Java Server Pages

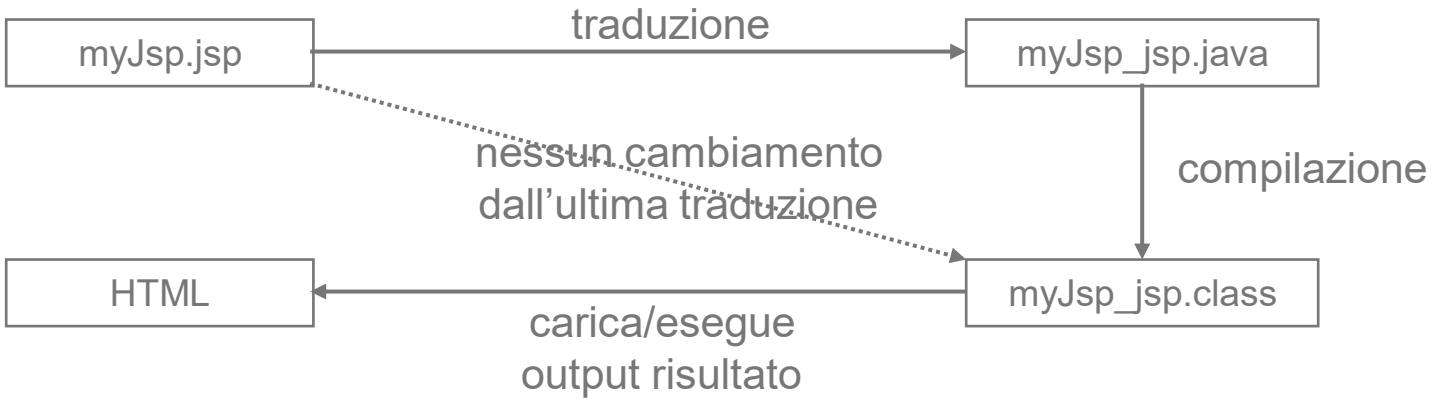
Home Page del corso: http://lia.disi.unibo.it/Courses/twt2021-info/
Versione elettronica: 2.03.JSP.pdf
Versione elettronica: 2.03.JSP-2p.pdf

Java Server Pages

- **Le JSP sono uno dei due componenti di base della tecnologia J2EE, relativamente alla parte Web:**
  - template per la generazione di contenuto dinamico
  - estendono HTML con codice Java custom
- **Quando viene effettuata una richiesta a una JSP:**
  - parte HTML viene direttamente trascritta sullo stream di output
  - **codice Java viene eseguito sul server** per la generazione del contenuto HTML dinamico
  - pagina HTML così formata (parte statica + parte generata dinamicamente) viene restituita al client
- **Assimilabili a linguaggio di script** (avete visto, vero ☺, altri esempi di scripting? PHP, Perl sono altri esempi notevoli…): in realtà **vengono trasformate in servlet dal container**

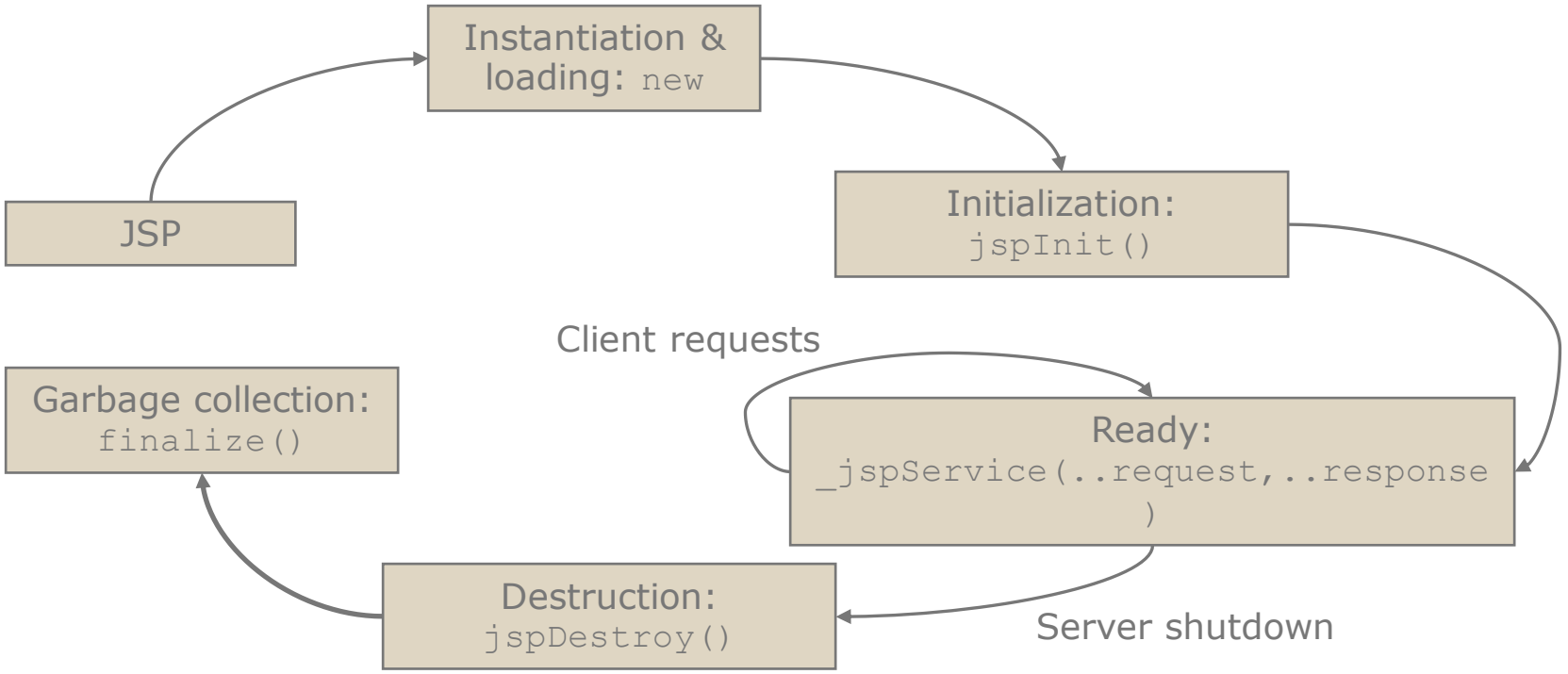
JspServlet

- Le richieste verso JSP sono gestite da una particolare servlet (in Tomcat si chiama JspServlet) che effettua le seguenti operazioni:
  - traduzione della JSP in una servlet
  - compilazione della servlet risultante in una classe
  - esecuzione della JSP
- I primi due passi vengono eseguiti solo quando cambia il codice della JSP



Ciclo di vita delle JSP

Dal momento che JSP sono compilate in servlet, ciclo di vita delle JSP, dopo compilazione, è controllato sempre dal medesimo Web container



Servlet e JSP: perché usare JSP?

Nella servlet logica per la generazione del documento HTML è implementata completamente in Java

- Il processo di generazione delle pagine è time-consuming, ripetitivo e soggetto a errori (sequenza di println())
- L'aggiornamento delle pagine è scomodo

JSP nascono per facilitare la progettazione grafica e l'aggiornamento delle pagine

- Si può separare agevolmente il lavoro fra grafici e programmatori
- Web designer possono produrre pagine senza dover conoscere i dettagli della logica server-side
- La generazione di codice dinamico è implementata sfruttando il linguaggio Java

Servlet o JSP?

- Le JSP non rendono inutili le servlet
- Le servlet forniscono agli sviluppatori delle applicazioni Web un ***completo controllo*** dell'applicazione
- Se si vogliono fornire *contenuti differenziati a seconda di diversi parametri quali l'identità dell'utente, condizioni dipendenti dalla business logic*, etc. è conveniente continuare a lavorare con le servlet
- **JSP rendono viceversa molto semplice presentare documenti HTML o XML (o loro parti) all'utente;** dominanti per la realizzazione di pagine dinamiche semplici e di uso frequente
- (vedremo) Come in tutti i linguaggi di script che poi generano codice (tipicam non visibile allo sviluppatore), ***maggiori problemi di controllo della correttezza e testing***

Come funzionano le JSP

- Ogni volta che arriva una request, server compone dinamicamente il contenuto della pagina
- Ogni volta che incontra un tag <%...%>
  - valuta l'espressione Java contenuta al suo interno
  - inserisce al suo posto il risultato dell'espressione
- Questo meccanismo permette di generare pagine dinamicamente

Anche considerazioni sul flusso

- Ricordiamoci come funziona HTTP e quale è la struttura delle pagine HTML
- **Il Client si aspetta di ricevere tutto response header prima di response body:**
  - JSP deve effettuare tutte le modifiche all'header (ad es. modifica di cookie) prima di cominciare a creare body
- **Una volta che Web server comincia a restituire risposta non può più interrompere il processo, altrimenti browser mostra solo la frazione parziale che ha ricevuto:**
  - se JSP ha cominciato a produrre output non si può più effettuare forward ad un'altra JSP (esattamente in parallelo con quanto già visto per servlet)

Esempio: Hello world

JSP, denominata helloWorld.jsp, che realizza il classico esempio “Hello World!” in modo parametrico:

```
<html>
  <body>
  <% String visitor=request.getParameter("name");
     if (visitor == null) visitor = "World"; %>
     Hello, <%= visitor %>!
  </body>
</html>
```

http://myHost/myWebApp/helloWord.jsp

```
<html>
  <body>
     Hello, World!
  </body>
</html>
```

http://myHost/myWebApp/helloWord.jsp?name=Mario

```
<html>
  <body>
     Hello, Mario!
  </body>
</html>
```

Tag

Le parti variabili della pagina sono contenute all'interno di tag speciali

- Sono possibili due tipi di sintassi per questi tag:
  - Scripting-oriented tag
  - XML-Oriented tag

Le **scripting-oriented tag** sono definite da delimitatori entro cui è presente lo scripting (self-contained)

- Sono di quattro tipi:
  - <%! %> Dichiarazione
  - <%= %> Espressione
  - <% %> Scriptlet
  - <%@ %> Direttiva

XML-oriented tag

XML-oriented tag seguono la sintassi XML

- Sono presenti XML tag equivalenti ai delimitatori visti nella pagina precedente
  - `<jsp:declaration>`*declaration*`</jsp:declaration>`
  - `<jsp:expression>`*expression*`</jsp: expression>`
  - `<jsp:scriptlet>`*java_code*`</jsp:scriptlet>`
  - `<jsp:directive.`*dir_type dir_attribute*` />`
- Nel seguito useremo scripting-oriented tag che sono più diffusi

Dichiarazioni

- Si usano i delimitatori <%! e %> per dichiarare variabili e metodi
- Variabili e metodi dichiarati possono poi essere referenziati in qualsiasi punto del codice JSP
- **I metodi diventano metodi della servlet quando la pagina viene tradotta**

```
<%! String name = "Paolo Rossi";
    double[] prices = {1.5, 76.8, 21.5};

    double getTotal() {
       double total = 0.0;
       for (int i=0; i<prices.length; i++)
         total += prices[i];
       return total;
    }
%>
```

Espressioni

- Si usano i delimitatori `<%=` e `%>` per valutare espressioni Java
- **Risultato dell'espressione viene convertito in stringa inserito nella pagina al posto del tag**

Continuando l'esempio della pagina precedente:

JSP

```
<p>Sig. <%=name%>,</p>
<p>l'ammontare del suo acquisto è: <%=getTotal()%> euro.</p>
<p>La data di oggi è: <%=new Date()%></p>
```

Pagina HTML risultante

```
<p>Sig. Paolo Rossi,</p>
<p>l'ammontare del suo acquisto è: 99.8 euro.</p>
<p>La data di oggi è: Tue Feb 20 11:23:02 2010</p>
```

Scriptlet

- Si usano <% e %> per aggiungere un frammento di codice Java eseguibile alla JSP (**scriptlet**)
- Lo scriptlet consente tipicamente di inserire logiche di controllo di flusso nella produzione della pagina
- La combinazione di tutti gli scriptlet in una determinata JSP deve definire un blocco logico completo di codice Java

```
<% if (userIsLogged) { %>
  <h1>Benvenuto Sig. <%=name%></h1>
<% } else { %>
  <h1>Per accedere al sito devi fare il login</h1>
<% } %>
```

Direttive

- Sono comandi JSP valutati a tempo di compilazione
- Le più importanti sono:
  - page: permette di importare package, dichiarare pagine d'errore, **definire modello di esecuzione JSP relativamente alla concorrenza** (ne discuteremo a breve), ecc.
  - include: include un altro documento
  - taglib: carica una libreria di custom tag implementate dallo sviluppatore
- Non producono nessun output visibile

```
<%@ page info="Esempio di direttive" %>
<%@ page language="java" import="java.net.*" %>
<%@ page import="java.util.List, java.util.ArrayList" %>
<%@ include file="myHeaderFile.html" %>
```

La direttiva page

La direttiva **page** definisce una serie di attributi che si applicano all'intera pagina

Sintassi:

```
<%@ page
  [ language="java" ]
  [ extends="package.class" ]
  [ import="{package.class | package.*}, ..." ]
  [ session="true | false" ]
  [ buffer="none | 8kb | sizekb" ]
  [ autoFlush="true | false" ]
  [ isThreadSafe="true | false" ]
  [ info="text" ]
  [ errorPage="relativeURL" ]
  [ contentType="mimeType [ ;charset=characterSet ]"|
    "text/html ; charset=ISO-8859-1" ]
  [ isErrorPage="true | false" ]
%>
```

N.B. valori sottolineati sono quelli di default

Attributi di page - 1

- `language="java"` linguaggio di scripting utilizzato nelle parti dinamiche, allo stato attuale l'unico valore ammesso è "java"
- `import="{package.class|package.*},..."` lista di package da importare. Gli import più comuni sono impliciti e non serve inserirli (`java.lang.*`, `javax.servlet.*`,`javax.servlet.jsp.*`, `javax.servlet.http.*`)
- `session="true|false"` : indica se la pagina fa uso della sessione (altrimenti non si può usare `session`)
- `buffer="none|8kb|sizekb"` dimensione in KB del buffer di uscita
- `autoFlush="true|false"` dice se il buffer viene svuotato automaticamente quando è pieno. Se il valore è false viene generata un'eccezione quando il buffer è pieno

Attributi di page - 2

- `isThreadSafe="true|false"` **indica se il codice contenuto nella pagina è thread-safe. Se vale** `false` **le chiamate alla JSP vengono serializzate**
- `info="text"` **testo di commento.** Può essere letto con il metodo Servlet.getServletInfo()
- `errorPage="relativeURL"` **indirizzo della pagina a cui vengono inviate le eccezioni**
- `isErrorPage="true|false"` **indica se JSP corrente è una pagina di errore.** Si può utilizzare l'oggetto eccezione solo se l'attributo è true
- `contentType="mimeType [;charset=charSet ]" | "text/html;charset=ISO-8859-1"` **indica il tipo MIME e il codice di caratteri usato nella risposta**

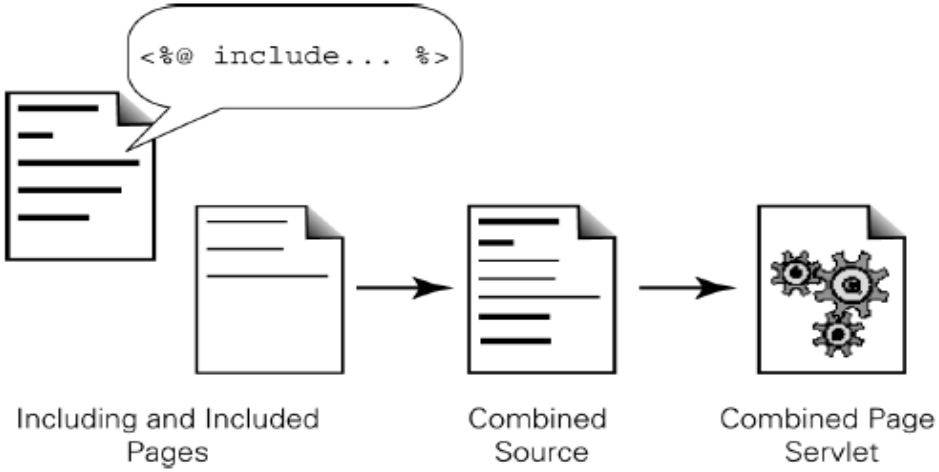
La direttiva include

Sintassi: `<%@ include file = "localURL"%>`

Serve ad includere il contenuto del file specificato

- **È possibile nidificare un numero qualsiasi di inclusioni**
- **L'inclusione viene fatta a tempo di compilazione:**
  eventuali modifiche al file incluso non determinano una ricompilazione della pagina che lo include

Esempio: `<%@ include file="/shared/copyright.html"%>`



Direttiva taglib

- JSP permettono di definire tag custom oltre a quelli predefiniti
- Una `taglib` è una collezione di questi tag non standard, realizzata mediante una classe Java

Sintassi: `<%@ uri="tagLibraryURI" prefix="tagPrefix"%>`

Data la sua scarsa rilevanza implementativa per questo corso, ne vedremo il dettaglio solo alla fine di questo blocco di lucidi…

Built-in objects (con scope differenziati)

- Le specifiche JSP definiscono 9 **oggetti built-in** (o **impliciti**) utilizzabili senza dover creare istanze
- Rappresentano utili riferimenti ai corrispondenti oggetti Java veri e propri presenti nella tecnologia servlet

Oggetto	Classe/Interfaccia
`page`	`javax.servlet.jsp.HttpJspPage`
`config`	`javax.servlet.ServletConfig`
`request`	`javax.servlet.http.HttpServletRequest`
`response`	`javax.servlet.http.HttpServletResponse`
`out`	`javax.servlet.jsp.JspWriter`
`session`	`javax.servlet.http.HttpSession`
`application`	`javax.servlet.ServletContext`
`pageContext`	`javax.servlet.jsp.PageContext`
`exception`	`Java.lang.Throwable`

L'oggetto page

L'oggetto page rappresenta l'istanza corrente della servlet

- Ha come tipo l'interfaccia HTTPJspPage che discende da JSP page, la quale a sua volta estende Servlet
- **Può quindi essere utilizzato per accedere a tutti i metodi definiti nelle servlet**

JSP

```
<%@ page info="Esempio di uso page." %>
<p>Page info:
  <%=page.getServletInfo() %>
</p>
```

Pagina HTML

```
<p>Page info: Esempio di uso di page</p>
```

Oggetto config

Contiene la configurazione della servlet (parametri di inizializzazione)

- Poco usato in pratica in quanto in generale nelle JSP sono poco usati i parametri di inizializzazione
- Vedi anche (più avanti) stato a livello di applicazione in application/ServletContext

Metodi di `config`:

- `getInitParameterName()` : restituisce tutti i nomi dei parametri di inizializzazione
- `getInitParameter(name)` : restituisce il valore del parametro passato per nome

Oggetto request

Rappresenta la richiesta alla pagina JSP

- È il parametro request passato al metodo service() della servlet
- Consente l'accesso a tutte le informazioni relative alla richiesta HTTP:
  - indirizzo di provenienza, URL, headers, cookie, parametri, ecc.

```
<% String xStr = request.getParameter("num");
try
{
   long x = Long.parseLong(xStr); %>
   Fattoriale: <%= x %>! = <%= fact(x) %>
<%}
catch (NumberFormatException e) { %>
Il parametro <b>num</b>non contiene un valore intero.
<%} %>
```

Alcuni metodi di request

- `String getParameter(String parName)` restituisce valore di un parametro individuato per nome
- `Enumeration getParameterNames()` restituisce l'elenco dei nomi dei parametri
- `String getHeader(String name)` restituisce il valore di un header individuato per nome sotto forma di stringa
- `Enumeration getHeaderNames()` elenco nomi di tutti gli header presenti nella richiesta
- `Cookie[] getCookies()` restituisce un array di oggetti cookie che client ha inviato alla request

Per l'elenco completo dei metodi di request vedere parte su servlet (i metodi sono gli stessi)

Oggetto response

Oggetto legato all'I/O della pagina JSP

- Rappresenta la risposta che viene restituita al client
- Consente di inserire nella risposta diverse informazioni:
  - content type ed encoding
  - eventuali header di risposta
  - URL Rewriting
  - i cookie

```
<%response.setDateHeader("Expires", 0);
  response.setHeader("Pragma", "no-cache");
  if (request.getProtocol().equals("HTTP/1.1"))
  {
    response.setHeader("Cache-Control", "no-cache");
  }
%>
```

Metodi di response

- `public void setHeader(String headerName, String headerValue)` imposta header
- `public void setDateHeader(String name, long millisecs)` imposta data
- `addHeader, addDateHeader, addIntHeader` aggiungono nuova occorrenza di un dato header
- `setContentType` determina content-type
- `addCookie` consente di gestire i cookie nella risposta
- `public PrintWriter getWriter`: restituisce uno stream di caratteri (un'istanza di `PrintWriter`)
- `public ServletOutputStream getOuputStream()`: restituisce uno stream di byte (un'istanza di `ServletOutputStream`)

Oggetto out

Oggetto legato all'I/O della pagina JSP

- È uno stream di caratteri e rappresenta lo stream di output della pagina

Esempio:

```
<p>Conto delle uova
  <%
    int count = 0;
    while (carton.hasNext())
    {
      count++;
      out.print(".");
    }
  %>
<br/>
Ci sono <%= count %> uova.
</p>
```

Metodi dell'oggetto out

- `isAutoFlush()`: dice se output buffer è stato impostato in modalità autoFlush o meno
- `getBufferSize()`: restituisce dimensioni del buffer
- `getRemaining()` indica quanti byte liberi ci sono nel buffer
- `clearBuffer()` ripulisce il buffer
- `flush()` forza l'emissione del contenuto del buffer
- `close()` fa flush e chiude stream

Oggetto session

Oggetto che fornisce informazioni sul contesto di esecuzione della JSP in termini di SESSIONE UTENTE

- L'attributo **session** della direttiva **page** deve essere true affinché JSP partecipi alla sessione

```
<% UserLogin userData = new UserLogin(name, password);
   session.setAttribute("login", userData);
%>
<%UserLogin userData=(UserLogin)session.getAttribute("login");
  if (userData.isGroupMember("admin")) {
     session.setMaxInactiveInterval(60*60*8);
  } else {
     session.setMaxInactiveInterval(60*15);
  }
%>
```

Metodi di session

- `String getID()` restituisce ID di una sessione
- `boolean isNew()` dice se sessione è nuova
- `void invalidate()` permette di invalidare (distruggere) una sessione
- `long getCreationTime()` ci dice da quanto tempo è attiva la sessione (in ms)
- `long getLastAccessedTime()` ci dice quando è stata utilizzata l'ultima volta

Oggetto application

Oggetto che fornisce informazioni su contesto di esecuzione della JSP con scope di visibilità comune a tutti gli utenti (è `ServletContext`)

- Rappresenta la Web application a cui JSP appartiene
- Consente di interagire con l'ambiente di esecuzione:
  - fornisce la versione di JSP Container
  - garantisce l'accesso a risorse server-side
  - permette accesso ai parametri di inizializzazione relativi all'applicazione
  - consente di gestire gli attributi di un'applicazione

Oggetto pageContext

Oggetto che fornisce informazioni sul contesto di esecuzione della pagina JSP

- Rappresenta l'insieme degli oggetti built-in di una JSP
  - Consente accesso a tutti gli oggetti impliciti e ai loro attributi
  - *Consente trasferimento del controllo ad altre pagine*

Nota: poco usato in caso di scripting, più utile per costruire custom tag (quindi lo utilizzeremo raramente all'interno di questo corso…)

Oggetto exception

Oggetto connesso alla gestione degli errori

- **Rappresenta l'eccezione che non viene gestita da nessun blocco catch**
- Non è automaticamente disponibile in tutte le pagine ma solo nelle Error Page (quelle dichiarate con l'attributo errorPage impostato a true)

Esempio:

```
<%@ page isErrorPage="true" %>
  <h1>Attenzione!</h1>
  E' stato rilevato il seguente errore:<br/>
  <b><%= exception %></b><br/>
  <%
    exception.printStackTrace(out);
  %>
```

Azioni

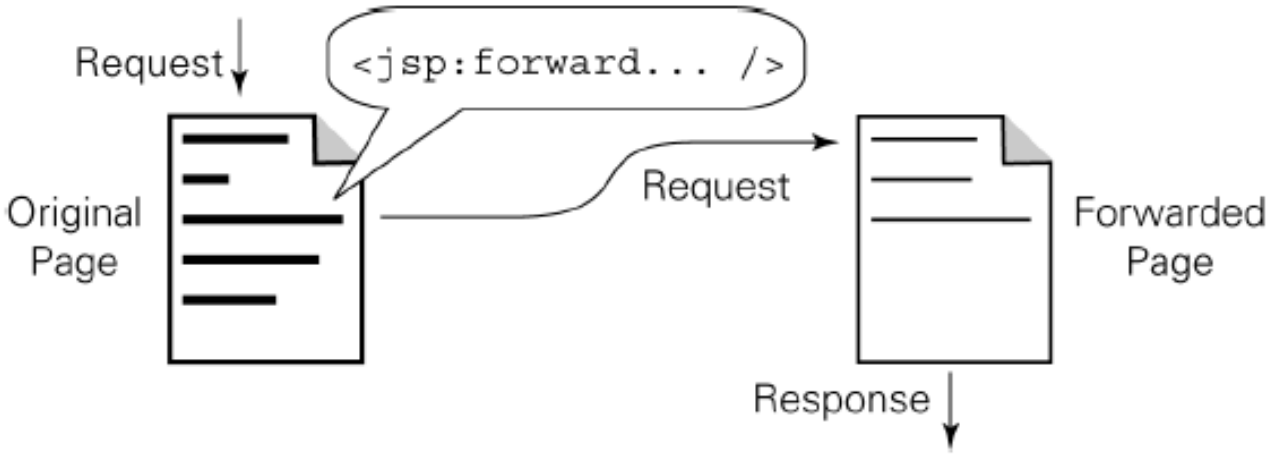
Le azioni sono comandi JSP tipicamente per l'interazione con altre pagine JSP, servlet, o componenti JavaBean; sono espresse usando sintassi XML

- **Sono previsti 6 tipi di azioni definite dai seguenti tag:**
  - `useBean`: istanzia JavaBean e gli associa un identificativo
  - `getProperty`: ritorna property indicata come oggetto
  - `setProperty`: imposta valore della property indicata per nome
  - `include`: include nella JSP contenuto generato dinamicamente da un'altra pagina locale
  - `forward`: cede controllo ad un'altra JSP o servlet
  - `plugin`: genera contenuto per scaricare plug-in Java se necessario

```
<html>
  <body>
    <jsp:useBean id="myBean" class="it.unibo.deis.my.HelloBean"/>
    <jsp:setProperty name="myBean" property="nameProp" param="value"/>
    Hello, <jsp:getProperty name="myBean" property="nameProp"/>!
  </body>
</html>
```

Azioni: forward

- Sintassi: `<jsp:forward page="localURL" />`
- Consente trasferimento del controllo dalla pagina JSP corrente ad un'altra pagina sul server locale
  - L'attributo page definisce l'URL della pagina a cui trasferire il controllo
  - La request viene completamente trasferita in modo trasparente per il client



Azioni: forward

- È possibile generare dinamicamente attributo page

  `<jsp:forward page='<%="message"+statusCode+".html"%>'/>`
- Oggetti `request`, `response` e `session` della pagina d'arrivo sono gli stessi della pagina chiamante, ma viene istanziato un nuovo oggetto `pageContext`
- **Attenzione:** forward è possibile soltanto se non è stato emesso alcun output
- È possibile aggiungere parametri all'oggetto request della pagina chiamata utilizzando il tag **`<jsp:param>`**

```
<jsp:forward page="localURL">
  <jsp:param name="parName1" value="parValue1"/>
  ...
  <jsp:param name="parNameN" value="parValueN"/>
</jsp:forward>
```

Azioni: include

Sintassi: `<jsp:include page="localURL" flush="true" />`

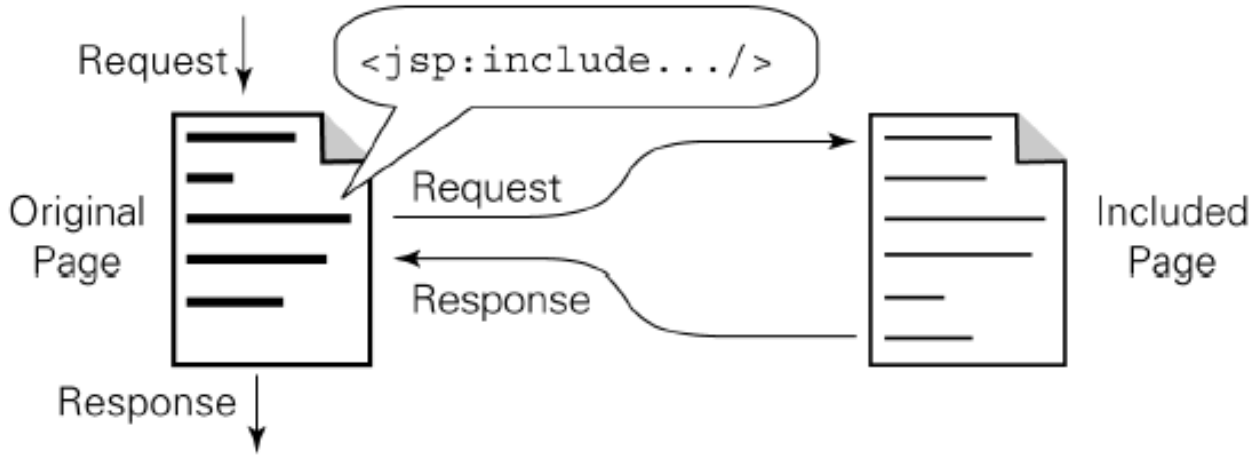
Consente di includere il contenuto generato dinamicamente da un'altra pagina locale all'interno dell'output della pagina corrente

- Trasferisce temporaneamente controllo ad un'altra pagina
- L'attributo page definisce l'URL della pagina da includere
- L'attributo flush stabilisce se sul buffer della pagina corrente debba essere eseguito flush prima di effettuare l'inclusione
- Gli oggetti session e request per pagina da includere sono gli stessi della pagina chiamante, ma viene istanziato un nuovo contesto di pagina

Azioni: include

È possibile aggiungere parametri all'oggetto request della pagina inclusa utilizzando il tag `<jsp:param>`

```
<jsp:include page="localURL" flush="true">
  <jsp:param name="parName1" value="parValue1"/>
  ...
  <jsp:param name="parNameN" value="parValueN"/>
</jsp:include>
```



JSP e modello a componenti

Scriptlet ed espressioni consentono uno sviluppo centrato sulla pagina

- Questo modello non consente una forte separazione tra logica applicativa e presentazione dei contenuti
- Applicazioni complesse necessitano di maggiore modularità ed estensibilità, tramite una architettura a più livelli

A tal fine, JSP consentono anche sviluppo basato su un modello a componenti

Il modello a componenti:

- consente di avere una maggiore separazione fra logica dell'applicazione e contenuti
- Permette di costruire architetture molto più articolate
- Fate mente locale su quanto avete già visto in altri corsi sui componenti…

JavaBeans (già visti? Solo reminder…)

JavaBeans è il modello di "base" per componenti Java, il più semplice

- Un **JavaBean**, o semplicemente **bean**, non è altro che una classe Java dotata di alcune caratteristiche particolari:
  - Classe **public**
  - Ha un costruttore **public** di default (senza argomenti)
  - Espone proprietà, sotto forma di coppie di metodi di accesso (accessors) costruiti secondo le regole che abbiamo appena esposto (get… set…)
  - Espone eventi con metodi di registrazione che seguono regole precise

Proprietà

- Le proprietà sono elementi dello stato del componente che vengono esposti in modo protetto
  - In alcuni linguaggi (ad esempio C#) esiste sintassi specifica per definire le proprietà
- In altri (come Java) le proprietà sono solo una convenzione: sono coppie di metodi di accesso che seguono regole di denominazione
- La proprietà *prop* è definita da due metodi `getProp()` e `setProp()`
- Il tipo del parametro di `setProp()` e del valore di ritorno di `getProp()` devono essere uguali e rappresentano il tipo della proprietà (può essere un tipo primitivo o una qualunque classe Java)
  - Per esempio `void setLength(int Value)` e `int getLength()` identificano proprietà `length` di tipo `int`

Proprietà

- Se definiamo solo il metodo get avremo una proprietà in sola lettura (read-only)
- Le proprietà di tipo boolean seguono una regola leggermente diversa: metodo di lettura ha la forma `isProp()` anziché `getProp()`
  - Per esempio la proprietà `empty` sarà rappresentata dalla coppia `void setEmpty(boolean value)` e `boolean isEmpty()`
- Esiste anche la possibilità di definire proprietà indicizzate per rappresentare collezioni di valori (pseudoarray). In questo caso sia get che set prevedono un parametro che ha la funzione di indice
  - Ad es. `String getItem(int index)` e `setItem(int Index, String value)` definiscono la proprietà indicizzata `String item[]`

Componenti e container

*I componenti vivono all'interno di contenitori (**component container**) che gestiscono:*

- *tempo di vita dei singoli componenti*
- *collegamenti fra componenti e resto del sistema*

- I contenitori non conoscono a priori i componenti che devono gestire e quindi interagiscono con loro mediante meccanismi di tipo dinamico (spesso reflection)
- Un contenitore per JavaBean prende il nome di **bean container**
- Un **bean container** è in grado di interfacciarsi con i **bean** utilizzando **Java Reflection** che fornisce strumenti di **introspezione** e di **dispatching**
- L'obbligo del costruttore di default ha proprio lo scopo di consentire creazione dinamica delle istanze

Esempio

Creiamo un bean che espone due proprietà in sola lettura (ore e minuti) e ci dà l'ora corrente

```
import java.util.*
public class CurrentTimeBean
{
  private int hours;
  private int minutes;
  public CurrentTimeBean()
  {
    Calendar now = Calendar.getInstance();
    this.hours = now.get(Calendar.HOUR_OF_DAY);
    this.minutes = now.get(Calendar.MINUTE);
  }
  public int getHours()
  { return hours; }
  public int getMinutes()
  { return minutes; }
}
```

JSP e JavaBean

JSP prevedono una serie di tag per agganciare un bean e utilizzare le sue proprietà all'interno della pagina

Tre tipi:

- Tag per creare un riferimento al bean (creazione di un'istanza)
- Tag per impostare il valore delle proprietà del bean
- Tag per leggere il valore delle proprietà del bean e inserirlo nel flusso della pagina

Esempio di uso di bean

```
<jsp:useBean id="user" class="RegisteredUser" scope="session"/>

<jsp:useBean id="news" class="NewsReports" scope="request">
 <jsp:setProperty name="news" property="category" value="fin."/>
 <jsp:setProperty name="news" property="maxItems" value="5"/>
</jsp:useBean>

<html>
  <body>
    <p>Bentornato
    <jsp:getProperty name="user" property="fullName"/>,
    la tua ultima visita è stata il
    <jsp:getProperty name="user" property="lastVisitDate"/>.
    </p>
    <p>
    Ci sono <jsp:getProperty name="news" property="newItems"/>
    nuove notizie da leggere.</p>
  </body>
</html>
```

Tag jsp:useBean

Sintassi:
```
<jsp:useBean id="beanName" class="class"
    scope="page|request|session|application"/>
```

- Inizializza e crea il riferimento al bean
- Gli attributi principali sono id, class e scope
  - **id** è il nome con cui l'istanza del bean verrà indicata nel resto della pagina
  - **class** è classe Java che definisce il bean
  - **scope** definisce ambito di accessibilità e tempo di vita dell'oggetto (default = page)

Tempo di vita dei bean

- Per default ogni volta che una pagina JSP viene richiesta e processata viene creata un'istanza del bean (scope di default = page)
- Con l'attributo scope è possibile estendere la vita del bean oltre la singola richiesta:

Scope	Accessibilità	Tempo di vita
page	Solo la pagina corrente	Fino a quando la pagina viene completata o fino al forward
request	La pagina corrente, quelle incluse e quelle a cui si fa forward	Fino alla fine dell'elaborazione della richiesta e restituzione della risposta
session	Richiesta corrente e tutte le altre richieste dello stesso client	Tempo di vita della sessione
application	Richiesta corrente e ogni altra richiesta che fa parte della stessa applicazione	Tempo di vita dell'applicazione

Tag jsp:getProperty

Sintassi: `<jsp:getProperty name="beanName" property="propName"/>`

- Consente l'accesso alle proprietà del bean
- Produce come output il valore della proprietà del bean
- Il tag non ha mai body e ha solo 2 attributi:
  - `name`: nome del bean a cui si fa riferimento
  - `property`: nome della proprietà di cui si vuole leggere il valore

Esempio 1: uso di CurrentTimeBean

JSP

```
<jsp:useBean id="time" class="CurrentTimeBean"/>
<html>
 <body>
    <p>Sono le ore
     <jsp:getProperty name="time" property="hours"/> e
     <jsp:getProperty name="time" property="minutes"/> minuti.
    </p>
  </body>
</html>
```

Output HTML

```
<html>
  <body>
    <p>Sono le ore
     12 e 18 minuti.></p>
  </body>
</html>
```

Esempio 2: un caso un po' più complesso

JSP

```
<jsp:useBean id="style" class="beans.CorporateStyleBean"/>
<html>
  <body bgcolor="<jsp:getProperty name="style" property="color"/>">
    <center>
      <img src="<jsp:getProperty name="style" property="logo"/>">
      Welcome to Big Corp!
    </center>
  </body>
</html>
```

Output HTML

```
<html>
  <body bgcolor="pink">
    <center>
      <img src="http://imageserver/logo.gif">
      Welcome to Big Corp!
    </center>
  </body>
</html>
```

Tag jsp:setProperty

Sintassi: `<jsp:setProperty name="beanName" property="propName" value="propValue"/>`

Consente di modificare il valore delle proprietà del bean

Esempi:

```
<jsp:setProperty name="user"
  property="daysLeft" value="30"/>

<jsp:setProperty name="user"
  property="daysLeft" value="<%=15*2%>"/>
```

Proprietà indicizzate

- I tag per JavaBean non supportano proprietà indicizzate
- Però un bean è un normale oggetto Java: è quindi possibile accedere a variabili e metodi

Esempio:

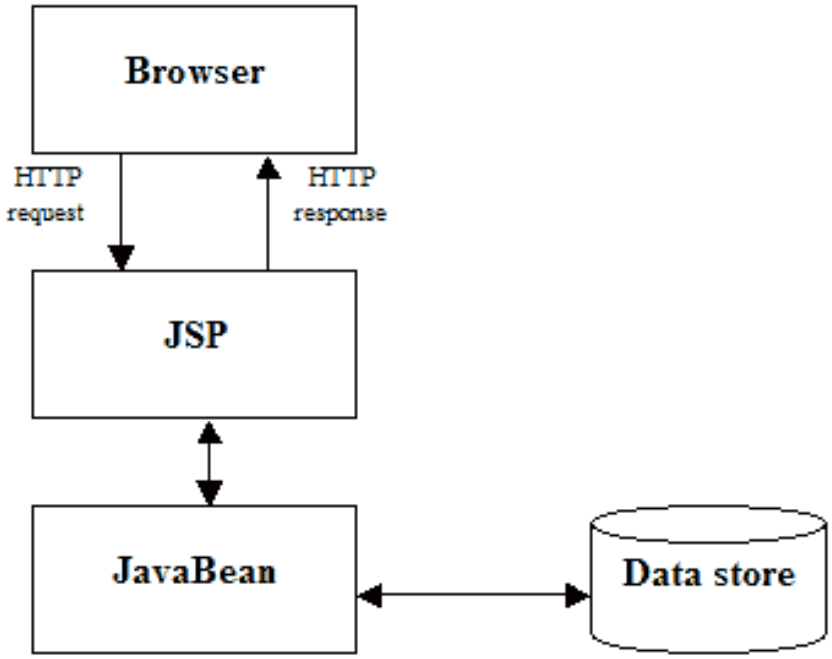
```
<jsp:useBean id="weather" class="weatherForecasts"/>

<p><b>Previsioni per domani:</b>:
  <%= weather.getForecasts(0)%>
  </p>
<p><b>Resto della settimana:</b>
<ul>
   <% for (int index=1; index < 5; index++) { %>
     <li><%= weather.getForecasts(index) %></li>
   <% } %>
</ul>
</p>
```

JSP + JavaBean = Model 1

L'architettura J2EE a due livelli costituita da

- JSP per il livello di presentazione
- JavaBean per il livello di business logic

viene denominata **Model 1**



JSP e servlet in un'applicazione Web

Qualche domanda, spero con risposta ovvia ☺:

- Possiamo avere una applicazione Web che include più servlet e più JSP?
- Posso fare «forward» e «include» da JSP a servlet e da servlet a JSP?
- Posso vedere un attributo di sessione definito in una servlet all'interno di una JSP?
- Posso vedere un attributo di applicazione definito in una servlet all'interno di una JSP?
- Posso vedere un Javabean definito in una JSP all'interno di una servlet? Con quali scope?
- …

Come?

Fate qualche prova ed esercizio…

Appendice su: Custom tag e tag libraries

- JSP permettono di definire tag personalizzati (**custom tag**) che estendono quelli predefiniti
- Una **taglib** è una collezione di questi tag non standard, realizzata mediante una classe Java
- Per utilizzarla si usa la direttiva taglib con la sintassi:
  `<%@ taglib uri="tagLibraryURI" prefix="tagPre"%>`
  - L'attributo `uri` fa riferimento ad un file xml, con estensione tld (tag library descriptor), che contiene informazioni sulle classi che implementano i tag
  - L'attributo `prefix` indica il prefisso da utilizzare nei tag che fanno riferimento alla tag library (tag library è un namespace)

Definizione di taglib

Per definire una tag library occorrono due elementi:

- File **TLD** (Tag Lib Definition) che specifica i singoli Tag e a quale "classe" corrispondono
- Le classi che effettivamente gestiscono i tag

File TLD è un file XML che specifica:

- i tag che fanno parte della libreria
- i loro eventuali attributi
- "body" del tag (se esiste)
- classe Java che gestisce il tag

⇨ **Dovremo quindi sviluppare le classi che implementano il comportamento dei tag**

- Una singola libreria può contenere centinaia di tag o uno solo

Esempio di taglib

Un semplice esempio di file TLD è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>hellolib</shortname>
  <uri>hellodir</uri>
  <tag>
    <name>helloWorld</name>
    <tagclass>helloTagClass</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>who</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Esempio di taglib

Questo file stabilisce che:

- la versione della libreria è la 1.0 (`tlibversion`)
- il nome della libreria è `hellolib` (attributo `shortname`)
- i file .class si trovano nella directory `hellodir` (attributo `uri`)

Viene definito un solo tag denominato `helloWorld`:

- senza contenuto (`bodycontent` è `empty`)
- con solo "attributo“, denominato `who`, obbligatorio (`required` è `true`)
- "gestito" dalla classe denominata `helloTagClass`

Uso della taglib

- Innanzitutto inseriamo nella JSP la direttiva che include la libreria di tag:

```
<%@ taglib uri="hellolib.tld" prefix="htl" %>
```

- Il prefisso definisce un namespace e quindi elimina le eventuali omonimie causate dall'inclusione di più librerie. Possiamo quindi usare il tag con la sintassi:

```
<htl:helloWorld who="Mario">
```

Implementazione del tag

- Dobbiamo scrivere una apposita classe Java che estende `TagSupport`
- `TagSupport` è la classe base per i tag "semplici", per quelli complessi sono disponibili altre classi base
- La classe deve implementare
  - i metodi `doStartTag()` e `doEndTag()`
  - Una coppia di metodi di accesso (`setAttrName()` e `getAttrName()` per ogni attributo
- `doStartTag()` utilizza l'oggetto out restituito da PageContext per scrivere nell'output della pagina e se tag non ha nessun "body" deve ritornare come valore la costante `SKIP_BODY`
- `doEndTag()` restituisce usualmente la costante `EVAL_PAGE` che indica che, dopo il tag, prosegue la normale elaborazione della pagina

Implementazione

```
import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class helloTagClass extends TagSupport
{
  private String who;
  public int doStartTag() throws JspException
  {
    try
      pageContext.getOut().println("Hello"+who+"<br>");
    catch( Exception e )
      throw new JspException( "taglib:" + e.getMessage() );
    return SKIP_BODY;
  }
  public int doEndTag()
  { return EVAL_PAGE; }
  public void setWho(String value)
  { who = value; }
  public String getWho()
  { return who; }
}
```

Esempio di uso

Scriviamo una versione di HelloWorld che utilizza la nostra tag library:

```
<%@ taglib uri="hellolib.tld" prefix="htl" %>
<html>
  <body>
    <htl:helloWorld
        who=<%=request.getParameter("name")%>
  </body>
</html>
```