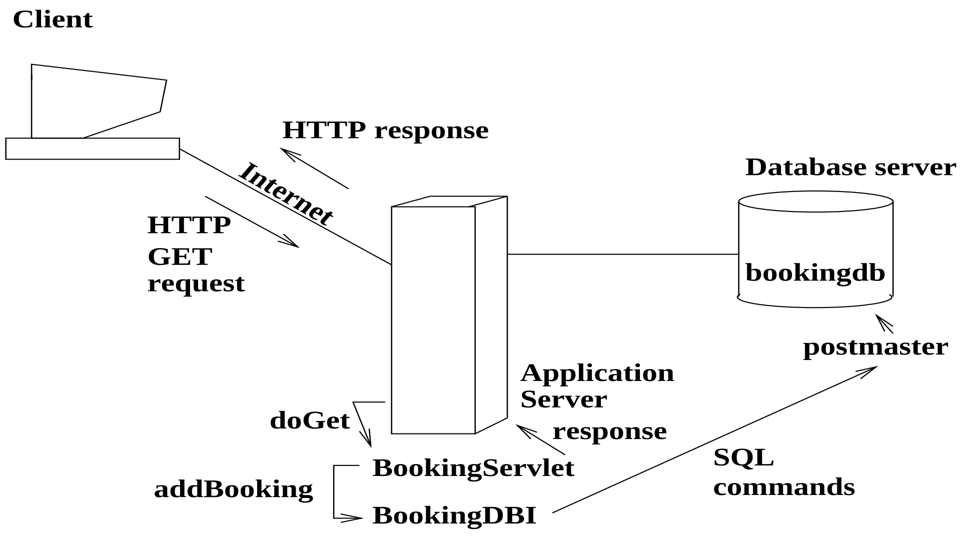
```
    { dbi.addBooking(name,month,day,number,smoke);
      pw.println("<H2>Booking added to database</H2>");
    }
    else
    { pw.println("<H2>Booking NOT added to database</H2>"); }
    pw.println("</body>");
    pw.close();
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
  { doGet(req,res); }

  public void destroy()
  { dbi.logoff(); }
}
```

init method establishes database connection, *destroy* closes it down. In more intensively-used applications where may be hundreds of client sessions running concurrently, database connections would need to be acquired from a connection pool when needed by a client and released once database operation is complete.

To make all this work, need three external pieces of software running at same time:

1. Database server needs to be running, ie, *postmaster* process needs to be invoked as above.
2. Web server such as Resin needs to be running, with *BookingServlet.java* and *BookingDBI.java* files placed in appropriate *WEB-INF/classes* directory.
3. A web browser is needed to view the web pages.

Structure of Example 3-Tier Application



Installing Servlets

To install servlet in Resin, simply place *java* files of servlet in *resin-ver/doc/WEB-INF/classes* directory.

Comparing JSP and Servlets

Property	*JSP*	*Servlets*
Efficiency	Delay when first compiled	No compilation delay
Separation of HTML and code	By using beans	By using *HtmlPage*, *HtmlForm*, etc
Debugging	Very difficult	Using Java tools
Structuring	Using files and server-side includes.	Using classes.

Design Patterns for Internet Applications

Internet applications have specific patterns, such as connection pooling for database connections, related to proxy and singleton patterns; guest books; session tracking; and shopping carts.

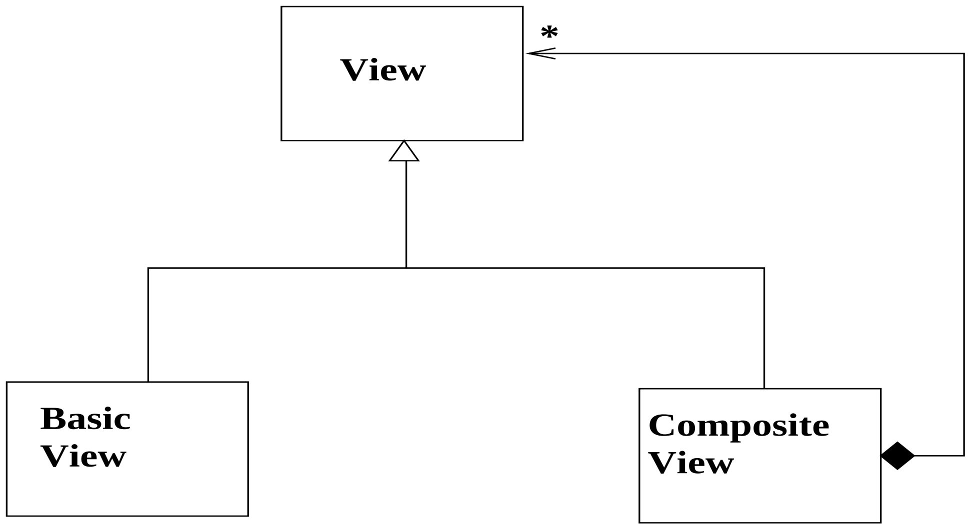
Patterns for J2EE framework include:

Front Controller (presentation tier): Centralises handling of client requests in a single component (usually a servlet) instead of providing multiple entry points to system, eg, by multiple JSP pages directly referenced in web page ACTION attributes. Avoids duplication of code for authentication of client, etc. Front controller component may delegate choice of view presented to client in response to request to a dispatcher.

Composite View (presentation tier): View construction using composition of subviews, such as separate parts of web page which provide independent information groups (news headlines, a sidebar with navigation links, share price alerts, etc, in share management system). Composition can be carried out using JSP server-side includes, or by automated generation of HTML page fragments using *HtmlBody* and related classes.

Value Object (business tier): A value object is a group of data returned from integration tier, which will be used by business tier. Providing a group of data, such as complete booking record, avoids need for expensive multiple query operations to obtain individual attributes.

Composite View Pattern



Revision Topics

1. Two and Three-tier architectures; describing a module decomposition for a new system
2. UML class diagrams and statecharts – writing these for simple systems
3. Operation specifications – adding EFFECTS, REQUIRES to a method
4. Design patterns: all those introduced in the course – explanation of purpose of pattern; diagram; selecting and applying pattern to particular system
5. Internet systems: explaining difference between server and client-side techniques, comparing Servlets versus JSP, etc.

Details of code are not needed.