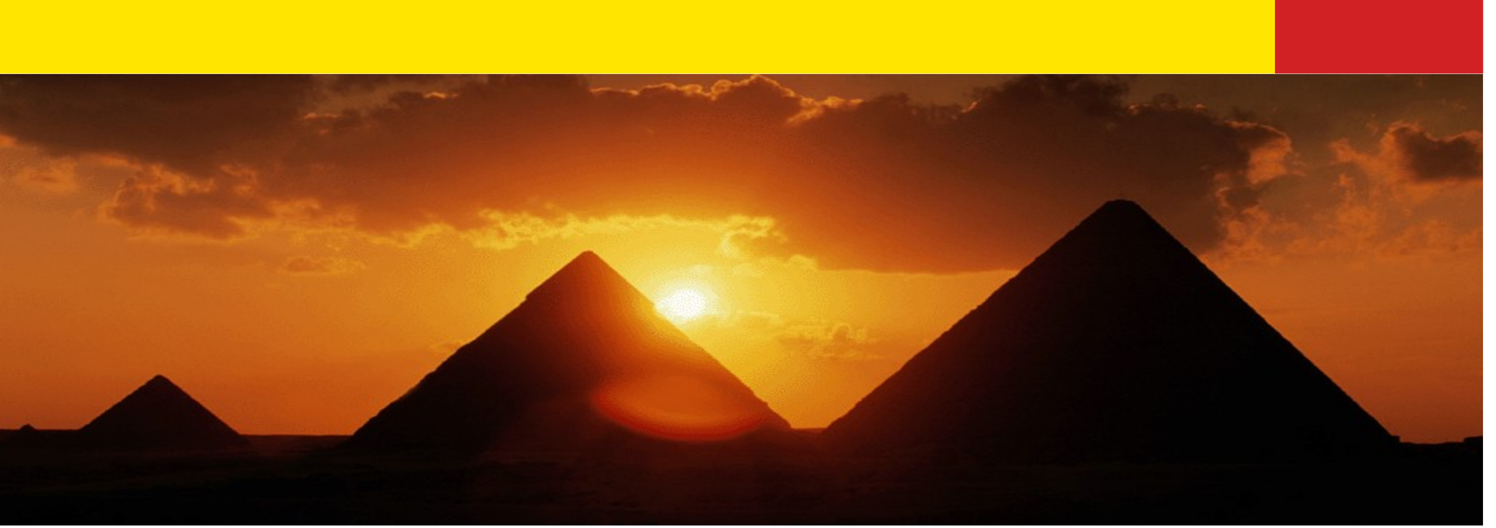
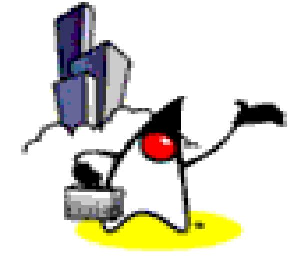


Java Server Pages (JSP) Basics

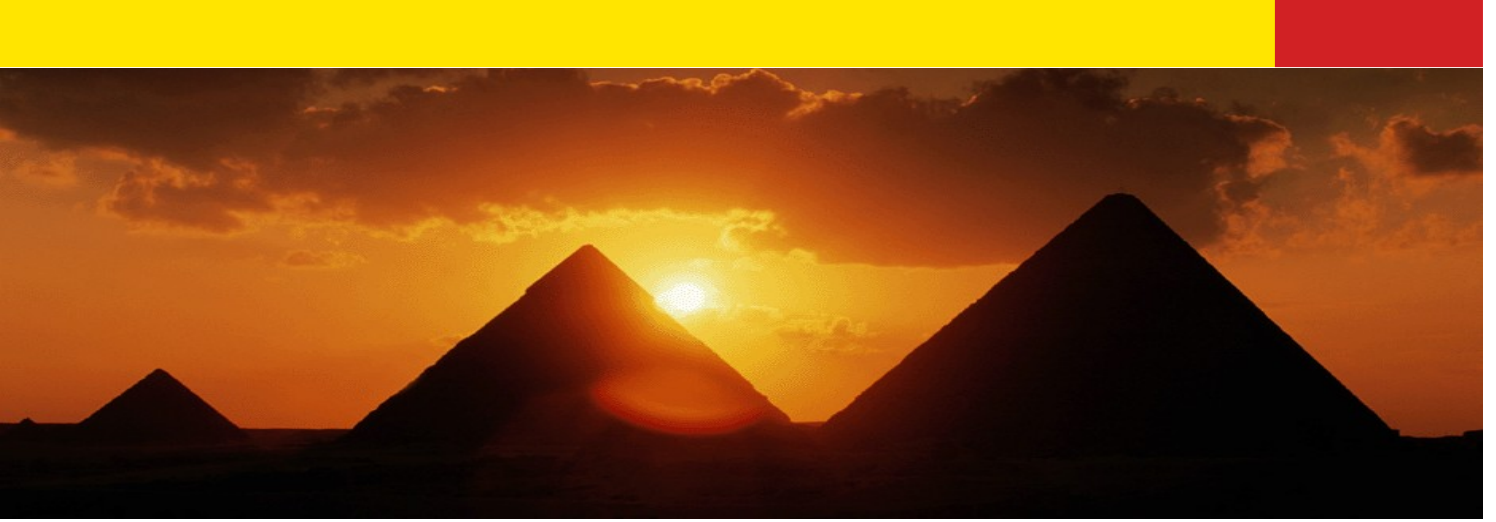


Disclaimer & Acknowledgments

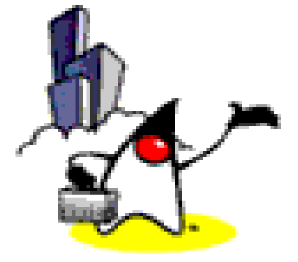
- Even though Sang Shin and Sameer Tyagi are full-time employees of Sun Microsystems, the contents here are created as their own personal endeavor and thus does not reflect any official stance of Sun Microsystems.
- Sun Microsystems is not responsible for any inaccuracies in the contents.
- Acknowledgments
  - Large portion of slides, speaker notes, and example code of this presentation are created from "JSP" section of Java WSDP tutorial written by Stephanie Bodoff of Sun Microsystems
  - Many slides are borrowed from "Servlet/JSP" codecamp material authored by Doris Chen of Sun Microsystems
  - Some example codes and contents are borrowed from "Core Servlets and JavaServer Pages" book written by Marty Hall

Revision History

- 12/24/2002: version 1 created by Sameer Tyagi
- 01/12/2003: version 2 revised by Sang Shin with speaker notes
- 05/13/2003: version 3 revised (Sang Shin)



Sub-topics of JSP



Sub-topics of JSP

- JSP Basic
- JavaBeans for JSP
- Custom tags
- JSP Standard Tag Library (JSTL)

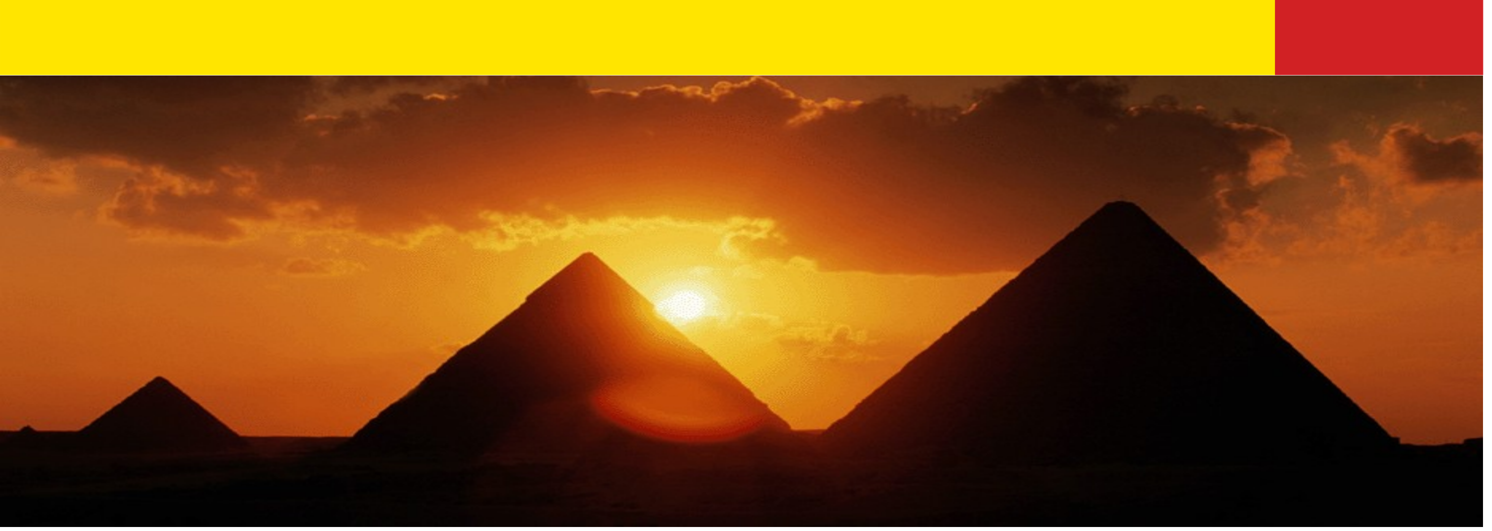
We will deal with only the first two topics in this session

Advanced Topics of Servlet/JSP

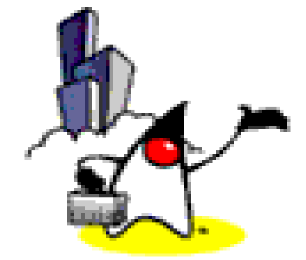
- MVC (Model-View-Controller) Pattern
- Model 2 Framework
- Apache Struts
- Java Server Faces (JSF)
- Other frameworks

Agenda

- JSP in big picture of Java EE
- Introduction to JSP
- Life-cycle of JSP page
- Steps for developing JSP-based Web application
- Dynamic contents generation techniques in JSP
- Invoking Java code using JSP scripting elements
- JavaBeans for JSP
- Error handling



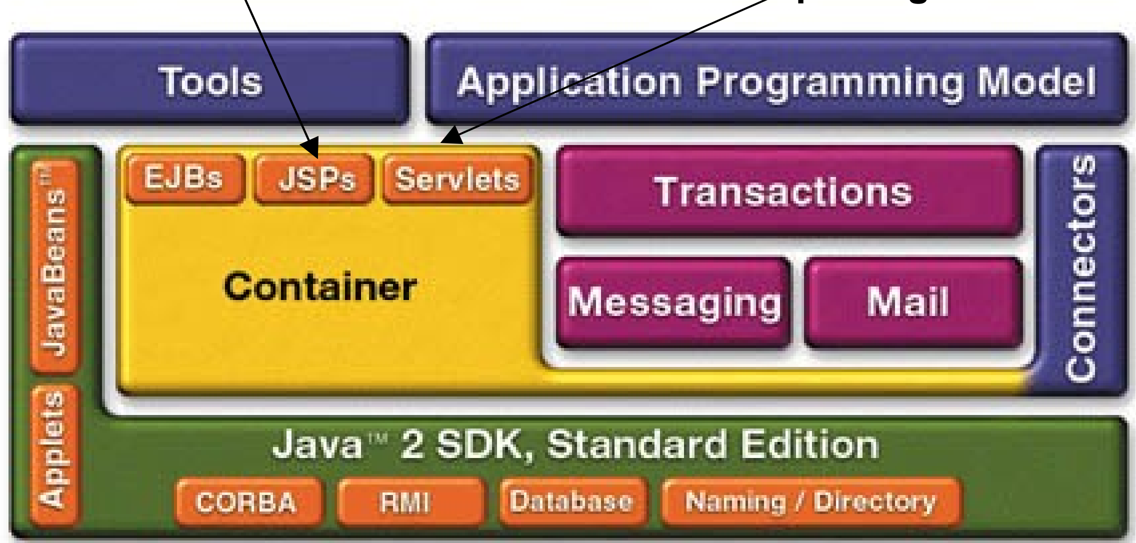
JSP in a Big Picture of Java EE



JSP & Servlet in Java EE Architecture

An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to **return dynamic content to a client**. Typically the template data is HTML or XML elements. The client is often a **Web browser**.

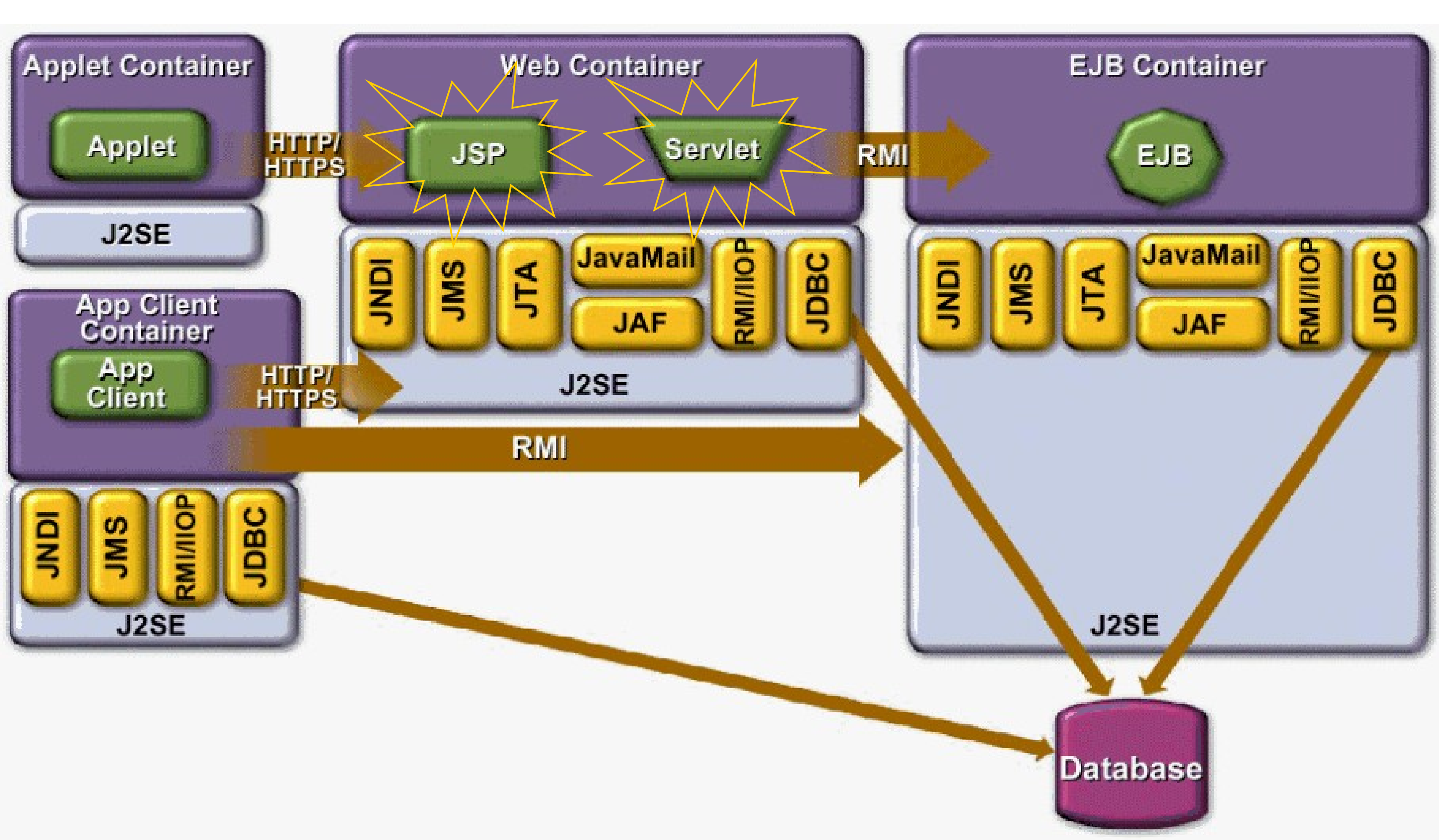
Java Servlet A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a **request-response paradigm**.



What do you mean by Static & Dynamic Contents?

- Static contents
  - Typically static HTML page
  - Same display for everyone
- Dynamic contents
  - Contents is dynamically generated based on conditions
  - Conditions could be
    - User identity
    - Time of the day
    - User entered values through forms and selections
  - Examples
    - Etrade webpage customized just for you, my Yahoo

JSP & Servlet as Web Components



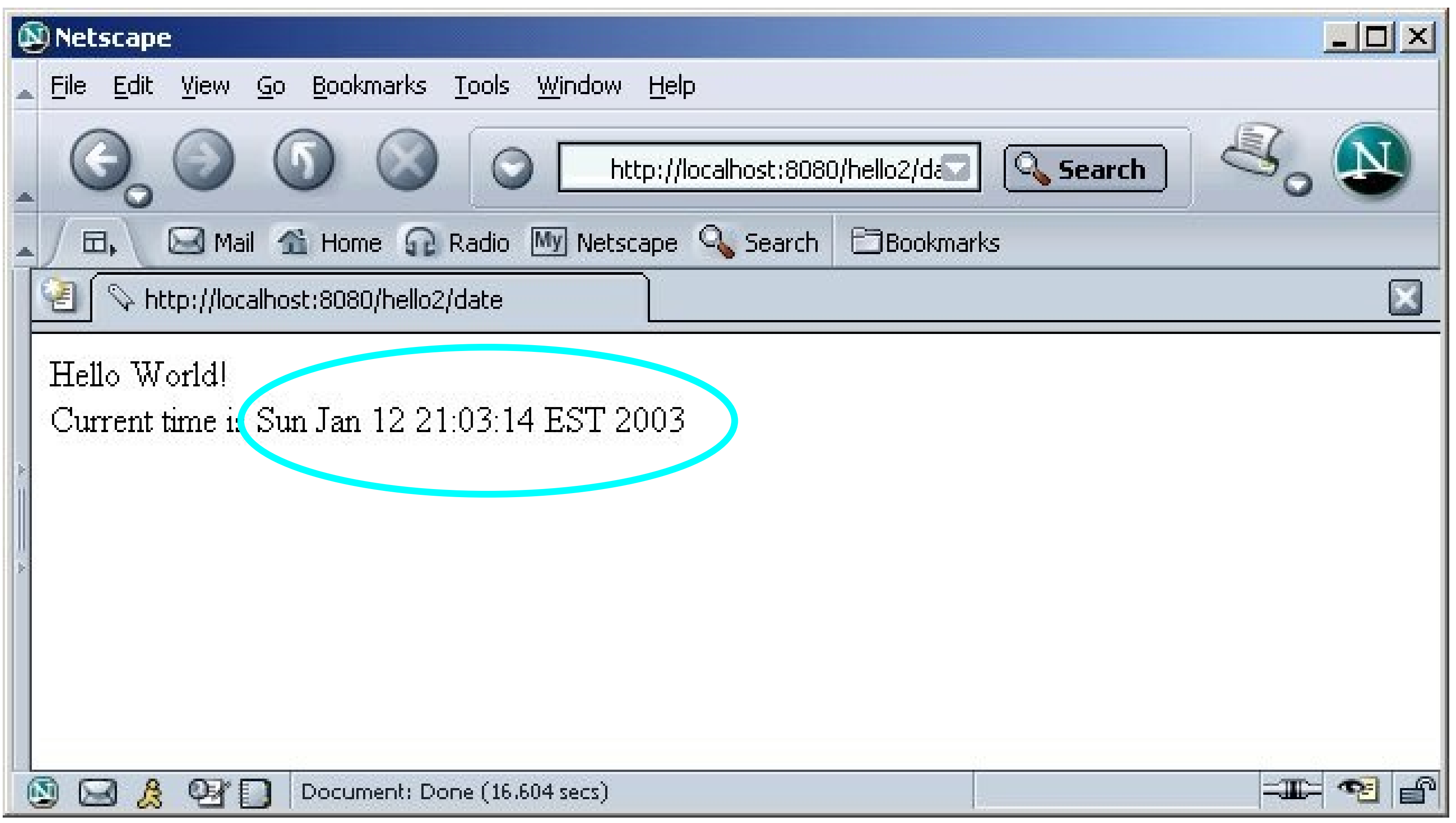
What is JSP Page?

- A text-based document capable of returning both static and dynamic content to a client browser
- Static content and dynamic content can be intermixed
- Static content
  - HTML, XML, Text
- Dynamic content
  - Java code
  - Displaying properties of JavaBeans
  - Invoking business logic defined in Custom tags

A Simple JSP Page
(Blue: static, Red: Dynamic contents)

```
<html>
<body>
   Hello World!
  <br>
  Current time is <%= new java.util.Date() %>
</body>
</html>
```

Output



Servlets

- HTML code in Java
- Not easy to author

JSP

- Java-like code in HTML
- Very easy to author
- Code is compiled into a servlet

JSP Benefits

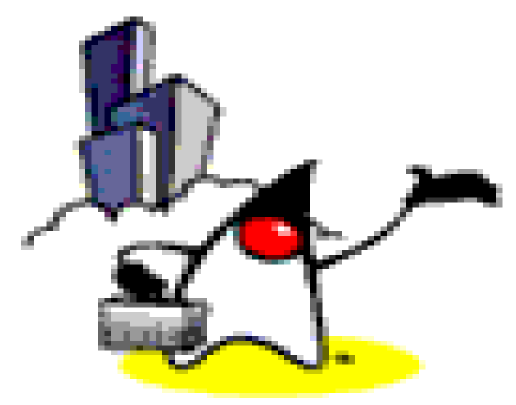
- Content and display logic are separated
- Simplify web application development with JSP, JavaBeans and custom tags
- Supports software reuse through the use of components (JavaBeans, Custom tags)
- Automatic deployment
  - Recompile automatically when changes are made to JSP pages
- Easier to author web pages
- Platform-independent

Why JSP over Servlet?

- Servlets can do a lot of things, but it is pain to:
  - Use those println() statements to generate HTML page
  - Maintain that HTML page
- No need for compiling, packaging,CLASSPATH setting

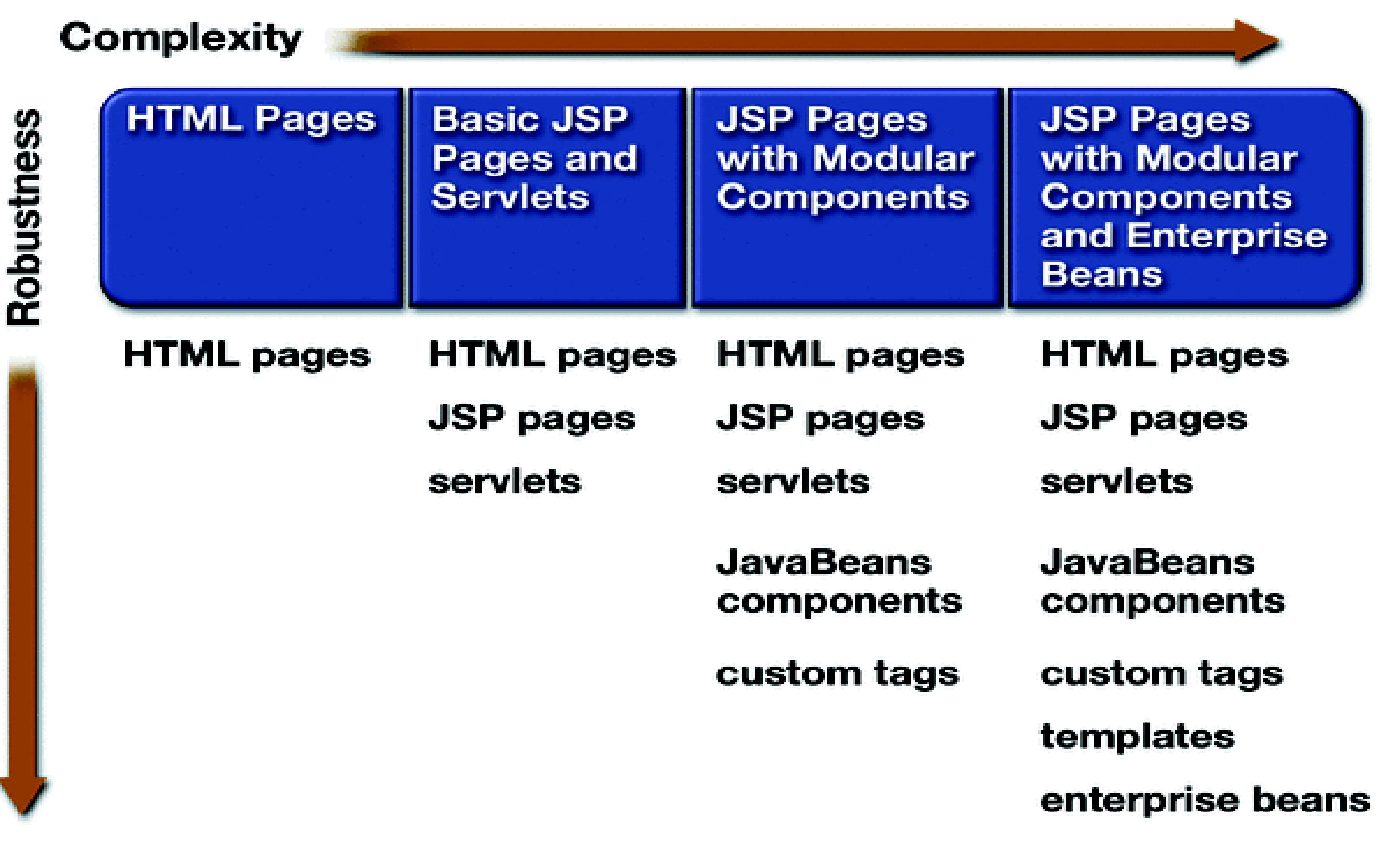
Do I have to Use JSP over Servlet or vice-versa?

- No, you want to use both leveraging the strengths of each technology
  - Servlet's strength is “controlling and dispatching”
  - JSP's strength is “displaying”
- In a typically production environment, both servlet and JSP are used in a so-called MVC (Model-View-Controller) pattern
  - Servlet handles controller part
  - JSP handles view part



JSP Architecture

Web Application Designs



Separate Request processing From Presentation

Pure Servlet

```
Public class OrderServlet…{
   public void doGet(…){
      if(isOrderValid(req)){
         saveOrder(req);

         out.println("<html>");
         out.println("<body>");
            ……
   private void isOrderValid(….){
   ……
   }

   private void saveOrder(….){
   ……
   }
}
```

Request processing

Servlet

```
Public class OrderServlet…{
   public void doGet(…){
      ……
      if(bean.isOrderValid(..)){
         bean.saveOrder(….);

         forward("conf.jsp");
   }
}
```

presentation

JSP

```
<html>
   <body>
    <ora: loop name ="order">
       …..
    </ora:loop>
   <body>
</html>
```

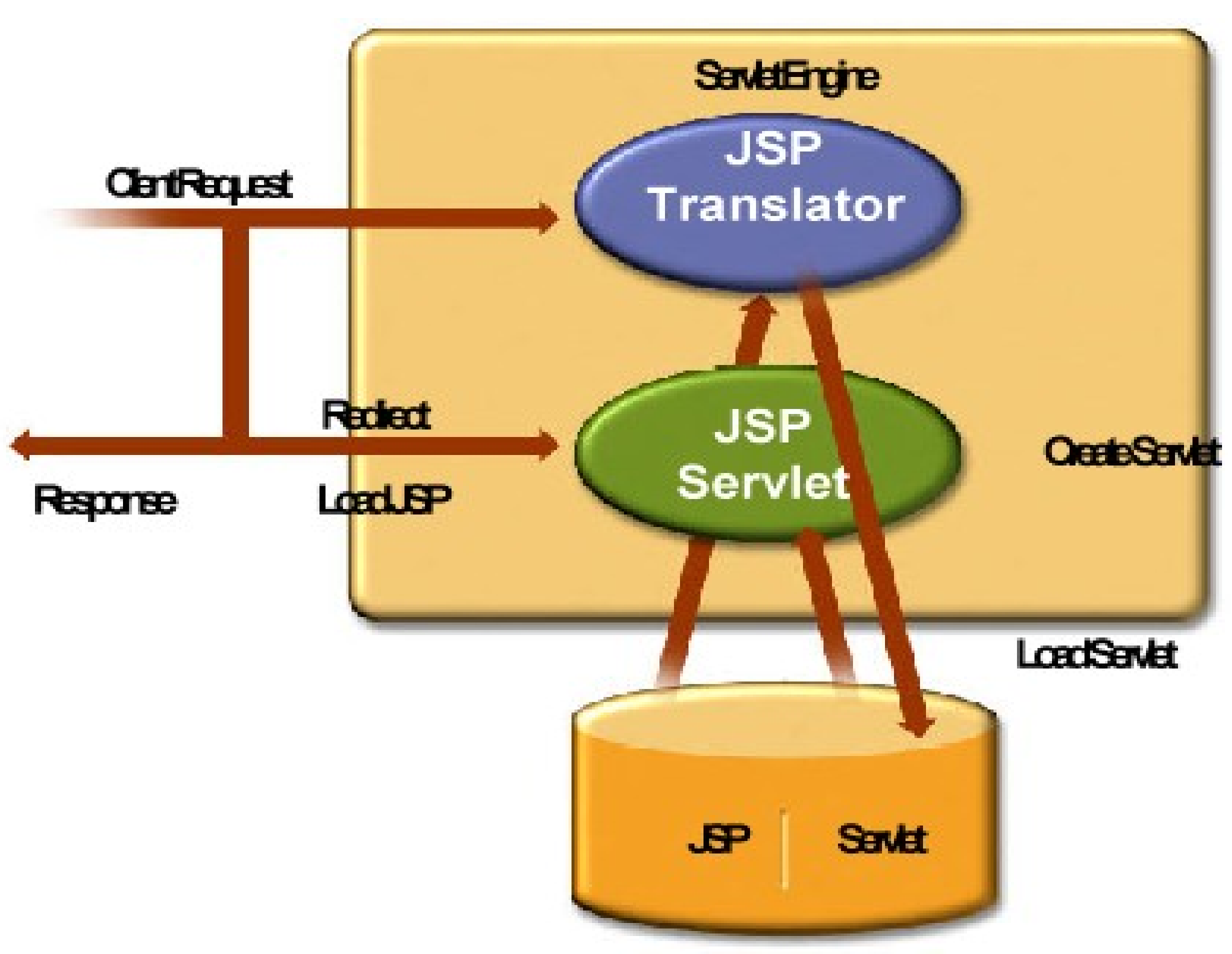
Business logic

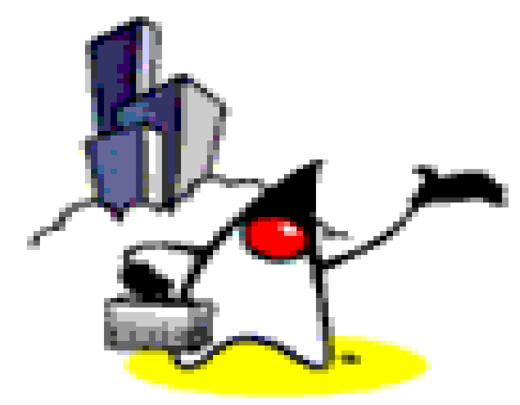
JavaBeans

isOrderValid( )

saveOrder( )

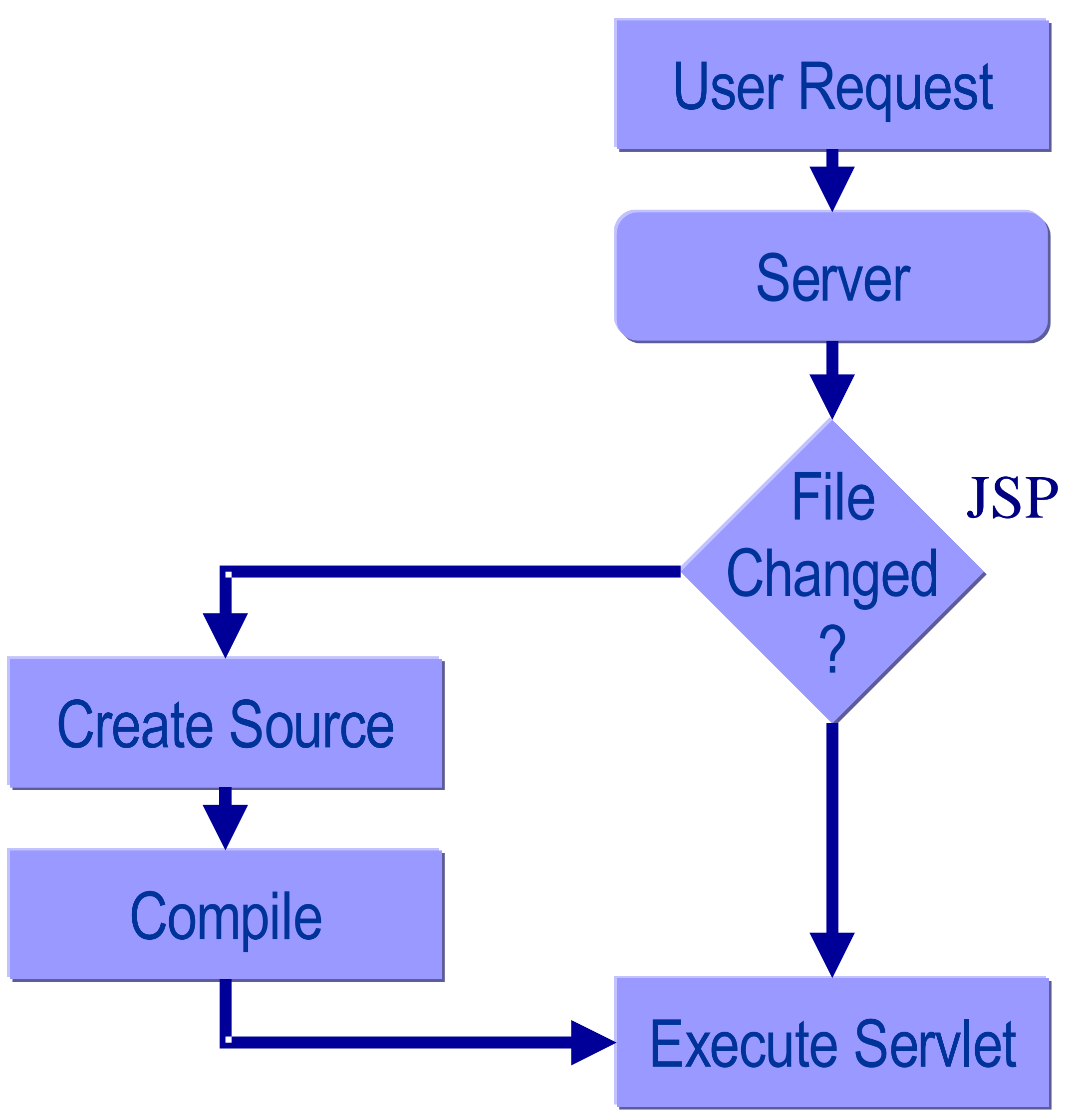
JSP Architecture





Life-Cycle of a JSP Page

How Does JSP Work?



JSP Page Lifecycle Phases

- Translation phase
- Compile phase
- Execution phase

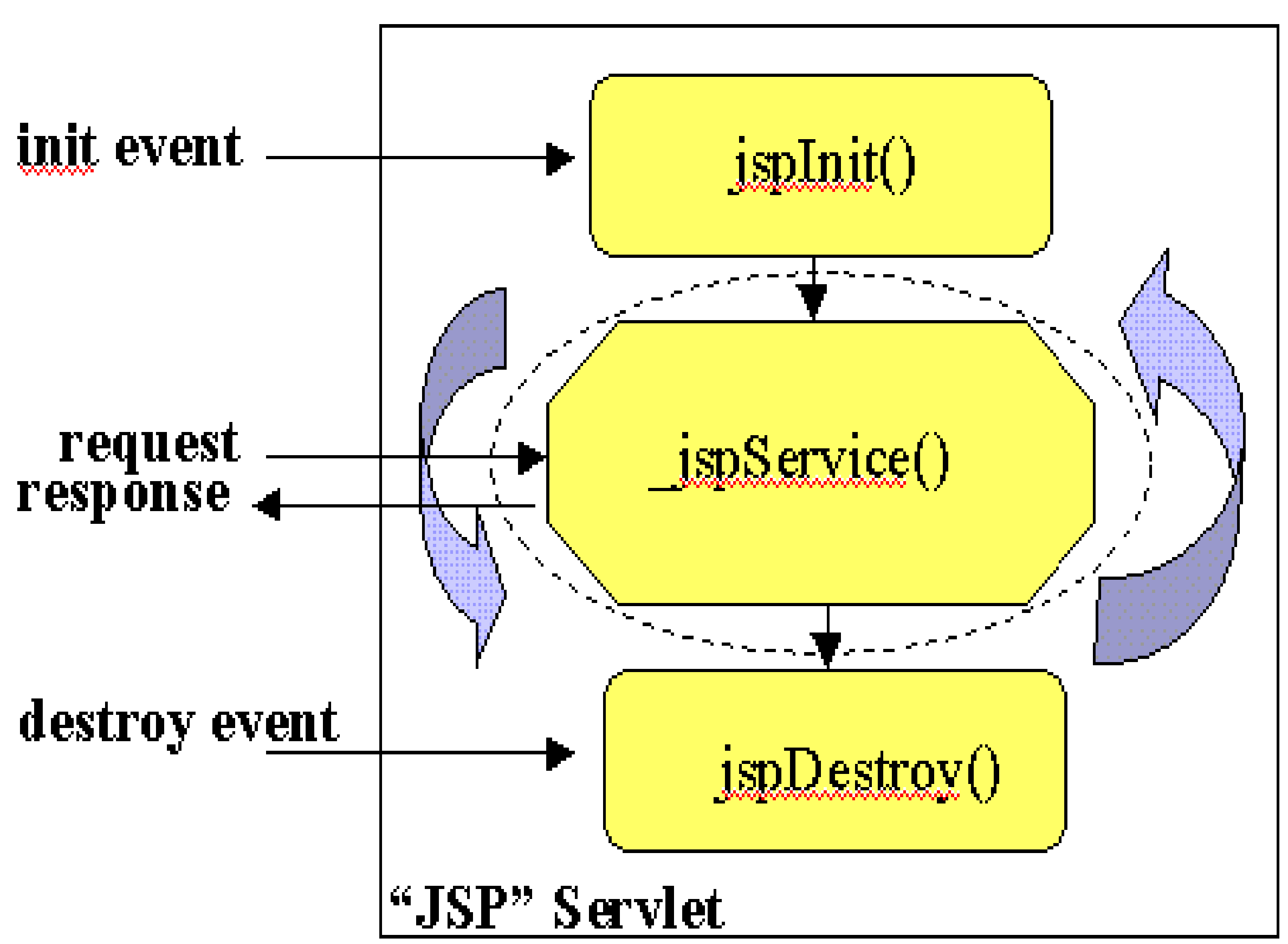
Translation/Compilation Phase

- JSP files get translated into servlet source code, which is then compiled
- Done by the container automatically
- The first time JSP page is accessed after it is deployed (or modified and redeployed)
- For JSP page “pageName”, the source code resides
  - <AppServer_HOME>/work/Standard Engine/localhost/context_root/pageName$jsp.java
  - <AppServer_HOME>/work/Standard Engine/localhost/date/index$jsp.java

Translation/Compilation Phase

- Static Template data is transformed into code that will emit data into the stream
- JSP elements are treated differently
  - Directives are used to control how Web container translates and executes JSP page
  - Scripting elements are inserted into JSP page's servlet class
  - Elements of the form <jsp:xxx .../> are converted into method calls to JavaBeans components

JSP Lifecycle Methods during Execution Phase



Initialization of a JSP Page

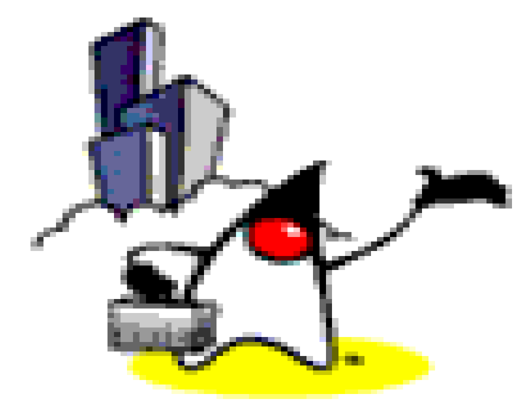
- Declare methods for performing the following tasks using JSP declaration mechanism
  - Read persistent configuration data
  - Initialize resources
  - Perform any other one-time activities by overriding jspInit() method of JspPage interface

Finalization of a JSP Page

- Declare methods for performing the following tasks using JSP declaration mechanism
  - Read persistent configuration data
  - Release resources
  - Perform any other one-time cleanup activities by overriding jspDestroy() method of JspPage interface

Example: initdestroy.jsp

```
<%@ page import="database.*" %>
<%@ page errorPage="errorpage.jsp" %>
<%-- Declare initialization and finalization methods using JSP declaration --
  %>
<%!

 private BookDBAO bookDBAO;
 public void jspInit() {

  // retrieve database access object, which was set once per web
   application
  bookDBAO =
   (BookDBAO)getServletContext().getAttribute("bookDB");
  if (bookDBAO == null)
     System.out.println("Couldn't get database.");
 }

 public void jspDestroy() {
  bookDBAO = null;
 }
```



Steps for Developing JSP-based Web Application

Web Application Development and Deployment Steps

1. Write (and compile) the Web component code (Servlet or JSP) and helper classes referenced by the web component code
2. Create any static resources (for example, images or HTML pages)
3. Create deployment descriptor (web.xml)
4. Build the Web application (*.war file or deployment-ready directory)
5. Install or deploy the web application into a Web container
   - Clients (Browsers) are now ready to access them via URL

1. Write and compile the Web component code

- Create development tree structure
- Write either servlet code and/or JSP pages along with related helper code
- Create build.xml for Ant-based build (and other application life-cycle management) process

Development Tree Structure

- Keep Web application source separate from compiled files
  - facilitate iterative development
- Root directory
  - build.xml: Ant build file
  - context.xml: Optional application configuration file
  - src: Java source of servlets and JavaBeans components
  - web: JSP pages and HTML pages, images

Example: Hello1 Example Tree Structure (before “ant build” command)

- hello1 directory (from Java EE 1.4 tutorial)
  - web directory
    - duke.waving.gif
    - index.jsp
    - response.jsp
    - WEB-INF directory
      - web.xml
  - build.xml
  - (it does not have src directory since this does not use any Java classes as utility classes)

2. Create any static resources

- HTML pages
  - Custom pages
  - Login pages
  - Error pages
- Image files that are used by HTML pages or JSP pages
  - Example: duke.waving.gif

3. Create deployment descriptor (web.xml)

- Deployment descriptor contains deployment time runtime instructions to the Web container
  - URN that the client uses to access the web component
- Every web application has to have it

web.xml for Hello1

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
  Application 2.3//EN' 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <display-name>Hello2</display-name>
  <description>no description</description>
  <servlet>
    <servlet-name>greeting</servlet-name>
    <display-name>greeting</display-name>
    <description>no description</description>
    <jsp-file>/greeting.jsp</jsp-file>   <!-- what gets called -->
  </servlet>
  <servlet-mapping>
    <servlet-name>greeting</servlet-name>
    <url-pattern>/greeting</url-pattern>  <!-- URL from browser --
  >
  </servlet-mapping>
</web-app>
```

4. Build the Web application

- Either *.WAR file or unpacked form of *.WAR file
- Build process is made of
  - create build directory (if it is not present) and its subdirectories
  - copy *.jsp files under build directory
  - compile Java code into build/WEB-INF/classes directory
  - copy web.xml file into build/WEB-INF directory
  - copy image files into build directory

Example: Hello2 Tree Structure (after "ant build" command)

- Hello2
  - web directory
  - build.xml
  - build directory
    - WEB-INF directory
      - classes directory
      - web.xml
    - duke.waving.gif
    - greeting.jsp
    - response.jsp

5. Install or Deploy Web application

- There are 2 different ways to install/deploy Web application
  - By asking Tomcat Manager via sending a command to it (Tomcat Manager is just another Servlet app that is always running)
    - ant install
    - ant deploy
  - By manually copying files to Tomcat's webapps directory and then restarting Tomcat

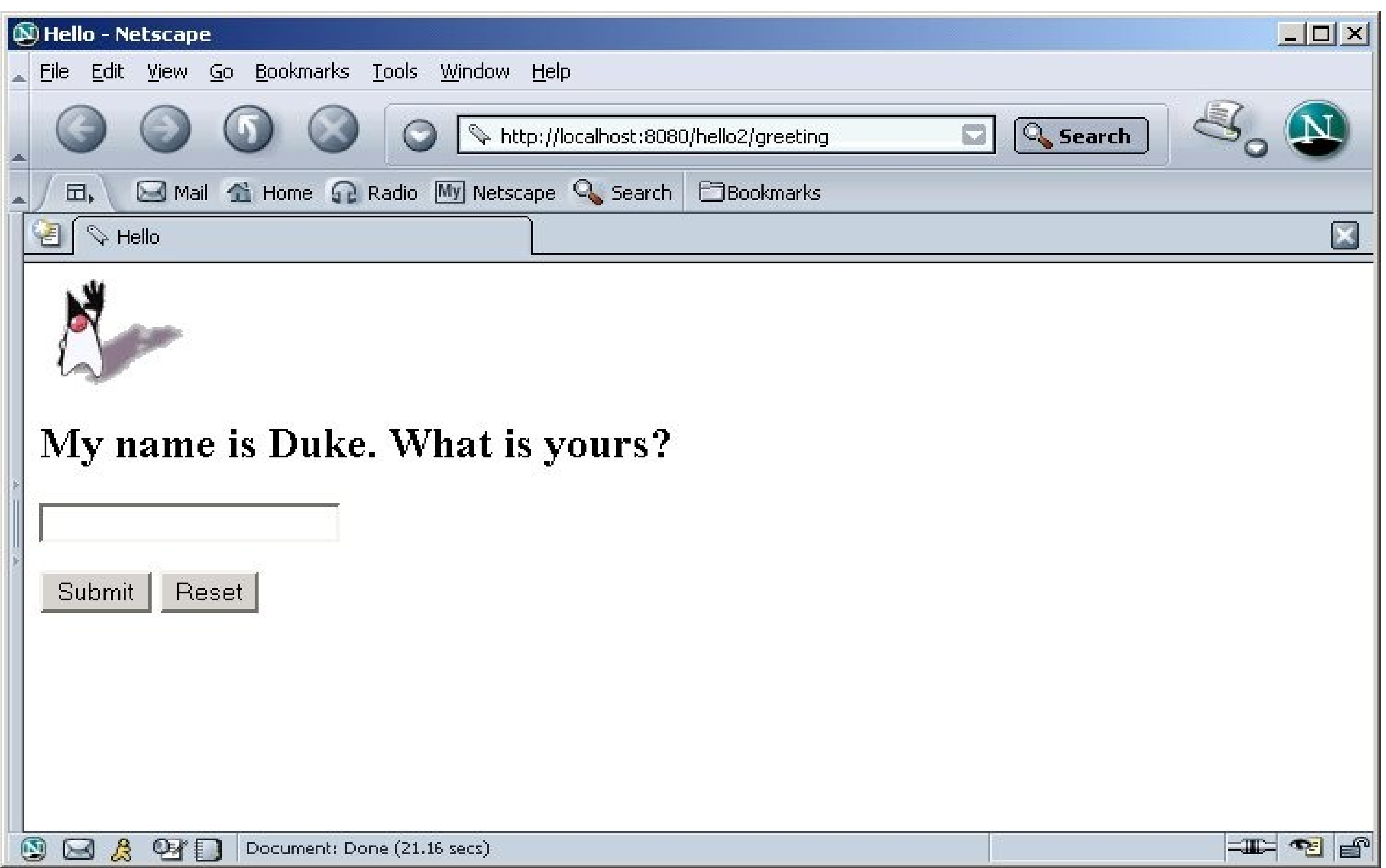
5.1 Install or Deploy Web application

- Via Tomcat manager
  - You need proper credential to perform this
    - This is why you need build.properties file with proper userid and password, otherwise you will experience HTTP 401 error
  - ant install for temporary deployment
  - ant deploy for permanent deployment
- Manual Method
  - Copying *.war file or unpacked directory to <tomcat-install>/webapps/ directory manually and then restart Tomcat

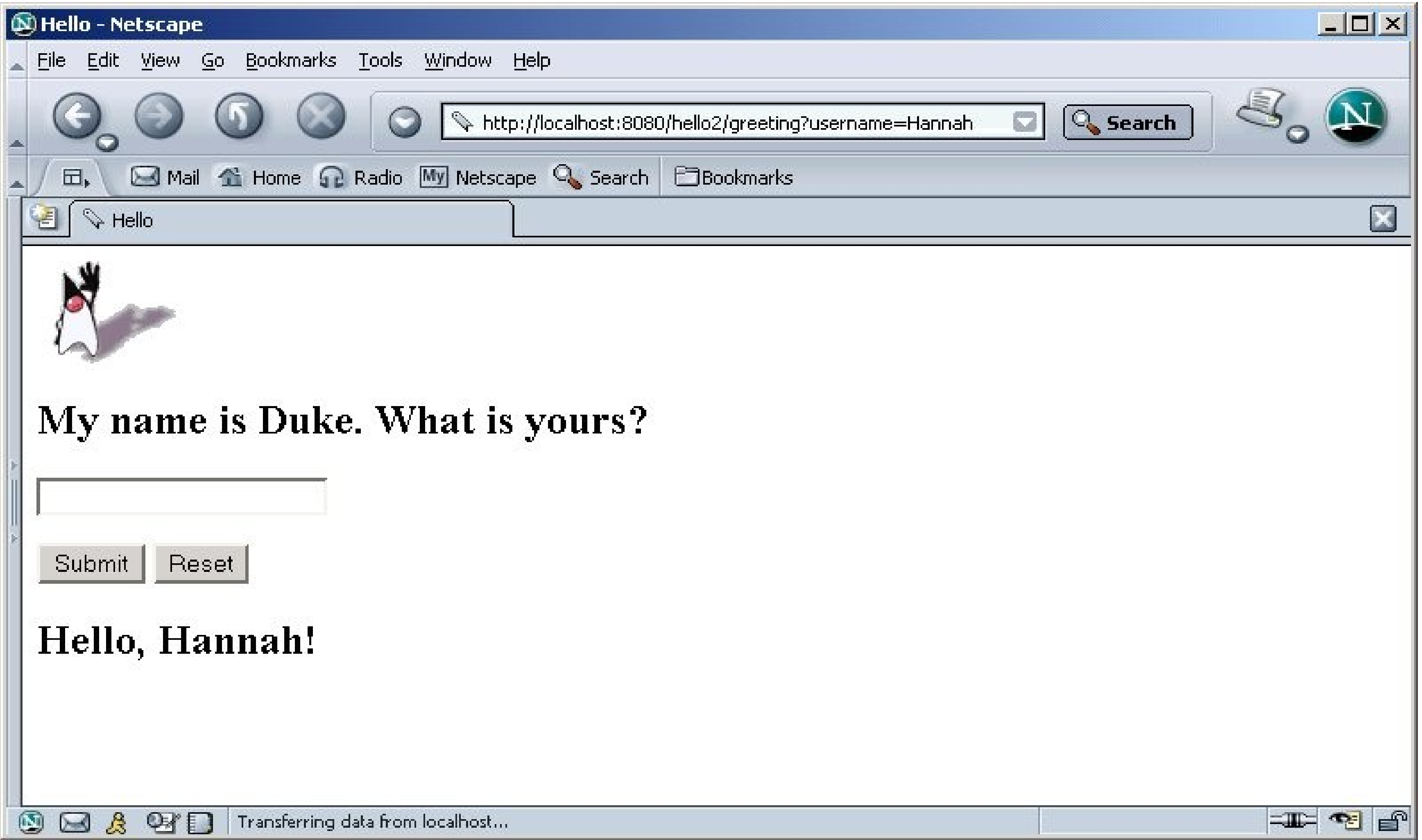
6. Perform Client Access to Web Application

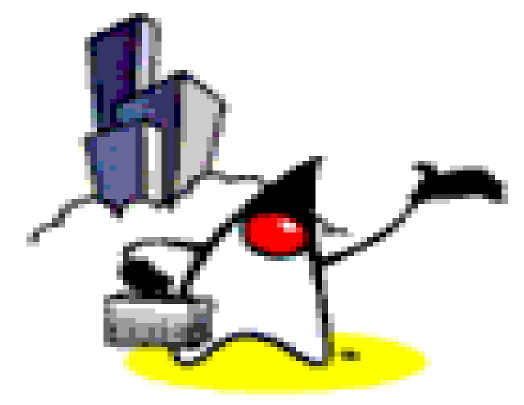
- From a browser, go to URN of the Web application
  - http://localhost:8080/hello2/greeting

http://localhost:8080/hello2/greeting



response.jsp





Comparing Hello1 Servlet & Hello2 JSP code

GreetingServlet.java (1)

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

/**
 * This is a simple example of an HTTP Servlet.  It responds to the GET
 * method of the HTTP protocol.
 */
public class GreetingServlet extends HttpServlet {

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

      response.setContentType("text/html");
      response.setBufferSize(8192);
      PrintWriter out = response.getWriter();

      // then write the data of the response
      out.println("<html>" +
                  "<head><title>Hello</title></head>");
```

GreetingServlet.java (2)

```
// then write the data of the response
out.println("<body  bgcolor=\"#ffffff\">" +
   "<img src=\"duke.waving.gif\">" +
   "<h2>Hello, my name is Duke. What's yours?</h2>" +
   "<form method=\"get\">" +
   "<input type=\"text\" name=\"username\" size=\"25\">" +
   "<p></p>" +
   "<input type=\"submit\" value=\"Submit\">" +
   "<input type=\"reset\" value=\"Reset\">" +
   "</form>");

String username = request.getParameter("username");

// dispatch to another web resource
if ( username != null && username.length() > 0 ) {
      RequestDispatcher dispatcher =
         getServletContext().getRequestDispatcher("/response");

      if (dispatcher != null)
      dispatcher.include(request, response);

}
out.println("</body></html>");
out.close();
}
```

GreetingServlet.java (3)

```
    public String getServletInfo() {
        return "The Hello servlet says hello.";

    }
}
```

greeting.jsp

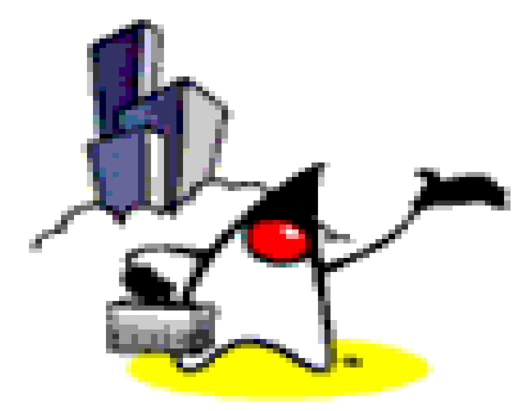
```
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<img src="duke.waving.gif">
<h2>My name is Duke. What is yours?</h2>

<form method="get">
<input type="text" name="username" size="25">
<p></p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>

<%
    String username = request.getParameter("username");
    if ( username != null && username.length() > 0 ) {
%>
    <%@include file="response.jsp" %>
<%
    }
%>
</body>
</html>
```

ResponseServlet.java

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

// This is a simple example of an HTTP Servlet.  It responds to the GET
// method of the HTTP protocol.
public class ResponseServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException{
        PrintWriter out = response.getWriter();

        // then write the data of the response
        String username = request.getParameter("username");
        if ( username != null && username.length() > 0 )
          out.println("<h2>Hello, " + username + "!</h2>");
    }

    public String getServletInfo() {
        return "The Response servlet says hello.";
    }
 }
```

response.jsp

```
<h2><font color="black">Hello, <%=username%>!</font></h2>
```



JSP “is” Servlet!

JSP is “Servlet”

- JSP pages get translated into servlet
  - Tomcat translates greeting.jsp into greeting$jsp.java
- Scriptlet (Java code) within JSP page ends up being inserted into jspService() method of resulting servlet
- Implicit objects for servlet are also available to JSP page designers, JavaBeans developers, custom tag designers

greeting$jsp.java (1)

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;


public class greeting$jsp extends HttpJspBase {


  static {
  }
  public greeting$jsp( ) {
  }

  private static boolean _jspx_inited = false;

  public final void _jspx_init() throws
   org.apache.jasper.runtime.JspException {
  }
```

greeting$jsp.java (2)

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse  response)
              throws java.io.IOException, ServletException {

  JspFactory _jspxFactory = null;
  PageContext pageContext = null;
  HttpSession session = null;
  ServletContext application = null;
  ServletConfig config = null;
  JspWriter out = null;
  Object page = this;
  String  _value = null;
```

greeting$jsp.java (3)

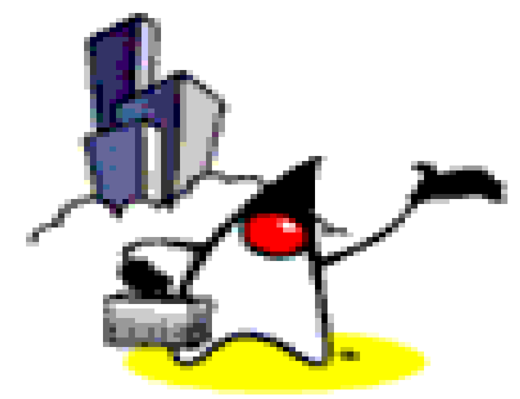
```
try {

  if (_jspx_inited == false) {
    synchronized (this) {
      if (_jspx_inited == false) {
        _jspx_init();
        _jspx_inited = true;
      }
    }
  }
  _jspxFactory = JspFactory.getDefaultFactory();
  response.setContentType("text/html;charset=ISO-8859-1");
  pageContext = _jspxFactory.getPageContext(this, request,
                response,"", true, 8192, true);

  application = pageContext.getServletContext();
  config = pageContext.getServletConfig();
  session = pageContext.getSession();
  out = pageContext.getOut();
```

greeting$jsp.java (4)

```
    // HTML // begin [file="/greeting.jsp";from=(38,4);to=(53,0)]
      out.write("\n\n<html>\n<head><title>Hello</title></head>\n<body
bgcolor=\"white\">\n<img src=\"duke.waving.gif\"> \n<h2>My name is Duke.
What is yours?</h2>\n\n<form method=\"get\">\n<input type=\"text\"
name=\"username\" size=\"25\">\n<p></p>\n<input type=\"submit\"
value=\"Submit\">\n<input type=\"reset\"
value=\"Reset\">\n</form>\n\n");


    // end
    // begin [file="/greeting.jsp";from=(53,2);to=(56,0)]

         String username = request.getParameter("username");
          if ( username != null && username.length() > 0 ) {
    // end
    // HTML // begin [file="/greeting.jsp";from=(56,2);to=(57,4)]
      out.write("\n    ");

    // end
    // HTML // begin [file="/response.jsp";from=(38,4);to=(40,31)]
      out.write("\n\n<h2><font color=\"black\">Hello, ");

    // end
    // begin [file="/response.jsp";from=(40,34);to=(40,42)]
      out.print(username);
```

greeting$jsp.java (5)

```
      // HTML // begin [file="/response.jsp";from=(40,44);to=(55,0)]
        out.write("!</font></h2>\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

      // end
      // HTML // begin [file="/greeting.jsp";from=(57,37);to=(58,0)]
        out.write("\n");

      // end
      // begin [file="/greeting.jsp";from=(58,2);to=(60,0)]

              }
      // end
      // HTML // begin [file="/greeting.jsp";from=(60,2);to=(63,0)]
        out.write("\n</body>\n</html>\n");

      // end
    } catch (Throwable t) {
      if (out != null && out.getBufferSize() != 0) out.clearBuffer();
      if (pageContext != null) pageContext.handlePageException(t);
    } finally {
      if (_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```



Dynamic Content Generation Techniques in JSP

Dynamic Contents Generation Techniques with JSP Technology

- Various techniques can be chosen depending on the following factors
  - size and complexity of the project
  - Requirements on re usability of code, maintainability, degree of robustness
- Simple to incrementally complex techniques are available

Dynamic Contents Generation Techniques with JSP

a) Call Java code directly within JSP

b) Call Java code indirectly within JSP

c) Use JavaBeans within JSP

d) Develop and use your own custom tags

e) Leverage 3rd-party custom tags or JSTL (JSP Standard Tag Library)

f) Follow MVC design pattern

g) Leverage proven Model2 frameworks

(a) Call Java code directly

- Place all Java code in JSP page
- Suitable only for a very simple Web application
  - hard to maintain
  - hard to reuse code
  - hard to understand for web page authors
- Not recommended for relatively sophisticated Web applications
  - weak separation between contents and presentation

(b) Call Java code indirectly

- Develop separate utility classes
- Insert into JSP page only the Java code needed to invoke the utility classes
- Better separation of contents generation from presentation logic than the previous method
- Better re usability and maintainability than the previous method
- Still weak separation between contents and presentation, however

(c) Use JavaBeans

- Develop utility classes in the form of JavaBeans
- Leverage built-in JSP facility of creating JavaBeans instances, getting and setting JavaBeans properties
  - Use JSP element syntax
- Easier to use for web page authors
- Better re usability and maintainability than the previous method

(d) Develop and Use Custom Tags

- Develop sophisticated components called custom tags
  - Custom tags are specifically designed for JSP
- More powerful than JavaBeans component
  - More than just getter and setter methods
- re usability, maintainability, robustness
- Development of custom tags are more difficult than creating JavaBeans, however

(e) Use 3rd-party Custom tags or JSTL

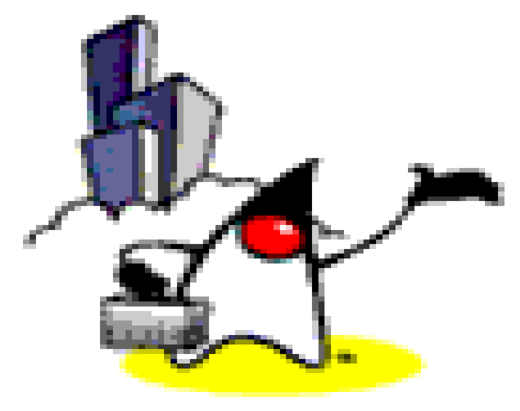
- There are many open source and commercial custom tags available
  - Apache Struts
- JSTL (JSP Standard Tag Library) standardize the set of custom tags that should be available over Java EE platform at a minimum
  - As a developer or deployer, you can be assured that a standard set of custom tags are already present in Java EE compliant platform (J2EE 1.3 and after)

(f) Design/Use MVC Design Pattern

- Follow MVC design pattern
  - Model using some model technologies
  - View using JSP
  - Controller using Servlet
- Creating and maintaining your own MVC framework is highly discourage, however

(g) Use Proven MVC Model2 Frameworks

- There are many to choose from
  - Apache Struts
  - JavaServer Faces (JSF)
  - Other frameworks: Echo, Tapestry, WebWorks, Wicket



Invoking Java Code with JSP Scripting Elements

JSP Scripting Elements

- Lets you insert Java code into the servlet that will be generated from JSP page
- Minimize the usage of JSP scripting elements in your JSP pages if possible
- There are three forms
  - Expressions: <%= Expressions %>
  - Scriptlets: <% Code %>
  - Declarations: <%! Declarations %>

Expressions

- During execution phase
  - Expression is evaluated and converted into a String
  - The String is then Inserted into the servlet's output stream directly
  - Results in something like out.println(expression)
  - Can use predefined variables (implicit objects) within expression
- Format
  - <%= Expression %> or
  - <jsp:expression>Expression</jsp:expression>
  - Semi-colons are not allowed for expressions

Example: Expressions

- Display current time using Date class
  - Current time: <%= new java.util.Date() %>
- Display random number using Math class
  - Random number: <%= Math.random() %>
- Use implicit objects
  - Your hostname: <%= request.getRemoteHost() %>
  - Your parameter: <%= request. getParameter(“yourParameter”) %>
  - Server: <%= application.getServerInfo() %>
  - Session ID: <%= session.getId() %>

Scriptlets

- Used to insert arbitrary Java code into servlet's jspService() method
- Can do things expressions alone cannot do
  - setting response headers and status codes
  - writing to a server log
  - updating database
  - executing code that contains loops, conditionals
- Can use predefined variables (implicit objects)
- Format:
  - <% Java code %> or
  - <jsp:scriptlet> Java code</jsp:scriptlet>

Example: Scriptlets

- Display query string

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

- Settng response type

```
<% response.setContentType("text/plain"); %>
```

Example: Scriptlet with Loop

```
<%
  Iterator i = cart.getItems().iterator();
  while (i.hasNext()) {
    ShoppingCartItem item =
      (ShoppingCartItem)i.next();
    BookDetails bd = (BookDetails)item.getItem();
%>

    <tr>
    <td align="right" bgcolor="#ffffff">
    <%=item.getQuantity()%>
    </td>
    <td bgcolor="#ffffaa">
    <strong><a href="
    <%=request.getContextPath()%>/bookdetails?bookId=
    <%=bd.getBookId()%>"><%=bd.getTitle()%></a></strong>
    </td>
    ...
<%
  // End of while
  }
%>
```

Example: Scriptlet Result

Example: JSP page fragment

- Suppose we have the following JSP page fragment
  - <H2> sangHTML </H2>
  - <%= sangExpression() %>
  - <% sangScriptletCode(); %>

Example: Resulting Servlet Code

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession(true);
    JSPWriter out = response.getWriter();

    // Static HTML fragment is sent to output stream  in "as is" form
    out.println("<H2>sangHTML</H2>");

    // Expression is converted into String and then sent to output
    out.println(sangExpression());

    // Scriptlet is inserted as Java code within _jspService()
    sangScriptletCode();
    ...
}
```

Declarations

- Used to define variables or methods that get inserted into the main body of servlet class
  - Outside of _jspSevice() method
  - Implicit objects are not accessible to declarations
- Usually used with Expressions or Scriptlets
- For initialization and cleanup in JSP pages, use declarations to override jspInit() and jspDestroy() methods
- Format:
  - <%! method or variable declaration code %>
  - <jsp:declaration> method or variable declaration code </jsp:declaration>

Example: JSP Page fragment

```
<H1>Some heading</H1>
<%!
   private String randomHeading() {
       return(“<H2>” + Math.random() + “</H2>”);
   }
%>
<%= randomHeading() %>
```

Example: Resulting Servlet Code

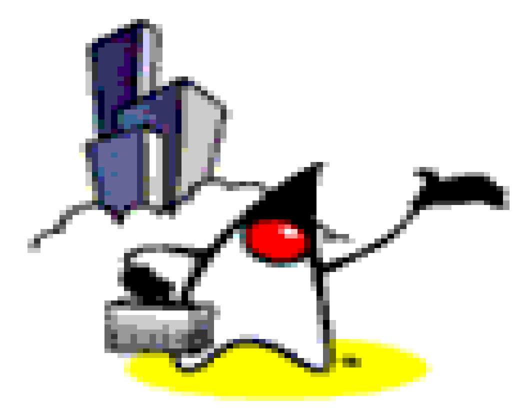
```
public class xxxx implements HttpJSPPage {
  private String randomHeading() {
     return(“<H2>” + Math.random() + “</H2>”);
  }

  public void _jspService(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException {
    response.setContentType(“text/html”);
    HttpSession session = request.getSession(true);
    JSPWriter out = response.getWriter();
    out.println(“<H1>Some heading</H1>”);
    out.println(randomHeading());
    ...
  }
  ...
}
```

Example: Declaration

```
<%!
  private BookDBAO bookDBAO;

  public void jspInit() {
    ...
  }
  public void jspDestroy() {
    ...
  }
%>
```

Why XML Syntax?

- From JSP 1.2
- Examples
  - <jsp:expression>Expression</jsp:expression>
  - <jsp:scriptlet> Java code</jsp:scriptlet>
  - <jsp:declaration> declaration code </jsp:declaration>
- You can leverage
  - XML validation (via XML schema)
  - Many other XML tools
    - editor
    - transformer
    - Java APIs



Including and Forwarding to Other Web Resource

Including Contents in a JSP Page

- Two mechanisms for including another Web resource in a JSP page
  - include directive
  - jsp:include element

Include Directive

- Is processed when the JSP page is translated into a servlet class
- Effect of the directive is to insert the text contained in another file-- either static content or another JSP page--in the including JSP page
- Used to include banner content, copyright information, or any chunk of content that you might want to reuse in another page
- Syntax and Example
  - <%@ include file="filename" %>
  - <%@ include file="banner.jsp" %>

jsp:include Element

- Is processed when a JSP page is executed
- Allows you to include either a static or dynamic resource in a JSP file
  - static: its content is inserted into the calling JSP file
  - dynamic: the request is sent to the included resource, the included page is executed, and then the result is included in the response from the calling JSP page
- Syntax and example
  - <jsp:include page="includedPage" />
  - <jsp:include page="date.jsp"/>

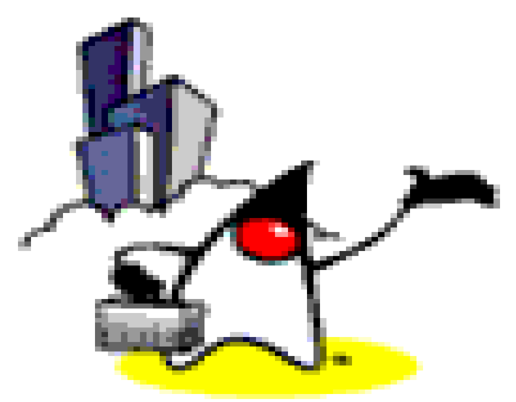
Which One to Use it?

- Use include directive if the file changes rarely
  - It is faster than jsp:include
- Use jsp:include for content that changes often
- Use jsp:include if which page to include cannot be decided until the main page is requested

Forwarding to another Web component

- Same mechanism as in Servlet
- Syntax
  - <jsp:forward page="/main.jsp" />
- Original request object is provided to the target page via jsp:parameter element

```
<jsp:forward page="..." >
  <jsp:param name="param1" value="value1"/>
</jsp:forward>
```



Directives

Directives

- Directives are messages to the JSP container in order to affect overall structure of the servlet
- Do not produce output into the current output stream
- Syntax
  - <%@ directive {attr=value}* %>

Three Types of Directives

- page: Communicate page dependent attributes and communicate these to the JSP container
  - <%@ page import="java.util.* %>
- include: Used to include text and/or code at JSP page translation-time
  - <%@ include file="header.html" %>
- Taglib: Indicates a tag library that the JSP container should interpret
  - <%@ taglib uri="mytags" prefix="codecamp" %>

Page Directives

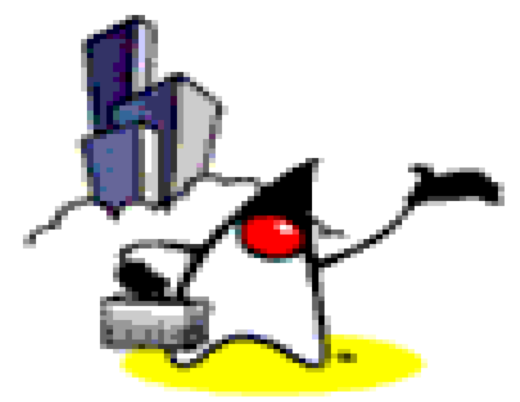
- Give high-level information about the servlet that results from the JSP page.
- Control
  - Which classes are imported
    - <%@ page import="java.util.* %>
  - What MIME type is generated
    - <%@ page contentType="MIME-Type" %>
  - How multithreading is handled
    - <%@ page isThreadSafe="true" %> <%!--Default --%>
    - <%@ page isThreadSafe="false" %>
  - What page handles unexpected errors
    - <%@ page errorPage="errorpage.jsp" %>

Implicit Objects

- A JSP page has access to certain implicit objects that are always available, without being declared first
- Created by container
- Corresponds to classes defined in Servlet

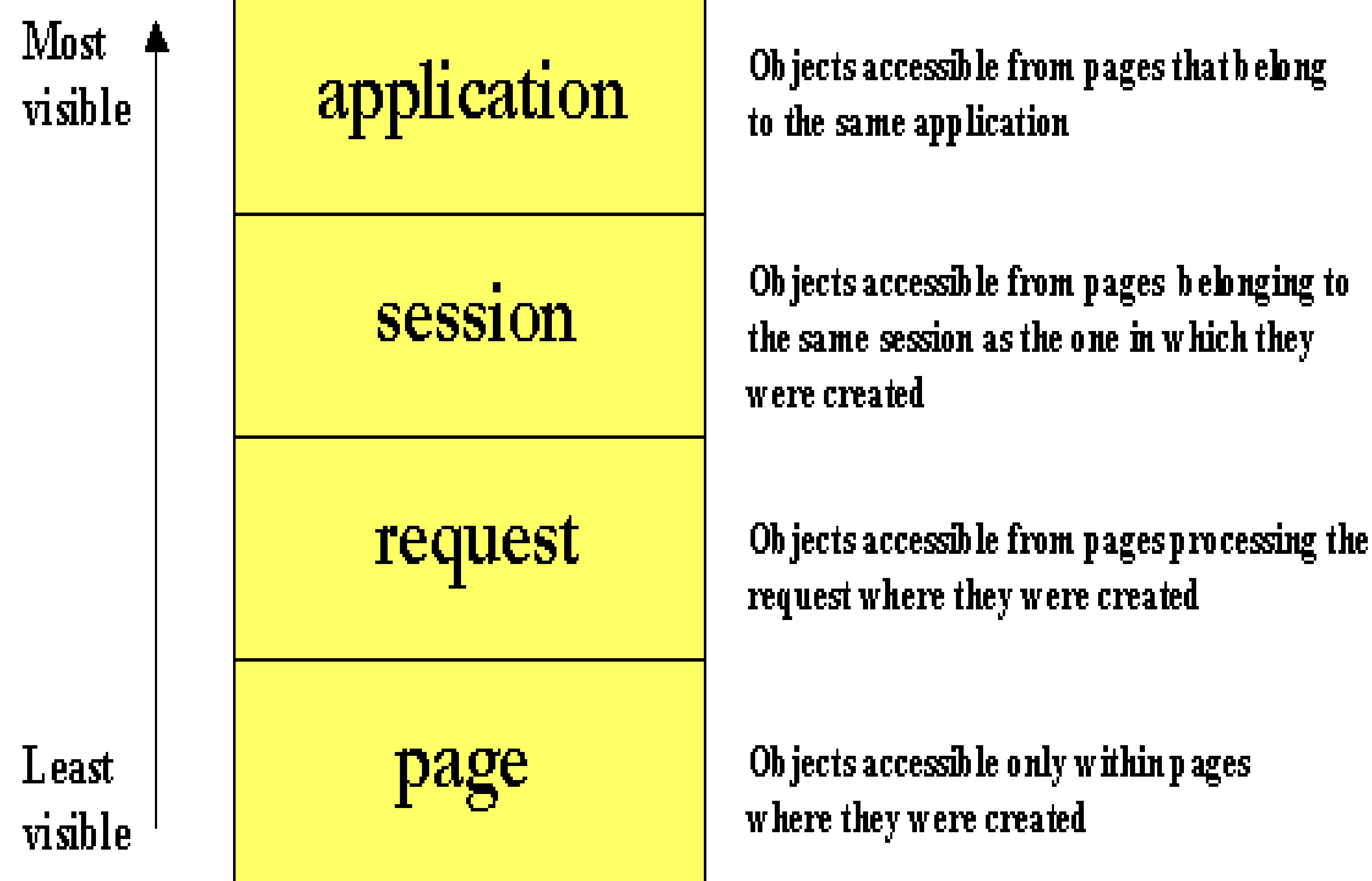
Implicit Objects

- request (HttpServletRequest)
- response (HttpServletRepsonse)
- session (HttpSession)
- application(ServletContext)
- out (of type JspWriter)
- config (ServletConfig)
- pageContext

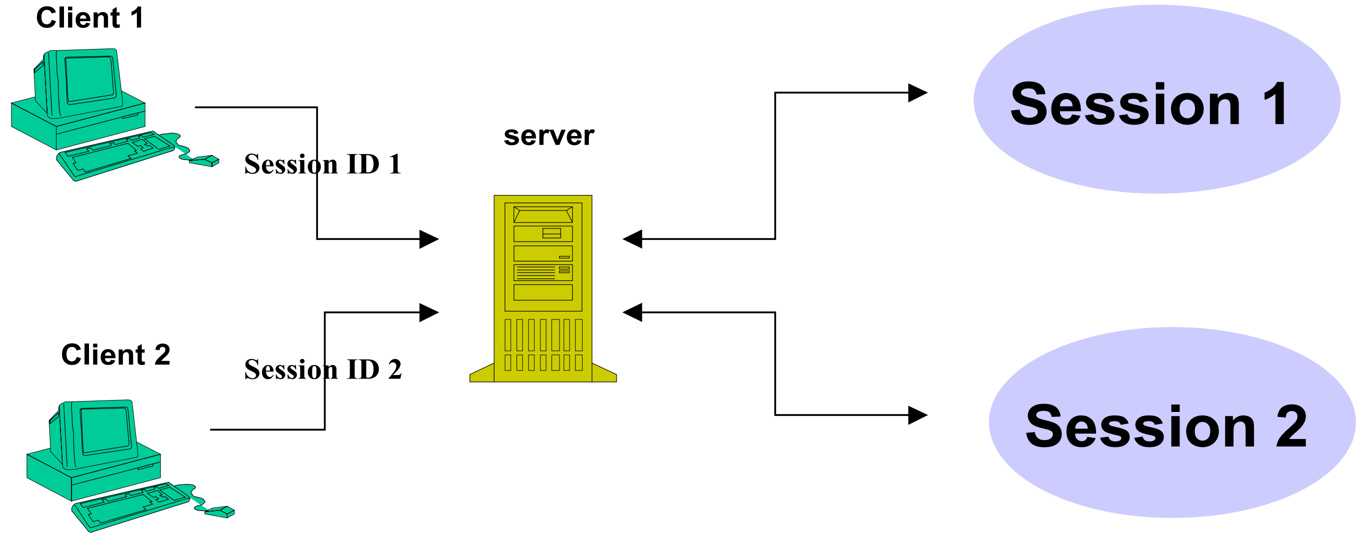


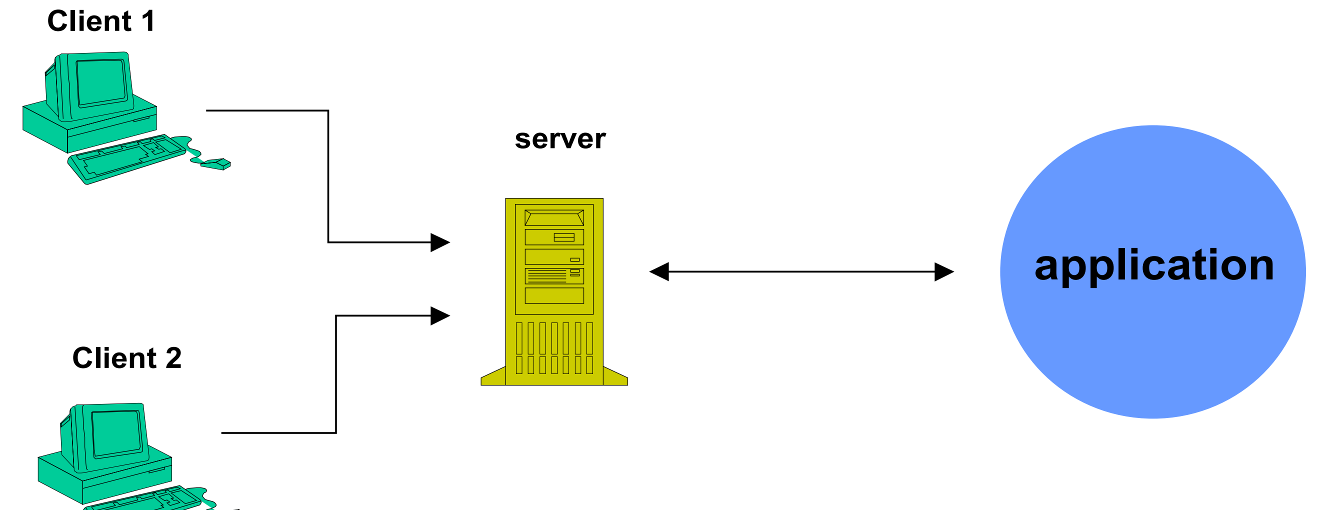
Scope Objects

Different Scope

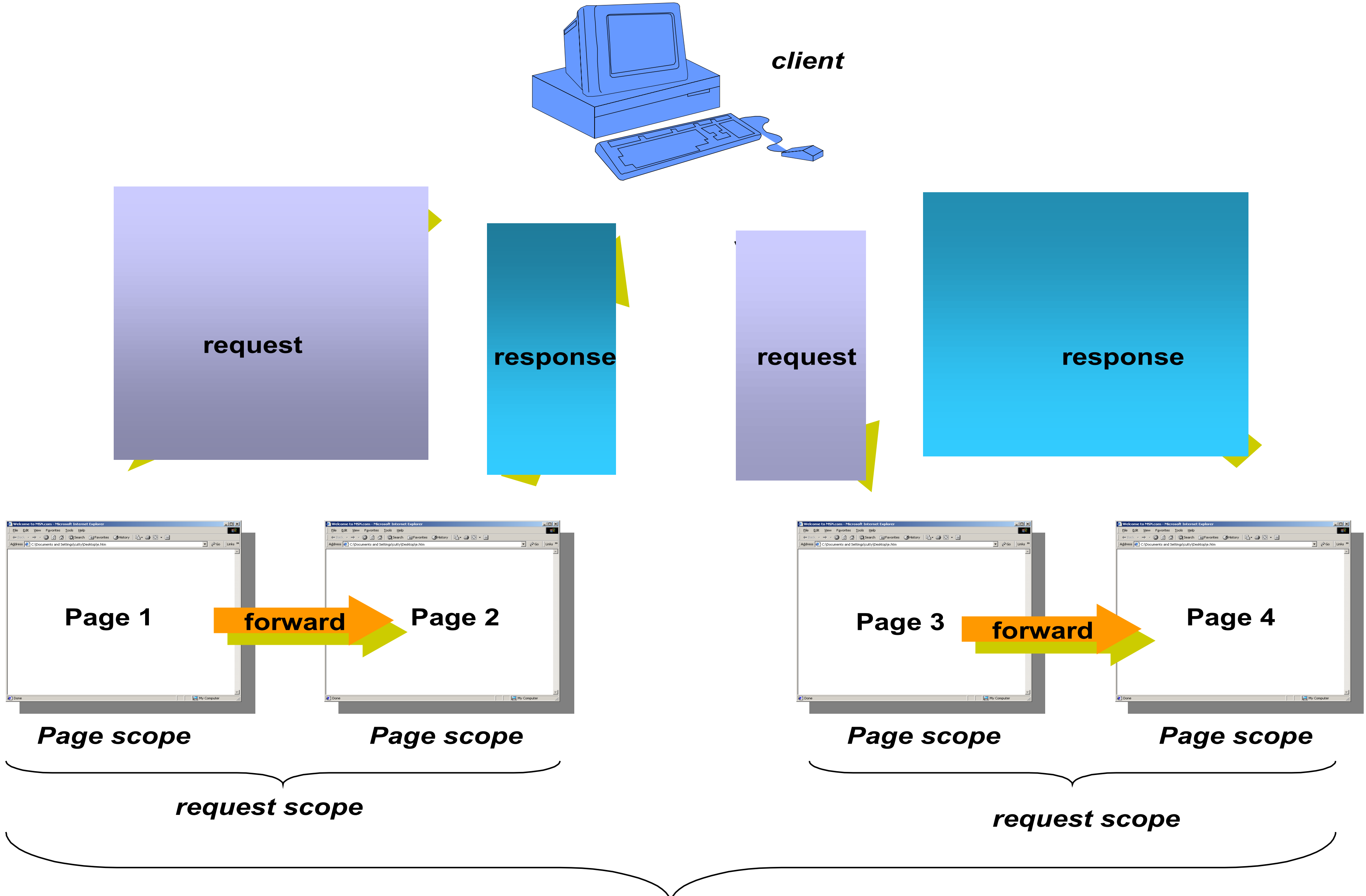


Session, Application Scope

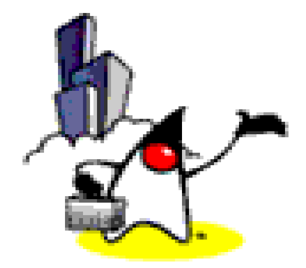




Session, Request, Page Scope



JavaBeans Components for JSP



What are JavaBeans?

- Java classes that can be easily reused and composed together into an application
- Any Java class that follows certain design conventions can be a JavaBeans component
  - properties of the class
  - public methods to get and set properties
- Within a JSP page, you can create and initialize beans and get and set the values of their properties
- JavaBeans can contain business logic or data base access logic

JavaBeans Design Conventions

- JavaBeans maintain internal properties
- A property can be
  - Read/write, read-only, or write-only
  - Simple or indexed
- Properties should be accessed and set via getXxx and setXxx methods
  - PropertyClass getProperty() { ... }
  - PropertyClass setProperty() { ... }
- JavaBeans must have a zero-argument (empty) constructor

Example: JavaBeans

```
public class Currency {
  private Locale locale;
  private double amount;
  public Currency() {
    locale = null;
    amount = 0.0;
  }
  public void setLocale(Locale l) {
    locale = l;
  }
  public void setAmount(double a) {
    amount = a;
  }
  public String getFormat() {
     NumberFormat nf =
     NumberFormat.getCurrencyInstance(locale);
     return nf.format(amount);
  }
}
```

Why Use JavaBeans in JSP Page?

- A JSP page can create and use any type of Java programming language object within a declaration or scriptlet like following:

```
<%
  ShoppingCart cart = (ShoppingCart)session.getAttribute("cart");
  // If the user has no cart, create a new one
  if (cart == null) {
      cart = new ShoppingCart();
      session.setAttribute("cart", cart);
  }
%>
```

Why Use JavaBeans in JSP Page?

- JSP pages can use JSP elements to create and access the object that conforms to JavaBeans conventions

```
<jsp:useBean id="cart" class="cart.ShoppingCart"
  scope="session"/>
```

Create an instance of “ShoppingCart” if none exists, stores it as an attribute of the session scope object, and makes the bean available throughout the session by the identifier “cart”

Compare the Two

```
<%
  ShoppingCart cart = (ShoppingCart)session.getAttribute("cart");
  // If the user has no cart object as an attribute in Session scope
  // object, then create a new one.  Otherwise, use the existing
  // instance.
  if (cart == null) {
      cart = new ShoppingCart();
      session.setAttribute("cart", cart);
  }
%>
```

versus

```
<jsp:useBean id="cart" class="cart.ShoppingCart"
                   scope="session"/>
```

Why Use JavaBeans in JSP Page?

- No need to learn Java programming language for page designers
- Stronger separation between content and presentation
- Higher re usability of code
- Simpler object sharing through built-in sharing mechanism
- Convenient matching between request parameters and object properties

Creating a JavaBeans

- Declare that the page will use a bean that is stored within and accessible from the specified scope by jsp:useBean element

```
<jsp:useBean id="beanName"
   class="fully_qualified_classname" scope="scope"/>
      or
<jsp:useBean id="beanName"
   class="fully_qualified_classname" scope="scope">
      <jsp:setProperty .../>
</jsp:useBean>
```

Setting JavaBeans Properties

- 2 ways to set a property of a bean
- Via scriptlet
  - <% beanName.setPropName(value); %>
- Via JSP:setProperty
  - <jsp:setProperty name="beanName" property="propName" value="string constant"/>
  - "beanName" must be the same as that specified for the id attribute in a useBean element
  - There must be a setPropName method in the bean

Setting JavaBeans Properties

- jsp:setProperty syntax depends on the source of the property
  - <jsp:setProperty name="beanName" property="propName" value="string constant"/>
  - <jsp:setProperty name="beanName" property="propName" param="paramName"/>
  - <jsp:setProperty name="beanName" property="propName"/>
  - <jsp:setProperty name="beanName" property="*"/>
  - <jsp:setProperty name="beanName" property="propName" value="<%= expression %>"/>

Example: jsp:setProperty with Request parameter

```
<jsp:setProperty name="bookDB" property="bookId"/>
```

is same as

```
<%
   //Get the identifier of the book to display
   String bookId = request.getParameter("bookId");
   bookDB.setBookId(bookId);
   ...
%>
```

Example: jsp:setProperty with Expression

```
<jsp:useBean id="currency" class="util.Currency"
   scope="session">
   <jsp:setProperty name="currency" property="locale"
                    value="<%= request.getLocale() %>"/>
</jsp:useBean>

<jsp:setProperty name="currency" property="amount"
  value="<%=cart.getTotal()%>"/>
```

Getting JavaBeans Properties

- 2 different ways to get a property of a bean
  - Convert the value of the property into a String and insert the value into the current implicit “out” object
  - Retrieve the value of a property without converting it to a String and inserting it into the “out” object

Getting JavaBeans Properties & and Convert into String and insert into out

- 2 ways
  - via scriptlet
    - <%= beanName.getPropName() %>
  - via JSP:setProperty
    - <jsp:getProperty name="beanName" property="propName"/>
- Requirements
  - "beanName" must be the same as that specified for the id attribute in a useBean element
  - There must be a "getPropName()" method in a bean

Getting JavaBeans Properties without Converting it to String

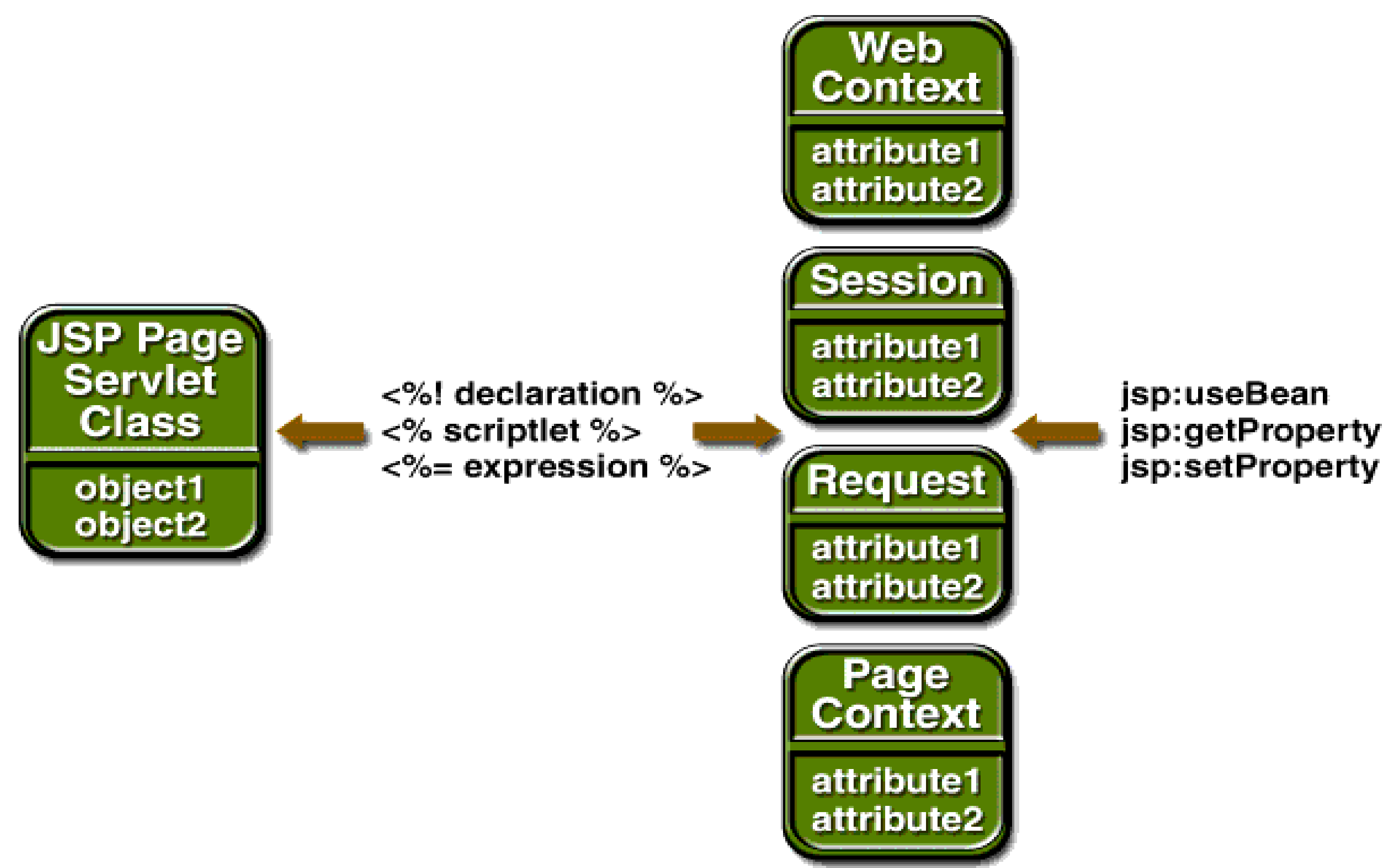
- Must use a scriptlet
- Format

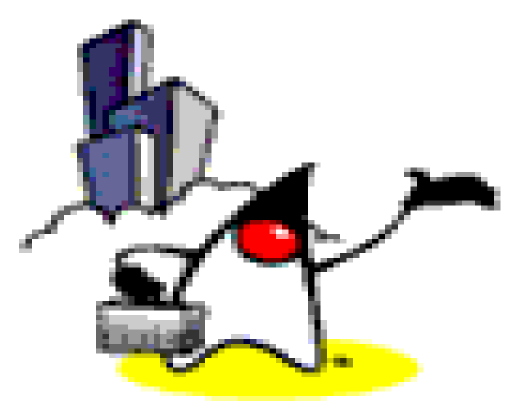
```
<% Object o = beanName.getPropName(); %>
```

- Example

```
<%
  // Print a summary of the shopping cart
  int num = cart.getNumberOfItems();
  if (num > 0) {
%>
```

Accessing Objects from JSP Page





Error Handling

Creating An Exception Error Page

- Determine the exception thrown
- In each of your JSP, include the name of the error page
  - **<**%@ page errorPage="errorpage.jsp" %>
- Develop an error page, it should include
  - <%@ page isErrorPage="true" %>
- In the error page, use the exception reference to display exception information
  - <%= exception.toString() %>

Example: initdestroy.jsp

```
<%@ page import="database.*" %>
<%@ page errorPage="errorpage.jsp" %>
<%!

  private BookDBAO bookDBAO;
  public void jspInit() {

    // retrieve database access object, which was set once per web
     application
    bookDBAO =
      (BookDBAO)getServletContext().getAttribute("bookDB");
    if (bookDBAO == null)
        System.out.println("Couldn't get database.");
  }

  public void jspDestroy() {
    bookDBAO = null;
  }
%>
```

Example: errorpage.jsp

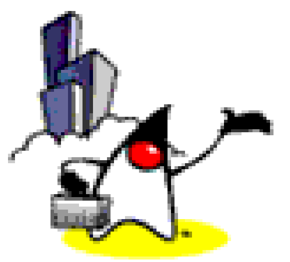
```
<%@ page isErrorPage="true" %>
<%@ page import="java.util.*" %>
<%
   ResourceBundle messages =
     (ResourceBundle)session.getAttribute("messages");
   if (messages == null) {
      Locale locale=null;
      String language = request.getParameter("language");

      if (language != null) {
         if (language.equals("English")) {
          locale=new Locale("en", "");
         } else {
          locale=new Locale("es", "");
         }
      } else
         locale=new Locale("en", "");

      messages = ResourceBundle.getBundle("BookStoreMessages", locale);
      session.setAttribute("messages", messages);
   }
```

Example: errorpage.jsp

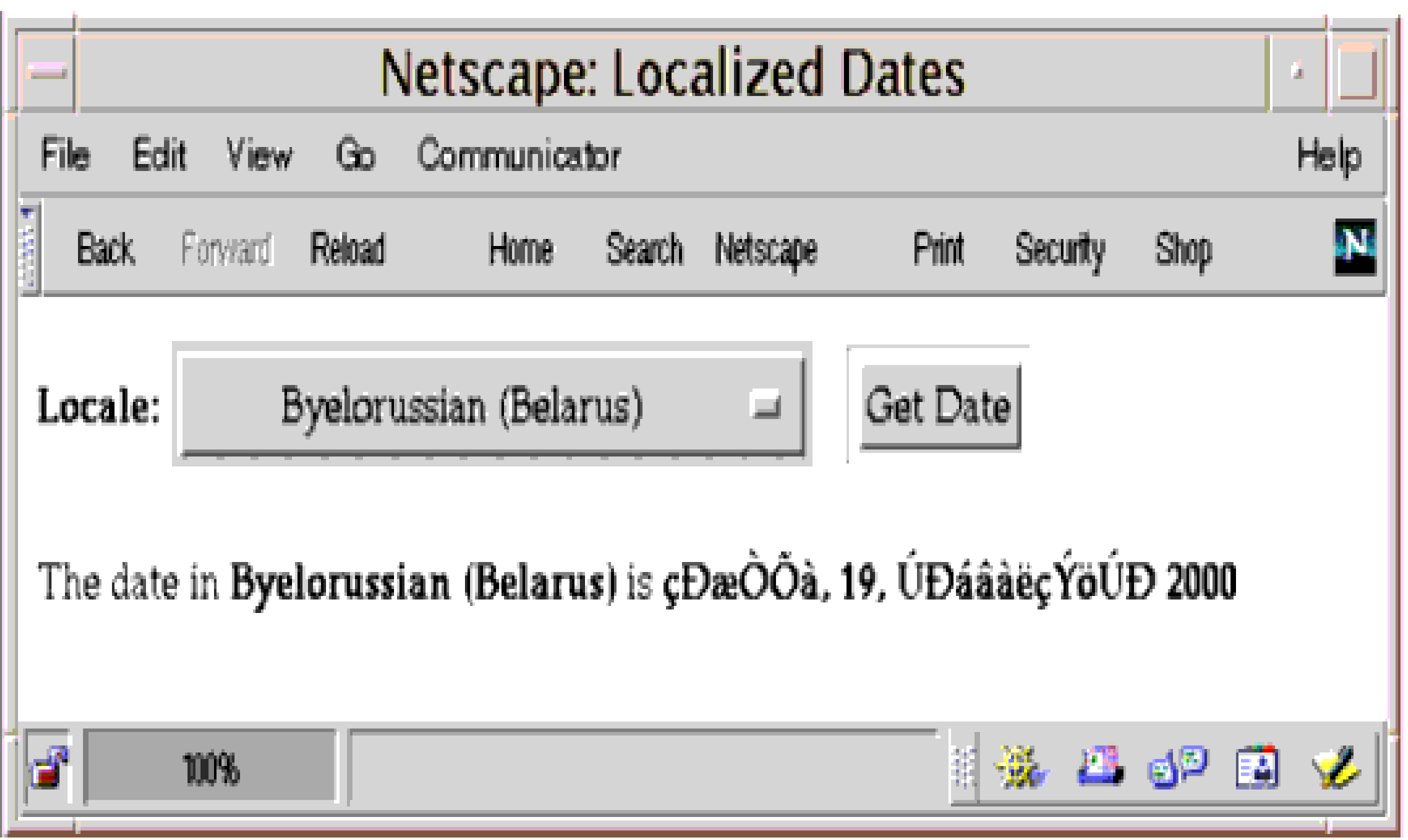
... (continued)

```
<html>
<head>
<title><%=messages.getString("ServerError")%></title>
</head>
<body bgcolor="white">
<h3>
<%=messages.getString("ServerError")%>
</h3>
<p>
<%= exception.getMessage() %>
</body>
</html>
```

Date Example



Date Example Output



Date Example

```
<%@ page import="java.util.*" %>
<%@ page import="MyLocales" %>
<%@ page contentType="text/html; charset=ISO-8859-5" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application" class="MyLocales"/>
<form name="localeForm" action="index.jsp" method="post">
<b>Locale:</b>
```

Date Example

```
<select name=locale>
<%
  Iterator i = locales.getLocaleNames().iterator();
  String selectedLocale = request.getParameter("locale");
          while (i.hasNext()) {
                String locale = (String)i.next();
                if (selectedLocale != null && selectedLocale.equals(locale) )
    { %>
     <option selected><%=locale%></option>
<%              } else { %>
     <option><%=locale%></option>
<%              }
  }
%>
</select>
<input type="submit" name="Submit" value="Get Date">
</form>
```

Date Example

```
<p>
<jsp:include page="date.jsp" flush="true" />
</body>
</html>
```

Passion!