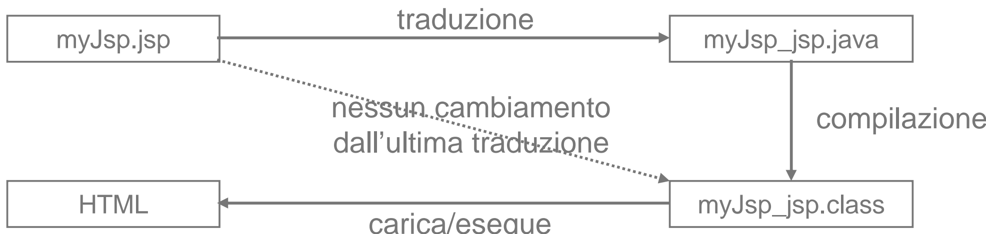
Java Server Pages

Java Server Pages

- Le JSP sono uno dei due componenti di base della tecnologia J2EE, relativamente alla parte web:
  - Sono template per la generazione di contenuto dinamico
  - Estendono HTML con codice Java custom.
- Quando viene effettuata una richiesta ad una JSP:
  - la parte HTML viene direttamente trascritta sullo stream di output
  - il codice Java viene eseguito sul server per la generazione del contenuto HTML dinamico
  - la pagina HTML così formata (parte statica e + parte generata dinamicamente) viene restituita al client
- Sono assimilabili ad un linguaggio di scripting: in realtà vengono trasformate in servlet dal container

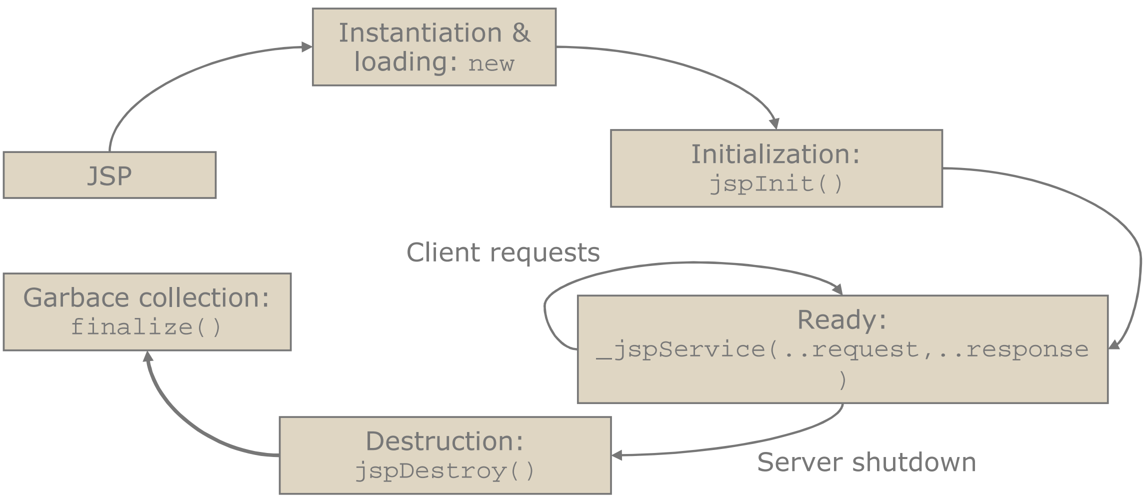
JspServlet

- 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 le JSP sono compilate in servlet, il ciclo di vita delle JSP è controllato dal web container



Servlet e JSP

- Nella servlet la logica per la generazione del documento HTML è implementata completamente in Java
  - Il processo di generazione delle pagine è tedioso e soggetto ad errori (una sequenza di println())
  - L'aggiornamento delle pagine è scomodo
- Le JSP nascono per facilitare la progettazione grafica e l'aggiornamento delle pagine
  - Si può separare agevolmente il lavoro fra grafici e programmatori
  - I 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 lavorare con le servlet
- Le JSP rendono viceversa molto semplice presentare documenti HTML o XML all'utente

Come funzionano le JSP

- Ogni volta che arriva una request il 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

Considerazioni sul flusso

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

Esempio: Hello world

- Creiamo una 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

- La parti variabili della pagina sono contenute all’interno 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

- Le **XML-oriented tag**: seguono la sintassi XML.
- Sono presenti XML tag equivalenti ai delimitatori visti nella pagina precedente
- Rispettivamente:
  - `<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 le scripting-oriented tag che sono più diffuse

Dichiarazioni

- Si usano i delimitatori `<%!` e `%>` per dichiarare variabili e metodi
- Le variabili e i 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 vien 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 il modello di esecuzione della JSP relativamente alla concorrenza, 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
- La sua sintassi è:

```
<%@ 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. I valori sottolineati sono quelli di default

Attributi di page - 1

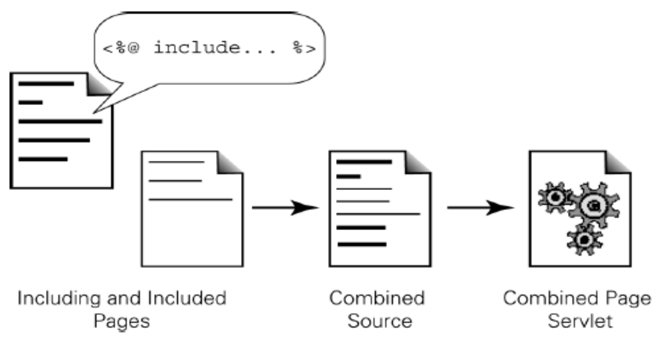
- `language="java"` linguaggio di scripting utilizzato nelle parti dinamiche, allo stato attuale l'unico valore ammesso è "java"
- `import="{package.class|package.*},…"` lista di package da importare. Gli import più comuni sono impliciti e non serve inserirli (`java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `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 kilobyte 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 sel la 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
- E' 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

- Le 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"%>`
- L'attributo uri fa riferimento ad un file xml, con estensione tld (tag library descriptor), che contiene informazioni sulle classe che implementa i tag.
- Nell'esempio sotto riportato, nel descrittore associato alla taglib è presente un elemento <endProgram> che definisce la classe Java associata al tag.
  - direttiva: `<%@ taglib uri="/EncomTags" prefix="mcp"%>`
  - utilizzo: `<mcp:endProgram/>`

Built-in objects

- Le specifiche JSP definiscono 8 **oggetti built-in** (o **impliciti**) utilizzabili senza dover di creare istanze
- Rappresentano utili riferimenti ai corrispondenti oggetti presenti nelle 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 quindi utilizzato per accedere a tutti i metodi definiti nelle servlet

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

JSP

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
- 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
- E' 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 il 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 dei nomi di tutti gli header presenti nella richiesta
- `Cookie[] getCookies()` restituisce un array di oggetti cookie che il client ha inviato alla request
- Per l'elenco completo vedere parte su servlet

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:
  - il content type e l'enconding
  - 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 un header
- `public void setDateHeader(String name, long millisecs)` imposta la data
- `addHeader, addDateHeader, addIntHeader` aggiungono una nuova occorrenza di un dato header
- `setContentType` determina il 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
- E' 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 l'output buffer è stato impostato in modalità autoFlush o meno
- `getBufferSize()`: restituisce le dimensioni del buffer
- `getRemaining()` indica quandi byte liber ci sono nel buffer
- `clearBuffer()` ripulisce il buffer
- `flush()` forza l'emissione del contenuto del buffer
- `flose()` fa il flush e chiude lo stream

Oggetto session

- Oggetto che fornisce informazioni sul contesto di esecuzione della JSP
- Rappresenta la sessione corrente per un utente
- L'attributo `session` della direttiva `page` deve essere valorizzata a true affinché la 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 l'ID di una sessione
- `boolean isNew()` dice se la sessione è nuova
- `void invalidate()` permette di invalidare (distruggere) una sessione
- `long getCreationTime()` ci dice da quanto è attiva la sessione (in millisecondi)
- `long getLastAccessedTime()` ci dice quando è stata utilizzata l'ultima volta

Oggetto application

- Oggetto che fornisce informazioni sul contesto di esecuzione della JSP (è il `ServletContext`)
- Rappresenta la web application a cui la JSP appartiene
- Consente di interagire con l'ambiente di esecuzione:
  - fornisce la versione del 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 JSP
- Rappresenta l'insieme degli oggetti impliciti di una JSP
  - Consente l'accesso a tutti gli oggetti impliciti e ai loro attributi
  - Consente il trasferimento del controllo ad altre pagine
- Poco usato per lo scripting, utile per costruire custom tags

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);
  %>
```

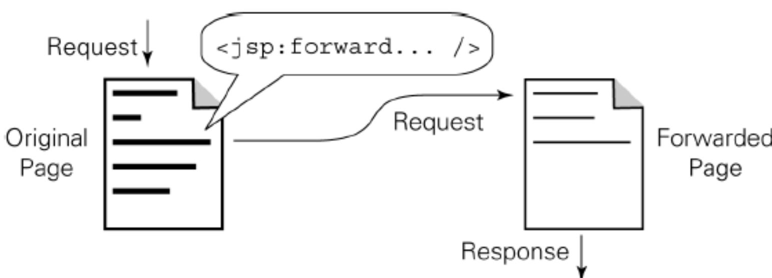
Azioni

- Le azioni sono comandi JSP valutati a request time.
- Sono previsti 6 tipi di azioni definite dai seguenti tag:
  - `useBean`: istanzia un JavaBean e gli associa un identificativo
  - `getProperty`: ritorna la property indicata come un oggetto
  - `setProperty`: imposta il valore della property indicata per nome
  - `include`: include un file nella JSP
  - `forward`: cede il controllo ad un'altra JSP o servlet
  - `plugin`: genera contenuto per scaricare plug-in Java se necessario
- Sono espresse usando sintassi XML

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


Azioni: forward

- Sintassi: `<jsp:forward page="localURL" />`
- Consente il 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

- E' possibile generare dinamicamente l'attributo page
  `<jsp:forward page='<%="message"+statusCode+".html"%>'/>`
- Gli oggetti `request`, `response` e `session` della pagina d'arrivo sono gli stessi della pagina chiamante, ma viene istanziato un nuovo oggetto `pageContext`
- **Attenzione:** Il forward è possibile soltanto se non è stato emesso alcun output
- E' 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>
```

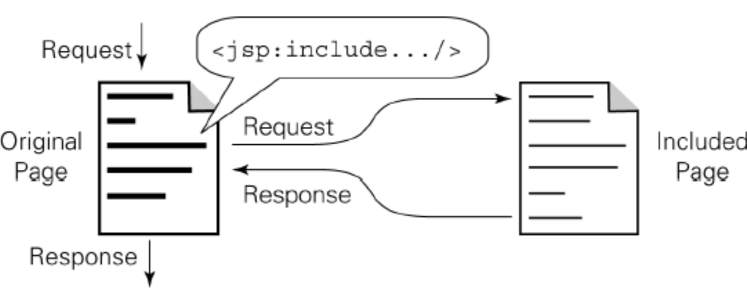

Azioni: include

- 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 il 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 un flush prima di effettuare l'inclusione
  - Gli oggetti session e request per la pagina da includere sono gli stessi della pagina chiamante, ma viene istanziato un nuovo contesto di pagina

Azioni: include

- E’ 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 e 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 un architettura a più livelli
- A tal fine le JSP consentono uno 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

Componenti software

- Un componente è un "**circuito integrato**" software che comunica con l'esterno attraverso una serie di "piedini"
- Un'entità in grado di incorporare componenti viene definita **container** ed è l'equivalente software di una scheda elettronica
- Abbiamo tre tipi di "piedini": proprietà, metodi, eventi (**modello PME**):
  - **Proprietà** (property): "piedini di stato", pseudo-variabili che consentono di interagire in modo protetto con lo stato interno
  - **Metodi**: "piedini di ingresso", comandi che provocano l'esecuzione di azioni
  - **Eventi**: "piedini di uscita", provocano l'esecuzione di metodi nel container (callback) in seguito a qualcosa che si verifica nel componente

JavaBeans

- I JavaBeans (chicchi di caffè) sono il modello di componenti di Java
- Un **JavaBean**, o semplicemente un **bean**, non è altro che una classe Java dotata di alcune caratteristiche particolari:
  - E' una 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 - 1

- Le proprietà sono elementi dello stato del componente che vengono esposti in modo protetto
- In alcuni linguaggi (ad esempio C#) esiste una sintassi specifiche 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 `get`*`Prop`*`()` e `set`*`Prop`*`()`
- 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 setLenght(int Value)` e `int getLenght()` identificano la proprietà `lenght` di tipo `int`

Proprietà - 2

- Se definiamo solo il metodo get avremo una proprietà in sola lettura (read-only)
- Le proprietà di tipo boolean seguono una regola leggermente diversa: il metodo di lettura ha la forma `isProp()` anziché `getProp()`
- Per esempio la proprietà `empty` sarà rappresentata dalla coppia `void setEmpty(boolean value)` e `boolean getEmpty()`
- Esiste anche la possibilità di definire proprietà indicizzate per rappresentare collezioni di valori (pesudoarray)
- In questo caso sia get che set prevedono un parametro che ha la funzione di indice
- Per 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 containers**) che gestiscono
  - il tempo di vita dei singoli componenti
  - i 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 (reflection)
- Un contenitore per JavaBean prende il nome di **bean container**
- Un **bean container** è in grado di interfacciarsi con i **bean** utilizzando la **Java Reflection** che fornisce strumenti di **introspezione** e di **dispatching**
- L'obbligo del costruttore di default ha lo scopo di consentire la creazione dinamica delle istanze

Introspezione e dispatching

- **Introspezione:** capacità di descrivere le proprie caratteristiche (proprietà e metodi con relativi parametri).
- Consente l'esplorazione da parte del container
- **Dispatching** (o very-late binding): capacità di invocare i metodi in modo completamente dinamico.
- Estende una caratteristica tipica dell'OOP
  - **Early-binding**: sia l'interfaccia che l'implementazione devono essere note a tempo di compilazione
  - **Late-binding**: a tempo di compilazione viene fissata solo l'interfaccia mentre l'implementazione dei metodi è definita a runtime
  - **Very-late binding**: sia interfaccia che implementazione sono definite solo a runtime

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

- Le JSP prevedono una serie di tag per agganciare un bean e utilizzare le sue proprietà all'interno della pagina:
- Sono di 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:setProprety 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 e class e scope
  - **id** è il nome con cui l'istanza del bean verrà indicata nel resto della pagina
  - **class** è la classe Java che definisce il bean
  - **scope** definisce l'ambito di accessibilità e il tempo di vita dell'oggetto (il default è 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 il forward	Fino alla fine dell'elaborazione della richiesta e ala restituzione della rispostal
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 bean tag non supportano le proprietà indicizzate
- Però un bean è un normale oggetto Java: è quindi possibile accedere a variabili e metodi.
- Esempio:

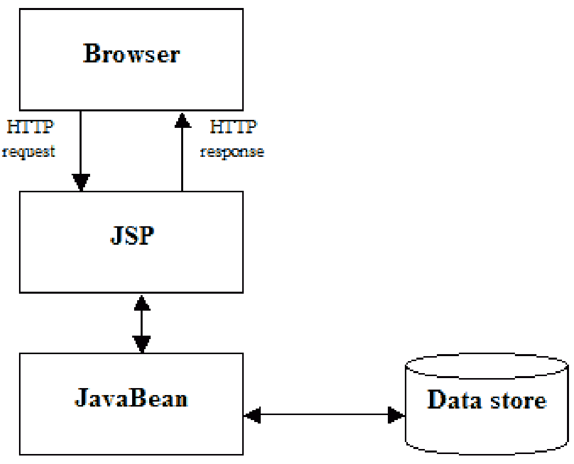
```
<jsp:useBean id="weather" class="weatherForecasts"/>

<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**



Custom tag e tag libraries

- Le 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 una la direttiva taglib con la sintassi:

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

  - L'attributo `uri` fa riferimento ad un file xml, con estensione tld (tag library descriptor), che contiene informazioni sulle classe che implementa i tag.
  - L'attributo `prefix` indica il prefisso da utilizzare nei tag che fanno riferimento alla tag library (la tag library è un namespace)

Definizione di taglib

- Per definire una tag library occorrono due elementi:
  - Un file **TLD** (Tag Lib Definition) che specifica i singoli Tag ed a quale "classe" corrispondono
  - Le classi che effettivamente gestiscono i tag
- Il file TLD è un file XML che specifica:
  - i tag che fanno parte della libreria
  - i loro eventuali attributi
  - il "body" del tag (se esiste)
  - la 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>emtpy</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 (set*AttrName*() e get*AttrName*() per ogni attributo
- `doStartTag()` utilizza l'oggetto out restituito da PageContext per scrivere nell'output della pagina e se il 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"+ <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>
```