

# Programmazione WEB server-side JSP

TECNICO SUPERIORE PER LO  
SVILUPPO SOFTWARE  
2013

# **Programmazione WEB server-side JSP**

50 ore

L'unita' formativa si pone l'obiettivo di fornire gli strumenti e far acquisire le competenze necessarie per sviluppare autonomamente siti web dinamici utilizzando il linguaggio Java ed il server Tomcat.

# Competenze

## Sviluppare software

- Tradurre i moduli nel linguaggio di programmazione
- Formalizzare l'interfaccia dell'applicazione
- Individuare moduli di librerie
- Diagnosticare le anomalie
- Gestire le verifiche e le modifiche funzionali
- Strutturare le informazioni per l'utente
- Gestire il processo di sviluppo secondo standard di qualità

# Competenze

## Individuare gli strumenti di sviluppo software

- Valutare gli strumenti di sviluppo software presenti sul mercato
- Scegliere il sistema operativo
- Scegliere la piattaforma hardware
- Scegliere l'architettura di rete

# Attività

## Realizzare le applicazioni

- Creare le strutture dati
- Scrivere il codice ed includere le librerie
- Disegnare l'interfaccia video ed i report
- Utilizzare l'ambiente di sviluppo
- Modificare le applicazioni
- Realizzare la guida utente
- Argomenti: Ambienti di sviluppo
- Saperi: Java EE, Tomcat

# Attività

## Realizzare la manutenzione dell'applicazione

- Effettuare collaudi
- Determinare le cause dei malfunzionamenti
- Installare aggiornamenti

## Argomenti: Tecniche di debug

### Saperi:

- Trovare errori pagine web, siti, programmazione
- Trovare errori di codice in grado di esporre le applicazioni ad attacchi di tipo cross-site scripting e di provocare problemi di caching.
- Software per il debug

# Modalità

- Analisi di casi
- Autoistruzione
- Discussione e confronto
- Esercitazione di laboratorio
- Esercitazione pratica
- Lezione frontale
- Problem solving

# JSP Introduzione

- Introduzione alla Piattaforma Java Enterprise Edition
- Il modello di applicazioni distribuite multilivello di Java EE
- Client, componenti, moduli, container e servizi
- Panoramica su servizi e API della piattaforma
- L'application Server JBoss
- Il web container Tomcat
- L'ide Eclipse; Netbeans



# JSP Design Pattern

- Design Pattern per Java EE
- Definizione di Design Pattern
- Tipologie di pattern: creazionali, strutturali, comportamentali, architetturali
- Il pattern architetturale MVC (Model View Controller)

# JSP Caratteristiche del linguaggio

- Creazione di pagine dinamiche con JSP
- Ruolo di JSP nel contesto Java EE
- Caratteristiche principali delle pagine JSP
- Direttive JSP
- Azioni JSP
- Oggetti impliciti
- Collaborazione tra JSP e servlet
- JavaBeans
- JSP Standard Tag Libraries (JSTL) (cenni)
- JSP Custom Tags (cenni)
- Scrivere pagine JSP usando Expression language (EL)

# JSP e servlet

- Estensione del web server con le servlet
- Ruolo delle servlet nel contesto Java EE
- Caratteristiche principali delle servlet
- Oggetti Request e Response
- Cookie e gestione delle sessioni

# JSP – EJB Enterprise JavaBeans

- Sviluppo delle componenti Enterprise JavaBeans lato server con gli EJB - Enterprise JavaBeans
- Panoramica sui componenti EJB e sul ruolo di EJB container
- Tipologie di EJB: session, entity, message-driven.
- Accesso agli EJB tramite le interfacce
- Sviluppo di client per il bean
- Packaging e deployment degli EJB

# JSP Framework

- Framework di sviluppo
- Definizione di framework
- Spring
- Hibernate
- STRUTS

# Una pagina JSP

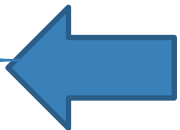
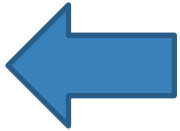
Una pagina **JSP** è un modello di una pagina Web che utilizza codice **Java** per generare dinamicamente un documento **HTML**.

Le pagine **JSP** vengono eseguite in un componente operante sul server chiamato container JSP che le traduce nei corrispondenti servlet **Java**.

Per questo motivo i servlet e le pagine JSP sono intimamente collegati. Ciò che è possibile fare negli uni è, in genere, possibile eseguire anche nelle altre.

# Prima pagina JSP

```
<%@ page language="java" import="java.lang.*,java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Prima pagina JSP</title>
<meta http-equiv="description" content="Questa è la mia prima pagina JSP"/>
</head>
<body>
<h1>Prima pagina JSP</h1>
<br/>
<%
Calendar data = Calendar.getInstance();
out.print("Oggi è il giorno "
data.get(data.DATE)+ "/" +(data.get(data.MONTH)+1)+ "/" +data.get(data.YEAR));
%>
</body>
</html>
```



# **La tecnologia JSP ha numerosi vantaggi rispetto ad altre tecnologia oggi comunemente usate.**

## ***Rispetto alle Active Server Pages (ASP)***

- ASP è una tecnologia concorrente creata da Microsoft. I vantaggi delle JSP si notano principalmente su due fronti. In primo luogo, la parte dinamica è scritta in Java, non in VBScript o qualche altro linguaggio specifico per ASP.
- Il codice è così più potente e meglio si adatta alla produzione di applicazioni complesse che richiedono componenti riutilizzabili. Inoltre, JSP è portabile anche su altri sistemi operativi e Web servers; non si è così limitati all'uso di Windows NT/2000 e IIS.

**ASP:** le *Active Server Pages* sono basate su script (VBScript o JavaScript) che vengono eseguiti sul Server e generano dinamicamente il contenuto HTML. Richiedono sistemi Windows e IIS (Internet Information Services) come Web Server per gestire le richieste.

**ASP.NET:** evoluzione della tecnologia ASP. Si basa su codice VB.NET o C# che viene compilato sul Web Server IIS utilizzando le classi del Framework .NET. Grazie ad essa è possibile implementare i Web Services basati XML ed il protocollo SOAP.



# Rispetto a PHP

PHP è un linguaggio di scripting che ha la caratteristica di essere free, open source, HTML-embedded e che è in qualche modo simile sia a ASP che a JSP. Il vantaggio in questo caso consiste nel fatto che la parte dinamica è scritta in Java, che ha un'API esauriente per il networking, accesso ai database e consente di lavorare con oggetti distribuiti.

- PHP: Hypertext Preprocessor è una tecnologia basata su codice scritto in PHP (un misto di C e Perl) e richiede Apache come web server.
- Può girare sia su sistemi Windows che Linux
- (anche se l'ambiente ideale è Linux).

# Rispetto ai puri servlets

Le JSP non forniscono alcuna possibilità di lavoro che non possa essere sfruttata utilizzando semplicemente un servlet. Infatti i documenti JSP sono automaticamente tradotti in servlet per essere eseguiti. Ma è molto più conveniente scrivere e modificare una pagina in HTML normale, piuttosto che avere milioni di statement *println* che generano l'HTML. Inoltre, in ambito business, separando la presentazione dal contenuto si può ottimizzare il lavoro di più persone: gli esperti di design possono costruire il codice HTML con i comuni tool di lavoro e passare il codice così ottenuto ai programmatori servlet per l'inserimento del contenuto dinamico.

# rispetto agli script CGI

Le pagine JSP hanno migliori prestazioni e scalabilità rispetto agli script CGI poiché sono persistenti in memoria e sono multithread ; ai vantaggi dei servlet ne aggiungono altri più specifici :

- Vengono ricomilate automaticamente quando necessario
- Poiché esistono nel normale spazio dei documenti del server Web, l'indirizzamento delle pagine JSP è più semplice rispetto a quello dei servlet .
- Dato che le pagine JSP sono di tipo HTML sono compatibili con gli strumenti di sviluppo Web .

# gli script CGI

**CGI:** le *Common Gateway Interfaces* sono state le prime a comparire e di conseguenza presentano notevoli limitazioni prestazionali e di concorrenza (intesa come gestione degli accessi simultanei). Sono basati su applicativi compilati (scritti in C o in Perl) che girano sul server e rispondono alle richieste dei client.

# Rispetto ai Server-Side Includes (SSI)

SSI è una tecnologia ampiamente supportata per l'inserimento in pagine Web statiche di parti definite esternamente. JSP ha maggiori vantaggi di uso poiché ha un set di tool per la costruzione delle parti esterne molto più ricco ed ampio. Si hanno inoltre maggiori possibilità per quanto riguarda la parte di risposta HTTP in cui viene ora inserita la parte esterna. SSI viene utilizzata normalmente per semplici inclusioni, non per l'utilizzo di parti di codice complesso, come possono essere l'utilizzo di data form, la creazione di connessioni a database, ed altre applicazioni di livello avanzato.

# Rispetto a JavaScript

JavaScript, che è completamente distinto dal linguaggio di programmazione Java, è normalmente usato per generare dinamicamente HTML sul client, costruendo parti della pagina Web mentre il browser carica il documento. Questa è una possibilità molto utile ma risolve solo situazioni in cui le informazioni dinamiche sono basate sull'ambiente del client. Ad eccezione dei cookies, la richiesta dati HTTP non è utilizzabile dalle routine JavaScript sul client-side. Quindi, mancando i JavaScript di routine per il network programming, il codice JavaScript sul client non può avere accesso alle risorse contenute sul server-side, come database, cataloghi, informazione sul prezzo e molto altro. JavaScript può anche essere usato sui server, come viene notevolmente fatto sui server Netscape o come linguaggio di script per IIS. Java è però di gran lunga più potente, flessibile, affidabile e portabile.

# Rispetto all' HTML statico

Il normale HTML non può contenere informazione dinamica, così le pagine di HTML statico non possono basarsi su informazione inserita dall'utente o su risorse dati del server-side. JSP è di semplice utilizzo, tale per cui sia abbastanza naturale e ragionevole incrementare il codice HTML con l'informazione dinamica che le JSP possono generare; si beneficia quindi dell'utilizzo di informazione dinamica in modo conveniente. Precedentemente l'inserimento di informazione dinamica era talmente complesso da precluderne l'utilizzo anche in molte applicazioni di grande valore.

# Java Web Application, Servlet o pagine JSP?

Spesso si tende ad associare lo stesso significato ai tre termini.

Una **Java Web Application** è un insieme di Servlet, pagine JSP, componenti Bean, pagine HTML e fogli di stile CSS.

Una **Servlet** è una classe Java che utilizza il protocollo http per ricevere ed inviare delle richieste tra client e Server.

Una **pagina JSP** è una pagina che racchiude del codice statico (HTML) e del codice dinamico (Java) che verrà eseguito sul Server (ha la stessa struttura di una pagina PHP o ASP).





## **Java Web Application, Servlet o pagine JSP?**

Le Java Web Application inizialmente si basavano solo sulle Servlet, offrivano comunque notevoli potenzialità ma erano una tecnologia alla portata dei programmatori più esperti (in una classe Java bisognava inserire sia il codice gestionale che il codice che curava il layout della pagina Web).

Successivamente furono introdotte le JSP per semplificare il lavoro dei web designer che potevano così dedicarsi al layout della pagina senza entrare in merito della logica di gestione dell'applicazione. Una pagina JSP infatti, dovrebbe contenere poco codice Java e definire solo il template HTML dell'applicazione.

# JSP o Servlet

## JSP

- Hanno l'aspetto e la struttura di pagine XHTML  
Contengono markup HTML o XHTML
- Vengono utilizzate quando la maggior parte del contenuto che deve essere visualizzato segue una struttura fissata.  
In generale una piccola parte del contenuto deve essere generata dinamicamente

## Servlet

- Utilizzate invece quando solo una piccola porzione del contenuto deve seguire una struttura fissata
- La maggior parte del contenuto deve essere generata dinamicamente

# Una pagina JSP

- 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 + parte generata dinamicamente) viene restituita al client
- Sono assimilabili ad un linguaggio di scripting: in realtà vengono trasformate in servlet dal container



# Prima pagina JSP

```
<%@ page language="java" import="java.lang.*,java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Prima pagina JSP</title>
<meta http-equiv="description" content="Questa è la mia prima pagina JSP"/>
</head>
<body>
<h1>Prima pagina JSP</h1>
<br/>
<%
Calendar data = Calendar.getInstance();
out.print("Oggi è il giorno "
data.get(data.DATE)+ "/" +(data.get(data.MONTH)+1)+ "/" +data.get(data.YEAR));
%>
</body>
</html>
```

# script in JSP

Gli script in JSP sono vere e proprie porzioni di codice inserite all'interno di pagine HTML che possono rappresentare dichiarazioni o espressioni. Il codice del file Java deve essere inserito all'interno dei tag **<%**  
**%>**, mentre per le dichiarazioni la sintassi cambia leggermente.

# Esistono tre tipi di pagine JSP 1

**CODICE SORGENTE JSP**: questa è la forma che viene effettivamente scritta dallo sviluppatore. Si tratta di un **file di testo con l'estensione .jsp** contenente codice HTML, istruzioni Java e direttive ed azioni JSP che descrivono il modo in cui generare la pagina Web di risposta per una determinata richiesta.

# Esistono tre tipi di pagine JSP 2

**CODICE SORGENTE JAVA:** il container JSP traduce il codice sorgente Java in codice sorgente servlet Java. Questo codice viene normalmente salvato nell'area di lavoro e spesso è utile per eseguire operazioni di debugging.

# Esistono tre tipi di pagine JSP 3

**CLASSE JAVA COMPILATA:** come per ogni altra classe Java , il codice generato per il servlet viene compilato per creare un file **.class**, pronto per essere caricato ed eseguito.



# Il container JSP

Il container JSP gestisce automaticamente ognuna di queste forme sulla **base** della data in cui è avvenuta l'ultima modifica a ciascun file.

In risposta ad una richiesta HTTP , il container controlla se il codice sorgente .jsp è stato modificato dall'ultima compilazione del codice sorgente .java. In caso affermativo il container ritraduce il codice sorgente JSP per produrre la nuova versione del codice sorgente Java.

# Il container JSP

In realtà il container (nel nostro caso Tomcat 7) ha eseguito svariate operazioni in seguito alla richiesta del browser:

- ha convertito la pagina JSP in una Servlet (classe Java)
- ha compilato la classe in bytecode
- ha istanziato la classe nel servlet container
- ha esaudito la richiesta restituendo la risposta al browser.

Tutto questo accade **solo alla prima richiesta della pagina**, perché alle successive richieste Tomcat avrà già la pagina compilata in “memoria”, ottenendo quindi un notevole miglioramento delle prestazioni.

Le pagine JSP infatti sono molto performanti rispetto alle altre tecnologie (ASP e PHP) che ad ogni richiesta interpretano il codice lato server senza compilarlo.

# Il container JSP

Risulta evidente di come sia molto più semplice scrivere una JSP che una Servlet. Esaminando meglio il codice generato, notiamo come le istruzioni Java che noi abbiamo inserito e le istruzioni di scrittura dei tag HTML sono stati copiati nel metodo `_jspService()`. Nel corso delle successive richieste il *container* avrà già in memoria la servlet creata dalla pagina JSP e richiamerà sempre il metodo `_jspService()` sull'istanza conservata.

In conclusione, alla base di tutto ci sono comunque le classi Java, solo che anche un web designer senza troppa esperienza in programmazione Java, potrà realizzare le proprie applicazioni con uno sforzo minimo perché il *container* farà tutto il lavoro di conversione.

A questo punto si potrebbe pensare di realizzare la nostra applicazione solo con pagine JSP, potrebbe essere una soluzione valida, ma è nella realizzazione delle Servlet e dei componenti Bean che risiedono le maggiori differenze tra le Java Web Application e le altre tecnologie *tradizionali* come ASP o PHP.

# Preparazione dell'ambiente

In questa lezione vedremo come preparare la nostra macchina all'esecuzione delle pagine JSP.

Le soluzioni che verranno proposte non sono le uniche possibili. La piattaforma J2EE è a specifiche aperte, di conseguenza sono disponibili sul mercato innumerevoli ambienti di sviluppo sia commerciali che gratuiti.

Nella scelta degli strumenti mi sono orientato su prodotti gratuiti e open source però allo stesso tempo con potenzialità notevoli, paragonabili ad ambienti commerciali:

**J2SE:** sono indispensabili per la compilazione e l'esecuzione delle classi Java

- **Apache Tomcat:** un Application Server molto valido che integra un servlet container ed un Web Server, volendo lo si può integrare anche in Apache e IIS di Microsoft.

- **MySQL:** un ottimo server di database molto prestante, gratuito per l'utilizzo sul Web (le procedure per la sua installazione saranno descritte nelle lezioni successive).

## Installazione di J2SE

Dopo aver scaricato il pacchetto di Java 2 Standard Edition dal sito della Sun Microsystems (prelevate anche i file di help utili per i riferimenti al linguaggio Java), fare doppio click per eseguire l'installazione. Selezionare il percorso in cui installarlo (personalizzatelo come da figura C:\Java\J2SE ):

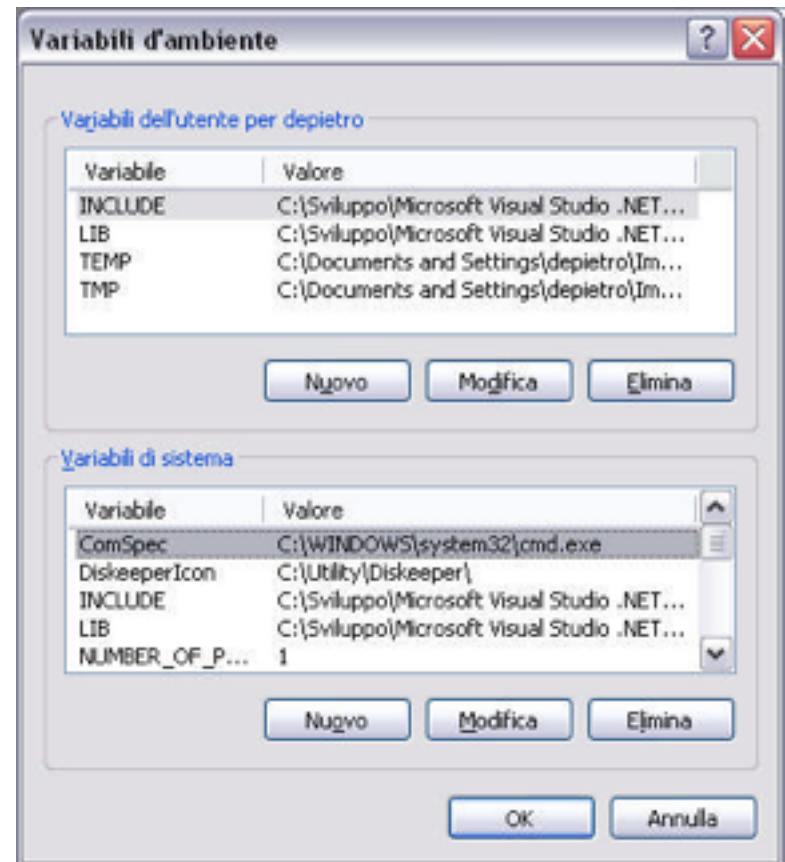


Andate avanti con l'installazione selezionando tutti i componenti da installare.

A questo punto bisognerà impostare le variabili d'ambiente.

Andare su **Pannello di Controllo | Sistema** selezionare la tabella **Avanzate** premere il pulsante

**Variabili d'ambiente:**



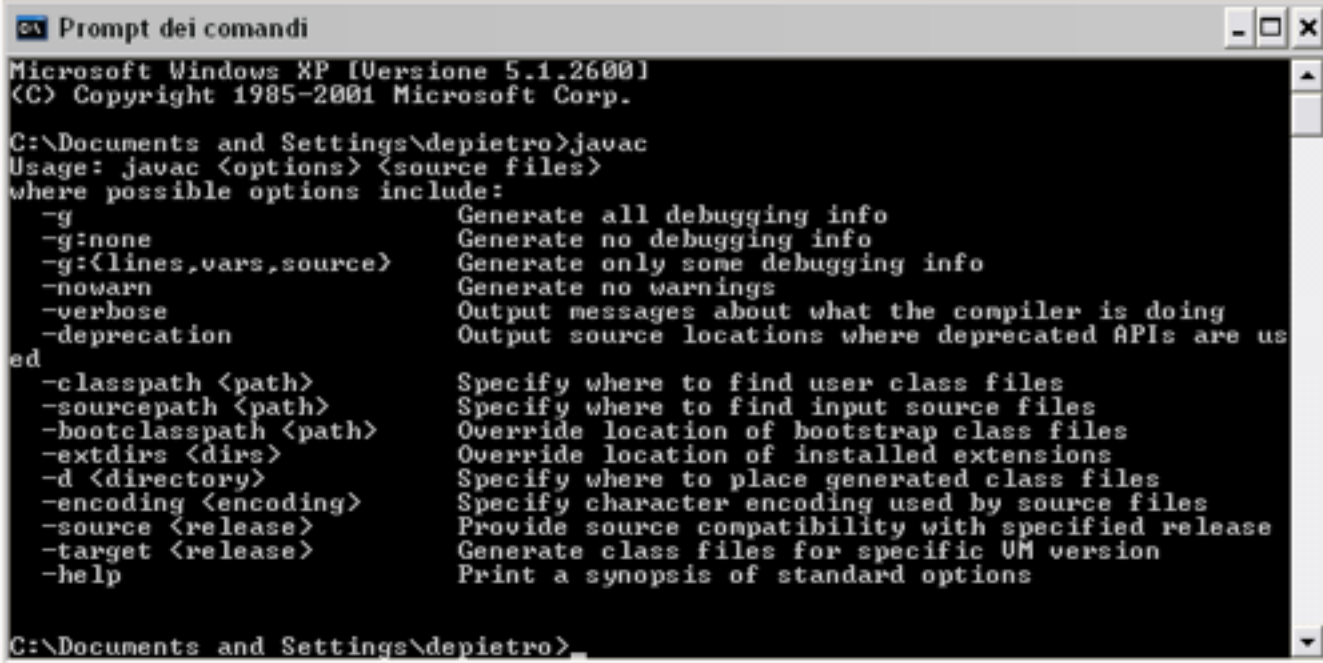
Premere il pulsante **Nuovo** nel frame delle **Variabili di sistema**:

È importante inserire nella casella di testo **Valore variabile** il percorso completo della cartella in cui si è installato lo JDK. Poi bisogna selezionare **path** delle variabili di sistema e premere il pulsante **Modifica**:



Aggiungere alla fine del **Valore variabile** la stringa evidenziata in figura (compreso il punto e virgola). Premere OK ed il gioco è fatto. Per testare che il tutto sia configurato a dovere, aprire una finestra DOS dagli accessori e digitare il comando **javac**, dovrebbe presentarsi qualcosa di simile:





```
Prompt dei comandi
Microsoft Windows XP [Versione 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\depietro>javac
Usage: javac <options> <source files>
where possible options include:
  -g               Generate all debugging info
  -g:none          Generate no debugging info
  -g:<lines,vars,source> Generate only some debugging info
  -nowarn          Generate no warnings
  -verbose         Output messages about what the compiler is doing
  -deprecation     Output source locations where deprecated APIs are used
  -classpath <path> Specify where to find user class files
  -sourcepath <path> Specify where to find input source files
  -bootclasspath <path> Override location of bootstrap class files
  -extdirs <dirs>   Override location of installed extensions
  -d <directory>    Specify where to place generated class files
  -encoding <encoding> Specify character encoding used by source files
  -source <release> Provide source compatibility with specified release
  -target <release> Generate class files for specific VM version
  -help            Print a synopsis of standard options

C:\Documents and Settings\depietro>
```

Se viene visualizzato un messaggio di comando non riconosciuto, verificate di aver eseguito correttamente le operazioni e soprattutto i percorsi delle cartelle.

# Installazione di Tomcat

Tomcat è un progetto dell'Apache Software Foundation ed è quello più utilizzato tra gli sviluppatori di tutto il mondo. Esistono varie versioni di Tomcat:

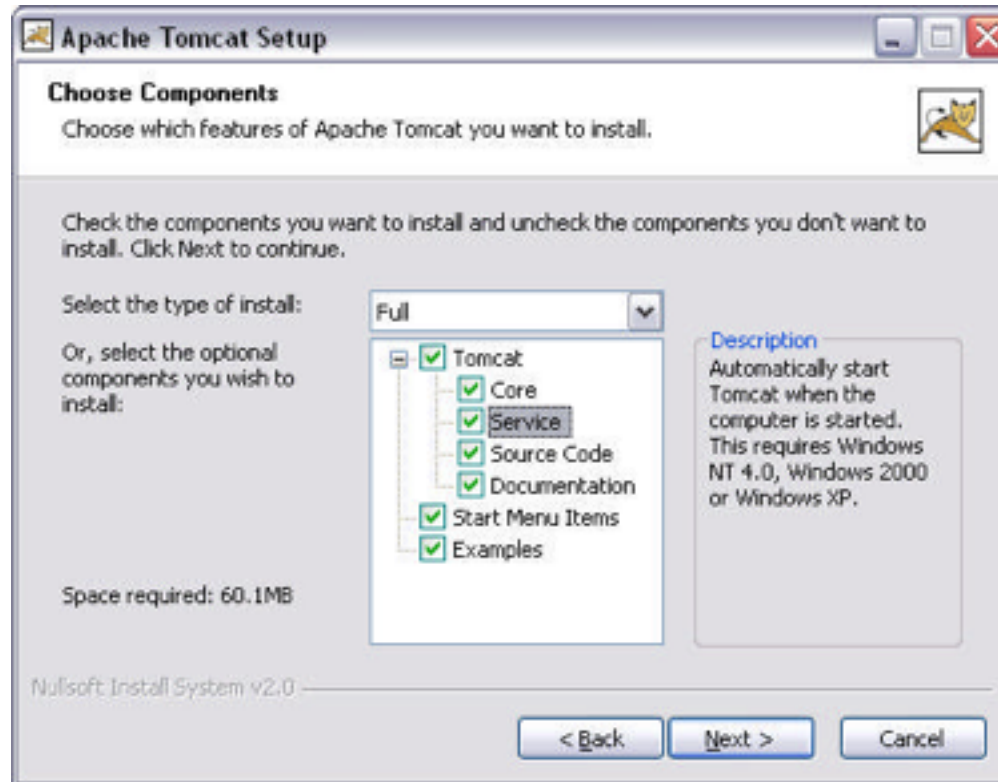
- Tomcat v7.x fa riferimento alle specifiche delle Servlet v3.0 e delle JSP v2.2
- Tomcat v6.x fa riferimento alle specifiche delle Servlet v2.5 e delle JSP v2.1
- Tomcat v5.x fa riferimento alle specifiche delle Servlet v2.4 e delle JSP v2.0
- Tomcat v4.x fa riferimento alle specifiche delle Servlet v2.3 e delle JSP v1.2
- Tomcat v3.x fa riferimento alle specifiche delle Servlet v2.2 e delle JSP v1.1

La seguente installazione si riferisce alla versione di Apache Tomcat 7.

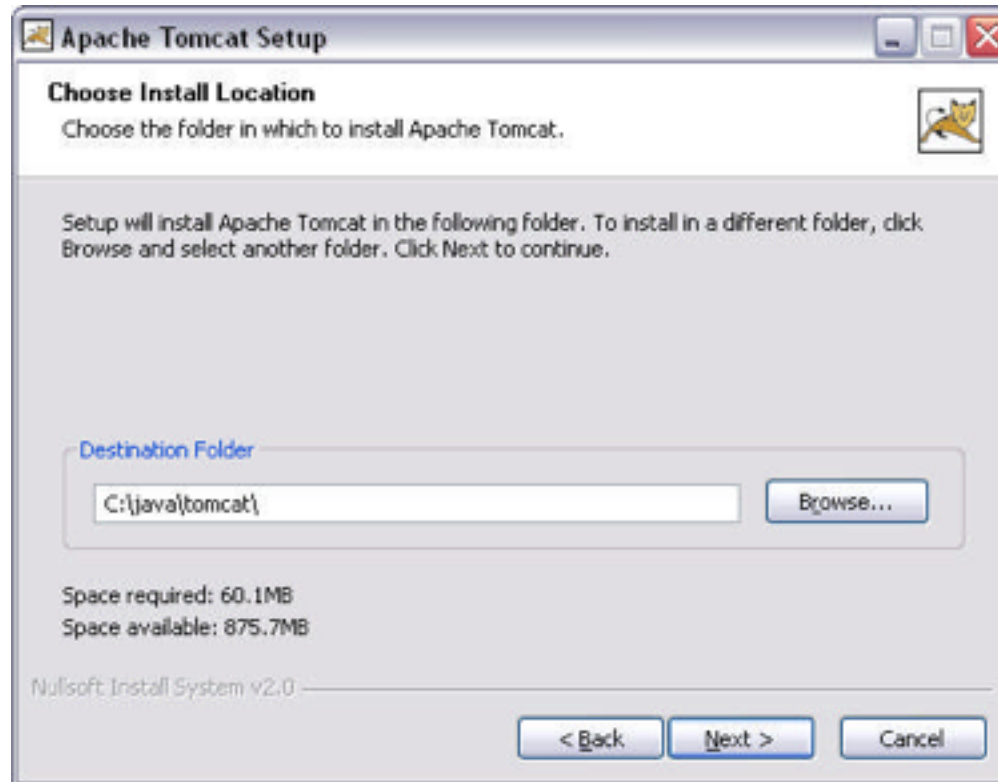
Scaricate il file .exe all'indirizzo seguente <http://tomcat.apache.org/download-70.cgi>

Assicuratevi di installare l'applicativo in versione Full di modo che venga installato come servizio di Windows e possa quindi partire automaticamente all'avvio del sistema:

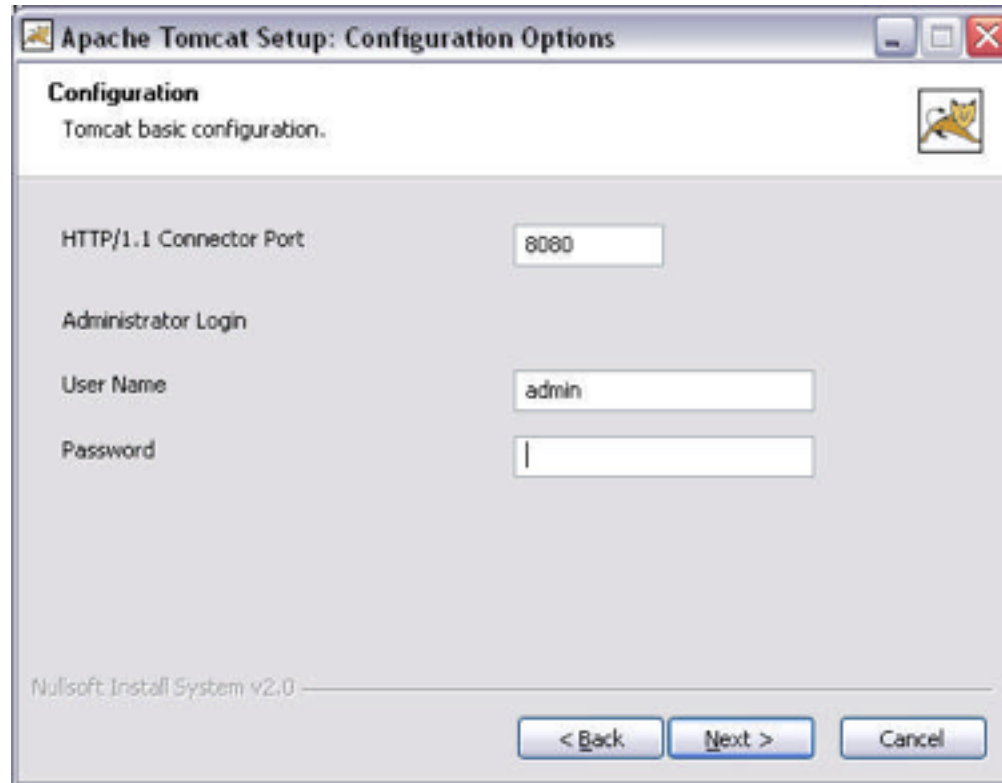
# Installazione di Tomcat



# Installazione di Tomcat



# Installazione di Tomcat



The screenshot shows the 'Apache Tomcat Setup: Configuration Options' window. The title bar includes the text 'Apache Tomcat Setup: Configuration Options' and standard window control buttons. The main content area is titled 'Configuration' with the subtitle 'Tomcat basic configuration.' and a small Tomcat logo. It contains four configuration fields: 'HTTP/1.1 Connector Port' with the value '8080', 'Administrator Login' (empty), 'User Name' with the value 'admin', and 'Password' (empty). At the bottom, there is a footer 'Nullsoft Install System v2.0' and three buttons: '< Back', 'Next >', and 'Cancel'.

Apache Tomcat Setup: Configuration Options

**Configuration**  
Tomcat basic configuration.

HTTP/1.1 Connector Port: 8080

Administrator Login:

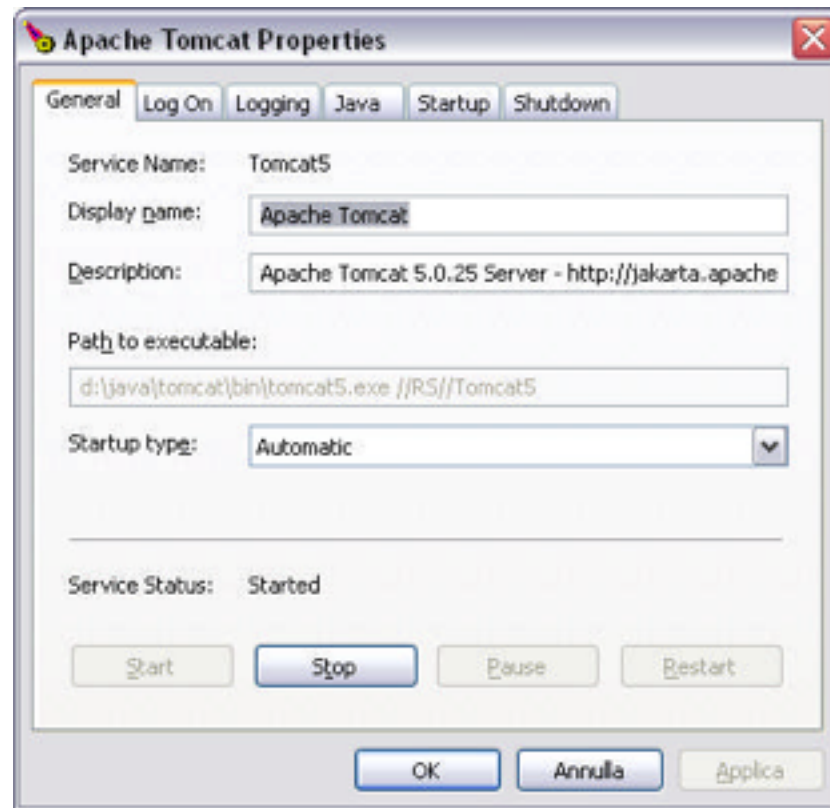
User Name: admin

Password:

Nullsoft Install System v2.0

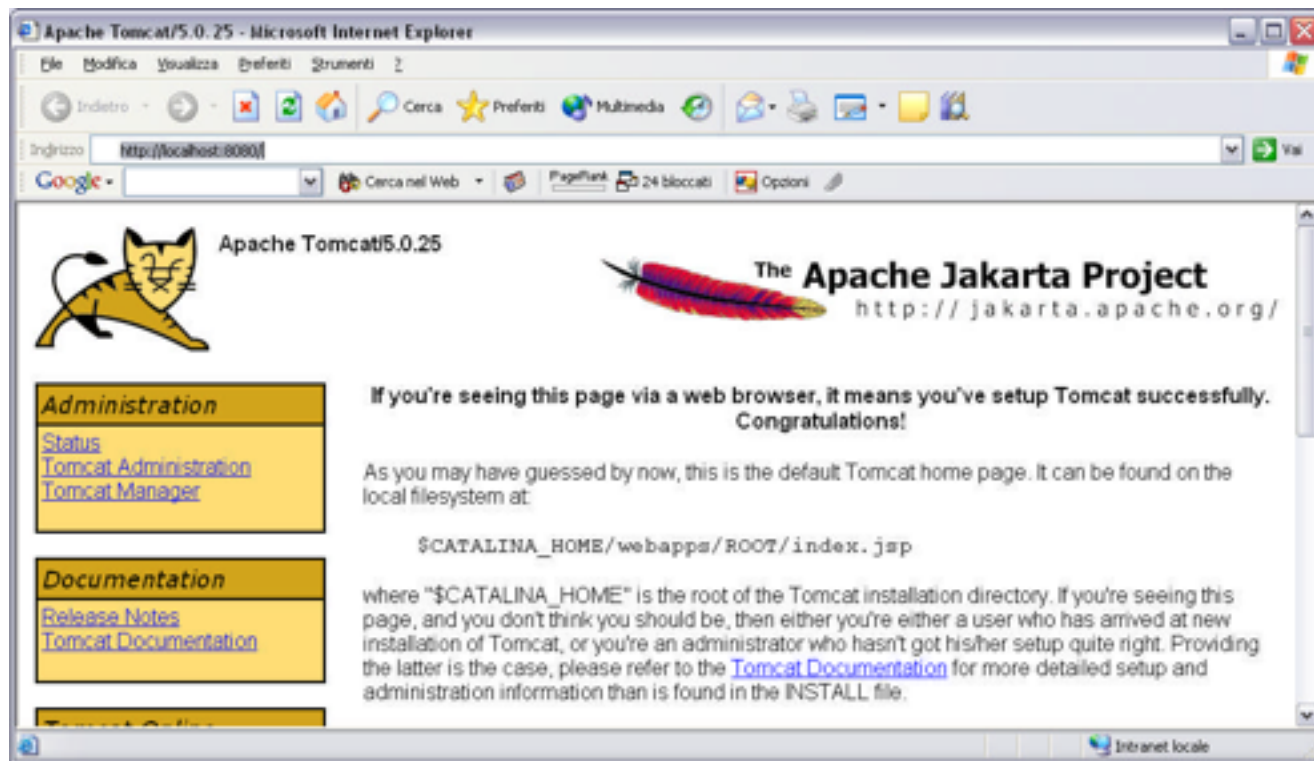
< Back   Next >   Cancel

# Installazione di Tomcat



# Installazione di Tomcat

Per verificare che il tutto sia andato a buon fine, aprite il browser e digitate l'url <http://localhost:8080/> , dovrebbe aprirsi la seguente pagina:



Da notare che Tomcat lavora sulla porta **8080** e non crea alcun conflitto con gli altri Web Server che di solito lavorano sulla porta 80.

# Installazione di Tomcat

## Creiamo una Web Application

Se guardiamo nella directory di installazione di Tomcat, noteremo la seguente struttura:

- *bin* contiene gli script per eseguire e fermare Tomcat
- *common* contiene le librerie accessibili a Tomcat e alle *web application*
- *conf* contiene i file di configurazione di Tomcat
- *logs* contiene i file di log
- *server* contiene le librerie accessibili solamente a Tomcat (Catalina)
- *shared* contiene le classi e le librerie comuni a tutte le *web application*
- *temp* contiene file temporanei di Tomcat
- *webapps* contiene le *web application*
- *work* contiene i file temporanei per ciascuna *web application*. È qui che vengono create le servlet generate dalle JSP.



# *CorsoJSP*

All'interno della directory *webapps* saranno presenti già alcune Web Application predefinite ed in più una cartella di nome ROOT. Questa è la cartella che contiene la radice del sito web gestito da Tomcat, la pagina *index.jsp* che troviamo al suo interno è la risorsa che abbiamo aperto durante il test del web server.

Ora creiamo la nostra prima Web Application. Secondo le specifiche J2EE, una WA deve avere una determinata struttura a directory. Creiamo quindi una nuova cartella di nome *CorsoJSP* (sarà il nome della nostra WA) all'interno di *webapps* e diamole la seguente struttura:

```
\CorsoJSP\  
\CorsoJSP\WEB-INF\  
\CorsoJSP\WEB-INF\web.xml  
\CorsoJSP\WEB-INF\classes\  
\CorsoJSP\WEB-INF\lib\
```

# CorsoJSP

*CorsoJSP* diventerà la radice della nostra applicazione, tutto ciò che sarà inserito al suo interno sarà reso pubblico e quindi disponibile a tutti gli utenti. La risorsa che Tomcat aprirà di default sarà un file di nome *index.htm* o *index.jsp*.

E' opportuno organizzare l'applicazione in più sottocartelle (cercando di associare ogni sottolivello ad una sottostruttura logica della nostra applicazione), ogni sottocartella dovrà contenere un file *index.jsp* o *.htm* così da permettere una navigazione lineare per livelli.

# CorsoJSP

Differente invece sarà la funzione della cartella WEB-INF che non sarà esposta dall'Application Server e quindi diventa il luogo ideale per riporre le classi (all'interno di *classes*), gli archivi *jar* (all'interno di *lib*) o le risorse che dovranno essere protette dagli utenti esterni.

Importante è la funzione del *Deployment Descriptor* ovvero il file `web.xml` (un file xml editabile con qualsiasi editor testuale), che conterrà la descrizione delle informazioni e delle risorse aggiuntive necessarie al funzionamento della WA.

# Una semplice pagina html

Ora realizziamo una semplice pagina html che chiederà l'inserimento di due valori numerici ed il server dovrà risponderci con una pagina che visualizzi la somma dei due numeri inviati.

I file utilizzati saranno *index.htm* e *somma.jsp* che saranno collocati nella cartella *CorsoJSP* all'interno di *webapps*.

Di seguito si riporta il codice della pagina index.htm:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Corso JSP - Lezione 2</title>
<meta http-equiv="description" content="Invio di due valori numerici"/>
</head>
<body>
<form method="post" action="somma.jsp">
<fieldset>
<legend>Inserisci due valori numerici</legend>
<label for="valore1">Valore 1:</label>
<input type="text" id="valore1" name="valore1"><br>
<label for="valore2">Valore 2:</label>
<input type="text" id="valore2" name="valore2"><br>
<input type="submit" value="Invia valori">
</fieldset>
</form>
</body>
</html>
```

# La pagina *somma.jsp*

Il valore dell'attributo action del tag form definisce la pagina che dovrà intercettare i valori. La pagina *somma.jsp* sarà così composta:

```
<%@ page language="java" import="java.lang.*,java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="IT">
<head>
<title>Somma di due numeri</title>
</head>
<body>
<h1>Somma di due numeri</h1>
<br/>
<%
String risultato;
// si preleva i due valori
String valore1=request.getParameter("valore1");
String valore2=request.getParameter("valore2");
// si convertono i valori stringa in valori numerici decimali e si sommano
try
{
int somma=Integer.parseInt(valore1)+Integer.parseInt(valore2);
risultato=valore1 +" + " + valore2 +" = " + Integer.toString(somma);
} catch (Exception e) {
//si verifica quando i valori sono errati
risultato="Valori numerici errati";
}
//si stampa il messaggio
out.write(risultato);
%>
</body>
</html>
```

# La pagina *somma.jsp* commento

La pagina *somma.jsp* preleva i valori inviati dal client grazie al metodo *getParameter* dell'oggetto implicito *request* (nelle lezioni successive analizzeremo nel dettaglio le componenti delle JSP). I valori passati saranno ovviamente in formato stringa, quindi dovremo convertirli in interi utilizzando il metodo *parseInt* della classe *Integer*.

Dopodichè costruiremo la stringa di risposta (variabile risultato).

Il costrutto ***try...catch*** serve per gestire gli errori che potrebbero riscontrarsi nelle istruzioni contenute nel blocco *try*. In Java la gestione degli errori avviene in una modalità intrinsecamente diversa da linguaggi usuali come può essere il linguaggio C o Visual Basic.

# La pagina *somma.jsp* commento

Con le eccezioni del linguaggio Java il programmatore è “obbligato” a gestire le eccezioni o comunque deve dichiarare esplicitamente di non volerle gestire, in questo modo viene sempre reso coscio che una parte di codice potrebbe generare delle eccezioni e qui sta la robustezza del linguaggio Java.

Nel nostro caso i valori inviati potrebbero essere dei caratteri alfanumerici, o valori decimali (l'algoritmo presentato gestisce solo la somma di interi) che causerebbero errori nell'esecuzione della pagina, introducendo il *try..catch* ci assicuriamo sempre il controllo della nostra applicazione anche in caso di errore.

# La somma!

Inserisci due valori numerici

Valore 1:

Valore 2:

Dopo aver creato i file proviamo che il tutto funzioni. Apriamo il browser al seguente indirizzo

<http://localhost:8080/CorsoJSP/>

Inseriamo dei valori interi e premiamo il pulsante “Invia valori” dovremmo ottenere:

## Somma di due numeri

$10 + 20 = 30$



# IDE

Per editare i documenti JSP basta un qualsiasi editor testuale, ma è preferibile usare ambienti di sviluppo che facilitano il compito di stesura del codice.

Per chi utilizza già Dremweaver è avvantaggiato dal fatto che l'ambiente di Macromedia supporta le pagine JSP. Ma per avere una gestione completa delle Java Web Application bisogna orientarsi su prodotti scritti per l'ambiente Java.

Un ottimo prodotto commerciale è il JBuilder di Borland che permette la realizzazione di progetti Java e progetti Web secondo le specifiche J2EE supportando numerosi Application Server.

Mentre in ambito Open Source c'è il progetto di IBM Eclipse (scaricabile gratuitamente all'indirizzo <http://www.eclipse.org/downloads/index.php>) che integrato con tool commerciali e gratuiti permette l'edit (con funzione di completamento automatico del codice) e il deploy delle java Web Application. Segnalo tra i plug-in commerciali il MyEclipse Enterprise WorkBench (<http://www.myeclipseide.com>) e tra quelli open source l'ottimo Lomboz ([http://forge.objectweb.org/project/showfiles.php?group\\_id=97](http://forge.objectweb.org/project/showfiles.php?group_id=97)) scaricate la versione per Eclipse 3.0 RC2 (ce ne sono tante!).

Segnalo anche gli ottimi prodotti della Sun Microsystems, Sun One Studio evoluzioni del progetto Forte for Java.

# JspServlet

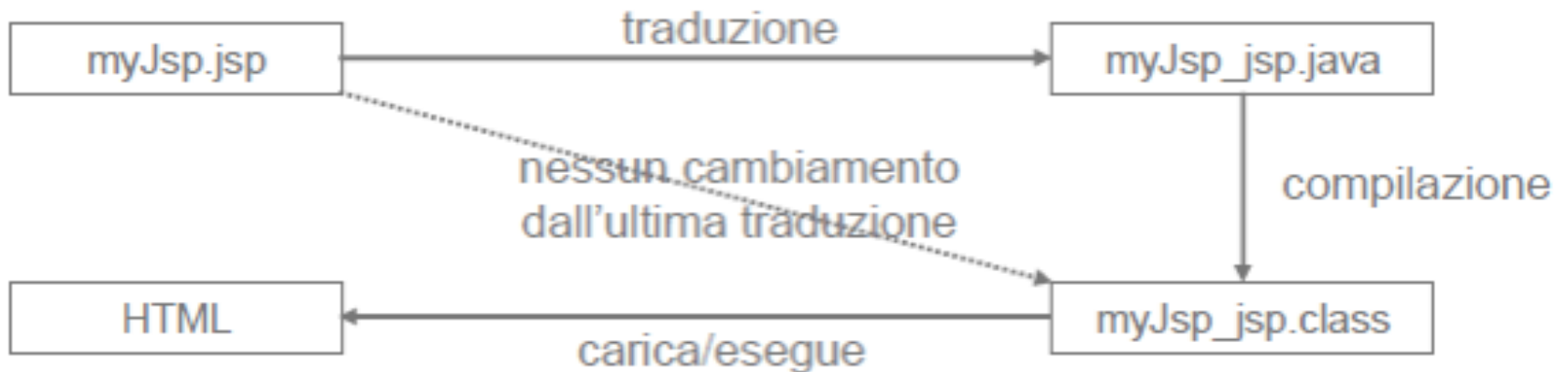
Le richieste verso JSP sono gestite da una particolare servlet (in Tomcat si chiama JspServlet) che effettua le seguenti operazioni:

traduzione della JSP in una servlet

compilazione della servlet risultante in una classe

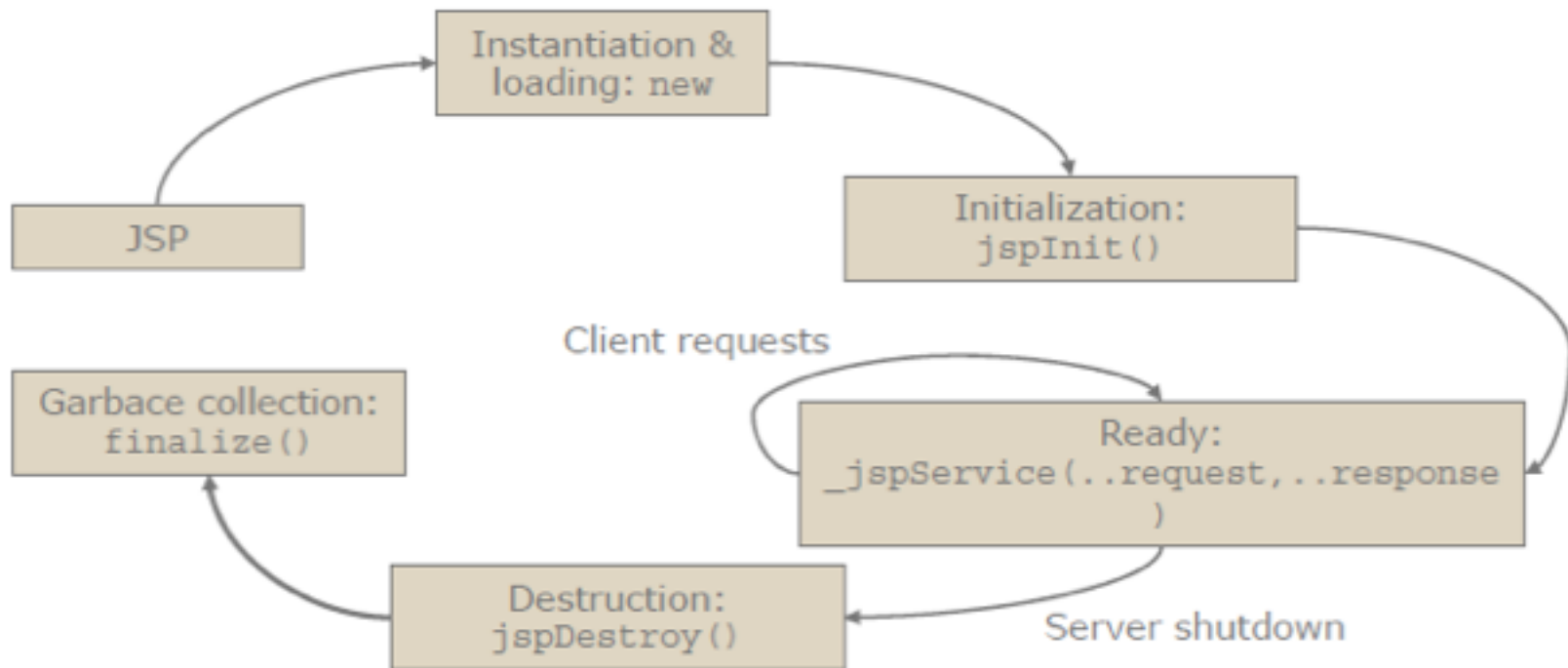
esecuzione della JSP

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



# Dichiarazioni

```
<%! dichiarazione %>
```

sia per la dichiarazioni di variabili, sia per la dichiarazione di metodi.

Esempi:

```
<%// dichiarazione di una stringa %>
```

```
<% ! String stringa=new string("ciao a tutti") %>
```

```
<% // dichiarazione di una funzione che dati due numeri in ingresso
```

```
// restituisce la loro somma %>
```

```
<% ! public int somma (int primo, int secondo){
```

```
return (primo + secondo);
```

```
}//somma %>
```

NOTA: Java mette a disposizione degli sviluppatori una classe (inclusa nel package `java.lang`) chiamata `String`, che permette, appunto, la gestione delle stringhe (da notare che da questa classe non è possibile derivare altre sottoclassi). Gli oggetti di tipo *String* possono essere inizializzati in più modi, ma normalmente i costruttori più utilizzati sono due:

# String

- `public String ()`: che costruisce una stringa vuota (perché sia allocata è necessario utilizzare l'istruzione *nomeStringa=new String []*);
- `public String (string value)`: che costruisce una stringa contenente la stringa value. I metodi implementati da questa classe più utilizzati sono:
  - `string.length ()`: restituisce il numero di caratteri che compongono la stringa escluso il terminatore;
  - `string.charAt (int i)`: restituisce il carattere della stringa in posizione i;
  - `string.concat (string str)`: restituisce una nuova stringa ottenuta dalla concatenazione delle stringa *string* con la stringa *str*;

# String

- `string.equals (string str)`: confronta la stringa *string* con la stringa *str*, restituendo `true` se risultano uguali;
- `string.equals.IgnoreCase (string str)`: confronta la stringa *string* con la stringa *str* tenendo conto delle lettere minuscole e maiuscole;
- `string.compareTo (string str)`: confronta la stringa *string* con la stringa *str*, restituendo 0 se risultano uguali, un numero positivo se *str* precede in ordine alfabetico *string* e un numero negativo se *str* anticipa segue in ordine alfabetico *string*.

# Espressioni

Sono utilizzate per inserire direttamente sull'output dei valori. La sintassi è la seguente:

`<% espressione %>`

L'espressione viene calcolata, convertita in una stringa e inserita nella pagina. Questo calcolo viene effettuato in run time, ossia quando la pagina viene richiesta, e perciò si suppone abbia completo accesso all'oggetto di interesse della richiesta.

## Esempio:

`<%= somma (2,3)%>`

Questa istruzione se inserita all'interno dei tag e stamperà a video l'output della funzione somma (in questo caso il numero 5, ottenuto dalla somma dei due valori in ingresso 2 e 3).



# oggetti impliciti

- Nel definire il codice di pagine JSP, possiamo usare tutta una serie di oggetti che non dobbiamo creare o definire noi, ma che sono già **implicitamente creati e definiti**.
- Poiché a basso livello una JSP è tradotta automaticamente in una Servlet. In questa parte useremo il termine servlet per intendere quella generata a partire dalla JSP.

## **request**

Sottotipo di javax.servlet.HttpServletRequest

Rappresenta la richiesta del client

## **response**

Sottotipo di javax.servlet.HttpServletResponse

Rappresenta la risposta del server

Poco utilizzato (la risposta è il contenuto della pagina)

## **application**

Oggetto di classe javax.servlet.ServletContext

Rappresenta il contesto della servlet derivata dalla pagina JSP

e di tutti i componenti Web della stessa applicazione

## **config**

Oggetto di classe javax.servlet.ServletConfig

Rappresenta le informazioni di inizializzazione per la Servlet derivata dalla pagina JSP

## **exception**

Oggetto di classe java.lang.Throwable

È accessibile solo da una pagina di errore

## **out**

Oggetto di classe javax.servlet.jsp.JspWriter

Rappresenta lo stream di output

## **page**

Oggetto di classe java.lang.Object

Rappresenta l'istanza della pagina JSP che sta processando la richiesta corrente

Di solito, non è utilizzata

## **pageContext**

Oggetto di classe javax.servlet.jsp.PageContext

Rappresenta il contesto della pagina JSP

Fornisce una API per gestire vari attributi

Usato soprattutto dai tag handler

## **session**

Oggetto di classe javax.servlet.http.HttpSession

Rappresenta la sessione

# oggetti impliciti *RESPONSE*

Per semplificare queste espressioni è possibile utilizzare una serie di variabili predefinite (talvolta chiamate oggetti impliciti), le cui più importanti sono:

- *RESPONSE* : fornisce tutti i parametri per inviare il risultato dell'esecuzione della pagina JSP: quando un Web Server risponde ad una richiesta di un browser, tipicamente invia prima del documento vero e proprio una *status line*, nella quale è riportata la versione del protocollo HTTP, il codice di stato (che è un valore intero), e alcuni brevi messaggi relativi al codice di stato (ad es. "OK").

# oggetti impliciti

- Questa operazione è effettuata tramite il metodo `GetStatus` dell'oggetto `response`. In particolare, i valori assumibili dal codice di stato sono i seguenti:
  - 100-199: sono valori di informazione, a cui il client deve rispondere con qualche altra azione;
  - 200-299: indicano che la request è stata effettuata con successo;
  - 300-399: tipicamente usati per file che sono stati spostati, contengono un'intestazione di location relativa al nuovo indirizzo;
  - 400-499: indicano un errore del client;
  - 500-599: indicano un errore del server

# oggetti impliciti

Di solito ci si riferisce a queste costanti tramite il loro significato, per evitare errori tipografici: per esempio, per indicare al browser di continuare a visualizzare il precedente contenuto di una pagina in quanto non ne esistono altri disponibili si userà il codice *Response.SetStatus(Response.SC\_NO\_CONTENT)* piuttosto che *Response.SetStatus(204)*. Uno dei valori che si possono trovare più frequentemente è 404 (SC\_NOT\_FOUND), il quale comunica al client che la risorsa richiesta non è reperibile a quel dato indirizzo.

# oggetti impliciti *REQUEST*

*REQUEST*: dà accesso ai parametri di richiesta, al tipo di richiesta (GET o POST), e all'intestazione HTTP (ad es. i cookies); permette quindi di accedere alle informazioni di intestazione specifiche del protocollo HTTP. Al momento della richiesta questo metodo incapsula le informazioni sulla richiesta del client e le rende disponibili attraverso alcuni suoi metodi. L'uso più comune è quello di accedere ai parametri inviati (i dati provenienti da un form per esempio) con il metodo `getParameter("nomeParametro")`, che restituisce una stringa con il valore del parametro specificato. Altro metodo molto importante è `getCookies()`, che restituisce un array di cookies. Gli altri metodi, i più importanti, sono: `getAttributeNames()` restituisce una variabile di tipo `Enumeration` contenente i nomi di tutti gli attributi coinvolti nella richiesta.

# oggetti impliciti

`getLength()`

restituisce un intero che corrisponde alla lunghezza in byte dei dati richiesti.

`getContentType()`

restituisce il tipo MIME della richiesta, cioè il tipo di codifica dei dati.

`getInputStream()`

restituisce un flusso di byte che corrisponde ai dati binari della richiesta. Particolarmente utile per funzioni di upload di file da client a server.

`getParameter(String)`

restituisce una stringa con il valore del parametro richiesto.

# oggetti impliciti

`getParameterNames()`

restituisce una variabile di tipo `Enumeration` contenente i nomi dei parametri della richiesta.

`getParameterValues(String)`

restituisce un array contenente tutti i valori del parametro specificato (nel caso ci siano più parametri con lo stesso nome).

`getProtocol()`

Rappresenta il protocollo e la versione della richiesta.

`getRemoteHost()`

restituisce l'intero nome del dominio (ad es. `whitehouse.gov`) a cui appartiene il client richiedente.

`getServerName()`

restituisce l'indirizzo IP del server

# oggetti impliciti

`getServerPort()`

Indica la porta alla quale il server è in ascolto, tipicamente sotto forma di numero intero.

`getRemoteAddr()`

Restituisce in pratica l'indirizzo IP del visitatore sotto forma di una stringa (ad es. "198.137.240.15").

`getRemoteUser()`

restituisce lo username della macchina richiedente, ed è utilizzato nei siti che richiedono una protezione.

`getPathInfo()`

restituisce informazioni extra sul path, che vengono inserite nell'URL dopo l'indirizzo del servlet.



# **oggetti impliciti**

`getQueryString()`

restituisce una stringa contenente l'intera querySting (tutti i caratteri dopo il punto di domanda).

`request.getServletPath()`

restituisce il percorso relativo della pagina jsp

# oggetti impliciti *SESSION*

*SESSION* : Una delle funzionalità più richieste per un'applicazione Web è mantenere le informazioni di un utente lungo tutto il tempo della sua visita al sito. Questo problema è risolto dall'oggetto implicito *Session* che gestisce appunto le informazioni a livello di sessione, relative ad un singolo utente a partire dal suo ingresso alla sua uscita con la chiusura della finestra del browser. È possibile quindi creare applicazioni che riconoscono l'utente nelle varie pagine del sito, che tengono traccia delle sue scelte e dei suoi dati. È importante sapere che le sessioni vengono memorizzate sul server e non con dei cookies che devono però essere abilitati per poter memorizzare il così detto SessionID che consente di riconoscere il browser e quindi l'utente nelle fasi successive.

# oggetti impliciti

I dati di sessione sono quindi riferiti e riservati ad un utente a cui viene creata un'istanza dell'oggetto *Session* e non possono essere utilizzati da sessioni di altri utenti. Per memorizzare i dati all'interno dell'oggetto *session* è sufficiente utilizzare il metodo `setAttribute` specificando il nome dell'oggetto da memorizzare e una sua istanza. Per esempio, per memorizzare il nome dell'utente al suo ingresso alla pagina ed averlo a disposizione i seguito è sufficiente fare `session.setAttribute("nomeUtente", nome)` a patto che `nome` sia un oggetto di tipo stringa che contenga il nome dell'utente.

# oggetti impliciti

La lettura di una variabile di sessione precedentemente memorizzata è possibile grazie al metodo `getAttribute` che ha come unico ingresso il nome della variabile di sessione con cui avevamo memorizzato il dato che ci interessa reperire. `Session.getAttribute("nomeUtente")` restituisce il nome dell'utente memorizzato come visto in precedenza; se non vengono trovate corrispondenza con il nome dato in ingresso, restituisce `null`.

# oggetti impliciti

Queste 2 sono le operazioni fondamentali relative all'oggetto *session*, mentre altri metodi utili sono : `getAttributeNames()` (restituisce un oggetto di tipo enumerativo di stringhe contenente i nomi di tutti gli oggetti memorizzati nella sessione corrente), `getCreationTime()` (restituisce il tempo di quando è stata creata la sessione), `getId()` (restituisce una stringa contenente il `sessionId` che come detto permette di identificare univocamente una sessione), `getLastAccessedTime()` (restituisce il tempo dall'ultima richiesta associata alla sessione corrente) , `getMaxInactiveInterval()` (restituisce un valore intero che corrisponde all'intervallo massimo di tempo tra una richiesta dell'utente ad un'altra della stessa sessione), `removeAttribute(nome_attributo)` rimuove l'oggetto dal nome specificato dalla sessione corrente.

# oggetti impliciti *OUT*

*OUT* : Questo oggetto ha principalmente funzionalità di stampa di contenuti.

Con il metodo `print(oggetto/variabile)` è possibile stampare qualsiasi tipo di dato, come anche per `println()` che a differenza del precedente termina la riga andando a capo.

Si capisce comunque che l'andare o meno a capo nella stampa dei contenuti serve solo a migliorare la leggibilità del codice HTML.

Si noti che *Out* è utilizzata quasi esclusivamente negli scriptlets, mentre le espressioni sono messe direttamente nello stream di output e quindi necessitano raramente di fare riferimento a *out* esplicitamente;

# oggetti impliciti

- *SETBUFFERSIZE(int)*: imposta la dimensione in byte del buffer per il corpo della risposta, escluse quindi le intestazioni;
- *FLUSHBUFFER()*: forza l'invio dei dati contenuti nel buffer al client;
- *CONFIG*: permette di gestire tramite i suoi metodi lo startup del servlet associato alla pagina JSP, di accedere quindi a parametri di inizializzazione e di ottenere riferimenti e informazioni sul contesto di esecuzione del servlet stesso;
- *EXCEPTION*: questo oggetto è accessibile solo dalle pagine di errore (dove la direttiva `isErrorPage` è impostata a `true`). Contiene le informazioni relative all'eccezione sollevata in una pagina in cui il file è stato specificato come pagina di errore. Il metodo principale è `getMessage()` che restituisce una stringa con la descrizione dell'errore;

# oggetti impliciti *APPLICATION*

*APPLICATION* : consente di accedere alle costanti dell'applicazione e di memorizzare oggetti a livello di applicazione e quindi accessibili da qualsiasi utente per un tempo che va dall'avvio del motore JSP alla sua chiusura, in pratica fino allo spegnimento del server. Gli oggetti memorizzati nell'oggetto application come appena detto sono visibili da ogni utente e ogni pagina può modificarli. Per memorizzare i dati all'interno dell'oggetto application è sufficiente utilizzare il metodo `setAttribute` specificando il nome dell'oggetto da memorizzare e una sua istanza. Per esempio, per memorizzare il numero di visite alla pagina è sufficiente fare `application.setAttribute("visite" , "0")`.



# oggetti impliciti

La lettura di un oggetto application precedentemente memorizzato è possibile grazie al metodo `getAttribute` che ha come unico ingresso il nome dell'oggetto application con cui avevamo memorizzato il dato che ci interessa reperire.

`application.getAttribute("visite")` restituisce l'oggetto corrispondente, in questo caso semplicemente il valore 0.

Se non vengono trovate corrispondenze con il nome viene restituito il valore null.

Anche l'oggetto application come session possiede il metodo `getAttributeNames()` che restituisce un oggetto di tipo enumerativo di stringhe contenente i nomi di tutti gli oggetti memorizzati nell'applicazione in esecuzione.

# oggetti impliciti

Per rimuovere un oggetto si utilizza il metodo

`removeAttribute("nomeoggetto")`

Infine per accedere alle costanti di applicazioni, segnaliamo due metodi: il primo molto utile è

`getRealPath("")` che restituisce (con quella stringa in ingresso) il percorso completo su cui è memorizzato il file.

Il secondo metodo restituisce delle informazioni, solitamente riguardanti la versione, del motore jsp che si sta utilizzando per l'esecuzione della pagina:

`application.getServerInfo()`

# ***Scriptlets***

Consentono di inserire nel corpo del codice qualcosa di più complesso di una semplice espressione, ossia consentono di inserire un codice arbitrario tramite la sintassi:

`<% codice %>.`

Essi hanno accesso alle medesime variabili automaticamente definite delle espressioni (*request, response, session, out, ecc..*), e consentono di eseguire alcuni task che non è permesso dalle espressioni: permettono infatti di settare il codice di stato, l'intestazione di risposta, aggiornamento di database, esecuzione di porzioni di codice contenenti loop, e altri costrutti complessi.

Vediamo un esempio: se si desidera che l'output di un'elaborazione compaia nella pagina risultante, ciò è reso possibile dalla variabile out con la seguente sintassi:

```
<%
```

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

```
%>
```

In questo caso particolare avrei ottenuto lo stesso risultato con questa espressione JSP:

Attached GET data: `<%=request.getQueryString() %>`

Questi oggetti sono finalizzati ad un migliore incapsulamento del codice, che è del resto lo spirito della programmazione Java, e descrivono le operazioni che è possibile compiere sugli oggetti attraverso l'uso delle JSP . Tale incapsulamento è da intendere in termini di inclusione e utilizzo di Java Bean.

Le azioni standard per le pagine JSP sono le seguenti:

- : permette di utilizzare i metodo implementati all' interno di un JavaBean;
- : permette di impostare il valore di un parametro di un Bean;
- : permette di acquisire il valore di un parametro di un Bean;
- : permette di dichiarare ed inizializzare dei parametri all'interno della pagina.

Sintassi:

```
name="nomeParametro" value="valore">
```

dove i parametri indicano:

name: nome del parametro dichiarato;

value: valore del parametro appena dichiarato

: permette di includere risorse aggiuntive di tipo sia statico che dinamico. Il risultato è quindi quello di visualizzare la risorsa, oltre alle informazioni già inviate al client.



Sintassi:

```
page="URLRisorsa" flush="true | false" />
```

dove i parametri indicano:

page: URL della risorsa da includere

flush: attributo booleano che indica se il buffer deve essere svuotato o meno.

Questa azione può venire completata con la dichiarazione di eventuali parametri legati agli oggetti inclusi:

```
page="URLRisorsa" flush="true|false">
{
```

.

: consente di eseguire una richiesta di una risorsa statica, un servlet o un'altra pagina JSP interrompendo il flusso in uscita. Il risultato è quindi la visualizzazione della sola risorsa specificata

Sintassi:

```
page="URLRisorsa"/>
```

dove page specifica l'URL della risorsa a cui eseguire la richiesta del servizio. Ovviamente questa azione può anch'essa venire completata con la dichiarazione di parametri:

```
page="URLRisorsa">
```

```
{
```

```
.
```

Una delle caratteristiche di Java è quella di poter gestire le eccezioni, cioè tutti quelli eventi che non dovrebbero accadere in una situazione normale e che non sono causati da errori da parte del programmatore. Dato che JSP deriva esplicitamente da Java, e ne conserva le caratteristiche di portabilità e robustezza, questo argomento è di grande rilevanza.

# Gestione Errori

Errori al momento della compilazione

Questo tipo di errore si verifica al momento della prima richiesta, quando il codice JSP viene tradotto in servlet. Generalmente sono causati da errori di compilazione ed il motore JSP, che effettua la traduzione, si arresta nel momento in cui trova l'errore ed invia al client richiedente una pagina di "Server Error" (o errore 500) con il dettaglio degli errori di compilazione.

## Errori al momento della richiesta

Questi altri errori sono quelli su cui ci soffermeremo perché sono causati da errori durante l'esecuzione della pagina e non in fase di compilazione. Si riferiscono all'esecuzione del contenuto della pagina o di qualche altro oggetto contenuto in essa. I programmatori java sono abituati ad intercettare le eccezioni innescate da alcuni tipi di errori, nelle pagine JSP questo non è più necessario perché la gestione dell'errore in caso di eccezioni viene eseguita automaticamente dal servlet generato dalla pagina JSP. Il compito del programmatore si riduce al creare un file .jsp che si occupi di gestire l'errore e che permetta in qualche modo all'utente di tornare senza troppi problemi all'esecuzione dell'applicazione JSP.

# ErrorPage

Creazione di una pagina di errore

Una pagina di errore può essere vista come una normale pagina JSP in cui si specifica, tramite l'opportuno parametro della direttiva page, che si tratta del codice per gestire l'errore.

# ErrorPage

Ecco un semplice esempio: PaginaErrore.jsp

```
<%@ page isErrorPage = "true" %>
```

Siamo spiacenti, si è verificato un errore  
durante l'esecuzione:

```
<%= exception.getMessage()%>
```



# ErrorPage

A parte la direttiva page il codice Java è composto da un'unica riga che utilizza l'oggetto *exception* (implicitamente contenuto in tutte le pagine di errore) richiamando l'oggetto *getMessage()* che restituisce il messaggio di errore.

Uso delle pagine di errore

Perché nelle nostre pagine JSP venga utilizzata una determinata pagina di errore l'unica cosa da fare è inserire la seguente direttiva:

```
<% page errorPage = "PaginaErrore.jsp" %>
```

che specifica quale pagina di errore deve essere richiamata in caso di errore in fase di esecuzione.

# cookies

I cookies non sono altro che quell'insieme di informazioni associate ad un particolare dominio che vengono memorizzate sul client sotto forma di file di solo testo, in modo da poter riconoscere un particolare utente dell'applicazione.

Con lo sviluppo di applicazioni sempre più complesse ed efficienti, l'utilizzo dei cookies è diventato indispensabile, perché se usati nel modo giusto, non fanno altro che migliorare la navigazione.

# cookies

Fatte queste premesse, vediamo quali sono le istruzioni che ci servono. Naturalmente la prima fase è quella di scrivere un cookie, compito del metodo `addCookie()` dell'oggetto `response`:

```
response.addCookie(cookie)
```

dove il parametro di ingresso `cookie` non è altro che una classe che ha alcuni metodi che ci permettono di impostare le proprietà del cookie. Ecco come:

```
Cookie mioCookie = new Cookie ("nome" , "valore");
```

definisce un oggetto `Cookie` con il rispettivo nome e valore;

# cookies

```
mioCookie.setPath("/");
```

specifica il percorso che ha il privilegio di scrittura e lettura del cookie, se omesso è inteso il percorso corrente;

```
mioCookie.setAge(secondiDiVita);
```

imposta il periodo di vita, espresso in secondi;

```
mioCookie.setSecure(false);
```

indica se il cookie va trasmesso solo su un protocollo sicuro, protetto cioè da crittografia;

# cookies

```
response.addCookie(mioCookie);
```

scrive il cookie.

A questo punto mioCookie è scritto sul client. Per leggerlo l'operazione è leggermente più complicata. L'oggetto request dispone infatti di un metodo `getCookies()` che restituisce un array di oggetti cookie trovati nel client. Una volta ottenuto l'array, è quindi necessario passare in rassegna tutti i suoi elementi fino a trovare quello che ci interessa.

```
Cookie mioCookie = null;
```

definisce in cookie che verrà letto;

# cookies

```
Cookie[] cookiesUtente = request.getCookies();  
definisce l'array di cookie e legge quelli dell'utente;
```

```
int indice = 0;  
indice per la gestione del ciclo;
```

```
while (indice < cookiesUtente.length) {  
esegue il ciclo fino a quando ci sono elementi in cookieUtente;
```

```
    if (cookiesUtente[indice].getName().equals("nomeMioCookie");  
break;  
se trova un cookie con il nome che stiamo cercando esce dal ciclo;  
    indice++;  
} //while
```

A questo punto controllando il valore dell'indice siamo in grado di capire se abbiamo trovato il nostro cookie.

# cookies

If (indice < cookiesUtente.length)

il cookie è stato trovato e viene messo nell'oggetto *mioCookie*;

mioCookie = cookiesUtente[indice];

else

mioCookie = null;

il cookie non è stato trovato.

# cookies

Ora, se il cookie è stato trovato, è possibile accedere ai suoi dati con i metodi dell'oggetto stesso, principalmente `getValue()` per ottenere il valore contenuto.

```
Valore = mioCookie.getValue()
```

Altri metodi sono:

`getName()`: restituisce il nome

`getAge()`: restituisce il periodo di vita in secondi

`getPath()`: restituisce il percorso che ha il permesso di lettura/scrittura

`getSecure()`: restituisce un booleano che dice se il cookie è sicuro oppure no.



# *Deployment Descriptor*

In seguito analizzeremo le funzionalità avanzate di questo file, per ora dovrà avere la seguente struttura:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!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>  
</web-app>
```

- Tomcat è un JSP/Servlet container (che si appoggia alla J2SE) che permette il funzionamento di pagine web e appositi programmi (servlet) scritti in java che hanno a che fare con il mondo web.
- Tomcat è un modulo integrato in JBoss, che offre più funzionalità del solo Tomcat
- Apache invece è un server web classico, molto più potente di quello integrato in Tomcat, è possibile far funzionare insieme (dividendo i compiti) Apache + Tomcat o Apache + JBoss.

- In breve Tomcat fornisce un Servlet Container che supporta tutte le specifiche delle servlet (la versione 7 di Tomcat supporta anche le Servlet 3.0). JBoss AS (Application Server), oltre al supporto delle Servlet 3.0 supporta tutto JEE 6.
- Se, invece, la nostra applicazione ha bisogno di un **supporto integrato** a diverse tecnologie **JEE** forse è il caso di pensare all'utilizzo di **JBoss**. Infatti questo server ha numerose tecnologie già integrate.
- Se avete bisogno di un server incentrato molto sul web e sulla parte di front-end delle applicazioni, **Tomcat** è perfetto. Qualora aveste bisogno di integrazione, tecnologia JEE spinta e un back-end più vasto, allora è più che consigliato guardare ad un Application Server come **JBoss**.

# Argomenti della lezione

- Preliminari
  - Architettura
  - Ciclo di Vita
- Sintassi JSP: elementi principali
  - Blocchi di Istruzioni (“Scriptlet”)
  - Oggetti Predefiniti
  - Espressioni
  - Dichiarazioni
  - Direttive
- Corrispondenza con i (o le) Servlet

# Introduzione

- Java Server Pages (JSP)
- Tecnologia per scripting server-side
  - pagine con codice HTML misto a codice Java
  - proposta da Sun
- Si fonda su tecnologia Java
  - linguaggio Java
  - classi ed interfacce specifiche
    - package javax.servlet.jsp
    - package javax.servlet.jsp.tagext
  - riuso di componenti Java (JavaBean)
  - senza dover necessariamente programmare in Java

# Introduzione

- La tecnologia JSP rappresenta un livello costruito sulla tecnologia servlet
  - sintassi rapida per sviluppare Servlet
  - *JSP container* (contenitore) o *JSP engine*
- programma (es. Tomcat) che compila/esegue JSP
- parte di un Web application server

## Architettura

# Ciclo di vita

- Il JSP container traduce la JSP in una classe servlet, che gestisce la richiesta corrente e quelle future
- Esecuzione in quattro passi:
  - 1.il *JSP engine* (o *JSP container*) parserizza la pagina e crea un sorgente Java (una classe servlet)
  - 2.il file viene compilato in un file Java *.class*
  - 3.il contenitore carica e inizializza il servlet
  - 4.il servlet elabora le richieste e restituisce i risultati
- Note:
  - I passi 1 e 2 vengono eseguiti al primo utilizzo della pagina e ad ogni aggiornamento
  - il passo 3 è eseguito solo alla prima richiesta del servlet
  - il passo 4 può essere eseguito più volte

## Ciclo di vita

# Servlet generato da una JSP

- Si ricompila e inizializza ogni volta che si modifica la JSP
  - Estende *javax.servlet.jsp.HttpJspPage*
  - Ha un unico metodo di servizio **\_jspService**
    - chiamato sia per le richieste *get* che per le richieste *post*
- ```
public void _jspService (HttpServletRequest request,  
HttpServletResponse response)
```



# Ciclo di vita: Servlet generato

## Elementi della sintassi JSP

- Blocchi di codice HTML
- Elementi di scripting
  - Commenti
  - Blocchi di istruzioni Java (“**scriptlet**”)
  - Espressioni
  - Dichiarazioni
- Direttive
  - Messaggi al JSP container (es. Tomcat) che permettono di specificare impostazioni della pagina, inclusione di contenuti da altre risorse, librerie di tag personalizzati
- Azioni
  - Tag predefiniti che incapsulano funzionalità ricorrenti
  - Consentono di creare ed usare oggetti Java (visibili anche nelle scriptlets)

# Sintassi: blocchi HTML e commenti

- Blocchi di codice HTML
  - sequenze arbitrarie di codice HTML
  - i tag vengono riscritti identicamente nella risposta
- generano `out.print(tag)` in `_jspService()`
- Due tipi di commenti:
  - commenti HTML  
`<!-- testo commento -->`
  - commenti JSP (non sono riportati nella risposta)  
`<%-- testo commento --%>`

# Scriptlet

- **Blocchi di istruzioni Java**
  - codice Java utilizzato per costruire la pagina HTML finale
  - permettono di generare dinamicamente porzioni di codice HTML in funzione dell'input
- analogamente ai servlet
  - appaiono nel metodo `_jspService()` del servlet
- **Sintassi: `<% istruzione Java %>`**
  - si può specificare qualsiasi istruzione ammessa nei metodi `doGet()` o `doPost()` di un servlet
  - nei blocchi è visibile una serie di oggetti predefiniti

# Oggetti Predefiniti

Oggetti visibili negli scriptlet (e nelle espressioni)

- `HttpServletRequest request:`
  - rappresenta la richiesta HTTP
- `HttpServletResponse response:`
  - rappresenta la risposta HTTP
- `HttpSession session:`
  - rappresenta la sessione HTTP
- `ServletContext application:`
  - rappresenta il contesto dell'applicazione
- `JspWriter out`
  - rappresenta il *PrintWriter* su cui si stampa il corpo della risposta
  - il Content-Type predefinito è "text/html"

*Scriptlet:*

## Codice implicito

- Struttura implicita di `_jspService()`
- al corpo di `_jspService()` si aggiungono:
  - il codice degli scriptlet e delle espressioni
  - le stampe dei tag HTML

```
public void _jspService ( HttpServletRequest request,  
HttpServletResponse response )  
throws java.io.IOException, ServletException {  
    ServletContext application=getServletContext();  
    HttpSession session=request.getSession();  
    JspWriter out = response.getWriter();  
    response.setContentType("text/html");  
    ...  
}
```

*Scriptlet:*

## Esempio

# *Scriptlet:* **Codice implicito**

- Struttura implicita di `_jspService()`
- al corpo di `_jspService()` si aggiungono:
  - il codice degli scriptlet e delle espressioni
  - le stampe dei tag HTML

```
public void _jspService (  
    HttpServletRequest  
    request,  
    HttpServletResponse  
    response )  
    throws  
    java.io.IOException,  
    ServletException {  
    ServletContext  
    application=getServletC  
    ontext();  
    HttpSession  
    session=request.getSes  
    sion();  
    JspWriter out =  
    response.getWriter();  
    response.setContentTyp  
    e("text/html");  
    ...  
}
```

# Scriptlet: Esempio

```
<html>
<body>
La data di oggi e' :
<%
java.util.Date d =
new java.util.Date();
out.print(d);
%>
</body>
</html>

public void _jspService (HttpServletRequest
request, HttpServletResponse response)
throws java.io.IOException, ServletException {
ServletContext Application=
getServletContext();
HttpSession session=
request.getSession();
JspWriter out = response.getWriter();
response.setContentType("text/html");
out.println("<html>"); out.println("<body>");
out.println("La data di oggi e' :");
Date d = new java.util.Date();
out.print(d);
out.println("</body>"); out.println("</html>");
}
```

**Pagina JSP**

# *Scriptlet*: Esempio

- E' possibile immergere i tag nelle strutture di controllo
  - può diminuire la leggibilità e la mantenibilità della pagina

```
<% if ( Math.random() < 0.5 ) { %>
Have a <B> nice </B> day!
<% } else { %>
Have a <B> lousy </B> day!
<% } %>
```

# Espressioni

- Forma rapida per la stampa di stringhe
  - Sintassi: `<%= espressioneJava %>`
  - Semantica
    - l'espressione è valutata, convertita in String e stampata nella pagina html generata
    - equivalente allo scriptlet:  
`<% out.print(espressione.toString()); %>`
- Esempio: `<%= new java.util.Date() %>` equivale a  
`<% out.print((new java.util.Date()).toString()); %>`



# Espressioni

- Aumentano la leggibilità del codice
- Consiglio: ridurre l'uso del metodo **print** nelle pagine JSP  
ricorrendo ad espressioni equivalenti

- Esempio:

```
<html>
<body>
La data di oggi e' :
<%= new java.util.Date() %>
</body>
</html>
```

# Alcuni esempi

- Uso di espressioni e scriptlet JSP
- Accesso ai parametri di form

# Esempio di uso delle espressioni JSP

```
<BODY>
<H2>JSP Expressions</H2>
<UL>
<LI>Current time: <%= new java.util.Date() %>
<LI>Your hostname: <%= request.getRemoteHost() %>
<LI>Your session ID: <%= session.getId() %>
<LI>The <CODE>testParam</CODE> form parameter:
<%= request.getParameter("testParam") %>
</UL>
</BODY>
```

# Oggetto *request*: accesso ad informazioni della richiesta

- Accesso ai parametri di form
  - `request.getParameter("nome")`
    - ritorna il valore della prima occorrenza del parametro *nome* nella query string
    - tratta allo stesso modo richieste GET e POST
    - ritorna *null* se il parametro non esiste
    - ritorna " " se il parametro non ha un valore
  - `request.getParameterValues("nome")`
    - ritorna un *array* con i valori di tutte le occorrenze del parametro *nome* nella query string
    - ritorna *null* se il parametro non esiste
  - `request.getParameterNames()`
    - ritorna una Enumeration di tutti i parametri della richiesta
- Altri metodi
  - `request.getMethod()`: ritorna il metodo della FORM (GET o POST)
  - `request.getHeader("nome")`: ritorna il valore di un header

# Oggetto *request*: accesso agli header della richiesta

- `request.getHeader("nome")`
  - Ritorna il valore della prima occorrenza del parametro *nome* nella query string
  - Tratta allo stesso modo richieste GET e POST
  - Ritorna *null* se il parametro non esiste
  - Ritorna " " se il parametro non ha un valore
- `request.getParameterValues("nome")`
  - Ritorna un array con i valori di tutte le occorrenze di del parametro *nome* nella query string
  - Ritorna *null* se il parametro non esiste
- `request.getParameterNames()`
  - Ritorna un oggetto *Enumeration* contenente tutti i parametri della richiesta

# Una form HTML con 3 parametri

```
<FORM ACTION="/servlet/  
coreservlets.ThreeParams">  
First Parameter: <INPUT TYPE="TEXT"  
NAME="param1"><BR>  
Second Parameter: <INPUT TYPE="TEXT"  
NAME="param2"><BR>  
Third Parameter: <INPUT TYPE="TEXT"  
NAME="param3"><BR>  
<CENTER><INPUT TYPE="SUBMIT"></CENTER>  
</FORM>
```

## *Esempio2:*

# Stampa di 3 parametri in JSP

```
<HTML>
<BODY>
<H1 ALIGN=CENTER> Reading Three Request Parameters </H1>
<UL>
<LI><B>param1</B>:
<%= request.getParameter("param1") %>
<LI><B>param2</B>:
<%= request.getParameter("param2") %>
<LI><B>param3</B>:
<%= request.getParameter("param3") %>
</UL>
</BODY>
</HTML>
```

*Esempio2:*

# Stampa di 3 parametri in JSP

***Esempio2:***



# Dichiarazioni

- Servono a dichiarare elementi esterni a *\_jspService*

- variabili di istanza

- comuni a tutte le richieste, se un solo oggetto servlet gestisce le varie richieste con thread di *\_jspService*

- variabili di classe (*static*), comuni a tutte le istanze

- metodi (di istanza o di classe)

- **Attenzione**

- nelle dichiarazioni non sono visibili gli oggetti predefiniti

- richiesta, risposta, sessione e writer sono inaccessibili

- sono visibili solo in *\_jspService()*

- soluzione: passarli come parametri ai metodi statici

# Dichiarazioni

- In particolare, permettono di ridefinire i metodi
  - `jspInit()`
    - chiamato da `init()`
  - `jspDestroy()`
    - chiamato da `destroy()`
  - operazioni tipiche di `jspInit()`:
    - `inizializzazione delle proprietà del servlet`
    - `inizializzazione del contesto dell'applicazione (ottenibile con getServletContext())`
    - `creazione di una connessione a database`

# Dichiarazioni

## esempio 1

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<%!
private String randomHeading() {
return("<H2>" + Math.random() + "</H2>");
}
%>
<HEAD> <TITLE> Random heading </TITLE> </HEAD>
<BODY>
<H1>Some Heading</H1>
<%= randomHeading() %>
```

# Dichiarazioni

## esempio 2

```
<!DOCTYPE ...>
<HTML>
<%!
public String getGreeting (String user) {
String greeting;
if (user.equals ("Bob"))
greeting = new String ("Hello, Bob!");
else
greeting = new String ("Hello there!");
return greeting;
}
%>
<BODY>
<H1>
<%
String user = request.getParameter ("user");
if (user == null) user = new String ("anonymous");
%>
<%= getGreeting (user) %>
</H1>
</BODY>
</HTML>
```

# Dichiarazioni

## esempio 3

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>JSP Declarations</TITLE>
<LINK REL=STYLESHEET
HREF="JSP-Styles.css"
TYPE="text/css">
</HEAD>
<BODY>
<H1> JSP Declarations </H1>
<%! private int accessCount = 0; %>
<H2> Accesses to page since server reboot:
<%= ++accessCount %> </H2>
</BODY>
</HTML>
```

Dichiararlo static se non si è  
sicuri che le varie richieste siano  
servite da un solo servlet



# Dichiarazioni

## esempio 3

Dopo 15 visite effettuate da un numero arbitrario di client, viene visualizzata la pagina seguente:

# Direttive

Slides parzialmente tratte da materiale di Giansalvatore Mecca (*Tecnologie di Sviluppo per il Web*) e Marty Hall (<http://www.coreservlets.com>)

- Messaggi al **JSP container**
  - il programma (es. Tomcat) che compila/esegue le JSP
- Permettono al programmatore di specificare:
  - Impostazioni della pagina (direttiva *page*)
  - Inclusione di contenuti da altre risorse (direttiva *include*)
  - Tag personalizzati usati nella pagina (direttiva *taglib*)
- Sintassi:  
<%@ direttiva attributo1= “valore1”  
...  
attributoN= “valoreN” %>

# Direttive

- Messaggi al **JSP container**
  - il programma (es. Tomcat) che compila/esegue le JSP
- Permettono al programmatore di specificare:
  - Impostazioni della pagina (direttiva *page*)
  - Inclusione di contenuti da altre risorse (direttiva *include*)
  - Tag personalizzati usati nella pagina (direttiva *taglib*)
- Sintassi:  
<%@ direttiva attributo1= “valore1”  
...  
attributoN= “valoreN” %>



# Direttiva include

- Direttiva **include**
  - attributo **file**
  - *valore: URI di una pagina jsp o html*
  - esempio: `<%@ include file="intestazione.jsp" %>`
  - consente di includere a tempo di traduzione il *codice* della pagina specificata nel servlet generato
  - nella posizione in cui si trova la direttiva include
  - **ATTENZIONE:** quando si modifica la pagina inclusa il contenitore NON è obbligato a rigenerare e ricompilare il servlet
- Alternativa: il tag `<jsp:include>` (vedremo in seguito)
  - *include a tempo di esecuzione il corpo della risposta prodotta*

# ***Direttiva include***

## Inclusione di contenuto JSP (*ContactSection.jsp*)

```
<%@ page import="java.util.Date" %>
<%
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I> No previous access</I>";
%>
<P>
<HR>
This page &copy; 2000
<A HREF="http://www.my-company.com/">my-company.com</A>.
This page has been accessed <%= ++accessCount %>
times since server reboot. It was last accessed from
<%= accessHost %> at <%= accessDate %>.
<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```

# *Direttiva include*

## Inclusione di contenuto JSP

```
...  
<BODY>  
<TABLE BORDER=5 ALIGN="CENTER">  
<TR><TH CLASS="TITLE">  
Some Random Page</TABLE>  
<P>  
Information about our products and services.  
<P>  
Blah, blah, blah.  
<P>  
Yadda, yadda, yadda.  
<%@ include file="ContactSection.jsp" %>  
</BODY>  
</HTML>
```

### *Direttiva include*

## Inclusione del contenuto JSP: risultato

## ***Direttiva include***

Inclusione del contenuto JSP:  
risultato

# Direttiva page

- Direttiva **page**

`<%@ page attr1="val1", attr2="val2", ... %>`

– attributi principali della direttiva page

- *import*
- *errorPage*
- *isErrorPage*
- *isThreadSafe*
- *contentType*

– vale per tutta la pagina

- non è posta necessariamente all'inizio

# *Direttiva page*

## Importazione di moduli Java

- Attributo *import* di page
  - specifica i package e le classi da importare
  - normalmente all'inizio della pagina
  - importazioni automatiche (da non specificare)
    - javax.servlet.\*
    - javax.servlet.http.\*
    - javax.servlet.jsp.\*
- Esempi
  - `<%@ page import="java.util.Vector" %>`
  - `<%@ page import="java.util.*" %>`
  - importazione delle librerie JDBC:  
`<%@ page import="java.sql.*" %>`  
nel servlet corrispondente si ha : `import java.sql.*;`

## *Direttiva page*

# Importazione di moduli Java

- Nota
  - è opportuno organizzare i componenti in package
  - importare esplicitamente i package con la direttiva page e l'attributo import
  - normalmente sono visibili tutte le classi nella cartella *WEB-INF/classes*
  - non è necessario importare *java.io*
  - non si usa la classe `PrintWriter` ma `JspWriter` (definita in `javax.servlet.jsp`)

## *Direttiva page*

# Pagine di errore

- Attributo **errorPage** di page
  - specifica l'URI di una pagina (HTML o JSP o Servlet) di errore
  - a cui reindirizzare il browser in caso di errori nell'esecuzione della pagina
- Esempio

```
<%@ page errorPage="errore.jsp" %>
```

  - in caso di errore il browser viene reindirizzato alla pagina “errore.jsp”



## *Direttiva page*

# Pagine di errore

- Attributo *isErrorPage* di page
  - Indica se la pagina è utilizzabile come pagina di errore
  - valore *true* o *false* (default)
  - se *true* rende visibile l'oggetto predefinito *exception*
- l'eccezione che ha prodotto la chiamata della pagina
- Esempio
  - nella pagina "errore.jsp" deve essere specificato

```
<%@ page isErrorPage="true" %>
```
  - si può così accedere all'oggetto *exception*, ad es.:

```
<% if(exception!=null)out.print(exception); %>
```

# Esempio di pagine di errore: *computeSpeed.jsp*

```
...
<BODY>
<%@ page errorPage="SpeedErrors.jsp" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE"> Computing Speed </TABLE>
<%! // toDouble non gestisce eccezioni NumberFormatException
private double toDouble(String value) {
return(Double.valueOf(value).doubleValue());
}
%>
<%
double furlongs = toDouble(request.getParameter("furlongs"));
double fortnights = toDouble(request.getParameter("fortnights"));
double speed = furlongs/fortnights;
%>
<UL>
<LI>Distance: <%= furlongs %> furlongs.
<LI>Time: <%= fortnights %> fortnights.
<LI>Speed: <%= speed %> furlongs per fortnight.
</UL>
...
```

# Esempio di pagine di errore:

## *SpeedErrors.jsp*

```
...
<BODY>
<%@ page isErrorPage="true" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
Error Computing Speed</TABLE>
<P>
ComputeSpeed.jsp reported the following error:
<I><%= exception %></I>. This problem occurred in the
following place:
<PRE>
<% exception.printStackTrace(
new java.io.PrintWriter(out)); %>
</PRE>
...
```

**Esempio di pagine di errore:  
risposta “normale”**

# **Esempio di pagine di errore: risposta in caso di errore**

# Direttiva page

## Altri attributi

- Attributo **session** di page
  - controlla l'inizializzazione dell'oggetto `session`
  - possibili valori: `true` (implicito) | `false`
  - se è `true` l'oggetto `session` viene inizializzato
  - se è `false` l'oggetto `session` non viene inizializzato automaticamente (potrebbe non esistere)
- Attributo **isThreadSafe** di page
  - valore: `true` (implicito) | `false`
  - **true**: la pagina non pone problemi di sincronizzazione
  - l'eventuale accesso concorrente deve essere gestito con sezioni critiche (`synchronized`)
  - **false**: per gestire la sincronizzazione il servlet generato implementa *SingleThreadModel*
  - garantisce che non ci siano due thread diversi che accedono alla stessa istanza di Servlet contemporaneamente
  - Attenzione: i campi statici potrebbero non essere safe

# Direttiva page

## Esempio di codice Non-Threadsafe

- La pagina JSP assegna ID agli utenti (gli ID devono essere unici!)

```
<%! private int idNum = 0; %>
<%
String userID = "userID" + idNum;
out.println(" Tuo ID=" + userID );
idNum = idNum + 1;
%>
```

- Problema: potrebbero esserci accessi concorrenti

- Soluzioni

- Impostare *isThreadSafe* = false
- Definire una sezione critica (soluzione più efficiente)

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
}
%>
```

## *Direttiva page*

# Attributo contentType

- Attributo **contentType** di page
  - serve a specificare il formato della risposta come tipo MIME
  - per default si assume una pagina HTML (tipo= **text/html**)
  - **sintassi**
    - `<%@ page contentType="MIME-Type" %>`
    - `<%@ page contentType="MIME-Type; charset=Character-Set" %>`
  - **esempi:**
    - `<%@ page contentType="text/plain" %>`
    - `<%@ page contentType="application/vnd.ms-excel" %>`
  - dall'header *accept* della richiesta è possibile ricavare quali sono i formati accettati dal client
    - `request.getHeader("Accept")`



# Principali tipi MIME

**application/msword** documento Microsoft Word  
**application/octet-stream** dati binary  
**application/pdf** file Acrobat (.pdf)  
**application/postscript** file PostScript  
**application/vnd.ms-excel** foglio Excel  
**application/vnd.ms-powerpoint** presentazione Powerpoint  
**application/x-gzip** archivio Gzip  
**application/x-java-archive** file JAR  
**application/x-java-vm** file bytecode Java (.class)  
**application/zip** archivio Zip  
**audio/basic** file musicale .au o .snd  
**audio/x-aiff** file musicale AIFF  
**audio/x-wav** file musicale Windows  
**audio/midi** file musicale MIDI  
**text/css** foglio di stile HTML  
**text/html** documento HTML  
**text/plain** file di testo  
**text/xml** documento XML  
**image/gif** immagine GIF  
**image/jpeg** immagine JPEG  
**image/png** immagine PNG  
**image/tiff** immagine TIFF  
**video/mpeg** video clip MPEG  
**video/quicktime** video clip QuickTime

# Esempio d'uso di contentType

## visualizzazione di un foglio Excel

```
<%@ page contentType="application/vnd.ms-excel" %>  
<!-- NB: le colonne sono separate da tab e non da spazi --%>  
1997 1998 1999 2000 2001  
12.3 13.4 14.5 15.6 16.7
```

# Visualizzazione condizionale di un foglio Excel

- *Idea:* scelta del formato della risposta sulla base dei parametri della richiesta
  - Excel può interpretare tabelle HTML
- *Problema:* nella direttiva page non è possibile inserire valori dinamici
- *Soluzione:* uso del metodo *setContentType* dell'oggetto implicito *response*

```
<%  
if (someCondition) {  
response.setContentType("type1");  
} else {  
response.setContentType("type2");  
}  
%>
```

# Visualizzazione condizionale di un foglio Excel

```
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
response.setContentType("application/vnd.ms-excel");
}
%>
<!DOCTYPE ...>
<HTML><HEAD> <TITLE>Comparing Apples and Oranges</TITLE> </HEAD>
<BODY> <CENTER>
<H2>Comparing Apples and Oranges</H2>
<TABLE BORDER=1>
<TR><TH></TH><TH>Apples<TH>Oranges
<TR><TH>First Quarter<TD>2307<TD>4706
<TR><TH>Second Quarter<TD>2982<TD>5104
<TR><TH>Third Quarter<TD>3011<TD>5220
<TR><TH>Fourth Quarter<TD>3055<TD>5287
</TABLE>
</CENTER> </BODY>
</HTML>
```

# Visualizzazione condizionale di un foglio Excel: risultato

Formato html (default) Foglio Excel

# Sommario

- Sintassi JSP – elementi principali:

- blocchi di codice HTML

- commenti:

`<%-- commento --%>`

- scriptlet:

`<% istruzione Java %>`

- espressioni:

`<%= espr. Java %>`

- dichiarazioni:

`<%! dich. Java %>`

- direttive:

`<%@ direttiva %>`

# Corrispondenza con le Servlet

## ***Sommario:***

- **Pagine JSP**
  - Appaiono come HTML or XHTML
  - Utilizzate soprattutto quando gran parte del contenuto è costituito da fixed-template data
- **Servlets**
  - Utilizzati quando il contenuto è quasi tutto generato dinamicamente
  - Alcuni servlet non producono contenuti
- Invocano altri servlet e pagine JSP

# Corrispondenza con le Servlet

## *Sommario:*

- Le JSP sono eseguite come parte di un Web server
  - JSP container
  - Il JSP container traduce la JSP in una servlet
- Gestisce la richiesta corrente e quelle future
- Errori per le pagine JSP
  - Translation-time
  - Durante la traduzione della pagina JSP in servlet
  - Request-time
  - Durante l'elaborazione della richiesta.



# Corrispondenza con le Servlet

## *Sommario:*

- la pagina JSP viene tradotta in una classe servlet dal contenitore
  - è una sottoclasse di servlet
- `javax.servlet.jsp.HttpJspPage`
  - metodi fondamentali
  - `_jspService()`
    - Traduzione del contenuto della pagina
    - Usato per qualunque tipo di richiesta (GET, POST)
  - `jspInit()` e `jspDestroy()`
    - Invocati dal container all'inizializzazione e alla terminazione
  - altri metodi
- si possono definire con opportuni script (dichiarazioni)
- ora servlet viene ricompilato e re-inizializzato ogni volta che il file jsp viene modificato

# Corrispondenza con i Servlet

## *Sommario:*

- Costruzione del servlet
  1. vengono importati i package standard e quelli specificati nella direttiva page
  2. vengono incluse le proprietà ed i metodi definiti nelle dichiarazioni
  3. vengono inclusi i commenti Java
  4. nel metodo `_jspService()` vengono inizializzati gli oggetti predefiniti
  5. viene assegnato il *Content-Type* predefinito
  6. i tag della pagina *jsp* danno origine a chiamate `out.print("tag");`
  7. il codice degli scriptlet viene incluso ordinatamente nel metodo `_jspService()`

# Stampa di tutti i parametri

## *Esempio 2*

```
...<BODY>
<H1 ALIGN=CENTER>"Reading all parameters"</H1>
<TABLE BORDER=1 ALIGN=CENTER>
<TR> <TH>Parameter Name<TH>Parameter Value(s)
<% Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) { %>
<TR><TD> <%= paramNames.nextElement() %> <TD>
<% String[] paramValues = request.getParameterValues(paramName);
if (paramValues.length == 1) {
String paramValue = paramValues[0];
if (paramValue.length() == 0) out.println("<I>No Value</I>");
else out.println(paramValue);
}
else { out.println("<UL>");
for(int i=0; i<paramValues.length; i++) {
out.println("<LI>" + paramValues[i]);
}
out.println("</UL>");
}
} %>
</TABLE>
</BODY></HTML>
```





















