Lezione 1 Socket java

0) Introduzione

Modalità esame:

- L'esame consiste in:
    1. un progetto che prevede la realizzazione di un’applicazione Web e una Android che comunicano tra loro;
    2. la stesura della documentazione;
    3. la discussione orale del progetto;
    4. un orale con domande sul programma del corso
- il progetto può essere svolto in gruppi di 2/3 persone. Eventualmente anche da soli;
- la consegna del progetto deve avvenire una settimana prima della data fissata per l'orale. Eventuali deroghe vanno concordate prima con il docente.

Sono disponibili anche progetti più complessi che possono valere come tesi triennale. E' anche possibile proporre al docente un proprio progetto o implementato in un linguaggio diverso. In ogni caso, è necessario accordarsi preventivamente.

Il protocollo TCP/IP prevede che la comunicazione tra due entità sia molto complessa dal punto di vista della sequenza di pacchetti da inviare, della temporizzazione degli degli invii e della gestione degli errori. I **socket** sono stati introdotti nel sistema operativo BSD 3 decenni fa come un'API che permetteva di usare la comunicazione TCP/IP in maniera semplice: come se si trattasse di una semplice accesso ad un file locale. Java supporta un accesso ancora più semplificato e object-oriented aisocket. Questo rende la comunicazione di rete sensibilmente più semplice anche rispetto al C.

1) Client/Server

Oltre ai protocolli a livello trasporto e sottostanti, molto importanti dal punto di vista del programmatore sono i protocolli a livello di applicazione. La prima grossa distinzine tra i protocolli e quella tra:
1) protocolli **Client/Server**;
2) protocolli **Peer To Peer**.
I protocolli Client/Server sono molto più semplici da descrivere e per certi versi da implementare rispetto ai protocolli Peer To Peer. Nei protocolli Client/Server la temporizzazione degli eventi è molto semplice: il **server** rimane perennemente in attesa di richieste da parte dei **client**. Ricordiamo che stiamo parlando di applicazioni quindi sia il client che il server sono due applicazioni (processi) che possono girare sulla stessa macchina o su macchina diverse.
Un esempio di client è il browser che usiamo per navigare nel Web, mentre il relativo server è il processo perennemente in attesa delle nostre richieste di pagine che indichiamo attraverso parte dell'URL. Altri esempi noti di applicazioni che sfruttano il paradigma Client/Server sono: e-mail, ftp, telnet.
I protocolli Peer To Peer hanno una temporizzazione molto più libera, esempi dei quali sono Napster, Gnutella etc. In questa lezione ci occuperemo dei protocolli Client/Server anche se i socket possono essere indipendentemente usati anche per programmare applicazioni non Client/Server. Per fissare le idee definiremo: client il processo che inizia la conversazione; server il processo che aspetta le richieste.
Per i nostri scopi, la più importante differenza tra client e server è che il client può in qualunque istante creare una socket per iniziare una conversazione con un server, mentre un server si deve preparare ad ascoltare in anticipo le possibili conversazioni in arrivo.

2) Porte standard (well-known)

Un client ha bisogno di due informazioni per connettersi a un processo server su Internet:

1) un **indirizzo** IP (o un nome di host per recuperare l'indirizzo IP tramite un server DNS);
2) un numero di **porta** (dato che il protocollo prevede la possibilità di comunicazioni contemporanee anche di tipo diverso, la porta serve al sistema operativo per individuare il processo destinatario della comunicazione stessa).
Una processo server ascolta su una porta predefinita (cioé nota a priori al client) mentre attende una connessione. I client selezionano il numero di porta corrispondente al servizio desiderato.
I numeri di porta sono codificati nelle RFC (Es. Telnet 23, FTP 21, ecc.) ma possono anche essere scelti (quasi) arbitrariamente per nuove applicazioni. In Unix le porte sotto la 1024 non possono essere usate dalle applicazioni degli utenti normali, bisogna essere "root".
Viceversa il numero di porta del client è irrilevante e viene tipicamente assegnato automaticamente dal sistema operativo durante la richiesta di cumunicazione. (Per conoscere il numero di porta creato automaticamente dal sistema possiamo usare una funzione apposita dell'API dei Socket).
Quando una client inizia la comunicazione con un server socket, la richiesta include la specifica della porta e dell'indirizzo del client: così la richiesta raggiunge correttamente non solo la macchina dove risiede il processo server, ma anche l'opportuno processo server.

3) Protocolli con e senza connessione (TCP e UDP)

Oltre a specificare indirizzo e porta il protocollo TCP/IP prevede che si deve specificare anche il tipo di trasporto per la comunicazione sulla rete. I trasporti previsti da TCP/IP sono due:

1. **UDP**: senza connessione (*connectionless*).
2. **TCP**: con connessione (*connection-oriented*);

I due tipi di trasporto offrono comunicazioni molto diversi in termini di prestazioni e affidabilità.

3.1) UDP

Un protocollo connectionless somiglia al servizio postale. Le applicazioni possono inviarsi brevi messaggi, ma non viene tenuta aperta una connessione tra un messaggio e l'altro.
Il protocollo **non** garantisce che la consegna dei dati, e **non** garantisce che questi arrivino nell'ordine corretto.
Per contro, la spedizione di un singolo dato è più efficiente nelle reti locali usando UDP rispetto a TCP ed è possibile inoltre effettuare comunicazioni in broadcast e multicast. Un messaggio trasmesso viene detto *pacchetto*.
In Java la classe `DatagramSocket` usa il protocollo connectionless UDP per inviare le informazioni.

3.2) TCP

Un protocollo connection-oriented offre l'equivalente di una conversazione telefonica. Dopo aver stabilito la connessione, due applicazioni possono scambiarsi dati.
La connessione rimane in essere anche se nessuno parla. Il protocollo garantisce che non vengano persi dati e che questi arrivino sempre nell'ordine corretto. Dato che il protocollo TCP si appoggia al protocollo IP che è inaffidabile, il costo da pagare per l'affidabilità è quello di prestazioni più basse specie nella fase iniziale della connessione (*handshake* e *slow-start*).
Con un connection-oriented protocol, il socket del client stabilisce, alla sua creazione, una connessione con il socket del server. Stabilita la connessione, un protocollo connection-oriented assicura la consegna affidabile dei dati, ovvero:

1. Ogni byte inviato viene consegnato. Ad ogni spedizione, il socket si aspetta di ricevere un acknowledgement che i byte siano stati ricevuti con sucesso. Se la socket non riceve l'acknowledgement entro un tempo prestabilito, il socket invia novamente i byte. Il socket continua a provare finché la trasmissione ha successo, o finché decide che la consegna è impossibile.
2. I byte sono letti dal socket ricevente nello stesso ordine in cui sono stati spediti. Per il modo in cui

la rete funziona, i byte possono arrivare alla macchina destinataria in un ordine diverso da quello in cui sono stati spediti, ma è compito dell'implementazione TCP riordinarli.

Un protocollo affidabile permette alla socket ricevente di riordinare i paccheti ricevuti, così che possano essere letti dal programma ricevente nell'orine in cui erano stati spediti. Al contrario dei messaggi trasmessi dal protocollo UDP, nel protocollo TCP non c'è il concetto di messaggio e anche se la nostra applicazione invia le informazioni in blocchi distinti, il protocollo TCP sono considerati un flusso continuo di informazione e il protocollo TCP può legittimamento unire o spezzare tali blocchi e l'unica garanzia è che viene preservato l'ordine dei singoli byte trasmessi.
La classe Java `Socket` usa il protocollo connection-oriented TCP.

4) Socket

La classe `java.net.Socket` rappresenta un singolo lato di una connessione socket indifferentemente su un client o su un server.
Inoltre, il server usa la classe `java.net.ServerSocket` per attendere connessioni dai client. Un applicazione server crea un oggetto `ServerSocket` e attende, bloccato in una chiamata al suo metodo `accept()`, finché non giunge una connessione. A quel punto, il metodo accept() crea un oggetto `Socket` che il server usa per comunicare con il client.
Un server può mantenere molte conversazioni simultaneamente usando un *solo* oggetto della classe `ServerSocket` e un oggetto della classe `Socket` per ogni client attivo.

4.1) `java.net.Socket`

I constructori della classe creano il socket e lo connettono allo host e porta specificati nel costruttore.
Una volta creato l'oggetto socket, i metodi `getInputStream()` e `getOutputStream()` ritornano oggetti della classe `InputStream` e `OutputStream` che possono essere usati come se fossero canali di I/O su file.
I metodi `getInetAddress()` e `getPort()` restituicono l'indirizzo remoto e la porta remota a cui il socket è connesso.
Il metodo `getLocalPort()` ritorna la porta locale usata dal socket .

4.2) `java.net.ServerSocket`

Questa classe è usata dai server per ascoltare le richieste di connessione. Quando si crea una `ServerSocket`, si specifica su quale porta si ascolta.
Il metodo `accept()` iniza ad ascoltare su quella porta, e si blocca finche' un client non richiede una connessione su quella porta.
A quel punto, `accept()` accetta la connessione, creando e ritornando un oggetto `Socket` che il server può usare per comunicare col client.

5) Client

Un applicazione client apre una connessione con un server costruendo un oggetto `Socket` che specifica hostname e port number del server desiderato:

```
try
{
    Socket sock = new
Socket("www.dsi.unive.it", 80);
    //Socket sock = new
Socket("157.138.20.6", 80);
}
catch ( UnknownHostException e )
```

```
{
// Il nome dell'host non e' valido
    System.out.println("Can't find
host.");
}
catch ( java.io.IOException e )
{
// Si e' verificato un errore di
connessione
    System.out.println("Error connecting
to host.");
}
```

5.1) Read raw byte

Una volta stabilita la connessione, input e output streams possono essere ottenuti con i metodi `getInputStream()`e `getOutputStream()` della classe `Socket` .

```
try
{
    Socket socket = new
Socket("www.dsi.unive.it", 80);
    InputStream in =
socket.getInputStream();
    OutputStream out =
socket.getOutputStream();
    // Write a byte
    out.write(42);
    // Read a byte
    int back = in.read();
    if (back<0) System.out.println("no
byte read");
    else System.out.println("1 byte read:
"+back);
    socket.close();
}
catch (IOException e )
{
  ...
}
```

5.2) Incaspulamento con `DataInputStream` e `PrintStream`

Incapsulando `InputStream` e `OutputStream` è possibile accedere agli streams in modo più semplice. Infatti con questi nuovi oggetti abbiamo a disposizione i metodi per inviare e leggere i più comuni tipi di dati Java.

```
try {
    Socket socket = new Socket("www.dsi.unive.it", 80);
    InputStream in = socket.getInputStream();
    DataInputStream din = new DataInputStream( in );

    OutputStream out = socket.getOutputStream();
    PrintStream pout = new PrintStream( out );

    // Say "Hello" (send newline delimited string)
```

```
    pout.println("Hello!");
    // Read a newline delimited string
    String response = din.readLine();
    socket.close();
} catch (IOException e ) { }
```

5.3) altri metodi dei socket

Tra i vari metodi messi a disposizione dalla classe Socket, ne vediamo ora alcuni dei più utili:

1. `getInetAddress()`: ritorna l'indirizzo IP dell'altro lato della connessione (remoto); Utile ad un server per conoscere l'indirizzo IP del client. Il cliente invece conosce già l'indirizzo del server perché prima di effetturare la connessione deve conoscere IP address e porta del processo server;
2. `getLocalAddress()`: ritorna l'indirizzo IP locale. Potrebbe essere utile ad una macchina con più indirizzi IP (proxy, router, gateway etc.);
4. `getPort()`: ritorna la porta dell'altro lato della connessione; Utile ad un server per conoscere la porta del client;
5. `getLocalPort()`: ritorna la porta locale della connessione; Utile al client per conoscere il numero di porta creato automaticamente dal sistema.

6) Server

Il lato server di un protocollo connection-oriented tipicamente esegue questa sequenza:

1. crea un oggetto della classe `ServerSocket`;
2. si mette in attesa di connessioni tramite il metodo `accept` dell'oggetto appena creato;
3. per ogni nuova connessionerichiesta da un client, il metodo `accept` ritorna un nuovo oggetto di tipo `Socket`;
4. tramite l'oggetto Socket possiamo accedere agli oggetti `InputStream` e `OutputStream` per leggere e scrivere bytes da e verso la connessione;
5. l'oggetto `ServerSocket` può ritornare ad aspettare altre connessioni attraverso il metodo `accept`.

6.1) Accettare connessioni

```
try {
    ServerSocket master = new ServerSocket( 8080 );
    while ( true )
    {
        // wait for connection
        Socket slave = master.accept();

        InputStream in = slave.getInputStream();
        OutputStream out = slave.getOutputStream();
        // Read a byte
        byte res = (byte) in.read();
        // Write a byte
        out.write(43);
        slave.close();
    }
    //master.close();
} catch (IOException e ) { }
```

6.2) Incaspulamento con `DataInputStream` e `PrintStream`

```
try {
    ServerSocket master = new ServerSocket( 8080 );
    while ( true ) {
        // wait for connection
        Socket slave = master.accept();

        InputStream in = slave.getInputStream();
        DataInputStream din = new DataInputStream( in );
        OutputStream out = slave.getOutputStream();
        PrintStream pout = new PrintStream( out );
        // Read a string
        String request = din.readLine();
        // Say "Goodbye"
        pout.println("Goodbye!");
        slave.close();
    }
    //master.close();
} catch (IOException e ) { }
```

7) Esercizi

1. iscriversi al corso su moodle.unive.it con chiave TAW2013;
2. scrivere un client per il servizio di echo;

Lezione 2 Applicazioni Web

0) Server Iterativi e concorrenti

Il server appena visto gestisce una sola connessione alla volta infatti non viene nuovamente invocata `accept()` per ascoltare nuove connessioni fino a quando non ha finito con quella corrente. Questo tipo di server viene detto **iterativo**.
Un server più realistico dovrebbe accettare più richieste (connessioni in questo caso) contemporaneamente e viene detto **concorrente**. In generale la concorrenza del server può essere ottenuta in vari modi. In C ad esempio possiamo ottenere la concorrenza attraverso concorrenza dei processi (fork), thread e I/O asincorno. Normalmente in Java si usano i thread per implementare un server concorrente. In ogni caso bisogna ricordare che la definizione di server concorrente riguarda **non** l'implementazione, ma il fatto che sia in grado di accettare e servire più richieste contemporaneamente.

0.1) Server concorrente

Vediamo ora un esempio di server concorrente che implementa il servizio di echo. Tale servizio prevede che una richiesta del client sia una qualsiasi sequenza di carattere, e che il server esaudisca la richiesta inviandola tale e quale indietro al client. Il server multi-thread che vediamo, accetta una nuova connessione, e appena ottenuto l'oggetto che rappresenta il socket, crea un nuovo thread a cui delega il compito di rispondere al client relativo alla connessione. Dopo aver creato il thread slave, il thread principale ritorna ad aspettare ulteriori connessioni attraverso il SocketServer.

```
import java.net.*;
import java.io.*;
import java.util.*;


public class TinyServer {
   public static void main( String argv[] ) throws IOException
   {
    ServerSocket ss = new ServerSocket(7);
    while ( true )
    {
        Socket temp = ss.accept();
        new ManageConnection( temp );
    }
   }
}
//classe privata
class ManageConnection extends Thread {
  Socket sock;
  ManageConnection ( Socket s )
  {
    sock = s;
    start();
  }
  public void run() {
    try {
      OutputStream out = sock.getOutputStream();
      PrintStream pout = new PrintStream(sock.getOutputStream());
      BufferedReader d = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
      String req = null;
      while ( (req=d.readLine())!=null )
```

```
      {
        pout.println( req );
      }
      sock.close();
    }
    catch ( IOException e )
   {
     System.out.println( "I/O error " + e.getMessage() );
    }
 }
}
```

Per controllare il funzionamento del server possiamo utilzzare come client un qualsiasi client telnet aggiungendo dopo il nome dell'host anche la porta del server al quale connettersi, che in questo caso è la porta numero 7. Ad esempio:
`>telnet localhost 7`
Permette di testare il funzionamento del server che gira nella medesima macchina del client.

Nell'esempio il server aspetta la richiesta del client leggendo i caratteri dal client finché non vede i caratteri di fine linea. Diverso è l'approccio quando utilizziamo un protocollo binario.

Esercizi

Scrivere un metodo per leggere una richieste (o analogamente una risposta) di un protocollo Client/Server binario. Il protocollo prevede che le richieste e le risposte siano di lunghezza variabile e la lunghezza stessa sia indicata nel primo byte binario della richiesta (max 255 byte per messaggio). Il metodo deve ritornare la lunghezza della richiesta se questa viene letta correttamente, altrimenti -1 se non si è letta una richiesta corretta (ovvero di lunghezza corretta). Nota che se non si ricevono byte, è un errore. Almeno deve essere ricevuto il byte della lunghezza, se questo byte vale 0, allora abbiamo ricevuto una richiesta corretta.

Problema, se richieste e risposte possono essere più lunghi di 255 bytes, come posso rappresentare la lunghezza? Scrivete un metodo che legge una richiesta di un protocollo Client/Server binario. Il protocollo prevede che le richieste e le risposte siano di lunghezza variabile e la lunghezza sia indicata nei primi due byte binari del messaggio stesso. Si assuma che il byte più significatgivo sia il primo (Big endian o Network byte order)

Per implementare entrambi gli esercizi studiare prima la documentazione dei metodi read(), read(byte[] b) e read(byte[] b,int off,int len) dell'interfaccia InputStream.

Per la consegna utilizzate l'apposita funzionalità di moodle.

1) Struttura del Web

Vedremo in questa lezione come sono strutturate le applicazioni web.
Il Web è composto da 3 elementi:

- URL: i nomi delle risorse nel Web;
- HTTP: il modo in cui vengono trasferite le risorse nel Web;
- HTML: il linguaggio delle risorse più importanti del Web.

1.1) URL

Gli Uniform Resource Locator ci permettono di individuare in maniera univoca una risorsa nel Web.

Un Url è composto dalle seguenti parti:

- protocollo: normalmente http, altri valori sono ftp, telnet, mailto;
- user e password (opzionali): sono le credenziali per accedere al Web Server;
- l'indirizzo del Web Server: può essere specificato tramite un I.P. address (157.138.20.15) o un nome (www.dsi.unive.it);
- una porta (opzionale): indica il numero di porta TCP alla quale connettersi;
- il path della risorsa (file normalmente) cercata;
- i parametri della richiesta (opzionali): importanti per le applicazioni Web permettono di scambiare utili parametri tra il browser e l'applicazione tramite ad esempio i FORM HTML.

Concludendo gli Url servono a individuare univocamente una risorsa nella rete.

1.2) HTTP

L'Hyper Text Transfer Protocol stabilisce delle regole per il trasferimento delle informazioni.
Il protocollo HTTP prevede come vedremo meglio più avanti, le seguenti fasi:

- il browser apre una connessione col Web server specificato nel URL;
- il browser invia la richiesta al Web server attraverso la connessione;
- il Web server invia la risposta al browser attraverso la connessione;
- il server chiude la connessione.

Le richieste inviate al server normalmente sono richieste di lettura di file (file HTML, testo, immagini etc.). Le richieste di lettura di file avvengono tramite il richieste HTTP di tipo GET. Talvolta le richieste fatte al Web server riguardano l'invio di dati dal Browser al Server. Queste richieste avvengono tramite richieste HTTP di tipo POST.

1.3) HTML

L'HyperText Markup Language è il linguaggio con il quale scrivere i documenti scambiati nel Web. Anche se non tutti i file che vengono scambiati nel Web sono scritti in HTML, i file che permettono un'interazione nelle applicazioni Web sono scritti in HTML. Allo stesso modo in cui nella busta con cui inviamo una lettera ci possiamo accludere una foto o una banconota e lettera illustra al destinatario il motivo degli altri oggetti inviati, così il file HTML specifica al browser l'aspetto grafico e alcuni aspetti semantici della pagina da visualizzare. Il linguaggio HTML ha una sintassi molto semplice che prevede i seguenti TAG principali:

- `<html>` è il tag di apertura che indica l'inizio del contenuto HTML del file;
- `<body>` è il tag di inizio della parte "visibile" del file HTML;
- `<a href="URL">` è il tag principale per specificare gli aspetti semantici di un file html, cioè i collegamenti allo stesso o altri file. Questi collegamenti vengono specificati tramite il loro "nome" ovvero l'Url;
- `<form action="URL">` è il tag che indica l'inizio di un modulo dove l'utente può inserire o selezionare dei dati da inviara dal browser al Web server (vedremo più avanti come).

I tag HTML possono avere un corrispondente tag di chiusura composta nel seguente modo: `</tag>`. Ad esempio i tag di chiusura per i tag elencati prima sono: `</html>,</body></a></form>`.

1.4) Form

I form sono usati principalmente per trasferire dati dal client al server. All'interno della coppia di tag di apertura e chiusura del form possono comparire una varietà di altri tag quali:

- `<INPUT NAME=nome TYPE=tipo VALUE=valore>` che serve per introdurre un testo. Può

avere uno dei seguenti tipi:
    - `TEXT`: per introdurre un testo libero di una riga;
    - `SUBMIT`: visualizza un pulsante per la sottomissione;
    - `PASSWORD`: per introdurre un testo nascosto;
    - `FILE`: per introdurre un file.
- `<TEXTAREA NAME=`*`nome`* `ROWS=`*`n`*`>` per introdurre più linee di testo.

2) Peculiarità delle applicazioni Web

Analizzando il funzionamento del protocollo HTTP possiamo vedere alcune sue particolarità che condizionano il funzionamento delle applicazioni Web.
Prima di tutto ogni richiesta di risorsa richiede una nuova connessione dal browser al Web server, questo comporta un rallentamento dovuto al fatto che la gestione della connessione TCP ne incrementa le prestazioni con l'uso della connessione stessa.
Sempre perché dopo l'invio della risorsa il server chiude la connessione, non c'è nessuna interazione tra browser e server fino alla prossima richiesta da parte del client (PULL). Inoltre il protocollo HTTP è senza stato e questo ci complica le cose quando dobbiamo mantenere delle informazioni riguardo le richieste precedenti.
Se vogliamo mantenere delle informazioni tra una richiesta e la successiva possiamo farlo tramite l'url rewriting e parametri nascosti oppure i cooky.
Altro problema è rappresentato dal fatto che il Web server ritorna file HTML (gli altri tipi o sono più stupidi o non sono supportati da tutti i browser) e i tipi di funzionalità è quella permessa dal linguaggio HTML all'inzio ideato solo per facilitare la navigazione, ora invece si tende ad usare il browser per implementare le funzionalità di un generico client.
Una delle funzionalità aggiunte al linguaggio HTML è **Javascritp** un linguaggio di scripting in grado di esegure script anche complessi nel browser. Javascript permette di aumentare l'interattività del browser senza bisogno di interagire col server, quindi una risposta più veloce e inoltre. Nel caso ciò non basti si può ricorrere all'uso di **Applet** Java che prevedono la realizzazione di un client ad hoc da includere nelle pagine HTML.

Lezione 3

Soluzione Esercizio

```
public int readShortRequest(InputStream in, byte[] buffer) {
 try
 {
  int offset = 0;
  int res;
  int len = in.read();
  if (len < 0) {
   return len;
  }

  while ((res = in.read(buffer, offset, Math.min(len,buffer.length) -
offset)) > 0) {
   offset += res;
   }
  if (len == offset) {
    return len;
  }
  else {
    return -1;
  }
 }
 catch (IOException ioe) {
  return -1;
 }
}

public int readLongRequest(InputStream in, byte[] buffer) {
 try
 {
  int offset = 0;
  int res =in.read();
  if (res<0)
   return -1;
  int len = res * 256;
  res = in.read();
  if(res<0)
   return -1;
  len+=res;
  while ((res = in.read(buffer, offset, Math.min(len ,buffer.length)-
offset)) > 0) {
   offset += res;
  }
  if (len == offset) {
   return len;
  }
  else {
    return -1;
  }
 }
 catch (IOException ioe) {
  return -1;
```

```
  }
}
```

1) Il protocollo Http

HyperText Transfer Protocol è il protocollo usato per trasferire ipertesti.

Si possono trovare tutti i dettagli su RFC: 1945, 2617

numero porta standard: 80.
nome porta: www.

CLIENT (BROWSER)	SERVER
apre connessione	
GET risorsa HTTP/1.0	
eventuali altre linee dello header	
(linea vuota)	
	HTTP/1.1 200 OK
	eventuali altre linee dello header
	(linea vuota)
	contenuto risorsa
	chiude connessione

1.1) Tipi di richieste del Client

Le richieste del client possono essere: GET, POST, PUT, DELETE e HEAD di cui solo GET, POST e HEAD sono attualmente usate.

1.1.1) GET

Il metodo GET serve per recuperare una qualsiasi risorsa. Se nello header viene specificato il campo If-Modified-Since la richiesta diventa condizionata al fatto che la risorsa viene trasferita solo se modificata dopo la data specificate nel campo If-Modified-Since questo per ridurre il numero di trasferimenti.

1.1.2) HEAD

Il significato del metodo HEAD è simile a quello del metodo GET e si differenzia per il fatto che il metodo HEAD ritorna solo lo header che sarebbe stato ritornato come risposta al metodo GET, ma senza il contenuto della risorsa. Questo metodo può essere utilizzato per ottenere informazioni sulla risorsa senza bisogno di trasferire la risorsa.

1.1.3) POST

Il metodo POST è usato per traferire risorse dal Client al Server.

1.2) Risposte del server

Le risposte del server possono essere:

1.2.1) 200 OK

La richiesta ha avuto successo. L'informazione restituita con la risposta dipende dal metodo usato per la richiesta nel seguente modo:

- GET: una entità che corrisponde alla risorsa richiesta viene inviata nella risposta;
- HEAD: la risposta contiene solo informazione nello header e nessun corpo;
- POST: una entita che descrive e/o contiene il risultato dell'azione.

1.2.2) 201 Created

La richiesta è stata soddisfatta e ha comportato la creazione di una nuova risorsa che può essere recuperata tramite il link specificato nella risposta. Solo il metodo POST può causare una tale risposta.

1.2.3) 202 Accepted

La richiesta è stata accettata, ma non è stata ancora completata.

1.2.4) 204 No Content

Il server ha soddisfatto la richiesta e non c'è nessuna informazione di risposta.

3.2.5) 300 Multiple Choices

La risosrsa richiesta è disponibile in una o più locazioni.

1.2.6) 301 Moved Permanently

Alla risorsa richiesta è stato assegnato un nuovo indirizzo.

1.2.7) 302 Moved Temporarily

La risorsa richiesta è stata spostata temporaneamente ad un nuovo indirizzo.

1.2.8) 304 Not Modified

La risorsa richiesta non è stata modificata. Risposta ottenuta da una richiesta GET con campo If-Modified-Since field attivo.

1.2.9) 400 Bad Request

La richiesta non è sintatticamente corretta.

1.2.10) 401 Unauthorized

La risorsa richiesta richiede l'utenticazione dell'utente. La risposta include il campo WWW-Authenticate. Il cliente può rifare la richiesta con un campo Authorization adatto.

1.2.11) 403 Forbidden

La risorsa non è disponibile per essere pubblicata.

1.2.12) 404 Not Found

Il server non ha trovato la risorsa.

1.2.13) 500 Internal Server Error

Il server ha trovato un errore inaspettato che non gli ha permesso di portare a termine la richiesta.

1.2.14) 501 Not Implemented

Il server non supporta la funzionalità richiesta.

1.2.15) 502 Bad Gateway

Il server mentre si comportava da gateway o proxy ha ricevuto una risposta non valida.

1.2.16) 503 Service Unavailable

Il server è temporaneamente non in grado di gestire la richiesta.

1) Server HTTP

Il server HTTP o server Web (ad esempio Apache o IIS), è un programma sempre in esecuzione che aspetta richieste dai client e ritorna le risorse specificate nelle richieste. Ci sono due tipi fondamentali di risorse disponibili nel Web server:

1. risorse statiche: le risorse esistono prima della richiesta e/o il loro contenuto è fissato;
2. risorse dinamiche: le risorse vengono create al momento della richiesta e in maniera automatica con programma.

In generale le risorse che un Web server gestisce sono principalmente pagine html, file di testo, formati per lo scambio di documento di testo (Adobe Portable Document Format, PostScript,), immagini (GIF, JPEG, PNG), animazioni e altri contenuti multimediali. Tutti queste risorse possono essere statiche o dinamiche, anche se generalmente vengono generate pagine html. Per la gestione delle risorse statiche è sufficiente un implementazione utilizzando le tecniche già viste la scorsa lezione:

1.1) Server Http

TinyHttpd ascolta su una porta specificata e serve semplici richieste HTTP:

```
GET /path/filename HTTP/1.0
```

Il Web browser manda una o più di queste linee per ogni documento da ottenere. Letta la richiesta, il server prova ad aprire il file specificato e ne spedisce il contenuto. Se il documento contiene referenze ad immagini o altro da mostrare online, il browser continua con richieste GET addizionali. Per ragioni di performance, TinyHttpd serve ogni richiesta in una sua thread. Perciò TinyHttpd può servire molte richieste concorrentemente.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class TinyHttpd {
 public static void main( String argv[] ) throws IOException {
 ServerSocket ss =
 new ServerSocket(Integer.parseInt(argv[0]));
 while ( true )
 new TinyHttpdConnection( ss.accept() );
 }
}
class TinyHttpdConnection extends Thread {
 Socket sock;
 TinyHttpdConnection ( Socket s ) {
 sock = s;
 setPriority( NORM_PRIORITY - 1 );
 start();
 }
 public void run() {
 try {
 OutputStream out = sock.getOutputStream();
 BufferedReader d = new BufferedReader(new InputStreamReader(sock.getInputStream()));
 String req = d.readLine();
 System.out.println( "Request: "+req );
 StringTokenizer st = new StringTokenizer( req );
 if ( (st.countTokens() >= 2) && st.nextToken().equals("GET") )
 {
 if ( (req = st.nextToken()).startsWith("/") )
```

```
 req = req.substring( 1 );
 if ( req.endsWith("/") || req.equals("") )
 req = req + "index.html";
 try
 {
 FileInputStream fis = new FileInputStream ( req );
 byte [] data = new byte [ fis.available() ];
 fis.read( data );
 //Esercizio: scrivere header
 out.write( data );
 }
 catch ( FileNotFoundException e )
 {
 new PrintStream( out ).println("404 Not Found");
 }
 }
 else new PrintStream( out ).println( "400 Bad Request" );
 sock.close();
 }
 catch ( IOException e )
 {
 System.out.println( "I/O error " + e );
 }
}
}
```

1.2) Esempio d'uso del server

Dopo aver compilato il file `TinyHttpd.java` lo si inserisce nel class path. In una directory in cui esiste il file `prova.html` si avvia il daemon, specificando un numero di porta libero come argomento. Per esempio:

```
> java TinyHttpd 8080
```

Utilizzando un client standard HTTP che in questo caso è un browser Web tipo Internet Explorer o Netscape possiamo verificare il funzionamento del server appena creato. L'URL dovrà essere: `http://localhost:8080/prova.html`. Se il computer è collegato in rete e ha un IP address o un nome, allora le pagine potranno essere accessibili anche da altri computer specificando: `http://nome.dominio:8080/prova.html` oppure `http://ip.address:8080/prova.html`.

1.3) note sull'implemetazione

Abbassando la sua priorita' a `NORM_PRIORITY-1`, assicuriamo che il thread che serve le connessioni già stabilite non blocchi la main thread di `TinyHttpd` impedendo di accettare nuove richieste.
Anche se l'esempio serve allo scopo, in ogni casi ci sono tutta una serie di aspetti che l'implementazione di un Web server reale deve considerare:

1. consuma un sacco di memoria allocando un array enorme per leggere un intero file tutto d'un colpo. Un'implementazione piu' realistica userebbe un buffer e invierebbe i dati in più passi.
2. `TinyHttpd` inoltre non riconosce semplici directories. Non sarebbe difficile aggiungere alcune lineee per leggere directories e generare liste di link HTML come fanno molti web

server.

3. `TinyHttpd` soffre di limitazioni imposte dalla debolezza di accesso al filesystem. È importante ricordare che i pathnames dipendono dal sistema operativo, così come il concetto stesso di filesystem. TinyHttpd funziona, così com'è, su sistemi UNIX e DOS-like, ma richiede alcune variazioni su altre piattaforme. È possibile scrivere del codice più elaborato che usa informazioni sull'ambiente offerte da Java per adattarsi al sistema locale.
4. mancano tutti i meccanismi per la generazione di pagine dinamiche: CGI, SSI, php, servlet etc;
5. il problema principale con `TinyHttpd` è che non ci sono restrizioni sui files a cui puo' accedere. Con qualche trucco, il daemon spedirebbe un qualunque file del suo filesystem al client.

Facoltativo:
Per ovviare all'ultimo problema, potremmo fare in modo che prima di aprire il file, controllare che il path del file sia di un file (o directory) che sia nelle nostre intenzioni rendere pubblico. Un tale approccio è semplicemente realizzabile ma ha lo svantaggio di richiedere che in tutta la nostra applicazione ad ogni accesso di un file dobbiamo ricordarci di controllare se quel file è leggibile. Un approccio diverso è quello di appoggiarsi a delle caratteristiche standard di Java. In questo caso useremo un'estensione della classe `SecurityManager` per gestire in automatico l'accesso al file system. La nascita di Java è stata motivata dalla necessità di un linguaggio in cui la gestione degli accessi fosse sicura per poter far giare delle Applet in macchine ospiti. Quindi sono già presenti meccanismi che in automatico limitano l'accesso al File System. Questo meccanismo è appunto il `SecurityManager`. Di solito il `SecurityManager`
standard non permette di fare nessuna delle operazioni di seguito elencate:

1. `checkAccess(g)`: si può accedere ai dati del thred g?
2. `checkListen(p)`: si può accettare connessioni dalla porta p?
3. `checkLink(l)`: si può collegare con la libreria dinamica l?
4. `checkPropertyAccess(k)`: si puo accedere alla proprietà di sistema k?
5. `checkAccept(h,p)`: si può accettare connessioni dall'host h e porta p?
6. `checkWrite(f)`: si può scrivere il file f?
7. `checkRead(f)`: si può leggere il file f?

La classe `SecurityManager` impedisce tutte queste operazioni generando una eccezione `SecurityException` per ognuno dei metodi elencati. Quindi per fare in modo che il nostro server faccia tutto quello che gli serve eccetto per la lettura dei file non contenuti nella sottocartella, dobbiamo sovrascrivere i metodi in modo che non generino eccezzioni eccetto quando si tenta di leggere un file non consentito.

```
import java.io.*;

class TinyHttpdSecurityManager
extends SecurityManager {
  public void checkAccess(Thread
g) { };
  public void checkListen(int
port) { };
  public void checkLink(String
lib) { };
  public void
```

```
checkPropertyAccess(String key)
{ };
  public void checkAccept(String
host, int port) { };
  public void
checkWrite(FileDescriptor fd) { };

  public void
checkRead(FileDescriptor fd) { };
  public void checkRead( String
s )
  {
    if ( s.startsWith("/") ||
(s.indexOf("..") != -1) )
       super.checkRead(s);
  }
}
```

Il cuore di questo security manager e' il metodo `checkRead()`.
Controlla due cose: assicura che il pathname non sia assoluto, e che il pathname non contenga una coppia di punti (..) che risale l'albero delle directories.
Con questi vincoli siamo certi (almeno su un filesystem UNIX o DOS-like) di aver ristretto l'accesso alle solo subdirectories della directory corrente. Se il pathname è assoluto o contiene "..", `checkRead()` lancia una `SecurityException`.
Gli altri metodi che nell'esempio non fanno nulla a esempio checkAccess(), vengono invocati dal security manager e non impediscono al demone di fare il proprio lavoro.
Quando installiamo un security manager, ereditiamo implementazioni di molte "check" routines. L'implementazione default non permette di fare nulla; Lancia una security exception appena chiamata. Dobbiamo permettere cosicchè il daemon possa fare il suo lavoro; deve accettare connections, ascoltare sockets, creare threads, leggere property lists, ecc. Per questo sovrascriviamo i metodi default.
Per installare il security manager, aggiungere la seguente riga all'inizio del metodo `run` di `TinyHttpdConnection`:

```
System.setSecurityManager( new TinyHttpdSecurityManager()
);
```

Per catturare le security exception, aggiungere il seguente catch dopo la catch di `FileNotFoundException`:

```
    catch ( SecurityException e )
    {
        new PrintStream( out ).println( "403
    Forbidden" );
   }
```

Per la creazione delle pagine html, ma non solo, sono stati predisposti dei meccanismi da agganciare ai Web server in modo da aggiungere la generazione delle pagine dinamiche. Tra le varie tecniche adottate, la prima con un alto grado di flessibilità e potenza è stato lo standard CGI che vedremo nel prossimo paragrafo. Altre soluzioni alternative sono i SSI, linguaggi di scripting come PHP, Perl etc, le servlet e le JSP che vedremo più avanti.

2) CGI

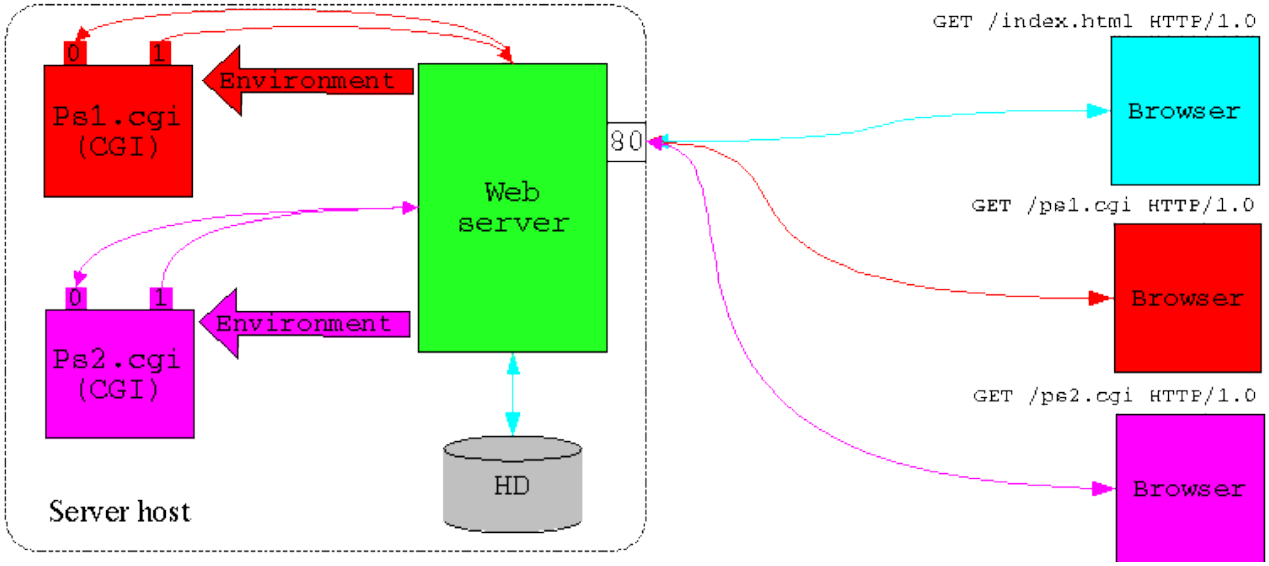
La Common Gateway Interface è uno standard di comunicazione tra web-server e applicazioni. E' stato definito da Netscape. Lo standard CGI permette ad un web server di colloquiare con un'applicazione esterna in modo da integrare i servizi forniti dal server web alle particolali informazioni dinamiche fornite dall'applicazione. I vantaggi che si ottengono da questo approccio sono:

- sviluppo delle sole funzionalità peculiari del sistema (uso del server Web per tutte le funzionalità standard);
- gestione della concorrenza affidata al server Web;
- pre-elaborazioni delle richieste dei client fatta dal server Web (vedi ENVIRONMENT);
- sviluppatore delle applicazioni ignora quasi i problemi di comunicazione e di concorrenza;
- essendo uno standard, posso scrivere applicazioni portabili per quanto riguarda la comunicazione con i web server.

Lezione 5

Architettura CGI

Le richieste di risorse statiche vengono esaudite direttamente dal web server, mentre per quelle in cui c'è bisogno di eseguire un'applicazione separata, si crea un processo figlio e tramite la ridirezione dell'input/output la richiesta viene inviata all'applicazione. Schematicamente abbiamo la seguente situazione:



Vediamo dallo schema che il web-server deve essere in grado di:

- distinguere tra richieste di pagine statiche e invocazioni di applicazioni CGI: nella figura abbiamo aggiunto il suffisso .cgi per indicare le applicazioni CGI.
- distinguere tra applicazioni CGI;
- creare l'**environment** per l'applicazione CGI;
- creato un **processo** per ogni invocazione di applicazione CGI;
- "deviare" (**ridirezionare)** la connessione con il client nello stdin e stdout del processo creato.

Per distinguere le applicazioni CGI dalle pagine statiche, ci sono due metodi principali:

- i file che terminano con un suffisso particolare (generalmente `.cgi`);
- i file contenuti in particolari directory (generalmente la directory `cgi-bin`).

Ad esempio il web server Apache, ha il file di configurazione `/etc/httpd/conf/httpd.conf` in cui possiamo specificare i suffissi che rappresentano le applicazioni CGI nel seguente modo:

```
AddHandler cgi-script .cgi
```

Inoltre possiamo specificare le directory che contengono solo applicazioni CGI e quindi indipendentemente dal suffisso vengono trattati come tali, nel seguente modo:

```
ScriptAlias /cgi-bin/ "/home/httpd/cgi-bin/"
```

Dobbiamo ricordarci anche che ogni directory in cui sono contenuti delle applicazioni CGI deve avere impostato l'opzione per eseguire i CGI stessi.

```
#
# "/home/httpd/cgi-bin" should be changed to whatever your
ScriptAliased
# CGI directory exists, if you have that configured.
#
<Directory "/home/httpd/cgi-bin">
    AllowOverride None
```

```
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```
Per avere una directory di applicazioni CGI nella home di ogni utente bisogna aggiungere la seguente dichiarazione:
```
<Directory /home/*/public_html/cgi-bin>
    Options ExecCGI
    SetHandler cgi-script
</Directory>
```

Esempio

Il primo esempio consiste nel programma C per visualizzare la pagina html con scritto “Hello World!”.

```
/*********************************************************
*  cgiweb.c                                              *
*********************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define BUFLEN 1024
char buf[BUFLEN];
main(int argc, char *argv[])
{
    write(1,"Content-type: text/html\015\012",25);
    write(1,"Status: 200 OK\015\012",17);
    write(1,"\015\012",2);
    while(1,"<html><body>Hello World!</body></html>",38);
    close(file);
}
```

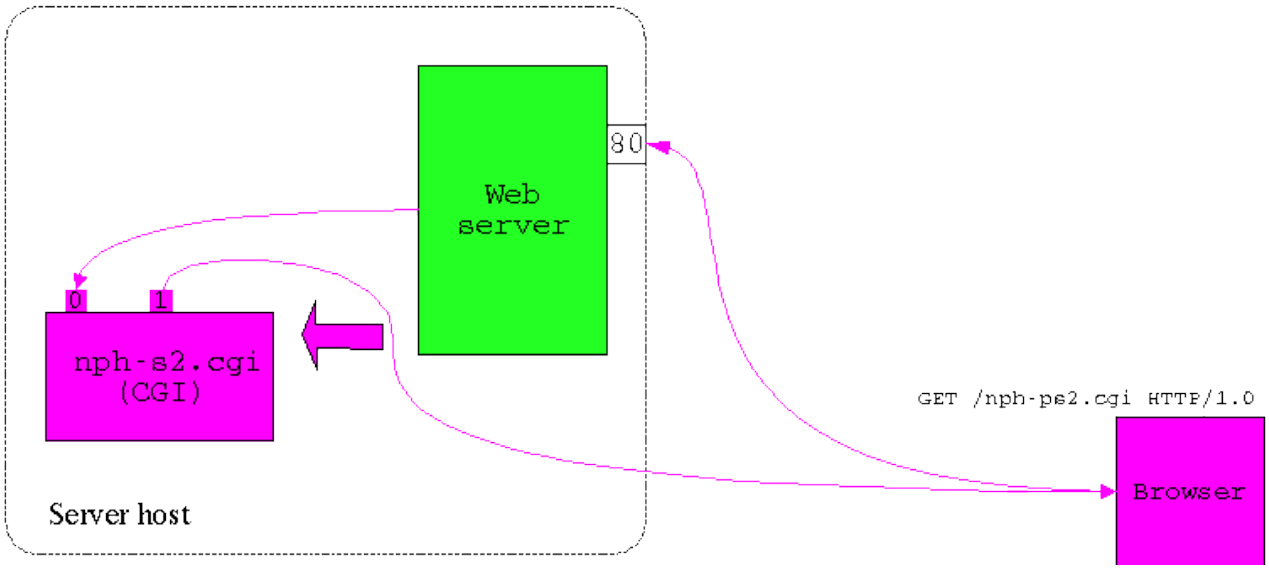
L'esempio seguente visualizza il file env.html:

```
/*********************************************************
*  cgiweb.c                                              *
*********************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define BUFLEN 1024
char buf[BUFLEN];
main(int argc, char *argv[])
{
    int n=0,i=0;
    int file;

    file=open("/public/taw/lezione3/env.html", O_RDONLY);
    if (file<0)
    {
      write(1,"Content-type: text/html\015\012",25);
      write(1,"Status: 404 NOT FOUND\015\012",24);
      write(1,"\015\012",2);
      return 0;
    }
    write(1,"Content-type: text/html\015\012",25);
    write(1,"Status: 200 OK\015\012",17);
    write(1,"\015\012",2);
    while ((n = read(file, buf, BUFLEN)) > 0)
       write(1, buf, n);
    close(file);
}
```

Possiamo anche inviare direttamente la risposta al cliente senza bisogno di passare dal server, in questo caso bisogna inviare glie header esattamente come previsto dal protocollo HTTP perché lo standard CGI prevede che per le applicazioni che iniziano per `nph-` (not parse headers), l'output venga inviato direttamente al client senza che il server esamini il contenuto dello header.

```
/*********************************************************
*  nph-cgiweb.c                                          *
*********************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define BUFLEN 1024
char buf[BUFLEN];
main(int argc, char *argv[])
{
    int n=0,i=0;
    int file;

    file=open("/public/taw/lezione2/env.html", O_RDONLY);
    if (file<0)
    {
      write(1,"HTTP/1.0 404 NOT FOUND\015\012",24);
      write(1,"\015\012",2);
      return 0;
    }
    write(1,"HTTP/1.0 200 OK\015\012",17);
    write(1,"Content-type: text/html\015\012",25);
    write(1,"\015\012",2);
    while ((n = read(file, buf, BUFLEN)) > 0)
       write(1, buf, n);
    close(file);
}
```

Questo comportamento è illustrato dal seguente schema:



Invocazione di un'applicazione CGI

Un'applicazione CGI può essere invocata da un client in questi modi:

- tramite link in questo caso il metodo di invocazione e GET (o HEAD);
- tramite la sottomissione di un FORM in questo caso il metodo può essere sia GET che POST.

Nel caso del metodo GET, il server riceve una richiesta del tipo:

```
GET /nome-cgi HTTP/1.0
...
```

oppure nel caso siano specificati degli argomenti da passare alla applicazione:

```
GET /nome-cgi?
nome_arg1=val_arg1&nome_arg2=val_arg2&..&nome_argn=val_arg_n HTTP/1.0
...
```

Nel caso del metodo POST, il server riceve sempre una richiesta del tipo:

```
POST /nome-cgi HTTP/1.0
....
Content-type: ....
Content-lenght: nnnn

XXX
```

e gli eventuali argomenti opportunamente codificati rappresentano il contenuto della risorsa inviata al server, quindi compaiono dopo lo header e sono indicati da XXX nell'esempio. La codifica può essere:

- `application/x-www-form-urlencoded`
- `multipart/form-data`

Nel primo caso, la codifica è la stessa che per il metodo GET, quindi `XXX` sarebbe uguale a `nome_arg1=val_arg1&nome_arg2=val_arg2&..&nome_argn=val_arg_n` . Tutti i caratteri con significato particolare vengono codificati in ASCII esadecimale anteponendo il carattere `%`, quindi ad esempio se nel valore compare il simbolo & questo viene sostituito dai caratteri `%26`.
Nel secondo caso, la codifica la esamineremo successivamente.

environment

Tutti i dati dello header della richiesta del client vengono analizzati dal server che li deve rendere disponibili alla applicazione CGI. Il passaggio di queste informazioni e di altre che solo il server conosce avviene attraverso l'ambiente passato al processo dell'applicazione CGI. La lista completa delle informazioni passate al processo è:

Le seguenti informazioni non dipendono dalla richiesta e sono le stesse per ogni richiesta inviata:

- `SERVER_SOFTWARE`

  Il nome e la versione del web server. Formato nome/versione;

- `SERVER_NAME`

  Il nome, DNS alias, o IP address della macchina che ospita il server.

- `GATEWAY_INTERFACE`

  La versione delle specifiche CGI che il server adotta.Formato: CGI/revisione.

Le seguenti informazioni sono specifiche della richiesta inviata:

- `SERVER_PROTOCOL`

  Il nome del protocollo e la versione specificato nella richiesta del client. Formato: protocollo/revisione

- `SERVER_PORT`

  Il numero di porta alla quale la richiesta è stata inviata, di solito 80.

- `REQUEST_METHOD`

  Il metodo specificato nella richiesta: GET, HEAD, POST, etc.

- `PATH_INFO`

  Il path specificato nella richiesta che segue il path del CGI stessto (extra path). In altre parole gli script CGI possono essere richiamati usando il loro path virtuale (cioè il path visto dal client) seguito da ulteriori informazioni alla fine del path virtuale. Le ulteriori informazioni sono presenti in PATH_INFO.

- `PATH_TRANSLATED`

  Il server mappa il contenuto di PATH_INFO trasformandolo nel path corrispondente della macchina locale.

- `SCRIPT_NAME`

  Il path virtuale dello script CGI richiamato. Usato per scrivere codice autoreferente.

- `QUERY_STRING`

  L'informazione che segue il simbolo '?' nell'URL della richiesta.

- `REMOTE_HOST`

  Il nome della macchina del client. Se il server non conosce il nome della macchina del client, dovrebbe impostare solo REMOTE_ADDR.

- `REMOTE_ADDR`

  L'IP address della macchina del client.

- `AUTH_TYPE`

  Il protocollo di autenticazione utilizzato per autenticare il client.

- `REMOTE_USER`

  L'username del client.

- `REMOTE_IDENT`

  Se il server HTTP supporta il protocollo RFC 931 per identificazione, allora questa variabile contiene il nome dell'utente remoto recupaerato dal server.

- `CONTENT_TYPE`

  Per richieste con inviano informazioni come HTTP POST e PUT, questa variabile dovrebbe contenere il tipo di contenuto.

- `CONTENT_LENGTH`

  La lunghezza del contenuto così come viene indicata dal client.

In aggiunta agli header appena visti, tutti glie eventuali altri header inviati dal client vengono messi nell'environment con il prefisso HTTP_ seguiti dal nome dell'header. Ogni carattere '-' nel nome dell'header viene sostituito dal carattere '_ '. Il server può escludere tutti gli header che ha già processato quali Authorization, Content-type, e Content-length. Se necessario il server può escludere ogni altro header se si superano eventuali limiti nell'uso delle risorse di sistema,

Esempi di questi variabili che possono essere escluse:

- `HTTP_ACCEPT`

  I tipi MIME che il cliente accetta come risposta. Ogni tipo dev'essere separato da una virgola.

  Formatp: tipo/sottotipo, tipo/sottotipo.

- `HTTP_USER_AGENT`

  Il nome del programma software usato come client. Formato: `software/version library/version`.

Dal punto di vista del programmatore di applicazioni CGI ci interessa sapere come accedere a queste informazioni. Da un programma scritto in C possiamo accedere alle variabili d'ambiente con la funzione `getenv`, mentre da uno script della shell basta anteporre il simbolo `$` ai nomi delle variabili.

stdin

Dallo standard input si può leggere tutto quello che il client invia dopo lo header al server. Quindi nulla per i metodi GET e HEAD. Mentre per i metodi POST e PUT potremmo leggere un contenuto di tipo `CONTENT_TYPE` e lunghezza `CONTENT_LENGTH`.

stdout

L'output dell'applicazione viene o meno interpretato dal server web. Se il nome dell'applicazione inizia per `nph-` allora l'output viene inviato direttamente al client. In questo caso è compito dell'applicazione CGI inviare tutti gli header in maniera corretta. In tutti gli altri casi è il server web che invia lo header di risposta e tutti gli altri campi dello header ritornato dall'applicazione eccetto per i seguenti campi:

1) Content-type:

E' il tipo MIME del documento ritornato dall'applicazione.

2) Location:

Indica al server web che la risposta dell'applicazione CGI indica dove trovare il documento vero e proprio invece che ritornarlo direttamente. In questo caso ci sono due possibilità:

1. l'argomento del campo Location è un URL, quindi il client ridireziona il client al nuovo indirizzo;
2. l'argomento del campo Location è un path, quindi il server ritorna il documento come se il client lo avessi richiesto direttamente.

3) Status:

Viene usato per indicare al server il tipo di risposta da inviare al client. Il formato dell'argomento è:
`Status: nnn XXXXX`
dove nnn è un numero di 3 cifre con il codice di stato e XXXXX è la stringa che descrive lo stato, ad esempio
`Status: 404 NOT FOUND`
viene trasformato dal server in
`HTTP/1.0 404 NOT FOUND`

Esempio accesso Environment

Un esempio di accesso alle variabili d'ambiente da uno script di shell è lo script seguente:

```
#!/bin/sh
echo Content-type: text/plain
echo
echo CGI/1.0 test script report:
echo
echo argc is $#. argv is "$*".
echo
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = $PATH_INFO
echo PATH_TRANSLATED = $PATH_TRANSLATED
echo SCRIPT_NAME = $SCRIPT_NAME
echo QUERY_STRING = $QUERY_STRING
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

Un esempio di accesso allea variabile d'ambiente `PORT` da un programma in C è il programma:

```
printf("La porta:%s\n",getenv("SERVER_PORT"));
```

Esempio Form HTML con metodo GET

Un esempio di una pagina html per creare un form (supponiamo che la pagina sia visibile con l'URL: `http://www.dsi.unive.it/~taw/Lezione3.html`):

```
<html>
<body>
<FORM METHOD="GET" ACTION="testform.cgi" ENCTYPE="application/x-www-form-urlencoded">
Nome:
<INPUT TYPE="text" NAME="nome" VALUE="">
<BR>
Cognome:
<INPUT TYPE="text" NAME="cognome" VALUE="">
<BR>
<TEXTAREA NAME="linee" ROWS="5">
Linea1
Linea2
Linea3
</TEXTAREA>
```

```
<BR>
<INPUT TYPE="submit" VALUE="ok">
</FORM>
</body>
</html>
```

Il browser visualizza la pagine predente nel seguente modo:

Nome:

Cognome:

ok

Se in corrispondenza dell'URL `http://www.dsi.unive.it/~taw/testform.cgi` troviamo uno script CGI col seguente codice:

```
#!/bin/sh
echo Content-type: text/plain
echo
echo argc is $#. argv is "$*".
echo
echo REQUEST_METHOD = $REQUEST_METHOD
echo PATH_INFO = $PATH_INFO
echo PATH_TRANSLATED = $PATH_TRANSLATED
echo SCRIPT_NAME = $SCRIPT_NAME
echo QUERY_STRING = $QUERY_STRING
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
cat
```

premendo il pulsante l'output che otterremo nella finestre del browser sarà:

```
argc is 0. argv is .
REQUEST_METHOD = GET
PATH_INFO =
PATH_TRANSLATED =
SCRIPT_NAME = testform.cgi
QUERY_STRING = nome=&cognome=&linee=Linea1%0D%0ALinea2%0D%0ALinea3%0D%0A
CONTENT_TYPE =
CONTENT_LENGTH =
```

Nel browser l'URL visualizzato sarà:
`http://www.dsi.unive.it/~taw/lezione3/testform.cgi?nome=&cognome=&linee=Linea1%0D%0ALinea2%0D%0ALinea3%0D%0A`
Come si può vedere tutti i campi del form vengono codificati e aggiunti all'url con il quale viene invocata l'applicazione CGI. Dato che ci sono dei limiti nella lunghezza degli url (alcuni browser considerano al massimo url di 255 caratteri), questo metodo non può essere applicato sempre.
Notiamo inoltre che i valori dei parametri vengono codificati con la codifica url-encoded che trasforma gli spazi in '+' e tutti i caratteri che non sono lettere o numeri in codifica esadecimale preceduta dal simbolo '%'. I vari parametri sono separati dal simbolo '&'.

Esempio Form con metodo POST con codifica url-encoded

Il metodo GET va usato quando ci sono operazioni di lettura e ricerca. Non andrebbe mai usato quando ci sono operazioni che comportano una modifica alla base dati in senso lato. Inoltre possiamo usare il metodo POST:

- per trasferire molti dati dal client al server;
- per non far vedere all'utente del browser che dati vengono inviati.

Il codice HTML per usare il metodo POST è il seguente:

```
<html>
<body>
<form method="POST" action="testform" enctype="application/x-www-form-urlencoded">
NOME:
<input type="text" name="nome" value="">
<BR>
COGNOME:
<input type="text" name="cognome" value="">
<BR>
<textarea name="linee" rows=5>
Linea1
Linea2
Linea3
</textarea>
<BR>
<input type="submit" value="ok">
</form>
</body>
</html>
```

La visualizzazione del form da parte del browser **NON** cambia rispetto al metodo GET. Rispetto al metodo get il primo cambiamento riguarda l'URL visualizzato che sarà:
`http://www.dsi.unive.it/~taw/lezione3/testform.cgi`
Mentre la pagina di risposta sarà:

```
argc is 0. argv is .
REQUEST_METHOD = POST
PATH_INFO =
PATH_TRANSLATED =
SCRIPT_NAME = testform.cgi
QUERY_STRING =
CONTENT_TYPE = application/x-www-form-urlencoded
CONTENT_LENGTH = 57
nome=&cognome=&linee=Linea1%0D%0ALinea2%0D%0ALinea3%0D%0A
```

Come si può vedere tutti i campi del form vengono codificati e passati nello standard input dell'applicazione CGI. In questo caso non abbiamo il problema della lunghezza dell'url perché i parametri codificati vengono passati sullo standard input. In ogni caso la codifica dei parametri richiede del tempo di codifica al browser e un'eventuale decodifica da parte dell'applicazione CGI. Queste operazioni di codifice e decodifica posso essere onerose se i dati trasferiti sono molti come nel caso di trasferimento di file.

Esempio Form con metodo POST e codifica multipart

Nel caso di sottomissione di file, il metodo migliore è usare la codifica multipart. In questo caso i dati non vengono codificati, ma i singoli parametri compaiono nello standard input dell'applicazione separati da un'opportuno delimitatore.

```
<html>
<body>
<form method="POST" action="testform.cgi" enctype="multipart/form-data">
NOME:
<input type="text" name="nome" value="">
<br>
COGNOME:
<input type="text" name="cognome" value="">
<br>
File:
<input type="file" name="filename">
<br>
Ultimo campo:
<input type="text" name="end">
<br>
<input type="submit" value="ok">
</form>
</body>
</html>
```

Che produrrà il browser ci visualizzerà nel seguente modo:

NOME:
COGNOME:
File:
Ultimo campo:
ok

Ecco il risultato dell'invocazione dello script CGI come viene visualizzato dal browser:

```
argc is 0. argv is .
REQUEST_METHOD = POST
PATH_INFO =
PATH_TRANSLATED =
SCRIPT_NAME = testform.cgi
QUERY_STRING =
CONTENT_TYPE = multipart/form-data; boundary=---------------------------
24464570528145
CONTENT_LENGTH = 2921
-----------------------------24464570528145
Content-Disposition: form-data; name="nome"
alessandro
-----------------------------24464570528145
Content-Disposition: form-data; name="cognome"
roncato
-----------------------------24464570528145
Content-Disposition: form-data; name="filename"; filename="Gruppi200203.html"
Content-Type: text/html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Laboratorio Ingegneria del Software: Composizione gruppi 2002/03</title>
  <meta http-equiv="content-type"
 content="text/html; charset=ISO-8859-1">
</head>
<body>
<h1>Laboratorio di Ingegneria del Software</h1>
<h2>Composizione gruppi 2002/03</h2>
Gruppo 1: Progetto 1<br>
```

<u>Antonello Mauro</u> 784113
Franchetto Stefano 783094
De Nes Francesco 784105
Galesso Daniele 786466

Gruppo 2: Progetto 2
<u>Rota Bulo' Samuel</u> srotabul@dsi.unive.it
Casagrande Massimo mcasagra@dsi.unive.it
Fornaro Emanuele efornaro@dsi.unive.it
Gerarduzzi Michele mgerardu@dsi.unive.it
Luisetto Andrea aluisett@dsi.unive.it
Niero Luca lniero@dsi.unive.it

Gruppo 3: Progetto 3
<u>Trolese Paolo</u>
Orsenigo Marco
Favaro Susanna
Busolin Katia

Gruppo 4: Progetto 2
<u>Pietro Ferrara</u> 784578
Teresa Scantamburlo 783915
Patrizia Zucconi 783916
Luigi Runfola 784386

Gruppo 5: Progetto 1
<u>Casotto Briano</u> bcasotto@dsi.unive.it 785251
Ravagnan Emiliano eravagna@dsi.unive.it 786353
Marvulli Donatella dmarvull@dsi.unive.it 783738
Ramelintsoa Carole cramelin@dsi.unive.it 786413
Scavazzon Marco mscavazz@dsi.unive.it 786242

Gruppo 6: Progetto 3
Fiori Alessandro afiori@libero.it 777191
Gastaldi Riccardo rigasta@tin.it
Bernacchia Francesco s.checco@libero.it 778611
Pozzobon Giovanni GiovanniP@aton.it 791120
Rampazzo Pietro rampaz79@tin.it
Cian Francesco francesco.cian@t-systems.it 778885

Gruppo 7: Progetto 4
<u>Piccin Massimiliano</u>
Carraretto Alessandro
Ministeri Dario
Rui Massimo

Gruppo 8: Progetto 4
<u>Corradin Michele</u>
 Borin Francesca
 Bordin Fabio
 Scomello Stefano

Gruppo 9: Progetto 5
<u>Carrer Andrea</u> 783089
Longo Irene 783528
Vianello Michele 784026
Molinari Marco 784162

Gruppo 10: Progetto 5
<u>Leonardo Scattola</u> lscattol@dsi.unive.it
Filippo Cervellin fcervell@dsi.unive.it
Roberto Fietta rfietta@dsi.unive.it
Luca Giacomazzi lgiacoma@dsi.unive.it
Lino Possamai lpossmai@dsi.unive.it

</body>

```
</html>
-----------------------------24464570528145
Content-Disposition: form-data; name="end"
valore ultimo campo
-----------------------------24464570528145--
```

Nota: se usiamo il tag file con la codifica url-encoded viene inviato solo il nome del file e non il contenuto.

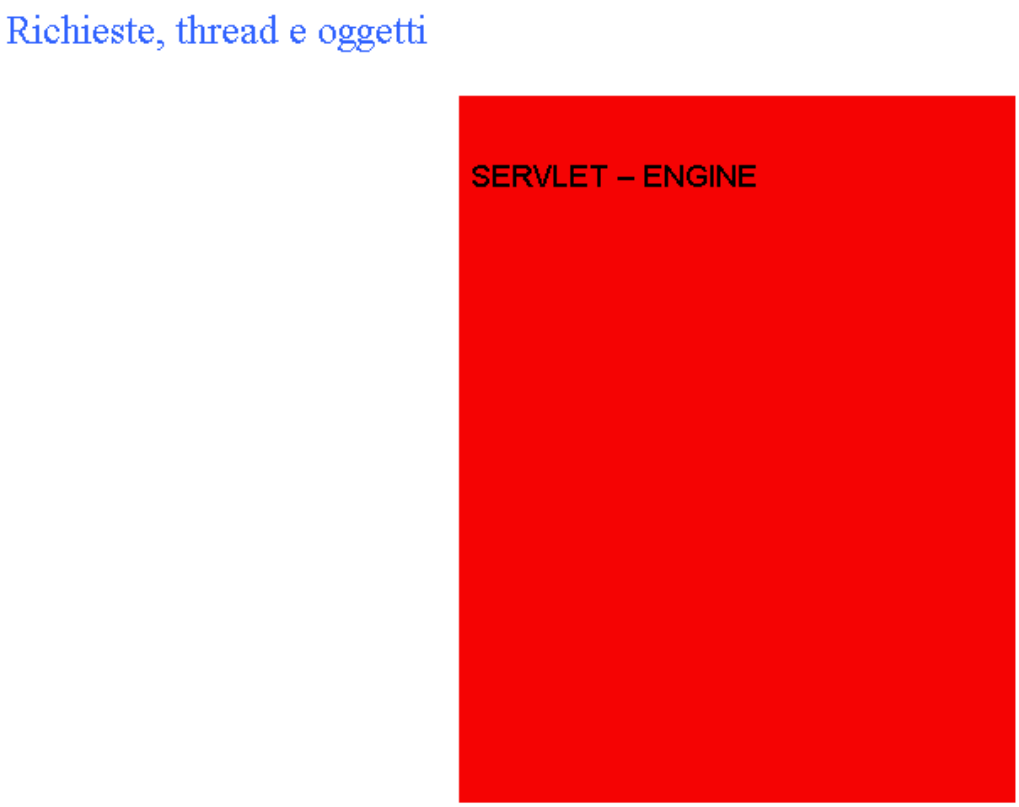
Esercizio:

Negli esempi soprastanti c'è un errore, eseguire gli esempi di sopra e trovare l'errore.

Lezione 7: Approfondimento sulle servlet

1) Richieste, Thread e Oggetti

Abbiamo visto che i CGI creano un processo per ogni richiesta. Lo standard input e l'environment sono usati per inviare informazioni dal Web server al CGI, mentre lo standard output è usato per inviare le informazioni dal CGI al Web server. Dato che ogni processo ha i propri standard input, environtment e standard output il Web server è in grado di comunicare con ogni processo CGI in maniera indipendente. Ma nel caso delle servlet come avviene la comunicazione tra Servlet e Web server? Per ogni nuova richiesta vengono creati due nuovi oggetti di tipo HttpServletRequest e HttpServletResponse. Questi oggetti sono passati come parametri al metodo doGet (o doPost) in un nuovo thread. Come sappiamo i thread condividono tutto nel processo eccetto lo stack ma dato che nello stack vanno anche i parametri dei metodi allora ogni thread avrà la possibilità di comunicare in maniera esclusiva con il motore delle servlet attraverso questi oggetti. Il motore delle servlet a sua volta comunicherà col Web server tramite un apposito protocollo per fare in modo che a client diversi corrispondano oggetti diversi.



Dove trovare documentazione sulle servlet:
http://java.sun.com/products/servlet/index.html
http://java.sun.com/products/servlet/2.2/javadoc/index.html

2) Applicazioni Web e Servlet

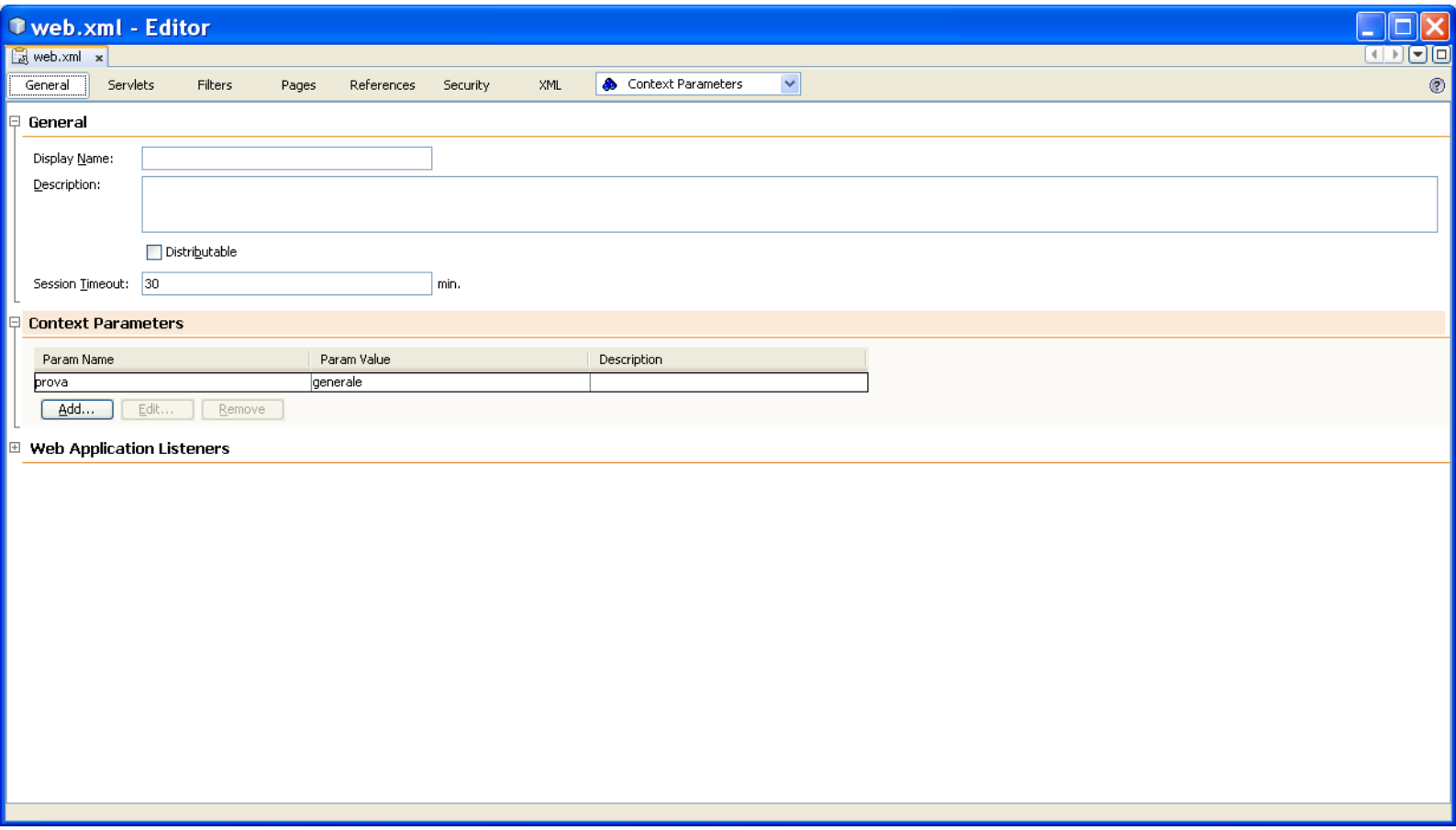
Abbiamo visto come è possibile generare una pagina dinamica con una servlet. In generale, un'applicazione Web è composta da molte pagine. L'API delle servlet ci mette a disposizione il concetto di “Web Application” in cui possiamo gestire in modo uniforme più Servlet (e anche altre risorse come le JSP e risorse statiche come vedremo). Una Web Application è quindi composta da varie risorse che poi vengono raggruppate in un file jar particolare (con estensione war). Un file che contiene le classi, le

eventuali librerie e il file web.xml.

Esigenza comune a molte applicazione è quella di avere la possibiltà di *parametrizzare* il funzionamento dell'applicazione stessa senza dover riscrivere il codice. Questo può essere utile per:

- adattare il funzionamento dell'applicazione piccole modifiche di alcuni fattori;
- personalizzare il funzionamento per i vari clienti/utenti.

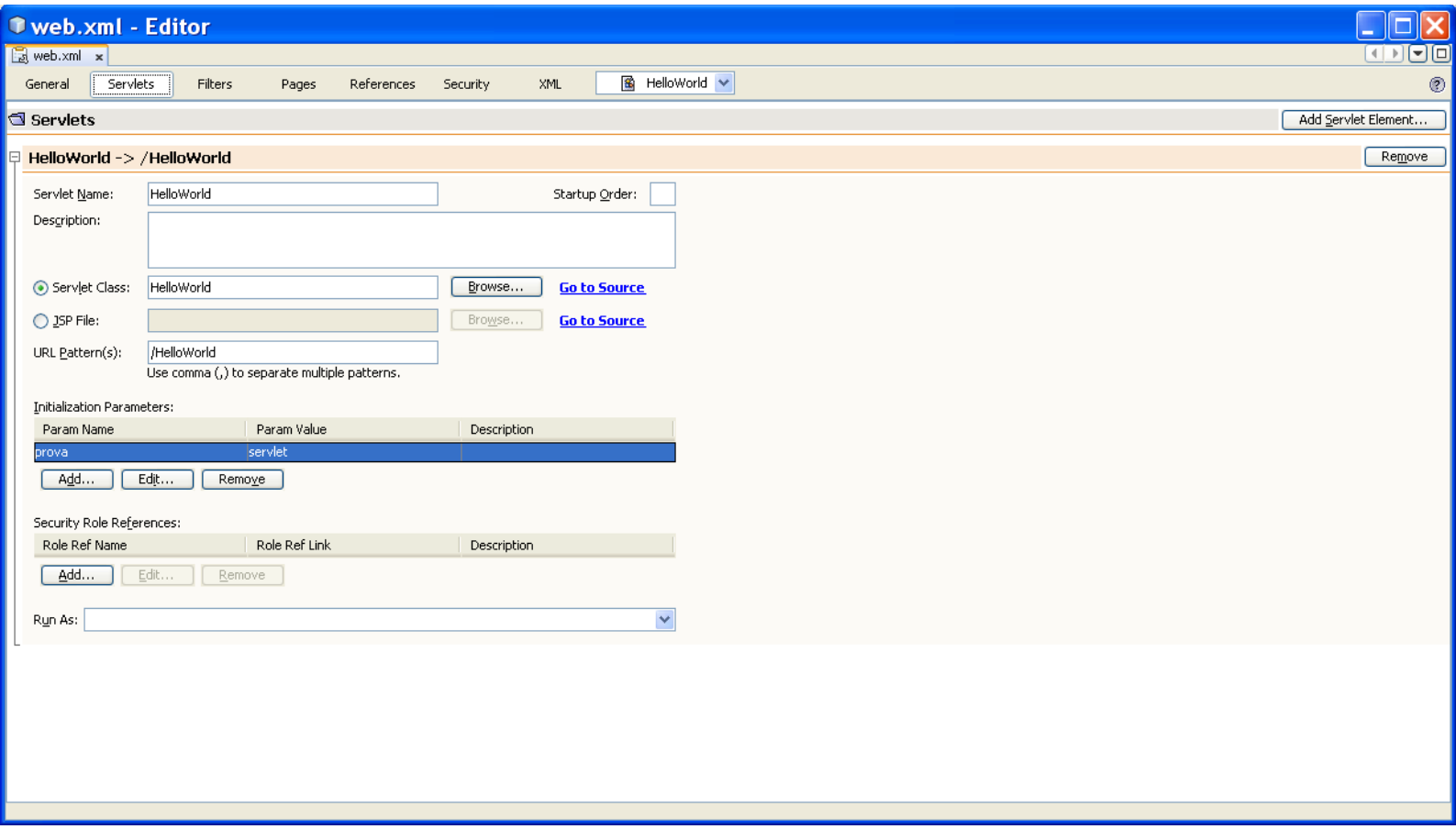
Le servlet mettono a disposizione dei parametri globali per tutta l'applicazione e dei paramentri relativi alla singola servlet.



Abbiamo inserito il parametro con nome “prova” e valore “generale” e questo è relativo a tutta l'applicazione. Quindi ogni servlet lo può recuperare con il seguente metodo:

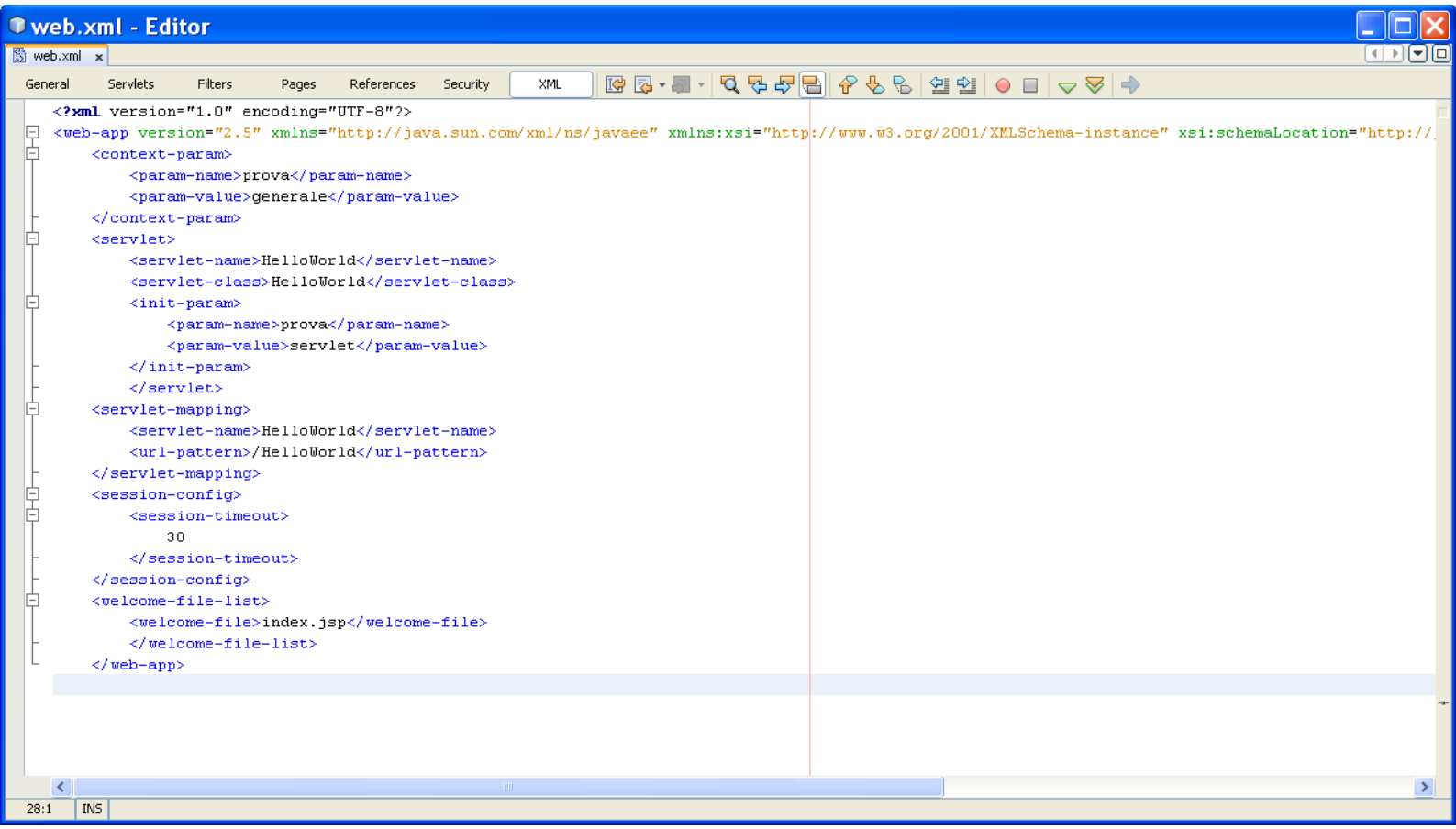
String s=getServletContext().getInitParameter("prova");

Mentre in questo caso abbiamo inserito un parametro sempre con nome “prova” ma con valore “servlet”. Solo la servlet HelloWorld può recuperare questo valore nel seguente modo:



String s=getInitParameter("prova");

Il tool NetBeans scrive per noi il file web.xml che codifica tutto quello che rappresentato visivamente dalle due schermate di sopra.



Altra informazione importante è quella del riguardante il nome della servlet, il nome della classe della servlet e il mapping tra url e la servlet. Se torniamo alla seconda figura, vediamo tre informazioni importanti: "Servlet Name", "Servlet Class" e "URL Pattern(s)". Il primo stabilisce il nome della servlet e serve a individuare l'istanza della servlet. Il secondo definisce il tipo dell'istanza della servlet: una stessa classe può essere usata per creare più servlet diverse e i parametri della servlet possono permetterne la diversificazione già dal momento della istanziazione. L'ultimo stabilisce l'URL (o gli URL) dove è "visibile" la servlet stessa.

3) Ciclo di vita di una servlet

Per razionalizzare e velocizzare il funzionamento di un applicazione web le API delle servlet prevedono il seguente ciclo di vita:

1. creazione (istanziazione) dell'oggetto servlet;
2. invocazione del metodo `init` prima di ogni altro metodo;
3. invocazione di zero, uno o più volte (anche concorrenti) dei metodi doGet e doPost;
4. incocazione del metodo destroy;
5. distruzione dell'oggetto servlet.

In pratica il motore delle servlet garantisce che:

1. il metodo init venga chiamato una sola volta prima dei metodi doGet e doPost;
2. il metodo destroy venga chiamato una sola volta e dopo tutti gli altri metodi.

3.1) metodo `init`

Abbiamo detto che le servlet offrono la possibilità di eseguire una parte di codice una sola volta e di sfruttare il lavoro fatto per rispondere a tutte le successive richieste. In questo modo possiamo ottimizzare i tempi di risposta perché non c'è bisogno di ricalcolare questa parte di codice. Per ottenere questo

dobbiamo scrivere una classe in cui re-implementare il metodo `init` della classe `Servlet`. Il metodo init viene chiamato una sola volta prima della prima invocazione da parte del servlet engine di ogni altro metodo della classe. Ecco un esempio di implementazione di tale metodo:

```
public class Test extends HttpSetvlet{

 int count;

 public void init(ServletConfig config)

 {

   count = 0;

 }

 public void doGet(HttpServletRequest req,HttpServletResponse res)

 {

  res.setContentType("text/html");

  PrintWriter out = res.getWriter();



  out.println("<htlm><body>Times" + count++ +"</html></body>");

  out.close();

}

}
```

Esercizio:

Fare in modo che il valore del contatore non venga riazzerato ad ogni nuova esecuzione del motore delle servlet.

Ad esempio supponiamo di scrivere una servlet per gestire una banca on-line. La servlet stessa potrà essere riusata per banche diverse. Ma come gestiamo quelle piccole differenze tra le varie installazioni della stessa applicazione? Ad esempio, il nome della banca dove andrà messo? La soluzione più semplice è quella di mettere nella classe della servlet una variabile d'istanza con il nome della banca:

```
public class Banca extends HttpServlet {
    String intestazione="Banca Etica";
   ....
}
```

Lo svantaggio di tale soluzione è che per una banca differente, bisogna modificare il codice Java e ricompilarlo. Una soluzione migliore è quella di mettere tutti i paramentri in un file che andrà letto prima di iniziare a gestire le richieste dei client.
Le API delle servlet non solo mettono a disposizione il metodo init che viene invocano nel momento giusto per fare queste operazioni di configurazione, ma fornisce anche un formato di file apposito e delle funzionalità apposite che permettono di accedere al contenuto dei parametri senza bisogno di conoscere come accedere ai file e/o effetturare complesse operazioni di parsing. Basterà infatti accedere a delle semplici proprietà di oggetti Java.

3.2) Parametri delle servlet

Per configurare i parametri di la servlet bisogna modificare il file `web.xml` nella directory `WEB-INF`. Ecco un esempio di tale file:

```
    <servlet>
        <servlet-name>
            BancaEticaOnLine     => nome della servlet
        </servlet-name>
        <servlet-class>
            Banca           =>come si chiama il file .class contenuto in
<docBase>WEB-INF/classes dentro il file .war
        </servlet-class>
        <init-param>       => evenutali parametri di inizializzazione
            <param-name>intestazione</param-name>
            <param-value>Banca Etica</param-value>
            <param-name>tasso</param-name>
            <param-value>3.15</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>      => altri modi di pubblicare la servlet.

        <servlet-name>
            BancaEticaOnLine       => servlet da richiamare
        </servlet-name>
        <url-pattern>
            /BancaEtica            => la servlet BancaEticaOnLineè visibile all'URL
http://<host>:<port>/<WebApp>/BancaEtica
        </url-pattern>
  </servlet-mapping>
```

Esempio

Inoltre possiamo sfruttare il metodo init per accedere a dei parametri di inizializzazione che permettono di adattare il funzionamento della servlet senza bisogno di riscirvere il codice.

```
public class Banca extends HttpServlet{
    String intestazione;
    double tassoDebitore;

public void init(ServletConfig config) throws ServletException
   {
      super.init(config);  // call the init method of base class

      intestazione = config.getInitParameter("intestazione"));
      tassoDebitore =
Double.parse(config.getInitParameter("tasso")));

   }
...
}
```

Ecco ora un esempio di implementazione che visualizza tutti i parametri di inizializzazione passati alla servlet:

```
   public void init(ServletConfig config) throws ServletException
```

```
   {
      Enumeration initParams = null;

      super.init(config);  // call the init method of base class

      System.out.println(config.getServletName());

      initParams = config.getInitParameterNames();
      String paramName = null;
      // Iterate over the names, getting the init parameters
      while ( initParams.hasMoreElements() )
      {
         paramName = (String)initParams.nextElement();
         System.out.println(paramName + ": " +
config.getInitParameter(paramName));
      }
      // initialization

   }
```

nota: i parametri iniziali vengono gestiti in maniera simile ai parametri di una richiesta. Bisogna però fare attenzione al fatto che i parametri iniziali sono relativi alla servlet, mentre i parametri della richiesta sono relativi ad una richiesta fatta dal browser.
nota: nel caso la servlet sia configurata come sopra essa è visibile tramite l'url `http://<host>:<port>/<path> /BancaEtica`.

nota: il motore delle servlet ci garantisce che il metodo `init` viene richiamato una solo volta. Quindi la seguente implementazione della chiamata al metodo init è errata!

```
class MyClass extends HttpServlet{
  boolean started=false;
  public void doGet(HttpRequest request, HttpResponse response)
  {
     if (! started)
     {
       intialize(); //errato! possibile accesso concorrente
       started=true;//ed esecuzione multipla
     }
     ...
  }
}
```

3.3) Metodo destroy

Il metodo destroy è in qualche senso simmetrico a quello init. Nel metodo destroy dobbiamo liberare tutte le risorse in uso da parte della servlet. Inoltre possiamo utilizzarlo per salvare tutte quelle informazioni che ci possono essere utile al successivo riavvio della servlet stessa.

4) Sessioni

Abbiamo già visto che nel protocollo http il server non può distingure se due richieste sono generate dallo stesso utente o da due utenti diversi. Molte applicazioni web hanno l'esigenza di identificare due richieste successive dello stesso utente in modo da semplificare all'utente la fruizione dell'applicazione stessa. Un’oggetto HttpSession serve per simulare una sessione di lavoro da parte di un utente nel protocollo HTTP. Lo scopo fondamentale dell'ogetto è quello di veicolare informazione tra una richiesta e la successiva di uno stesso utente. In Java otteniamo questo tramite il passaggio di riferimenti ad oggetti da una richiesta all'altra attraverso l'oggetto HttpSession.

Il motore delle servlet si occupa di gestire gran parte della complessità di gestione della sessione.

```
public void doGet(HttpServletRequest req, ..)
{
   HttpSession hs = req.getSession(true);
   // prima di inviare l'output
   // se non esiste, viene creato
}
```

Un'oggetto `HttpSession` viene usato per memorizzare le informazioni riguardanti lo "stato" del colloquio tra utente e server.
Gli oggetti vengono "memorizzati" nella sessione usando il seguente schema:

```
HttpSession hs = req.getSession(true);
MiaClasse o = (Classe) hs.getAttribute("oggetto");
if (o==null)
{
     o = new MiaClasse();
     hs.setAttribute("oggetto",o);
}
```

Un oggetto gestito come appena visto avrà:

- la stessa istanza per due richieste con uguale sessione e quindi con lo stesso utente;
- istanze diverse per sessioni diverse;
- le session sono relative all'applicazione web (più servlet che condividono lo stesso contesto).

Le sessioni non sono più **valide** quando:

- vengono invalidate esplicitamente dal programmatore col metodo invalidate; (es. logout)
- passa un certo tempo (configurabile).

altri metodi dell'oggetto HttpSession

public void removeAttribute(String name);
Rimuove l'attributo dalla sessione

public Enumeration getAttributeNames()
Ritorna la lista (Enumeration) di tutti i nomi delgi attributi memorizzati nella sessione;

public long getCreationTime();
Ritorna il tempo della creazione della sessione misurato in secondi trascorsi dalla mezzanotte del 1.1.1970.

public long getLastAccessdTime();
Ritorna il tempo dall'ultima richiesta fatta con sessione misurato in secondi trascorsi dalla mezzanotte del 1.1.1970.

public int getMaxInactiveInterval();
public void setMaxInactiveInterval(int sec);
Imposta e ritorna il tempo massimo di inattività della sessione. Scaduto il tempo senza ulteriori richieste da parte del client, la sessione sarà invalidata dal motore. Un valore negativo indica che la sessione non dovrà mai scadere;

public void invalidate();
Invalida la sessione immediatamente.

Esempio

In questo esempio vediamo quante volte un utente durante la sua sessione di lavoro ha avuto accesso alla servlet.

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

public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
response)
              throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    HttpSession session = request.getSession(true);
    String heading;
    Integer accessCount =
(Integer)session.getAttribute("accessCount");
    if (accessCount == null)
    {
      accessCount = new Integer(0);
      heading = "Welcome Newcomer";
    } else {
      heading = "Welcome Back";
      accessCount = new Integer(accessCount.intValue() + 1);
    }

    session.putAttribute("accessCount", accessCount);


    out.println(("<HTML><HEAD><TITLE>Test HttpSession</TITLE></HEAD>"
+
                "<BODY>\n" +
                "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
                "<H2>Information on Your Session:</H2>\n" +
                "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "  <TH>Info Type<TH>Value\n" +
                "<TR>\n" +"  <TD>ID\n" +"  <TD>" + session.getId() +
"\n" +
                "<TR>\n" +"  <TD>Creation Time\n" +
                "  <TD>" + new Date(session.getCreationTime()) + "\n"
+
                "<TR>\n" +"  <TD>Time of Last Access\n" +
                "  <TD>" +new Date(session.getLastAccessedTime()) +
"\n" +
                "<TR>\n" +"  <TD>Number of Previous Accesses\n" +"
<TD>" +
                accessCount + "\n" + "</TABLE>\n" +"</BODY></HTML>");
  }
}
```

Lezione 8 Cookie e JSP

Cookies

I cookies sono delle informazioni che il server invia al client nella sezione header della risposta http. Il client a seconda del tipo di cookie memorizza sul disco o meno questa informazione. Ad ogni richiesta fatta verso lo stesso server, il client nello header della richiesta ripropone i cookie che precedentemente il server gli ha inviato.
Ci sono dei limiti alla dimensione dei cookie e al numero di cookie che un server può inviare.
I cookie vengono usati dal motore delle servlet per gestire le sessioni e quindi risolvere tutti quei problemi legati alla sessione. Altre applicazione dei cokies sono:

- Personalizzazione di un sito in base alle pagine precedentemente visitate;
- Pubblicità mirata alle pagine precedentemente visitate;
- Eliminazione di username e password per l'accesso al sito;

Altro vantaggio offerto dai cookie è la possibilità di gestire informazioni senza bisogno di memorizzarle sul server stesso, ma in maniera distribuita sui client. Nei cookie si memorizzano informazioni che possono essere ricavate in altri modi, sono di solito una scorciatoia.
I cookie sono rappresentati da un nome e un valore:
Il nome deve rispettare le regole dettate in RFC 2109. Questo significa che esso può contenere solo caratteri ASCII alfanumerici esclusi virgole, punti e virgola e spazi, inoltre non possono iniziare col simbolo '$'.

```
...
Set-Cookie: CFID=145112387;domain=.unive.it;expires=Sun, 22-Feb-2043 08:39:00
GMT;path=/
Set-Cookie: CFTOKEN=38040093;domain=.unive.it;expires=Sun, 22-Feb-2043 08:39:00
GMT;path=/
Set-Cookie: JSESSIONID=703011da1cd525641e60;path=/
Set-Cookie: OP_NEWVISIT=0;path=/
Set-Cookie: PUBLIC_NEWVISIT=0;path=/
...
Cookie: JSESSIONID=703011da1cd525641e60; name2=value2
```

Nelle API Java, il nome del cookie non può essere cambiato dopo la sua creazione. Il valore del cookie non ha vicoli sul contenuto.
I metodi principali degli oggetti della classe javax.servlet.http.Cookie sono:

```
public void setComment(String s);
public String getComment();
```

Impostano e leggono il valore del commento al cookie. Il commento viene visualizzato dal browser nel caso chieda conferma se accettare o meno il cookie;

```
public void setVersion(int c);
public int getVersion ();
```

Impostano e leggono il valore della versione come sotto:
c== 0: Netscape standard
c== 1: RFC 2109

```
public void setMaxAge(int c);
public int getMaxAge();
```

Impostano e leggono il tempo di vita del cookie. Quando il cookie arriverà al browser verrà intrepresa una delle seguenti:
c >0: lo terrà in vita per c secondi;

c==0: elimiare il cookie;
c<0: lo terrà in vita finché dura la sessione del browser.

```
public void setDomain(String d);
public String getDomain();
```
Impostano e leggono il dominio per il quale i browser deve presentare il cookie quando effettua una richiesta. Di default il cookie viene presentato solo al dominio dal quale lo ha ricevuto.

```
public void setPath(int c);
public int getPath();
```
Impostano e leggono il path nel dominio per il quale i browser deve presentare il cookie quando effettua una richiesta.

Codice Esempio:

```
Cookie coo = new Cookie("nome","Alessandro");
userCookie.setMaxAge(60*60*24*365);
response.addCookie(userCookie);
```

Esempio impostazione cookie

Il seguente esempio imposta due cookie: uno che dura per un giorno e uno che dura finché il browser rimarrà aperto.

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

public class TestSet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
response)
              throws ServletException, IOException {


    Cookie cookie = new Cookie("Session-Cookie","Cookie-Value-S");
    response.addCookie(cookie);

    cookie = new Cookie("Persistent-Cookie","Cookie-Value-P");

    cookie.setMaxAge(60*60*24);
    response.addCookie(cookie);

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println (("<HTML><HEAD><TITLE>Impostazione
Cookie</TITLE></HEAD>" +
       "<BODY><H1 >Impostazione Cookie</H1>Prova impostazione 2
cookie</BODY></HTML>");
  }
}
```

Esempio lettura cookie

Il seguente esempio legge tutti i cookie presentati dalla richiesta.

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

```
public class ShowCookies extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
response)
              throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println(("<HTML><HEAD><TITLE>Lettura Cookie</TITLE></HEAD>" +
                "<BODY><H1>Lettura Cookie</H1>" +
                "<TABLE>\n" +
                "<TR>\n" +
                "  <TH>Nome</TH><TH>Valore</TH>");
    Cookie[] cookies = request.getCookies();
    for(int i=0; i<cookies.length; i++)
      out.println("<TR><TD>" + cookie[i].getName() + "</TD><TD>" +
cookie[i].getValue()+"</TD></TR>\n");
    out.println("</TABLE></BODY></HTML>");
  }
}
```

Esempio uso cookie per gesione sessioni

```
String sessionID = makeUniqueString();
Hashtable sessionInfo = new Hashtable();
Hashtable globalTable = getTableStoringSession();
globalTable.put(sessionID,sessionInfo);
Cookie sessionCookie=new Cookie(“SessionID”,sessionID);
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
```

JSP

Le Java Server Page (JSP) sono una alternativa alle Servlet quando la parte di presentazione (codice HTML) è preponderante rispetto al codice JAVA necessario per la generazione della parte dinamica. In un primo momento vedremo le JSP come alternativa alle servlet, successivamente vedremo invece come sfruttare i benefici di entrambi all'interno della stessa richiesta.
Ad esempio la servlet che stampa la data odierna viene più convenientemente
Esempio:

```
<HTML>
<BODY>
<H1>
<%= new java.util.Date()
%>
</H1>
</BODY>
</HTML>
```

Le JSP sono implementate dal motore delle servlet generando automaticamente il codice Java di una servlet che esegue le operazioni necessarie a stampare il codice HTML presente nella pagina JSP. Inoltre include il codice java presente nella pagina JSP nella giusta posizione nella servlet.
Ad ogni richiesta di una pagina JSP il motore delle servlet verifica se la pagina è stata modificata e in questo caso crea e compila la servlet corrispondente.
Il codice Java presente in una pagina JSP si divide in:

- Direttive: `<%@ codice %>`;
- Dichiarazioni: `<%! codice %>`;
- Scriptlets: `<% codice %>`;

- Azioni Standard: `<jsp:azione />`;
- Espressioni: `<%= codice %>`;

Trasformazione da JSP a Servlet:

- Direttive => nel file della servlet;
- Dichiarazioni => attributi e metodi della servlet;
- Scriptlets e azioni standard => codice nel metodo `doGet`;
- Espressioni => argomenti di `out.print` nel metodo `doGet`.

Esempio (puramente indicativo):

JSP file prova.jsp	Servlet file prova.jsp$32342.java

```
<%@ page import="java.util.*"%>



<%! private String htmlencode(String s)
{
   int i;
   ...
}
    MyClass obj = new MyClass();%>




<html><body>
<%= request.getParameter("nome")%>

<%
if("store".equals(request.getParameter("op
")))
      obj.storeDb(request);
%>

</body></html>
```

```
import java.util.*;
public class prova.jsp$32342 extends
HttpServlet
{
   private String htmlencode(String s)
   {
      int i;
      ...
   }

    MyClass obj = new MyClass();
  public void doGet(HttpRequest request,
HttpResponse response)
{
... inizializzazione;
PrintWriter out = request.getWriter();
out.print("<html><body>\n");

out.print(request.getParameter("nome"));
out.print("\n\n");

if("store".equals(request.getParameter("op
")))
        obj.storeDb(request);

out.print("</body></html>\n");
   }
}
```

Lezione 9 – JSP

Esempio:

```
<HTML>
<BODY>
<H1>
<
%=request.getParameter("eta")
%>
</H1>
</BODY>
</HTML>
```

Oggetti Predefiniti

Negli scriptlets e nelle espressioni sono definiti una serie di variabile descritte dalla seguente tabella:

variabile	tipo	Descrizione	Scope
out	Writer	Un oggetto wrapper che scrive nello stream di output.	page
request	HttpServletRequest	La richiesta che ha comportato la chiamata della pagina.	request
response	HttpServletResponse	La risposta alla request.	page
session	HttpSession	La sessione creata per il client che ha richiesto la pagina	session
page	Object	equivalente a `this`	page
application	ApplicationContext	`getSerletConfig().getServletContext()`, area condivisa tra le servlet	application
config	ServletConfig	Equivalente al parametro config del metodo `init` di una servlet.	page
pageContext	PageContext	sorgente degli oggetti, raramente usato	page
exception	Throwable	L'eccezione lanciata dalla pagina che ha generato l'errore.	page solo nella errorPage

Esempi

richiesta.jsp

```
<%@ page errorPage="errorpage.jsp" %>
<html>
  <head>
    <title>Richiesta</title>
```

```
  </head>
  <body>

<b>Hello</b>: "<%=request.getParameter("user"))%>

  </body>
</html>
```

sessione.jsp

```
<%@ page errorPage="errorpage.jsp" %>
<html> <head> <title>Sessione</title> </head>
  <body>
    <%
      Integer count = (Integer)
session.getAttribute("count");
      if ( count == null )
        count = new Integer(0);
      count = new Integer(count.intValue()+1);
      session.setAttribute("count", count);
     %>
      <b>Tu hai visitato il nostro sito <
%=count.toString()%>" volte.</b>
  </body>
</html>
```

Esercizi

1. Collegatevi a subito.it ed effettuate una ricerca. Collegatevi al sito chegiochi.it e fate una ricerca e visionate un oggetto. Ritornate a subito.it e ricaricate l'ulima pagina.
2. provare a introdurre degli errori di compilazione nei file richiesta.jsp e sessione.jsp;
3. provare a introdurre degli errori srun-time nelle pagine di sopra;
4. scrivere una servlet che conta il numero di accessi di uno stesso utente indipendentemente dalla sessione.

Direttive

Le direttive sono usate per passare informazione dalla JSP al contenitore della servlet
La sintassi della direttiva è: `<%@ tipo attributo="valore" %>`
Le principali tipi delle direttive sono:

- `page (vedi sotto);`
- `include` (per l'inclusione statica di risorse al momento della compilazione) ;
- `taglib` (per l'inclusione di librerie di tag vedremo più avanti).

Le direttive di pagina sono indipendenti dalla posizione e sono uniche: un attributo non può essere ridefinito ma solo aggiunto (eccetto **import** che può essere usato più volte). I principali attributi delle direttive di pagina sono:

- `<%@ page language="java" %>`: specifica il tipo di linguaggio usato negli scriptlets del resto della pagina (solo java e per default).
- `<%@ page session="true" %>`: indica se usare o meno le sessioni, per default true;
- `<%@ page import="java.awt.*,java.util.*" %>`: analogo alla direttiva **`import`** di Java (ricorsarsi delle virgolette nel caso della JSP).
- `<%@ page isThreadSafe="false" %>`: indica se il codice contenuto negli scriptlets è thread safe, cioè se può essere eseguito concorrentemente da due thread senza conflitti;

- `<%@ page errorPage="URL" %>`: in caso di errore visualizza la pagina specifica;
- `<%@ page isErrorPage="true" %>`: questa è una pagina di errore ed è quindi definita la variabile `exception` (vedi sotto).

Ecco l'elenco completo degli attributi definibile nella direttiva di pagina (dal documento di specifica delle servler):

attribute	Description
language	Definisce il linguaggio di scripting usato nelle scriptlets, espressioni e definizioni. In JSP 1.2, l'unico valore ammesso è “java”.
extends	Il valore è una classe java completa di package e che sarà la superclasse della classe generata dalla trasformazioni di quest pagina JSP. Da usare con cautela.
import	Descrive le classi o anche interi packages che saranno importati e quindi disponibili nell'ambiente di scripting. Es: import="java.util.*,java.lang.*" Default: import=”java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*”
session	Indica se la sessione viene gestita o meno. Nel caso sia “true”, allora l'oggetto session di tipo javax.servlet.http.HttpSession rappresenta la sessione della richiesta attuale. Se “false” la pagina non gestisce la sessione e l'oggetto session non è disponibile. Default: “true”
buffer	Specifica la presenza e dimensione del buffer per loggetto out. Se “none” non c'é buffering e tutto l'output viene direttamente inviato al client. Altrimenti è possibile specificare un numero con il suffisso “kb” che rappresenta la dimensione in kilobyte minima del buffer. A seconda del valore dell'attributo “autoFlush”, quando il buffer è pieno ne viene inviato il contentuto al client o lanciata un'eccezione. Se non specificato viene predisposto un buffer di almeno 8kb.
autoFlush	Se “true” quando il buffer è pieno il suo contenuto debba essere inviato automaticamente al client oppure se “false” viene lanciata un'eccezione. Default “true”. Nota: non si può usare autoFlush = “false” quando “buffer=none”.
isThreadSafe	Se “false” il motore delle servlet deve accodare eventuali richieste concorrenti in modo da invarne una alla volta alla pagina JSP. Se “true” richieste concorrenti possono essere inviate contemporaneamente. Default “true”. Nota: se “false” il programmatore deve in ogni caso prestare attenzione all'accesso concorrente dei contesti session e application in quanto condivisi con altre Servlet/JSP della Web application.
info	Definisce una stringa di descrizione libera che è verrà poi ritornata dal metodo `Servlet.getServletInfo()` creato nella servlet della pagina JSP.
isErrorPage	Indica che la pagina corrente è una destinazione di un attributo “errorPage” di un'altra pagina JSP.

	Se “true” allora viene definita la variabile “exception” che contiene l'eccezione sollevata dall'altra pagina. Default“false”
errorPage	Definisce l'URL a una risorsa alla quale viene forwardata in caso di eccezione non catturata nel codice della pagina stessa. Se l'URL è quella di una pagina JSP con isErrorPage=”true” allora in questa pagina di errore viene definita la variabile exception che conterrà l'eccezione sollevata. Default: dipende dall'implementazione del motore delle servlet usato. Nota: il passaggio dell'eccezione da pagina che ha sollevato l'eccezione a pagina di errore avviene tramite attributo “javax.servlet.jsp.jspException” dell'oggetto request. Nota: se autoFlush=true e il buffer della pagina che ha sollevato l'eccezione è già stato svuotato, allora il traferimento alla pagina di errore può non funzionare. Qaundo è specificata una pagina di errore nel file web.xml e anche nella pagina JSP, viene scelta quella presente nella pagina JSP.
contentType	Definisce la codifica dei carattere, il tipo di risposta e il MIME type della risorsa creata dalla pagina JSP. Default “text/html”;
pageEncoding	Definisce la codifica della pagina JSP. Se il CHARSET è specificato nel contentType è usato come default o, altrimenti ISO-8859-1.

Dichiarazioni

Le dichiarazioni sono blocchi di codice Java usati per definire variabili e metodi della classe servlet che verrà generata; Le dichiarazioni seguono esattamente la stessa sintassi che hanno in java.

Sintassi: `<%! dichiarazione%>`.
Esempio:

```
<%! String driver="sun.jdbc.odbc.JdbcOdbcDriver";
public String getDriver() {return driver;} %>
```

Java Bean

JavaBeans è un modello di programmazione a componenti per il linguaggio Java. L'obiettivo è quello di ottenere componenti software riusabili ed indipendenti dalla piattaforma (WORA: Write Once, Run Everywhere).
Inoltre tali componenti (beans) possono essere manipolati dai moderni tool di sviluppo visuali e composti insieme per produrre applicazioni.
Ogni classe Java che aderisce a precise convenzioni sulla gestione di proprietà ed eventi può essere un bean (non è necessario estendere una particolare classe).
La classe non deve avere variabili d'istanza pubbliche e deve avere il costruttore di default (ovvero il costruttore senza parametri).
Una **proprietà** è un singolo attributo pubblico. Le proprietà possono essere in lettura/scrittura, sola lettura o sola scrittura,

Una **proprietà** rappresenta un singolo valore e può essere definita da una coppia di metodi `set`/`get`. Il nome della proprietà deriva dal nome di tali metodi. La sola presenza del metodo `get` indica che la proprietà è in sola lettura, la sola presenza del metodo `set` indica che la proprietà è in sola scrittura, mente la presenza di entrami i metodi indica una proprietà in letture e scrittura.
Ad esempio `setX` e `getX` indicano una proprietà X. La presenza di un metodo `isX` indica che X è una proprietà booleana
Esempio:

```
public class MyBean extends Canvas
{
  String myString="Hello";

  public Prova1(){
   setBackground(Color.red);
   setForeground(Color.blue);
  }

  public void setMyString(String
newString){
   myString = newString;
  }

  public String getMyString() {
   return myString;
  }

  public void print(){
   ...
  }
}
```

MyBean ha una proprietà chiamata myString.

Azioni standard

Le azioni standard sono specificate usando la sintassi dei tag XML. Questi tag influenzano il comportamento al run time delle JSP e della risposta inviata indietro al client.
Sintassi: `<jsp:action attributo="valore" />`
oppure
`<jsp:action attributo="valore">`
...
`</jsp:action>`
L'azione specificata (action) può essere:

- `<jsp:useBean .../>`, `<jsp:setProperty .../>` e `<jsp:getProperty .../>`:

Per usare un Bean in una JSP devo prima di tutto definirlo come nell'esempio seguente:
`<jsp:useBean id="bean" class="MyBean" scope="session" />`
L'`id` è il nome che assegno al bean, la `class` è la classe Java del bean, lo `scope` riguarda il campo di esistenza del bean. Altri scope possibili sono application, request, page. Significato degli scope:

application	l'esistenza del bean è legata alla durata dell'applicazione
session	l'esistenza del bean è legata alla durata della sessione

request	l'esistenza del bean è legata alla durata della richiesta

page	l'esistenza del bean è legata alla durata della pagina (this)

Leggere il valore di una proprietà:

```
<jsp:getProperty name="bean" property="myString" />
```

Impostare il valore di una proprietà:

```
<jsp:setProperty name="bean" property="myString" value="alfa" />
```

In maniera analoga possiamo usare ad esempio `setAttribute` della classe HttpRequest per scambiare dei dati tra una Servlet e la pagina JSP.

Le azioni standard non aggiungono niente alle potenzialità di Java ma sono semplicemente una maniera diversa di scrivere le stesse cose, le seguenti righe di codice JSP:

```
<jsp:useBean id="bean" class="MyBean"
scope="session" />
<jsp:getProperty name="bean" property="costo" />
```

sono equivalenti al seguente codice Java:

```
MyBean bean = session.getAttribute(“bean”);

if (bean ==null){
bean=new MyBean(); //costruttore di default
session.setAttribute(“bean”,bean);
}

out.print(bean.getMyString()); //jsp:getProperty
```

Lezione 10 - JDBC

Azioni standard

`<jsp:include page="url" />`: Per includere risorse staticamente oppure dinamicamente al request time.

`<jsp:forward page="url" />`: Per inoltrare al client la pagina specificata nell'url (la jsp che richiama questa azione non deve aver inviato output, altrimenti otteniamo un'eccezione del tipo IllegalStateException;

`<jsp:param name="prezzo" value="34" />`: Per specificare parametri da passare alla pagina inclusa o inoltrata (vedi 2 azioni precedenti);

`<jsp:plugin type="applet" code="Molecule.class" codebase="/html" >`: per includere i tag per la gestione delle Applet (vedremo più avanti).

Esempi:

```
<jsp:forward page="urlSpec">
  <jsp:param name="nome"
value="valore"/>
</jsp:forward>
```

In una servlet possiamo "inoltrare" la generazione della pagina HTML alla pagina JSP con il seguente comando:

```
ServletContext sc =
getServletContext();
RequestDispatcher rd =
sc.getRequestDispatcher(jsp);
rd.forward(req,res);
```

Nella servlet possiamo usare il comando visto prima solo se prima non abbiamo mai scritto sull'oggetto response. Nella pagina JSP è più difficile controllare se è stato scritto qualcosa, per questa ragione il motore delle servlet mantiene un buffer con i caratteri scritti nel out. Finché i caratteri scritti nel buffer non vengono inviati effettivamente al client tramite il socket, è possibile usare jsp:forward. Eventualmente è possibile agire sulla grandezza del buffer con la direttiva “buffer”.

Esempio uso applet:

```
<jsp:plugin type="applet" code="Molecule.class"
codebase="/html" >
  <jsp:params>
    <jsp:param name="molecule"
value="molecules/benzene.mol"/>
  </jsp:params>
  <jsp:fallback>
    <p> unable to start plugin </p>
  </jsp:fallback>
</jsp:plugin>
```

Differenza tra direttiva include e azione include

Tipo	Sintassi	File o Pagina inclusa	Descrizione	Tempo esecuzione
Include Directive	`<%@ include file=... %>`	Statica	Contenuto tradotto dal motore delle servlet	Traduzione
Include Action	`<jsp:include page= />`	Statica o dinamica	Contenuto incluso tale e quale (senza traduzione)	Esecuzione

Esercizi

1. Collegare una servlet e una JSP in modo che (parte del)l'elaborazione venga fatta dalla servlet mentre la creazione della pagina html dalla jsp.

2. Usare la direttiva include per includere l'intestazione di un sito in più pagine jsp;

3. Usare l'azione include per visualizzare nella pagina "`login.jsp`", la pagina "`logged.jsp`" in caso di successo nell'autenticazione, e "`loginError.jsp`" in caso di insuccesso. Per semplicità assumete che tutte le stringhe `user` e `password` vadano bene se e solo se "`user.equals(password)`".

JDBC

Nella maggior parte delle applicazioni reali si ha bisogno di accedere a un database. Per questo motivo è stata introdotta la Java DataBase Connectivity (JDBC) libreria per accedere ai database.

Per poter accedere a differenti tipi di DataBase, JDBC è stata suddivisa in due parti: 1) l'interfaccia standard uguale per tutti i DB e 2) il driver che implementa l'interfaccia che dipende dal DB. Il driver è una libreria di norma contenuta in un file .jar.

Per accedere ad un DB dobbiamo quindi aggiungere al classpath dell'applicazione il driver JDBC opportuno. Per le servlet possiamo mettere il driver nella cartella lib dell'applicazione (e in questo modo ogni applicazione che usa JDBC anche con lo stesso DB dovrà avere la propria copia della libreria) oppure nella cartella lib del motore delle servlet (in questo modo la libreria è condivisa tra tutte le applicazione del motore delle servlet).

Il fatto che il driver sia separato dall'interfaccia permette di poter cambiare il tipo di DB (per esempio passando da Oracle a DB2) senza bisogno di ricompilare l'applicazione ma solamente cambiano il driver.

Dal punto di vista della programmazione per accedere ai dati di un DB con questa libreria dobbiamo fare tre cose:

1. stabilire una **connessione** con la sorgente dei dati;
2. inviare **comandi** di interrogazione e modifica dei dati;
3. elaborare i **risultati**.

Per ognuna delle operazioni precedenti è presente una classe JDBC incaricata della gestione delle operazioni relative alle funzionalità della stessa:

1. `Connection` per collegarsi al DB;
2. `Statement` per eseguire query SQL al "volo" o PreparedStatement per SQL dinamico;
3. ResultSet racchiude il risultato di una query.

Il ResutlSet è sostanzialmente un array bidimensionale: di righe e colonne. Per navigare tra le colonne possiamo usare l'eccesso tramite la posizione della colonna, la colonna con posizione 1 è la prima (attenzione che gli array in java partono da 0 invece che 1) o tramite il nome della colonna se esiste. Mentre per navigare tra le righe possiamo usare uno dei metodi che sposta il cursore di una certa posizione:

- `next()` per passare alla prossima riga (indice ++);
- `previous()` per tornare alla riga precedente(indice--);
- `last()` per passare all'ultima riga(indice=size);
- `first()` per andare alla prima riga(indice=1);
- `absolute(int pos)` per andare alla riga pos(indice=pos);

Attenzione che non tutti i `ResultSet` sono "scrollabili" all'indietro o in maniera assoluta, quando si crea un `ResultSet` ci sono dei parametri da specificare per chiedere esplicitamente di poter navigare in tutte le direzioni.

tabella: amici	a	b	c	
1	23	"Alberto"	11.23	first()
2	43	"Bianca"	12.34	
3	33	"Evelina"	-12.05	
4	11	"Bruno"	3.23	absolute(4)
5	-12	"Nicola"	22.22	
6	17	"Sandro"	56.00	
7	10	"Valerio"	32.12	
8	11	"Matteo"	00.00	
9	12	"Paola"	23.89	last()

Nel seguente esempio vediamo come interagiscono le tre classi in questione:

```
  Connection con =
DriverManager.getConnection("jdbc:myDriver:wombat","myLogin", "myPassword");
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM amici");
  while (rs.next()) //per passare alla prossima riga
  {
    int x = rs.getInt("a"); // o int x = rs.getInt(1);
    String s = rs.getString("b"); //o String s = rs.getString(2);
    float f = rs.getFloat("c"); //o float f = rs.getFloat(3);
  }
  rs.close();
  stmt.close();
  con.close();
```

I comandi sono espressi tramite un comando SQL. Alternativamente agli `Statement` che vengono creati al volo e usati una volta solo, esiste un'altro tipo di comandi, detto `PreparedStatement`.

I `PreparedStatement` sono da preferire quando:

1. abbiamo delle istruzioni SQL con parametri (gestione in automatico da parte del driver JDBC della formattazione da tipi Java a tipi del DB ad esempio le date e le stringhe);
2. velocizzare le query ripetute;
3. sicurezza sulla esatta sintassi della query SQL risultante: evita i problemi di "SQL Injection" (http://en.wikipedia.org/wiki/SQL_injection).

Ecco un esempio d'uso di un prepared statement:

```
PreparedStatement ps t= conn.prepareStatement("SELECT * FROM Tab WHERE
s= ? ");
String nome = req.getParameter("nome");
pst.setString(1,nome);
ResultSet rs = stmt.executeQuery();
while (rs.next())
{
   int x = rs.getInt("a");
}
```

Un esempio completo di Servlet che accede a un DB MicrosoftAccess tramite bridge JDBC ODBC è il seguente:

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

public class Leggi extends HttpServlet {
    /** Processes requests for both HTTP <code>GET</code> and
<code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Leggi</title>");
        out.println("</head>");
        out.println("<body>");

        out.println("<ul>");

try{
          Class driver = Class.forName("sun.jdbc.odbc.JdbcOdbcConnection");
          java.sql.Connection c = java.sql.DriverManager.getConnection("jdbc:odbc:nome");
          java.sql.Statement s = c.createStatement();
          java.sql.ResultSet rs = s.executeQuery("SELECT * FROM dati");

          while (rs.next()){
            out.println("<li>"+rs.getString("descrizione"));
          }
        }
        catch (Exception e )
        {
```

```
            e.printStackTrace();
        }

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

    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
    /** Returns a short description of the servlet.
     */
    public String getServletInfo() {
        return "Short description";
    }

}
```

Esempio codice accesso a un Database

In questo semplice esempio possiamo vedere come accedere al database Microsoft Access di cui abbiamo creato un DNS ("nome" nell'esempio di sopra) che collega il DB vero e proprio all'applicazione Java.
nota: db url, nome utente e password per accedere al database andrebbero inseriti tra i parametri di configurazione della servlet o della web-application per essere letti e impostati nel metodo `init` della servlet stessa in questo modo è possibile cambiare DB, user e password senza bisogno di ricompilare l'applicazione, inoltre il programmatore non viene a conoscenza delle credenziali di accesso al DB.

Un esempio di inserimento dei dati di un DB è il seguente:

```
try{
  Class driver = Class.forName("sun.jdbc.odbc.JdbcOdbcConnection");
  java.sql.Connection c = java.sql.DriverManager.getConnection("jdbc:odbc:nome");
  preparedStatement =
     connection.prepareStatement("INSERT  INTO commenti (login,commento) VALUES (?,?)");

  preparedStatement.setInt(1,request.getParameter("LOGIN"));
  preparedStatement.setString(2,Integer.parseInt(request.getParameter("Commento")));
  int res = preparedStatement.executeUpdate();
  preparedStatement.close();
  connection.close();
  //Inserite  res righe
}
catch (Exception e )
{
```

```
      e.printStackTrace();
}
```

Esempio inserimento dati del Database

In questo esempio aggiungiamo una nuova riga al database tramite il comando SQL INSERT.

Altri comandi SQL molto usati sono DELETE per cancellare una riga e l'UPDATE per modificare i valori di una riga.
Un esempio di modifica di una riga in un DB è il seguente:

```
try{
     Class driver = Class.forName("sun.jdbc.odbc.JdbcOdbcConnection");
     java.sql.Connection c = java.sql.DriverManager.getConnection("jdbc:odbc:nome");
     preparedStatement =
          connection.prepareStatement("UPDATE  commenti SET commento=? WHERE login=?)");

     preparedStatement.setInt(2,request.getParameter("LOGIN"));
     preparedStatement.setString(1,Integer.parseInt(request.getParameter("Commento")));
     int res = preparedStatement.executeUpdate();
     preparedStatement.close();
      connection.close();
     //Modificate  res righe
}
catch (Exception e )
{
      e.printStackTrace();
}
```

Esempio modifica dati del Database

Nell'esempio vengono modificate tutte le righe con un dato login: in particolare viene impostata la colonna commento con il valore del parametro "Commento" della richiesta.

Un esempio di cancellazione di righe da un DB è il seguente:

```
try{
     Class driver = Class.forName("sun.jdbc.odbc.JdbcOdbcConnection");
     java.sql.Connection c = java.sql.DriverManager.getConnection("jdbc:odbc:nome");
     preparedStatement =  connection.prepareStatement("DELETE FROM commenti where login=?"

     preparedStatement.setInt(1,request.getParameter("LOGIN"));
     int res = preparedStatement.executeUpdate();
     preparedStatement.close();
     connection.close();
     //Cancellate  res righe
}
catch (Exception e )
{
      e.printStackTrace();
}
```

Esempio di cancellazione dati del Database

In questo esempio vengono cancellate tutte le righe di un certo login.

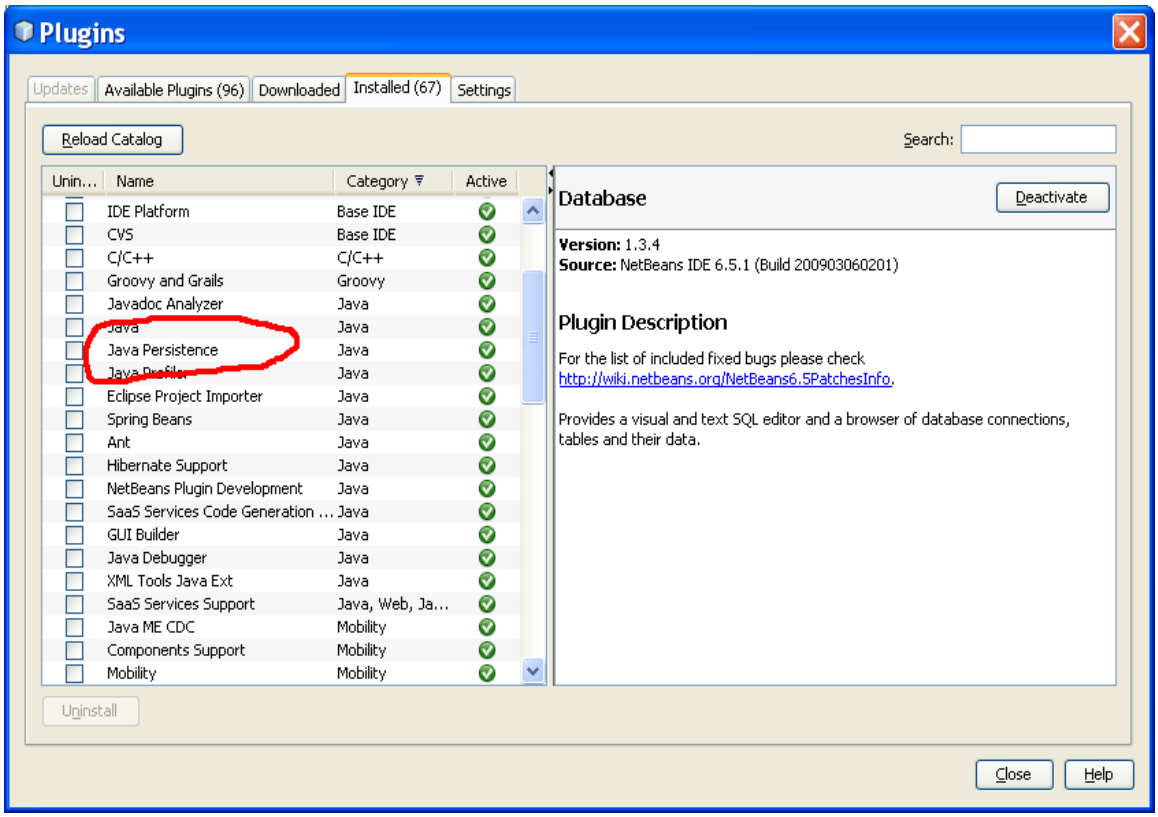
Lezione 11 – Approfondimento JDBC

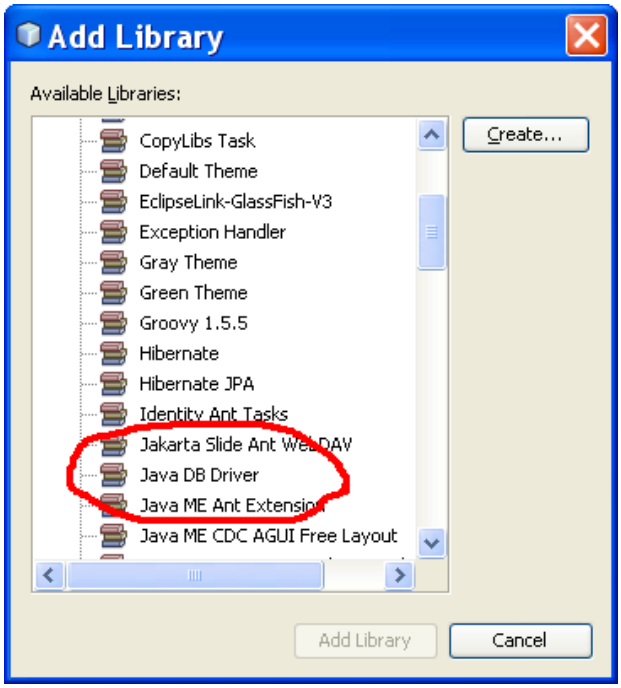
1) Derby

Integrato con Netbeans c'è la possibilità di usare Derby, un database scritto interamente in Java e che nella modalità di utilizzo più semplice non richiede installazione. Tale database può quindi essere usato per lo sviluppo e il test delle applicazioni e successivamente in produzione possiamo scegliere se installarlo nella versione server oppure cambiare DB. Per usare Derby in Netbeans, controllare che sia installato il plug-in “Java-Persistence” e aggiungere la libreria “Java DB Driver”.

Ulteriore semplificazione per l'installazione è la creazione della base dati dall'applicazione web stessa (vedi esempio sotto). In questo modo, alla prima esecuzione su una nuova macchina, il collegamento al database fallisce e nella gestione dell'eccezione si cerca di creare il database stesso usando l'opzione “`create=true`” nell'url del collegamento al database. Un a volta creato il database vuoto possiamo popolarlo tramite Data Definition Language dell'SQL per creare tutte le tabelle necessarie. Eventualmente, sarà anche necessario prevedere l'inserimento di qualche riga di default in qualche tabella (esempio, aggiungere l'utente amministratore con la relativa password di default).

Particolarità della gestione embedded è quella di ricordarsi di “chiudere” il database prima di uscire dall'applicazione. Per chiudere il DB è necessario creare una nuova connessione con l'opzione “`shutdown=true`” nell'url di collegamento al DB. La chiusura del DB non può ritornare una connessione ma lancia una opportuna eccezione con stato “`08006`”.





Esempio con Derby

```
java.sql.Connection con = null;
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

try {
     con = DriverManager.getConnection("jdbc:derby:sample");
    }
    catch (java.sql.SQLException sqle) {
     con = DriverManager.getConnection("jdbc:derby:sample;create=true");
     con.createStatement().executeUpdate("CREATE TABLE PROVA (nome VARCHAR(250), id
INT NOT NULL, cognome VARCHAR(250))");
     //creazione altre tabelle
    }

PreparedStatement s = con.prepareStatement("INSERT INTO PROVA (nome,id,cognome)
VALUES(?,?,?)");

s.setString(1, “Alessandro”);
s.setInt(2, 0);
s.setString(3, “Roncato”);

s.executeUpdate();

ResultSet rs = con.createStatement().executeQuery("select * from PROVA");
while (rs.next()) {
       System.out.println("nome: " + rs.getString(1) + " id:" + rs.getInt(2));
     }
try {//è una versione embedded quindi va chiusa quando l'applicazione termina
     DriverManager.getConnection("jdbc:derby:sample;shutdown=true");
     } catch (SQLException se) {//Stato = 08006 chiusura OK, altrimenti si è
verificato un errore
     if (!se.getSQLState().equals("08006")) {
         throw se;
        }
    }
```

2) Transazioni

JDBC permette di sfruttare una delle caratteristiche più importanti dei Database ovvero la gestione delle transazioni. Tale gestione si effettua attraverso due semplici metodi della classe `Connection`: `commit` e `rollback`. Il significato dei due metodi è quello che ci si aspetta: il metodo `commit` rende definitive

tutte le modifiche apportate usando la connessione fino al precedente `commit` o `rollback`, viceversa il metodo `rollback` le annulla fino al precedente `commit`. All'apertura di una connessione la connessione stessa può rendere definitiva ogni singola modifica senza bisogno di chiamare esplicitamente il metodo `commit` (Auto Commit). Per fare in modo che la semantica dei metodi `commit` e `rollback` sia quella descritta prima dobbiamo togliere la modalitàAuto Commit nella connessione usando il metodo `setAutoCommit(false)`.
E' chiaro che i metodi per la gestione delle transazioni hanno effetto su tutte e sole le operazioni effettuate sul DB tramite la connessione su cui vengono invocati i metodi stessi.
Ecco un semplice schema di utilizzo per implementare una transazione. In questo caso o entrambe le modifiche al database hanno successo (non vengono generate eccezioni), oppure la prima modifica viene annullata dal `rollback`:

```
Connection conn =null;
try{
    conn = DriverManager.getConnection("...");
    conn.setAutoCommit(false);
    Statement st = conn.createStatement();
    st.executeUpdate("DELETE  ...");
    st.executeUpdate("INSERT ...");
    conn.commit();
}
catch (SQLException sqle)
{
    if (conn!=null)
        try{
        conn.rollback();
    }
    catch (SQLException sqle2)
    {
        //log error
    }
}
```

3) Stored procedure (facoltativo)

Le chiamate a stored procedure sono gestite da JDBC in maniera semplice e simile a quanto appena visto, ecco un esempio:
Supponiamo di avere una funzione SQL definita nel modo seguente:

```
FUNCTION procedura (ent_type IN   VARCHAR(20),  num_errori IN   NUMBER
, errore OUT  NUMBER)  RETURN   BOOLEAN IS
```

Il codice java per invocare la funzione appena vista impostando i valori dei parametri di input e recuperare il valore dei parametri di output:

```
Connection conn = null;
CallableStatement cs = null;
ResultSet rs = null;
try {
     conn = getConnection();
     cs = conn.prepareCall("{? = call procedura(?,?,?)}");

     // Registro i parametri di INPUT
     cs.setString(2, parametro1);
     cs.setInt(3, parametro2);
     // Registro i parametri di OUTPUT
     cs.registerOutParameter(1, java.sql.Types.INTEGER);
```

```
     cs.registerOutParameter(4, java.sql.Types.NUMERIC);
     // Esecuzione della stored procedure.
     rs = cs.executeQuery();
     // Note that you need to retrieve the ResultSet _before_
retrieving OUTPUT parameters.
     if ( rs == null) {
            result = false;
      } else {
         rs.next ();
         // Recupero i parametri di OUTPUT
         int intResult = cs.getInt(1);
         BigDecimal errorNumber = cs.getBigDecimal(4);
        }
}
catch (....)
```

Chiamata di store procedure con parametri di input e parametri di output

4) Metadati (facoltativo)

Tramite JDBC è anche possibile conoscere la struttura della query stessa e quindi conoscere il tipo delle colonne ed eventuali altre informazioni.
esempio:

```
ResultSet rs = ...
ResultSetMetaData md = rs.getMetaData();
md.getColumnCount(); //numero totale di colonne;
for (int i=1; i<= md.getColumnCount();i++){
  System.out.print("Name:"+md.getColumnName(i));//il nome della
colonna
  System.out.print("Type:"+md.getColumnType(i));//il tipo della
colonna
  System.out.println("Table:"+md.getTableName(i);//il nome della
tabella da dove proviene la colonna
}
```

E' anche possibile accedere ai metadati della connessione dai quali si possono ricavare informazioni riguardo il database e il driver dedicato al Database stesso. Ecco un esempio che visualizza i nomi delle tabelle utente del DB:

```
     DatabaseMetaData md = con.getMetaData();
     ResultSet rs=md.getTables(null, null, null, new String[]
{"TABLE"});
     while (rs.next()){
        System.out.println( rs.getString("TABLE_NAME"));
     }
```

5) Chiavi primarie generate dal DB

Come spesso accade per ogni tabella è necessario definire una chiave primaria. Ogni record della tabella dovrà avere quindi un diverso valore per la chiave primaria così definita. Se è possibile definire come chiave primaria una delle colonne (o più colonne) di cui si prevede l'inserimento da parte dell'utente, non ci sono in genere problemi di sorta per la gestione della stessa. Nel caso invece che la chiave debba essere generata in automatico dal Database insorgono delle difficoltà nell'uso delle chiavi stessa da parte di applicazioni esterne al DB. Infatti se tra i dati che inseriamo in un record non ce ne sono che ci permettono di individuare in maniera univoca il record stesso, come facciamo ad interrogare il DB per

farci ritornare il record con i dati appena inseriti? Il DB è in grado di creare la chiave primaria in maniera autonoma, ma dall'applicazione non siamo in grado di collegare la chiave primaria ai dati che abbiamo appena inserito.
Vediamo un esempio:
Supponiamo quindi di avere una tabella Ordini in cui memorizziamo tutti gli ordini dei nostri clienti, nella tabella ordini non ci sono chiavi primarie "naturali" e quindi ne definiamo una che sarà creata dal DB stesso (autoincrement?). Questa chiave la chiamiamo IDORDINE. Gli altri dati memorizzati nell'ordine saranno IDCLIENTE e DATA. Ad ogni Ordine sono associate una o più Righe d'ordine che rappresentano gli articoli e le relative quantità che compongono l'ordine. I record della tabella Righe d'ordine devono per forza avere una chiave esterna alla tabella Ordini che per ogni riga d'ordine a che ordine è relativa. Le colonne della tabella Righe saranno quindi: IDORDINE; IDARTICOLO; QUANTITA. Quindi la nostra applicazione Java dovrebbe fare qualcosa del genere:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDCLIENTE,DATA) VALUES (?,?)");
insertOrdine.setInt(1,idCliente);
insertOrdine.setData(2,Data);

res = insertOrdine.executeUpdate();

PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,???);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

La prima soluzione che viene in mente è quella di fare una query per farsi ritornare il valore della chiave appena creata dal DB:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDCLIENTE,DATA) VALUES (?,?)");
insertOrdine.setInt(1,idCliente);
insertOrdine.setData(2,Data);
res = insertOrdine.executeUpdate();

PreparedStatement readIdOrdine = connection.prepareStatement("SELECT
IDORDINE FROM Ordini WHERE IDCLIENTE=? AND DATA=?");

readIdOrdine.setInt(1,idCliente);
readIdOrdine.setData(2,Data);
```

```
ResultSet rs = readIdOrdine.executeQuery();
int idOrdine = -1;
if (rs.next())
idOrdine=rs.getInt(1);

else return false; //qualcosa di meglio
PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,idOrdine);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

Ma cosa succede se il result set trova più di un Ordine con lo stesso IDCLIENTE e DATA? Dato che queste due colonne non sono univoche (altrimenti le avrei scelte come chiave primaria) l'ipotesi di trovare più ordini con lo stesso chiente e stessa data non è da escludere.

5.1) Sequenze

Quindi qual'è una soluzione sicura del problema? Le sequenze sono state pensate per risolvere questo tipo di problemi. Nei DB professionali esistono delle piccole differenze nella gestione delle sequenze ma sostanzialmente il problema viene risolto allo stesso modo. Sfortunatamente Accesse e MySQL non implementano le sequenze, ma solo l'autoincrement.
Le sequenze sono particolari entità del DB che permettono di gestire dei contatori. La sintassi per creare ed utilizzare una sequenza cambia da DB a DB.
La sintassi Postgress:

- creazione: `CREATE SEQUENCE progressivoOrdine START 1;`
- selezione: `SELECT NEXTVAL('progressivoOrdine').`

La sintassi Oracle:

- creazione: `CREATE SEQUENCE progressivoOrdine START WITH 1 INCREMENT BY 1 CACHE 20 ORDER;`
- selezione: `SELECT progressivoOrdine.NEXTVAL FROM DUAL.`

Ecco come dovrebbe essere implementata l'inserimento di un ordine per il Database Postgress:

```
public boolean insertOrdine(int idCliente, Data data, Righe[] righe)
{
connection = DriverManager.getConnection("url", "utente","password");

PreparedStatement readIdOrdine = connection.prepareStatement("SELECT
NEXTVAL('progressivoOrdine')");

ResultSet rs = readIdOrdine.executeQuery();
int idOrdine = -1;
if (rs.next())
idOrdine=rs.getInt(1);
else
```

```
throws new Exception("Error: NEXTVALsequence progressivoOrdine");


PreparedStatement insertOrdine = connection.prepareStatement("INSERT
INTO Ordini (IDORDINE,IDCLIENTE,DATA) VALUES (?,?,?)");
insertOrdine.setInt(1,idOrdine);
insertOrdine.setInt(2,idCliente);
insertOrdine.setData(3,Data);

res = insertOrdine.executeUpdate();

PreparedStatement insertRiga = connection.prepareStatement("INSERT
INTO Righe (IDORDINE,IDARTICOLO,QUANTITA) VALUES (?,?,?)");

for (int i=0; i<Rgihe.length; i++)
{
insertRiga.setInt(1,idOrdine);
insertRiga.setInt(2, righe[j].idArticolo);
insertRiga.setInt(3, righe[j].quantita);
res2 = insertRiga.executeUpdate();
}
insertOrdine.close();
connection.close();
return true;
}
```

Nota: le sequenze possono simulare l'autoincrement di MySQL e Access: Ad esempio creando una tabella Studenti nel seguente modo:
`CREATE TABLE studenti ( idStudente INTEGER PRIMARY KEY DEFAULT NEXTVAL('sequenza'), ...)` dove sequenza è un nome di un'opportuna sequenza.

5.2) getGeneratedKeys

Il problema è molto frequente tanto che le API JDBC stesse prevedono la possibilità di recuperare in automatico la chiave creata. Purtroppo pochi sono i driever disponibili che implementano questa funzionalità. Fortunatamente il database Derby lo permette. Ad esempio:

```
 java.sql.Connection con = null;
 Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

try {
       con = DriverManager.getConnection("jdbc:derby:sample");
    } catch (java.sql.SQLException sqle) {
       con =
DriverManager.getConnection("jdbc:derby:sample;create=true");
          con.createStatement().executeUpdate("CREATE TABLE PROVA
(nome VARCHAR(250),
           id  INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START
WITH 1, INCREMENT BY 1),
           cognome VARCHAR(250))");
     }

PreparedStatement s = con.prepareStatement("INSERT INTO PROVA
(nome,cognome) VALUES(?,?)",Statement.RETURN_GENERATED_KEYS);

s.setString(1, “Alessandro”);
s.setString(2, “Roncato”);
```

```
s.executeUpdate();
ResultSet rs = s.getGeneratedKeys();
if (rs.next()){
     System.out.println("generated id "+rs.getInt(1));
}


ResultSet rs = con.createStatement().executeQuery("select * from
PROVA");
while (rs.next()) {
     System.out.println("nome: " + rs.getString(1) + " id:" +
rs.getInt(2));
       }
try {//embedded quindi chiudiamo il DB quando l'applicazione termina
     DriverManager.getConnection("jdbc:derby:sample;shutdown=true");
     } catch (SQLException se) {//Stato = 08006 chiusura OK, altrimenti
si è verificato un errore
       if (!se.getSQLState().equals("08006")) {
           throw se;
        }
  }
```

6) Perché Class.forName (facoltativo)

La classe DriverManager deve per forza gestire al run-time il collegamento tra url e Driver relativo. Ricordiamo che JDBC permette di cambiare il tipo di DB server (e quindi il relativo Driver) successivamente alla compilazione è chiaro il collegamento tra url e relativo Driver debba essere fatto al run-time. Per questo il DriverManager pubblica il metodo statico registerDriver(Driver d) che permette a un Driver di registrarsi presso il DriverManager. Ma chi chiama questo metodo. E' compito del Driver stesso. Ma come fa il Driver a invocare questo metodo? Basta che nella classse del Driver sia definito il seguente codice:

```
public classs MyDriver implements Driver {

static {

DriverManager.registerDriver(new MyDriver());

}

…//resto del driver

}
```

Quando viene eseguito questo codice (fuori da ogni metodo della classe)? Al momento della prima uso della classe. Dato che il Driver risiede su un file jar esterno all'applicazione, il primo uso lo induciamo con il metodo Class.classForName(“MyDriver”).

Successivamente, il Driver manager farà qualcosa di simile per implementare il getConnection:

```
public class DriverManager {

Set<Driver> drivers = new ListSet<Driver>();

public Connection getConnection(String url) throws … {

for (Driver d: getDrivers()){

if (d.acceptsUrl(url))

return d.connect(String url, Properties info)

}

throw new …;

}

public void registerDriver(Driver d){

driver.add(d);

}

}
```

7) Vantaggi JDBC

- comandi in SQL (standard);
- driver indipendenti dal codice: per usare un DB diverso cambio i driver ma non ricompilo;

Lezione 12 - Extension Tag

Esempi di Tag

Tag senza corpo
```
<jsp:useBean id="bean" class="MyBean" scope="session" />
```

Tag con corpo

```
<jsp:forward page="urlSpec">
<jsp:param name="nome" value="valore"/>
</jsp:forward>
```

```
<mylib:traudci lang="it">
Hello world!
</mylib:traduci>
```

Librerie di Tag

Una libreria di tag è una ulteriore potenzialità che permette agli sviluppatori delle servlet di fornire nuovi tag potenti e facilmente utilizzabili per la realizzazione di pagine JSP. In questo modo è possibile scrivere una pagina JSP anche a chi non conosce il linguaggio di programmazione Java.
Per aggiungere un nuovo tag dobbiamo fornire le seguenti cose:

1. la pagina JSP che utilizza il tag;
2. un file di configurazione che illustra gli aspetti sintattici (ma non solo) del nuovo tag;
3. il codice Java che indica le operazioni da compiere.

Vediamo prima di tutto come si utilizzano i nuovi tag dentro una pagina jsp. Nella prima riga, utilizziamo la direttiva **`taglib`** per indicare dove si trova il file di configurazione che descrive i tag presenti nella pagina. La direttiva ha due attributi: uri indica il file di configurazione (`/esempio`), `prefix` indica il prefisso con cui vengono indicati i tag di questa libreria (`ex`). Una volta dichiarata nella pagina la libreria, possiamo utilizzare tutti i tag che sono definiti nel file di configurazione.

```
<%@ taglib uri="/esempio" prefix="ex" %>
<HTML><HEAD>
<TITLE>Esempio Tag</TITLE></HEAD>
<BODY>
Testo libero.
<h1><ex:mytag /></h1>
Altro Testo
</BODY>
</HTML>
```

Ora vediamo il file di configurazione disponibile all'url "/esempio":
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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>esempio</shortname>
<info>Libreria di tag.</info>
<tag>
<name>mytag</name>
<tagclass>provatag.MyTag</tagclass>
<bodycontent>empty</bodycontent>
<info>Esempio</info>
</tag>
</taglib>
```

Per fare in modo che nell'url "/esempio" si trovi il contenuto di sopra possiamo configurare il file `web.xml` nel seguente modo:

```
...
<taglib>
<taglib-uri>/esempio</taglib-uri>
<taglib-location>/WEB-INF/tlds/esempio.tld</taglib-location>
</taglib>
```

La configurazione nel file web.xml non è necessaria, ma facilita l'installazione. Infatti anche se le librerie risiedono in cartelle diverse da quella data, non è necessario cambiare il codice delle pagine JSP, ma solo il file di configurazione.

Vediamo ora come programmare il gestore del tag. Il metodo `doStartTag` del gestore del tag verrà richiamato dal motore delle servlet in corrispondenza di ogni tag di apertura, `<ex:mytag>` in questo caso, mentre il metodo `doEndTag` verrà richiamato ad ogni tag di chiusura, `</ex:mytag>` nel nostro caso. Nell'esempio che segue, verrà semplicemente inviata alla JSP del testo per indicare quando viene invocato il metodo `doStartTag` e `doEndTag`. In pratica è come se i `<ex:mytag>` venisse sostituito con `"Start tag"`, mentre `</ex:mytag>` con `"End tag"`.

```
package provatag;
import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class MyTag extends TagSupport
{
public int doStartTag() throws JspTagException
{
try
{
pageContext.getOut().write("Start tag");
}
catch (IOException e)
{
throw new JspTagException("Error: write to JSP out");
}
return EVAL_BODY_INCLUDE; // return SKIP_BODY;
}

public int doEndTag() throws JspTagException
{
try
{
pageContext.getOut().write("End tag");
}
catch (IOException e)
{
throw new JspTagException("Error: could not write to JSP out");
}
```

```
return EVAL_PAGE; // return SKIP_PAGE;
}
}
```

file configurazione della libreria

- `<taglib>`: inizio della definizione della libreria, tutti i tag seguenti devono essere nidificati dentro questo tag;
- `<tlibversion>`: versione delle librerire;
- `<jspversion>`: versione delle JSP;
- `<shortname>`: nome della libreria;
- `<info>`: descrizione della libreria

tag

Per ogni tag definito nella libreria ci dovrà essere:

- `<tag>`: inizio della definizione del tag, tutti i tag seguenti devono essere nidificati dentro questo tag.
- `<name>`: il nome del tag, quello che useremo nella pagina JSP dopo i ':';
- `<tagclass>`: la classe Java che definisce il comportamento del tag;
- `<body>`: il tipo di trattamento del testo nel corpo del tag, cioè dell'eventuale testo che si trova tra `<ex:mytag>` e `</ex:mytag>`:
  - `empty`: nessun corpo, eventualmente il corpo viene ignorato;
  - `JSP`: il corpo è valutato prima dal motore delle servlet e poi eventualmente dal gestore del tag;
  - `tagdependent`: il corpo è valutato solo dal gestore del tag, l'eventuale codice JSP (scriplets, esepressioni) **non** viene valutato.
- `<info>`: una descrizione libera sul tag;
- `<teiclass>`: eventuale classe che definisce le eventuali variabili di scripting definite dal tag stesso (vedi esempio sotto);

attributi

Ogni tag definito può avere 0, 1 o più attributi, un attributo ha la stessa sintassi degli attributi html. Es `<ex:mytag attributo="valore">`. Gli attributi possono essere definiti solo sul tag di apertura. Per ogni attributo ci dovrà essere:

- `<attribute>`: inizio della definizione del singolo attributo, tutti i tag seguenti devono essere nidificati dentro questo tag:
- `<name>`: il nome dell'attributo ('attributo' nell'esempio di prima);
- `<required>`: true se l'attributo è obbligatorio, false atrimenti.
- `<rtexpvalue>`: se il valore dell'attributo è statico (deciso al momento della scrittura della pagina JSP o dinamico cioè deciso al momento dell'esecuzione.

Dal punto di vista della classe che gestisce gli attributi il problema si riduce semplicemente all'implementazione di un metodo **`set`** per ogni attributo definito. Di solito in aggiunta al metodo `set` viene implementato anche il metodo `get`. Ad esempio per definire un attributo nome, basterà che la classe implementi il metodo `void setNome(String nome)`. Bisogna prestare attenzione ai tipi: tutti i metodi `set` hanno come parametro una stringa, quindi se l'attributo Java (variabile d'istanza) ha un tipo diverso, la conversione da Stringa al tipo corretto dev'essere fatta nel metodo `set`. Ovviamente nel file JSP gli attributi sono tutti di tipo stringa.

handler dei tag

Il significato del tag viene definito tramite una classe che lo gestisce. A seconda del tipo di funzionalità richieste bisognerà implementare i seguenti metodi della classe `TagSupport`:

funzionalità tag	metodi addizionali da implementare	valore ritornato `doStartTag`
senza corpo	`doStartTag, doEndTag`	`SKIP_BODY`
con attributi	`get/set` attributi	
con corpo	`doStartTag`, `doEndTag`	`EVAL_BODY_INCLUDE`
modifica body	`doInitBody`, `doAfterBody`	`EVAL_BODY_TAG`

body

Vediamo ora un esempio di tag con corpo. Questo tag trasforma il testo del corpo in maiuscolo.

```
<%@ taglib uri="/esempio" prefix="ex" %>
<HTML><HEAD><TITLE>Esempio</TITLE></HEAD>
<BODY>
Testo statico
<ex:captag>corpo del tag</ex:captag>
Altro testo statico
</BODY>
</HTML>
```

Ecco la classe che implementa questo:

```
package provatag;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.util.*;

public class CapTag extends BodyTagSupport {
public int doStartTag() throws JspTagException {
return EVAL_BODY_TAG;
}

public int doAfterBody() throws JspTagException {
BodyContent bodyContent = getBodyContent();
if (bodyContent != null)
{// Do nothing if there was no body content
String output = bodyContent.getString().toUpperCase;
try
{
bodyContent.getEnclosingWriter().write(output);
}
catch (java.io.IOException ex)
{
throw new JspTagException("Fatal IO error");
}
}
return EVAL_PAGE;
```

```
}

public int doEndTag() throws JspTagException {
return EVAL_PAGE;
}
}
```

Tag e motore delle servlet, ovvero ciclo di vita del tag

Per capire a fondo come funzionano i tag bisogna conoscere come interagiscono con il motore delle servlet.

Il motore delle servlet richiama nell'ordine `setPageContext`, `setParent` e i metodi per impostare gli eventuali attributi prima di chiamare il metodo `doStartTag`. Il motore delle servlet ci garantisce anche che il metodo release verrà richiamato prima della chiusura della pagina.
Una tipica sequenza di invocazione è la sequente:

```
SomeTag tag = new someTag();
tag.setPageContext(...);
tag.setParent(...);//vedi tag nidificati
tag.setAttribute1(value1);
tag.setAttribute2(value2);
tag.doStartTag();
tag.doEndTag();
tag.release();
```

Il metodo release del gestore del tag dovrebbe resettare il proprio stato e liberare ogni eventuale risorsa usata. Di solito non è necessario ridefinire questo metodo (vedi esempi precedenti).

Se un tag è derivato da `BodyTagSupport`, ci sono dei metodi aggiuntivi: `setBodyContent`, `doInitBody` e `doAfterBody`. Questi metodi ci permettono di accedere al corpo del tag che è composto da tutto quello che si trova tra lo start tag e l'end tag.

L'oggetto `BodyContent` è una sottoclasse di `JspWriter`, che è l'oggetto usato internamente dalla variabile `out` delle JSP. L'oggetto `BodyContent` è disponibile attraverso la variabile `bodyContent` nei metodi `doInitBody`, `doAfterBody` e `doEndTag`. Questo è importante perché questo oggetto contiene i metodi che possono essere usati per leggere, scrivere cancellare e recuperare il contenuto e incorporarlo nel `JspWriter` originele durante il metodo `doEndTag`.

Il metodo `setBodyContent` crea un contenuto per il corpo e aggiunge al gestore del tag.Il metodo `doInitBody` è chiamato una sola volta prima della valutazione del corpo del tag è viene usato di solito per l'inizializzazione che dipende dal contenuto del tag stesso. Il metodo `doAfterBody` è chiamato dopo che il corpo del tag è stato valutato: se `doAfterBody` ritorna `EVAL_BODY_TAG`, il corpo e valutato di nuovo, altrimenti se viene ritornato `SKIP_BODY` la valutazione del corpo del tag è terminata.

Una tipica sequenza di chiamate fatte dal motore delle Servlet ai metodi di `BodyTagSupport` potrebbe essere la seguente:

```
tag.doStartTag();
out = pageContext.pushBody();
tag.setBodyContent(out);
// perform any initialization needed after body content is set
tag.doInitBody();
tag.doAfterBody();
// while doAfterBody returns EVAL_BODY_TAG we
// iterate the body evaluation
...
```

```
tag.doAfterBody();
tag.doEndTag();
tag.pageContext.popBody();
tag.release();
```

tag con corpo

Avere un tag che valuta il corpo è una possibilità molto interessante. Tra le varie possibilità offerte c'è anche quella di iterare sul corpo stesso: questo ci permette in maniera semplice di manipolare tutte quelle strutture dati che sono enumerazioni: enumeration, array, ResultSet, Collection etc.
Per iniziare a usare il corpo in un tag, dobbiamo ricordarci di fare due cose:

1) impostare il `<bodyContent>` del tag a JSP o tagdependent. Per esempio

```
<tag>
   ...
   <bodycontent>JSP|tagdependent</bodycontent>
</tag>
```

2) derivare il gestore del tag dalla classe `BodyTagSupport`, che è la classe di supporto per l'interfaccia `BodyTag`. In teoria non servirebbe implementare nessun altro metodo dell'interfaccia in quanto già presenti nella classe base. In questo modo si ottiene un tag che lascia inalterato il corpo del tag stesso.
Il modo di implementare il gestore del tag dipende da come il gestore ha bisogno di interagire col corpo. Interagire significa che il gestore legge o modifica il contenuto del corpo oppure produce più valutazioni del corpo stesso.

Se il gestore del tag interagisce con il corpo, il valore di ritorno del metodo `doStartTag` vine interpretato nel seguente modo:

- `SKIP_BODY`: il motore delle servlet non valuta il testo del corpo;
- `EVAL_BODY_TAG`: questo valore può essere ritornato solo dalla classe `BodyTagSupport`. (Se il gestore non ha bisogno di interagire con il corpo dovrebbe implementare solo l'intergaccia Tag o derivare da `TagSupport`).;
- `EVAL_BODY_INCLUDE`; questo valore può essere ritornato solo dalla classe `TagSupport`.

Esempio:

```
<tag>
  <name>loop</name>
  <tagclass>provatag.LoopTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>enum</name>
    <required>true</required>
    <rtexpvalue>true</rtexpvalue>
  </attribute>
</tag>
```

Nel seguente esempio vediamo il codice del gestore del tag che visualizza tutti i parametri di una richiesta HTTP: `import java.util.Enumeration;`

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;

public class LoopTag extend BodyTagSupport
```

```
{
  java.util.StringTokenizer enum =null

  public void setEnum(String enum)
  {
    this.enum = new java.util.StringTokenizer(enum,",;");
  }
  public int doStartTag() throws JspException
  {
     if (enum != null && enum.hasMoreElements())
     {
        pageContext.setAttribute("name", enum.nextToken());
        return EVAL_BODY_TAG;
     }
     return SKIP_BODY;
   }
   public void doAfterBody() throws JspException
   {
     if (enum!=null && enum.hasMoreElements())
     {
        pageContext.setAttribute("name", enum.nextToken());
        return EVAL_BODY_TAG;
     }
     return SKIP_BODY;
   }
   public int doEndTag() throws JspException
  {
     try
     {
      pageContext.getOut().print(bodyContent.getString());
      return EVAL_PAGE;
     }
     catch (IOException ioe)
     {
        throw new JspException(ioe.getMessage());
     }
   }

}
```

Una pagina JSP che usa il tag appena creato potrebbe essere:

```
<%@ taglib uri="/esempio" prefix="ex" %>
<html>
<body>
<h1>Esempio loop</h1>
<table border="2">
<tr>
 <th>Nome</th>
 <th>Valore</th>
 <ex:loop enum="name1,name2,name3" >
<tr>
  <td><%= pageContext.getAttribute("name") %></td>
  <td>
<%= request.getParamter((String)pageContext.getAttribute("name"))
%></td>
</tr>
```

```
</ex:loop>
</table>
</body>
<html>
```

Lezione 13 - Model View Controler

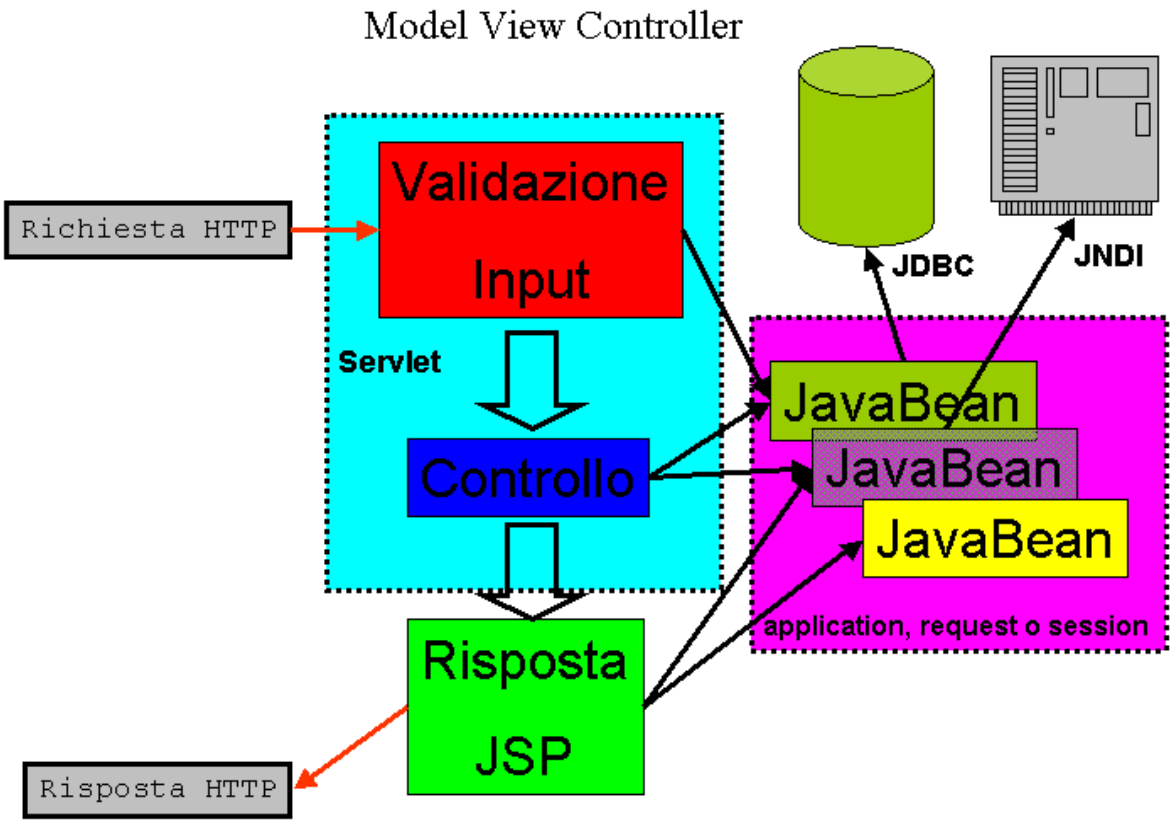
Abbiamo visto che conviene usare le pagine JSP per tutte quelle operazioni in cui la parte di visualizzazione è predominante rispetto a quella di "calcolo"
Le JSP possono essere impiegate per rendere più semplice lo sviluppo e la manutenzione della applicazione nei seguenti scenari:

- Solo JSP: per applicazioni molto semplici, con poco codice Java, che viene gestito con semplici scriptlet e espressioni;
- Utilizzo di Bean e JSP: per applicazioni moderatamente complesse; i bean nascondono la parte di codice più complessa a chi sviluppa la pagina JSP;

Ma questi due semplici scenari non possono essere adottati in tutte le situazioni. Infatti:

- se le applicazioni sono molto complesse, la singola JSP può risultare troppo complicata: troppo codice;
- nonostante sia facile separare il codice in bean, una singola JSP risponde a una singola richiesta: troppe JSP.

Per unire i benefici delle Servlet e JSP e separazione tra controllo (logica) e presentazione (aspetto) si può utilizzare l'archietettura Model-View-Controller schematizzata nella seguente figura:



Integrare Servlet e JSP:

1. La richiesta originale è processata dalla una servlet che esegue la validazione dei dati della richiesta e impartisce gli "ordini" ai bean;
2. I bean conservano le informazioni per il successivo uso da parte della seguente pagina JSP o dell'applicazione stessa;
3. La richiesta è reindirizzata a una pagina JSP per visualizzare il risultato. Inoltre possono essere usate JSP differenti in risposta alla stessa servlet per ottenere presentazioni differenti.

Questo modello è chiamato "MVC (Model View Controller" o “Approccio JSP Model 2").
Dato che nella servlet non viene generato l'output, alla fine della fase di controllo della servlet stessa bisogna "inoltrare" la generazione della pagina HTML alla pagina JSP con le seguenti istruzioni:

```
ServletContext sc = getServletContext();

RequestDispatcher rd =sc.getRequestDispatcher(jsp);

rd.forward(req,res);
```

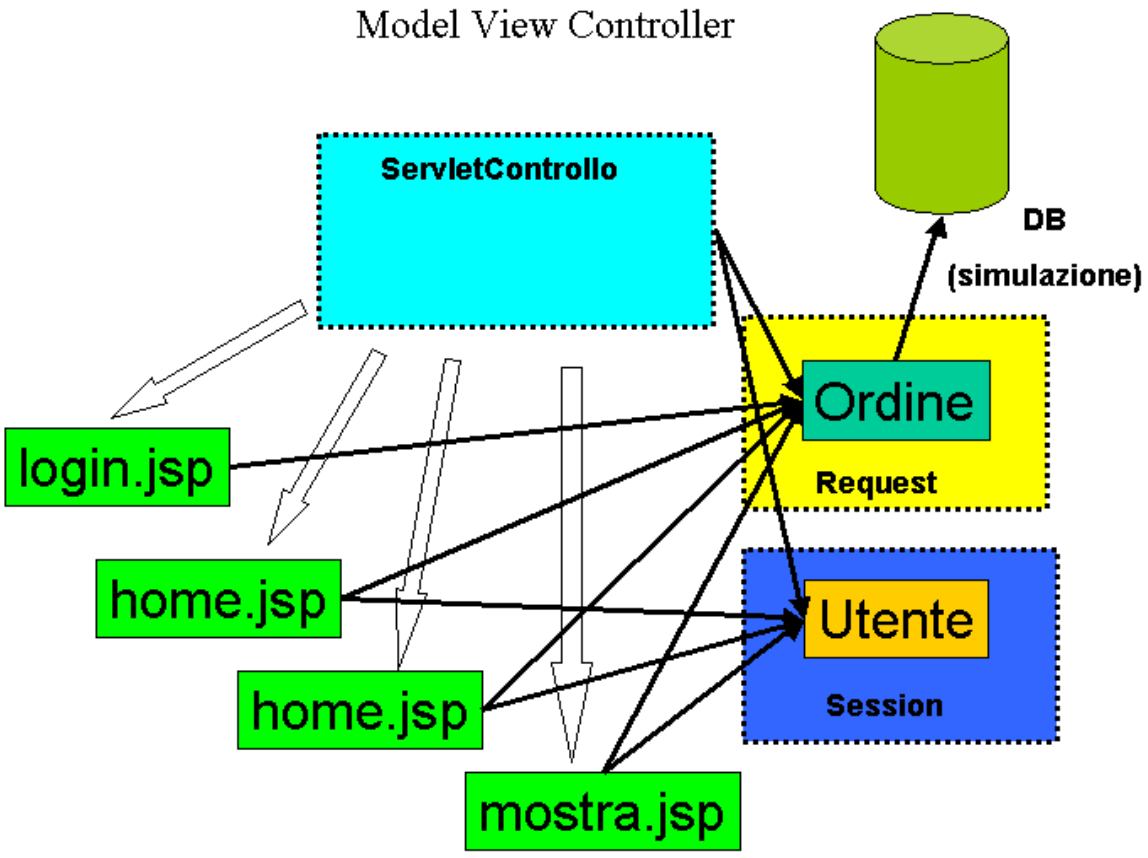
Le istruzioni di sopra possiano essere racchiuse in un metodo della servlet come il seguente:

```
private void forward(HttpServletRequest request, HttpServletResponse
response, String page)
{
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher(page);
rd.forward(request,response);
}
```

Esempio

Nell'esempio vediamo un possibile bean e come viene utilizzato da una servlet di controllo e dalle varie JSP responsabili della visualizzazione delle diverse risposte.

In questo esempio assumiamo che ci sia una sola servlet che effettua tutte le operazioni della nostra applicazione Web. La servlet stessa individua la particolare operazione da fare grazie ad un parametro (nascosto) che si aspetta di trovare nelle richieste. Tale parametro nascosto è chiamato “op”. La servlet esegue l'operazione e a seconda dell'operazione e dell'esito dell'operazione stessa inoltra la visualizzazione alla pagina opportuna.



Nota: nell'esempio di sotto l'accesso al DB non viene implementato. Gli oggetti risiedono solo in memoria RAM e vengono memorizzati un una HashMap. La chiave primaria degli oggetti si suppone venga fornita dall'utente dell'applicazione Web (ipotesi non realistica).

```
public class Ordine
{
   static java.util.Map memory = new java.util.HashMap();
   int progressivo=-1;
   String descrizione=null;

   public int getProgressivo()
   {
      return progressivo;
   }

   public void setProgressivo(int progressivo)
   {
      this.progressivo=progressivo;
   }

   public String getDescrizione()
   {
      return descrizione;
   }

   public void setDescrizione(String descrizione)
   {
      this.descrizione=descrizione;
   }
   public void insert()
   {
     //... sostituire con accesso DB
     memory.put(new Integer(progressivo),this);
   }
   public void update()
   {
     //... sostituire con accesso DB
     memory.put(new Integer(progressivo),this);
   }
   static public Ordine getOrdine(int id)
   {
     //... sostituire con accesso DB
     return (Ordine) memory.get(new Integer(progressivo));
   }
   static public java.util.Set getOrdini()
   {
     //... sostituire con accesso DB
      return memory.entrySet();
   }
}
```

File Ordine.java

```
public Servlet extends HttpServlet{

  public void init(...){
    super.init(...);
    // creazione e inizializzazione oggetti con scope application
  }
  private void forward(... request, ... response, ... page).../vedi sopra
  public void doGet/doPost(... request, ... response){
    ...

    String op = request.getParameter("op");
    HttpSession session = request.getSession(true);
    Utente u = (Utente) session.getAttribute("user");
    if ((u==null || op==null) && !“login”.equals(op)){
       forward(request,response,"/login.jsp");
       return;
    }
    if ("login".equals(op) )
    {
       u = new Utente(request.getParameter("account"));
       if (u==null || !u.checkPassword(request.getParameter("password"))
         forward(request,response,"/login.jsp");
       else
       {
         session.setAttribute("user",u);
         forward(request,response,"/home.jsp");
       }
       return;
     }

     if ("inserimento".equals(op))
     {
        Ordine nuovo = new Ordine();
        try
        {
          nuovo.setProgressivo(Integer.parse(request.getParameter("progressivo")));
        }
        catch (Exception e)
        {
          forward(request,response,"/inputError.jsp");
          return;
        }
        nuovo.setDescrione(request.getParameter("descrizione"));
        nuovo.insert();
        forward(request,response,"/home.jsp");
        return;
     }
    else if ("moduloInserimento".equals(op))
     {
        forward(request,response,"/moduloInserimento.jsp");
        return;
     }
     else if ("mostra".equals(op))
     {
        try
        {
          ordine=Ordine.load(Integer.parse(request.getParameter("progressivo")));
        }
        catch (Exception e)
        {
          forward(request,response,"/inputError.jsp");
          return;
        }
        request.setAttribute("ordine",ordine);
        forward(request,response,"/mostra.jsp");
        return;
    }
```

```
    else ...
  }
}
```

file Servlet.java

Pagine JSP

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Ordini</title>
</head>
<body>
<h2>Login</h2>
<form action="<%=application.getContextPath()%>/Servlet" method="POST">
<input type=hidden name="op" value="login">
<table>
<tr><td>User:</td><td><input name="account" type="text"></td></tr>
<tr><td>Password:</td><td><input name="password" type="password"></td></tr>
<tr><td> </td><td><input type="submit" value="OK"></td></tr>
</table>
</form>
</body>
</html>
```

file login.jsp

```
...
<% Utente user = (Utente) session.getAttribute("user");%>
   if (user==null){%>
<jsp:forward page="/login.jsp" />
<%}%>
<html>
<head>
<title>Home page</title>
</head>
<body>
<h2>Home</h2>
<ul>
<li><a href="<%=application.getContextPath()+"/Servlet"%>
?op=moduloInserimento">Inserisci nuovo ordine</a>
<li><a href="<%=application.getContextPath()+"/Servlet"%>
?op=visualizza">Visualizza Ordini</a>
<li>...
</ul>
</body>
</html>
```

file home.jsp

Invece di controllare in ogni servlet se l'utente e' presente in sessione, basta salvare le pagine jsp nella cartella WEB-INF. Tutto il contenuto di tale cartella non e' accessibile all'esterno del motore delle servlet. L'unico modi di raggiungerle e' tramite forward da una Servlet o da un Jsp visibile. Nelle rimanenti pagine non riportiamo piu' tale controlli.

```
...
```

```
<html>
<head>
<title>Ordini</title>
</head>
<body>
<h2>Inserimento Ordine</h2>
<form action="<%=application.getContextPath()%>/Servlet" method="POST">
<input type=hidden name="op" value="inserimento">
<table>
<tr><td>Progressivo:</td><td><input name="progressivo" type="text"></td></tr>
<tr><td>Descrizione:</td><td><input name="descrizione" type="text"></td></tr>
<tr><td> </td><td><input type="submit" value="OK"></td></tr>
</table>
</form>
</body>
</html>
```

file moduloInserimento.jsp

```
...
<%@ page import="package.Ordine"%>
<% Ordine ordine = (Ordine) request.getAttribute("ordine");%>
<html>
<head>
<title>Ordini</title>
</head>
<body>
<h2>Visualizza Ordine</h2>
<table>
<tr><td>Progressivo:</td><td><%=""+ordine.getProgressivo()%></td></tr>
<tr><td>Descrizione:</td><td><%=ordine.getDescrizione()></td></tr>
</table>
<a href="<%=application.getContextPath()%>/Servlet"?op=modifica&progressivo=<%=""+ordine.g
</body>
</html>
```

file mostra.jsp

```
...
<jsp:useBean id="ordine" class="package.Ordine" scope="request" />
<html>
<head>
<title>Ordini</title>
</head>
<body>
<h2>Visualizza Ordine</h2>
<table>
<tr><td>Progressivo:</td><td><jsp:getProperty name="ordine" property="progressivo" /></td>
</tr>
<tr><td>Descrizione:</td><td><jsp:getProperty name="ordine" property="descrizione" /></td>
</tr>
</table>
<a href="<%=application.getContextPath()%>/Servlet?op=modifica&progressivo=
<jsp:getProperty name="ordine" property="progressivo"/>">Modifica questo ordine</a>
</body>
</html>
```

file alternativo mostra.jsp

```
...
```

```
<%@ page import="package.Ordine"%>
<% java.util.Iterator iterator = Ordine.getOrdini().iterator();
   Ordine ordine =null; %>
<html>
<head>
<title>Ordini</title>
</head>
<body>
<h2>Visualizza Ordini</h2>
<table>
<tr><td>Progressivo</td><td>Descrizione</td></tr>
<% while(iterator.hasNext())
{
  ordine = (Ordine) iterator.next();%>
<tr><td><%=""+ordine.getProgressivo()%></td><td><%=ordine.getDescrizione()></td></tr>
<%}%>
</table>
<a href="<%=application.getContextPath()%>/Servlet?op=moduloInserimento">
Inserisci ordine</a>
</body>
</html>
```

file mostraOrdini.jsp

Regole generali:

- le pagine JSP dovrebbero ridurre al minimo l'utilizzo dei metodi che modificano gli stati del bean;
- la servlet dovrebbe prima controllare l'input, gli utenti e gli accessi e poi effettuare gli accessi in lettura e scrittura ai bean;

- Se le pagine JSP sono contenute entro la direttori “WEB-INF” dell'applicazione, il motore delle servlet non le rende disponibili all'utente tramite accesso diretto (ovvero digitando l'url sul browser). Saranno accessibili solo indirettamente tramite il forward della servlet o di altra pagina JSP. Quindi di norma vanno messe sotto la directory “WEB-INF” tutte le pagine JSP che richiedono autenticazione.

Vantaggi:

1. JSP molto semplici, con le librerie di tag possono essere scritte senza codice Java (quindi da un esperto di codice HTML senza esperienza di programmazione);
2. la possibilità che una stessa servlet o pagina JSP esegua il forward su differenti pagine JSP semplifica enormemente le JSP che in questo modo possono non contenere costrutti di controllo;
3. il controllo è centralizzato nella servlet, posso riutilizzare il codice Java. Per la stessa ragione è preferibile ridurre al minimo il codice Java presente nella JSP o spostandolo nel controllo oppure definenfo un nuovo tag;
4. riuso del modello in contesti applicativi diversi. I Bean di per se stessi non sono legati ne al protocollo HTTP, ne al codice HTML. Quindi possono essere riusati in applicazioni differenti (ad esempio un applicazione non Web);

Esercizi:

1. Completare l'esempio visto sopra implementando la persistenza nel DB Derby;
2. Definire un tag che permetta di iterare sugli elementi di un insieme tramite un iteratore.

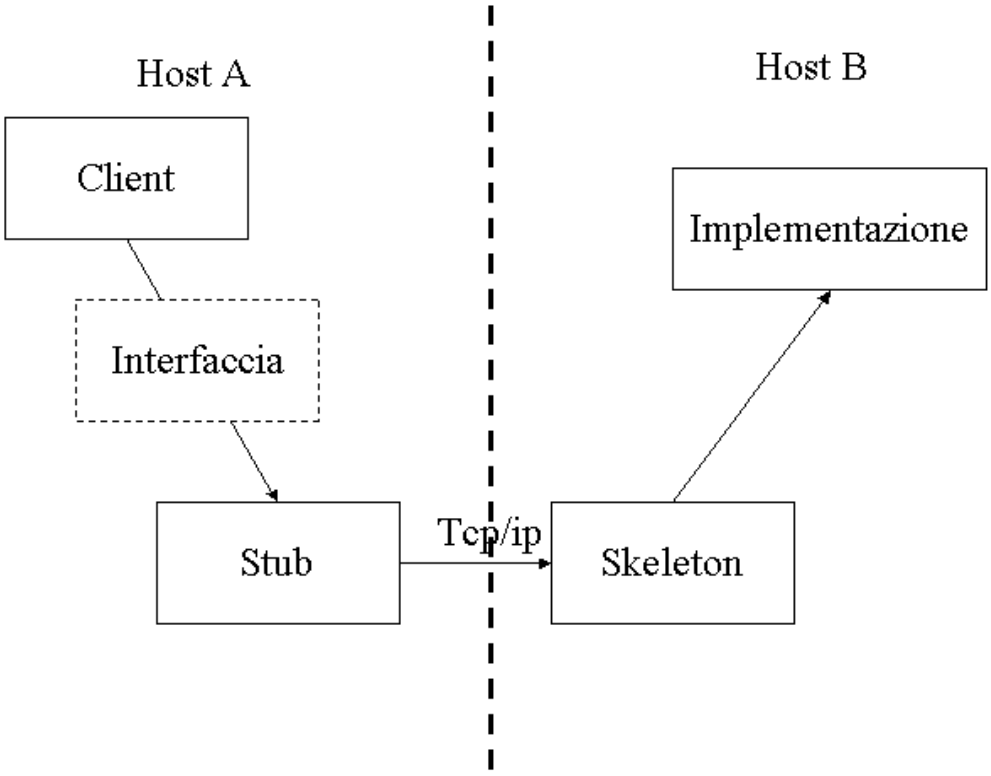
Lezione 14 - RMI

In questa lezioni vediamo una nuova API Java che permette lo scambio di messaggi (invocazione di metodi) tra due host. L'applicazione può essere una servlet, una applet o un'applicazione stand-alone.

RMI

La Remote Method Invocation è un'estensione del RPC (Remote Procedure Call). L'RPC permette a un processo che gira su una macchina A di invocare una procedura che gira su una macchina B. Allo stesso modo RMI nell'ambito della programmazione a oggetti, permette l'invio di un messaggio (o invocazione di un metodo) da una macchina A a un macchina B. Come al solito esiste un'opportuna API per semplificare il compito del programmatore che deve scrivere del codice per inviare i metodi.

In pratica, sia client che server possono quasi ignorare il fatto di risiedere su due macchine diverse.



Il minimo necessario per poter usare RMI è:

1) definire un'interfaccia con i metodi che devono essere invocati remotamente;
2) l'iterfaccia deve estendere l'interfaccia java.rmi.Remote;
3) ogni metodo dell'interfaccia deve dichiarare throws java.rmi.RemoteException;
4) bisogna implementare l'iterfaccia nell'oggetto remoto;
5) l'oggetto remoto deve avere un costruttore;
6) l'oggetto remoto deve implementare i metodi dell'interfaccia;
7) nel server bisogna creare e installare un security manager; //non serve sempre
8) nel server creare almeno un istanza dell'oggetto remoto;
9) nel server registrare uno degli oggetti remoti nel registro degli oggetti remoti;
10) nel client recuperare il riferimento all'oggetto remoto;

11) nel client invocare il metodo nell'oggetto remoto;
12) compliare i files;
13) creare gli stub e gli skeletons con rmic (RMI compiler);
14) far partire registro RMI;
15) far partire il server;
16) far partire il client;

Vediamo passo passo i punti raggruppati per colore che evidenzia funzionalità correlate.
:

Definire l'interfaccia:

```
package it.unive.dsi.labreti.rmi;

import java.rmi.*;
public interface Prova extends Remote
{
  String method(Integer i) throws RemoteException;
}
```

Implementazione dell'oggetto remoto e del server;

```
package it.unive.dsi.labreti.rmi;

import java.rmi.*;

public class ProvaImpl extends UnicastRemoteObject
implements Prova
{

Integer i;
  public ProvaImpl() throws RemoteException
  {
    super(); // esporta l'oggetto;
             // esportandolo si possono generare
eccezioni
  }

  public String method(Integer i)
  {

this.i=i;
    return "risultato;
  }

  public static void main(String[] args)
```

```
  {
    if ( System.getSecurityManager()==null)
      System.setSecurityMaanger(new
RMISecurityManager());

    try
    {
      ProvaImpl obj = new ProvaImpl();

       Naming.rebind("//host/ServerDiProva",obj);
     }
     catch(Exception e)
     {
       System.out.printl("ProvaImpl error:"+
e.getMessage())
       e.printStacjTrace();
     }
   }
}
```

Implementazione del client:

```
package it.unive.dsi.labreti.rmi;

import java.rmi.*;
public class ProvaClient
{

   public static void main(String[] args)
   {
       try
        {
       Prova obj = (Prova)
Naming.lookup("//host/ServerDiProva");
        String res = obj.method(4);
        System.out.printl(res);
       }
      catch (RemoteException re)
      {
        System.out.println("ProvaClient
error:"+re.getMessage());
         re.printStackTrace();
      }
    }
}
```

Compilazione:

```
javac Prova.java ProvaImpl.java ProvaClient.java;

rmic it.unive.dsi.labreti.rimi.ProvaImpl
```

Lanciare le applicazioni:

```
rmiregistry&
java it.unive.dsi.labreti.rmi.ProvaImpl &
java. it.unive.dsi.labreti.rmi.ProvaClient &
```

Considerazioni aggiuntive

Come abbiamo visto, i metodi invocati remotamente possono avere dei tipi di ritorno. Inoltre i metodi possono avere dei parametri e più in generale anche il tipo dell'oggetto remoto stesso può beneficiare del polimorfismo. Nasce quindi una domanda: cosa succede se la virtual machine della macchina client non conosce "il tipo reale" del valore di ritorno, diei parametri o dell'oggetto stesso. E' possibile configurare in maniera semplice il registry e il server in modo che il bytecode contente il necessario al client per conoscere il tipo del server venga "scaricato" dal client. In questo modo è possibile utilizzare il polimorfismo anche tra macchine diverse.

L'url con il quale il server collega l'oggetto al nome ha la seguente forma:

- Non è necessario specificare nessun protocollo;
- Se non viene specificato il nome o l'IP dell'host, viene assunto l'host locale;
- Si può specificare una porta se questa è diversa da la porta di default 1099;
- Per raqgioni di sicurezza un'applicazione puo collegare e scollegare (bind e unbind) oggetti solo su un registro che gira sulla stessa macchina, metre ovviamente la ricerca può avvenira anche da altri server.

Il SecurityManager è necessario solo se il client non può accedere alla classe (generata da rmic) degli stub.
Le caratteristiche offerte dalla classe base UnicastRemoteObject sono:

- usa il trasporto di default tramite socket;
- lo skeleton è sempre in esecuzione e in attesa di chiamate.

Notiamo che la sincronizzazione degli accessi non viene gestita da RMI: se ci sono due client che contemporaneamente invocano un metodo dell'oggetto remoto, nel server ci saranno due thread in esecuzione sul codice del metodo.

Lezione 15– Gestione sicurezza

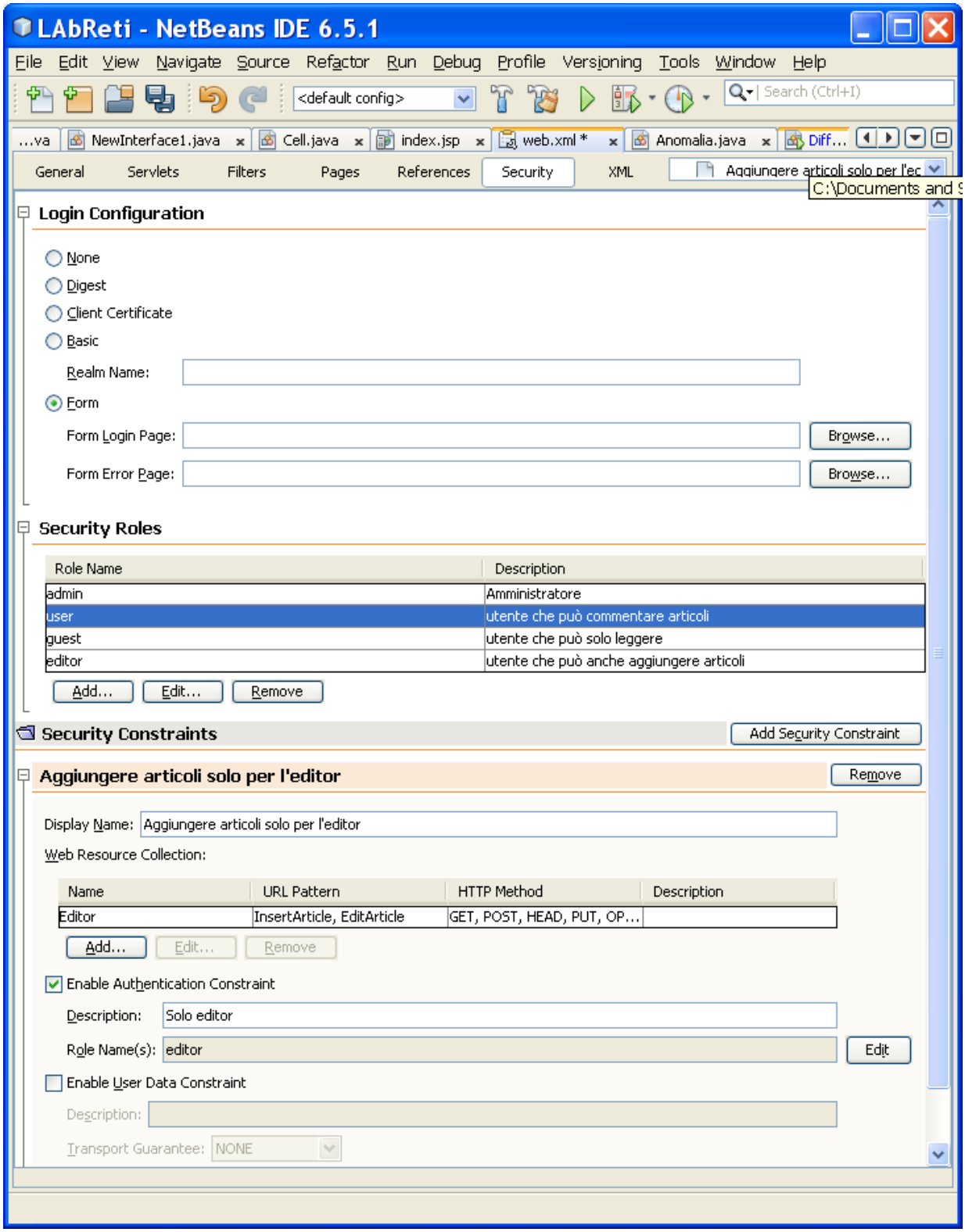
Gestione accessi tramite motore delle servlet

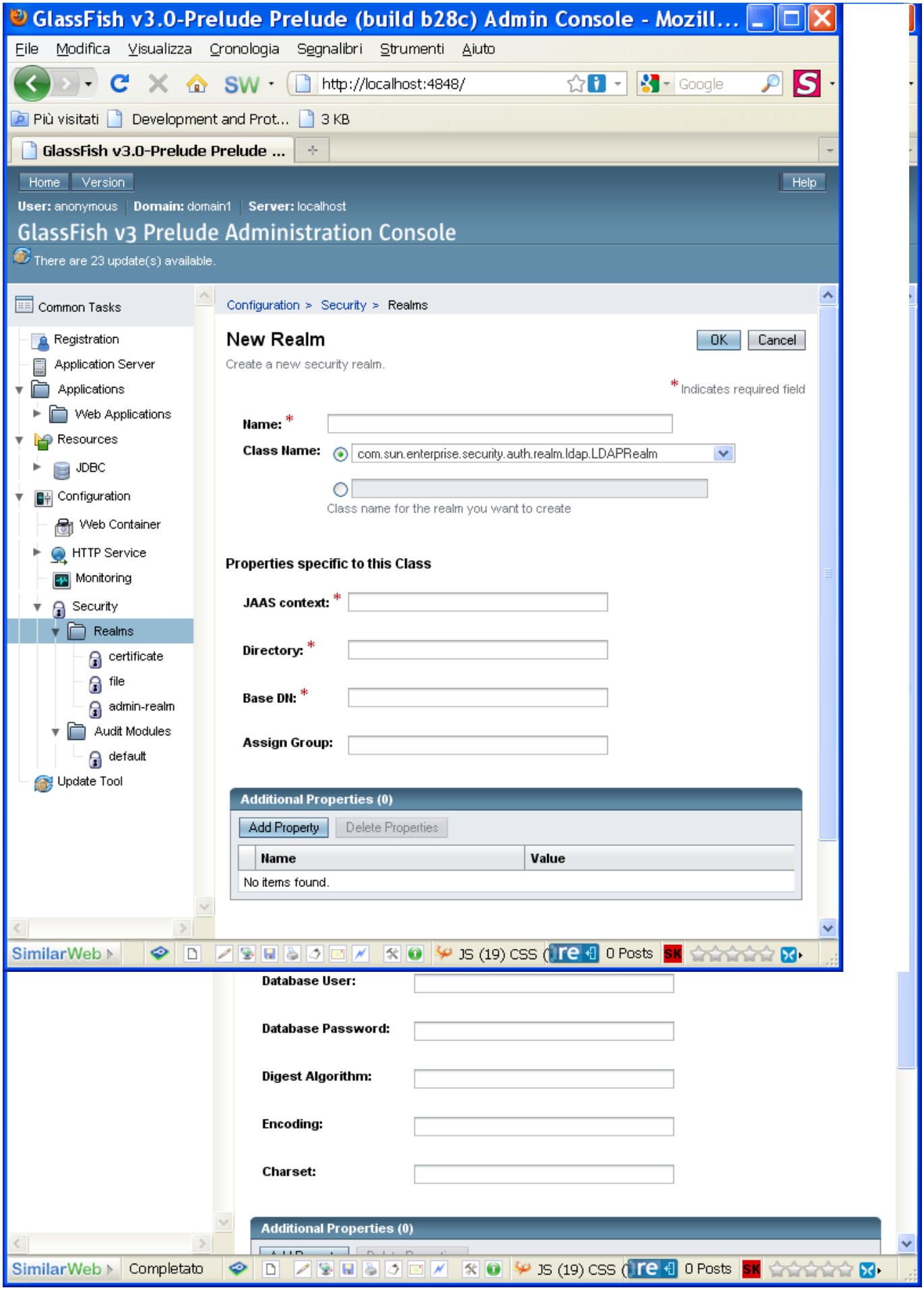
Gli accessi all'applicazione Web possono essere gestiti in automatico dal motore delle servlet. Sono necessarie due configurazioni: una nell'applicazione (file web.xml) e una nel server (dipendente dal server).

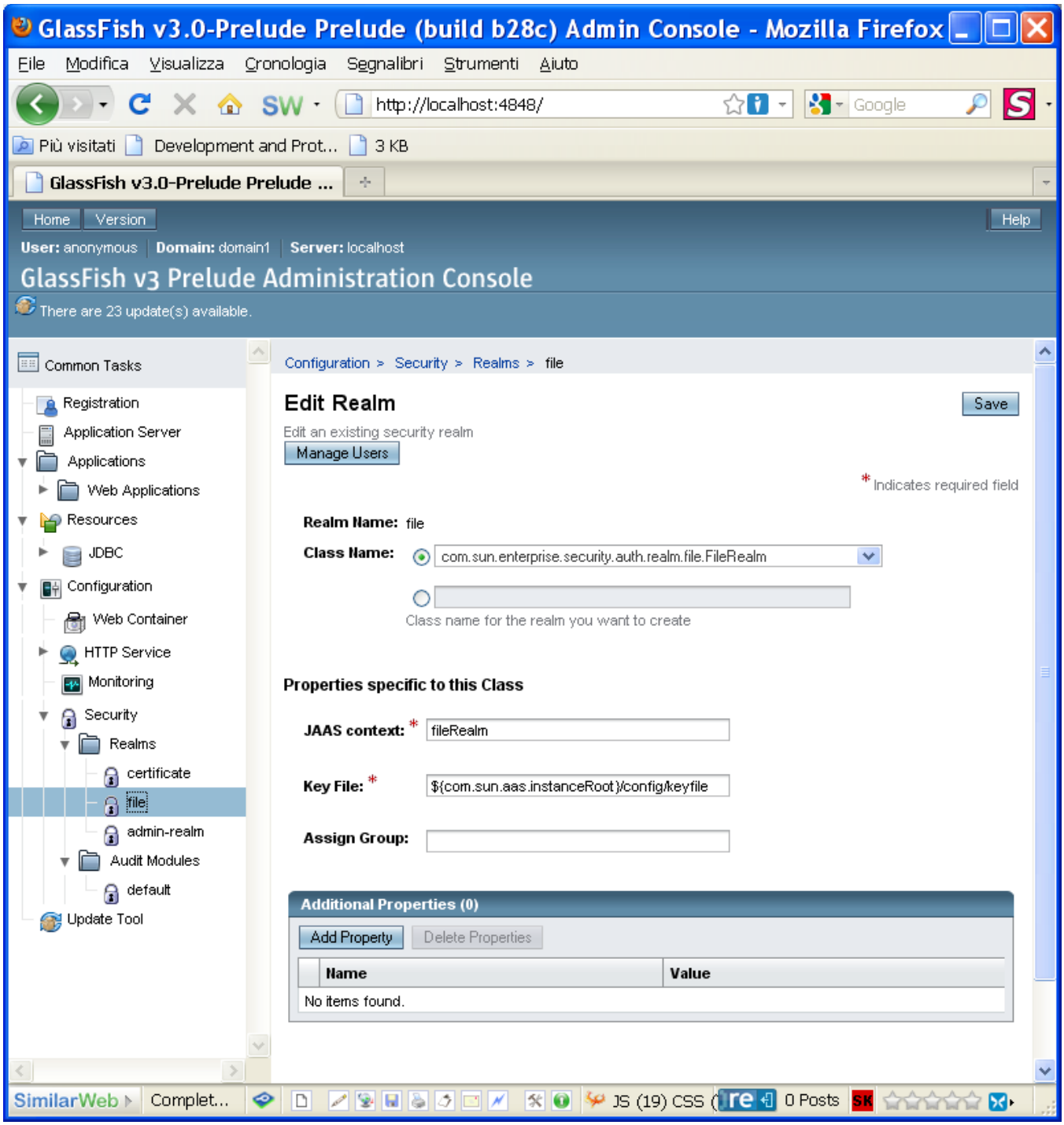
Nell'applicazione si decide quali:

1. come fare il login
2. quali sono i ruoli
3. che risorse possono essere viste dai vari ruoli

Nel server si decide in quale modo recuperare i dati degli utenti (login, password e ruolo di ogni utente). Questi dati possono essere recuperati in vari modi: file di testo, DB o LDAP.


LAbReti - NetBeans IDE 6.5.1
File Edit View Navigate Source Refactor Run Debug Profile Versioning Tools Window Help
<default config>
Search (Ctrl+I)
NewInterface1.java
Cell.java
index.jsp
web.xml *
Anomalia.java
Diff...
General Servlets Filters Pages References Security XML
Aggiungere articoli solo per l'ec
C:\Documents and S
Login Configuration
None
Digest
Client Certificate
Basic
Realm Name:
Form
Form Login Page:
Browse...
Form Error Page:
Browse...
Security Roles
Role Name
Description
admin
Amministratore
user
utente che può commentare articoli
guest
utente che può solo leggere
editor
utente che può anche aggiungere articoli
Add...
Edit...
Remove
Security Constraints
Add Security Constraint
Aggiungere articoli solo per l'editor
Remove
Display Name: Aggiungere articoli solo per l'editor
Web Resource Collection:
Name
URL Pattern
HTTP Method
Description
Editor
InsertArticle, EditArticle
GET, POST, HEAD, PUT, OP...
Add...
Edit...
Remove
Enable Authentication Constraint
Description: Solo editor
Role Name(s): editor
Edit
Enable User Data Constraint
Description:
Transport Guarantee: NONE


GlassFish v3.0-Prelude Prelude (build b28c) Admin Console - Mozill...
File Modifica Visualizza Cronologia Segnalibri Strumenti Aiuto
http://localhost:4848/
Google
Più visitati
Development and Prot...
3 KB
GlassFish v3.0-Prelude Prelude ...
Home
Version
Help
User: anonymous
Domain: domain1
Server: localhost
GlassFish v3 Prelude Administration Console
There are 23 update(s) available.
Common Tasks
Registration
Application Server
Applications
Web Applications
Resources
JDBC
Configuration
Web Container
HTTP Service
Monitoring
Security
Realms
certificate
file
admin-realm
Audit Modules
default
Update Tool
Configuration > Security > Realms
New Realm
OK
Cancel
Create a new security realm.
* Indicates required field
Name: *
Class Name:
com.sun.enterprise.security.auth.realm.ldap.LDAPRealm
Class name for the realm you want to create
Properties specific to this Class
JAAS context: *
Directory: *
Base DN: *
Assign Group:
Additional Properties (0)
Add Property
Delete Properties
Name
Value
No items found.
SimilarWeb
JS (19) CSS (
0 Posts
SK
Database User:
Database Password:
Digest Algorithm:
Encoding:
Charset:
Additional Properties (0)
SimilarWeb
Completato
JS (19) CSS (
0 Posts
SK



Cenni JNDI

Come un database server, anche un directory server memorizza e gestisce delle informazioni ma, a differenza da un database relazionale general purpose, in un directory server:

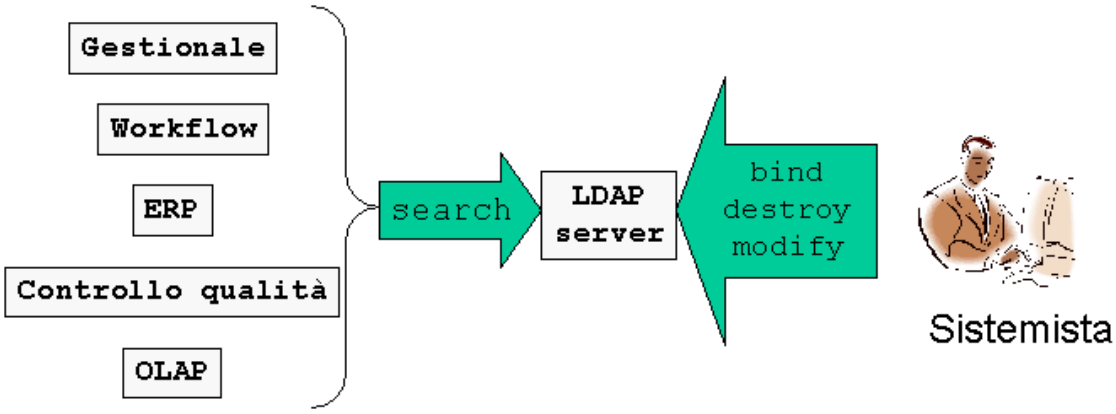
- le ricerche (letture) sono molto maggiori rispetto alle scritture;
- non è adatto a gestire informazioni che variano molto di frequente;
- tipicamente non vengono supportate le transazioni ;
- è previsto un linguaggiod'interrogazione più semplice dell'SQL.

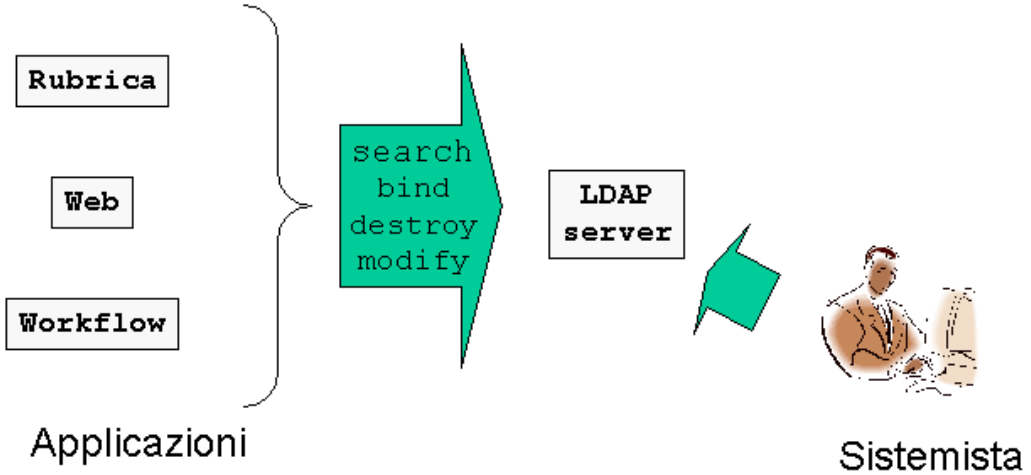
La ricerca delle informazioni è privilegiata per due ordini di motivi:

1. le ricerche sono veloci;
2. le applicazioni che possono beneficiare della ricerca (lettura) in un server LDAP sono molte, mentre le applicazioni che devono modificare i dati in un server LDAP sono poche;

Gerarchia dei nomi

Un nome LDAP può essere pensato come un path di un file in una cartella del sistema operativo. Quindi un nome è specificato da una gerarchia che è adatto in tutti quei casi in cui l'applicazione modella una gerarchia.





Per accedere a un server LDAP, JAVA fornisce l'API JNDI che permette di accedere a un generico servizio di Naming e Directory. Questo significa che alcune operazioni per accedere a LDAP non sono

così semplici e dirette come potrebbero esserlo usando un API specifica per LDAP.
Le operazioni fatte dall'API JNDI per accedere a un LDAP server sono le seguenti:

```
java.net.Socket soc = new
java.net.Socket("ldap.dsi.unive.it",389);

java.io.OutputStream out = soc.getOutputStream();
java.io.InputStream in = soc.getInputStream();

compose(request); // OK semplice

out.write(request);
in.read(response);

parse(response); // Difficile
```

Ecco il codice della classe Ldap che fornisce i principali servizi per accedere a un server LDAP usando JNDI.

```
import javax.naming.directory.*;
import javax.naming.*;

public class Ldap
{
  public static String driver = "com.sun.jndi.ldap.LdapCtxFactory";

  public final String AUTH_SIMPLE="simple";

  DirContext ctx = null;

  public void connect(String ldapUrl) throws NamingException
  {
    //creazione di una hashtable vuota
    java.util.Hashtable env = new java.util.Hashtable();

    //Specifica del driver da usare
    env.put(Context.INITIAL_CONTEXT_FACTORY, driver);

    //Specifica del LDAP server da contattare
    env.put(Context.PROVIDER_URL, ldapUrl);

    //caricamento driver e collegamento al server
    ctx = new directory.InitialDirContext(env);

  }

  public NamingEnumeration search(String base, String filtro,
SearchControls vincoli)
    throws NamingException
  {
   try{
     return ctx.search(base,filtro,vincoli);
   }
   catch(NameNotFoundException nnfe)
   {
     return null;
```

```
   }
  }
...
}
```

Esempio d'uso:

```
try
{
Ldap ldap = new Ldap();

ldap.connect("ldap:/ldap.dsi.unive.it:389");
}
catch(NamingException ne)
{ne.printStackTrace();}

String base = "dc=unive,dc=it";
String filtro = "cn=*";

SearchControls vincoli= new SearchControls();
vincoli.setSearchScope(SearchControls.SUBTREE_S
COPE);

...= ldap.search(base,filtro,vincoli);
```

Sintassi dei Search Filter

La ricerca dei dati è filtrata attraverso una stringa con sintassi articolata, prima di vedere tutti gli elementi della sintassi ne vediamo un esempio e il suo significato.

```
(& (sn=Roncato) (mail=*) ( ! (cn=Alessandro Roncato)) )
```

Il significato è il seguente: un'entry che abbia l'attributo mail presente, il surname sia Roncato e il commonName non sia Alessandro Roncato.
Come si può vedere, sono già standardizzati dei concetti che aiutono a esprimere ricerche che riguardano gli utenti.
La query è composta da test su degli attributi, questi test sugli attributi compongono i **termini** del linguaggio che sono poi combinati attraverso gli **connettivi logici**.
I test sui valori degli attributi (o termini):

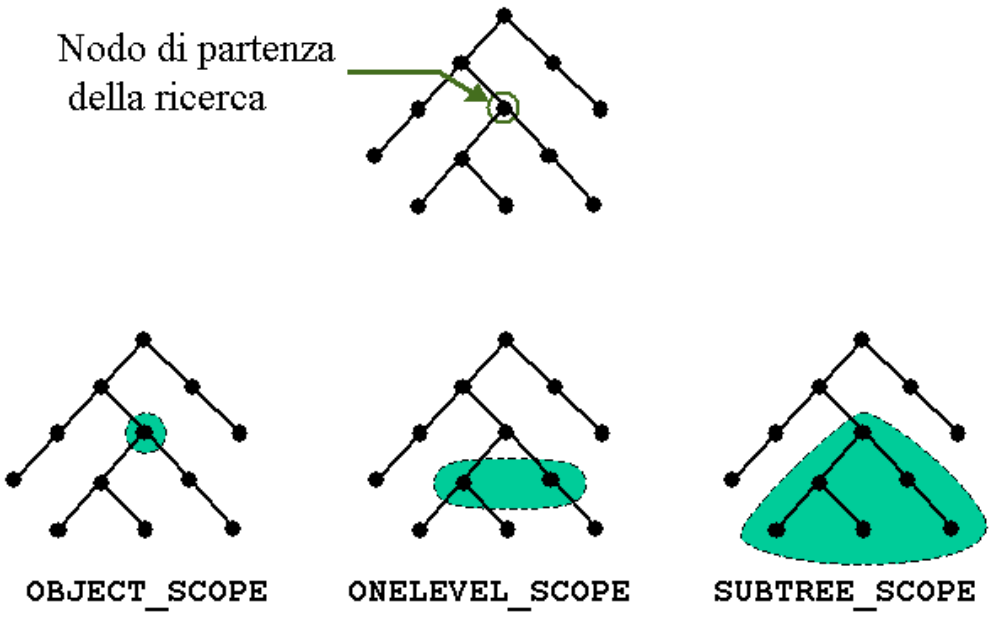
- Uguaglianza:
  - (sn=Roncato) il surname dev'essere Roncato;
- Substring:
  - (sn=*ato) il surname deve terminare in 'ato';
  - (sn=Ron*) il surname deve iniziare con 'Ron';
  - (sn=*ca*) il surname deve contenere 'ca';
  - (sn=R*ca*to) il surname deve iniziare con 'R', finire in 'to' e contenere 'ca';
- Approssimazione:
  - (sn~=Roncato)il surname deve "suonare" come Roncato;
- Confronti di Ordinamento:
  - (sn>=Roncato) il surname dev'essere Roncato o uno che viene dopo nell'ordine alfabetico;
  - (sn<=Roncato) il surname non deve venire dopo Roncato nell'ordine alfabetico;
- Presenza:

- (mail=*) dev’esserci l’attributo mail.

I connettivi logici sono:
AND &
OR |
NOT !
Es. Assenza (!(mail=*)) l’entry non deve avere l’attributo mail

Vincoli

Oltre al filtro, che seleziona quali entry restituire, è possibile specificare anche degli ulteriori vincoli su come effettuare la ricerca e come restituire i risultati.
La funzione poiù importante è quella di specificare lo scope della ricerca:



Ad esempio:
```
SearchControls.OBJECT_SCOPE
SearchControls.ONELEVEL_SCOPE
SearchControls.SUBTREE_SCOPE

vincolo.setSearchScope(...);
```

Per indicare il Search Filter si specifica una combinazione Booleana di test sui valori di attributi:
```
filtro=String();
```

Per indicare gli attributi da restituire:
```
vincoli.setReturningAttributes(String[])
```
Si può anche scegliere di non volere il valore, ma soltanto sapere se gli attributi ci sono:
```
vincoli.setReturningObjFlag(boolean);
```
Limiti: si possono specificare limiti in tempo di ricerca:
```
vincoli.setTimeLimit(int);
```
e in quantità di entry restituite:
```
vincoli.setCountLimit(long);
```

Risultato della ricerca

Vediamo ora come analizzare il risultato di una ricerca. Supponiamo di avere la seguente situazione logica: 3 entry DN1, DN2 e DN3. Ogni entry ha alcuni attributi e qualche attributo ha più valori (es. mail).

- DN1:
  - cn:
    - pippo
  - mail:
    - pippo@localhost
    - pippo@unive.it
- DN2:
  - cn:
    - pluto
  - sn:
    - pluto
- DN3:
  - cn:
    - quak
  - mail:
    - quak@localhost
    - quak@unive.com

Per esaminare il risultato della query abbiamo bisogno quindi di tre cicli:

1. ciclo sulle 0 o più entry, quindi dobbiamo esaminare tutte le entry con un ciclo;
2. ciclo sugli attributi cercati;
3. ciclo sui 0 o più valori degli attributi;

```
NamingEnumeration nomi = ldap.search(base,filtro,vincoli);

while (nomi != null && nomi.hasMore())
{
  SearchResult sr=(SearchResult)nomi.next();

  String dn = sr.getName() + "," + base;

  System.out.println("DN = "+dn);

  String[] nomeAttributi = new String[]{"sn","cn","mail"};

  Attributes ar = ldap.getAttributes(dn, nomeAttributi);

  if (ar != null)
  for (int i =0;i<nomeAttributi.length;i++)
  {
    Attribute attr = ar.get(nomeAttributi[i]);
    if (attr != null)
    {
      System.out.println(nomeAttributi[i]+":");
      Enumeration vals = attr.getAll();

      while (vals.hasMoreElements())
        System.out.println("\t"+vals.nextElement());
```

```
        }
       System.out.println("\n");
      }
    }
```

Per effettuare delle operazioni di modifica dobbiamo effettuare una connessione con Autenticazione e mplementare l'interfaccia DirContext;
Modifica vera e propria che può essere :

- Aggiungere: funzionalità `bind`;
- Modificare: funzionalità `modify`;
- Eliminare: funzionalità `destroy`.

Esempio:

```
java.util.Hashtable env = new
java.util.Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
driver);
env.put(Context.PROVIDER_URL,ldapUrl);

//impostiamo il tipo di autenticazione
env.put(Context.SECURITY_AUTHENTICATION,AUTH_SI
MPLE);

//impostiamo l'utente con cui collegarsi
env.put(Context.SECURITY_PRINCIPAL,user);

//impostiamo la password dell'utente
env.put(Context.SECURITY_CREDENTIALS,password);

ctx = new InitialDirContext(env);
```

Per aggiungere una nuova entry bisogna definire una nuova classe che implementi l'interfaccia DirContext.
Questa classe deve:

- essere in grado di creare il proprio DN;
- sapere come memorizzare i propri attributi;
- e fornire altri meccanismi per gestire i dati recuperati.

Nel nostro esempio vedremo la classe Persona che implementa i seguenti metodi dell'interfaccia DirContext:

- `getAttributes()`;
- il costruttore `Persona()`.

I metodi rimanenti che il compilatore richiede essere definiti affinché la classe implementi l'interfaccia DirContext, sono definiti e generano solamente delle eccezioni. Possiamo farci aiutare da Netbeans o JBuilder per scrivere in automatico tutti questi metodi.

```
public class Person implements DirContext
{
```

```
  String type;
  Attributes myAttrs;

   public Person(String uid,String givenname, String sn, String ou,String
mail)
   {
      myAttrs = new BasicAttributes(true);

      Attribute oc = new BasicAttribute("objectclass");
      oc.add("organizationalPerson");
      oc.add("person");
      oc.add("top");

      Attribute ouSet = new BasicAttribute("ou");
      ouSet.add("People");
      ouSet.add(ou);
      type = uid;

      myAttrs.put(oc);
      myAttrs.put(ouSet);
      myAttrs.put("uid",uid);
      myAttrs.put("sn",sn);
      myAttrs.put("givenname",givenname);
      myAttrs.put("mail",mail);
   }
   public Attributes getAttributes(String name) throws NamingException
   {
      return myAttrs;
   }

   public Attributes getAttributes(Name name) throws NamingException
   {
      return getAttributes(name.toString());
   }

  public Attributes getAttributes(String name, String[] ids) throws
NamingException
  {
   Attributes answer = new BasicAttributes(true);
   Attribute target;
   for (int i = 0; i < ids.length; i++)
   {
     target = myAttrs.get(ids[i]);
     if (target != null)
        answer.put(target);
   }
   return answer;
  }

  public Attributes getAttributes(String name, String[] ids) throws
NamingException
  {
   Attributes answer = new BasicAttributes(true);
   Attribute target;
   for (int i = 0; i < ids.length; i++)
   {
     target = myAttrs.get(ids[i]);
```

```
    if (target != null)
       answer.put(target);
   }
   return answer;
  }

  public Attributes getAttributes(Name name, String[] ids) throws
NamingException
   {
      return getAttributes(name.toString(), ids);
   }

   public String toString()
   {
      return type;
   }
...// tutti gli altri metodi che non fanno niente se non generare
un'eccezione
```

Aggiunta di un Entry

Una volta definita la classe che implementi l'interfaccia DirContext., possiamo procedere ad inserire un oggetto della classe nella directory.
L'inserimento corrisponde a collegare (bind) un nome all'oggetto.
Il nome è ovviamente un DN.
Esempio:

```
Persona p = new Persona("Rossi", "Paolo", "Rossi",
"ou=Accounting","rossi@unive.it");
ctx.bind("uid=prossi,ou=Users,dc=unive,dc=it", p);
```

Modifica di un entry

Modificare un'entry significa:

- aggiungere un attributo;
- eliminare un attributo;
- sostituire il valore di un attributo.

Per effettuare queste operazioni utilizziamo la classe: ModificationItem nel seguente modo:

```
ModificationItem[] mods = new ModificationItem[3];
// vogliamo effettuare tre modifice
// sui seguenti 3 attributi dell'entry:

Attribute a0 =
  new BasicAttribute("telephonenumber", "02-555-2555");

Attribute a1 = new BasicAttribute("sn", "Bianchi");

Attribute a2 =
       new BasicAttribute("mail", "rossi@unive.it");
mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE,
a0);
mods[1] = new
```

```
ModificationItem(DirContext.REPLACE_ATTRIBUTE, a1);
mods[2] = new
ModificationItem(DirContext.DELETE_ATTRIBUTE, a2);

ctx.modifyAttributes(MY_ENTRY, mods);
```

Cancellare un entry

Cancellare un entry è l'operazione più semplice: basta indicare il DN della entry da eliminare:

```
ctx.destroySubcontext("uid=prossi,ou=Users,dc=unive,d
c=it");
```

Altre operazioni

`ctx.createSubcontext()`: per aggiungere un'entry;
`ctx.lookup()`: per recuperare una data entry di cui si conosce il DN;
`ctx.rebind()`: per riassegnare il DN ad un altro oggetto/entry;
`ctx.rename()`: per cambiare un DN

Lezione 16 - Applet

Un'applet è un particolare tipo di programma Java il cui output è pensato per essere incluso in pagine Web. Quando un utente visualizza una pagina Web che contiene un'applet, l'applet viene eseguita **localmente** (cioè sulla macchina dell'utente) e **non** **remotamente** (cioè sul server come le servlet). L'applet, come tutte le applicazioni Java, gira su una Virtual Machine su un processo separato da quello del browser.

Dato che il codice dell'applet viene fornito dal server ma gira sulla macchina del client l'applet stessa è soggetta a delle limitazioni per renderne sicura l'esecuzione stessa.
Le principali limitazioni imposte alle applet sono:

- Non si può leggere dal disco locale;
- Non si può scrivere nel disco locale;
- Non si possono aprire connessioni con server diversi dal server da cui proviene l'applet;
- Non si possono invocare programmi locali;
- Non si possono scoprire informazioni privare riguardo all'utente.

Per ovviare a queste limitazioni è possibile firmare le applet. In questo caso è possibile associare un file policy file che descrive tutte le operazioni che possono essere compiute dall'applet (Es. leggere un certo file sul disco locale).

HTML

Il browser dev'essere istruito su come visualizzare l'applet stessa. Questo lo otteniamo attraverso il tag HTML "APPLET" che associa il file .class alla pagina Web. L'attributo CODE indica con un URL relativo il file .class da caricare. I parametri WIDTH e HEIGHT indicano rispettivamente la larghezza e l'altezza dello spazio da riservare all'applet nella pagina Web.
Ecco un esempio di codice HTML per la visualizzazione di un'applet:

```
<html>
<head>
  <title>AppletReload</title>
</head>
<body>
<applet code="ReloadApplet.class" width="120" height="60">
<b>Attenzione: Devi abilitare le Applet!</b></applet>
</body>
</html>
```

java 2 (jdk 1.2)

Il tag Applet è usabile solo se la nostra applet utilizza la versione del linguaggio 1.
In particolare nella prima versione non sono disponibili:

1. Swing (a meno di non scaricare le classi che la contengono);
2. Java 2D;
3. collezioni (List, Map etc.);
4. ottimizzazioni per l'esecuzione più veloce.

Per ottenere tutti i vantaggi di Java 2 dobbiamo operare in maniera diversa e usare i seguenti tag:

- Internet Explorer: OBJECT;

- Netscaep: EMBED.

In particolare dovremmo procedere nel seguente modo:

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = "600" HEIGHT = "300" NAME = "ReloadApplet"
codebase="http://java.sun.com/products/plugin/1.1.1/jinstall-111-
win32.cab#Version=1,1,1,0">
<PARAM NAME = CODE VALUE = "ReloadApplet.class" >
<PARAM NAME = NAME VALUE = "ReloadApplet" >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.1" java_CODE =
"ReloadApplet.class" NAME = "ReloadApplet" WIDTH = "600" HEIGHT =
"300"   pluginspage="http://java.sun.com/products/plugin/1.1.1/plugin-
install.html">
<NOEMBED></COMMENT>

</NOEMBED></EMBED>
</OBJECT>
```

java 2 e JSP

La gestione dei tag per le applet viene fatto in automatico dal motore delle servlet nel seguente modo:

```
<jsp:plugin type="applet"
            code="ReloadApplet.class"
            width="475" height="350">
</jsp:plugin>
```

che produce in automatico i tag HTML necessari.

Java

```
import java.applet.Applet;
import java.awt.Graphics;
public class Ciao extends Applet {
   public void paint(Graphics g) {
       g.drawString("Ciao", 50, 25);
       }
}
```

La classe Applet è sottoclasse di Panel, nella libreria Awt. Questo implica in particolare che:
Applet ha come layout manager di default il Flow Layout Manager, che dispone gli elementi grafici da sinistra a destra, con allineamento centrale
Applet eredita le variabili ed i metodi delle sue superclassi Component, Container e Panel.
Con un applet è possibile visualizzare all'interno della finestra della pagina Html tutto quello che è possibile fare con una normale applicazione Java senza le limitazione dell'html (vedi sotto esempio dell'applet con testo scorrevole).

ciclo vita di una Applet

Anche l'applet analogamente alla servlet ha un ciclo di vita è una serie di metodi che vengono invocati automaticamente dalla Virtual Machine dove gira l'applet. I metodi invocati sono:

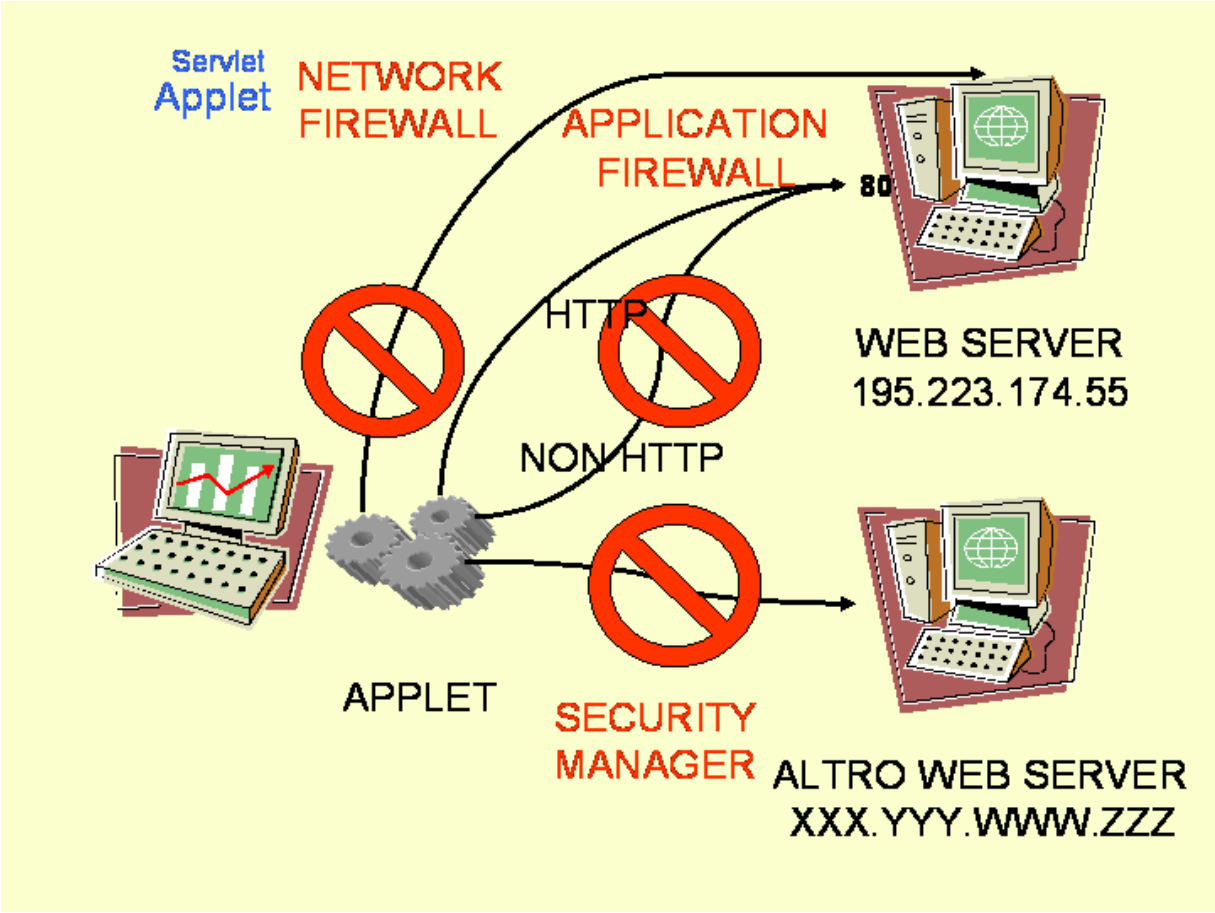
- `init()`: quando si apre un documento con un applet, viene invocato automaticamente il metodo `init()` per l'inizializzazione dell'applet. Può essere usato per inizializzare le variabili. Questo

metodo è chiamato solo una volta per applet;

- `start()`: quando un documento con un applet viene aperto, viene chiamato il metodo `start()` automaticamente dopo `init()`, per iniziare l'applet. Questo metodo, ad esempio, è invocato ogni qual volta l'applet viene "rivisitato".
- `stop()`: viene invocato automaticamente quando l'utente si sposta fuori dalla pagina su cui è situato l'applet . Serve per bloccare attività che possono rallentare il sistema quando l'utente non sta utilizzandole (es. animazioni) . I metodi start e stop formano una coppia: start attiva un comportamento, stop lo disattiva.
- `destroy()` : Viene invocato quando l'applet viene dismesso (ad es. quando viene chiuso il browser), e in questo caso viene prima invocato il metodo stop().
  Tutte le risorse legate all'applet vengono rilasciate.
- `paint()` : Questo metodo è invocato direttamente dal browser dopo che `init()` è stato completamente eseguito e dopo che `start()` abbia iniziato l'esecuzione. E' invocato ogni qual volta l'applet deve essere ridipinto
- `repaint()` : invoca il metodo `update()` al più presto (non da overridden) `update()` permette di "ri-dipingere" l'applet, e può essere riscritto. Per default, "pulisce" lo sfondo e chiama il metodo `paint()`.

Comunicazione Servlet/Applet

Uno dei problemi che possono essere risolti da un'applet è quello di permettera una più immediata e tempestiva comunicazione tra server e browser. La limitazione del protocollo http non permette di avere un meccanismo semplice con cui il server in maniera "attiva" avvisi il client che nuove informazioni sono disponibili o più in generale di comunicare col client stesso. Vediamo ora una soluzione applet/servlet per ottenera una comunicazione in cui il server avvisa il client al verificarsi di certi eventi.



La Sandbox presente nel browser impedisce all'applet di aprire connessione con server diversi da quello dalla quale l'applet stessa è stata scaricata. La presenza di Firewall potrebbe impedire di aprire una connessione TCP su porte diverse dalla porta 80 (porta che deve per forza essere aperta in quanto porta standard del protocollo HTTP). Quindi l'unica porta sicuramente aperta è la porta 80 dove è in attesa di connessioni il server HTTP. Il fatto che ci sia il server HTTP in attesa di connessioni HTTP e dal fatto che ci potrebbe essere un firewall a livello applicativo che impedisci connessioni diverse da HTTP suggerisce di usare come schema iniziale di comunicazione tra Applet e Servlet il protocollo HTTP stesso.

Il fatto di usare come schema iniziale di comunicazione tra Applet e Server quello standard HTTP ha anche il vantaggio che è possibile usare la stessa Servlet sia per comunicare interattivamente con un Applet che per fornire una risposta statica generando una pagina HTML.
La servlet rimane “simile” a quelle viste.
Azioni da compiere nell'Applet per invocare una servlet.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
public class PushApplet extends Applet implements ActionListener
{
  TextArea taResults;
  Button btnStart;
  public void init()
  {
    setLayout(new FlowLayout(FlowLayout.LEFT));
    btnStart = new Button("Start");
    btnStart.addActionListener(this);
    add("North", btnStart);
    taResults = new TextArea(10, 80);
    add("Center", taResults);
  }
  public void execute()
  {
    char[] buf = new char[256];
    try
    {
      URL url = new URL(getDocumentBase(),"PushServlet");
      URLConnection uc = url.openConnection();
      uc.setDoInput(true);
      uc.setUseCaches(false);

      InputStreamReader in = new InputStreamReader(uc.getInputStream());
      int len = in.read(buf);
      while (len != -1 )
      {
        taResults.replaceRange(new String(buf,0,len),0,len);
        len = in.read(buf);
      }
      in.close();
    }
    catch(MalformedURLException e)
    {
      taResults.setText(e.toString());
    }
    catch(IOException e)
    {
      taResults.setText(e.toString());
    }
  }
  public void actionPerformed(ActionEvent ae)
  {
    execute();
  }
```

```
}
```

Nota: l'applet rimane bloccata in attesa di dati da parte del server. Andrebbe gestita la richiesta e l'attesa dei dati in un thred secondario (vedi esempio testo scorrevole).

Possiamo utilizzare un'applet per ricaricare tutta la pagina che contiene l'applet utilizzando la seguente istruzioni:

```
getAppletContext().showDocument(new
URL(getDocumentBase(),"AppletServletCommunication.html"));
```

Testo scorrevole

Un'esempio tipico di funzionalità ottenibili con le applet è il testo scorrevole:

```
public class Scroll extends Applet implements Runnable
{
Thread t=null;
String temp;
String text = "Questo testo scorre";
long velocitaBattitura=100;

boolean cont = true;
public void start()
{
t = new Thread(this);
cont = true;
t.start();
}
public void stop()
{
cont = false;
t.interrupt();
t=null;
}
public void run()
{
while (cont)
{
for(int j=0; j<=text.length(); j++)
{
temp = text.substring(j)+ " " + text.substring(0,j);
repaint();
try{t.sleep(velocitaBattitura);}
catch(InterruptedException eint){if (!cont) break;}
}
}
}
public void paint(Graphics g)
{
Font f = new Font("Arial",0,16);
FontMetrics fm = g.getFontMetrics(f);
int w = fm.stringWidth(temp);
g.setColor(Color.white);
g.fillRect(0,0,w+10,30);
g.setColor(Color.black);
g.setFont(f);
```

```
g.drawString(temp, 5, 20);
}

}
```

Sintassi del tag APPLET

```
<applet
[archive=ListaArchivio] // classi o risorse che saranno preloaded. Si
puo' indicare una lista di file JAR (Java Archive) dalla quale
estrarre l'applet
code=MioFile.class // nome del file che contiene il compilato,
relativo alla stessa URL
width=pixels heigh=pixels // dimensioni iniziali del display
dell'applet
[codebase=codebaseURL] // directory che contiene il codice dell'applet

[alt=testoAlternativo] // se il browser legge l'applet tag ma non può
eseguire l'applet
[name=nomeIstanzaApplet] // permette di identificare diversi applet
nella stessa pagina per esempio, permette ad applet di comunicare tra
loro…
[align=allineamento]// left right top texttop middle baseline bottom
[vspace=pixels] [hspace=pixels]
>
[<param name=attributo1 value=valore>]// valori specificati
dall'esterno
[<param name=attributo2 value=valore>]// ci sarà un getParameter()
nell'applet
. . .
</applet>
```

L'uso dei parametri permette in modo molto semplice di personalizzare il comportamente dell'applet.
Esempio:

```
public void init(){
String param = getParameter("testo");
...
}
```

Nel file HTML possiamo usare l'applet con due parametri diversi:

```
<html>
<title>Etichetta</title>
<body>
<applet code = "Etichetta.class"
width=500 height=100
name=primo
align = top
vspace=30
>
<param name = testo value ="testo primo">
</applet>

<p><hr><p>

<applet code = "Etichetta.class"
width=500 height=100
name=secondo
align = right
```

```
vspace=30
>
<param name = testo value ="testo diverso">

</applet>
</body>
</html>
```

Lezione 17 - Javascript

Javascript è un linguaggio di scripting con sintassi simile a Java, da cui deriva il nome, che gira nel processo del browser (e quindi non in una Virtual Machine esterna come le Applet). Alcuni degli oggetti predefinit su cui opera Javascript sono le "parti" del browser, e alcuni metodi agiscono come le funzionalità del browser. Per esempio, sono automaticamente definiti gli oggetti `window` (finestre del browser), `document` (il documento visualizzato), `history` (la "storia" dei siti già visitati dal browser), etc. Gli usi possibili di Javascritp sono innumenrevoli tra i quali validazione dell'input e animazione delle pagine html sono i principali.

Javascript e validazione dell'input

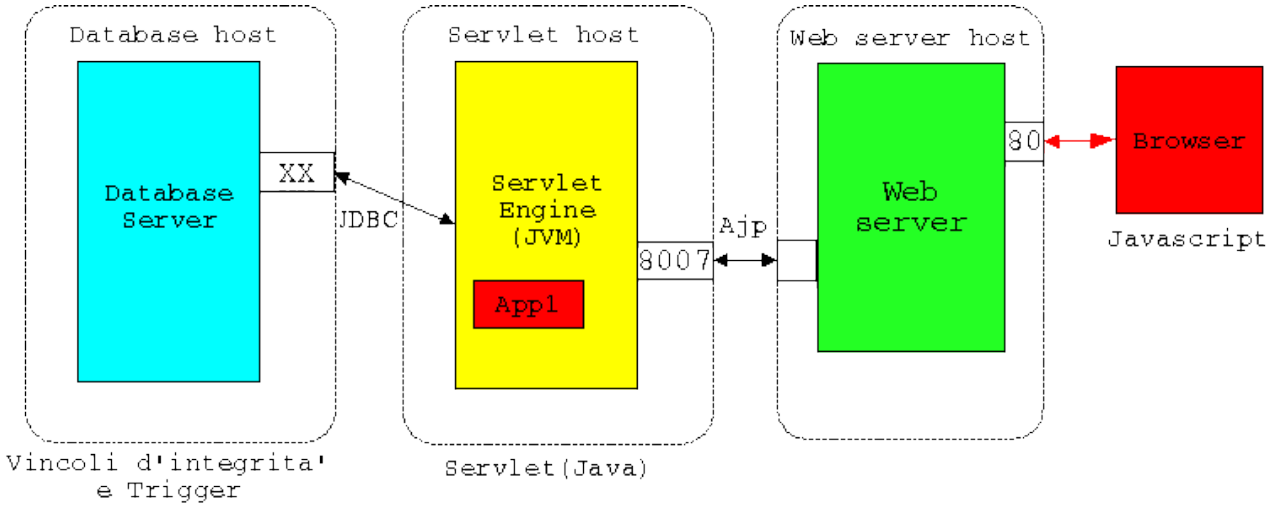
Per verificare la validita' dell'input nel form viene spesso usato javascript. In particolare esaminiamo il seguente codice:

```
<HTML>

<HEAD>
<TITLE>ESERCITAZIONE</title>
<script language="Javascript


<!--
function controlla()
{
log = document.Modulo.LOGIN.value;
if (log.length == 0)
{
alert("Inserire la LOGIN");
document.Modulo.LOGIN.select();
document.Modulo.LOGIN.focus();
return false;
}
if (log.length < 5 || log.length > 8)
{
alert("La LOGIN consiste in una stringa di almeno 5 caratteri e
massimo 8");
document.Modulo.LOGIN.select();
document.Modulo.LOGIN.focus();
return false;
}
numes = document.Modulo.NUMES.value;
if (numes.length == 0)
{
alert("Inserire il numero dell'esercitazione");
document.Modulo.NUMES.select();
document.Modulo.NUMES.focus();
return false;
}
ies = Number(numes);
if (String(ies) != numes)
{
alert("L'ESERCITAZIONE consiste di un numero");
document.Modulo.NUMES.select();
document.Modulo.NUMES.focus();
return false;
```

```
}

}
//-->

</script>
</HEAD>

<BODY bgColor="#8AC6F6">

<FORM ACTION="..." NAME="Modulo" onSubmit="return controlla();" >
<CENTER>
<BR>

LOGIN: <INPUT TYPE="TEXT" NAME="LOGIN" SIZE=8 MAXLENGHT=8 VALUE="">
NUMERO ESERCIZIO: <INPUT TYPE="TEXT" NAME="NUMES" SIZE=8 MAXLENGHT=8
VALUE=""><BR><BR>

<CENTER>
<P>
<INPUT TYPE="SUBMIT" VALUE="INVIA">
<INPUT TYPE="RESET" VALUE="CANCELLA">

</FORM>

</center>

</BODY>
</HTML>
```

Validazione input

Nello schema possiamo vedere dove intervengono i vari vincoli di integrità nel processo di elaboraione dei dati.



Validazione dei dati

	Browser	Application Server	Database
Eludibilità	sì	no	no

Risposta	immediata	intermedia	lenta
Linguaggio	Javascript	Java	SQL
Visibilità codice	sì	no	no
Computazione	distribuita		centralizzata
Internazionalizzazione	sì	sì	no

Problemi Javascript

Anche se Javascript può essere la soluzione di molti problemi è bene non abusarne perché:

1. il browser potrebbe averlo disattivato;
2. versioni diverse di browser si comportano in maniera leggermente diversa in visualizzazione, ma leggere differenze di funzionamento possono portare malfunzionamenti del codice;

Pushlet

Un uso leggermente diverso di Javascript lo vediamo tramite le Pushlet. Sappiamo che non c'é modo da parte del server di comunicare al browser un'informazione se la connessione è stata già chiusa. Sfruttano Javascript possiamo procedere nel seguente modo:

1. il browser richiede la pagina alla Servlet/JSP;
2. la Servlet/JSP fornisce la risposta con i codice HTML completo (quindi compreso il tag </HTML>);
3. la connessione viene mantenuta attiva e i dati eventualmente ancora da inviare vengono forzatamente inviati;
4. il browser è in grado di visualizzare la pagina HTML completa, ma resta ancora in attesa di dati dalla connessione;
5. la Servlet/JSP quando vuole far caricare una pagina, invia nella vecchia connessione un codice Javascript che forza il carcamento della pagina ;
6. il browser esegue il codice Javascript non appena lo riceve e in questo caso ricarica la pagina quando vuole il server.

Vediamo qui di seguito il codice completo di una servlet molto semplice che forza il ricaricamento di se stessa dopo un tempo casuale.

```
import javax.servlet.*;

import javax.servlet.http.*;

import java.io.*;

import java.util.*;

public class Pushlet extends HttpServlet

{
```

```
 private int c=0;

 /**Process the HTTP Get request*/

 public void doGet(HttpServletRequest request,

          HttpServletResponse response) throws ServletException,

  IOException

 {

  response.setContentType("text/html");

  response.setBufferSize(0);

  PrintWriter out = response.getWriter();

  out.println("<html><body><b>count "+c++ +"</b></body></html>");

  out.flush();

  response.flushBuffer();

  try

  {

  Thread.sleep((int)(1000*Math.random()*10));

  }

  catch( InterruptedException e)

  {

  }

  out.println("<script language=\"JavaScript\">");

  out.println("document.location='http://localhost:8080/servlet/Pushlet';");

  out.println("</script>");

  out.close();

 }

}
```

AJAX Asincronus-JAvascript-Xml

Con Javascrit possiamo sostanzialmente simulare la comunicazione Applet/Servlet. Infatti grazie al'uso di richieste Asincrone al server scritte in javascript possiamo sostanzialmente ottenere lo stesso effetto dell'applet. In più, e questo lo possiamo fare solo con Javascript, sfruttando l'API DOM per Javascript, possiamo modificare singole parti della pagina html sostituendala con “pezzi” di pagina inviati dal server. Questi pezzi di pagina sono inviati tramite XML. In conclusione sfruttando richieste Asincone scritte in Javascript con cui otteniamo risposte in XML (AJAX), possiamo ottenere applicazioni WEB con interattività molto simile alle applicazione Desktop.

Ecco la sequenza di azioni del browser in un accesso AJAX:

1. il browser richiede la pagina html
2. il server invia la pagina (statica o dinamica)
3. il cliente visualizza la pagina, se la pagina contiene richieste AJAX queste vengono inltrate al server con un thread secondario
4. l'attività nel browser continua normalmente (nel caso inltrando altre richieste AJAX o interagendo con l'utente)
5. quando il server invia la risposta al cliente, il thread secondario visualizza il risualtato nella pagina HTML modificandola tramite il DOM.

Di solito è conveniente utilizzare una libreria che semplifica la programmazione in AJAX, in questo modo non dobbiamo occuparci delle differenze tra i vari browser e dei dettagli di più basso livello. Per esempio una libreria molto usato è Prototype che potete scaricare e trovare documentazione ed esempio al link: http://www.prototypejs.org/

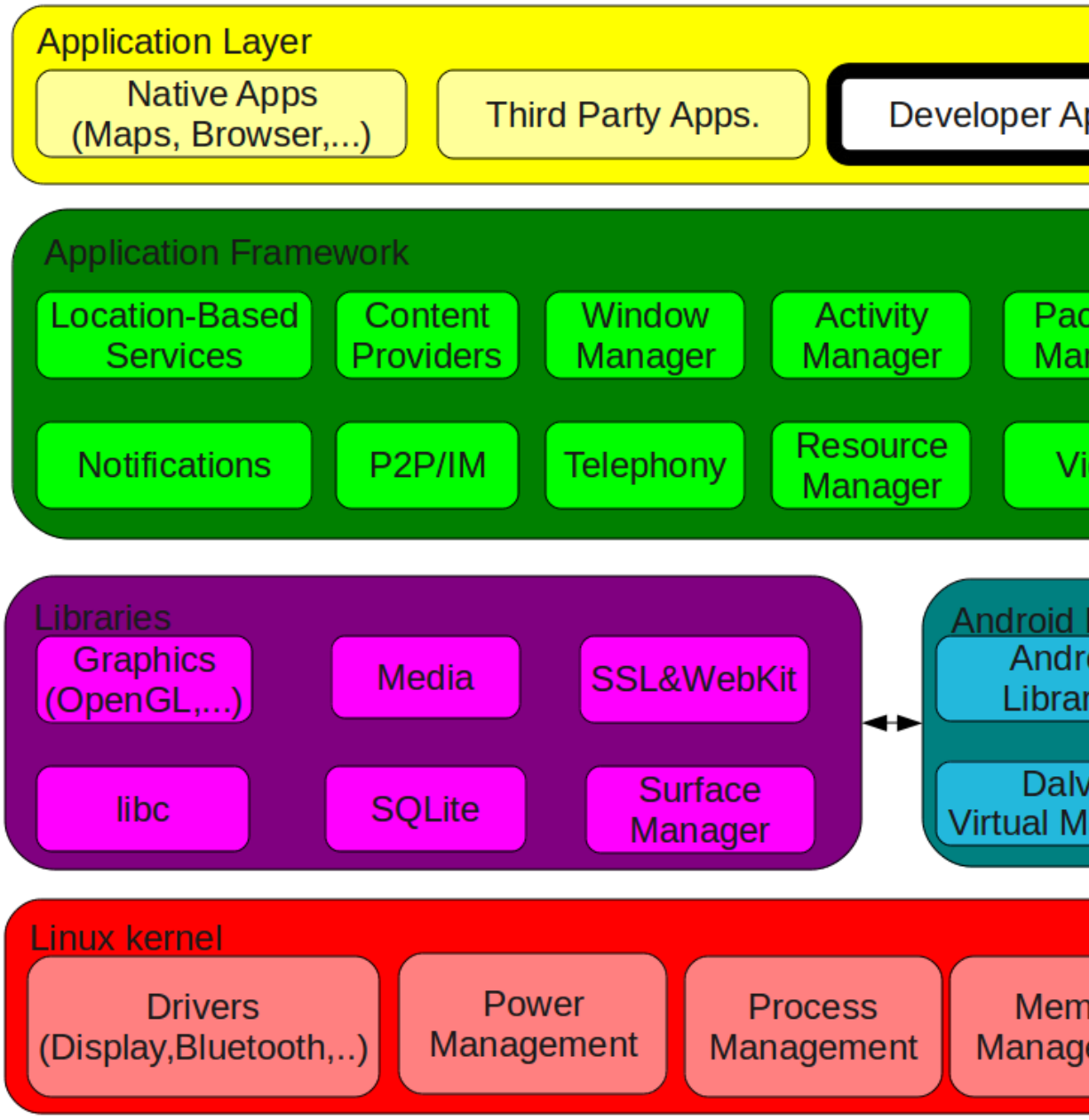
Esempio:

```
<html>
<head>
<script type="text/javascript" src="prototype.js"></script>
</head>
<body>
<script language="javascript">
Event.observe(window,'load',init);

function init(){
$('greeting-submit').style.display = 'none';
Event.observe('greeting-name','key-up',greetings);
}
function greetings(){
var myAjax = new Ajax.Updater('greeting','servlet',{method:'get',parameters:{ par:
$F('greeting-name') }});
return false;
}
</script>
<form method="get" action="servlet" id="greeting-form" onSubmit="return
greetings();">
<div>
<label for ="greeting-name">Enter your name:</label>
<input id="greeting-name" name="greeting-name" type="text" />
<input id="greeting-submit" name="greeting-submit" type="submit" value="Greet me!" />
</div>
<div id="greeting"></div>
</form>
</body>
</html>
```

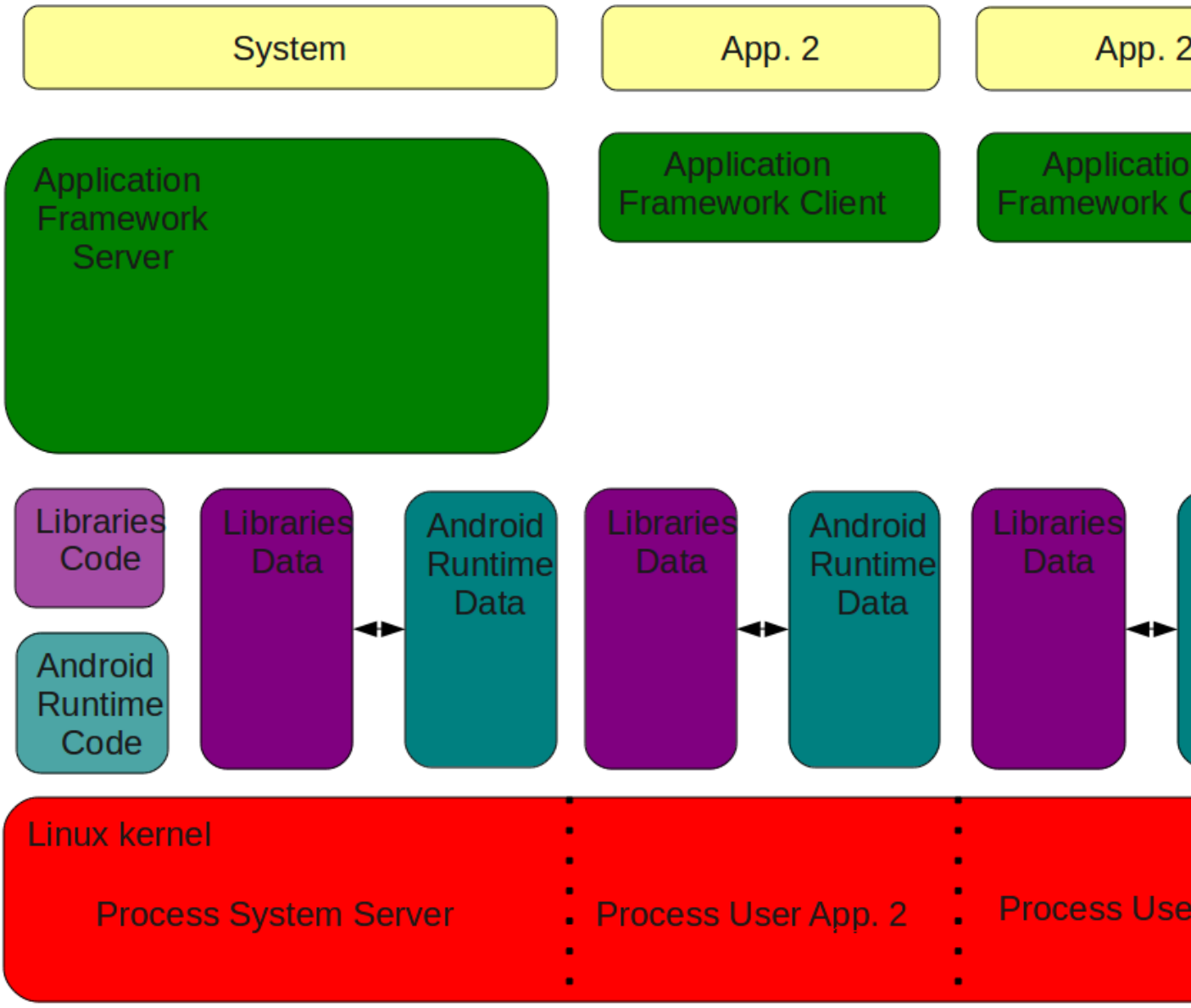
Lezione 19 – Cenni Sviluppo applicazioni su Android

Introduzione

Sebbene la programmazione sul sistema Android avvenga in Java, non viene utilizzata la J2ME per ragioni economiche e di performance. Al posto della V.M. per Java viene usata la Dalvik V.M.
Tale virtual machine ha le seguenti caratteristiche:
1) non accetta file .class ma un unico pacchetto .apk con un .dex che contiene tutte le classi. In questo modo di ottimizzano le dimensioni del file (no ripetizioni costanti, no ripetizione header etc.). Non è di contro possibile avere il caricamento dinamico delle classi, le classi caricabili sono solo quelle presenti nel .dex.
2) il bytecode è orientato ai registri (e non allo stack come la Java V.M.) per una maggiore velocità di esecuzione e compattezza del codice (a scapito di una maggiore difficoltà di compilazione e più basso riuso);
In questo modo si ottiene mediamente una riduzione del 50% della dimensione del bytecode prodotto e una diminuzione del 30% del numero di istruzioni da eseguire.
Inizialmente si pensava che bastasse implementare un gran numero di funzionalità in modo nativo e che potessero essere invocate staticamente per ottenere buone prestazioni. Per aumentare le performance dalla versione 2.2 di Android è previsto anche la compilazione Just In Time a granularità di Trace che ha queste caratteristiche rispetto a un JIT a granularità di metodo: 1) utilizzo minimo di memoria aggiuntiva 2) tempo veloce di avvio a scapito di una minore ottimizzazione (e possibilità di condivisione tra processi).
Per aumentare la velocità di startup delle DVM su cui girano i vari processi, il caricamento della DVM stessa deve essere molto veloce. Per questo si è usato una tecnica detta Zygote che permette la condivisione e il precaricamento di tutte le librerie core.



Per aumentare la sicurezza ogni applicazione gira su una propria istanza della VM che ha bisogno del proprio processo (analogamente alla VM standard di Sun). Per aumentare la sicurezza ogni processo diverso viene eseguito sotto le credenziali di un utente diverso e il kernel linux controlla i vincoli di accesso tra i vari processi.



In conclusione Android sfrutta il Kernel linux ma non è un sistema linux. Usa java come linguaggio ma le API che usa non sono quelle standard.

SDK

Si procede innanzitutto all'installazione della SDK:

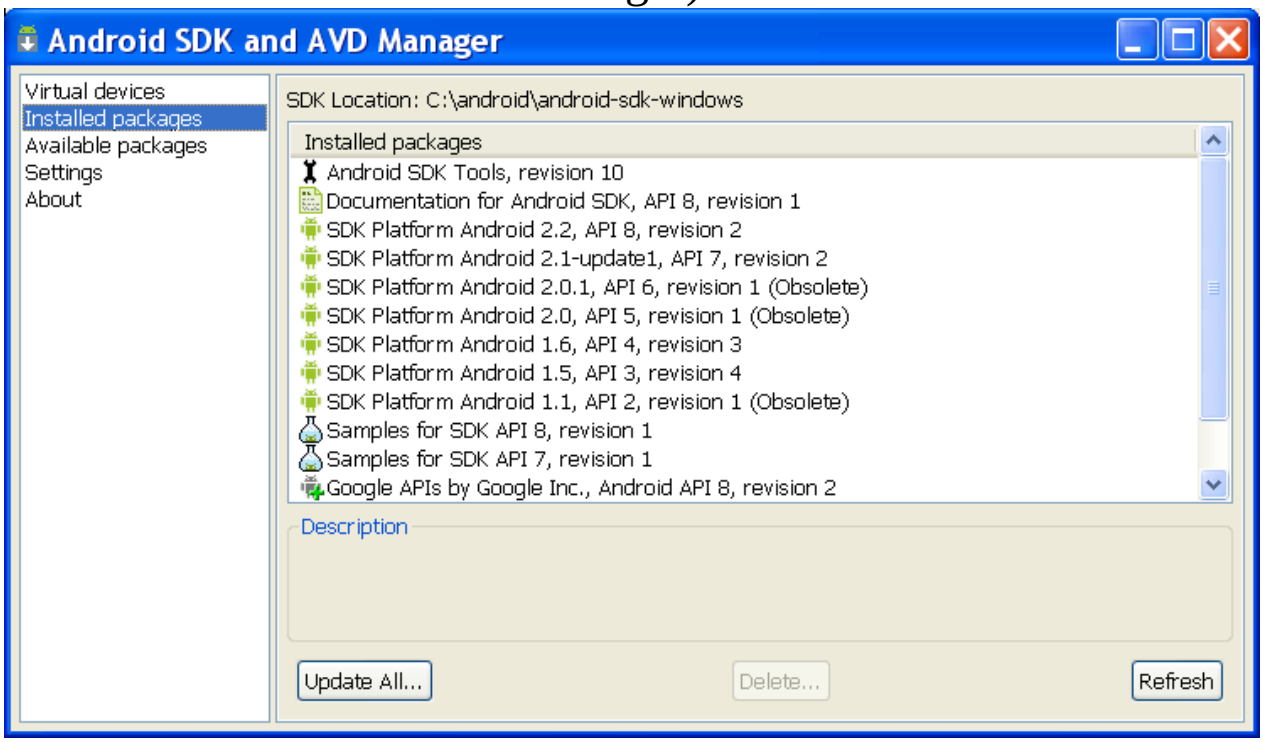
1. La SDK per lo sviluppo su Android richiede la preventiva installazione della SDK di Java.
2. Una volta installato Java, possiamo procedere all'installazione della SDK Started Package per Android
3. Opzionalmente ma vivamente consigliato e assunto nei seguito, si

può procedere all'installazione del plugin ADT per Eclipse

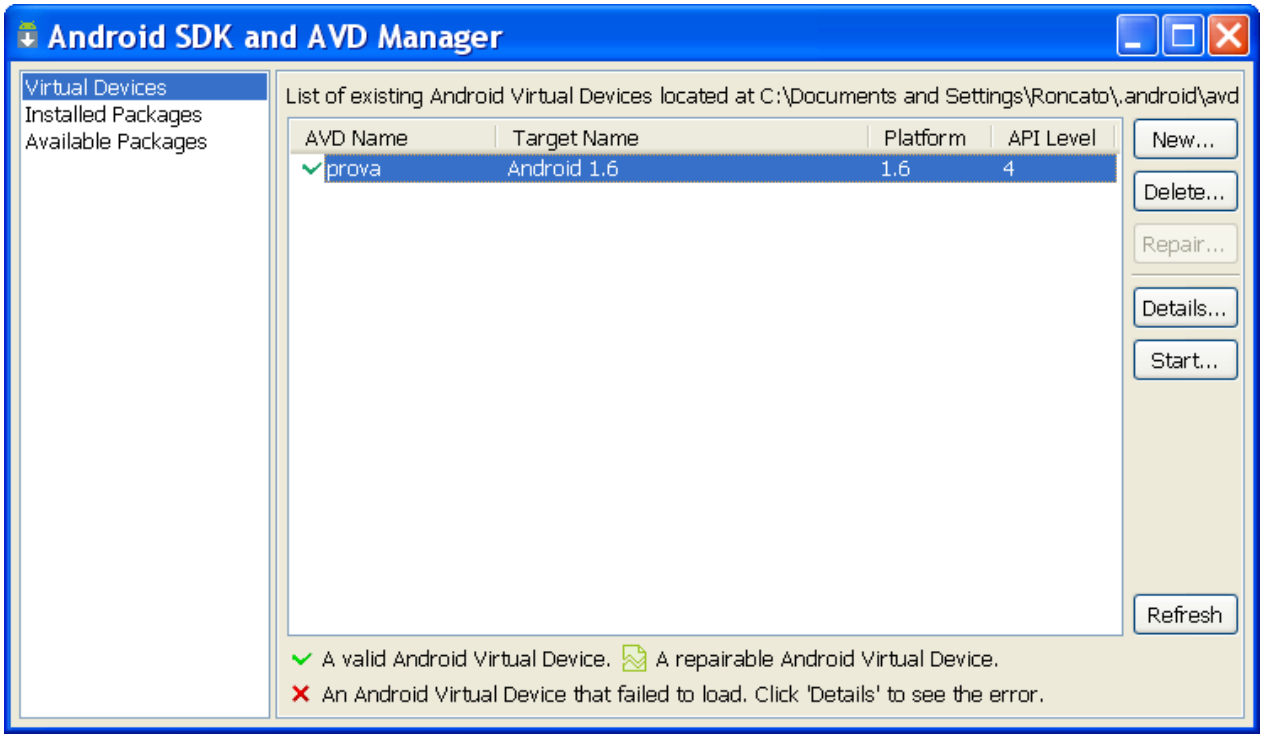
4. Successivamente si possono aggiungere altre piattaforme e componenti alla SDK di Android tramite l'Android SDK and AVD Manager. A esempio, possono essere installate varie versioni della SDK, pacchetti di documentazione o varie librerie. Consiglio di utilizzare le versioni più vecchie delle SDK in quanto più leggere.

Maggiori dettagli sono disponibili a link:
http://developer.android.com/sdk/installing.html

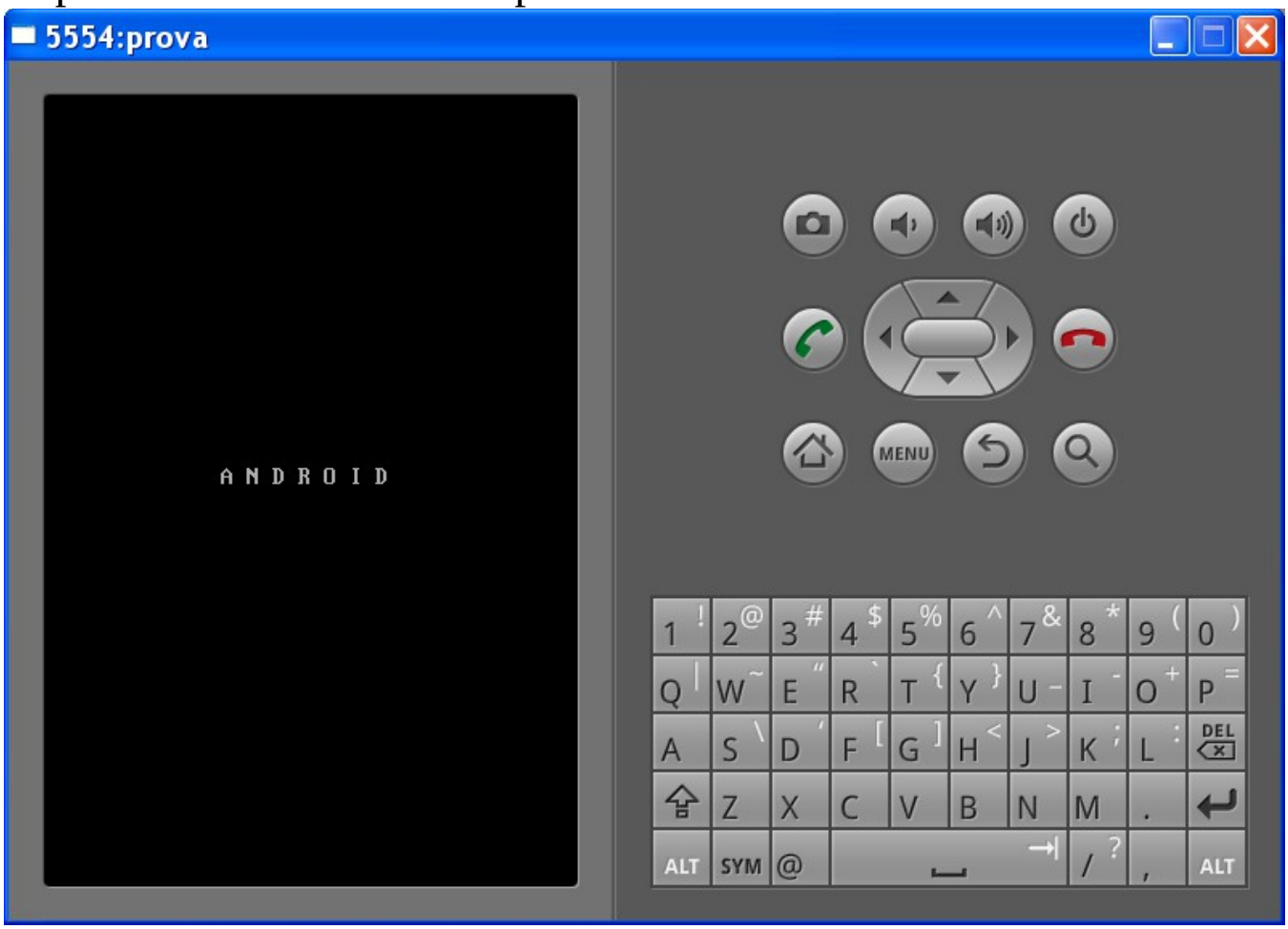
Aspetto SDK Manager dopo l'aggiunta delle piattaforme (Window ->Android SDK and AVD Manager):



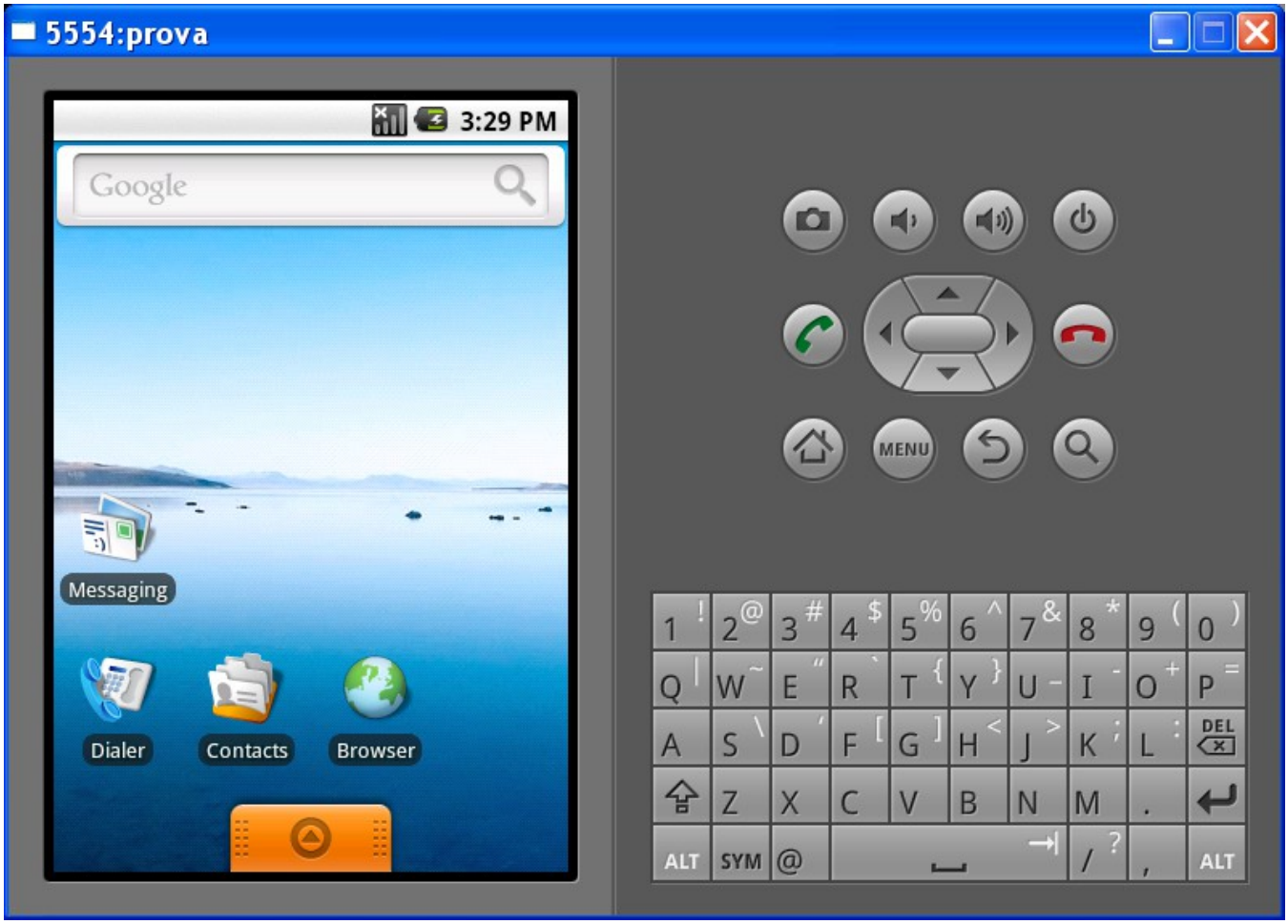
La SDK contiene anche un emulatore di terminare mobile con il quale è possibile selezionare il tipo di versione dell'API supportata dal terminale più la presenza o meno di vari dispositivi hardware (es GPS, accelerometri, quantità di memoria etc.)
Lista simulatori creati:



Aspetto del Simulatore alla partenza:



Aspetto simulatore in attesa esecuzione applicazione:



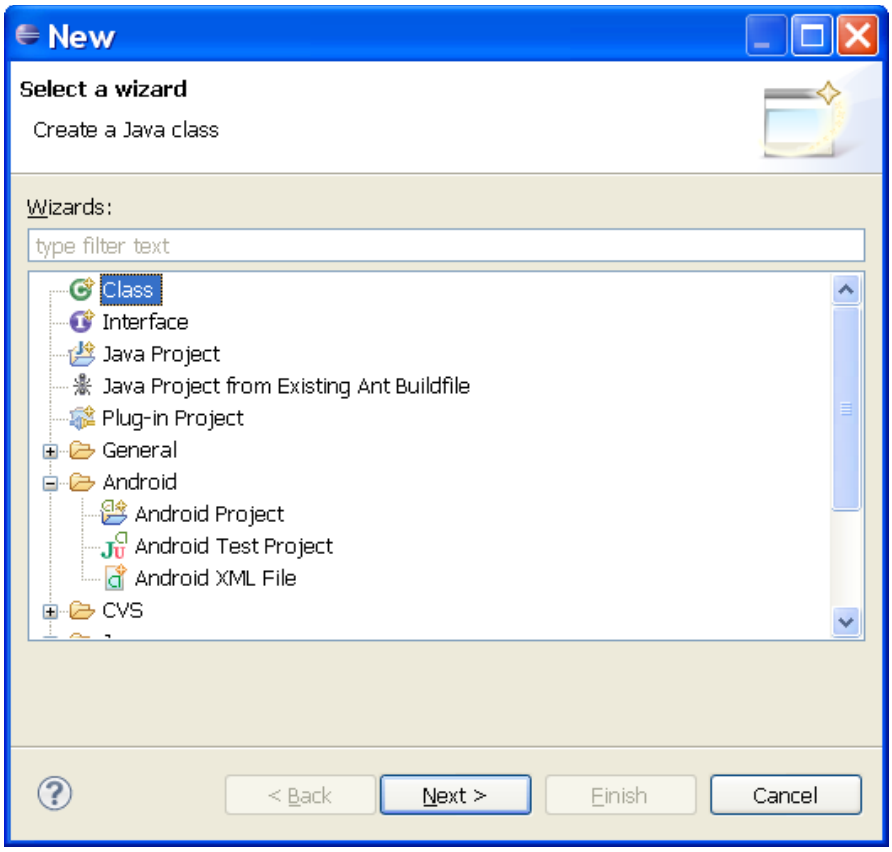
Architettura

Architettura orientata al riuso dei componenti ottenuto tramite file xml con la descrizione dell'aspetto dell'applicazione, file di risorse esterni al codice Java.
I componenti di un'applicazione possono essere:

- Activity: schermate visibili
- Services: background services
- Content provider: dati condivisi
- Broadcast receiver: ricevono e reagiscono a eventi in broadcast
- Intent: componenti che attivano altri componenti

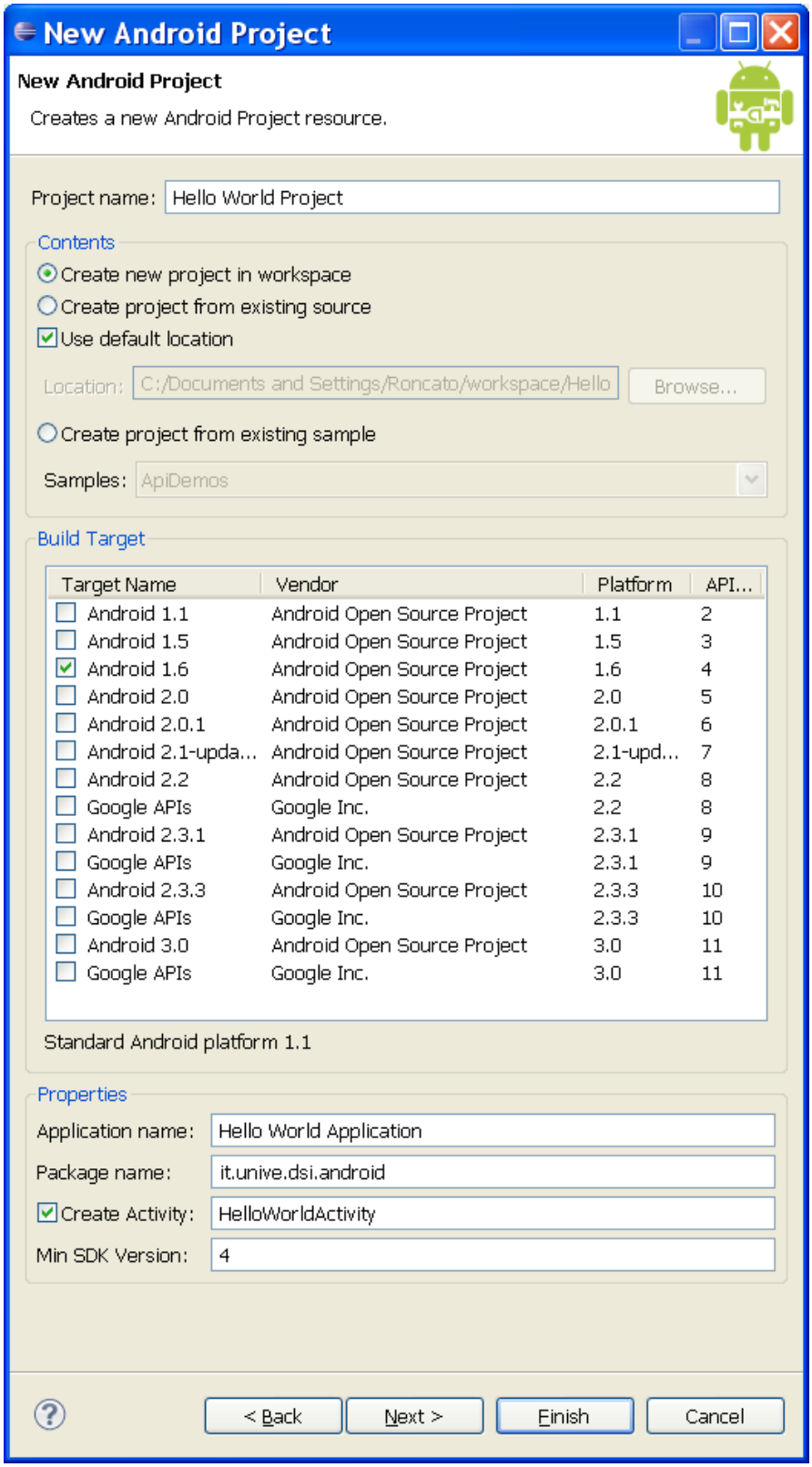
Esempio Applicazione

Da Eclipse selezionare New Other e quindi "Android Project".

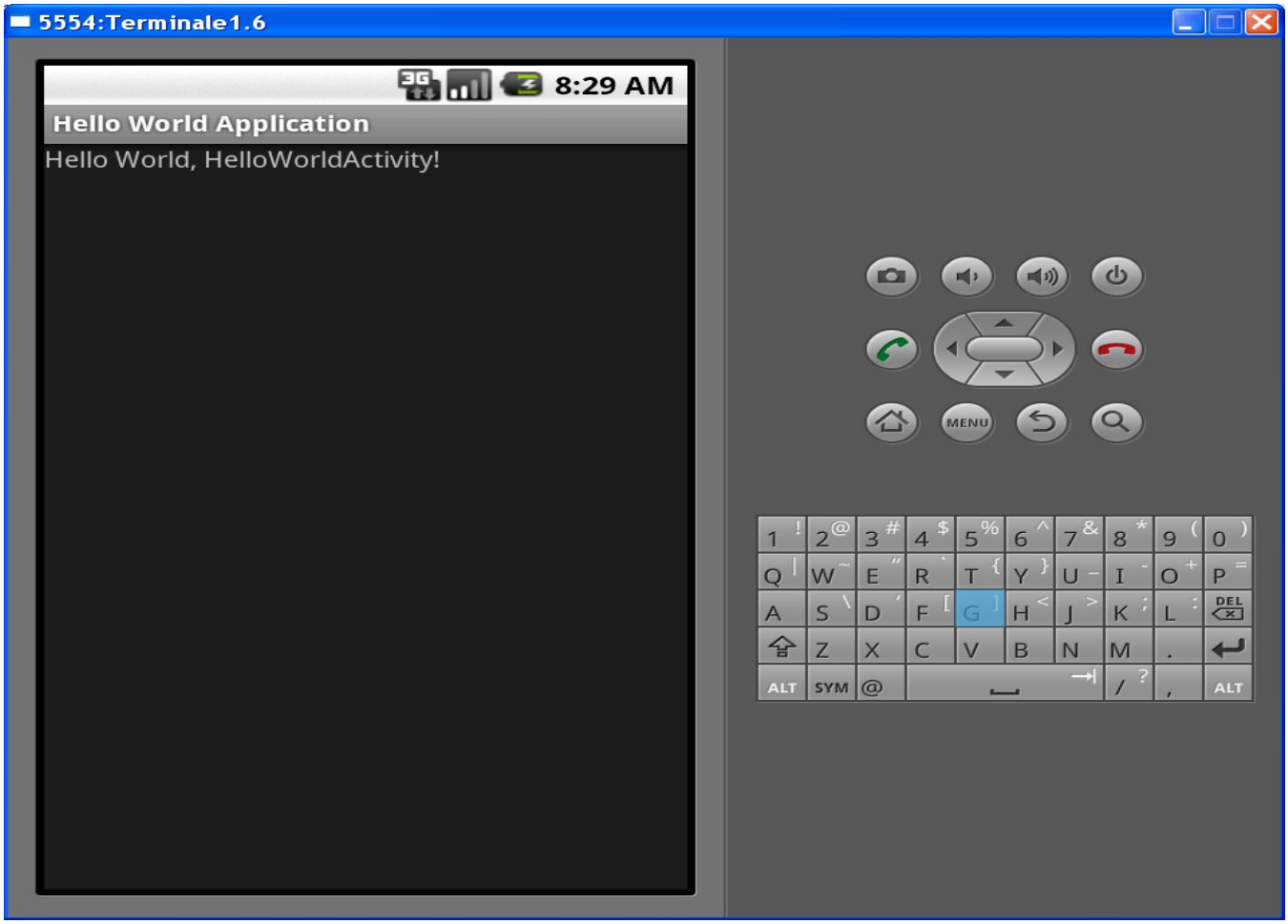


Successivamente compilare:

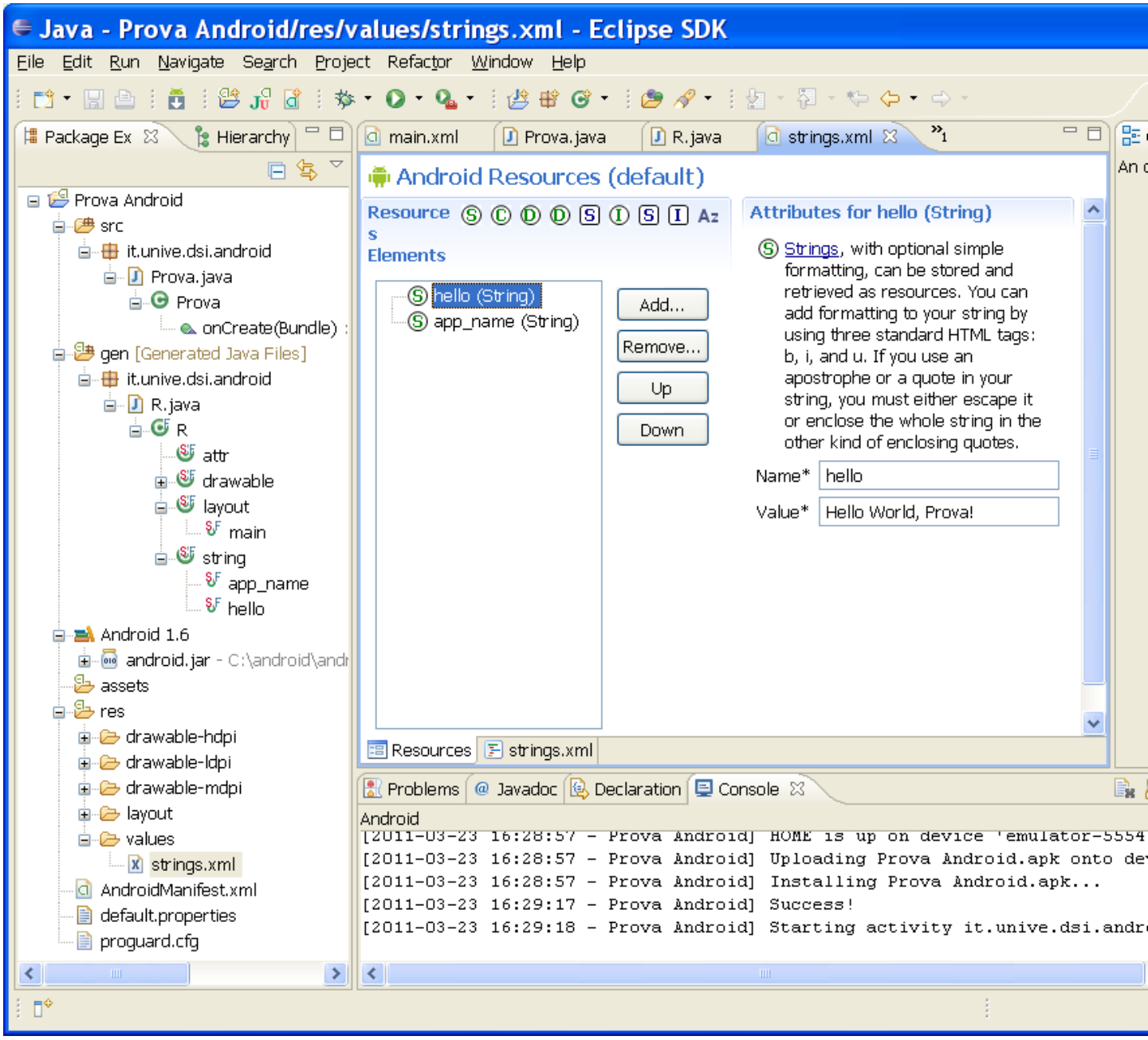
1) nome progetto
2) selezionare la/le piattaforma/e su cui far girare l'applicazione
3) il nome dell'applicazione
4) il package
5) il nome dell'attività
6) eventualmente la minima piattaforma su cui deve girare l'applicazione.



A questo punto viene creato in automatico tutto il necessario per la tipica applicazione Hello World. Se la facciamo girare nel simulatore fatto partire precedentemente otteniamo:



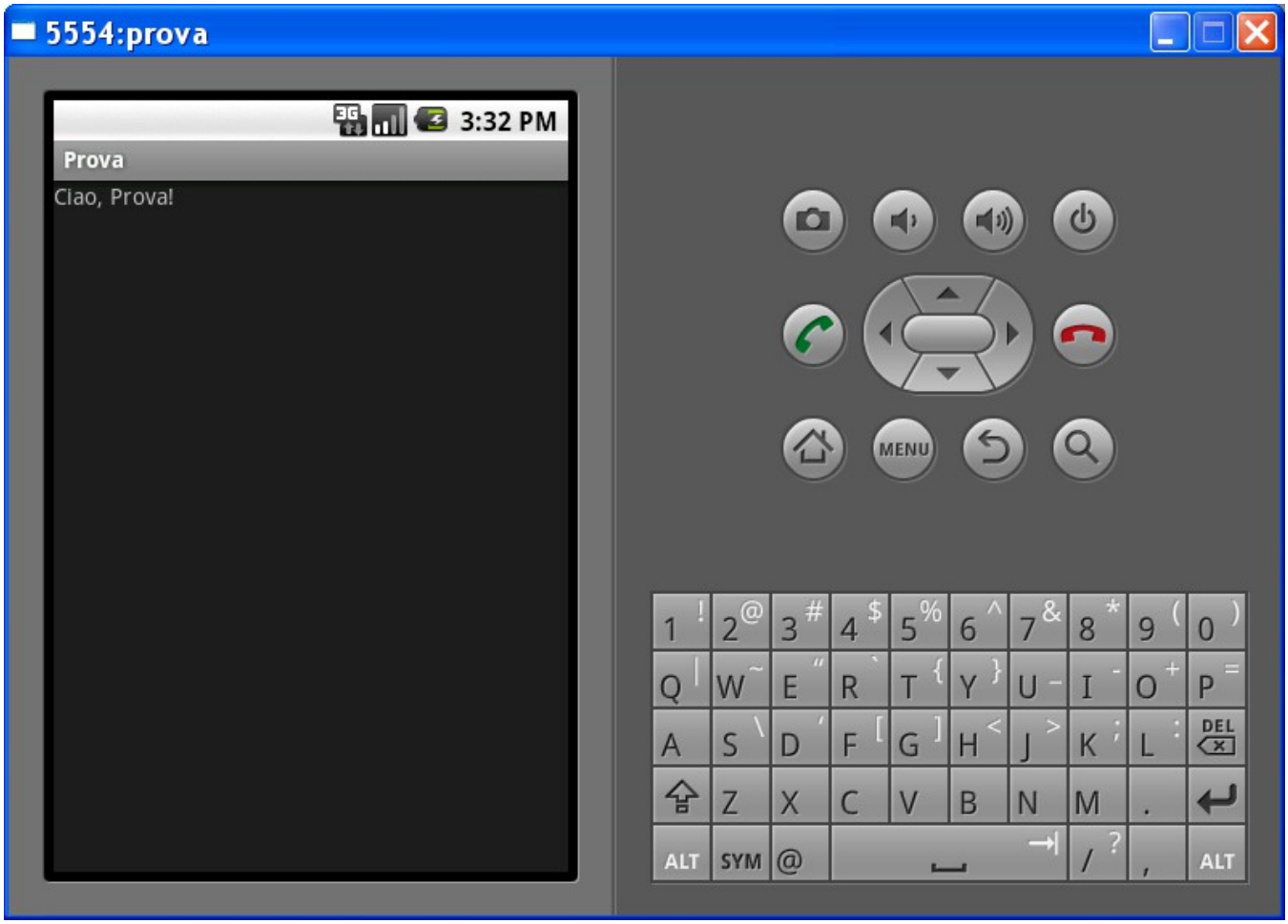
Senza bisogno di modificare il codice Java ma solo i file di risorse del progetto che contengono le stringhe possiamo modificare il progetto:



Che corrisponde al file xml strings.xml con il seguente contenuto:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="hello">Hello World,
HelloWorldActivity!</string>
   <string name="app_name">Hello World
Application</string>
</resources>
```

E cambiando la stringa in "Ciao, Prova!" otteniamo il seguente risultato:



Codice Java:

```
package it.unive.dsi.android;
import android.app.Activity;
import android.os.Bundle;
public class HelloWorldActivity extends
Activity {
  /** Called when the activity is first
created. */
  @Override
  public void onCreate(Bundle
savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
}
```

Lezione 20 – Applicazioni Android

Componenti android

- Activity: ogni schermata dell'applicazione,
- Service: attività in background
- Content Providers: fornitori di contenuti condivisi
- Intents: Meccanismo di Message passign
- Broadcas Receiver: Consumatori di Intenti
- Notifications: Notificazioni agli utenti

Manifesto dell'applicazione

Il file AndroidManifest.xml permette di definire la struttura e i metadati dell'applicazione.

Include un nodo per ogni componente (Attività, Servizi, Content Providers e Broadcast Receiver) e attraverso gli Intent Filter e i Permission determina come ogni componente interagisce con gli altri e le altre applicazioni.

La root del file xml è il tag “manifest”:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="it.unive.dsi.android"
android:versionCode="1"
android:versionName="1.0"> … </manifest>
```

I tag presenti dentro al manifesto sono:

- application: un solo tag di tipo application che specifica i metadati dell'applicazione(es. `<application android:icon="@drawable/icon" android:label="@string/app_name">`). Questo tag può contenere nidificati i seguenti tag:
- activity: un tag per ogni schermata dell'applicazione (es. `<activity android:name=".Prova" android:label="@string/app_name">`). android.name la classe dell'attività Ogni schermata deve essere dichiarata (altrimenti viene lanciata un'eccezione. Ogni nodo attività supporta il tag intent-filter:
  (es. `<intent-filter> <action android:name="android.intent.action.MAIN" /> <category android:name="android.intent.category.LAUNCHER" /></intent-filter>`) . Gli intent filter indicano quali sono i componenti che possono gestire i vari intenti. In questo caso, si dice che questa attività è quella che può gestire l'intento MAIN di categoria LAUNCHER (ovvero questa attività verrà mostrata quando si fa partire l'applicazione).
- service: un tag per ogni servizio (es .`<service android:name=".MyService" android:enabled="true"></service>`). Anche i service possono avere dei intent-filter.
- provider: un tag per ogni provider
- receiver
- uses-permissions: `<uses-permission android:name="android.permission.INTERNET"></uses-permission>`
- permission:
- instrumentation

Esempio completo file AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="it.unive.dsi.android"
android:versionCode="1"
android:versionName="1.0">
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
<uses-permission
android:name="android.permission.SIGNAL_PERSISTENT_PROCESSES"></uses-permission>


<application android:icon="@drawable/icon" android:label="@string/app_name">
<activity android:name=".Prova"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

</application>
</manifest>
```

Risorse

Esternalizzazione delle risorse

Vantaggi:

1. diventano più facili da mantenere, aggiornare e gestire

2. si possono gestire più facilmente risorse diverse per hardware diversi e l'internazionalizzazione. (in automatico viene scelta la risorsa più idonea in base alla lingua, localizzazione, tastiera, schermo, senza scrivere una linea di codice).

3. Possiamo anche gestire l'aspetto dell'applicazione (layout).

Con Android possiamo gestire l'esternazionalizzazione di stringhe, colori, dimensioni, stili e temi, disegnabili (immagini), layout, animazioni.

Esempio file “string.xml”.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="hello">Ciao, comincia a navigare</string>
<string name="app_name">Chat</string>
<string name="textEdit">Inserisci l\'url qui</string>
</resources>
```

Utilizzo delle risorse nel codice

Le risorse sono accessibili tramite la classe R che viene generata automaticamente a partire dai file xml che contengono le definizioni delle risorse. La classe R contiene sottoclassi statiche per ognuno dei tipi di risorsa per cui e' stata definita almeno una risorsa. Ognuno delle sottoclassi ha come variabili statiche le risorse dichiarate nel rispettivo file xml il cui nome e' lo stesso dell'identificativo della risorsa (attibuto name del tag). Il valore di queste variabili e' un riferimento alla corrispondente locazione nella tabella delle risorse (*identificatore di risorsa*) (e non una istanza della risorsa stessa). Alcune metodi accettano direttamente degli identificatori di risorsa (come setContentView che abbiamo gia' visto).

```
package it.unive.dsi.android;

public final class R {
public static final class attr {
}
public static final class drawable {
public static final int icon=0x7f020000;
}
public static final class id {
public static final int editText1=0x7f050001;
public static final int listView1=0x7f050000;
}
public static final class layout {
public static final int main=0x7f030000;
}
public static final class string {
public static final int app_name=0x7f040001;
public static final int hello=0x7f040000;
public static final int textEdit=0x7f040002;
}
}
```

```
setContentView(R.layout.main);
```

Quando si ha bisogno di un'istanza della risorsa, si usa un metodo opportuno per estrarre l'istanza della risorsa dalla tabella delle risorse.

```
Resources res = getResources();

String s = res.getString(R.string.app_name);

CharSequence cs = res.getText(R.string.app_name);
```

Utilizzo delle risorse in altre risorse

```
attribute= "@packagename:resourcetype/resourceidentifier"
```

oppure se il package è lo stesso dell'applicazione:

```
attribute= "@resourcetype/resourceidentifier"
```

esempio: `android:text="@string/textEdit"`

system resource

```
packagename=android
```

esempio in codice: `getString(android.R.string.selectAll)`

esempio in file risorse: `attribute="@android:string/selectAll"`

Creare risorse per vari linguaggi e hardware

Progetto/

res/

values/

values-it/

values-it-CH/

La lista seguente mostra quali sono i qualificatori con cui e' possibile personalizzare le risorse:

- Mobile Country Code e Mobile Network Code (MCC/MNC)
- lingua
- regione (r minuscola piu' il codice dello stato maiuscolo
- Screen Size One of small (smaller than HVGA), medium (at least HVGA and smaller than VGA), or large (VGA or larger).
- Screen Width/Length Specify long or notlong for resources designed specifically for widescreen (e.g., WVGA is long, QVGA is notlong).
- orientamento dello schermo port/land/square
- densita' di pixel dello schermo pixel per pollice (92dpi)
- Tipo di touchscreen notouch, stylus, finger
- disponibilita' della tastiera keysexposed, keyshidden
- tipo di tastiera: nokeys, querty, 12key
- tipo navigazione notouch, dpad, trackball, wheel
- risoluzione schermo: risoluzione in pixel con la piu' grande per prima (320x240)

Bisogna seguire l'ordine e si possono specificare quelli che servono. Al run time android quando ricerca una risorsa restituisce quella che soddisfa piu' criteri, in caso di parita' sul numero di criteri, si sceglie come importanza seguendo l'ordine della lista.

Cambiamenti di configurazione al runtime

Android gestisce cambiamenti al run time di linguaggio, location e hardware terminando e facendo ripartire l'attività. Quando questo comportamento non ci e' gradito bisogna:

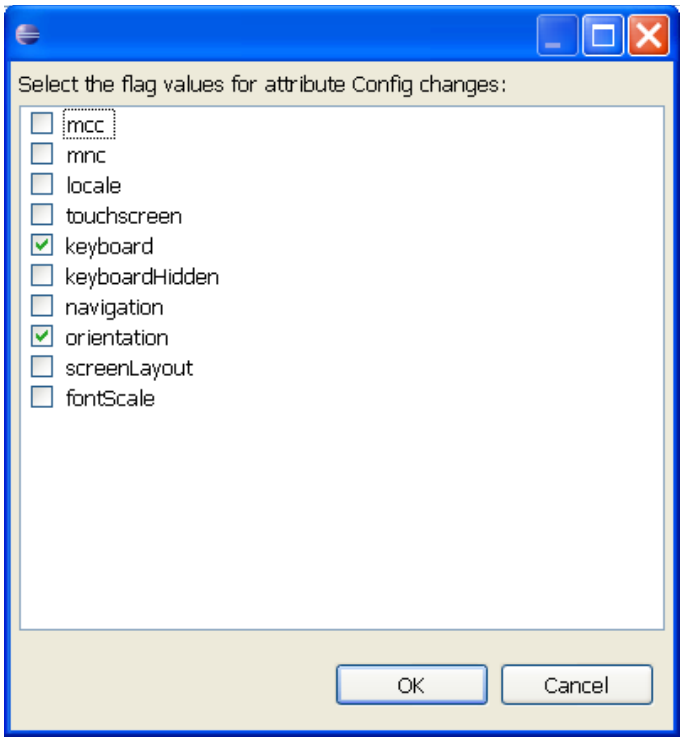
1. nel manifesto aggiungere nell'attivita' l'attributo android:configChanges specificando i cambiamenti di configurazione che si intende gestire:

- orientation
- keyboardHidden
- fontScale
- locale

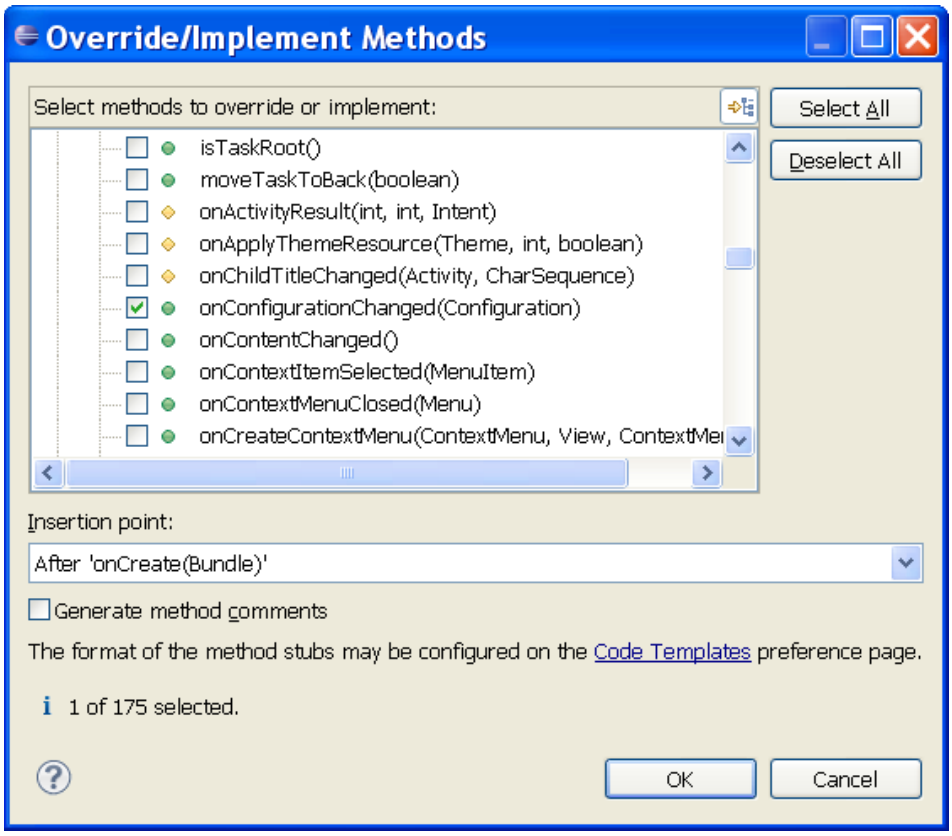
- keyboard
- touchscreen
- navigation

si possono selezionare più eventi da gestire separando i valori con |.

Per esempio `android:configChanges="keyboard|orientation"`



2. sovrascrivere il metodo onConfigurationChanged



```
@Override
public void onConfigurationChanged(Configuration newConfig) {
super.onConfigurationChanged(newConfig);

if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
//orientamento landscape
}
else {
//orientamento portrait
}

if (newConfig.keyboardHidden == Configuration.KEYBOARDHIDDEN_NO) {
//tastiera visibile
}
else {
//tastiera nascosta
}
}
```

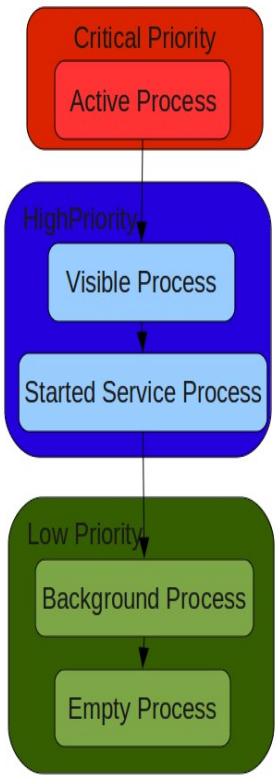
Ciclo di vita delle applicazioni Android

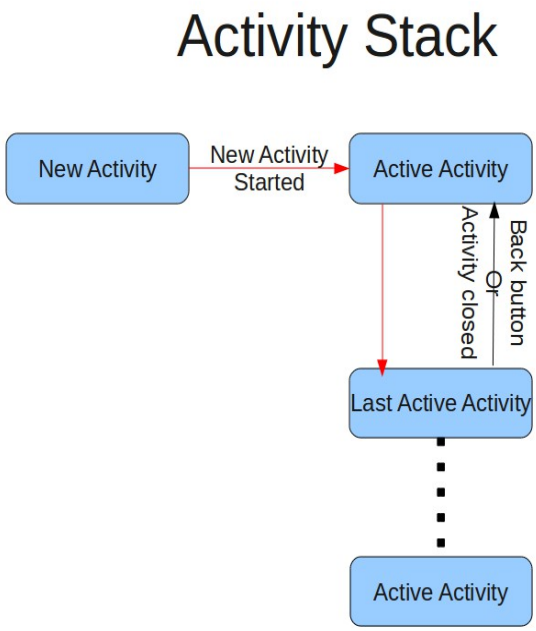
Active Process: processi che ospitano applicazioni con componenti attualmente in iterazione con l'utente. Questi processi sono gli ultimi a essere eliminati. I componenti attualmente in iterazione includono:

- Attività in primo piano e che stanno rispondendo a eventi dell'utente.
- Attività, servizi o broadcast receviver che stanno eseguendo un `onReceive`
- Serivzi che stanno eseguendo `onStart`, `onCreate` o `onDestroy`

Visible Process: processi che ospitano applicazioni visibili ma inattive cioè che ospitano attività visibili ma che non interagiscono con l'utente. (attività parzialmente oscurate da attività visibili)
Started Service Process: processi che ospitano servizi in esecuzione
Background Process: processi che ospitano attività che non sono visibili e che non hanno servizi che sono in esecuzione che vengono eliminati iniziando dall'ultimo che e' stato “visto”.
Empty Process: processi che sono terminati ma di cui Android mantiene l'immagine della memoria dell'applicazione per farla ripartire più velocemente in caso venga richiesta nuovamente.

Process Priority

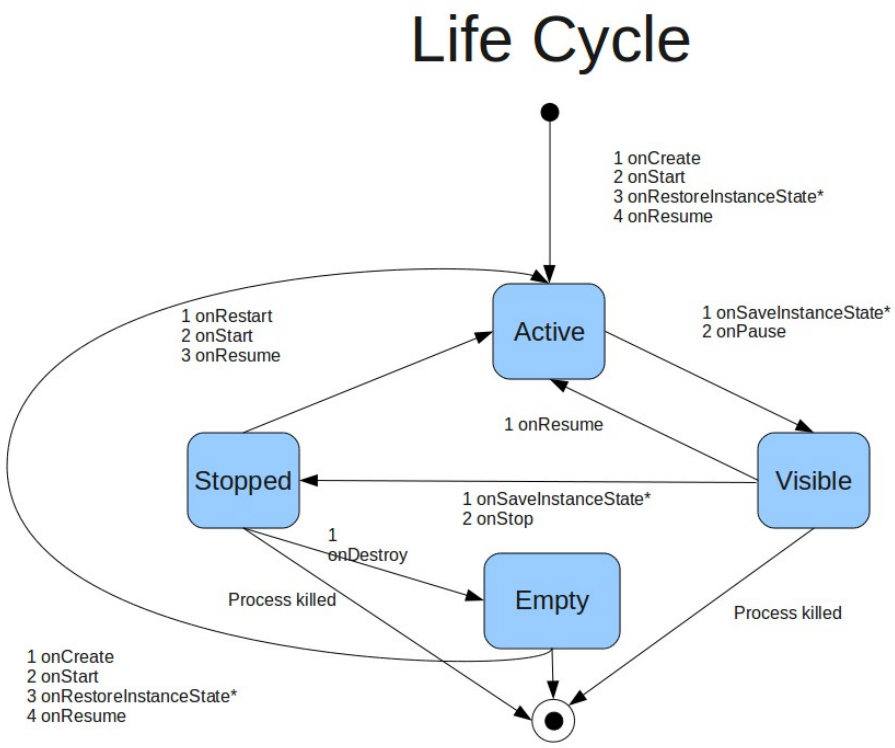




Ciclo di vita delle Attività

Lezione 21 – Applicazioni Android

Ciclo di vita delle Attività



Quando si sovrascrive uno dei metodi elencati in figura, come prima cosa bisognerebbe sempre chiamare il metodo sovrascritto.

Approfondimento sul ciclo di vita

Durante l'intero tempo di vita dell'attività che va dalla creazione alla distruzione, l'attività stessa entra una o più volte negli stati visibile e attivo. Ogni transizione da uno stato a un altro fa scattare l'invocazione dei gestori (handler) descritti nella figura. In particolare:

Il tempo di vita

Il tempo di vita dell'attività inizia alla prima chiamata del metodo `onCreate` e finisce all'ultima chimata del metodo `onDestroy`. E' possibile in alcuni casi che il processo dell'attività venga terminato senza una chiamata al metodo `onDestroy`.
Il metodo onCreate va utilizzato per inizializzare l'attivita: gonfiare l'interfaccia utente (ovvero estrarla dalla classe R per renderla visibile), allocare oggetti (in modo da poterli riusare in tutto il ciclo di vita), legare i dati ai relativi controlli e creare servizi e thread. Al metodo `onCreate` è passato un oggetto Bundle che contiene lo stato dell'interfaccia utente salvato dal metodo `onSaveInstanceState`. Nel metodo `onCreate` o nel metodo **`onRestoreInstanceState`** bisognerebbe sempre usare questo oggetto per ricreare la precedente interfaccia utente.

La sovrascrittura del metodo `onDestroy` dovrebbe rilasciare le risorse create da onCreate e assicurarsi che tutte le connessioni (connessioni di rete, connessioni a DB etc.) vengano chiuse.
Le linea guida per lo sviluppo di applicazioni su android raccomandano evitare la continua creazione e distruzione di oggetti (garbage collector). Per questa ragione si suggerisce di creare gli oggetti di utilità

all'attività nel metodo `onCreate` e di riusarli nelle chiamate degli altri metodi.

Il tempo di visibilità

Il tempo di visibilità di un'attività inizia con la chiamata al metodo `onStart` e finisce con la chiamata al metodo `onStop`. Tra queste due chiamate l'attività è visibile all'utente anche se può non avere il fuoco ed essere parzialmente oscurata. Le attività possono diventare più volte visibili, anche se non probabile, Android può terminare attività nello stato visibile senza invocare il metodo `onStop`.
Il metodo `onStop` dovrebbe essere usato per mettere in pausa o fermare le animazioni, i theread, i listner dei sensori, il lookup del GPS, i timer, i servizi e ogni altro processo che viene usato esclusivamente per aggiornare l'interfaccia. (non serve a niente aggiornare un'interfaccia utente che non è vista dall'utente).
Si usa invece il metodo `onStart` o `onRestart` per riesumare i task fermati da `onStop`.
Il metodo `onRestart` è chiamata immediatamente prima del metodo `onStart` (a parte la prima volta in cui non viene chiamato). Quindi può essere usato per implementare eventuali comportamenti particolari quando l'attività ritorna visibile dopo essere stata oscurata.
I metodi `onStart/onStop` sono anche usati per registrare/cancellare i Broadcast Receiver che sono usati esclusivamente per aggiornare l'interfacia utente.

Il tempo di attività

Il tempo di attività inizia con la chiamata al metodo `onResume` e termina con la chiamata al metodo `onPause`.
Una Attività attiva è in primo piano e sta ricevendo l'input dall'utente. Un'attività probabilmente entrerà nello stato attivo più volte nel corso del suo tempo di vita in quanto verrà disattivata ogni volta che: 1) una nuova attività diventa attiva; 2) il dispositivo va in sleep; 3) l'attività perde il focus. Quindi bisogna cercare di rendere il codice dei metodi `onResume` e `onPause` relativamente veloci e leggeri.
Proprio prima di `onPause` viene chiamato il metodo **`onSaveInstanceState`**. Questo metodo deve provvedere a salvare lo stato dell'interfaccia utente nell'oggetto Bundle (che sarà poi passato ai metodi `onCreate` e `onRestoreInstanceState`**)**. Quindi in `onSaveInstanceState` di dovrebbe salvare lo stato dell'interfaccia (come lo stato dei checkbox, il focus dell'utente, il testo inserito ma non salvato etc.) in modo che l'utente si ritrovi la stessa interfaccia quando l'attività ritornerà attiva. **Si può assumere che i metodi `onSaveInstanceState` e `onPause` vengano chiamati prima che il processo dell'attività venga distrutto**. Quindi è importante sovrascrivere almeno uno di questi due metodi per salvare le modifiche perché rappresentano un metodo sicuro che viene chiamato prima della distruzione del processo.
Il metodo `onResume` normalmente dovrebbe essere abbastanza leggero e si occupa della sola riregistrazione di quei Broadcast Receivers che sono stati deregistrati dal metodo `onPause` (mentre il recupero dell o stato dell'interfaccia è garantito dalle chiamate dei metodi `onCreate` e `onRestoreInstanceState`).

Le Classi Activity speciali

La SDK Android include anche alcune sottoclassi di Attività che sono di comune utilizzo, alcune delle più usate sono:

- **MapActivity**: include la gestione delle risorse necessarie per visulizzare un mappa;
- **ListActivity**: una classe contenitore per attività che gestisce una ListView collegata ad una sorgente dati e che espone handler per ricevere eventi di selezione degli elementi;
- **ExpandableListActivity**: simile alla ListActivity ma espandibile;
- **TabActivity**: permette di includere più Attività o View in un singolo schermo con la possibilità di usare il tab per cambiare Attività o View selezionata.

Classe Applicazione

Nelle applicazion android possiamo sfruttare anche la classe `Application` per gestire meglio il ciclo di vita dell'intera applicazione. Estendendo la classe `Application` possiamo:

1. Mantenere lo stato dell'applicazione
2. Trasfererie oggetti tra componenti dell'applicazione (per esempio tra due attività della stessa applicazione)

Quando la sottoclasse di `Application` viene registrata nel manifesto, ne viene creata un'istanza

quando il processo dell'applicazione viene creato. Per questa ragione questo oggetto si presta a essere gestito come singleton.
Una volta creata la classe che gestisce il ciclo di vita dell'applicazione bisogna registrare la classe nel manifesto specificando nel tag application il nome della classe nell'attributo `android:name`. Es.

```
<application android:icon="@drawable/icon"
android:name="MyApplication">
[... Manifest nodes ...]
</application>
```

Eventi del ciclo di vita dell'applicazione

La classe Application fornisce anche degli handler per gestire gli eventi principali del ciclo di vita dell'applicazione. Sovrascrivendo questi metodi si può personalizzare il comportamento dell'applicazione in queste circostanze:

- `onCreate`: è chiamato quando l'applicazione è creata e può essere usato per inizializzare il signelton e per creare e inizializzare tutte le variabili di stato e el risorse condivise.
- `onTerminate`: può essere chiamato (ma non c'è garanzia che venga chiamato se per esempio il processo viene terminato per liberare risorse) quando l'applicazione viene terminata.
- `onLowMemory`: permette di liberare della memoria quando il sistema ne ha poca disponibile. Viene generalmente chiamato quando i processi in secondo piano sono stati già terminati e l'applicazione attualmente in primo piano ha ancora bisogno di memoria.
- `OnConfigurationChanged`: viene chiamato quando c'è un cambiamento della configurazione. Al contrario delle attività che vengono terminate e fatte ripartire, all'applicazione viene notificato il cambiamento con la chiamata a questo metodo.

```
public class MyApplication extends Application {

private static MyApplication singleton;

// Returns the application instance
public static MyApplication getInstance() {
return singleton;
}

@Override
public final void onCreate() {
super.onCreate();
singleton = this;
...
}

@Override
public final void onTerminate() {
super.onTerminate();
...
}

@Override
public final void onLowMemory() {
super.onLowMemory();
...
}

@Override
public final void onConfigurationChanged(Configuration newConfig) {
super.onConfigurationChanged(newConfig);
...
}
}
```

Layout

I Layout sono delle classi che estendono la classe `ViewGroup` e permettono di gestire la disposizione di altri componenti detti componenti figli (estensione della classe View). I Layout possono essere nidificati permenttendo in questo modo la creazione di interfacce molto complesse.
Si può specificare il tipo di layout che è la root del documento. Tipi validi di layout predefiniti sono:

- `FrameLayout`: il più semplice fra i Layout che visualizza tutti i componenti figli nell'angolo in alto a sinistra. Se si aggiungono più figli, ogni nuovo figlio va a finire sopra il precedente nascondendo il precedente.
- `LinearLayout`: Allinea tutti i figli in una linea orrizontale o in verticale. Un Layout verticale ha una colonna di componeti, mentre un layout orrizontale ha una riga di componenti.
- `RelativeLayout`: il più flessibile dei Layout predefiniti permette di definire la posizione di ogni figlio relativamente agli altri e al bordo dello schermo.
- `TableLayout`: questo Layout permette di disporre i componenti in una griglia di righe e colonne.
- `Gallery`: Una Gallery visualizza una singola riga di componenti in una lista scorrevole orizzontalmente.

E' anche possibile creare controlli composti che estendono questi Layout predefiniti.

Esempio file layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello"
/>
<ListView android:layout_height="wrap_content"
android:id="@+id/listView1"
android:layout_width="fill_parent"></ListView>
<EditText android:layout_height="wrap_content"
android:id="@+id/editText1"
android:layout_width="fill_parent"
android:text="@string/textEdit"></EditText>
</LinearLayout>
```

Notiamo che per ogni elemento vengono usate le costanti `wrap_content` e `fill_parent` piuttosto che l'altezza o larrghezza esatta in pixel. In questo modo sfruttiamo al massimo la tecnica che ci permette di definire layout indipendente dalla dimensione dello schermo.
La costante `wrap_content` imposta la dimensione del componente al minimo necessario per contenere il contentuto del componente stesso mentre la costante `fill_parent` espande il componente in modo da riempire il componente padre.
Definire i Layout nei file xml diasaccoppia lo strato di presentazione dal codice Java e permette anche di creare Layout diversi in funzione dei vari hardware che sono caricati dinamicamente senza bisogno di modifiche al codice.

E' anche possibile implementare i Layout nel codice Java. In questo caso è buona norma utilizzare `LayoutParameters` da impostare con `setLayoutParams` method o passandoli al metodo `addView`.

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);
TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);
myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");
int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight,
lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight,
lWidth));
setContentView(ll);
```

Alcuni dei componenti predefiniti che possono essere visualizzati in un Layout (ma non solo):

- `TextView`: Una etichetta di testo in sola lettura. Supporta più linee, la foramttazione e il word wrapping..
- `EditText`: Un rettangolo per editare testo anche su più linee.
- `ListView`: Una lista verticale di elementi View.
- `Spinner`: Un componente composito che visualizza un TextView e una ListView associata che una volta selezionato un elemento della lista viene copiato nel TextView.

- `Button`: un pulsante.
- `CheckBox`: un pulsante a due stati che rappresenta lo stato tramite un quadrato che può essere spuntato o meno.
- `RadioButton`: un pulsante a due stati che se raggruppato a un numero arbitrario di altri RadioButton sono gestiti in modo tale che al massimo uno solo è selezionato.
- `ViewFlipper`: A ViewFlipper permette di disporre una collezzione di View in orizzontale in cui solo una View è visibile in ogni momento (la transizione traView è animata).

E' possibile personalizzare i componenti come vedremo nella prossima lezione.

Lezione 22 – Esempi personalizzazione View e introduzione Intenti

Esempio

Mettendo insieme tutti i file xml visti finora (manifesto, string, layout) possiamo scrivere la seguente attività che simula l'analoga applet vista a lezione.

```
package it.unive.dsi.android;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import android.app.Activity;
import android.os.Bundle;
import android.view.KeyEvent;
import android.view.View;
import android.widget.*;
public class Prova extends Activity {
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ListView list = (ListView) findViewById(R.id.listView1);
    final EditText text = (EditText) findViewById(R.id.editText1);
    final ArrayList<String> strings = new ArrayList<String>();
    final ArrayAdapter<String> aa =
       new ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_1, strings);
    list.setAdapter(aa);
    text.setOnKeyListener(new View.OnKeyListener() {
      public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (event.getAction() == KeyEvent.ACTION_DOWN)
          if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER
              || keyCode == KeyEvent.KEYCODE_ENTER) {
            try {
              URL url = new URL(text.getText().toString());
              URLConnection connection = url.openConnection();
              HttpURLConnection httpConnection = (HttpURLConnection)
connection;
              int responseCode = httpConnection.getResponseCode();
              if (responseCode == HttpURLConnection.HTTP_OK) {
                InputStream in = httpConnection.getInputStream();
                BufferedReader d = new BufferedReader(new
InputStreamReader(in));
                String res = d.readLine();
                while (res != null) {
```

```
                strings.add(res);
                res = d.readLine();
              }
            }
            else stringa.add("NO HTTP_OK");
          } catch (MalformedURLException e) {
            strings.add(e.getMessage());
          } catch (IOException e) {
            strings.add(e.getMessage());
          }
          if (strings.size() > 5) strings.remove(0);
          strings.notifyDataSetChanged();
          text.setText("");
          return true;
        }
        return false;
      }
    });
  }
```

Esempio personalizzazione Componenti

E' possibile personalizzare i componenti :

1. **estendendo** la classe del componente che li definisce (es. `TestView`)
2. **raggrupare** più controlli in modo che si comportino come un'unico nuovo controllo estendendo una classe dei `Layout`
3. anche partendo da zero **estendendo** la classe `View`. I metodi che devono essere riscritti sono:

- costruttore con parametro `Context` : chiamato dal codice Java.
- Costruttore con parametri `Context` e `Attributes`: richiesto dal gonfiaggio dell'interfaccia dal file di risorse. Anche il costruttore con parametri `Context`, `Attributes` e `int` con lo stile di default.
- `onMeasure` per calcolare le dimensoni del componente e impostarle tramite chiamata al metodo `setMeasuredDimension`.
- `OnDraw` per disegnare l'aspetto del componente;

Esempio personalizzazione con estensione TextView

```
public class MyTextView extends TextView {

Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
//creazione da codice
public MyTextView (Context context) {
super(context);
}
//gonfiaggio da resource file
public MyTextView (Context context, AttributeSet ats, int defStyle)
{
super(context, ats, defStyle);
}
```

```
//gonfiaggio da resource file
public MyTextView (Context context, AttributeSet attrs) {
super(context, attrs);
}

@Override
public void onDraw(Canvas canvas) {
// disegni che vanno sotto il testo
canvas.drawText("sotto", 0, 0, paint);
// onDraw della classe TextView disegna il testo
super.onDraw(canvas);
//disegni che vanno sopra il testo
canvas.drawText("sopra", 0, 0, paint);
}

@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
//personalizzare il comportamento del controllo
if (keyEvent.getEventTime()%2==0)
return true;

// e poi chiamare il metodo della superclasse
return super.onKeyDown(keyCode, keyEvent);
}
}
```

Esempio personalizzazione con estensione da View

```
public class MyView extends View {

Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);

// necessario per la crezione da codice
public MyView(Context context) {
super(context);
}

// necessario per il gonfiaggio da resource file
public MyView (Context c, AttributeSet ats, int defStyle) {
super(c, ats, defStyle );
}

// necessario per il gonfiaggio da resource file
public MyView (Context context, AttributeSet attrs) {
super(context, attrs);
}

@Override
protected void onMeasure(int wMeasureSpec, int hMeasureSpec) {
int measuredHeight = measureHeight(hMeasureSpec);
int measuredWidth = measureWidth(wMeasureSpec);


// in questo metodo bisogna chiamare setMeasuredDimension
// altrimenti viene sollevata un'eccezione durante il
// layout del componete
```

```
setMeasuredDimension(measuredHeight, measuredWidth);
}

private int measureHeight(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);

// dimesione di default

int result = 250;
if (specMode == MeasureSpec.AT_MOST) {
// calcolare la dimensione ideale per
// il componente
int mySize=....
// ritornalre la dimensione minima
// tra quella ideale e quella massima
result = Math.min(specSize,mySize);
}
else if (specMode == MeasureSpec.EXACTLY)
{
// nel caso il controllo sia contenuto
// nella dimensione, ritornare tale valore
result = specSize;
}//else MeasureSpec.UNSPECIFIED si usa il default
return result;
}

private int measureWidth(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
//... calcolo analogo al precedente ... ]
}

@Override
protected void onDraw(Canvas canvas) {
// recuperare le dimensioni ritornate da onMeasure.
int height = getMeasuredHeight();
int width = getMeasuredWidth();
// troviamo il centro
int px = width/2;
int py = height/2;
mTextPaint.setColor(Color.WHITE);
String displayText = "Hello World!";
// troviamo la dimensione del testo da scrivere
float textWidth = mTextPaint.measureText(displayText);
// disegnamo il testo nel centro del controllo
canvas.drawText(displayText, px-textWidth/2, py, mTextPaint);
}
@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
// Return true if the event was handled.
return true;
}
@Override
public boolean onKeyUp(int keyCode, KeyEvent keyEvent) {
// Return true if the event was handled.
return true;
```

```
}

@Override
public boolean onTrackballEvent(MotionEvent event ) {
// Get the type of action this event represents
int actionPerformed = event.getAction();
// Return true if the event was handled.
return true;
}
}
```

Eventi per l'interazione dei componenti con l'utente

Per fare in modo che l'estensione della View sia interattiva essa deve rispondere a tutta una serie di eventi generati dall'utente quali:

- `onKeyDown`
- `onKeyUp`
- `onTrackballEvent`
- `onTouchEvent`

Questi metodi ritornano true se l'evento è stato gestito dal metodo stesso

Menu

Non vedremo i menu in quanto sono relativamente semplici e non richiedono particolari attenzioni. Una volta che vediamo funzionare il menu stesso abbiamo la ragionevole certezza di non aver fatto errori.

Intenti

Gli intenti sono gli oggetti che Android usa per gestire il suo sistema di message passing. Uno degli usi più comune è quello di utilizzarli per far partire nuove Attività/servizi della stessa applicazione o anche di altre applicazioni. Usare gli intenti per propagare informazioni è uno dei principi di progettazione fondamentali di Android che favorisce il disaccoppiamento dei componenti e il loro riuso.

Nuove Attività

L'utilizzo più comune degli intenti è quello necessario per eseguire una nuova Activity, Service o Broadcast Receiver (della stessa applicazione o di una nuova).

Ci sono due modalità di esecuzione a seconda che vogliamo o meno esaminare il valore di ritorno dell'attività invocata. Ci sono anche due modalità di attivazione esplicita o meno a seconda se vogliamo accoppiare o disaccoppiare le due attività.

1) Esplicita senza esaminare il valore di ritorno:

```
Intent intent new Intent(this,OtherActivity.class);
startActivity(intent);
```

2) Implicita senza esaminare il valore di ritorno:

```
Intent intent new
Intent(Intent.ACTION_DIAL,URI.parse(“tel:0412348457”));
startActivity(intent);
```

Esempio: per far partire il browser web preinstallato nel terminale andorid sostituire il codice in rosso con:

```
Intent intent = new Intent(Intente.ACTION_VIEW,
Uri.parse(text.getText().toString());
startActivity(intent);
```

3) Esplicita con esame del valore di ritorno:

```
    int originalRequestCode;
... onKey(..)
  {..
   originalRequestCode = 2;
    Intent intent new Intent(this,OtherActivity.class);
    startActivityForResult(intent, originalRequestCode);
   …}
   @Override void onActionResult(int requestCode, int resultCode,
Intent data){
  if (requestCode==originalRequestCode){
    if (resultCode==Activity.RESULT_OK)
     {
       Uri uri = data.getData();
       boolean b = data.getBooleanExtra(“b”,false);
       int i = data.getIntExtra(“i”);
       Bundle extra = data.getExtras();
     }
  }
  if (requestCode == SHOW_2) {
   …}
}
```

La nuova attività prima di terminare (chiamando il metodo `finish()`) deve chiamare il metodo `setResult` specificando resultCode e un intento che può contenere i dati di risposta.
Es.

```
Uri data = ...
Intent result = new Intent(null,data);
result.putExtra(“b”,true);
result.putExtra(“i”,4);
result.put...
setResult(Activity.RESULT_OK,result); //Activity.RESULT_CANCELED
finish();
```

4) Implicita con esame del valore di ritorno.

```
private static final int PICK_CONTACT_SUBACTIVITY = 2;
Uri uri = Uri.parse("content://contacts/people");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, PICK_CONTACT_SUBACTIVITY);
```

L'Intent contiene informazioni che verranno elaborate del componente che riceve l'intento più informazioni che servono ad Android per selezionare il componente a cui inviare l'intento.
Le informazioni che servono ad Android sono:

1. per gli intenti espliciti il **nome del componente**, ovvero la classe (di solito per i componenti interni all'applicazione);
2. per gli intenti impliciti: action (costruttore, setAction), data (sia URI che il mimetype setUri, setType setData&Type) e le categorie (addCategory).

I dati che servono solo per il componente destinatario sono Extras (putIntExtras, putEstras e Flag) e servono per comunicare parametri tra il componente chiamante e il componente chiamanato.

Lezione 23 Intent Filter, Broadcast Event, Mappe, Geocoding, Servizi Location-Based

Intent Filter

Se l'intento una volta creato dal mittente deve essere ricevuto dal destinatario. Il destinatario deve indicare al sistema Android che è in grado di ricevere uno o più intenti. Usando gli Intent Filter la nostra applicazione può indicare al sistema Android a quali Intenti vuole/può rispondere.

```
<activity android:name=".CrashViewer" android:label="Crash View">
<intent-filter>
<action android:name="it.unive.dsi.intent.action.SHOW_CRASH"></action>
<category android:name="android.intent.category.DEFAULT"/>
<category
android:name="android.intent.category.ALTERNATIVE_SELECTED"/>
<data android:mimeType="vnd.earthquake.cursor.item/*"/>
</intent-filter>
</activity>
```

action: descrive il tipo di azione. Alcune azioni del sistema android sono:

- ACTION_ANSWER Inzia un'attività che risponde a una chimata telefonica;
- ACTION_CALL inizia una chiamanta corrispondente al numero descritto dall'Uri. (meglio usare ACTION_DIAL)
- ACTION_DELETE Inizia un'attività per eliminare il dato specificato nell'Uri (per esempio eliminare una ruga da un content provider);
- ACTION_DIAL effettua una chimata al numero specificato nell'Uri.
- ACTION_EDIT Inizia un'attività per editate il dato specificato nell'Uri.
- ACTION_INSERT Inizia un'attività per inserire un elemento nel Cursor specificato nell'Uri dell'intento. L'attività chiamata dovrebbe restituire un intento in cui l'Uri specifica l'elemento inserito.
- ACTION_PICK Inizia un'attività che permette di scegliere un elemento da il content provider specificato nel Uri. L'attività chiamanta dovrebbe ritornare un intento con l'Uri dell'elemento selezionato.
- ACTION_SEARCH Inizia un'attività che effettua la ricerca del termine passato su SearchManager.QUERY.
- ACTION_SENDTO Inizia un'attività per inviare un messaggio al contatto specificato nell'Uri.
- ACTION_SEND Inizia un'attività che invia i dati specificati nell'Intent. Il destinatario sarà individuato dalla nuova attività.
- ACTION_VIEW Inizia un'attività che visualizza il dato specificato nell'Uri.

- ACTION_WEB_SEARCH Inizia un'attività che effettua una ricerca sul Web del dato passato nell'Uri.

category l'attibuto android:name specifia sotto che circostanze un'azione sarà servita. Le categorie specificate possono essere più di una. Le categorie possono essere create dal programmatore oppure è possibile usare quelle di Android che sono:

- ALTERNATIVE Le azioni alternative all'azione standard sull'oggetto.
- SELECTED_ALTERNATIVE: Simile alla categoria ALTERNATIVE ma verrà risolta in una singola selezione
- BROWSABLE: specifica azioni disponibili dal Browser
- DEFAULT Per selezionare l'azione di default su un componente e per poter usare gli intenti espliciti.
- GADGET si specifica un'attività che può essere inclusa in un'altra attività.
- HOME Specificando questa categoria e non specificando l'azione, si propone un'alternativa allo schermo home nativo.
- LAUNCHER si specifica un'attività che può essere eseguita dal launcher del sistema Android.

data Questo tag permette di specificare che tipi di dati il componente può manipolare. Si possono specificare più tag di questo titpo. Sintassi: (<scheme>://<host>:<port>/<path>)

- android:scheme: uno schema (esempio: content or http).
- android:host: un hostname valido (es. google.com).
- android:port: la porta dell'host.
- android:path specifica un path dell'Uri (es. /transport/boats/).
- android:mimetype: Permette di specificare un dipo di dati che il componente è in grado di gestire. Per esempio <type android:value="vnd.android.cursor.dir/*"/> indica che il componente è in grado di gestire ogni Cursore di Android (per la definizione di Cursore, vedere i Content Provider).

Broadcast Event

Se un intento è pensato per attivare una singola attività, i broadcast event sono pensati per attivare molte attività contemporaneamente. Alcuni Broadcast Event del sistema Android sono:

`ACTION_BATTERY_LOW:` Batteria scarica.

`ACTION_HEADSET_PLUG` : Cuffiette collegate o scollegate dal terminale.

`ACTION_SCREEN_ON:` Lo schermo è stato acceso.

`ACTION_TIMEZONE_CHANGED:` Il fuso orario è stato cambiato.

Esempio di creazione di un nuovo Broadcast Event (CRASH_DETECTED):

```
Package it.unive.dsi.crash;
// l'attività o servizio che invia l'evento
public class Crash extend Activity{

final public static String
CRASH_DETECTED=”it.unive.dsi.action.CRASH_DETECTED”;

...
Intent intent = new Intent(CRASH_DETECTED);
//aggiungo alcuni parametri che descrivono l'evento
intent.putExtra("crashType", “car”);
intent.putExtra("level", 3);
intent.putExtra("description", “...”);
//invio l'evento a tutti gli ascolatori (BroadcastReceiver)
sendBroadcast(intent);
```

Broadcast Receiver

Ovviamente se non ci sono ascoltatori l'evento non genera nessun effetto. Per definire un BroadcastReceiver bisogna:

1) estendere la classe BroadcastReceiver e sovrascrivere il metodo onReceive. Es.:

```
public class CrashBroadcastReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
//TODO: React to the Intent received.
}
}
```

2) poi bisogna registrare su Android il BroadcastReceiver ovvero aggingere nel manifesto dell'applicazione che contiene il BroadcastReceiver che c'é un receiver e che eventi gestisce. Es.

```
<receiver android:name=".CrashBroadcastReceiver">
<intent-filter>
<action android:name="it.unive.dsi.Crash.CRASH_DETECTED"/>
</intent-filter>
</receiver>
```

Per i BroadcastReceiver è anche possibile creare e registrare il receiver dinamicamente (e quindi non necessariamente per tutto il tempo di vita dell'applicazione il Broadcast Receiver sarà attivo):

```
// Create and register the broadcast receiver.
IntentFilter filter = new
IntentFilter(it.unive.dsi.Crash.CRASH_DETECTED);
CrashBroadcastReceiver receiver = new CrashBroadcastReceiver();
registerReceiver(receiver, filter);
...
unregisterReceiver(receiver);
```

Una delle caratteristiche più interessanti delle applicazioni sviluppate su Android è la possibilità di

integrare facilmente posizionamento GPS, Mappe, geolocalizzazione etc.

Geolocalizzazione

Come configurare l'emulatore

Per poter verificare con il simulatore il cambiamenti di posizione del dispositivo mobile, bisogna configurare l'emulatore tramite la funzionalità “Location controls”.

Ci sono due modalità per indicare al simulatore le posizioni da usare durante la simulazione:

1. tramite file KML (generato by Google Earth);
2. tramite file GPX (generato da dispositivi GPS);
3. manualmente inviando una posizione alle volta.

Selezionare il tipo di localizzatore

Un dispositivo mobile può essere fornito di più dispositivi di localizzazione. Solitamente i due più usati sono:

1. tramite GPS
2. tramite Rete (GSM o WIFI).

Ogni tipo di localizzatore ha le sue peculiarità (precisione della localizzazione, consumo, costo etc.). Quindi per qualche applicazione può bastare un precisione bassa ma richiedere bassi consumi, in altre sarà necessario una localizzazione precisa e magari si è disposti a sopportare consumi o costi più alti.

Le API Android permettono di gestire diversi tipi di localizzatori tramite diversi “Location Provider”. Per gestire i diversi Location Provider si utilizza il “Location Manager” attraverso la classe “LocationManager.

Accedere Location Manager

Prima di iniziare a fare qualsiasi operazione con i Location Provider bisogna recuperare l'oggetto Location Manager nel seguente modo:

```
String serviceString = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager)getSystemService(serviceString);
```

Selezionare un Location Provider

Il Location Manager prevede varie modalità per selezionare un Location Provider. La più semplice è quella che prevede che il programmatoe già conosca il Location Provider più adatto e che quindi usi uno dei seguenti nomi di Location Provider:

```
LocationManager.GPS_PROVIDER
```

```
LocationManager.NETWORK_PROVIDER
```

Oppure possiamo richiedere tramite il Location Manager l'elenco di tutti i Location Provider presenti nel dispositivo tramite il metodo statico getProviders (l'argomento permette di specificare se devono venire ritornati solo i Location Manager attivati oppure anche quelli presenti ma non attivi).

```
List<String> providers = locationManager.getProviders(enabledOnly);

//select the provider based on name
```

Alternativamente il Location Manager metta a disposizione la possibilità di cercare il Location Provider che più si avvicina alle esigenze dell'applicazione e/o utente.

```
Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.ACCURACY_COARSE);
criteria.setPowerRequirement(Criteria.POWER_LOW);
criteria.setAltitudeRequired(false);
criteria.setBearingRequired(false);
criteria.setSpeedRequired(false);
criteria.setCostAllowed(true);

String bestProvider = locationManager.getBestProvider(criteria, true);
```

o alternativamente

```
List<String> matchingProviders =
locationManager.getProviders(criteria,
false);
```

Nel primo caso (getBestProvider) nel caso più provider soddisfino i criteri, viene scelto quello con precisione maggionre. Nel caso nessuno soddisfi i criteri, vengono rilassati nell'ordine i criteri:

1. consumo
2. precisione
3. possibilità di ritornare orientamento, velocità e altitudine.

Recupera la posizione

Per recuperare la posizione del device fornita da un Location Provider bisogna usare il metodo `getLastKnonwPosition` dell'oggetto location provider:

```
String provider = ...vedi sopra...
Location location = locationManager.getLastKnownLocation(provider);
```

Per invocare il metodo `getLastKnownLocation` bisogna che il manifesto dell'applicazione richieda il permesso di accedere alla posizione del device. Il permesso da richiedere può essere per la posizione precisa (fine) oppure approssimata (coarse). Il GPS fornisce una posizione precisa mentre la rete GSM fornisce una posizione approssimata. Ecco i due esempi:

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

```
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Anche se l'applicazione deve per forza richiedere i permessi per accedere alla posizione del device e' inoltre il caso di osservare le seguenti regole per preservare la privacy dell'utente:

1. Tracciare la locazione solo quando indispensabile all'applicazione;
2. Avvertire l'utente quando si sta tracciando la posizione e come tale posizione viene usate e memorizzata;
3. Permettere all'utente di disabilitare gli aggiornamenti della posizione.

La posizione ritornata può non essere aggiornata (vedi sotto)

L'oggetto `Location` ritornato include tutte le informazioni disponibili quando di utilizza quel dato Location Provider. Esempi di informazioni disponibili sono: latitudine, longitudine, orientamento, altitudine, velocità e tempo in cui è stata presa la posizione. Tutti queste informazioni sono disponibili tramite metodi get. Eventuali altre informazioni sono disponibili tramite l'oggetto `Bundle`.

Esempio

Aggiungiamo al file manifest.xml la richiesta del permesso di accedere alla posiozione accurata del device.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="it.unive.dsi.android">
<application
android:icon="@drawable/icon">
<activity
android:name=".GeoApp"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Modifichiamo il Layout main.xml in modo da aggiungere un attributo android:ID al controllo TextView control in modo da poter accedervi dall'attivita'.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:id="@+id/myLocationText"
```

```
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello"
/>
</LinearLayout>
```

Sovrascriviamo il metodo onCreate dell'attivita' GeoApp:

```
package it.unive.dsi.android;
import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.TextView;
public class GeoApp extends Activity {
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
LocationManager locationManager;
String context = Context.LOCATION_SERVICE;
locationManager = (LocationManager)getSystemService(context);
String provider = LocationManager.GPS_PROVIDER;
Location location =
locationManager.getLastKnownLocation(provider);
updateWithNewLocation(location);
}

private void updateWithNewLocation(Location location) {
String desc;
TextView myLocationText;
myLocationText = (TextView)findViewById(R.id.myLocationText);
if (location != null) {
desc=”Lat: “ + location.getLatitude(); //double
desc+= “Long:” + location.getLongitude(); //double
desc+=”Time: “+location.getTime(); //long ms dal 1970

if (location.hasAccuracy())
desc+=”Acc: ”+location.getAccuracy(); //in metri
if (location.hasAltitude())
desc+=”Alt: “+ location.getAltitude(); //in metri WGS84
if (location.hasBearing())
desc+=”Dir: “+location.getBearing(); //in gradi (Nord=0)
if (location.hasSpeed())
desc+=”Vel: “+location.getSpeed(); //in metri al secondo

Bundle b = location.getExtras();
if (b!=null)
for (String key:b.keySet()){
desc+= key + b.get(key); // solo satellites
}
} else {
latLongString = "No location found";
}
```

```
myLocationText.setText(latLongString);
}
}
```

L'applicazione appena vista non aggiorna la nuova posizione in caso di aggiornamenti e in realta' non visualizza la posizione corretta se qualche altra applicazione non ha gia' richiesto la posizione. Per avere un'attivita' che aggiorna la posizione visualizzata al cambiamento di posizone del device, bisogna usare un LocationListener e registrarlo attraverso il metodo requestLocationUpdates.

Tale metodo ha come parametri 1) il nome del Location Provider (o un insieme di criteri per individuare il provider da usare) 2) il tempo minimo che deve passare tra due aggiornamenti (per ridurre il consumo di batterie del dispositivo GPS e processire) 3) la distanza minima tra due aggiornamenti (come sopra) e 4) l'oggetto Listener.

Esempio:

```
String provider = LocationManager.GPS_PROVIDER;
int t = 5000; // ms
int distance = 5; // metri
LocationListener myLocationListener = new LocationListener() {
public void onLocationChanged(Location location) {
updateWithNewLocation(location);
}
public void onProviderDisabled(String provider){
updateWithNewLocation(null);
}
public void onProviderEnabled(String provider){
...
}
public void onStatusChanged(String provider, int status,
Bundle extras){
...
}
};
locationManager.requestLocationUpdates(provider, t, distance,
myLocationListener);
```

Quando il tempo minimo e la distanza minima sono stati superati, il relativo evento invochera' il metodo `onLocationChanged`.

Per fermare gli aggiornamenti della posizione e' possibile chiamare il metodo `removeUpdates`.

Dato che i moduli GPS consumano molta potenza, per minimizzare tale consumo bisognerebbe disabilitare gli aggiornamenti della posizione ogniqulvolta non necessario. In particolar modo quando l'applicazione non e' visibile e gli aggiornamenti di posizione servono solo per aggiornare l'interfaccia utente.

Proximity Alerts

I LocationListener reagiscono a ogni cambiamento di posizione (filtrato solo per tempo minimo e distanza). Talvolta c'e' la necessita' che l'applicazione reagisca solo quando il device entra o esca da una specifica locazione. I Proximity Alerts attivano l'applicazione solo quando il device entra o esce da una particolare locazione. Internamente, Android puo' usare differenti Provider in dipendenza di quanto vicino si trola alla locazione da monitorare e in questo modo e' possibile risparmiare energia.

Per impostare un Proximity Alert bisognaselezionare il centro (usando longitudine e latudine), un raggio

attorno al centro e una scadenza temporale dell'alert. L'alert verra' inviato sia che il device entri nel cerchio che ne esca. Quando si attiva un proximity alert, questo fa scattare degli Intenti e molto comunemente dei Broadcast Intent. Per specificare l'intento da scattare, si usa la classe PendingIntent che incorpora un Intento nel seguente modo:

```
Intent intent = new Intent(MY_ACTION);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, -1,
intent, 0);
```

Il prossimo esempio, imposta un proximity alert che non scade mai e che viene fatto scattare quando il device si allontana piu' di 10 metri dall'obiettivo.

```
private static String
DSI_PROXIMITY_ALERT = "it.unive.dsi.android.alert";
private void setProximityAlert() {
String locService = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager)getSystemService(locService);
double lat = 45.479272;
double lng = 12.255374;
float r = 100f; // raggio in metri
long time= -1; // non scade
Intent int = new Intent(DSI_PROXIMITY_ALERT);
PendingIntent proIntent = PendingIntent.getBroadcast(this, -1,int,0);
locationManager.addProximityAlert(lat, lng, r, time, proIntent);
}
```

Quando il LocationManager rileva che il device ha superato la circonferenza (verso il centro o verso l'esterno), l'intento incapsulato nel PendingIntent viene fatto scattare e la chiave `LocationManager.KEY_PROXIMITY_ENTERING` impostata a vero o falso in relazione alla direzione (entrata=true, uscita=false).

Per gestire i proximity alert bisogna creare un BroadcastReceiver del tipo:

```
public class ProximityIntentReceiver extends BroadcastReceiver {
@Override
public void onReceive (Context context, Intent intent) {
String key = LocationManager.KEY_PROXIMITY_ENTERING;
Boolean entering = intent.getBooleanExtra(key, false);
//effettuare le azioni opportune
}
}
```

Per cominciare a ricevere gli alert bisogna registrare il BroadcastReceiver nel seguente modo:

```
IntentFilter filter = new IntentFilter(DSI_PROXIMITY_ALERT);
registerReceiver(new ProximityIntentReceiver(), filter);
```

Lezione 23 – Geodecoding e SQLite

Geodecoding

Geocoding permette di traslare da un indirizzo a delle coordinate e viceversa. Questo permette di conoscera la posizione di un certo indirizzo o l'indirizzo piu' vicino di una certa posizione. La decodifica e' possibile interrogando un server remoto che contiene le posizione degli indirizzi. Per questa ragione l'applicazione che usa il geodecoding deve richiedere i permessi per accedere a Internet. Quindi nel manifesto dovra' comparire:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

La claase Geodecoder fornisce due funzionalita': una per conoscere la posizioen di un indirizzo e l'altra data la posizione di trovare l'indirizzo. Tali operazioni vengono contestualizzate attraverso i locale in quanto gli indirizzi dipendono fortemente dal paese e lingua dell'utente.

```
Geocoder geocoder = new Geocoder(getApplicationContext(),
Locale.getDefault());
```

Tutti e due i mentodi ritornano una lista di oggetti Addres che rappresentano i possibili risultati. In fase di chimata si specifica il numero massimo di risultiati che si vuole vengano restituiti.
I dati contenuti nel singolo indirizzo sono tutti quelli che e' stato possibile recuperare nel database.

L'accesso al database viene effettuato in modo sincrono e quindi il thread si blocca finche' non riceve la risposta. Per questa ragione, le chiamate a questi metodi andrebbero fatte in thread secondari e non nel thread principale come negli esempi che seguono.

Reverse Geocoding: dalla posizione all'indirizzo

```
location =
locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
double latitude = location.getLatitude();
double longitude = location.getLongitude();
List<Address> addresses = null;

int maxResults=5;
Geocoder gc = new Geocoder(this, Locale.getDefault());
try {
addresses = gc.getFromLocation(latitude, longitude, maxResults);
} catch (IOException e) {}
```

L'accuratezza e la granularita' della risposta dipende dai dati presenti nel database remoto e che dipendono dalla localita' e dal tempo.

Forward Geocoding: dall'indirizzo alla posizione

Forward geocoding (o semplicemente geocoding) trova gli indirizzi corrispondenti a una stringa. La stringa contiene un nome di indirizzo. Ogni stringa con cui possiamo interrogare Google Maps e' adatta.

La lista di indirizzi restituita include la longitudine e la latidutine dell'indirizzo.

```
Geocoder fwdGeocoder = new Geocoder(this, Locale.US);
String streetAddress = "via Torino 155, Venezia";
List<Address> locations = null;

int maxResult=5;
try {
locations = fwdGeocoder.getFromLocationName(streetAddress, maxResult);
} catch (IOException e) {}
```

Per una localizzazione piu' precisa e' possibile usare il metodo overloaded con la possibilita' di specificare il rettangolo delimitatore della zona dove vogliamo cercare l'indirizzo, cosa che puo' risultare perticolarmente utile se vogliamo restringere la ricerca alla zona visualizzata all'utente (prossimo paragrafo).

```
List<Address> locations = null;
try {
locations = fwdGeocoder.getFromLocationName(streetAddress, 10,
n, e, s, w);
} catch (IOException e) {}
```

Esempio di utilizzo dell'oggetto Address:

```
private void updateWithNewLocation(Location location) {
String latLongString;
TextView myLocationText;
myLocationText = (TextView)findViewById(R.id.myLocationText);
String addressString = "No address found";
if (location != null) {
double lat = location.getLatitude();
double lng = location.getLongitude();
latLongString = "Lat:" + lat + "\nLong:" + lng;
double latitude = location.getLatitude();
double longitude = location.getLongitude();
Geocoder gc = new Geocoder(this, Locale.getDefault());
try {
List<Address> addresses = gc.getFromLocation(latitude, longitude, 1);
StringBuilder sb = new StringBuilder();
if (addresses.size() > 0) {
Address address = addresses.get(0);
for (int i = 0; i < address.getMaxAddressLineIndex(); i++)
sb.append(address.getAddressLine(i)).append("\n");
sb.append(address.getLocality()).append("\n");
sb.append(address.getPostalCode()).append("\n");
sb.append(address.getCountryName());
}
addressString = sb.toString();
} catch (IOException e) {}
} else {
latLongString = "No location found";
}
myLocationText.setText("Your Current Position is:\n" +
latLongString + "\n" + addressString);
}
```

SQLite

Anche Android prevede la possibilità di accedere ad un Database. L'API che viene usata non è però

JDBC. Anche il DB non è un DB server ma è implementato tramite un libreria C che gira nello stesso processo dell'applicazione Android (molto simile al Derby nella versione Embedded).

La modalità di accesso al DB è molto simile a quella di accesso ai Content Provider che vedremo nel prossimo paragrafo.

Ecco un esempio per la creazione di un database con una tabella:

```
SQLiteDatabase db;
db = openOrCreateDatabase(“DB”, Context.MODE_PRIVATE, null);
db.execSQL("create table PROVA  ( id  integer primary key autoincrement, Nome text
not null);”);
```

Una query al DB viene fatta utilizzando il metodo **query** che restituisce un cursore (istanza oggetto Cursor). Gli argomenti del/i metodo/i query sono:

1. un booleano equivalente alla keyword SQL DISTINCT;
2. il nome della tabella;
3. un'array di stringhe con i nomi delle colonne che compaiono nel risultato della query (proiezione);
4. L'equivalente della clausola WHERE che seleziona le righe da restituire. Si possono includere i caratteri speciali '?' Per gestire i parametri analogamente ai PreparedStatement.
5. Un'array di stringhe che rimpiazzeranno nell'ordine i parametri della clausola WHERE rappresentati da '?'.
6. La clausola GROUP BY che definisce come le righe devono essere raggruppate.
7. La clausola HAVING che seleziona i gruppi di righe da restituire
8. Una stringa che specifica l'ordine
9. Una stringa opzionale per limitare il numero di righe ritornate.

Es

```
String[] result_columns = new String[] {“id”, “nome”, “cognome”};
Cursor allRows = db.query(true, “PROVA”, result_columns,null, null, null, null, null,
null);

String where = “cognome=?”;
String order = “nome”;
String[] parameters = new String[]{requiredValue};
Cursor res = db.query(“PROVA”, null, where,parameters, null, null, order);

while(res.moveToFirst()) {

String nome = res.getString(“nome”);

}

// Creiamo una riga di valori da inserire
ContentValues newValues = new ContentValues();
// assegnare un valore per ogni colonna
newValues.put(“nome”, newValue);
[ ... Ripetere per ogni colonna ... ]
// inserire la riga nella tabella del DB
db.insert(“PROVA”, null, newValues);
```

```
// Creiamo una riga per modificare una riga nel DB
ContentValues updatedValues = new ContentValues();
// assegnare un valore per ogni  colonna
updatedValues.put(“nome”, newValue);
[ ... Ripetere per ogni colonna ... ]
String where = “id=?";
String[] parameters = new String[]{“3”};
// Aggiornare la riga nel DB
db.update(“PROVA”, updatedValues, where, parameters);

db.delete(“PROVA”, “nome=?", nnew String[]{“alessandro”});
db.close();
```

Tecnologie e Applicazione Web - Domande orale

Esempi di alcune domande che risultano ostili agli studenti

1. Se l'indirizzo IP è sufficiente a individuare un host in Internet, a cosa serve la porta nelle connessioni TCP?
2. Quali sono i problemi nella lettura da un Socket TCP? Perché non ci sono nella lettura dai DatagramSocket?
3. A cosa serve un ServerSocket?
4. A cosa può servire un oggetto Socket a un server?
5. Perché il protocollo HTTP prevede di chiudere la connessione alla fine di ogni richiesta? Che vantaggi e svantaggi ci sono?
6. Descrivere il protocollo HTTP?
7. Che vangaggi offre un'applicazione che implementa lo standard CGI rispetto alle Servlet?
8. In una richiesta GET dove trova i parametri un'applicazione CGI?
9. In una richiesta POST dove trova i parametri un'applicazione CGI?
10. A che cosa può servire lo Standard Imput a un CGI?
11. A che cosa serve lo Standard Output in un CGI?
12. Perché una Servlet usa meno risorse di un CGI? È sempre vero?
13. Di che cosa ha bisogno una Servlet per poter essere eseguita?
14. Che differenza c'è tra implementare un CGI in java e una Servlet?
15. Che differenze c'è tra implementare una pagina dinamica con una Servlet e una JSP?
16. Che tecniche è possibile usare per implementare le sessioni utente con il protocollo HTTP?
17. A che cosa servono i Tag nella pagine HTML?
18. Descrivere il ciclo di vita delle Servlet (Applet, Activity e Application Android)
19. Come si accede a un Database in Java? A cosa servono i Driver? Perché è necessario usare Class.forName(...Driver)?
20. A cosa serve un sistema LDAP? Come si accede?
21. A cosa serve l'RMI? Che vantaggi e svantaggi ci sono?
22. Che particolarità ha la VM di Android?
23. Perché Android non usa i file .class?
24. A cosa serve il file manifest.xml?
25. A cosa serve la directory res in un progetto Android? Come vengono usate le risorse nelle applicazioni?