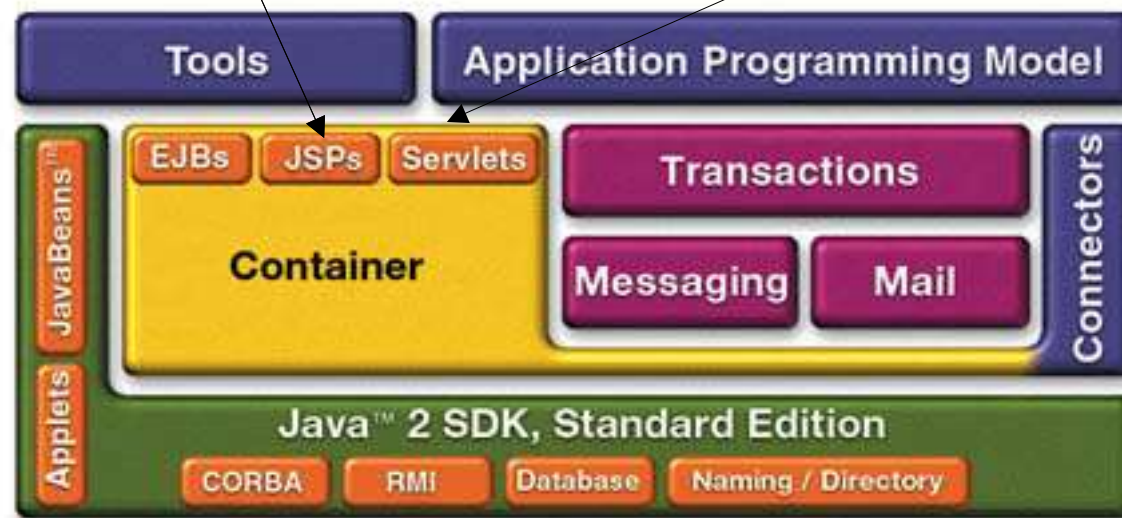


# Introduction to JSP

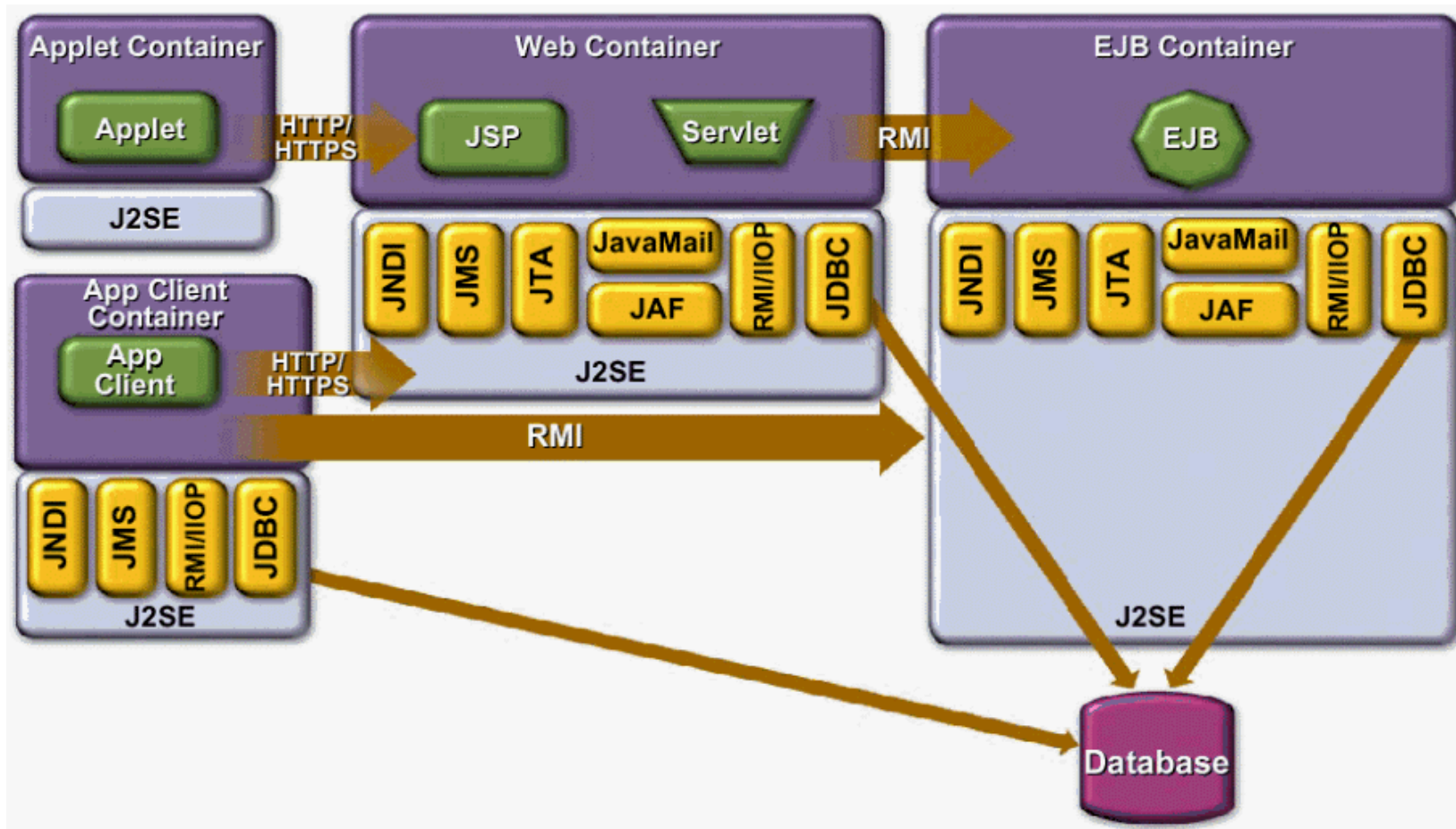
# JSP and Servlet in J2EE Architecture

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**.



# JSP and Servlet in J2EE Architecture




# What is a JSP page ?

- 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

# Example

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

dynamic content



# JSP versus Servlet

- Servlets
  - HTML Code in Java
  - Not easy to author (lot of println)
- JSP
  - Java-like code in HTML
  - Easier to author
  - Compiled into servlet
- Both have pro and cons
  - typical use is to combine them (e.g. MVC pattern)

# Separation of Concerns

From pure *servlet* to a *mix*

## 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>
```

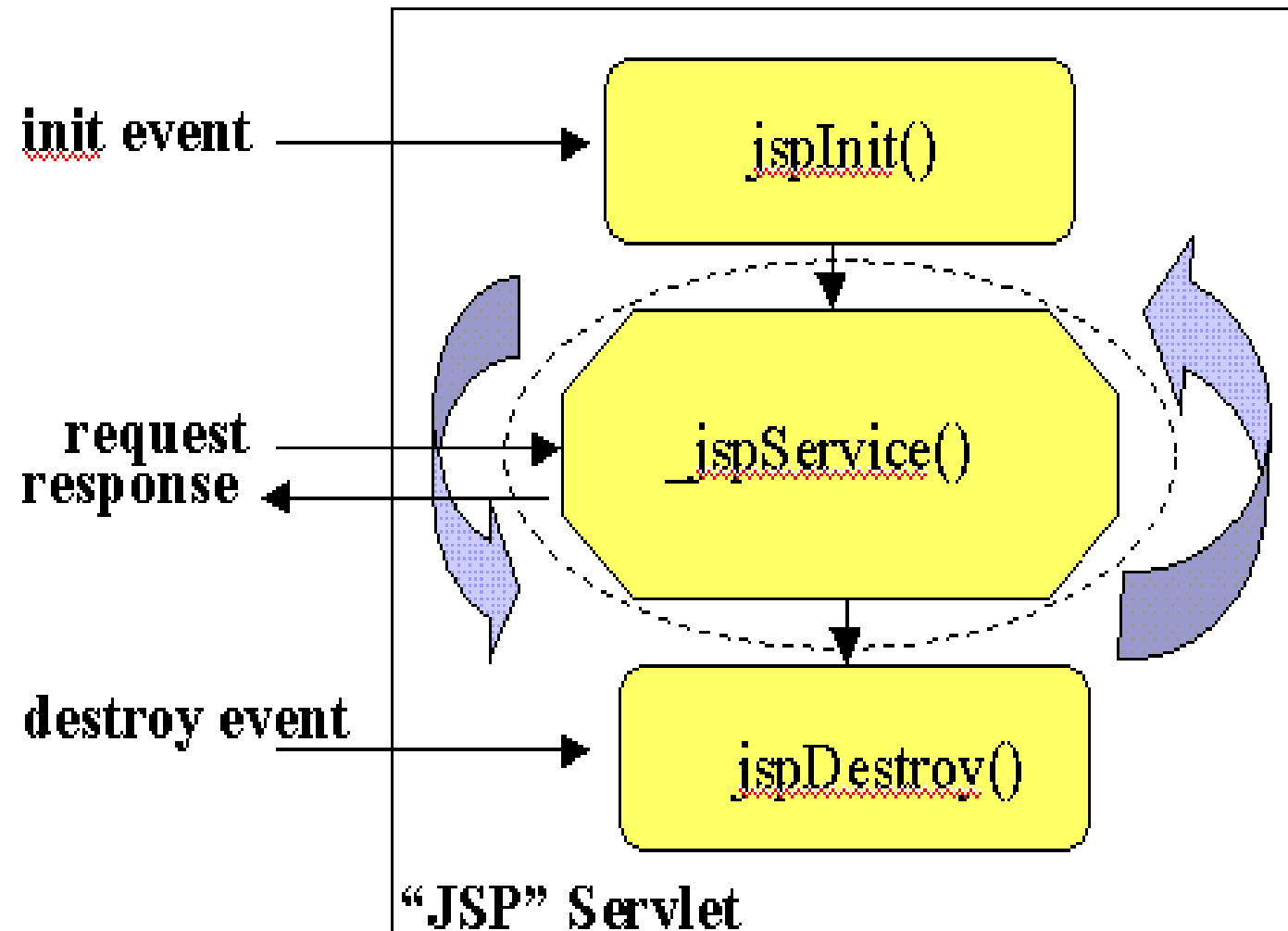
Business logic

## JavaBeans

isOrderValid( )

saveOrder( )

# JSP Lifecycle





# JSP Initialization

- 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

# JSP Finalization

- 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

```
<% @ 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;
```

```
}
```

```
%>
```

# Web application development steps

1. Write (and compile) the Web component code (Servlet or JSP) and helper classes referenced by the web component code
2. Create any 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
  - Web clients are now ready to access them via URL

# Step 1: write and compile Web component code

- Create development tree structure
  - Keep Web application source separate from compiled files
    - facilitate iterative development
  - Root directory
    - **src**: Java source of servlets and JavaBeans components
    - **web**: **JSP pages** and HTML pages, images
- Write either servlet code and/or JSP pages along with related helper code

## Step 2: Create static resources

- HTML pages
  - Custom pages
  - Login pages
  - Error pages
- Image files that are used by HTML pages or JSP pages

## Step 3: Create 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

```
<?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>
```

```
  </servlet>
```

```
  <servlet-mapping>
```

```
    <servlet-name>greeting</servlet-name>
```

```
    <url-pattern>/greeting</url-pattern>
```

```
  </servlet-mapping>
```

```
</web-app>
```



## Step 4: Build Web application

- Code Java supporting code
- Create \*.WAR file or unpacked form of \*.WAR file
  - packaging automated with JBoss IDE

# Step 5: Install/Deploy web application

- Automated with Jboss IDE

# Servlet example

```
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>");
    }
}
```

```

// 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().getRequestDispatche ("/response");

    if (dispatcher != null)
        dispatcher.include(request, response);
}
out.println("</body></html>");
out.close();
}
public String getServletInfo() {

    return "The Hello servlet says hello.";

}
}

```

# Response servlet

```
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.";
    }
}
```

# JSP example

```
<html>
<head><title>Hello</title></head>
<body bgcolor="white">

<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>
```

# JSP example - response

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

# Content Generation with JSP

- Possible techniques:
  - 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 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 reusability and maintainability than the previous method
- Still weak separation between contents and presentation

## (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 reusability 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
- Reusability, maintainability, robustness

## (e) Use 3<sup>rd</sup> 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 J2EE platform at a minimum
  - As a developer or deployer, you can be assured that a standard set of custom tags are already present in J2EE compliant platform (J2EE 1.3 and after)

## (f,g) Use MVC Pattern

- Follow MVC design pattern
- Or use open source or commercial Model2 frameworks
  - Apache Struts
  - JavaServer Faces
  - Sun ONE Application Framework
  - Other JavaServer Pages based implementations

# JSP Scripting Elements

- Lets you insert Java code into the servlet that will be generated from JSP page
- There are three forms
  - Expressions: `<%= Expressions %>`
  - Scriptlets: `<% Code %>`
  - Declarations: `<%! Declarations %>`
- Do not abuse JSP scripting elements in JSP pages if possible

# 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



# Examples

- 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>`

# Examples

- Display query string

`<%`

`String queryData = request.getQueryString();`

`out.println("Attached GET data: " + queryData);`

`%>`

- Setting response type

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

# Examples 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**

**}**

**%>**

# Declarations

- To define variables or methods that get inserted into the main body of servlet class
- For initialization and cleanup in JSP pages, used to override `jspInit()` and `jspDestroy()` methods
- Format:
  - `<%! method or variable declaration code %>`
  - `<jsp:declaration> method or variable declaration code </jsp:declaration>`

# Examples

<H1>Some heading</H1>

<%!

```
private String randomHeading() {
```

```
    return("<H2>" + Math.random() + "</H2>");
```

```
}
```

%>

<%= randomHeading() %>

# Examples

<%!

```
private BookDBAO bookDBAO;
```

```
public void jspInit() {
```

```
    ...
```

```
}
```

```
public void jspDestroy() {
```

```
    ...
```

```
}
```

%>

# Including Content in JSP Page

- Include directive
  - content that need to be reused in other pages (e.g. banner content, copyright information)
  - `<%@ include file="banner.jsp" %>`
- `jsp:include` element
  - to include either a static or dynamic resource in a JSP file
    - static: its content is inserted into the calling JSP file
    - dynamic: 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
  - `<jsp:include page="date.jsp"/>`



# Forwarding

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

`<jsp:include page="..." >`

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

`</jsp:include>`

# Directives

- 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}* %>`

# 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" %>`

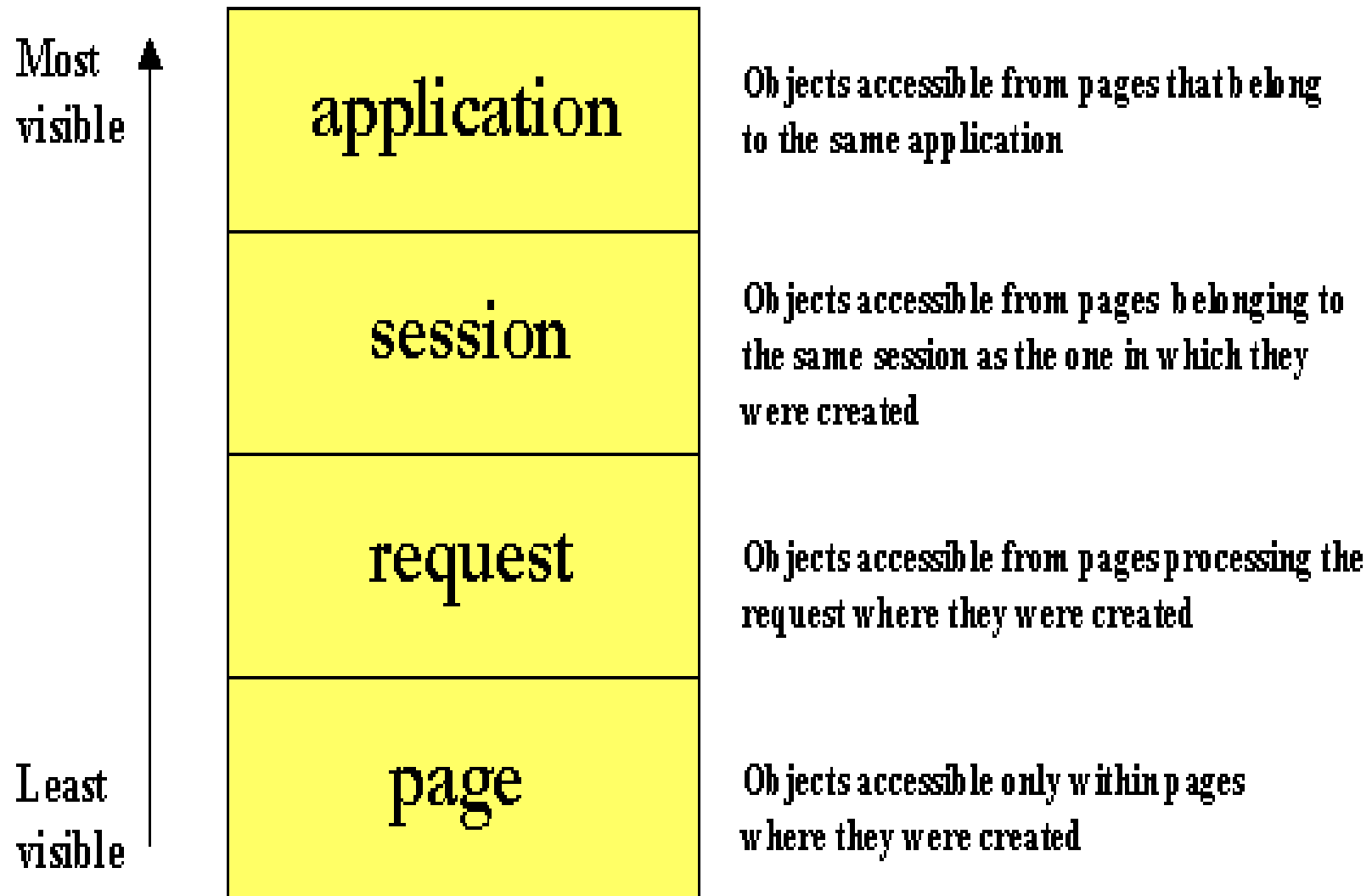
# Page Directives

- 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

- JSP page has access to a number of implicit objects (same as servlets)
  - request (HttpServletRequest)
  - response (HttpServletResponse)
  - session (HttpSession)
  - application(ServletContext)
  - out (of type JspWriter)
  - config (ServletConfig)
  - pageContext

# Scope of Objects



# Using 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 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 getProperty() { ... }**
- JavaBeans must have a zero-argument (empty) constructor



# Example

```
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);  
    }  
}
```

# JavaBeans and JSP

- 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”

# Advantage - Compare

**<%**

**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"/>**

# Creating a JavaBean

- 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 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: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 property, converting to String and inserting to 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 property without converting to String

- Must use a scriptlet

- Format

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

- Example

```
<%
```

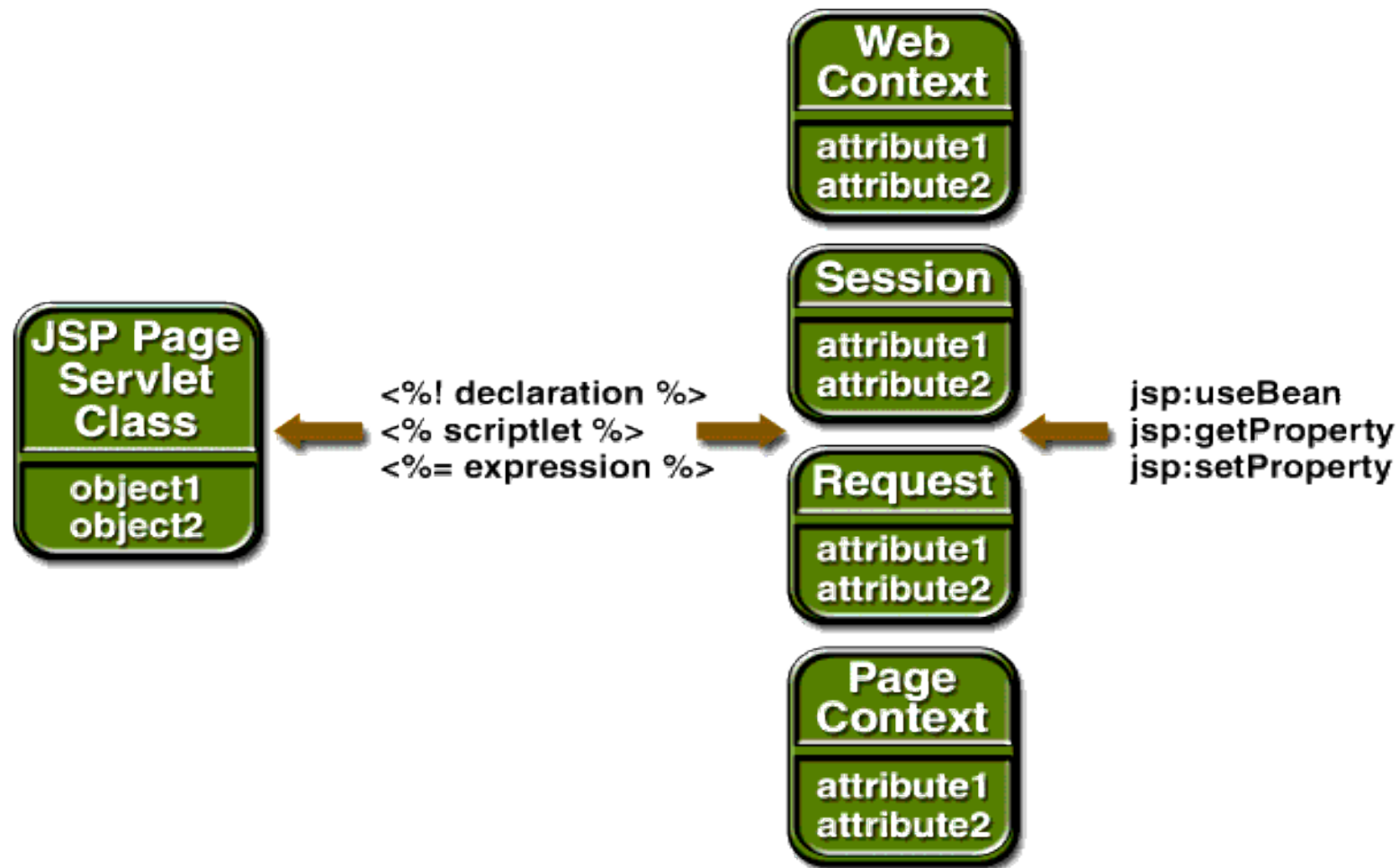
```
// Print a summary of the shopping cart
```

```
int num = cart.getNumberOfItems();
```

```
if (num > 0) {
```

```
%>
```

# Accessing Objects from JSP Page



# Error Handling

- 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

```
<% @ 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)getContext().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>
```