Appunti di Tecnologie e Applicazioni Web

Antonio Bucciol *e* Riccardo Brambilla

Ottobre 2014

Sommario

Appunti del corso di Tecnologie e Applicazioni per il Web - A.A. 2013/14.
Corso tenuto dal prof. Alessandro Roncato, Università Ca' Foscari di Venezia, Corso di laurea in Informatica.

Riferimenti:
Antonio Bucciol - antonio@bucciol.org - www.bucciol.org
Riccardo Brambilla - riccardo.brambilla.85@gmail.com

IMPORTANTE - NOTE SUL DOCUMENTO:

- Dovrebbero esserci quasi tutti gli argomenti trattati nel corso di TAW 2013/14.
- Lavoriamo entrambi, quindi abbiamo scritto questi appunti senza frequentare le lezioni, ma integrando ricevimenti dal docente con manuali e reference disponibili *online*.
- Alcune tecnologie esaminate nel corso iniziano ad essere desuete (Es. JSP): in alcuni casi abbiamo preso in considerazione la nuova tecnologia (es.: ultima versione di Android).
- Abbiamo inserito parecchi approfondimenti, in quanto crediamo che conoscere le origini di una soluzione aiuti a comprenderla.
- S\'i, ci saranno sicuramente degli errori.

Antonio e Riccardo

Indice

Parte I
Reti

1 Socket, applicazioni client/server

1.1 Livello Applicazione

1.1.1 Architettura Client/Server

Tra le due, è l'architettura più semplice da implementare: vi è un'applicazione server che resta perennemente in attesa di una connessione; il client si connetterà ad essa.

Esempi: Web, FTP, telnet, smtp, pop, etc.

Client può creare un *socket* in qualunque istante (per iniziare la conversazione con un server).

Server deve prepararsi in ascolto in anticipo per poter ricevere connessioni.

1.1.2 Architettura Peer to Peer

Più complessa da sviluppare, non verrà trattata in questo corso.

Esempi: Napster, Gnutella, Kademlia.

1.2 Protocollo TCP/IP

1.2.1 TCP

Prima di poter scambiare dati, viene stabilita una connessione stabile tra client e server. Garanzia di consegna e di ordine di consegna.

Per contro ha un maggior overhead, dovuto principalmente alle fasi di *handshake* e di *slow-start*.

1.2.2 UDP

Connectionless. Non garantisce nè avvenuta consegna, nè ordine di consegna dei datagrammi. Ha minore overhead e quindi più efficiente di TCP, ma molto meno sicuro. Utilizzato principalmente nelle LAN.

Può inviare datagrammi in *broadcast* ed in *multicast*.

In Java viene usata la classe **DatagramSocket**.

1.2.3 Porte standard

Nella creazione di un nuovo socket, il client deve indicare sia l'indirizzo IP del server a cui vuole connettersi, che la porta sulla quale è in ascolto l'applicazione server desiderata.

Il client si appoggia ad una porta non pre-definita per ogni nuova connessione. Il server resta invece in ascolto sempre sulla stessa porta, che è stata definita in precedenza.

Le porte disponibili nel protocollo TCP/IP, che sono 65535 (2^16) sono in buona parte riservate per specifici servizi. In particolare, le prime 1024 porte sono storicamente le "intoccabili", in quanto v'è gran probabilità di andare in conflitto con servizi standard. Nei sistemi *NIX, difatti, solamente applicazioni eseguite da *root* possono mettersi in ascolto sulle porte <=1024.

1.3 Socket

1.3.1 Definizione

I socket nell'architettura TCP/IP rappresentano il punto di accesso (un'API) delle **applicazioni** verso il livello **trasporto** (TCP o UDP). Concettualmente si trovano tra la porta e l'applicazione.

1.3.2 Implementazione in Java

2 Applicazioni Web

La struttura del Web consta di tre componenti fondamentali:

2.1 URL

URL - *Uniform Resource Locator* - nella terminologia è una sequenza di caratteri che identifica univocamente l'indirizzo di una risorsa in Internet, tipicamente presente su un host server, come ad esempio un documento, un'immagine, un video, rendendola accessibile ad un client che ne faccia richiesta attraverso l'utilizzo di un web browser.

Ogni Uniform Resource Locator si compone normalmente di sei parti, alcune delle quali opzionali. In grassetto gli attributi obbligatori:

protocollo://<username:password@>**nomehost**<:porta></percorso><?querystring><#fragmentIdentifier>

2.2 HTTP

L'HyperText Transfer Protocol (HTTP) (protocollo di trasferimento di un ipertesto) è usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server.

Le specifiche del protocollo sono gestite dal World Wide Web Consortium (W3C). Un server HTTP generalmente resta in ascolto delle richieste dei client sulla porta 80 usando il protocollo TCP a livello di trasporto.

L'HTTP funziona su un meccanismo richiesta/risposta (client/server): il client esegue una richiesta e il server restituisce la risposta. Nell'uso comune il client corrisponde al browser ed il server al sito web. Vi sono quindi due tipi di messaggi HTTP: messaggi richiesta e messaggi risposta.

HTTP differisce da altri protocolli di livello 7 come FTP, per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta (o una serie di richieste correlate) è stata soddisfatta. Questo comportamento rende il protocollo HTTP ideale per il World Wide Web, in cui le pagine molto spesso contengono dei collegamenti (link) a pagine ospitate da altri server diminuendo così il numero di connessioni attive limitandole a quelle effettivamente necessarie con aumento quindi di efficienza (minor carico e occupazione) sia sul client che sul server. Talvolta però pone problemi agli sviluppatori di contenuti web, perché la natura senza stato (stateless) della sessione di navigazione costringe ad utilizzare dei metodi alternativi - tipicamente basati sui cookie - per conservare lo stato dell'utente.

2.3 HTML

2.3.1 Definizione

L'HyperText Markup Language (HTML) (traduzione letterale: linguaggio a marcatori per ipertesti), in informatica è il linguaggio di markup solitamente usato per la formattazione di documenti ipertestuali disponibili nel World Wide Web sotto forma di pagine web.

È un linguaggio di pubblico dominio, la cui sintassi è stabilita dal World Wide Web Consortium (W3C), e che è derivato da un altro linguaggio avente scopi più generici, l'SGML.

2.3.2 Storia ed evoluzione

L'HTML è stato sviluppato verso la fine degli anni ottanta del XX secolo da Tim Berners-Lee al CERN di Ginevra assieme al noto protocollo HTTP che supporta invece il trasferimento di documenti in tale formato. Verso il 1994 ha avuto una forte diffusione in seguito ai primi utilizzi commerciali del web.

Nel corso degli anni, seguendo lo sviluppo di Internet, l'HTML ha subito molte revisioni, ampliamenti e miglioramenti, che sono stati indicati secondo la classica numerazione usata per descrivere le versioni dei software. Attualmente l'ultima versione disponibile è la versione 4.01, resa pubblica il 24 dicembre 1999.

Dopo un periodo di sospensione, in cui il W3C si è focalizzato soprattutto sulle definizioni di XHTML (applicazione a HTML di regole e sintassi in stile XML) e dei fogli di stile (CSS), nel 2007 è ricominciata l'attività di specifica con la definizione, ancora in corso, di HTML5.

Il W3C ha annunciato che la prima versione di tale standard sarà pronta per fine 2014 e l'html 5.1 per il 2016.

3 Protocollo HTTP

3.1 Ciclo di vita di una request/response

Ogni *request* ed ogni *response* HTTP sono così formate:

- Request line [per le *request*] oppure status line [per le *response*]
- Headers
- CRLF [cioè una riga vuota]
- Body [opzionale]

Lo header contiene la richiesta/risposta, con i suoi parametri.
Il body contiene l'eventuale corpo della richiesta/risposta.

- Nel caso di una POST:
  - request: il body contiene i dati inviati al server.
  - response: vuoto.
- Nel caso di una HEAD: è sempre vuoto.
- Nel caso di una GET:
  - request: vuoto.
  - response: contiene la pagina HTML (<html><head></head><body></body></html>).

Il ciclo di vita è il seguente:

1. Viene stabilita la connessione HTTP
2. Viene inviata la request
3. Viene ritornata la response
4. Viene terminata la connessione

3.2 Tipi di *request*

Ogni request può implementare uno tra i seguenti metodi:

- GET
- HEAD
- POST
- PUT [deprecato]
- DELETE [deprecato]
- OPTIONS
- TRACE
- CONNECT

I metodi principali sono i primi tre: GET, HEAD, POST. Andiamo ad analizzarli.

3.2.1 Metodo GET

Il metodo GET serve per richiedere una determinata risorsa ad un server. È il metodo che viene utilizzato dai browser quando si scrive un URL nella barra degli indirizzi.

È possibile accodare dei parametri all'URL (fino a 3000 byte).

La richiesta GET ha la seguente struttura:

Sezione	Es. di contenuto	Note
Request line	GET / HTTP/1.1	Es.: al posto di / potremmo avere /~taw oppure /~taw?p=123
Headers	Host: www.dsi.unive.it	Ogni header va separato su una sua riga. Vedi sotto per altri header.
Linea vuota	CRLF	
Entity body	<vuoto>	La GET non prevede corpo. Eventuali parametri sono codificati nell'URI.

Alcuni degli Header più comuni:

Host specifica l'host al quale si vuole richiedere la risorsa.

Accept-charset

Accept-encoding

Accept-language

Connection tipo di connessione preferita dall'User-agent. Es.: keep-alive, close.

Esempio di richiesta GET:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

3.2.2 Metodo HEAD

Il metodo HEAD è esattamente come il metodo GET, sia in quanto a proprietà che in quanto a tipologia.

L'unica differenza è che, a seguito di una richiesta HEAD, il server restituirà solamente l'header della risposta, e non header+body.

Questo torna utile quando si vuole conoscere il codice di risposta ad una certa richiesta, senza però scaricare tutto il relativo contenuto.

Esempio:

```
HEAD /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

3.2.3 Metodo POST

Il metodo POST serve per inviare dati al server. Aggiorna una risorsa esistente o fornisce dei dati in ingresso.

I dati sono accodati dopo l'header e possono avere lunghezza illimitata.

Possono essere codificati o meno.

3.2.4 Metodo PUT

Il metodo PUT serve per trasmettere delle informazioni dal client al server, creando o sostituendo la risorsa specificata.

- Differenza tra PUT e POST:
  - L'URI di POST identifica la risorsa che gestira l'informazione inviata nel corpo della richiesta.
  - L'URI di PUT identifica nel corpo della richiesta la risorsa che ha inviato: è la risorsa che ci si aspetta di ottenere facendo in seguito una GET con lo stesso URI.
  - *Praticamente, questo significa che, mentre POST invia dati a un URI esistente, PUT crea una nuova risorsa ad un URI in essa specificato.*
- È idempotente.
- Non offre nessuna garanzia di controllo degli accessi.

3.3 Tipi di *response*

I codici di stato di una *HTTP response* sono divisi in 5 classi.

È importante conoscere cosa rappresenta ognuna di queste classi, e conoscere uno o due codici di esempio per ognuna: non è necessario impararsi a memoria tutti i codici.

Vediamo queste classi nel dettaglio.

3.3.1 1xx [Informational] codici di informazione

Introdotto in HTTP/1.1. Consistono solamente della status-line e di header opzionali, il tutto terminato da un CLRF.

Quando un server dialoga con uno user agent in HTTP/1.0 non deve fare uso dei codici 1xx, in quanto non supportati.

100 Continue Il client dovrebbe continuare con la sua richiesta. Indica che il server ha ricevuto una parte della richiesta e non l'ha rigettata, e attende il resto della richiesta dal client.

101 Switching protocols When a client sends a request which contains an *Update* header, the server responses with this status line.

The 101 status line means that the server, right after the CLRF which closes this response, will switch to the protocol required by the client via the Upgrade request header.

3.3.2 2xx [Successful] codici per richieste evase con successo

Questa classe di codici indica che la richiesta del client è stata correttamente: ricevuta, compresa, ed accettata.

200 OK The request has succeeded. The information returned with the response is dependent on the method used in the request, for example:

GET an entity corresponding to the requested resource is sent in the response;

HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;

POST an entity describing or containing the result of the action;

TRACE an entity containing the request message as received by the end server.

201 Created La richiesta è stata evasa correttamente ed ha prodotto la creazione di una nuova risorsa. L'URI della risorsa specifica viene riportato nell'header: *Location*.

Se la richiesta non può essere evasa immediatamente, il server dovrebbe rispondere con un codice *202*.

202 Accepted La richiesta è stata accettata, ma non è ancora stata elaborata (nè vi è certezza che lo sarà). Tale risposta termina la connessione, pertanto è consigliabile che contenga un link ad una risorsa che permetta al client di controllare lo stato di elaborazione della richiesta inviata.

203 Non-authoritative information Una copia locale o *third-party* risponde con dei metadati cachati dal server, senza garanzia di correttezza. Il suo uso non è richiesto (e non ci è comunque chiaro quando venga usato).

204 No content Il server ha correttamente elaborato la richiesta, ma non è necessario che lo user-agent aggiorni la vista lato client. Non prevede un *entity body*.

205 Reset content Il server ha correttamente elaborato la richiesta, e lo user-agent deve resettare la view lato client. Questo solitamente avviene per consentire all'utente, ad esempio, di inserire un nuovo set di dati in un form. Non prevede *entity body*.

206 Partial content Il server ha correttamente elaborato una richiesta di GET parziale. Questo significa che la richiesta GET conteneva un header *Range* e, opzionalmente, un header *If-Range*.

La risposta 206 deve includere:

- l'header Content-Range
- la data
- l'header Content-location
- uno o più degli header Expires/Cache-Control/Vary.

3.3.3 3xx [Redirection] codici di redirezione

Questa classe indica allo user-agent che deve compiere ulteriori azioni per evadere la richiesta.

Tali azioni vengono compiute automaticamente dal browser solamente nel caso in cui l'ulteriore azione richiesta sia una GET o una HEAD.

Sta allo user-agent rilevare eventuali loop infiniti. N.B: precedenti versioni dettavano il limite di max 5 redirect; alcuni client potrebbero avere ancora questo limite.

300 Multiple choiches La risorsa richiesta corrisponde ad una qualche risorsa di un elenco, ognuna con la sua specifica locazione.

A meno che non fosse una richiesta di tipo HEAD, la risposta dovrebbe contenere un elenco delle locazioni disponibili. La scelta può essere effettuata manualmente dall'utente o automaticamente dallo user-agent.

Il server può eventualmente specificare la locazione più appropriata nello header *Location*.

301 Moved permanently La risorsa richiesta è stata permanentemente ad un nuovo URI. Per futuri usi, andrebbe usato uno degli URI restituiti da questa response nello header *Location*.

Per richieste diverse da HEAD, dovrebbe tornare anche un *entity body* con un link al nuovo URI.

Per le richieste diverse da HEAD e da GET, lo user-agent non deve redirigere automaticamente; deve bensì chiedere all'utente di scegliere, in quanto i dati verrebbero altrimenti inviati ad una diversa destinazione senza che l'utente se ne accorgesse.

302 Found Esattamente come per 301, ma in questo caso la risorsa è stata spostata solo temporaneamente. Lo user-agent non deve quindi salvarsi il nuovo URI, continuando invece ad usare quello corrente.

303 See other Indica un URI verso il quale l'user-agent dovrebbe eseguire una GET. Questo codice viene solitamente restituito a seguito di una POST. Dovrebbe includere anche un link all'URI nell'*entity body*.

304 Not modified Nel caso in cui il client operi una GET condizione, ad es. If-Not-Modified, ed il contenuto non sia stato modificato, il server ritorna lo stato 304. Tale stato non deve avere un *entity body*.

305 Use proxy La risorsa richiesta deve essere acceduta tramite il proxy indicato nell'header *Location* della response. Ci si aspetta che il client ripeta la richiesta tramite tale proxy.

306 (Unused) Nelle nuove versioni di HTTP, questo codice non viene più utilizzato.

307 Temporary redirect Come per stato 302.

3.3.4 4xx [Client error] codici di errore lato client

I codici 4xx riguardano errori lato client.

Eccetto per le richieste HEAD, il server dovrebbe ritornare un *entity body* contenente una spiegazione dell'errore, specificando se si tratta di un errore temporaneo o permanente. Questi codici sono applicabili a qualsiasi metodo di richiesta. Gli user-agent dovrebbero sempre mostrare tali dettagli all'utente.

[TODO: completare descrizioni dei codici di stato]

400 Bad request

401 Unauthorized

402 Payment required

403 Forbidden

404 Not found

405 Method not allowed

406 Not acceptable

407 Proxy authentication required

408 Request timeout

409 Conflict

410 Gone

411 Length required

412 Precodintion failed

413 Request entity too large

414 Request URI too long

415 Unsupported media type

417 Excpectation failed

3.3.5 5xx [Server error] codici di errore lato server

I codici di stato 5xx indicano i casi in cui un server si rende conto di avere un errore, o di essere incapace di soddisfare la richiesta.

Eccetto per le richieste HEAD, il server dovrebbe ritornare un *entity body* contenente una spiegazione dell'errore, specificando se si tratta di un errore temporaneo o permanente. Questi codici sono applicabili a qualsiasi metodo di richiesta. Gli user-agent dovrebbero sempre mostrare tali dettagli all'utente.

500 Internal server error Il server ha incontrato un errore inatteso che gli impedisce di espletare la richiesta.

501 Not implemented Il server non supporta la funzionalità richiesta. Ad es., quando non supporta il metodo della request.

502 Bad gateway Il server, mentre agiva da gateway o da proxy, non è stato in grado di ottenere una risposta (corretta) dal server principale.

503 Service unavailable Il server non è temporaneamente in grado di soddisfare la richiesta a causa di un sovraccarico, o di una manutenzione.

Se il server conosce la durata dell'indisponibilità, specifica lo header *Retry-After* nella response.

Da notare che molti server rispondono ai sovraccarichi semplicemente rifiutando la connessione.

504 Gateway timeout Il server, mentre agiva da gateway o da proxy, non ha ricevuto una risposta dal server principale entro il tempo prestabilito.

505 HTTP Version not supported Il server non supporta, o si rifiuta di supportare, la versione del protocollo HTTP usata nella request. Tale response dovrebbe includere un *entity body* nel quale viene spiegato il motivo di tale scelta, e che contiene un elenco dei protocolli accettati.

4 Server HTTP

Un server HTTP (ad es., Apache o IIS) è un programma sempre in esecuzione che attende richiesta dai client e ritorna le risorse specificate nella richiesta.

Ci sono due tipi fondamentali di risorse disponibili nel Web server:

- Risorse statiche: le risorse esistono prima della richiesta e il loro contenuto è fissato.
- Risorse dinamiche: le risorse vengono create in maniera automatica al momento della richiesta, tramite un programma.

Le principali risorse gestite da un Web server sono:

- Pagine HTML.
- File di testo.
- Documenti di testo (PDF, PS, etc).
- Immagini.
- Animazioni.
- Altri contenuti multimediali.

Tutte queste risorse possono essere sia statiche che dinamiche, e generalmente vengono generate pagine HTML.

4.1 Un esempio: TinyHTTPD

TinyHTTPD ascolta su una porta specificata e server semplici richieste HTTP:

GET /path/filename HTTP/1.0

Il Web browser invia una o più di queste linee per ogni documento da ottenere. Il server prova ad aprire il file specificato nella richiesta e ne restituisce il contenuto. Se il documento contiene referenze a risorse addizionali, ad es. immagini, il Web browser continua con richieste GET addizionali.

TinyHTTPD server ogni richiesta in un suo thread, in modo da poter servire più richieste concorrentemente.

4.1.1 Esempio di implementazione

Listing 1: TinyHTTPD.java

```
1  import java.net.*;
2  import java.io.*;
3  import java.util.*;
4
5  public class TinyHttpd {
6          public static void main( String argv[] ) throws IOException {
7                  ServerSocket ss =
8                  new ServerSocket(Integer.parseInt(argv[0]));
9                  while ( true )
10                 new TinyHttpdConnection( ss.accept() );
11         }
12 }
13
14 public class TinyHttpdConnection extends Thread {
15         Socket sock;
16
17         public TinyHttpdConnection ( Socket s ) {
18                 sock = s;
19                 setPriority( NORM_PRIORITY - 1 );
20                 start();
```

```
21          }
22
23          public void run() {
24                  try {
25                          OutputStream out = sock.getOutputStream();
26                          BufferedReader d = new BufferedReader(new InputStreamReader(sock.
                                getInputStream()));
27                          String req = d.readLine();
28                          System.out.println( "Request:␣"+req );
29                          StringTokenizer st = new StringTokenizer( req );
30
31                          if ( (st.countTokens() >= 2) && st.nextToken().equals("GET") ) {
32                                  if ( (req = st.nextToken()).startsWith("/") )
33                                          req = req.substring( 1 );
34
35                                  if ( req.endsWith("/") || req.equals("") )
36                                          req = req + "index.html";
37
38                                  try {
39                                          FileInputStream fis = new FileInputStream ( req );
40                                          byte [] data = new byte [ fis.available() ];
41                                          fis.read( data );
42                                          //Esercizio: scrivere header
43                                          out.write( data );
44                                  } catch ( FileNotFoundException e ) {
45                                          new PrintStream( out ).println("404␣Not␣Found");
46                                  }
47                          } else
48                                  new PrintStream( out ).println( "400␣Bad␣Request" );
49
50                          sock.close();
51                  } catch ( IOException e ) {
52                          System.out.println( "I/O␣error␣" + e );
53                  }
54          }
55  }
```

4.1.2 Esempio di funzionamento

Compilare il codice sopra listato, ed eseguirlo:

> java TinyHTTPD.java 8080

avendo cura di predisporre un file *prova.html* all'interno della directory in cui si esegue il programma.

Provare poi a raggiungere tale server tramite un Web browser digitando l'URI:

http://localhost:8080/prova.html

Oppure ancora, tramite telnet, digitando:

> telnet localhost 8080
(...attendere che la connessione vada a buon fine...)
GET /prova.html HTTP/1.0
Host: www.unive.it
[riga vuota]

Esempio di richiesta GET con risposta 200 OK:

Win7@BUCSTATION ~ $ telnet localhost 80

Trying 127.0.0.1...
Connected to localhost. Escape character is '^]'.
GET /index.html HTTP/1.1
Host: localhost

[*riga vuota*]

HTTP/1.1 200 OK
Date: Thu, 02 Oct 2014 09:26:03 GMT
Server: Apache/2.2.21 (Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1
Last-Modified: Sat, 19 Mar 2011 08:49:44 GMT
ETag: "c0000000b5eac-ca-49ed1f9d9da00"
Accept-Ranges: bytes
Content-Length: 202
Content-Type: text/html

[*riga vuota*]

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loos
<html> <head> <title></title> </head> <body> <h1>It works!</h1> </body> </html>

Connection closed by foreign host.

Esempio di richiesta GET con risposta 400 Bad Request:

Win7@BUCSTATION ~ $ telnet localhost 80

Trying 127.0.0.1...
Connected to localhost. Escape character is '^]'.
GET /index.html HTTP/1.1

[*riga vuota*]

HTTP/1.1 400 Bad Request
Date: Thu, 02 Oct 2014 09:24:36 GMT Server: Apache/2.2.21 (Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1
Content-Length: 368
Connection: close
Content-Type: text/html; charset=iso-8859-1

[*riga vuota*]

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> <html><head> <title>400 Bad Request</title> </head><body> <h1>Bad Request</h1> Your browser sent a request that this server could not understand.
  <hr> <address>Apache/2.2.21 (Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1 Server at localhost Port 80</address> </body></html>

Connection closed by foreign host.

4.1.3 Note sull'implementazione [problemi]

Abbassando la priorità dei thread creati per gestire le richiesta (a NORM_PRIORITY - 1), assicuriamo che il thread principale, il cui compito è di accettare le nuove richieste, non venga bloccato da thread che servono singole richieste.

Permangono invece le seguenti problematiche:

1. Elevato consumo di memoria: anzichè caricare i file in una volta sola in un grande array, la lettura andrebbe bufferizzata per inviare una parte alla volta.

2. Non riconosce le directory: gli si potrebbe far listare il contenuto di una directory, quando richiesta.
3. È poco portabile: andrebbe adattata la gestione dei path per gestirli sui vari sistemi diversi da *NIX e DOS.
4. Non gestisce le pagine dinamiche: mancano i meccanismi per gestirle (CGI, PHP, servlet, etc).
5. Non ci sono restrizioni per l'accesso ai file: con qualche trucco, spedirebbe qualsiasi file presente sul suo filesystem al client.

4.1.4 Possibile soluzione per la restrizione dell'accesso

Potremmo ovviare a tale problema implementando un controllo sul path prima di aprire un file. Questa implementazione richiederebbe però di verificare che il file sia leggibile.

Un altro approccio potrebbe estendere la classe *SecurityManager* per gestire in automatico l'accesso al filesystem. Tale classe implementa una serie di metodi, che solitamente rispondono sollevando un'eccezione *SecurityException*, ai quali dobbiamo aggiungere le casistiche desiderate per le quali devono ritornare *true*:

- checkAccess(thread)
- checkListen(port)
- checkLing(dinamicLibrary)
- checkPropertyAccess(systemProperty)
- checkAccept(host, port)
- checkWrite(file)
- checkRead(file)

Parte II
Tecnologie deprecate

5 CGI

5.1 Definizione

CGI - *Common Gateway Interface* - è un'interfaccia per il trasferimento di dati tra un *Web Server* ed una applicazione (*script CGI*), espandendo così i servizi offerti dal *Web server* stesso.

Tale tecnologia risulta deprecata: studiarla ci aiuta a comprendere i vantaggi di nuove tecnologie come Java EE.

Tale interfaccia è implementata tramite le **variabili di sistema**: un meccanismo universale presente in ogni Sistema Operativo.

Uno script - o applicazione - CGI può essere scritto in qualsiasi linguaggio; è sufficiente che accetti e restituisca i dati secondo quanto previsto dalle specifiche di CGI. I linguaggi più comunemente utilizzati sono: Perl, C, C++, Java, Javascript.

Solitamente, è compito del Web server inserire gli header della *response*: l'unico compito dello s*cript CGI* è che deve generare uno header *Content-type* valido (Es.: *Content-type: text/html*).

Il prefisso **nph-** nel nome di una CGI (es.: *nph-test.pl*) sta a rappresentare il fatto che questa CGI fornirà un output già provvisto degli header della *response*: pertanto, il Web server non dovrà aggiungere nulla, e solamente inoltrare la risposta fornita dalla CGI direttamente al client.

5.1.1 Approfondimento

Gli script CGI vengono comunemente richiamati da pagine HTML.

Uno script CGI deve essere posizionato all'interno di directory standard nella root del Web server: la più comune è /cgi-bin/

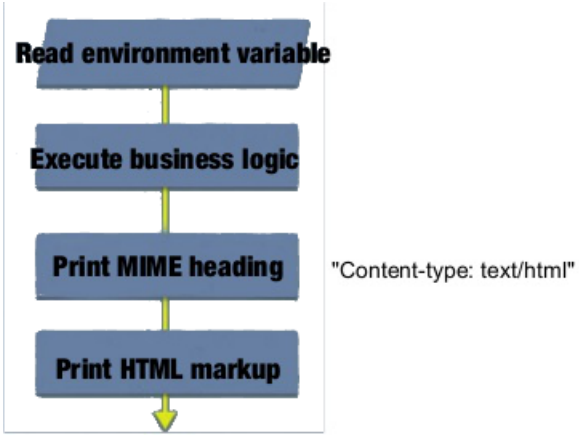
Un Web-server, per poter eseguire le CGI, deve saper distinguere tra richieste di pagine statiche ed invocazioni di applicazioni CGI.

L'estensione dello script va specificata esplicitamente.

Prima di richiamare uno script CGI, è necessario validare i dati che gli andranno passati in input: in caso contrario, lo script potrebbe ritornare un risultato errato. È possibile, e raccomandabile, validare i dati anche all'interno della CGI stessa.

Gli script CGI sono a volte interfacciati con una *Applet Java*, in modo tale da ricreare un vero ambiente client/server; gli script vengono eseguiti *server-side*, ma non devono per forza risiedere sullo stesso *host* che ospita il *Web server*.

- Gli script CGI non hanno modo di dialogare con il *Web server* durante la loro esecuzione:
  - Il Web server crea l'*environment* con i dati della *request* (ed altri dati che vedremo poi).
  - Il Web server richiama uno script CGI, passandogli l'*environment*.
  - Lo script CGI viene eseguito dal sistema, senza possibilità di dialogo tra esso ed il Web server.
  - L'output dello script CGI viene inviato al Web server.
  - Lo script CGI termina.
- L'esecuzione dello script CGI segue questi passi:



5.1.2 Cenni storici

Lo standard CGI è stata sviluppata al NCSA (National Center for Supercomputing Applications) durante lo sviluppo del primo Web server (NCSA HTTPD). Rob McCool potrebbe essere definito il padre di quello che nel 1993 diventò lo Standard CGI.

Fino a qualche anno fa, gli script CGI erano il principale metodo per l'elaborazione dei dati inviati tramite *form* HTML, piuttosto che per accedere ad un database. Sono ora deprecati e sostituiti da nuove tecnologie, quali CORBA (Common Object Request Broker Architecture) ed ILU (Inter-Language Unification).

5.1.3 Principali vantaggi

- Portabilità.
- Scalabilità.
- Indipendenza dal linguaggio.
- Forniscono interattività alle applicazioni Web.

5.1.4 Principali svantaggi

- Non conservano informazioni di stato: ogni richiesta viene gestita creando un nuovo processo, che poi termina una volta esaudita la richiesta, non conservando così nessuna informazione tra una chiamata e l'altra. Ad esempio, non gestisce le sessioni. Un workaround, che può però essere molto inefficiente, può essere quello di passare i dati tramite *hidden fields* nei form (quindi lato client). Altro problema: una connessione ad un database deve essere ricreata ogni volta.
- Utilizzano molta memoria in quanto lo script viene eseguito ex-novo ad ogni richiesta. Nel caso di linguaggi interpretati, l'overhead cresce ancora. In ogni caso, il sistema deve sempre creare il contesto del processo.
- Richiedono importanti abilità di programmazione.
- Se non programmati correttamente, possono compromettere la sicurezza del server su cui girano.
- La maggior parte degli script CGI sono conosciuti, liberi ed utilizzati da molti sviluppatori. Questo comporta che sia i loro punti di forza che le loro vulnerabilità sono note agli sviluppatori, che potrebbero approfittarsene.

5.2 Architettura

Una applicazione, per essere una CGI, può essere scritta in qualsiasi linguaggio di programmazione, e deve solamente:

- Leggere dallo *standard input* (stdin).
- Scrivere sullo *standard output* (stdout).

- Leggere le variabili di *environment*.

Molte delle limitazioni ascritte a CGI sono in realtà limitazioni di HTTP o di HTML.

5.3 Sicurezza

Sotto il profilo della sicurezza, gli script CGI sono potenzialmente molto pericolosi, in quanto hanno un accesso abbastanza libero al server su cui vengono eseguiti.

- Fondamentale accorgimento per aumentare la sicurezza consiste nel far girare il *Web server* con un utente di sistema dedicato e con permessi limitati.
- Un altro punto dolente non sta tanto in CGI stessa, quanto negli errori di programmazione: un codice scritto senza i dovuti accorgimenti potrebbe portare ad una escalation dei privilegi sul server e/o a danni allo stesso.
- In ultima, bisogna notare che vengono spesso incluse nelle CGI delle librerie scritte da altri, e magari scaricate da Internet. Prima di usarle, è altamente raccomandabile rivederne il codice, in quanto potrebbe contenere sia errori involontari, che codice malevolo inserito appositamente.

5.4 mod_perl

In Apache è possibile evitare l'overhead dovuto al caricamento del runtime Perl esternamente ad ogni richiesta tramite CGI, semplicemente caricando il runtime all'interno del Web server. Questo avviene caricando il modulo mod_perl all'interno di Apache.

5.5 FastCGI

FastCGI un tipo di gateway per Web servers che incrementa le prestazioni, caricando le CGI come processi sempre in esecuzione. È un protocollo aperto. Implementa un layer software che, quando riceve il controllo dal Web server, richiama gli script e i programmi conformi allo standard FastCGI.

Gli script FastCGI differiscono dagli script CGI per:

- **Processo sempre attivo**: una volta che uno script CGI ha completato la sua esecuzione, termina il processo. Uno script FastCGI resta invece attivo, in attesa (*wait*) di elaborare la prossima richiesta.
- **Meno accessi al disco**: restando il processo sempre attivo, lo script può salvare molti dati in memoria centrale anzichè sul disco, limitando così il numero di accessi a quest'ultimo.
- **Distribuzione dei processi**: FastCGI può distribuire i processi su più server. Da notare che questo non è sempre un procedimento immediato, in particolare per vaste applicazioni.

5.6 Esempio di una CGI (HTML + C)

5.7 Invocazione di un'applicazione CGI

5.7.1 Gestione della richiesta

Ogni *request* viene ricevuta dal Web server, il quale la analizza e decide se deve essere gestita direttamente da lui, oppure demandata ad uno script CGI. Noi analizzeremo questo secondo caso.

La richiesta, sia essa GET o POST, contiene una serie di headers, nonchè eventuali parametri. Vediamo nei prossimi paragrafi come vengono elaborate queste informazioni.

5.7.2 Parametri GET e POST

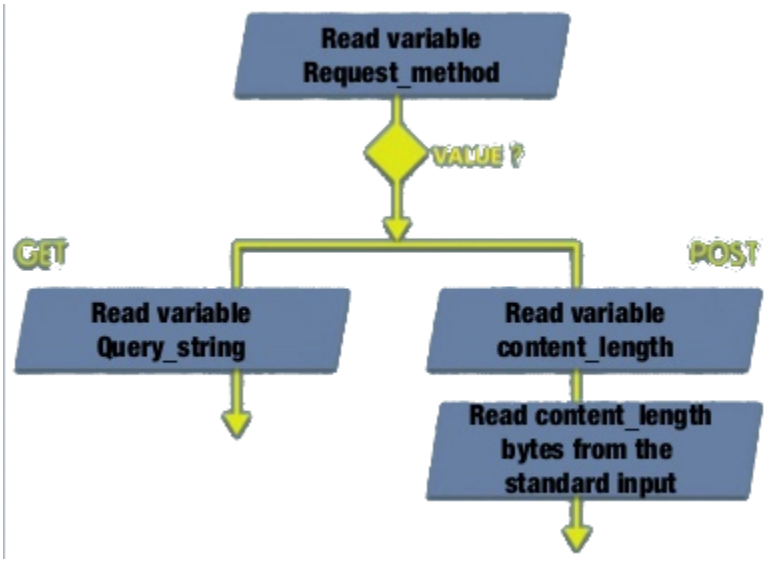
GET I parametri sono accodati all'URI.

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

POST I parametri sono inseriti nell'*entity body* della richiesta.
Richiede l'uso dei *form* HTML per consentire agli utenti di inviare i dati tramite POST.

```
POST /nome-cgi HTTP/1.0
Content-type: ....
Content-lenght: nnnn
[Altri header]
<CRLF>
<ENTITY BODY (i dati veri e propri)>
```

Decoding dei parametri Il decoding dei parametri avviene secondo la seguente logica:



5.7.3 Parametri Environment

Vi sono una serie di parametri che vengono gestiti in un contesto, detto *environment*, che corrisponde cioè ad una serie di variabili di sistema.
Il Web server salva in queste variabili gli header ed i parametri della *request*, come di seguito:

- SERVER VARIABLES:

  SERVER_SOFTWARE nome e versione del Web server.
  SERVER_NAME hostname o IP del server.
  GATEWAY_INTERFACE versione di CGI supportata.

- REQUEST VARIABLES:

  SERVER_PROTOCOL nome del protocollo di trasporto, e versione.
  SERVER_PORT porta a cui è stata inviata la *request*.
  REQUEST_METHOD metodo della richiesta HTTP.
  PATH_INFO informazione cosiddetta **extra path**.
  PATH_TRANSLATED traduzione del PATH_INFO da virtuale a fisico.
  SCRIPT_NAME URI dello script invocato.
  QUERY_STRING la stringa della query (ciò che è presente nell'URI dopo il carattere "?").
  REMOTE_HOST hostname o IP del client.

REMOTE_ADDR IP del client.

AUTH_TYPE tipo di autenticazione utilizzata dal protocollo.

REMOTE_USER username utilizzato per l'autenticazione.

REMOTE_IDENT username utilizzato per l'autenticazione, così come ritornato da *identd* (nel caso di utilizzo di auth. come da RFC 931).

CONTENT_TYPE ***content type***, in caso di *request* di tipo POST o PUT.

CONTENT_LENGTH la lunghezza, in byte, del contenuto (POST o PUT).

- HEADERS:

HTTP_USER_AGENT browser usato per inviare la *request*.

HTTP_ACCEPT_ENCODING mime-type dei file accettati dal client.

HTTP_ACCEPT_CHARSET charset accettati dal client.

HTTP_ACCEPT_LANGUAGE lingue accettate dal client.

Etc.

5.7.4 stdin

Da **stdin** lo script CGI può leggere tutto quello che il **client** invia dopo lo header, ovvero:

- Metodi GET e HEAD:
  - nulla.
- Metodi POST e PUT:
  - Potremo leggere un contenuto di tipo *CONTENT_TYPE* e di lunghezza *CONTENT_LENGTH*.

Possibili valori dell'header CONTENT_TYPE:

application/x-www-form-urlencoded I dati vengono salvati nell'*entity body* con la seguente forma (simile a quella della query GET):

nome_arg1=val_arg1&nome_arg2=val_arg2&..&nome_argn=val_arg_n

multipart/form-data I dati vengono salvati nell'entity body senza codifica. Vengono introdotte delle stringhe di escape per separare i vari parametri (stringhe, file, etc). Vedere l'esempio a fine capitolo.

5.7.5 stdout

L'output dell'applicazione viene o meno interpretato dal server Web.

Se lo script inizia per *nph-* allora l'output viene inviato direttamente al client, ed è compito dello script definire tutti gli header in maniera corretta.

In tutti gli altri casi è il Web server che invia la *linea di stato* e tutti gli altri campi dello header, eccetto per i seguenti campi, che devono essere ritornati dallo script CGI:

- **Content-type**: è il mime-type del documento ritornato dallo script.
- **Location**: indica al Web server che lo script non ritorna un contenuto, bensì dove trovare il documento creato:
  - Se è un URI, allora il Web server redireziona il client al nuovo indirizzo.
  - Se è un path, il Web server recupera tale risorsa e la invia al client, come se questi l'avesse richiesta direttamente.

5.8 Esempi lato server

5.8.1 Hello World

Listing 2: Esempio CGI in C

```
/************************************************************
* 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,"\015\012",2);
        while(1,"<html><body>Hello World!</body></html>",38);
        close(file);
}
```

5.8.2 Lettura di un file

Listing 3: Esempio CGI in C

```
/************************************************************
* 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);

        write(1,"Content-type: text/html\015\012",25);
        write(1,"\015\012",2);

        if (file<0) {
                return 0;
        }

        while ((n = read(file, buf, BUFLEN)) > 0)
                write(1, buf, n);

        close(file);
}
```

5.8.3 Lettura di un file [nph-]

Listing 4: Esempio CGI in C

```
/************************************************************
* 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,"Content-type:␣text/html\015\012",25);
                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);
}
```

5.8.4 Accesso a variabili Environment

Listing 5: Esempio CGI in C

```
#!/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
```

5.9 Esempi lato client (Tramite form HTML)

Dato lo script CGI testform.cgi:

Listing 6: testform.cgi

```
#!/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
```

Valutiamo nei prossimi paragrafi degli esempi che fanno riferimento al codice sopra listato.

5.9.1 HTML + GET

Pagina HTML:

```
<html>
    <body>
        <FORM METHOD="GET" ACTION="testform.cgi" ENCTYPE="application/x-
     www-form-urlencoded">
        ... input della form ...
        </FORM>
  </body>
</html>
```

Output della CGI:

```
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 =
```

URL visualizzato nel browser:

```
http://www.dsi.unive.it/~taw/lezione3/testform.cgi? nome=&cognome=&linee=Linea1%0D%0ALinea2%0D%0ALinea
```

5.9.2 HTML + POST con codifica URL encoded

Pagina HTML:

```
<html>
    <body>
        <FORM METHOD="POST" ACTION="testform.cgi" ENCTYPE="application/x-
     www-form-urlencoded">
        ... input della form ...
        </FORM>
  </body>
```

</html>

Output della CGI:

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

5.9.3 POST con codifica Multipart

Nel caso di sottomissione di file, il metodo migliore è usare la codifica multipart. In questo caso i dati non vengono codificati (ovvero convertiti in testo html-safe, ad es. " " diventa "%20"), ma i singoli parametri compaiono nello standard input dell'applicazione separati da un'opportuno delimitatore.

Pagina HTML:

```
<html>
    <body>
        <FORM METHOD="POST" ACTION="testform.cgi" ENCTYPE="multipart/form-
     data">
        ... input della form ...
        </FORM>
  </body>
</html>
```

Output della CGI:

Listing 7: Output

```
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
23/6/2014 www.dsi.unive.it/~taw/Lezione5.htm
http://www.dsi.unive.it/~taw/Lezione5.htm 10/11
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<br>
Franchetto Stefano 783094<br>
De Nes Francesco 784105<br>
Galesso Daniele 786466<br>
<br>
Gruppo 2: Progetto 2<br>
<u>Rota Bulo' Samuel</u> srotabul@dsi.unive.it<br>
Casagrande Massimo mcasagra@dsi.unive.it<br>
Fornaro Emanuele  efornaro@dsi.unive.it<br>
Gerarduzzi Michele mgerardu@dsi.unive.it<br>
Luisetto Andrea aluisett@dsi.unive.it<br>
Niero Luca  lniero@dsi.unive.it<br>
<br>
Gruppo 3: Progetto 3<br>
<u>Trolese Paolo</u><br>
Orsenigo Marco<br>
Favaro Susanna<br>
Busolin Katia<br>
<br>
Gruppo 4: Progetto 2<br>
<u>Pietro Ferrara</u> 784578<br>
Teresa Scantamburlo 783915<br>
Patrizia Zucconi 783916<br>
Luigi Runfola 784386<br>
<br>
Gruppo 5: Progetto 1<br>
<u>Casotto Briano</u> bcasotto@dsi.unive.it 785251<br>
Ravagnan Emiliano eravagna@dsi.unive.it 786353<br>
Marvulli Donatella dmarvull@dsi.unive.it 783738<br>
Ramelintsoa Carole cramelin@dsi.unive.it 786413<br>
Scavazzon Marco mscavazz@dsi.unive.it 786242<br>
<br>
Gruppo 6: Progetto 3<br>
Fiori Alessandro afiori@libero.it 777191<br>
Gastaldi Riccardo rigasta@tin.it  <br>
Bernacchia Francesco  s.checco@libero.it 778611<br>
Pozzobon Giovanni GiovanniP@aton.it 791120<br>
Rampazzo Pietro  rampaz79@tin.it<br>
Cian Francesco francesco.cian@t-systems.it 778885<br>
<br>
Gruppo 7:  Progetto 4<br>
<u>Piccin Massimiliano</u><br>
Carraretto Alessandro<br>
Ministeri Dario<br>
Rui Massimo<br>
<br>
Gruppo 8: Progetto 4<br>
<u>Corradin Michele</u><br>
Borin Francesca<br>
23/6/2014 www.dsi.unive.it/~taw/Lezione5.htm
http://www.dsi.unive.it/~taw/Lezione5.htm 11/11
Bordin Fabio<br>
Scomello Stefano<br>
```

```
<br>
Gruppo␣9:␣Progetto␣5<br>
<u>Carrer␣Andrea</u>␣783089<br>
Longo␣Irene␣783528<br>
Vianello␣Michele␣784026<br>
Molinari␣Marco␣784162<br>
<br>
Gruppo␣10:␣Progetto␣5<br>
<u>Leonardo␣Scattola</u>␣lscattol@dsi.unive.it<br>
Filippo␣Cervellin␣fcervell@dsi.unive.it<br>
Roberto␣Fietta␣rfietta@dsi.unive.it<br>
Luca␣Giacomazzi␣lgiacoma@dsi.unive.it<br>
Lino␣Possamai␣lpossmai@dsi.unive.it<br>
<br>
<br>
<br>
<br>
</body>
</html>
----------------------------24464570528145
Content-Disposition:␣form-data;␣name="end"
valore␣ultimo␣campo
----------------------------24464570528145--
```

6 Applet

6.1 Definizione e principali proprietà

L'applet è un particolare tipo di programma Java, che può essere scaricato ed eseguito da un qualsiasi browser che sia abilitato a eseguire codice Java.

Le applet sono perlopiù deprecate per quanto concerne lo sviluppo di siti Web, mentre tornano utili in determinati portali, locali piuttosto che remoti. Esempio tipico, la gestione di sistemi di videosorveglianza via Web.

NOTA BENE: quando un utente visualizza una pagina web contenente un applet, quest'ultima viene eseguita localmente nella VM sulla macchina dell'utente e non remotamente.

Per tale motivo, l'applet è soggetta a delle limitazioni per renderne l'esecuzione più sicura.:

Un'applet pertanto **non** può:

- Scrivere e/o leggere dal disco locale;
- Aprire connessioni con server diversi da quello da cui proviene;
- Scoprire informazioni private riguardo all'utente.

Per ovviare a tali limitazioni è possibile firmare l'applet, tramite un sistema a doppia chiave privata e pubblica.

6.2 Sintassi HTML

Un browser deve essere istruito su come visualizzare un applet, e questo avviene tramite l'utilizzo di un tag html specifico, <applet>, tramite il quale viene associato il file .class alla pagina web. All'interno del tag <applet> vengono definiti alcuni attributi che sono:

- *code="..."* contiene l'url relativo al file .class da caricare;
- *width* ed *heigth* che contengono la larghezza e l'altezza da riservare all'applet nella pagina web.

Esempio di uso del tag <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>
```

Purtroppo però il tag <applet> è utilizzabile solo per applet che utilizzano la versione 1 del linguaggio Java, e questo non rende disponibili svariate funzionalità, quali:

- Swing (a meno di non scaricare le classi che la contengono);
- Java2D;
- collezioni(List, Map, ...);
- ottimizzazioni per rendere più veloce l'esecuzione del codice.

Pertanto per ottenere i vantaggi presenti nella versioni di Java successive alla 1, è necessario usare tag diversi:

- in IE : <object>

- in NetScape <embed>

Il modo più recente per inserire un'applet all'interno di una pagina web è il seguente:

```
<object id="appletLake" codetype="application/java" codebase="applet_dir/" width="500" height="400"
>
    <param name="image" value="myimage.jpg">
    <!–html alternativo –>
        <img src="http://html.it/guide/esempi/guida_html/multimedia/java/logo.gif" width="263"
     height="65" alt="html.it">
  <!– fine html alternativo –>
</object>
```

Come si può vedere dal codice, al giorno d'oggi si utilizza sempre il tag <object> per inserire un applet all'interno di una pagina web, a cui però vengono aggiunti i seguenti attributi:

- codetype: che stabilisce il tipo di oggetto che stiamo inserendo, nel nostro caso è "application/java";
- codebase: indica l'indirizzo della cartella che contiene il file .class della nostra applet;
- width ed heigth: che sono, come per il tag <applet> le dimensioni del box in cui verrà eseguita la nostra applicazione.

6.3 Sintassi Java

6.3.1 Applet e jsp

La gestione dei tag per le applet viene gestita 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.

6.3.2 Programmazione di un'Applet

```
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 in java è una sottoclasse di **awt.Panel**, il che implica che Applet ha come layout manager di default il Flow Layout Manager, il quale dispone gli elementi grafici da sinistra verso destra, con allineamento centrale.

Inoltre Applet eredita le variabili e i metodi delle superclassi **Component**, **Container** e **Panel**.

6.4 Ciclo di vita di una Applet

Come le servlet anche l'Applet ha un ciclo di vita definito da una serie di metodi che vengono invocati automaticamente dalla JVM dove è in esecuzione l'applet.

Questi metodi sono:

- init(): quando si apre un documento contenente un'applet, viene invocato automaticamente il metodo init() per l'inizializzazione della stessa. Tale metodo può essere utilizzato anche per inizializzare le variabili. Il metodo init() viene chiamato solo una volta per applet;
- start(): quando un documento contentente un'applet viene aperto, viene invocato il metodo start() subito dopo al metodo init(), per far iniziare l'esecuzione dell'applet, questo metodo viene richiamato ad esempio ogniqualvolta l'applet viene rivisitato;
- stop(): questo metodo viene invocato automaticamente quando l'utente esce dalla pagina in cui è sita l'applet, il suo scopo è bloccare attività che possono rallentare il sistema quando l'utente non le utilizza. Forma pertanto una coppia con il metodo start(), il quale attiva una funzione, mentre stop() la disattiva;
- destroy(): questo metodo viene invocato quando l'applet viene dismessa, ad esempio quando si chiude il browser, e prima che venga invocato destroy() viene invocato il metodo stop(). Eseguendo il metodo destroy vengono rilasciate tutte le risosrse relative all'applet;
- paint(): questo metodo viene invocato direttamente dal browser dopo init() e start(), viene invocato ogni volta che il browser ha bisogno di ridipingere l'applet;
- repaint(): invoca il metodo update() al più presto, permettendo di ridipingere l'applet. Di default pulisce lo sfondo e richiama paint().

Quindi il ciclo di vita di un'applet può essere riassunto così:
init()->start()->paint()->stop()->destroy().

6.5 Esempio di comunicazione Servlet/Applet

Uno dei problemi che può essere risolto tramite le applet è la possibilità di rendere più reattiva la comunicazione tra server e browser, il protocollo HTTP infatti non permette di avere un modo semplice con cui il server avvisa in maniera attiva il client che sono disponibili nuove informazioni.

Partiamo dal presupposto che la sandbox presente nel browser impedisce all'applet di aprire connessioni con un server diverso da quello da cui è stata scaricata l'applet.

Inoltre, ipotizzando la presenza di un firewall, possiamo presuppore che quest'ultimo impedisca connessioni TCP su porte diverse da quella standard (la porta 80). Pertanto l'unica porta aperta è la 80, su cui è in attesa di connessioni il server HTTP, il che ci porta a usare il protocollo Http come protocollo di comunicazione tra Applet e Servlet.

Per capire meglio la relazione tra applet e servlet, utilizziamo un esempio:

Listing 8: PushApplet.java

```
1  import java.awt.*;
2  import java.applet.*;
3  import java.awt.event.*;
4  import java.io.*;
5  import java.net.*;
6  public class PushApplet extends Applet implements ActionListener
7  {
8    TextArea taResults;
9    Button btnStart;
10   public void init()
11   {
12     setLayout(new FlowLayout(FlowLayout.LEFT));
13     btnStart = new Button("Start");
```

```
14      btnStart.addActionListener(this);
15      add("North", btnStart);
16      taResults = new TextArea(10, 80);
17      add("Center", taResults);
18    }
19    public void execute()
20    {
21      char[] buf = new char[256];
22      try
23      {
24        URL url = new URL(getDocumentBase(),"PushServlet");
25        URLConnection uc = url.openConnection();
26        uc.setDoInput(true);
27        uc.setUseCaches(false);
28
29        InputStreamReader in = new InputStreamReader(uc.getInputStream());
30        int len = in.read(buf);
31        while (len != -1 )
32        {
33          taResults.replaceRange(new String(buf,0,len),0,len);
34          len = in.read(buf);
35        }
36        in.close();
37      }
38      catch(MalformedURLException e)
39      {
40        taResults.setText(e.toString());
41      }
42      catch(IOException e)
43      {
44        taResults.setText(e.toString());
45      }
46    }
47    public void actionPerformed(ActionEvent ae)
48    {
49      execute();
50    }
51  }
```

6.6 Esempio di testo scorrevole

Listing 9: Scroll.java

```
1  public class Scroll extends Applet implements Runnable
2  {
3          Thread t=null;
4          String temp;
5          String text = "Questo␣testo␣scorre";
6          long velocitaBattitura=100;
7
8          boolean cont = true;
9          public void start()
10         {
11                 t = new Thread(this);
12                 cont = true;
13                 t.start();
14         }
15         public void stop()
```

```
16          {
17                  cont = false;
18                  t.interrupt();
19                  t=null;
20          }
21          public void run()
22          {
23                  while (cont)
24                  {
25                          for(int j=0; j<=text.length(); j++)
26                                  {
27                                          temp = text.substring(j)+ "␣" + text.substring(0,j);
28                                          repaint();
29                                          try{t.sleep(velocitaBattitura);}
30                                          catch(InterruptedException eint){if (!cont) break;}
31                                  }
32                  }
33          }
34          public void paint(Graphics g)
35          {
36                  Font f = new Font("Arial",0,16);
37                  FontMetrics fm = g.getFontMetrics(f);
38                  int w = fm.stringWidth(temp);
39                  g.setColor(Color.white);
40                  g.fillRect(0,0,w+10,30);
41                  g.setColor(Color.black);
42                  g.setFont(f);
43                  g.drawString(temp, 5, 20);
44          }
45
46  }
```

6.7 Sintassi tag Applet

Attributi del tag applet:

- archive = ListaArchivio: classi o risorse che saranno preloaded, si può indicare una lista di file .jar dai quali estrarre l'applet;
- code = MioFile.class: nome del file che contiene il compilato
- width/heigth = pixels: misura in pixels del box in cui sarà visualizzata l'applet nella pagina;
- codebase = codebaseURL: directory che contiene il sorgente dell'applet;
- alt = TestAlternativo: testo alternativo che appare se il browser legge l'applet ma non può visualizzarla;
- name = nomeIstanzaApplet: permette di identificare diversi applet nella stessa pagina;
- align = allineamento: allineamento dell'applet;
- vspace/hspace = pixels;
- <param name=*nomeAttributo* value=*valoreDesiderato*>: valori specificati dall'esterno, sarà necessario un getParameter nell'applet.

Parte III
Javascript, RMI

7 Javascript

7.1 Introduzione e cenni storici

Nasce originariamente con il nome di "LiveScript" e viene sviluppato da Netscape come potenziamento del codice html semplice. Grazie a questo linguaggio di scripting diventava infatti posssibile alleggerire l'elaborazione dei dati lato server, delegando alcuni compiti - come il controllo dei dati personali e la validazione dei form - al browser dell'utente. Un altro utilizzo di LiveScript è quello di aggiungere effetti grafici alle pagine web, che fino a quel momento ne eraho sprovviste.

Successivamente, dalla collaborazione tra Netscape e Sun MicroSystems, nasce un nuovo linguaggio di scripting: **Javascript**, il cui intento è di fornire un'architettura completa anche lato client,

JS fa fa uso della stessa sintassi fondamentale di Java. Esiste inoltre una versione Microsoft di Javascript denominata Jscript.

Definiamo Javascript tramite le sue caratteristiche fondamentali:

- è un linguaggio interpretato: vuol dire che Javascript può essere eseguito solamente tramite l'utilizzo di un interprete, nel nostro caso, tale programma è il browser stesso. Questo implica che gli script in Javascript sono tendenzialmente più lenti di un programma direttamente eseguibile perchè compilato, inoltre essendo l'interprete degli script il browser stesso, è probabile che si ottengano risultati leggermente diversi su browser diversi. Tuttavia il vantaggio più evidente consiste nel fatto che l'aggiornamento del codice è pressoche immediato, basta infatti modificare il codice e riaricare la pagina in cui esso risiede per visualizzare le modifiche. Javascript è pertanto un linguaggio lato client, in quanto il codice è eseguito dal browser del client e non dal server;

- è un linguaggio loosely typed o scarsamente tipizzato: caratteristica comune a tutti i linguaggi di scripting è quella di poter omettere il tipo di una variabile in fase di dichiarazione, il che permette la scrittura di un codice più flessibile, permettendo di assegnare svariati tipi a una stessa variabile senza bisogno di type casting ;

- è **basato** sugli oggetti: in Javascript tutti gli elementi della pagina web html sono considerati oggetti secondo lo standard DOM o Document Object Model. Ogni oggetto ha proprie caratteristiche peculiari dette proprietà, determinate azioni che possono essere compiute su di esso, tramite i metodi e di eventi ai quale è soggetto. Tuttavia Javascript NON è un linguaggio **orientato** agli oggetti, in quanto non è possibile nè definire sottoclassi nè è possibile definire l'ereditarietà di una classe rispetto ad un altra già esistente.

7.2 Principali utilizzi

I principali utilizzi di Javascript sono riassumibili come segue:

- per arricchire le normali pagine html con effetti grafici come animazioni, roll over e quant'altro;

- per la convalida dei dati di un modulo prima dell'invio al server;

- per lo sviluppo di complete applicazioni lato client.

Oltre alla scrittura di applicazioni lato client è possibile scrivere anche applicazioni lato server con Javascript con l'ambiente SSJS o Server-Side JavaScript di Netscape o sciluppando applicazioni in ambiente ASP.

7.3 JS per validazione dell'input

Per integrare del codice javascript all'interno di una pagina html dobbiamo usare l'apposito tag:

<script language="javascript"></script>

Come possiamo vedere dal codice della funzione controlla() gli elementi del DOM sono acceduti tramite il loro attributo nome, e su questi vengono lanciati i metodi. Inoltre la funzione viene eseguita a partire dal codice html, ossia quando si completa il riempimento del form, alla submit dello stesso viene lanciata la funzione per il controllo di quanto inserito.

Listing 10: JS controlla()

```
<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>
                        <INPUT TYPE="SUBMIT" VALUE="INVIA">
                        <INPUT TYPE="RESET" VALUE="CANCELLA">
                </FORM>
        </BODY>
</HTML>
```

Al giorno d'oggi si preferisce usare il metodo *getElementByID()*, invece di usare select() e poi focus();

7.4 Pushlets

7.4.1 Introduzione

Le pushlets sono una tecnologia basata su HTTP streaming per ovviare al problema di comunicazione da server a client una volta che la connessione è stata chiusa. La base del funzionamento di queste ultime è la seguente, invece di chiudere la connessione http dopo aver inviato il contenuto della pagina html, la connessione viene mantenuta aperta mentre vengono inviati nuovi dati dal server al client.

Per fare ciò è possibile procedere nella seguente maniera:

- il browser richiede la pagina alla Servlet/JSP;
- la Servlet/JSP fornisce la risposta con il codice html completo (comprensivo del tag </html>);
- la connessione viene mantenuta attiva e i dati eventualmente ancora da inviare vengono forzatamente inviati;
- il browser, che è in grado di visualizzare la pagina html completa, rimane in attesa di dati della connessione;
- la Servlet/JSP quando vuole far caricare una pagina, invia nella ?vecchia? connessione un codice Javascript che forza il caricamento della pagina;
- il browser esegue il codice JS non appena lo riceve e quindi ricarica la pagina quando vuole il server;

7.4.2 Esempio di pushlet

Listing 11: Pushlet

```
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();
```

```
20              /* il metodo flushBuffer() forza qualsiasi contenuto del buffer a essere scritto
                    sul client, una chiamata a questo metodo invia direttamente la response e cio'
                     vuol dire che verranno scritti lo status code della stessa e gli header
                    relativi */
21              response.flushBuffer();
22
23              try {
24                      Thread.sleep((int)(1000*Math.random()*10));
25              }
26              catch( InterruptedException e) {
27              }
28
29              out.println("<script language=\"JavaScript\">");
30              out.println("document.location='http://localhost:8080/servlet/Pushlet';");
31              out.println("</script>");
32              out.close();
33      }
34 }
```

7.4.3 Sull'utilizzo di flushBuffer() anzichè flush()

Da un post del 2003: I have seen some people having trouble flushing the scripts to pushlet clients. Its not a bug in tomcat but in BrowserPushletClientAdapeter (refer to http://znutar.cortexity.com/BugRatViewer/ShowBugActions/18).

This is fixed by calling response.flushBuffer() instead of out.flush ().

According to the servlet specs (tomcat) out.flush will flush the buffer in the output stream to response and not to the wire. modifying the BrowserPushletClientAdapeter class worked fine for me.

Aggiornamento Tale bug è attualmente risolto, pertanto è sufficiente richiamare **out.flush()**.

7.5 AJAX (Asynchronous-Javascript-XML)

7.5.1 Introduzione

Ajax è una serie di tecnologie denominate RIA (rich internet application), che hanno dimostrato di saper migliorare l'interattività delle applicazioni web, fornendo all'utente una migliore esperienza d'uso. La tecnologia Ajax si basa su uno scambio di dati in background tra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricarimento della stessa da parte dell'utente.

Le tecnologie usate da AJAX comprendono:

- una presentazione standard creata con XHTML e CSS;
- un display dinamico di iterazione DOM;
- manipolazione e scambio di dati con XML e XSLT;
- DHTML o Dynamic HTML che aiuta a caricare i forms in modo dinamico, con comandi come , e altri elementi html;
- XMLHttpRequest per il recupero asincrono dei dati;
- Javascript che lega tutto insieme;
- SOAP (simple object access protocol) per dialogare con il server;
- PHP o un altro linguaggio di scrittura da utilizzare sul server (servlets in java).

La sequenza con cui vengono gestite le richieste AJAX è la seguente:

- il browser richiede la pagina html;
- il server invia la pagina (sia essa statica o dinamica);
- il client visualizza la pagina, se quest'ultima contiene richieste AJAX queste vengono inoltrate al server in un thread secondario in modo asincrono, permettendo quindi all'utente di effettuare ulteriori operazioni sulla pagina;
- quando il server invia la risposta al client della richiesta inoltrata asincronamente, il thread secondario visualozza il risultato nella pagina html modificandola tramite DOM.

7.5.2 L'oggetto XMLHttpRequest

L'oggetto XMLHttpRequest è usato per scambiare dati con un server in background, tale oggetto è definito come un sogno per i programmatori in quanto:

- aggiorna una pagina senza il bisogno di ricaricare la pagina;
- richiede dei dati al server dopo che la pagina è stata caricata;
- riceve dei dati dal server dopo che la pagina è stata caricata;
- invia dati al server in background.

Una volta creato l'oggetto XMLHttpRequest, tramite il seguente codice:

```
if (window.XMLHttpRequest)
    { // Mozilla, Safari, ... http_request = new XMLHttpRequest(); }
else if (window.ActiveXObject)
    { // IE http_request = new ActiveXObject("Microsoft.XMLHTTP"); }
```

Bisogna dire all'oggetto XMLHttpRequest quale funzione JS elaborerà il codice XML, per fare ciò si deve impostare la proprietà onreadystatechange dell'oggetto con il nome della funzione JS, nel seguente modo:

```
http_request.onreadystatechange= nomeFunzione
```

in tal modo si assegna un riferimento alla funzione ma non la si richiama.
Per inviare la richiesta bisogna utilizzare i metodi open() e send():

```
http_request.open('GET', 'http://www.nomeserver.org/qualsiasi.file', true);
http_request.send(null);
```

OPEN: apre la connessione con il server.

- *open(method,url,async)*
- il primo parametro di http_request_open è il metodo della richiesta http: GET, POST, HEAD,PUT e via dicendo;
- è l'URI della risorsa cercata, nota bene che non si può richiedere una risorsa che risiseda su un dominio diverso da quello della pagina corrente;
- il terzo parametro deve essere definito false per una richiesta asincrona e true per una richiesta sincrona.

SEND: invia la richiesta al server.

- *send(string)*
- Parametro *string*: utilizzato solamente per richieste di tipo POST.

8 RMI

8.1 Introduzione

8.1.1 Breve introduzione alle RPC

Prima di introdurre le RMI è bene definire le loro antesignane, ossia le RPC o Remote Procedure Call, il cui obbiettivo era la possibilità di consentire l'invocazione di procedure(funzioni) su altre macchine, tramite tale strumento la comunicazione tra processi diversi può essere ricondotta a un modello fondamentalmente sincrono.

Per ottenere questo risultato le RPC utilizzano due mediatori, il primo, il client stub, implementa sulla macchina locale l'interfaccia attraverso cui la funzionalità remota può essere nvocata e il secondo, il server stub che implementa la reale funzionalità.

Il client stub impacchetta i parametri e li invia al server utilizzando un canale di comunicazione, mentre il server stub spacchetta i parametri e li passa alla procedura reale, per i parametri di ritorno si ha una situzione simmetrica.

Il processo RPC può essere descritto pertanto tramite una sequeza di azioni di questo tipo:

1. La procedura chiamante invoca il client stub nel modo usuale(come se fosse una chimata locale);
2. Il client stub costruisce il messaggio e invoca il sistema operativo locale(usando ad esempio la libreria socket);
3. Il sistema operativo locale invia il messaggio al sistema operativo remoto;
4. Il sistema operativo remoto consegna il messaggio al server stub;
5. Il server stub spacchetta i parametri e chiama il server(effettua la procedura effettivamente);
6. Il server esegue il compito desisderato e restituisce i risultati al server stub;
7. Il server stub impacchetta i risultati e invoca il sistema operativo locale;
8. Il sistema operativo del server invia il messaggio al sistema operativo del client;
9. Il sistema operativo del client consegna il messaggio al client stub;
10. il client stub spacchetta il messaggio e lo restituisce al chiamante.

8.1.2 Introduzione RMI in Java

Le RMI possono essere considerate come un'estensione in chiave Object Oriented delle RPC.

La Remote Method Invocation permette infatti a un oggetto funzionante su una JVM di invocare metodi di un oggetto presente su un'altra JVM.

Le applicazioni RMI sono composte da due differenti programmi: uno per il server e uno per il client:

- il programma lato server crea alcuni oggetti remoti, ne rende accessibili i riferimenti e aspetta che il client invochi metodi su questi oggetti;
- il programma lato client ottiene i riferimenti a uno o più oggetti remoti presenti sul server e quindi invoca metodi su di essi.

Un'applicazione che fa utilizzo di RMI si dice applicazione distribuita e deve avere la possibilità di svolgere le seguenti funzioni:

- Localizzare gli oggetti remoti: per fare ciò si ricorre all'RMI Registry, un servizio di naming che permette al client di ottenere riferimenti ad oggetti remoti;
- Comunicare con oggetti remoti: in RMI questo avviene tramite l'implementazione di un'interfaccia da parte di client e server e tramite la creazione di uno stub lato client e di uno skeleton lato server.
- Oltre a comunicare con oggetti remoti una RMI deve poter caricare classi per oggetti che sono passati come parametri o restituiti come valore.

8.1.3 Pro e contro

Per valutare i pro ed i contro di RMI è necessario confrontare questo metodo con le RPC. Altro possibile confronto è con i Web services.

N.B.: RMI funzionano solo su client Java / server Java. Se si cambia piattaforma bisogna cambiare sistema.

8.2 Implementazione

8.2.1 Oggetti remoti

In Java un oggetto remoto è un oggetto i cui metodi possono essere invocati da remoto ed è descritto da uno o più interfacce: RMI di fatto consiste nell'invocazione dei metodi di un'interfaccia remota su un metodo remoto.

L'interfaccia remota, pertanto deve soddisfare i seguenti requisiti:

- estendere l'interfaccia java.rmi.Remote;
- dichiarare metodi che includano le eccezioni nella clausola throws del tipo java.rmi.RemoteException e che definiscano gli oggetti remoti passati come paramentri o restituiti come risultato mediante interfacce remote.

Esempio di interfaccia:

```
public interface MiaInterfaccia extends java.rmi.Remote {
    String getRisposta(String risposta) throws java.rmi.RemoteException;
}
```

A differenza delle chiamate locali le chiamate remote possono fallire a causa di:

- problemi di comunicazione;
- errori nel passaggio di parametri e nel recupero di valori;
- errori di protocollo;

Una volta definita l'interfaccia per implementare il comportamento di un oggetto remoto si estende la classe UnicastRemoteObject

(esiste anche MulticastRemoteObject), la quale può implementare più interfacce remote e può anche contenere metodi non definiti nelle interfacce stesse con la limitazione che tali metodi non potranno essere invocati da remoto.

Esempio di implementazione:

```
public class OggettoRemoto extends java.rmi.server.UnicastRemoteObject implements MiaInterfac-
cia {
    public String getRisposta(String risposta) throws java.rmi.RemoteException {
        return risposta;
    }
}
```

8.2.2 Stub e Skeleton

Uno stub è una rappresentazione locale di un oggetto: il client invoca cioè i metodi dello stub il cui compito sarà quello di invocare i metodi dell'oggetto remoto che rappresenta.

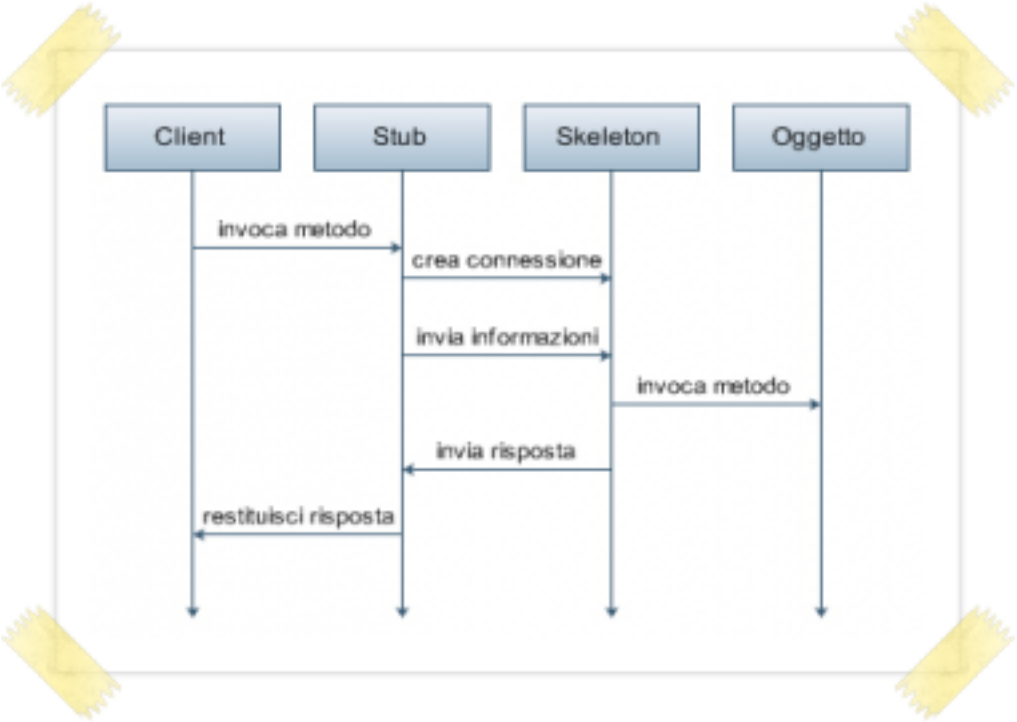
Quando viene invocato un metodo, lo Stub provvede a compiere le seguenti operazioni:

- inizia una connessione con la JVM che contiene l'oggetto remoto;
- trasmette i parametri alla JVM;
- aspetta il risultato dell'invocazione del metodo;
- legge il risultato o l'eccezione generata;

- restituisce il risultato a chi ha invocato il metodo;

Ogni oggetto remoto ha sulla propria JVM un corrispondente dello Stub - detto **Skeleton** - il cui scopo è quello di:

- leggere i parametri di un'invocazione remota;
- invocare il metodo remoto dell'oggetto;
- restituire il risultato al chiamante;



Stub e Skeleton non vengono scritti dal programmatore ma generati dal compilatore RMI tramite il comando:

```
rmic [-vcompat] oggettoremoto
```

il quale genera le classi OggettoRemoto_Stub.class e OggettoRemoto_Skel.class.

8.2.3 Registrazione di un oggetto

Come detto in precedenza il client deve essere in grado di localizzare il server che fornisce il servizio stesso.

Esistono svariati approcci al problema che sono:

- il client conosce già l'indirizzo del server;
- l'utente indica all'applicazione client come raggiungere il server;
- un naming service noto al client è in grado di fornire informazioni relative alla locazione di determinati servizi;

In java questo naming service é definito RMI Registry, il quale consente al client di ottenre i riferimenti degli oggetti remoti.

Ogni volta che un oggetto viene registrato (ovverosia viene creata un'associazione nell'RMI Registry (bind) fra un nome url formatted e un oggetto) i client posssono farne richiesta attraverso il nome ed invocarne i metodi.

Tale operazione però può generare i seguenti errori:

- AlreadyBoundException se il nome logico è già utilizzato;
- MalformedURLException per errori di sintassi dell'URL dell'oggetto remoto;

- RemoteException negli altri casi.

Esempio di binding di un nome:

```
...
OggettoRemoto oggettoremoto = new OggettoRemoto();
Naming.rebind("//localhost/OggettoRemoto ",oggettoremoto);
...
```

Una volta che il server ha registrato l'oggetto, il client può recuperare un riferimento a questo tramite il metodo *Naming.lookup(String name)*, dove *name* è nella forma: *//host:port/nomeoggettoremoto*. Ad esempio:

```
MiaInterfaccia oggetto= (MiaInterfaccia)Naming.lookup("//localhost/OggettoRemoto");
```

e quindi invocare richiamare i metodi dell'oggetto:

```
String risposta = oggetto.getRisposta("risposta");
```

La classe naming offre anche altri metodi che sono:

- **rebind**(String name) sovrascrive una registrazione (in realtà viene utilizzato anche per la prima registrazione);
- **unbind**(String name) elimina la registrazione di un oggetto;
- **list**(String name) ottiene una lista di oggetti registrati;

8.2.4 Sicurezza

Per rendere maggiormente sicure le applicazioni RMI è possibile impedire che le classi scaricate dal server ed eseguite sul client effettuino operazioni per le quali non sono state preventivamente abilitate.

Per tale motivo una delle prime cose che client e server dovrebbero fare è quella di "installare" sul sistema un Security Manager, il quale preleva i permessi da un file di policy specificato al momento del lancio delle applicazioni.

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

Compilando poi con le opzioni:

```
java -Djava.security.policy = echo.policy Server java -Djava.security.policy = echo.policy Client
```

8.2.5 Procedura e codice annesso

La procedura per poter usare una RMI può essere sintetizzata nei seguenti passaggi:

1. definire un'interfaccia con i metodi che devono essere invocati remotamente, tale interfaccia deve estendere l'interfaccia java.rmi.Remote e ogni metodo dell'interfaccia così definita deve lanciare un'eccezione del tipo java.rmi.RemoteException;
2. creare l'oggetto remoto di modo che estenda *UnicastRemoteObject* oppure *MulticastRemoteObject* ed estenda l'interfaccia precedentemente definita;
3. l'oggetto remoto deve avere un costruttore ed implementare i metodi definiti dall'interfaccia;
4. nel server bisogna creare e installare un security manager;
5. nel server bisogna creare un'istanza dell'oggetto remoto e registrare l'oggetto remoto nell'*RMI registry* tramite la funzione Naming.rebind(String name);
6. nel client va recuperato il riferimento all'oggetto remoto tramite la funzione: Naming.lookup(String name), va inoltre invocato il metodo nell'oggetto remoto;
7. vanno quindi compilati i files;

8. creati Skeleton e Stub compilando l'oggetto remoto tramite l'RMI compiler;
9. va lanciato il registro RMI;
10. va lanciato il server;
11. va lanciato il client.

8.3 Considerazioni

Come visto i metodi invocati remotamente possono avere dei tipi di ritorno, i metodi inoltre possono avere parametri e più in generale il tipo dell'oggetto remoto può beneficiare di polimorfismo. Ma cosa succede se la JVM non conosce il tipo reale del valore di ritorno, dei parametri o dell'oggetto stesso? Risulta possibile configurare il registry di modo che il bytecode contenente il necessario al client per conoscere il tipo del server venga scaricato dal client, in tal modo è possibile utilizzare il polimorfismo tra macchine diverse.

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

- non è necessario specificare un protocollo;
- se non viene specificato IP o hostname viene assunto un host locale;
- si può specificare una porta se questa è diversa da quella di default 1099;
- un'applicazione può collegare e scollegare oggetti solo su un registro che gira sula stessa macchina, mentre la ricerca può avvenire anche su altri server;

UnicastRemoteObject offre le seguenti caratteristiche:

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

RMI infine non gestisce la sincronizzazione degli accessi se ci sono due client che contemporaneamente invocano un metodo dell'oggetto remoto: nel server saranno presenti due thread in esecuzione sul codice del metodo.

Parte IV
Java EE

9 Introduzione

9.1 Definizione

Java EE - *Java Enterprise Edition* - è la piattaforma di Java volta allo sviluppo di applicazioni di livello *enterprise*.

Le applicazioni enterprise forniscono la business logic per un'azienda. Sono gestite centralmente e spesso interagiscono con altri software aziendali.

La piattaforma Java EE usa un modello di sviluppo semplificato:

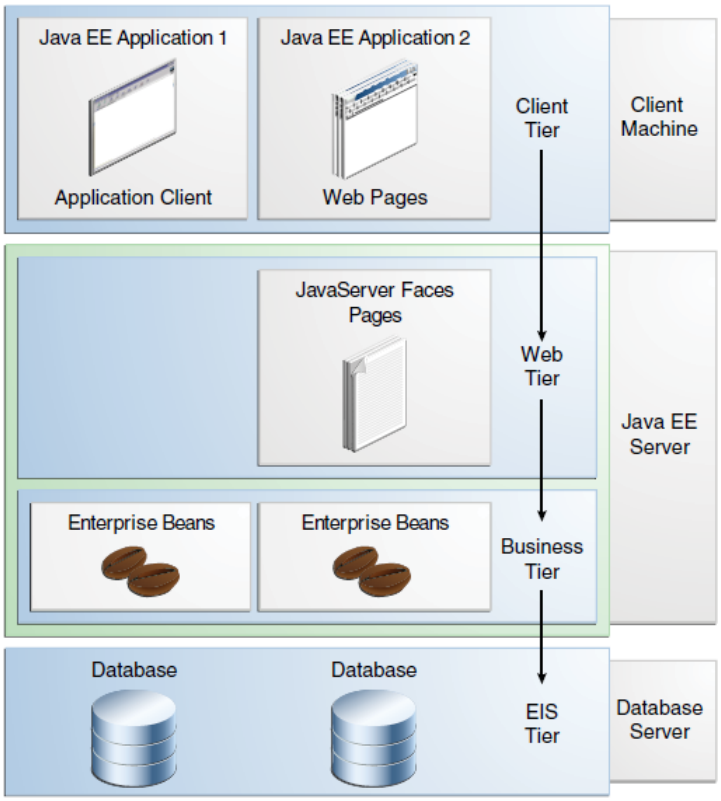
- I descrittori XML per la pubblicazione sono opzionali: è possibile specificare delle annotazioni direttamente nel codice, che il server Java EE interpreterà per configurare i componenti automaticamente.
- È possibile fare uso della *dependency injection* ad ogni risorsa necessaria ad un componente, nascondendo così la fase di creazione e ricerca delle risorse al codice dell'applicazione, che vengono incluse automaticamente seguendo le annotazioni specificate.

9.2 Struttura

La struttura di Java EE è multilivello, e prevede i seguenti quattro livelli (*tier*):

- Client tier
- Web tier
- Business tier
- EIS tier

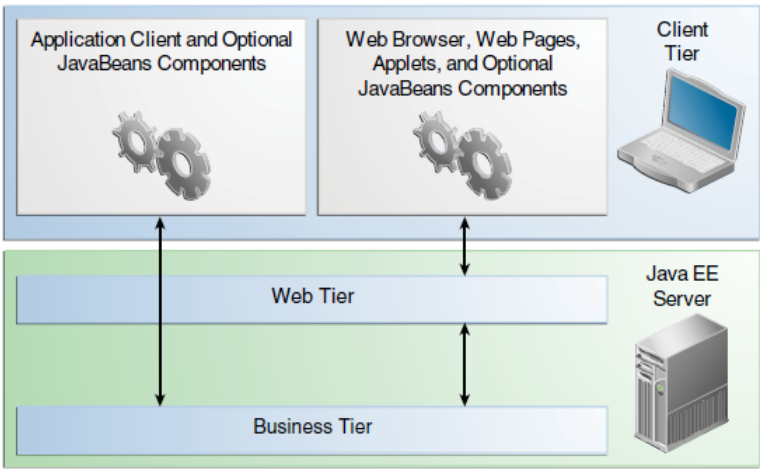
I livello sono così organizzati:



Vediamoli più nel dettaglio:

Client tier

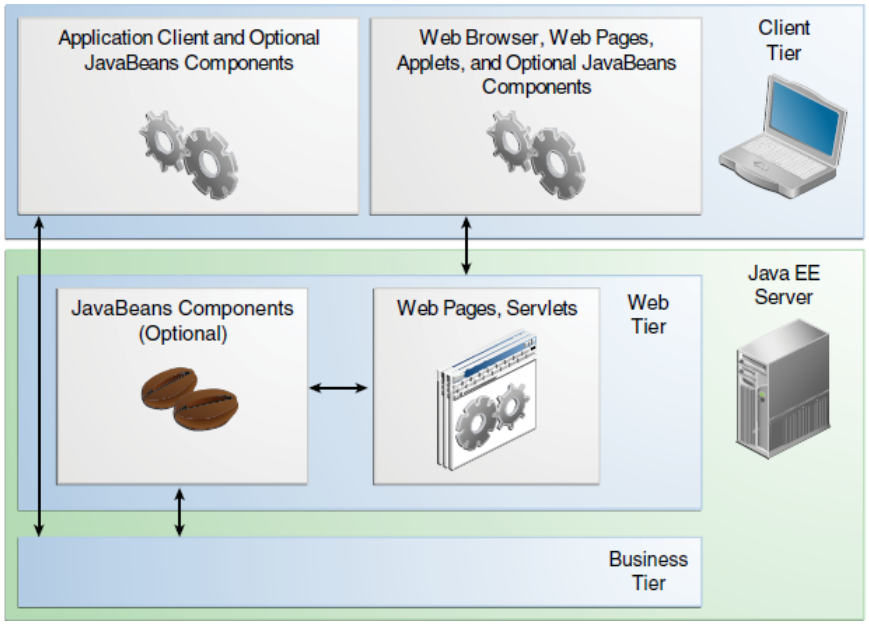
- Web clients: browsers.
- Application clients: applicazioni, che possono essere sia tramite GUI (cfr. AWT, Swing, etc.) che direttamente da riga di comando.
  - Componenti JavaBeans (opzionali).
- Applets: vedi capitolo dedicato.
  - Componenti JavaBeans (opzionali).



Web tier Le componenti Web possono essere:

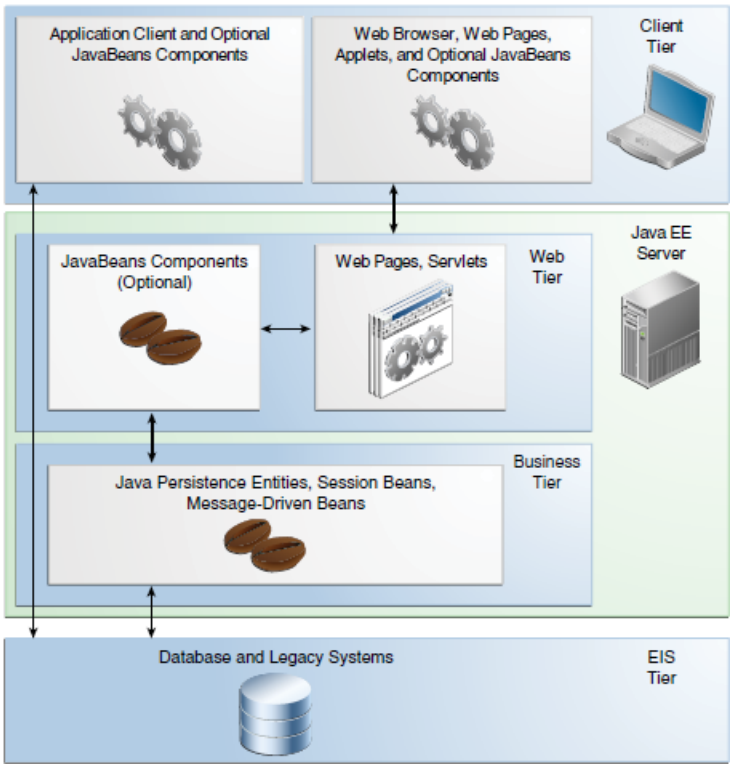
- Servlets
- Pagine Web, che possono essere create tramite:
  - JSP - *Java Server Pages*
  - JSF - *Java Server Faces* (crea un framework con una UI per la creazione di applicazioni Web. Usa Servlets e JSP).

Le pagine HTML statiche e le Applet sono accorpate con le componenti Web durante l'assemblaggio dell'applicazione, ma non sono considerate parte delle componenti Web.



Business tier Il *Business code*, che è la logica che risolve/fornisce le esigenze di un particolare dominio di business (ad es.: banche, commercio, finanza), viene gestito da *enterprise beans*. Alcuni beans possono trovarsi, opzionalmente, anche nel *Web tier*.

È praticamente l'interfaccia tra l'applicazione e le risorse dell'azienda.

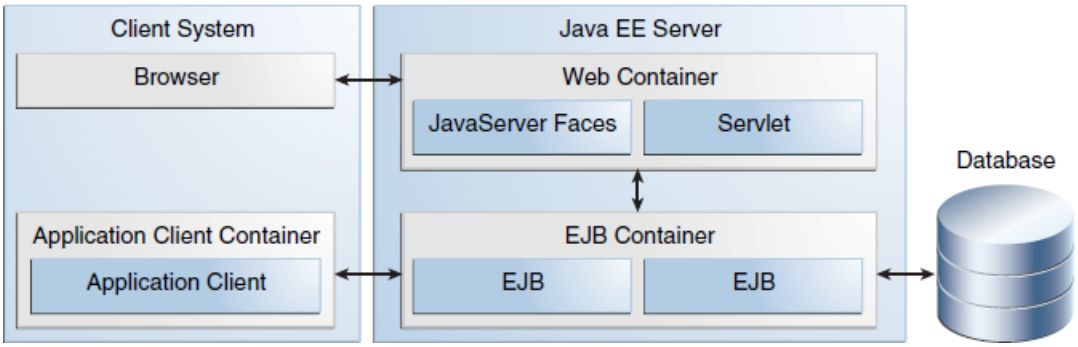


EIS tier Il livello EIS - *Enterprise Information System* - si occupa di gestire l'accesso alle risorse dell'azienda, quali ERP, mainframes, databases, e qualsiasi altro sistema.

9.2.1 Struttura nel dettaglio

Il livello Web si divide in due contenitori:

- Web Container
  - JSF - *Java Server Faces*
  - Servlet
- EJB Container [*Enterprise Java Beans*]



I componenti nel Web container hanno a disposizione una serie di API per Java SE (in JavaEE7, sono 23); tra cui:

- JSP
- WebSocket
- JSON-P
- EJB Lite
- JavaMail
- Connectors
- Bean validation
- Java Persistence

I beans contenuti nell'EJB container hanno a disposizione un diverso elenco di API per Java SE (in JavaEE7, sono 18). Vediamone alcune di queste API:

- CDI
- JavaMail
- Connectors
- JACC
- Bean validation

Il livello Client si divide in due rami:

- Browser
- **Contenitore** Application Client

Le API disponibili per il contenitore Application client sono poche (in JavaEE7, sono 11), e tra queste:

- Java Persistence
- Web Services
- JSON-P
- JMS
- Bean validation
- JavaMail
- CDI

9.2.2 Glassfish

Glassfish è un'implementazione di Java EE. È conforme alla definizione di Java EE.

In aggiunta alle API disponibili per Java EE, include una serie di altre API per dare maggiori API agli sviluppatori.

10 Servlet

10.1 Introduzione

10.1.1 Definizione

La tecnologia Java Servlet permette di definire classi servlet HTTP-specific. Una classe servlet estende le capacità di server che ospitano applicazioni accedute tramite un modello di *request/response*.

Nonostante le servlet possano risponde a qualsiasi tipo di richiesta, sono solitamente utilizzate per estendere le applicazioni ospitate da un Web server.

Al giorno d'oggi le servlet rappresentano una scelta popolare per la creazione di applicazioni Web interattive.

Le servlet necessitano di essere gestite tramite un c.d. *servlet container*; ne esistono di svariati, disponibili per tutte le principali piattaforme (ad es., per: Apache, IIS).

Esempi di *servlet container* - altresì chiamati *servlet engine* o, ancora, *web container* - sono:

- Apache Tomcat
- Oracle Glassfish
- JBOSS application server
- WebSphere

10.1.2 Punti di forza

Le servlet forniscono un metodo basato sui componenti, ed indipendente dalla piattaforma, per creare applicazioni Web-based, superando le limitazioni dei programmi CGI.

A differenza di altri meccanismi di estensione proprietari (ad es.: i moduli Apache), le servlet sono indipendenti sia dal server che dalla piattaforma.

Le servlet hanno accesso all'intera famiglia delle API Java, inclusa l'API JDBC, così come delle API specifiche di HTTP.

Le servlet godono di tutti i benefici del linguaggio Java, tra cui:

- Portabilità
- Maturità
- Performance
- Riusabilità
- Protezione dai crash

10.1.3 Qualche informazione in più

Context path Ogni servlet dispone di un *context path*, definito alla creazione dell'applicazione, che rappresenta il nome dell'applicazione nel *servlet engine*.

Ad esempio, se pubblico sul server *localhost* l'applicazione avente come *context path*: "MiaApp", questa avrà come root:

http://localhost:8080/MiaApp/

10.2 Definizione delle interfacce e delle classi

10.2.1 *javax.servlet.*Servlet

public interface Servlet {

```
        void destroy();
        ServletConfig getServletConfig();
        String getServletInfo();
        void init(ServletConfig config);
        service(ServletRequest request, ServletResponse response);
}
```

Implementata dalle classi:

- FacesServlet
- GenericServlet *implements* Serializable, ServletConfig (,Servlet)
  - HttpServlet *extends* GenericServlet

Implementata dalle interfacce:

- JspPage
  - Aggiunge: jspDestroy(), jspInit().
  - HttpJspPage *implements* JspPage

10.2.2 *javax.servlet.*ServletConfig

```
public interface ServletConfig {
    String getInitParameter(String name);
    Enumeration<String> getInitParameterNames();
    ServletContext getServletContext();
    String getServletName();
}
```

Implementata dalle classi:

- GenericServlet
  - HttpServlet *extends* GenericServlet

10.2.3 *javax.servlet.*ServletRequest

```
public interface ServletRequest {
    /*
    * Vi sono una cinquantina di metodi definiti.
    * Per brevità ne elencheremo solamente alcuni.
    */
    ServletInputStream getInputStream();
    Object getAttribute(String name);
    Enumeration<String> getAttributeNames();
    String getParameter(String name);
    Enumeration<String> getParameterNames();
    String[] getParameterValues(String name);
    String getCharacterEncoding();
    int getContentLength();
    String getContentType();
    String getProtocol();
    String getLocalAddr();
```

```
        Locale getLocale();
        String getScheme() // request made via: for example, http, https, or ftp.
        boolean isAsyncSupported();
        AsyncContext getAsyncContext();
        AsyncContext startAsync(); // Puts this request into asynchronous mode
        [...]
}
```

Implementata dalle classi:

- ServletRequestWrapper
  - HttpServletRequestWrapper *extends* ServletRequestWrapper. Aggiunge qualche metodo, come:
    - boolean authenticate(HttpServletResponse response);
    - String getAuthType();
    - Cookie[] getCookies();
    - String getHeader(String name);
    - String getRequestURI();
    - HttpSession getSession();
    - etc.

10.2.4 *javax.servlet.*ServletResponse

```
public interface ServletResponse {
    void flushBuffer();
    int getBufferSize();
    String getCharacterEncoding();
    String getContentType();
    Locale getLocale();
    ServletOutputStream getOutputStream(); // Returns a ServletOutputStream suitable for
  writing binary data in the response.
    PrintWriter getWriter(); // Returns a PrintWriter object that can send character text to
  the client.
    boolean isCommitted();
    void reset();
    void resetBuffer();
    void setBufferSize(int size);
    void setCharacterEncoding(String charset);
    void setContentLength(int len);
    void setContentLengthLong(long len);
    void setContentType(String type);
    void setLocale(Locale loc);
    // Questi sono tutti i metodi definiti.
}
```

Implementato dalle classi:

- ServletResponseWrapper
  - HttpServletResponseWrapper *extends* ServletResponseWrapper. Aggiunge qualche metodo, come:
    - void addCookie(Cookie cookie);
    - boolean containsHeader(String name);

* String getHeader(String name);
* String encodeURL(String url);
* void setHeader(String name, String value);
* etc.

10.2.5 *javax.servlet.*ServletContext

Il riferimento al contesto è contenuto all'interno di *ServletConfig*.

È presente un contesto per ogni Web Application (se l'applicazione è *distribuita*, vi sarà un contesto per ogni JVM su cui viene fatta girare l'applicazione).

```
public interface ServletContext {
    /*
    * Vi sono decine di metodi definiti in questa interfaccia.
    * Per brevità, ne elencheremo solamente alcuni.
    */
    void addListener(T t); // <T extends EventListener>
    ServletRegistration.Dynamic addServlet(String servletName, String className);
    Object getAttribute(String name);
    Enumeration<String> getAttributeNames();
    String getContextPath();
    ServletContext getContext(String uripath);
    void setAttribute(String name, Object object)
    Enumeration<String> getInitParameterNames();
    boolean setInitParameter(String name, String value)
    RequestDispatcher getRequestDispatcher(String path);
    String getServerInfo();
    String getServletContextName(); // nome dell'applicazione web
    SessionCookieConfig getSessionCookieConfig();
    void log(String msg); // Scrive sul log della servlet
    void removeAttribute(String name);
    [...]
}
```

10.2.6 Frammento di *javax.servlet.*GenericServlet

Il metodo *init* si occupa di inizializzare la configurazione ed, opzionalmente, le risorse della servlet. Viene richiamato un'unica volta per ogni servlet, quando essa viene istanziata (cioè alla prima richiesta di quella servlet da parte di un client).

Come notiamo dal codice sottostante, troviamo definiti:

- Il metodo *public void init(ServletConfig config)*:
  - effettua il salvataggio della ServletConfig nella servlet.
- Un suo overload, *public void init()*:
  - viene eseguito dal primo metodo. Dobbiamo effettuare l'override di questo metodo nella nostra implementazione della Servlet se necessitiamo di effettuare delle operazioni di inizializzazione sulla stessa.

Vediamone l'implementazione:

Listing 12: HttpServlet.java

```
public abstract class GenericServlet
         implements Servlet, ServletConfig, java.io.Serializable
       {

       public GenericServlet() {
       }

       public void destroy() {
       }

       public String getInitParameter(String name) {
              return getServletConfig().getInitParameter(name);
       }

       public ServletConfig getServletConfig() {
              return config;
       }

       public ServletContext getServletContext() {
              return getServletConfig().getServletContext();
       }

        /**
         * Called by the servlet container to indicate to a servlet that the
         * servlet is being placed into service. See {@link Servlet#init}.
         *
         * <p>This implementation stores the {@link ServletConfig}
         * object it receives from the servlet container for later use.
         * When overriding this form of the method, call
         * <code>super.init(config)</code>.
         *
         * @param config    the <code>ServletConfig</code> object
         *     that contains configutation
         *     information for this servlet
         *
         * @exception ServletException  if an exception occurs that
         *     interrupts the servlet's normal
         *     operation
         *
         * @see     UnavailableException
         *
         */
        public void init(ServletConfig config) throws ServletException {
              this.config = config;
              this.init();
        }


        /**
         *
         * A convenience method which can be overridden so that there's no need
         * to call <code>super.init(config)</code>.
         *
         * <p>Instead of overriding {@link #init(ServletConfig)}, simply override
         * this method and it will be called by
         * <code>GenericServlet.init(ServletConfig config)</code>.
         * The <code>ServletConfig</code> object can still be retrieved via {@link
         * #getServletConfig}.
```

```
59          *
60          * @exception ServletException  if an exception occurs that
61          *     interrupts the servlet's
62          *     normal operation
63          */
64         public void init() throws ServletException {
65
66         }
67 }
```

10.2.7 Frammento di *javax.servlet*.http.HttpServlet

La classe HttpServlet è un'estensione di GenericServlet creata per dialogare con il protocollo HTTP. Contiene tra gli altri il metodo *service*, che si occupa di:

1. Determinare il metodo HTTP della *HTTP request* (es.: POST, GET);
2. Richiamare il metodo opportuno per gestirla (es.: doGet(), doPost()).

Vediamone l'implementazione:

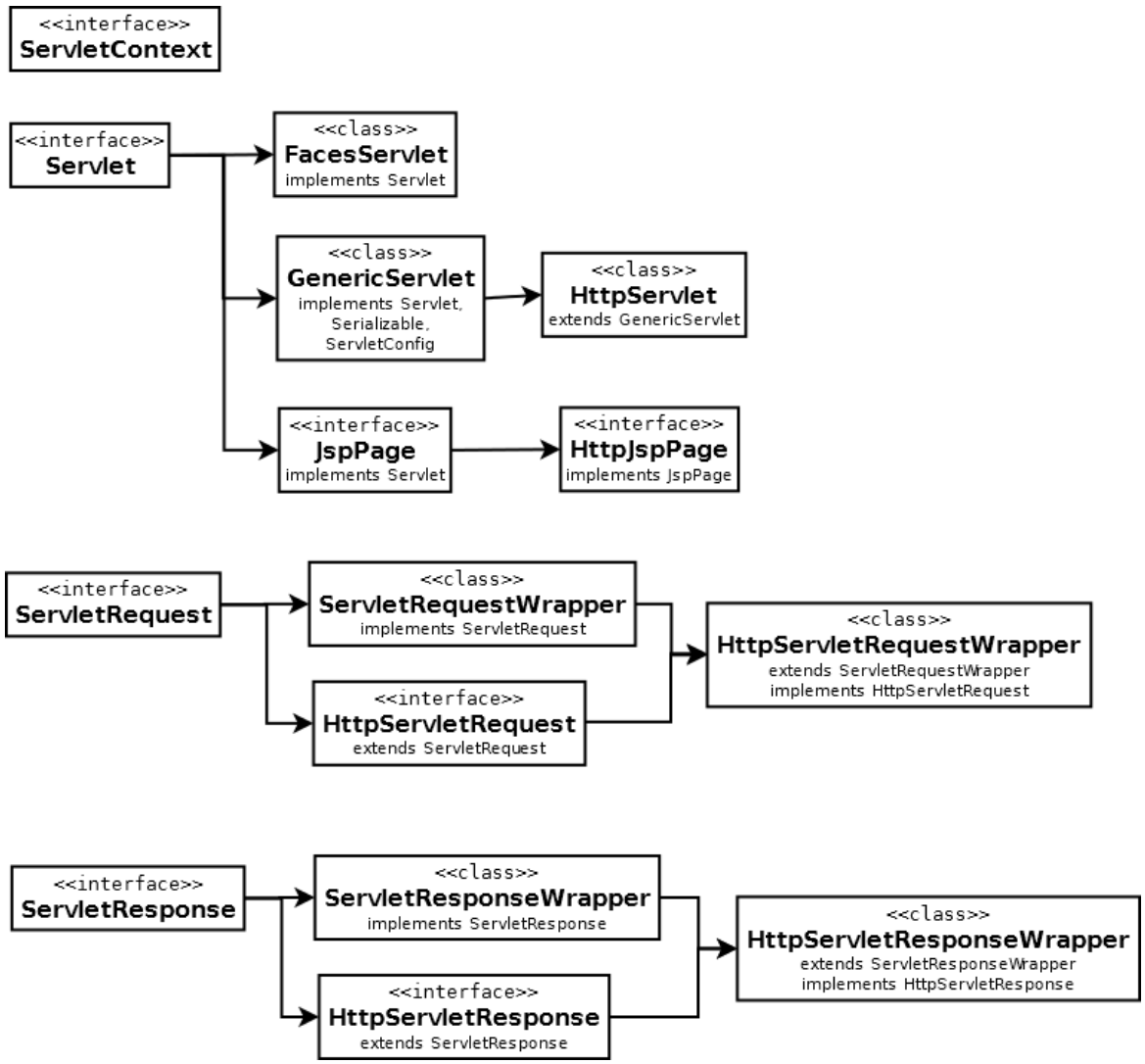
Listing 13: HttpServlet.java

```
1  public abstract class HttpServlet extends GenericServlet
2          implements java.io.Serializable {
3
4        private static final String METHOD_DELETE = "DELETE";
5        private static final String METHOD_HEAD = "HEAD";
6        private static final String METHOD_GET = "GET";
7        private static final String METHOD_OPTIONS = "OPTIONS";
8        private static final String METHOD_POST = "POST";
9        private static final String METHOD_PUT = "PUT";
10       private static final String METHOD_TRACE = "TRACE";
11
12       private static final String HEADER_IFMODSINCE = "If-Modified-Since";
13       private static final String HEADER_LASTMOD = "Last-Modified";
14
15       private static final String LSTRING_FILE = "javax.servlet.http.LocalStrings";
16       private static ResourceBundle lStrings = ResourceBundle.getBundle(LSTRING_FILE);
17
18       // [... ALTRI METODI ...]
19
20       protected void service(HttpServletRequest req, HttpServletResponse resp)
21                       throws ServletException, IOException {
22
23              String method = req.getMethod();
24
25              if (method.equals(METHOD_GET)) {
26                 long lastModified = getLastModified(req);
27                 if (lastModified == -1) {
28                        // servlet doesn't support if-modified-since, no reason
29                        // to go through further expensive logic
30                        doGet(req, resp);
31                 } else {
32                        long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
33                        if (ifModifiedSince < (lastModified / 1000 * 1000)) {
34                               // If the servlet mod time is later, call doGet()
35                               // Round down to the nearest second for a proper compare
36                               // A ifModifiedSince of -1 will always be less
```

```
                                maybeSetLastModified(resp, lastModified);
                                doGet(req, resp);
                            } else {
                                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
                            }
                        }

                    } else if (method.equals(METHOD_HEAD)) {
                        long lastModified = getLastModified(req);
                        maybeSetLastModified(resp, lastModified);
                        doHead(req, resp);

                    } else if (method.equals(METHOD_POST)) {
                        doPost(req, resp);

                    } else if (method.equals(METHOD_PUT)) {
                        doPut(req, resp);

                    } else if (method.equals(METHOD_DELETE)) {
                        doDelete(req, resp);

                    } else if (method.equals(METHOD_OPTIONS)) {
                        doOptions(req,resp);

                    } else if (method.equals(METHOD_TRACE)) {
                        doTrace(req,resp);

                    } else {
                        //
                        // Note that this means NO servlet supports whatever
                        // method was requested, anywhere on this server.
                        //

                        String errMsg = lStrings.getString("http.method_not_implemented");
                        Object[] errArgs = new Object[1];
                        errArgs[0] = method;
                        errMsg = MessageFormat.format(errMsg, errArgs);

                        resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
                    }
                } // Fine metodo service()
            } // Fine classe HttpServlet
```

10.2.8 Gerarchia

Riassumendo, questa è la gerarchia delle classi:



10.3 Deploy di un'applicazione

Per rilasciare una Web Application Java EE è necessario organizzarla in un formato standard, comprensibile dai Servlet Engines.

Questo si può fare sia esportando direttamente la struttura dei file non compressi, che raggruppando tutti i file in un unico archivio **WAR** (*.war - **W**eb **AR**chive*).

In ognuno dei due casi, la struttura dovrebbe seguire il seguente standard:

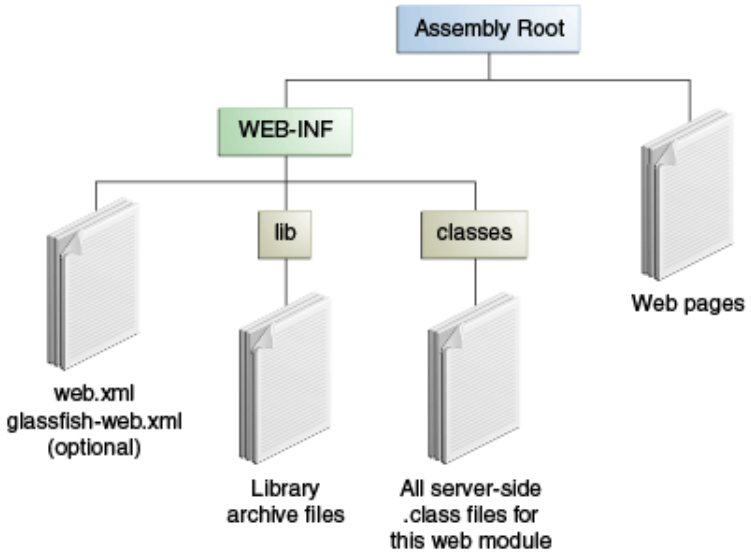
<**docBase**>/ è la root del file, e contiene:

- pagine (X)HTML.
- risorse statiche: immagini, etc.
- Classi ed archivi client-side.

<**docBase**>/**WEB-INF**/ Contiene:

- **classes**/ una directory che contiene le *classi server-side*:
- **lib**/ una directory che contiene i file JAR di:
  - Enterprise Beans (EJB)
  - Archivi JAR delle librerie richiamate dalle *classi server-side*.
- **Descrittori** per il deploy (*deployment descriptors*), come:
  - **web.xml** (descrittore per il deploy della Web Application).
  - **ejb-jar.xml** (descrittore per il deploy degli EJB).

Il seguente schema esemplifica tale struttura:



10.4 *Deployment descriptor* web.xml

10.4.1 Introduzione

Un *deployment descriptor* di un'applicazione Web descrive:

- Le classi.
- Le risorse.
- La configurazione.
- Come il Web server usa tutto ciò per rispondere alle richieste.

Per ogni applicazione Java EE va quindi creato un file di configurazione XML:

```
<docBase>/WEB-INF/web.xml
```

che ha come *tag* principale <web-app>:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    [altri tag]
</web-app>
```

Vediamo alcuni dei *tag* che possiamo specificare al suo interno:

10.4.2 <context-param>

Utilizzato per definire i parametri del contesto dell'applicazione, ovvero la configurazione globale che è accessibile da parte di tutte le servlet.

Tag figli:

- <description> (opzionale)
- <param-name>
- <param-value>

Esempio:

```
<context-param>
    <description>Enable debugging for the application</description>
    <param-name>debug</param-name>
    <param-value>true</param-value>
</context-param>
<context-param>
    <param-name>webmaster</param-name>
    <param-value>address@somedomain.com</param-value>
</context-param>
```

10.4.3 <servlet>

Utilizzato per dichiarare le servlet dell'applicazione, specificando a quale classe fare riferimento per ognuna di esse. Ad ognuna si assegna un nome univoco, che serve per riferirsi a tale servlet all'interno del file web.xml.

Tag figli:

- <servlet-name>
- <servlet-class> *oppure* <jsp-file>
- <description> *(opzionale)*
- <init-param> *(opzionale e molteplice)*
  - <param-name>
- <load-on-startup> *(opzionale, indica di caricare la servlet all'avvio dell'applicazione)*

Esempio:

```
<servlet>
    <servlet-name>Statistics</servlet-name>
    <servlet-class>mysite.server.GeneralStatistics</servlet-classe>
</servlet>
<servlet>
    <servlet-name>pathjsp</servlet-name>
    <jsp-file>pathfinder.jsp</jsp-file>
</servlet>
<servlet>
```

```
        <servlet-name>redteam</servlet-name>
        <servlet-class>mysite.server.TeamServlet</servlet-class>
        <init-param>
            <param-name>teamColor</param-name>
            <param-value>red</param-value>
    </init-param>
        <init-param>
            <param-name>bgColor</param-name>
            <param-value>#CC0000</param-value>
    </init-param>
</servlet>
    <servlet>
        <servlet-name>blueteam</servlet-name>
        <servlet-class>mysite.server.TeamServlet</servlet-class>
        <init-param>
            <param-name>teamColor</param-name>
            <param-value>blue</param-value>
    </init-param>
        <init-param>
            <param-name>bgColor</param-name>
            <param-value>#0000CC</param-value>
    </init-param>
</servlet>
```

10.4.4 <servlet-mapping>

Questo elemento specifica un URL ed il nome della servlet da usare per le richieste di quell'URL. Supporta la wildcard "*" all'inizio o alla fine dell'URL.

Tag figli:

- <servlet-name>
- <url-pattern>

Esempio:

```
    <servlet-mapping>
        <servlet-name>Statistics</servlet-name>
        <url-pattern>/common/Statistics</url-pattern>
</servlet-mapping>
    <servlet-mapping>
        <servlet-name>redteam</servlet-name>
        <url-pattern>/red/*</url-pattern>
</servlet-mapping>
    <servlet-mapping>
        <servlet-name>blueteam</servlet-name>
        <url-pattern>/blue/*</url-pattern>
</servlet-mapping>
```

10.4.5 <session-config>

Tag figli:

- <session-timeout>

Esempio:

```
<session-config>
    <session-timeout> 30 </session-timeout>
</session-config>
```

10.4.6 Altri tag

Abbiamo visto i principali tag, ma ne sono disponibili altri. Vediamoli tutti (v.2.4):

- <display-name>
- <description>
- <distributable /> *// Indica che l'applicazione è pronta per essere distribuita su più server*
- <jsp-config> *(una sola dichiarazione possibile)*
  - <jsp-property-group>
    - <url-pattern>
    - <include-prelude>
    - <include-coda>
  - <taglib> *(deprecato)*
    - <taglib-uri>
    - <taglib-location>
- <env-entry> *// variabili d'ambiente JNDI*
  - <env-entry-name>
  - <env-entry-value>
- <filter> *// Definisce dei filtri sulle richieste*
  - <filter-name>
  - <filter-class>
  - <init-param> *(opzionale e molteplice)*
    - <param-name>
    - <param-value>
- <filter-mapping> *// Associa dei filtri a delle servlet*
  - <filter-name>
  - <url-pattern>
- <listener> *// Definisce degli event listeners per l'applicazione*
  - <listener-class>
- <security-role> *// Definisce i ruoli di sicurezza*

  - <role-name>

- <security-constraint> *// Definisce i vincoli/ruoli di sicurezza da applicare agli oggetti*
  - <display-name>
  - <web-resource-collection>
    - <web-resource-name>
    - <url-pattern>
    - <http-method> *(opzionale e molteplice)*
  - <auth-constraint> *(opzionale)*
    - <role-name> *(molteplice)*

- <login-config> *// Form-based authentication*
  - <auth-method>
  - <form-login-config>
    - <form-login-page>
    - <form-error-page>

- <login-config> *// BASIC authentication*
  - <auth-method>
  - <realm-name>

- <security-constraint> *// Forza l'uso di SSL*
  - <web-resource-collection>
    - <web-resource-name>
    - <url-pattern>
  - <user-data-constraint>
    - <transport-guarantee>

- <error-page> *// Definisce le pagine di fallback in caso di errore*
  - <error-code>
  - <location>

- <mime-mapping> *// Mappa l'estensione di un file ad un mime-type*
  - <extension>
  - <mime-type>

- <locale-encoding-mapping-list> *// Definisce un elenco dei Locale disponibili*
  - <locale-encoding-mapping>
    - <locale>
    - <encoding>

- <welcome-file-list> *// Definisce i nomi dei file che vanno presi di default nei path puri (es.: www.a.it/test/)*
  - <welcome-file> *(molteplice e in ordine di priorità)*

11 Cookies

11.1 Definizione

I cookies consentono di rendere persistenti sul client delle informazioni legate alla sessione e che verrebbero altrimenti perse alla chiusura di ogni connessione HTTP.

Ad ogni richiesta fatta verso lo stesso server, il client ripropone nello header della *request* i cookie che precedentemente il server aveva creato.

I cookies sono file che contengono informazioni salvate in coppie chiave-valore. Questi file sono salvati sul filesystem del client. Ci sono delle limitazioni sui cookie che un server può inviare, e riguardano:

- le dimensioni.
- il numero.

11.1.1 Punti di forza

Alcuni dei punti di forza dei cookies sono i seguenti:

- Personalizzazione di un sito in base alle pagine precedentemente visitate.
- Pubblicità mirata alle pagine precedentemente visitate.
- Memorizzazione delle login, evitando di richiedere nuovamente username e password ad ogni accesso.
- Possibilità di gestire informazioni senza bisogno di memorizzarle sul server stesso, ma in modo distribuito sui client.

11.1.2 Struttura di un cookie

Il nome deve contenere solo caratteri ASCII alfanumerici esclusi virgole, punti e virgola e spazi, inoltre non possono iniziare col simbolo '$' (cfr. RFC 2109).

La struttura di un cookie è la seguente:

```
username=John Doe
lastItem=1249
```

11.2 Cookies in Java

I cookie vengono usati dal motore delle servlet per gestire le sessioni, risolvendo così i problemi legati a tale ambito.

Nelle API Java, il nome del cookie non può essere cambiato dopo la sua creazione. Non vi sono vincoli sul contenuto.

I principali metodi della classe *javax.servlet.http.Cookie* sono:

comment

- void setComment()
- String getComment()

Il commento viene visualizzato dal browser nel caso chieda conferma se accettare o meno il cookie.

version

- void SetVersion()
- int getVersion()

Il valore della versione può essere:

- **0**: Netscape standard
- **1**: RFC 2109

maxAge

- void setMaxAge()
- int getMaxAge()

Il tempo t di vita del cookie, espresso in secondi.
Quando il cookie arriva al browser, esso intraprende una delle seguenti azioni, in base a tale valore.

- **t>0**: lo terrà in vita per t secondi.
- **t==0:** eliminerà il cookie.
- **t<0:** lo terrà in vita finchè dura la sessione del browser.

domain

- void setDomain()
- String getDomain()

Il dominio del cookie. Quando il browser effettua una richiesta a quel dominio, presenta solo i cookie corrispondenti.

path

- void setPath()
- int getPath()

Il path nel dominio per il quale il browser deve presentare il cookie.
Questo in quanto, per un dominio, è possibile che siano stati impostati differenti cookies per differenti sezioni.

11.3 Esempi

11.3.1 Lettura cookie

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowCookies extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws Ser-
  vletException, 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>");
    }
}
```

11.3.2 Scrittura cookie

Esempio base

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

Esempio di impostazione di due cookies

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class TestSet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws Ser-
  vletException, IOException {
        // cookie che dura per la sessione:
        Cookie cookie = new Cookie("Session-Cookie","Cookie-Value-S");
        response.addCookie(cookie);
        // cookie che dura 24 ore:
        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>");
  }
}
```

11.3.3 Gestione sessione

```
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);
```

12 JSF

JSF - Java Server Faces - è un framework basato sul pattern MVC, utilizzato per la costruzione di interfacce utente.
A partire da Java EE 6, Oracle consiglia di usare le *Facelets* anzichè JSP. Dal tutorial Java EE 6 sulle JSF:

> *JavaServer Pages (JSP) technology, previously used as the presentation technology for JavaServer Faces, does not support all the new features available in JavaServer Faces 2.0. JSP technology is considered to be a deprecated presentation technology for JavaServer Faces 2.0. Facelets is a part of the JavaServer Faces specification and also the preferred presentation technology for building JavaServer Faces technology-based applications.*

13 JSP

13.1 Introduzione

13.1.1 Definizione

Le JSP - *Java Server Pages* - sono delle servlet, definite da un mix di contenuto statico (HTML) e di codice Java.

La tecnologia JSP dispone di tutte le capacità dinamiche delle *Servlets* ma fornisce un approccio più naturale alla creazione di contenuto statico.

I punti salienti della tecnlogia JSP sono:

- Un linguaggio per sviluppare dei documenti di testo che descrivono come processare le richieste.
- Un linguaggio di espressioni per accedere agli oggetti lato server.
- Un meccanismo per definire delle estensioni al linguaggio JSP.

È possibile installare delle *tag library*, da usare poi nelle JSP, attraverso il tag *<taglib>* in *web.xml*.

Struttura delle JSP Un file JSP è un documento di testo che contiene due tipi di dati:

- Dati statici, che possono essere di vari formati: HTML, SVG, XML, etc.
- Elementi JSP, che costruiscono il contenuto dinamico.

I file JSP possono essere di due diversi tipi, in base alla sintassi di cui fanno uso:

- JSP Page - fa uso della sintassi standard.
- JSP Document - fa uso della sintassi XML.

13.1.2 JSP Pages

Una pagina JSP è un file JSP che fa uso della sintassi JSP standard.

13.1.3 JSP Documents

Un documento JSP è un file JSP che fa uso della sintassi XML. Viene utilizzato più raramente rispetto alle *pagine JSP*.

La sintassi XML prevede:

- Un unico elemento root.
- Ogni tag aperto deve essere anche chiuso.

In realtà gli elementi JSP ma anche il contenuto statico, quindi tutta la pagina JSP, è per sua impostazione già abbastanza aderente allo standard XML.

Vediamo un esempio di conversione di una Pagina JSP per farla diventare un Documento JSP:

Pagina JSP Vediamo la pagina JSP che andremo a convertire:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
    <img src="duke.waving.gif">
    <h2>My name is Duke. What is yours?</h2>
    <form method="get">
```

```
            <input type="text" name="username" size="25">
            <p></p>
            <input type="submit" value="Submit">
            <input type="reset" value="Reset">
        </form>
            <jsp:useBean id="userNameBean" class="hello.UserNameBean" scope="request"/>
            <jsp:setProperty name="userNameBean" property="name" value="${param.username}"
        />
            <c:if test="${fn:length(userNameBean.name) > 0}" >
                <%@include file="response.jsp" %>
        </c:if>
    </body>
        </html>
```

Documento JSP Vediamo ora come risulterebbe la pagina JSP vista sopra, una volta convertita in documento JSP:

```
    <html xmlns:c="http://java.sun.com/jsp/jstl/core" xmlns:fn="http://java.sun.com/jsp/jstl/functions"
>
        <head><title>Hello</title></head>
        <body bgcolor="white" />
            <img src="duke.waving.gif" />
            <h2>My name is Duke. What is yours?</h2>
            <form method="get">
                <input type="text" name="username" size="25" />
                <p></p>
                <input type="submit" value="Submit" />
                <input type="reset" value="Reset" />
         </form>
            <jsp:useBean id="userNameBean" class="hello.UserNameBean" scope="request"/>
            <jsp:setProperty name="userNameBean" property="name" value="${param.username}"
         />
            <c:if test="${fn:length(userNameBean.name) gt 0}" >
                <jsp:directive.include="response.jsp" />
         </c:if>
    </body>
</html>
```

In questo caso, sono stati cambiati i seguenti costrutti:

- Rimosse le direttive *taglib,* rimpiazzate dai *namespace* di XML.
- Sono stati aggiunti gli end-tag che per quei tag che non disponevano (es.: <img> diventa <img />).
- Il simbolo ">" dell'espressione EL (*Expression Language*) è stato sostituito con "**gt**" (*greater than*).
- La direttiva *<%@include* è stata cambiata in *<jsp:directive.include*, che è una sintassi compatibile con XML.

13.1.4 Mapping delle JSP

Il compilatore Java esegue automaticamente delle operazioni sulle JSP:

- Le compila in classi servlet.
- Mappa queste classi al path corrispondente a quello in cui si trovano (es.: *Web pages/register/start.jsp* viene mappata in *<baseURI>/register/start.jsp*).

Se si desidera definire manualmente il mapping di una JSP, è possibile definirlo esplicitamente in *web.xml* tramite il tag <servlet> e usando poi <jsp-file> per indicare la classe.

13.2 Elementi JSP

Gli elementi JSP si dividono in più categorie. Ognuna di queste ha un proprio tag di apertura ed accetta contenuto specifico.

13.2.1 Direttive

Sintassi: <%@ *codice* %>

Le direttive sono usate per passare informazioni dalla JSP al contenitore della servlet.
Sono indipendenti dalla posizione, e sono uniche (un attributo non può essere ridefinito, ma solo aggiunto).
I principali tipi di direttiva sono:

- @page direttive di pagina (*vedi sotto).
- @include per includere staticamente risorse al momento della compilazione.
- @taglib per l'inclusione di librerie di tag (cfr. sottosezione su *Extension tag* in questo capitolo).

Direttive di pagina (@page) Vediamo i vari *oggetti impliciti* disponibili per la direttiva @page:

- **attribute** (descrizione)
- **language** (Linguaggio di scripting. In JSP 1.2, unico valore ammesso: "Java")
- **extends** (per estendere una classe Java. Usare con cautela)
- **import** (import di classi o interi package)
- **session**
- **buffer** (se presente, e dimensione)
- **autoFlush** (default: true)
- **isThreadSafe**
- **info** (stringa ritornata da *Servlet.getServletInfo()*)
- **isErrorPage**
- **errorPage**
- **contentType** (definisce la codifica del carattere, il tipo di risposta ed il *mime-type* della risorsa creata dalla pagina JSP. Default: *text/html*).
- **pageEncoding** Definisce la codifica del set di caratteri della pagina JSP. Se non è definito un charset nell'oggetto *contentType*, viene usato il default: ISO-8859-1.

Alcuni esempi pratici:

```
<%@page language="java" %>
<%@page session="java" %>
<%@page import="java.awt.*,java.util.*" %>
<%@page isThreadSafe="false" %>
<%@page errorPage="myErrPage.jsp" %>
<%@page isErrorPage="true" %>
```

13.2.2 Dichiarazioni

Sintassi: <%! *codice* %>

Le dichiarazioni sono blocchi di codice usati per definire variabili e metodi della classe servlet che verrà generata.

Esempio:

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

13.2.3 Scriptlets

Sintassi: <% *codice* %>

All'interno delle scriptlets è possibile scrivere blocchi di codice Java.

13.2.4 Azioni standard

Sintassi: <jsp:azione *attributo="valore"* />

Le azioni standard sono specificate usando la sintassi dei tag XML. Queste azioni influenzano il comportamento a run-time della JSP e della *response*.

Esempi:

- <jsp:useBean .../>
- <jsp:setProperty .../>
- <jsp:getProperty .../>

13.2.5 Espressioni

Sintassi: <%= *codice* %>

Un'espressione è un elemento che stampa in output il valore della variabile/oggetto/metodo indicato al suo interno.

Esempio:

```
<%= request.getParameter("eta") %>
```

13.3 Oggetti JSP

In una JSP sono definiti diversi oggetti, che sono disponibili in tutta la pagina. Questi oggetti sono disponibili all'interno delle *scriptlets* (<% ... %>) e delle *espressioni* (<%= ... %>).

Vediamo questi oggetti nel dettaglio:

Variabile	Tipo	Scope	Descrizione
out	Writer	page	Un oggetto wrapper che scrive nello stream di output.
request	HttpServletRequest	request	La richiesta che comportato la chiamata della pagina.
response	HttpServletResponse	page	La risposta alla request.
session	HttpSession	session	La sessione creata per il client che ha richiesto la pagina.
page	Object	page	Equivalente a *this*.
application	ApplicationContext	application	getServletConfig().getServletContext() - area condivisa tra le servlet.
config	ServletConfig	page	Equivalente al parametro *config* del metodo *init()* di una servlet.
pageContext	PageContext	page	Sorgente degli oggetti (raramente usato).
exception	Throwable	page	L'eccezione della pagina che ha lanciato l'errore. Solo in errorPage.

13.4 Ciclo di vita di una Pagina JSP

Una pagina JSP gestisce le richieste come una servlet; il suo comportamento dipende pertanto principalmente dalla tecnologia delle servlet.

Una pagina JSP viene a tutti gli effetti tradotta in una *servlet*.

Quando viene richiesta una pagina JSP, il *web container* controlla innanzitutto se la servlet creata per la pagina JSP è più vecchia della JSP: se così fosse, traduce la pagina JSP in una servlet e ricompila la classe.

13.4.1 Traduzione e compilazione

Vediamo come avviene la traduzione di una pagina JSP in una servlet:

Elementi statici Vengono trasformati in codice Java che stampa semplicemente il testo nella *response*.

Elementi JSP Vengono tradotti come segue (N.B.: le parti di codice vengono scritte nel metodo *doGet* della servlet):

Direttive: vengono usate per controllare come il *web container* traduce ed esegue la pagina JSP.

Elementi di scripting: vengono semplicemente inseriti nel corpo della classe servlet (sono già scritti in codice Java).

Espressioni EL (*Expression Language*): sono passate come parametri a chiamate dell'interprete delle *espressioni JSP*.

Elementi jsp:[set|get] Property: sono convertiti in chiamate a metodi di componenti *JavaBeans*.

Elementi jsp:[include|forward]: sono convertiti in invocazioni alla API delle servlet.

Elemento jsp:plugin viene tradotto in un linguaggio di markup, specifico per il browser interessato, per l'attivazione di una applet.

Custom tags: vengono convertiti in chiamate al *tag handler* che implementa il *custom tag* specificato.

Compilazione La servlet generata dalla traduzione della pagina JSP viene posizionata nel path:

domain-dir/generated/jsp/j2ee-modules/WAR-NAME/pageName_jsp.java

Un esempio reale:

domain-dir/generated/jsp/j2ee-modules/date/index_jsp.javac

Sia il sorgente tradotto, che la classe compilata, possono contenere errori, che vengono notati solamente quando la pagina viene richiesta per la prima volta. In questo caso, viene sollevata una *JasperException*.

13.4.2 Ciclo di vita della servlet risultante

Il ciclo di vita della servlet risultante dalla traduzione di una pagina JSP segue perlopiù il ciclo di vita di una comune servlet:

1. Se un'istanza della servlet della pagina JSP non esiste, il *web container*:
   (a) Carica la classe della servlet della pagina JSP.
   (b) Crea un'istanza della classe della servlet.
   (c) Inizializza l'istanza richiamando il metodo *jspInit*().
2. Il *web container* invoca il metodo *_jspService*(), passando come parametri gli oggetti *request* e *response*.

Se il *web container* deve rimuovere la servlet della pagina JSP, richiama il metodo *jspDestroy*().

13.4.3 Esecuzione

È possibile controllare vari parametri di esecuzione della pagina JSP usando le direttive *page*.

Buffering Il flusso scritto nell'oggetto *response* viene automaticamente gestito da un buffer.
È possibile cambiare la dimensione di tale buffer usando la direttiva:

<%@page buffer="none|xxxkb" %>

Gestione degli errori Durante l'esecuzione di una pagina JSP possono essere lanciate illimitate eccezioni.
È possibile indicare al *web container* che, in caso di eccezioni, debba effettuare il *forwarding* a una specifica pagina per la gestione degli stessi tramite la direttiva:

<%@page errorPage="file-name" %>

Una pagina creata per lo specifico scopo di gestire gli errori deve contenere la seguente direttiva:

<%@page isErrorPage="true" %>

È altresì possibile specificare pagine di errore a livello di WAR. Se per una data pagina JSP è disponibile sia una pagina di errore a livello di WAR che a livello specifico di pagina JSP, allora verrà preferita quest'ultima.

13.5 Java Beans

13.5.1 Definizione

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 Anywhere).

- Per essere un Java Bean, una classe:
- Non deve avere variabili d'istanza pubbliche.
- Deve avere il costruttore di default (ovvero senza parametri).
- Deve avere una o più proprietà (variabili d'istanza).

Inoltre, non è necessario che il Java Bean estenda alcuna classe particolare.

13.5.2 Dettagli circa le Proprietà

Una proprietà di un Java Bean può essere semplice (singolo valore) oppure indicizzata (array).
Inoltre, è possibile definire vari tipi di accesso alla stessa:

- read/write
- read-only
- write-only

Il tipo di accesso è automaticamente definito in base ai metodi creati per creati per ottenere/settare la stessa.
Ad esempio, data una proprietà X, potremo settare uno o entrambi i seguenti metodi:

- public <typeof(X)> getX();
- public void setX();

In una pagina JSP, questi metodi vengono richiamati tramite opportune *azioni JSP*.
Per ottenere il valore della proprietà *myProp*:

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

Per impostare il valore della proprietà *myProp*:

```
<jsp:setProperty name="bean" property="myProp" value="test" />
```

13.5.3 Usare un Bean in una JSP

Per usare un Bean in una JSP, è necessario innanzitutto definirlo usando l'*azione JSP* dedicata:

<jsp:useBean *id*="abc" *class*="MyBean" *scope*="session" />

id È il nome della variabile con cui ci si potrà riferire all'oggetto dalla pagina JSP.

class È la classe che implementa il Bean desiderato.

scope Definisce l'ambito di esistenza del Bean; può avere come valore:

1. **application:** il bean esiste finchè l'applicazione è in esecuzione.
2. **session:** il bean esiste fino a che dura la sessione.
3. **request:** il bean esiste per la durata della richiesta.
4. **page:** il bean esiste finchè esiste la pagina (*this*).

13.5.4 Esempio di Java Bean

Il Java Bean in oggetto estende Canvas, ma potrebbe anche non estendere nessuna classe, o una qualsiasi.

```
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() {
            ...
    }
}
```

13.6 Extension Tags (Custom tags)

13.6.1 Introduzione

I *custom tags* sono degli elementi del linguaggio JSP definiti dall'utente, il cui scopo è quello di incapsulare elaborazioni ricorrenti.

Tali tag vengono definiti nelle cosiddette *tag library*. La *tag library* vengono posizionate nella directory /WEB-INF/tag/.

Il codice che viene eseguito da un tag può essere di due tipi:

- **Un file .tag**. Tali file vengono posizionati in */WEB-INF/tags/*.
- **Un metodo di una classe Java dedicata**. Queste classi vengono posizionate in */WEB-INF/classes*.

13.6.2 Sintassi per dichiarare una *tag library* in una pagina JSP:

Sinossi:

```
<%@ taglib prefix="tt" [tagdir=/WEB-INF/tags/dir | uri=URI ] %>
```

Esempio pratico:

```
<%@ taglib prefix="tlt" uri="/WEB-INF/iterator.tld"%>
```

13.6.3 Sintassi per richiamare un *custom tag*:

Un *custom tag* osserva la seguente sintassi:

Tag senza corpo:

```
<prefix:tag attr1="value" ... attrN="value" />
```

oppure, per i tag con corpo:

```
<prefix:tag attr1="value" ... attrN="value" > body </prefix:tag>
```

13.6.4 Librerie di Tag

I *custom tags* sono distribuiti in una o più *tag library*, ognuna delle quali definisce una elenco di *custom tags* correlati tra loro nonchè gli oggetti che implementano quei tag.

Vediamoli nel dettaglio:

Tag Library Descriptor (.tld) Un *tag library descriptor* è un file XML con estensione .tld che contiene informazioni:

- Generiche, sulla libreria.
- Specifiche, su ogni tag contenuto in tale libreria.

I file .tld vanno posizionati nella directory */WEB-INF/* di un WAR oppure nella directory */META-INF/* di un JAR.

La versione di default di un TLD è 2.0; se diversa, va specificato.

Esempio di file .tld:

```
<taglib>
    // Informazioni sulla tag library:
    <tlib-vercsion>1.0</tlib-version>
    <short-name>bar-baz</short-name>
    // Dichiarazione di un tag file (.tag):
    <tag-file>
        <name>d</name>
        <path>/WEB-INF/tags/bar/baz/d.tag</path>
  </tag-file>
    // Dichiarazione di un tag handler (.java):
    <tag>
        <name>elenco</name>
        <tag-class>view.ElencoFilm</tag-class>
        <body-content>empty</body-content>
        [altre proprietà...]
        <attribute>
            <name>test</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>org.joda.time.LocalDate</type>
     </attribute>
        [altri attributi...]
    </tag>
</taglib>
```

Tag Handler (.java) Esempio di richiamo di un *custom tag* gestito tramite un *tag handler*, con settaggio di parametri:

```
<tlt:iterator var="departmentName" type="java.lang.String" group="${myorg.departmentNames}">
    <tr>
        <td>
            <a href="list.jsp?deptName=${departmentName}"> ${departmentName}</a>
     </td>
    </tr>
</tlt:iterator>
```

Esempio di Tag Handler:

```
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.SimpleTagSupport;
public class ElencoAzioni extends SimpleTagSupport {
    private org.joda.time.LocalDate date;
    @Override
    public void doTag() throws JspException {
```

```
        JspContext ctx = getJspContext();
        this.output = ctx.getOut();
        // Elaborazione....
        output.print("Output di esempio");
        // Codice per la gestione del body (se definito):
        // Se non devo manipolare il body:
        getJspBody().invoke(null);
        // Se devo manipolare il body:
        JspFragment f = getJspBody();
        if (f != null) {
            f.invoke(output);
            getOut().println(output.toString().toUpperCase());
        }
    }
```

13.6.5 Ciclo di vita dei Tag

13.7 Esempi di pagine JSP

13.7.1 Parametro da request

```
<%@page errorPage="errorpage.jsp" %>
<html>
    <head>
        <title>Richiesta</title>
    </head>
    <body>
        <b>Hello</b>: "<%=request.getParameter("user"))%>
    </body>
</html>
```

13.7.2 Variabili di sessione

```
<%@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>
```

13.7.3 Idempotenza delle azioni standard JSP

Le *azioni standard JSP* (<jsp:azione />) non aggiungono nulla alle potenziali di Java. Sono semplicemente un modo generalmente più comodo per svolgere operazioni equivalenti in Java.

Ad esempio, le seguenti azioni 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
```

13.7.4 Pagina JSP più completa

Listing 14: Esempio di pagina JSP

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core
"
        prefix="c" %>
<%@ taglib uri="/functions" prefix="f" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application"
    class="mypkg.MyLocales"/>

<form name="localeForm" action="index.jsp" method="post">
<c:set var="selectedLocaleString" value="${param.locale}" />
<c:set var="selectedFlag"
     value="${!empty selectedLocaleString}" />
<b>Locale:</b>
<select name=locale>
<c:forEach var="localeString" items="${locales.localeNames}" >
<c:choose>
    <c:when test="${selectedFlag}">
        <c:choose>
            <c:when
                 test="${f:equals(selectedLocaleString, localeString)}" >
                <option selected>${localeString}</option>
            </c:when>
            <c:otherwise>
                <option>${localeString}</option>
            </c:otherwise>
        </c:choose>
    </c:when>
    <c:otherwise>
        <option>${localeString}</option>
    </c:otherwise>
</c:choose>
</c:forEach>
</select>
<input type="submit" name="Submit" value="Get Date">
```

```
</form>

<c:if test="${selectedFlag}" >
    <jsp:setProperty name="locales"
        property="selectedLocaleString"
        value="${selectedLocaleString}" />
    <jsp:useBean id="date" class="mypkg.MyDate"/>
    <jsp:setProperty name="date" property="locale"
        value="${locales.selectedLocale}"/>
    <b>Date: </b>${date.date}</c:if>
</body>
</html>
```

14 JDBC

14.1 Introduzione

14.1.1 Cos'è

JDBC - Java DataBase Connectivity - è una libreria di Java per l'accesso ai database.
Per poter accedere a diversi tipi di database, JDBC è stata divisa in due layer:

- L'interfaccia standard, comune per tutti i tipi di database.
- Il driver specifico, che implementa l'interfaccia che dipende dal DB. Si tratta di una libreria normalmente contenuta in un file *.jar*.

Per accedere ad un DB dobbiamo quindi aggiungere al *classpath* (solitamente la directory *lib/*) dell'applicazione il driver JDBC opportuno (o al classpath del motore delle servlet, se vogliamo condividerlo con tutte le applicazioni).

14.1.2 Vantaggi

- Inoltro i comandi in SQL (standard).
- Driver indipendenti dal codice: per usare un DB diverso cambio i driver ma non ricompilo (non per forza, almeno).
  - In realtà, un cambio di DB con uno schema complesso implica solitamente una serie di altre modifiche che obbligano a ricompilare il codice.

14.2 Struttura

Per accedere ai dati di un DB con questa libreria dobbiamo:

1. Stabilire una connessione con il DB.
2. Effettuare una query.
3. Elaborare i risultati.

Per ognuna di queste azioni sono previsti oggetti specifici:

14.2.1 Registrazione driver JDBC

Da notare che, prima di creare qualsiasi connessione, è necessario registrare il o i driver JDBC necessari(o):

```
    try {
        Class.forName("org.postgresql.Driver");
} catch(ClassNotFoundException e) {
        System.out.println("Impossibile caricare il driver JDBC PostgreSQL." + e.getMessage());
}
```

è sufficiente farlo una volta sola, a livello di appliceazione.

Se il caricamento della libreria dovesse fallire, la registrazione del driver fallirà, lanciando un'eccezione di tipo *ClassNotFoundException*.

14.2.2 Connection

Per connettersi al DB è necessario istanziare un oggetto di tipo **Connection:**

Esempio:

```
try {
    java.sql.Connection con = java.sql.DriverManager.getConnection(parametri);
} catch(SQLException e) {
    e.printStackTrace();
}
```

Da notare che i *parametri* differiscono in base all'overloading di *getConnection()* che si desidera usare. Alcuni esempi:

Connessione a motore PostgreSQL:

```
String connectionString = "jdbc:postgresql://localhost:1234/myDbName";
String user = "root";
String password = "123";
getConnection(connectionString, user, password);
```

Connessione a motore MS Access, tramite bridge ODBC che fa uso di un DSN (*Data Source Name*) precedentemente salvato. Un DSN è un file di testo contenente le informazioni per stabilire una connessione.

```
getConnection("jdbc:odbc:mySavedDsn);
```

Nota: 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.

14.2.3 Statement(s)

Esistono tre tipi di *statement*:

Statement Esegue query scritte staticamente (stringhe) nel codice. Per poter modificare una di queste query, è necessario farlo da codice e poi ricompilare la classe.

Ogni istanza di tale oggetto è utilizzabile una sola volta.

PreparedStatement Per le query non banali, è largamente preferibile usare *PreparedStatement*, per le seguenti ragioni:

- Consente di eseguire query scritte dinamicamente, passandovi dei parametri.
- I parametri passati sono automaticamente convertiti da tipi Java ai tipi del DB (Es.: stringhe, date, etc.).
- Permette di riutilizzare uno stesso statement più volte (sia con la stessa query, che con query diverse).
- Le query ripetute vengono velocizzate rispetto all'esecuzione tramite *Statement*.
- Evita i problemi di SQL Injection, curando la correttezza sintattica della query.

Esempio di esecuzione di due query con uno stesso PreparedStatement; la prima effettua una *update*, la seconda una *select*:

```
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM clienti WHERE id>? AND
cognome like ?"); // Ogni "?" rappresenta un parametro; indicizzati a partire da 1.
stmt.setInt(1, 123); // Imposto il valore del parametro #1 (intero)
stmt.setString(2, "Rossi"); // Imposto il valore del parametro #2 (stringa)
int nrows = stmt.executeUpdate(); // nrows salva il valore ritornato da executeUpdate(), ovvero il
numero di righe modificate.
ResultSet rs = stmt.executeQuery("SELECT id, cognome, nome FROM clienti"); // Esegue la query
indicata, riutilizzando l'oggetto statement precedentemente istanziato. Ritorna un oggetto RecordSet.
```

CallableStatement Permette di richiamare una *stored procedure.* Accetta anche parametri a runtime.
Lato database, i parametri delle stored procedures sono di tre tipi:

IN parametri in ingresso.

OUT parametri dove la stored procedure salva dei valori in uscita.

INOUT parametri che fungono sia da valori di ingresso, che da segnaposto per i valori di uscita.

Lato JDBC, avrò quindi a disposizione due differenti metodi per registrare i parametri:

```
    try {
        String sql = "{call getEmpName (?, ?, ?)}";
        stmt = conn.prepareCall(sql);
        // I param:
        stmt.setInt(1, myIntVar); // Registro il valore del primo parametro [tipo: IN]
        // II param:
        stmt.registerOutParameter(2, java.sql.Types.VARCHAR); // Registro il tipo di ritorno del
      secondo parametro [tipo: OUT]
        // III param:
        stmt.setInt(3, myIntVarBis); // Registro il valore del terzo parametro [tipo: INOUT]
        stmt.registerOutParameter(3, java.sql.Types.INTEGER); // Registro il tipo di ritorno del
      terzo parametro [tipo: INOUT]
} catch(SQLException e) {
        e.printStackTrace();
}
```

14.2.4 ResultSet

Il ResultSet è sostanzialmente un array bidimensionale di N righe e di M colonne.

Righe [Indici: 1...N] Per navigare tra le righe, è possibile usare i seguenti metodi:

- next()
- previous()
- last()
- first()
- absolute(int pos)

N.B.: non tutti i ResultSet sono scrollabili all'indietro o tramite absolute(); per poterlo fare, bisogna specificare degli specifici parametri al costruttore di ResultSet.

Colonne [Indici: 0...M] Per accedere ad una specifica colonna, è possibile specificare, alternativamente:

- L'indice della colonna
- Il nome della colonna (se presente)

Esempio:

try {

ResultSet rs = stmt.executeQuery("SELECT id, cognome, nome FROM clienti"); // Esegue la query indicata, riutilizzando l'oggetto statement precedentemente istanziato. Ritorna un oggetto *RecordSet*.

while(rs.next()) {

int id = rs.getInt("id");

String cognome = rs.getString("cognome");

String nome = rs.getString(2); // Supponendo di non conoscere il nome della colonna, vi accedo tramite il suo indice.

}

} catch(SQLException e) {

e.printStackTrace();

}

14.2.5 Operazioni finali

rs.close();
stmt.close();
con.close();

14.2.6 Esempio completo PreparedStatement

Vediamo di seguito un esempio completo, che comprende un buon numero di operazioni effettuabili tramite *PreparedStatement*.

Va notato che introduce una serie di accorgimenti importanti. Il codice è commentato ed autoesplicante (grazie a *tutorialspoint.com* per tale codice).

Listing 15: Output

```
1  //STEP 1. Import required packages
2  import java.sql.*;
3
4  public class JDBCExample {
5     // JDBC driver name and database URL
6     static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
7     static final String DB_URL = "jdbc:mysql://localhost/EMP";
8
9     // Database credentials
10    static final String USER = "username";
11    static final String PASS = "password";
12
13    public static void main(String[] args) {
14            Connection conn = null;
15            PreparedStatement stmt = null;
16
17            try{
18                  //STEP 2: Register JDBC driver
19                  Class.forName("com.mysql.jdbc.Driver");
20
21                  //STEP 3: Open a connection
22                  System.out.println("Connecting␣to␣database...");
23                  conn = DriverManager.getConnection(DB_URL,USER,PASS);
24
25                  //STEP 4: Execute a query
26                  System.out.println("Creating␣statement...");
```

```
27                String sql = "UPDATE␣Employees␣set␣age=?␣WHERE␣id=?";
28                stmt = conn.prepareStatement(sql);
29
30                //Bind values into the parameters.
31                stmt.setInt(1, 35); // This would set age
32                stmt.setInt(2, 102); // This would set ID
33
34                // Let us update age of the record with ID = 102;
35                int rows = stmt.executeUpdate();
36                System.out.println("Rows␣impacted␣:␣" + rows );
37
38                // Let us select all the records and display them.
39                sql = "SELECT␣id,␣first,␣last,␣age␣FROM␣Employees";
40                ResultSet rs = stmt.executeQuery(sql);
41
42                //STEP 5: Extract data from result set
43                while(rs.next()){
44                      //Retrieve by column name
45                      int id = rs.getInt("id");
46                      int age = rs.getInt("age");
47                      String first = rs.getString("first");
48                      String last = rs.getString("last");
49
50                      //Display values
51                      System.out.print("ID:␣" + id);
52                      System.out.print(",␣Age:␣" + age);
53                      System.out.print(",␣First:␣" + first);
54                      System.out.println(",␣Last:␣" + last);
55                }
56                //STEP 6: Clean-up environment
57                rs.close();
58                stmt.close();
59                conn.close();
60          }catch(SQLException se){
61                //Handle errors for JDBC
62                se.printStackTrace();
63          }catch(Exception e){
64                //Handle errors for Class.forName
65                e.printStackTrace();
66          }finally{
67                //finally block used to close resources
68                try{
69                      if(stmt!=null)
70                            stmt.close();
71                }catch(SQLException se2){} // nothing we can do
72
73                try{
74                      if(conn!=null)
75                            conn.close();
76                }catch(SQLException se){
77                      se.printStackTrace();
78                }//end finally try
79          }//end try
80          System.out.println("Goodbye!");
81       }//end main
82 }//end JDBCExample
```

14.3 Transazioni

JDBC permette di gestire le transazioni, che sono uno dei fulcri dei database. Tale gestione di effettua attraverso due metodi della classe *Connection*:

conn.commit() rende definitive tutte le modifiche apportate su tale connessione, a partire dal precedente *commit* o *rollback*.

conn.rollback() annulla tutte le modifiche apportate su tale connessione, a partire dall'ultimo *commit*.

Di default, la classe *Connection* rende effettiva ogni modifica effettuata: praticamente, effettua un auto-commit ad ogni modifica apportata al database.

Possiamo invece controllare le transazioni manualmente, tramite i due metodi visti sopra, richiamando il seguente metodo prima di eseguire uno *statement*:

```
conn.SetAutoCommit(false);
```

Vediamo un esempio completo:

```
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
        }
}
```

14.4 Metadati

Tramite JDBC è anche possibile ottenere i metadati di uno Statement, piuttosto che di una Connection.

14.4.1 Statement(s)

Consente di conoscere la struttura della query stessa e quindi conoscere il tipo delle colonne ed eventuali altre informazioni.

Esempio:

```
ResultSet rs = .....; // eseguo una query che ritorni un ResultSet
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 provie-
  ne la colonna
}
```

14.4.2 Connection

Consente di ottenere informazioni circa il database, nonchè circa il driver di database.
Esempio:

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

14.5 Perchè Class.forName()

Ricordiamo che JDBC permette di cambiare il tipo di DB server (e quindi il relativo Driver) successivamente alla compilazione: da classe DriverManager deve quindi per forza gestire a *runtime* il collegamento tra url e Driver relativo.

Per questo il DriverManager pubblica il metodo statico *registerDriver(Driver d)* che permette a un Driver di registrarsi presso il DriverManager. Questo metodo deve venire richiamato da ogni Driver nel suo costruttore.

Quindi, quando avviene questa registrazione? Ovviamente al primo uso della classe.

Class.forName("MyDriver") non fa quindi altro che indurre un primo uso della classe di un *driver*, al fine di consentirgli di registrarsi presso il *DriverManager*.

14.6 Derby DB

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 applicazione.

Per usare Derby in Netbeans, controllare che sia installato il plug-in *Java-Persistence* e aggiungere la libreria *Java DB Driver*.

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*.

15 Sicurezza, LDAP/JNDI

15.1 Gestione accessi tramite il motore delle Servlet

Gli accessi all'applicazione Web possono essere gestiti in automatico dal motore delle *servlet*.

Per abilitare tale opzione, è necessario configurare sia l'applicazione che il server:

15.1.1 Configurazione Applicazione [web.xml]

Per configurare l'applicazione è necessario configurare il file *web.xml*.

In Netbeans, una volta aperto *web.xml*, navigare tramite il menù in alto nella sezione *Security*. Qui si potranno definire i ruoli degli utenti.

15.1.2 Configurazione Server

Nel server va specificato in che modo recuperare i dati degli utenti (login, password, ruolo).

Alcune opzioni:

- File di testo.
- Database.
- LDAP.

In Glassfish, ad esempio, sarà necessario creare un nuovo *Realm*.

15.2 JNDI

15.2.1 Cos'è

JNDI - Java Naming and Directory Interface - è una API Java per servizi di directory che permette di trovare e interrogare dati e oggetti tramite un nome.

JNDI è utilizzato, ad esempio, da Java RMI nonchè da *Java EE*, per cercare oggetti in una rete.

Questa API fornisce:

- Un meccanismo per effettuare il binding di un oggetto a un nome.
- Un'interfaccia per effettuare query a un *servizio di directory*.
- Un'interfaccia di eventi che consente ai client di determinare se gli oggetti della directory sono stati modificati.
- Estensione LDAP per supportare tale protocollo.

Gli usi tipici di JNDI sono:

- Connessione di una applicazione Java ad un *directory service* esterno.
- Permettere ad una Servlet Java di leggere le informazioni fornite dal *Web container* ospitante.

15.2.2 Directory server

Come un database server, anche un directory server memorizza e gestisce delle informazioni.

A differenza di un database relazionale *general purpose*, in un directory server si verificano le seguenti condizioni:

- 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 linguaggio per l'interrogazione più semplice rispetto a SQL.

15.3 LDAP

Parte V
Pattern MVC

16 Introduzione

16.1 Definizione

È un'applicazione dell'Observer pattern alle interfacce utente (GUI ma non necessariamente).

Non è propriamente un *design pattern*, ma più un *architectural pattern*, in quanto i vari ruoli possono essere ricoperti da insiemi di classi anzichè da singole classi.

Il suo intento è di disaccoppiare:

- Rappresentazione del modello di dominio (*model*).
- Interfaccia utente, non necessariamente una GUI (*view*).
- Controllo dell'interazione uomo-macchina (*controller*).

17 MVC in Java EE

17.1 Introduzione

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, che presentano però delle limitazioni:

- Solo JSP: per applicazioni molto semplici, con poco codice Java, che viene gestito con semplici scriptlet e espressioni.
  - Se le applicazioni sono molto complesse, la singola JSP può risultare troppo complicata: troppo codice;
- Utilizzo di Bean e JSP: per applicazioni moderatamente complesse; i bean nascondono la parte di codice più complessa a chi sviluppa la pagina JSP.
  - 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'architettura Model-View-Controller.

17.1.1 Linee guida per l'implementazione

Nell'implementazione dell'architettura MVC in J2EE è buona norma seguire le seguenti linee guida:

- 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 directory "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.

17.1.2 Vantaggi

I principali vantaggi dell'implementazione del pattern MVC sono:

- 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);
- 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;
- 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 definendo un nuovo tag;
- Riuso del modello in contesti applicativi diversi. I *Bean* di per se stessi non sono legati nè al protocollo HTTP, nè al codice HTML. Quindi possono essere riusati in applicazioni differenti (i.e.: in un'applicazione non Web);

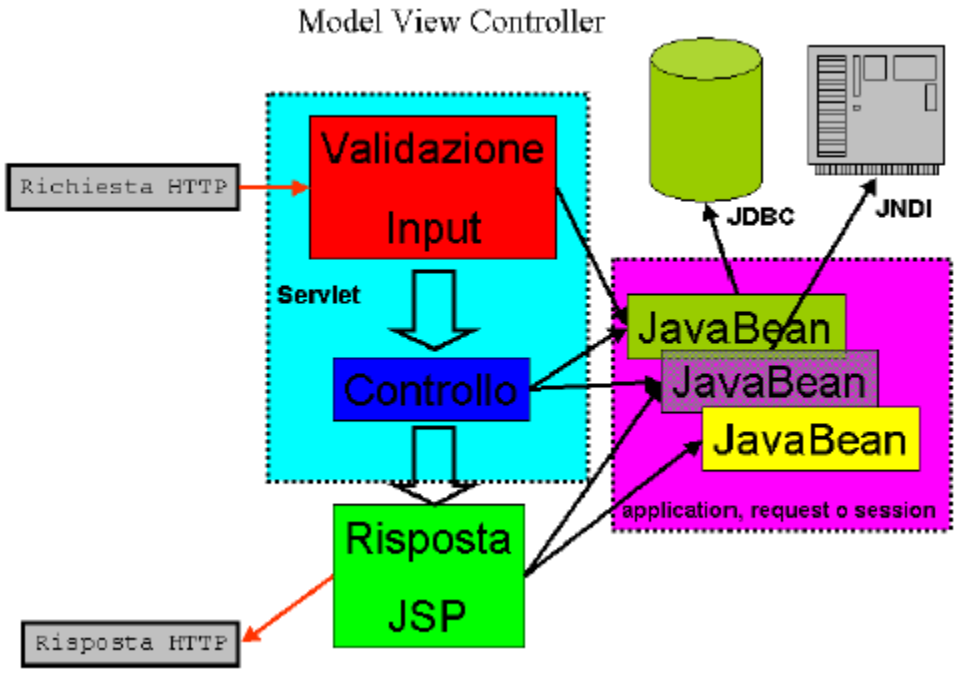
17.1.3 Svantaggi

MVC è complesso da implementare.

17.2 Architettura

17.2.1 Schema MVC

Questo lo schema di MVC in J2EE:



17.2.2 Gestione delle richieste

La richiesta viene gestita come segue, integrando Servlet e JSP:

1. La richiesta originale è processata da 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 ad una pagina JSP per visualizzare il risultato. Possono essere usate JSP differenti in risposta alla stessa servlet per ottenere presentazioni differenti per lo stesso contenuto.

Questo modello è chiamato MVC - *Model View Controller* - oppure anche: *Approccio JSP Model 2.*

17.2.3 Gestione delle richieste

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(jspPage);
rd.forward(request, response);
```

Tali istruzioni possono essere racchiuse in un metodo della servlet come nel seguente **esempio pratico:**

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

17.3 Esempio

17.3.1 Introduzione

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 quale determina l'operazione da effettuare basandosi sul valore di un parametro nascosto della *request*. Abbiamo chiamato tale parametro "op".

La servlet, in base all'azione eseguita ed al suo esito, inoltra la visualizzazione alla pagina opportuna.

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 è accessibile dall'esterno del motore delle servlet. L'unico modo di raggiungerle è tramite *forward* da una Servlet o da un Jsp visibile. Quindi nelle pagine riservate non visibili pubblicamente non riportiamo più tale controllo.

17.3.2 Bean: Ordine.java

Listing 16: Ordine.java

```
1  public class Ordine {
2          static java.util.Map memory = new java.util.HashMap();
3          int progressivo=-1;
4          String descrizione=null;
5
6          public int getProgressivo()
7          {
8                  return progressivo;
9          }
10         public void setProgressivo(int progressivo)
11         {
12                 this.progressivo=progressivo;
13         }
14         public String getDescrizione()
15         {
16                 return descrizione;
17         }
```

```
18          public void setDescrizione(String descrizione)
19          {
20                  this.descrizione=descrizione;
21          }
22          public void insert()
23          {
24                  //... sostituire con accesso DB
25                  memory.put(new Integer(progressivo),this);
26          }
27          public void update()
28          {
29                  //... sostituire con accesso DB
30                  memory.put(new Integer(progressivo),this);
31          }
32          static public Ordine getOrdine(int id)
33          {
34                  //... sostituire con accesso DB
35                  return (Ordine) memory.get(new Integer(progressivo));
36          }
37          static public java.util.Set getOrdini()
38          {
39                  //... sostituire con accesso DB
40                  return memory.entrySet();
41          }
42  }
```

17.3.3 Controller: Servlet.java

Listing 17: Servlet.java

```
1  public Servlet extends HttpServlet{
2
3          public void init(...){
4                  super.init(...);
5                  // creazione e inizializzazione oggetti con scope application
6          }
7
8          private void forward(... request, ... response, ... page) {
9                  [...] // vedi sopra
10         }
11
12         public void doGet/doPost(... request, ... response){
13                 [...]
14
15                 String op = request.getParameter("op");
16                 HttpSession session = request.getSession(true);
17                 Utente u = (Utente) session.getAttribute("user");
18
19                 if ((u==null || op==null) && !"login".equals(op)){
20                         forward(request, response, "/login.jsp");
21                         return;
22                 }
23
24                 if ("login".equals(op) )
25                 {
26                         u = new Utente(request.getParameter("account"));
27
28                         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 [...]

    }
}
```

17.3.4 View: login.jsp

Listing 18: login.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>
```

17.3.5 View: home.jsp

Listing 19: home.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>
```

17.3.6 View: moduloInserimento.jsp

Listing 20: moduloInserimento.jsp

```
[...]

<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>
```

17.3.7 View: mostra.jsp

Listing 21: mostra.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.getProgressivo()%>">Modifica␣questo␣ordine</a>
␣␣␣␣␣␣␣␣</body>
</html>
```

Oppure, un'altra implementazione potrebbe essere:

Listing 22: mostra.jsp (alternativa)

```
[...]

<%@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>
```

17.3.8 View: mostraOrdini.jsp

Listing 23: mostraOrdini.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>
```

17.4 [extra-corso] MVC in ambiente grafico

17.4.1 Problematiche

L'approccio MVC viene spesso applicato alle GUI.
Nasce però il problema di **conciliare**:

- Gli obiettivi di disaccoppiamento di MVC.
- I vantaggi dati dall'utilizzo di toolkit grafici.
- Portabilità del codice (o di gran parte di esso).

Ciò si verifica in quanto, spesso, i **toolkit grafici**:

- Sono dipendenti dalla piattaforma.
- Non impongono uno schema progettuale (MVC o altro).
- Se utilizzati in modo "ingenuo" portano a sistemi mal progettati:
  - Alto accoppiamento
  - Bassa coesione

17.4.2 Azioni utente

In un ambiente grafico, un'azione utente può essere rappresentata da un'icona, un nome o altro e può avere associata una o più descrizione (*tooltip*, *help contestuale*).

Un'azione può essere non ammessa: quando è così, tutti i componenti grafici che permettono quell'azione devono essere disabilitati.

17.4.3 Componenti

model Indipendente dall'ambiente grafico.

view Formate da componenti grafici.

controller Aggregano azioni che un utente può effettuare su un modello.

17.4.4 Controller: scelte progettuali

Il controller può essere impostato in vari modi, vediamoli:

Punto di vista vicino al *model* Il controller:

- Dovrebbe raggruppare le azioni/transazioni "complesse" che possono essere effettuate sul modello.
- È del tutto indipendente da aspetti tecnologici legati all'interfaccia grafica.

Punto di vista vicino alla *view*

- Il controller gestisce eventi (mouse clicks).
- Deve fornisce una interfaccia che sia "compatibile" con il toolkit grafico utilizzato.
- Può rappresentare il concetto di azione "elementare" effettuabile tramite l'interfaccia grafica – abilitazione/-disabilitazione, nomi, tooltip, etc.

Suddiviso per essere vicino sia al *model* che alla *view* È anche possibile un approggio che fonda le due soluzioni:

- Dividiamo le responsabilità in due categorie di oggetti distinti.
- Ciascuna categoria soddisfa una delle esigenze.

Categorie:

- **controller** (propriamente detti):
  - tutto ciò che non è legato al toolkit grafico.
- **azioni**:
  - Tutto ciò che è legato a toolkit grafico.
  - Sono spesso classi molto piccole che delegano la maggior parte delle attivitè al controller.
  - Possono essere considerate degli adapter verso il controller.

Parte VI
Android

18 Sistema operativo Android

18.1 Dalvik VM

18.1.1 Introduzione

Android è un sistema operativo orientato ai dispositivi di tipo *mobile*.

Java rende disponibile una Virtual Machine per sistemi *mobile*: la J2ME. Tuttavia, Android non fa uso di questa VM di Java.

Android usa - invece- la **Dalvik Virtual Machine** per ragioni economiche e di performance.

18.1.2 Caratteristiche

Dimensioni: pacchetti .apk Non accetta file *.class* ma un unico pacchetto *.apk* con un *.dex* che contiene tutte le classi.

Pro In questo modo si ottimizzano le dimensioni del file: no ripetizione costanti, no ripetizione header, etc. Mediamente, si risparmia il 50% delle dimensioni rispetto a J2ME.

Contro Non è possibile caricare dinamicamente le classi.

Velocità: registri, non stack Il bytecode è orientato ai registri (e non allo stack come la J2ME). Per una maggiore velocità di esecuzione e compattezza del codice.

Pro Codice più compatto e maggiore velocità di esecuzione. Mediamente, produce il 30% di istruzioni in meno rispetto a J2ME.

Contro Maggiore difficoltà di compilazione; più basso riuso del bytecode.

18.1.3 Applicazioni Android

Android è un sistema operativo derivato da Linux ma con delle differenze importanti nella gestione delle applicazioni.

Android, come Linux, esegue ogni applicazione in un processo separato. In più rispetto a Linux, esegue però ognuno di questi processi in una Virtual Machine dedicata, nonchè con le credenziali di un utente dedicato, isolando ogni applicazione in una sua *sandbox*. Questo dà vita al *principio dei minimi privilegi*: ogni applicazione ha accesso solamente ai componenti necessari per svolgere le sue mansioni, e null'altro. Questo aumenta la sicurezza del sistema.

Ogni volta che viene richiesto di eseguire una componente di una data applicazione, Android crea un processo per l'applicazione. Quando l'applicazione non è più necessaria, o quando il sistema deve liberare spazio in memoria per altre applicazioni, il processo viene terminato.

Permessi È tuttavia possibile che un'applicazione necessiti di dialogare con altre applicazioni o di accedere a servizi di sistema; in questi casi è possibile:

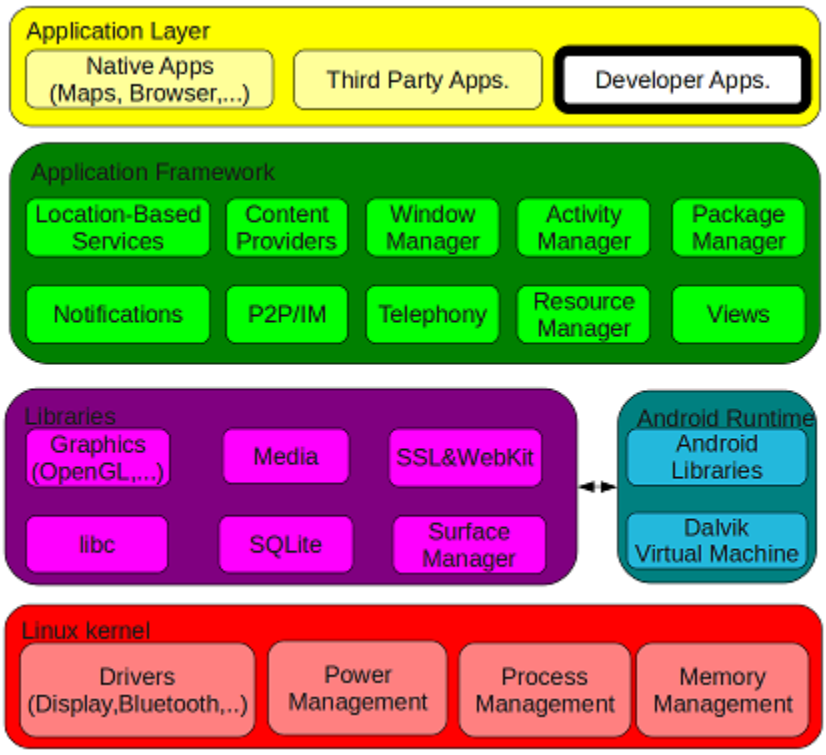
- Impostare che due applicazioni condividano lo stesso *Linux user ID*, in modo che possano accedere reciprocamente ai loro file. Per risparmiare risorse, è anche possibile far sì che queste applicazioni girino sullo stesso processo nonchè sulla stessa VM. A questo fine le due applicazioni vanno contrassegnate con lo stesso certificato.
- Un'applicazione può richiedere di accedere a dati del device (i.e.: contatti, fotocamera, Bluetooth, etc.). Questi permessi vanno richiesti esplicitamente all'utente in fase di installazione.

18.1.4 Tecnica Zygote

Per aumentare la velocità di startup delle DVM - *Dalvik Virtual Machine* - su cui girano i vari processi, il caricamento di ogni DVM dev'essere veloce.

Per questo motivo, Android sfrutta una tecnica chiamata **Zygote** che permette la condivisione ed il precaricamento di tutte le librerie core.

Questo lo schema dell'architettura di Android:



18.2 SDK e NDK

18.2.1 SDK

La SDK - *Standard Development Kit* - di Android fornisce tutte le librerie API e gli strumenti per sviluppare, testare e rilasciare applicazioni per O.S. Android. L'SDK di Android è chiamata ADT - *Android Developer* Tools.

La SDK include anche un emulatore di terminale mobile Android (*Android emulator*).

L'SDK può essere sia installata come plugin per altri IDE, tra cui il più comune è *Eclipse*, ma Google sta sviluppando *Android Studio*, che è basato su *IntelliJ IDEA* ed attualmente è in *beta* (v.0.8.6). Android Studio al momento non include il supporto per NDK.

18.2.2 NDK

L'NDK - *Native Development Kit* - è un insieme di strumenti che permettono di implementare parti dell'applicazione Android utilizzando linguaggi a codice nativo, come C o C++.

Questo può risultare utile per certi tipi di applicazione, in quanto consente di riutilizzare librerie scritte in tali linguaggi, ma la maggior parte delle applicazioni non necessita dell'NDK.

Prima di fare uso dell'NDK, è importante effettuare una valutazione che tenga conto dei pro e dei contro della sua introduzione. Difatti, nonostante il codice nativo dia un miglioramento delle performance, al contempo aumenta sensibilmente la complessità dell'applicazione. Riassumendo, conviene utilizzare NDK solo quando strettamente necessario, e non per semplice predilezione verso la programmazione in codice nativo.

Esempi di buoni candidati per l'utilizzo di NDK sono applicazioni che utilizzano intensamente la CPU: motori di giochi, processione di segnali, simulazioni fisiche, etc. Anche in questi casi, controllare prima che il framework delle API Android non fornisca già le funzionalità richieste.

18.3 Compilazione JIT

18.3.1 Premessa - Compilazione e bytecode

I linguaggi di programmazione possono essere:

Compilati in codice macchina Un programma compilato in codice macchina non richiede alcuna azione aggiuntiva per la sua esecuzione, ma deve essere stato compilato usando come *target* l'architettura della macchina su cui lo si vuole eseguire.

Diventa inoltre complicato applicare ottimizzazioni legate alla macchina su cui verrà eseguito: ad esempio, sfruttare le librerie disponibili per lo specifico processore. Il caso ottimale prevederebbe quindi di compilarlo direttamente sulla macchina su cui verrà eseguito (*cfr. filosofia di Gentoo Linux*), applicando le opportune ottimizzazioni, ma tale opzione risulta quasi sempre improponibile.

Interpretati a run-time Un programma interpretato ha il vantaggio di essere interpretato direttamente sulla macchina su cui viene eseguito, avendo così la possibilità di sfruttarne le specifiche capacità. Ha anche notevoli doti di portabilità.

Mentre un compilatore compila il codice in *linguaggio macchina*, l'interprete non fa altro che eseguire le azioni indicate ad alto livello nel codice oggetto.

Un interprete può anche disporre di un compilatore interno che compila parti di codice, per poi richiamare direttamente eventuali blocchi già compilati.

Un linguaggio interpretato ha un grande *overhead* legato all'interpretazione del codice ogni volta che viene eseguito.

Compilati in bytecode Definiamo il bytecode come una traduzione di codice sorgente in un *instruction set* definito, interpretabile/compilabile dalla VM in linguaggio macchina.

I principali vantaggi sono:

- Portabile, non essendo specifico di una particolare architettura (WORA: *Write Once, Run Anywhere*).
- Veloce da interpretare/compilare all'esecuzione da parte della VM, in quanto gran parte del tempo totale è richiesto per fase di compilazione da sorgente a bytecode, mentre la compilazione da bytecode a codice macchina è molto più rapida.

18.3.2 Compilatore JIT

Il bytecode comporta comunque un grosso problema: il ritardo iniziale dovuto alla compilazione del bytecode.

Questo ritardo iniziale viene ridotto drasticamente tramtie la compilazione *Just In Time* (a.k.a. *dynamic translation*), che non compila tutto il bytecode prima dell'esecuzione, bensì ne compila frammenti man mano che vengono richiesti.

Il *JIT compiler* si occupa anche di effettuare il *caching* delle parti già compilate.

Ogni compilatore JIT è impostato per lavorare con una specifica unità di compilazione:

- Per file.
- Per metodi.
- Per blocchi di codice arbitarti.

18.3.3 JIT a granularità di Metodo

Se la JIT della VM è a granularità di Metodo, allora la sezione atomica (*unità di compilazione*) che sarà compilata dal *JIT compiler* sarà il singolo metodo.

L'unità di compilazione standard è per l'appunto il singolo metodo.

18.3.4 JIT a granularità di Trace

Se il *JIT compiler* della VM è a granularità di Trace, allora l'unità di compilazione è costituita diversamente e può comprendere più metodi (o parti di più metodi).

Questo approccio è più complesso, ma consente migliori performance, in quanto consente di:

- Compilare porzioni di codice più specifiche.
- Ottimizzare le porzioni di codice usate più frequentemente..

Come funziona il *JIT compiler* a granularità di *trace*:

- **Fase di profilazione:** riconosce i cicli caldi (*hot loops*), ovvero quelli che vengono eseguito più di un certo numero di volte.
- **Fase di tracing:** crea un *trace* di questi cicli, mediante l'osservazione dell'esecuzione di un ciclo completo. Il *trace* risultante è codice macchina lineare che prevede:
  - Operazioni ottimizzate.
  - Spostamento di codice invariante: eventuali funzioni banali vengono trasformate in codice *inline*, risparmiando quindi le operazioni di JUMP ed il passaggio di parametri.
  - Eliminazioni di subespressioni costanti.
  - Eliminazione del codice mai eseguito.
  - Allocazione dei registri.
  - Inserimento di eventuali istruzioni di tipo *guard* (vedi sotto).
- **Fase di esecuzione:** quando viene richiamato un *hot loop*, richiama il *trace* corrispondente anzichè la compilazione originale.
  - Dato che differenti esecuzioni del ciclo potrebbero portare a differenti flussi di esecuzioni, sono inserite delle istruzioni *guard*: se si dovesse uscire dal flusso tracciato, viene terminata l'esecuzione del *trace* e viene avviata l'esecuzione del codice compilato partendo dal bytecode originale.
  - Le istruzioni *guard* corrispondono in genere ai costrutti IF.

Esempio di JIT compiler a granularità di trace Dato il programma:

```
    def square(x):
        return x * x
i = 0
    y = 0
    while True:
        y += square(i)
        if y > 100000:
            break
      i = i + 1
```

Il trace corrispondente potrebbe essere:

```
loopstart(i1, y1)
i2 = int_mul(i1, i1) # x*x, trasformata da funzione a operazione inline
y2 = int_add(y1, i2) # y += i*i
b1 = int_gt(y2, 100000)
guard_false(b1) # L'if viene trasformato in guard_false()
i3 = int_add(i1, 1) # i = i+1
jump(i3, y2)
```

19 Ciclo di vita Applicazioni Android

19.1 Ciclo di vita dell'applicazione

19.1.1 Tipi di processo

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
- Servizi 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)

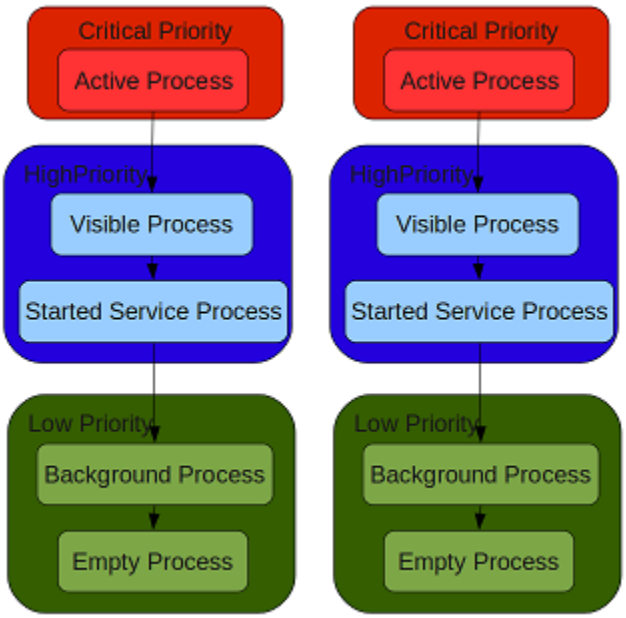
Started Service Process: processi che ospitano servizi in esecuzione

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.

19.1.2 Priorità dei processi

La priorità ai processi viene assegnata secondo la seguente idea:



19.2 Classe Application

19.2.1 Introduzione

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

- Mantenere lo stato dell'applicazione.
- Trasferire 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.

Ad esempio:

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

19.2.2 Eventi del ciclo di vita della classe Application

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 singleton e per creare e inizializzare tutte le variabili di stato e le 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.

19.2.3 Esempio

Esempio di estensione della classe *Application*:

```
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);
            ...
    }
}
```

20 Struttura applicazione Android

20.1 Manifest.xml

20.1.1 Definizione

Questo file presenta al sistema operativo Android le informazioni essenziali riguardo l'applicazione. Ogni applicazione deve avere un file *AndroidManifest.xml* nella sua root directory.

Vediamo le principali funzioni di AndroidManifest.xml:

- dà il nome al package Java dell'applicazione, il quale serve come identificatore univoco per l'applicazione;
- descrive i componenti dell'applicazione, activities, services, broadcast receivers, e content providers di cui l'applicazione è composta. Dà un nome alle classi che implementano e pubblica ciò di cui sono capaci. Tali dichiarazioni permettono al sistema operativo quali siano i componenti e di cosa sono capaci;
- determina quali processi ospiteranno i componenti dell'applicazione;
- determina quali permessi ha l'applicazione per comunicare con le altre applicazioni o con i componenti del sistema operativo;
- elenca le classi Instrumentation che permettono la profilazione e altre informazioni mentre l'applicazione è in uso. Queste dichiarazioni sono presenti nel manifest solo mentre l'applicazione viene sviluppata e testata, e successivamente vengono rimosse prima che venga pubblicata;
- determina la minima versione del sistema operativo necessaria a far girare l'applicazione;
- enumera le librerie necessarie all'applicazione;

20.1.2 Regole

Vengono applicate alcune regole a tutti gli elementi del manifest:

- Elementi: sono richiesti solo gli elementi <manifest> e <application>, devono essere entrambi presenti e devono occorrere solo una volta. La maggior parte degli altri elementi può apparire più volte o nemmeno una, tuttavia alcuni elementi dovrebbero apparire per rendere utile il manifest. Se un elemento non contiene nulla, contiene altri elementi. Tutti i valori sono definiti tramite attributi, non come dati all'interno di un elemento. Elementi dello stesso livello sono generalmente non ordinati.

- Attributi: in senso formale tutti gli attributi sono opzionali, tuttavia alcuni elementi non hanno senso senza di essi. E' utile in tal senso usare la documentazione come una guida; per gli attributi veramente opzionali infatti specifica un valore di default o spiega cosa siuccede in assenza diu una specifica. Eccetto per alcuni attributi del <manifest>, tutti gli attributi iniziano con un prefisso android: .
- Dichiarare nome di classe: molti elemnti corrispondono a degli oggetti Java, inclusi elementi per l'applicazione stessa o i suoi componenti. Se ad esempio si definisce una sottoclasse, quest'ultima viene dichairata tramite l'attributo name, che deve includere l'intera definizione del package. Esempio: <service android:name="com.example.project.SecretService">. Tuttavia se il primo elemnto della stringa è un punto questo viene appeso alla fine del package che da il nome all'applicazione definito nell'elemento <package> del <manifest>.
- Valori multipli: se può essere specificato più di un valore, l'elemento viene quasi sempre ripetuto, piuttosto che specificare più valori per un singolo elemento;
- Valori delle risorse: alcuni attributi hanno valori che possono essere fatti vedere agli utenti, pere esempio un icona per un activity. I valori di questi attributi dovrebbero essere localizzati e settati da una risorsa o da una theme. I valori presi da una risorsa son definiti nella seguente maniera: @[package:]type:name, ad esempio <activity android:icon="@drawable/smallPic">, mentre le variabili da una theme sono espresse in un modo simile, solo che al posto dell'@ usano ?.
- Valori delle stringhe: quando il valore di un attributo è uan stringa deve essere usato un doppio backslash per definire i caratteri di escape.

20.1.3 Esempio

Esempio di 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>
```

20.2 Esternalizzazione delle risorse (strings.xml, dimen.xml, ecc.)

20.2.1 Introduzione

Le applicazioni Android prevedono una gestione delle risorse estremamente centralizzata.
Queste risorse vengono:

- Se testuali, definite in file .xml salvati in specifici *path*.
- Se binarie (es. immagini), salvate in specifici *path*.

Un file XML di stringhe di esempio potrebbe essere:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello</string>
    <string name="hi">@string/hello</string>
    <color name="yellow">#f00</color>
    <color name="highlight">@color/red</color>
</resources>
```

20.2.2 Accesso alle risorse da file XML

È possibile accedere alle risorse da file XML come nei seguenti esempi.

Risorse definite dall'utente

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/submit" />
```

Risorse di sistema

```
attribute="@android:string/selectAll"
```

Attributi di stile

```
<EditText
    id="text"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondary" // Attributo di stile
    android:text="@string/hello_world" />
```

20.2.3 Accesso alle risorse da codice

È poi possibile **accedere da codice** a tutte le risorse create facendo riferimento all'oggetto **R** (*singleton*). Alcuni esempi:

```
// Carica un'immagine:
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
//
// Load a background for the current screen from a drawable resource:
getWindow().setBackgroundDrawableResource(R.drawable.my_background_image) ;
//
```

```
// Set the Activity title by getting a string from the Resources object, because
// this method requires a CharSequence rather than a resource ID:
getWindow().setTitle(getResources().getText(R.string.main_title));
//
// Load a custom layout for the current screen setContentView(R.layout.main_screen);
// Set a slide in animation by getting an Animation from the Resources object:
mFlipper.setInAnimation(AnimationUtils.loadAnimation(this, R.anim.hyperspace_in));
//
// Set the text on a TextView object using a resource ID:
TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText(R.string.hello_message);
```

20.3 La cartella res e le sue impostazioni

Una struttura di esempio per una semplice applicazione potrebbe essere:

```
MyProject/
    src/
        MyActivity.java
  res/
        drawable/
            icon.png
      drawable-hdpi/ //Risorse alternative a drawable
            icon.png
            background.png
      layout/
            main.xml
            info.xml
      values/
            strings.xml
```

20.4 Cambiamenti di configurazione a runtime

20.4.1 android:configChanges

Android gestisce cambiamenti al runtime di:

- lingua
- location
- hardware

e lo fa terminando e facendo ripartire l'attività.

Quando questo comportamento non ci e' gradito bisogna modificare il Manifest, aggiungendo nell'attività 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 il delimitatore: | (pipe).

20.4.2 Esempio

In questo esempio andiamo a comunicare all'applicazione che, per gli eventi di tipo keyboard e orientation, l'Activity **non** dovrà essere distrutta e ricreata.

Manifest.xml Dichiariamo nel manifest l'attributo *configChanges* per l'attività interessata:

```
<activity
    android:name="TestConfigHandling"
    android:configChanges="keyboard|orientation">
</activity>
```

Activity Andremo poi a sovrascrivere il metodo *onConfigurationChanged* nell'attività, gestendo da tale metodo il comportamento dell'attività di fronte a tali eventi:

```
@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
    }
}
```

21 Componenti applicazione Android

Le componenti di un'applicazione Android 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.

21.1 Activity

21.1.1 Introduzione

Una Activity è un componente di un'applicazione Android che fornisce una schermata con cui un utente può interagire.

Ad ogni Activity è fornita una finestra dove disegnare la sua interfaccia utente (*UI - User Interface*). Questa finestra può essere a tutto schermo, piuttosto che ridotta e flottante sopra ad altre finestre.

Solitamente un'applicazione consta di più Activity, i cui confini sono labili.

Sono disponibili alcune classi Activity predefinite, le più comunemente utilizzate sono:

- MapActivity.
- ListActivity.
- ExpandableListActivity.
- TabActivity.

21.1.2 Il *back stack*

Ogni Activity può avviarne un'altra; quando questo avviene:

1. l'attività precedente viene stoppata, ma il sistema la conserva nello *stack* (il *back stack*).
2. quando la nuova attività viene avviata, viene inserita nel *back stack* e posta in cima allo stesso.

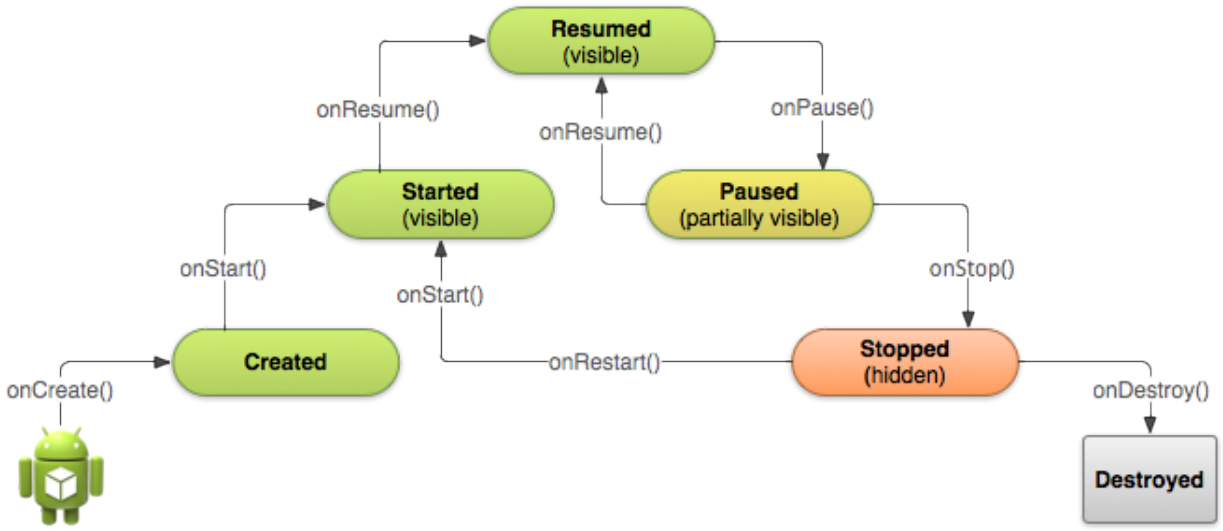
Il *back stack* segue una logica LIFO - *Last In, First Out* - quindi, quando l'utente preme il tasto *indietro*, viene effettuato il pop() della prima attività dallo stack, e viene fatta ripartire la precedente attività.

Quando un'attività viene stoppata, questo cambiamento di stato le è notificato tramite i metodi dell'*activity lifecycle*.

21.2 Ciclo di vita di un'Activity

21.2.1 Introduzione

Il ciclo di vita un'Activity può essere riassunto in questa visione semplificata:



Di questi stadi, alcuni sono solamente di transizione, mentre altri prevedono che l'attività possa soffermarvicisi anche per lungo tempo.

Gli stati in cui una Activity si può soffermare sono:

- Resumed
- Paused
- Stopped

Implementing your activity lifecycle methods properly ensures your app behaves well in several ways, including that it:

- Does not crash if the user receives a phone call or switches to another app while using your app.
- Does not consume valuable system resources when the user is not actively using it.
- Does not lose the user's progress if they leave your app and return to it at a later time.
- Does not crash or lose the user's progress when the screen rotates between landscape and portrait orientation.

21.2.2 Tempi delle attività

Ogni attività, secondo lo schema di cui sopra, può effettuare transizioni tra i vari stati. Tali transizioni possono essere considerate per delimitare concettualmente delle macro-fasi della vita dell'attività.

Vediamo come:

- Tempo di vita
  - Va dal metodo *onCreate(Bundle savedInstanceState)* al metodo *onDestroy*().
- Tempo di visibilità
  - Va dal metodo *onStart()* al metodo *onStop()*.
  - I metodi *onStart()* e *onStop()* sono anche usati per registrare/cancellare i Broadcast receiver, la cui unica funzione è di aggiornare l'interfaccia utente.
- Tempo di attività
  - Va dal metodo *onResume()* al metodo *onPause()* [praticamente, lo stato *Resumed*].
  - Un'attività esce dallo stato attivo ogni volta che:
    - Una nuova attività diventa attiva;
    - Il dispositivo va in sleep;
    - L'attività perde il focus.
  - Subito prima di *onPause()*, viene invocato *onSaveInstanceState(Bundle outState)*, che si occupa di salvare lo stato corrente della View in un oggetto Bundle.

21.2.3 Launcher Activity

Ogni applicazione può avere una cosiddetta *Launcher activity*, anche detta *main activity*. Quest'attività verrà lanciata automaticamente all'avvio dell'applicazione.

Per dichiararla, è necessario inserire all'interno della definizione dell'**activity** nel file Manifest.xml un intent-filter con definiti entrambi gli attributi di cui nell'esempio:

```
<activity android:name=".MainActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Un'applicazione sprovvista di una main activity non comprarirà tra le icone dell'home screen di Android.

21.2.4 onCreate(): istanziare un'Activity

Vediamo come potrebbe essere implementato, per puro esempio, il metodo onCreate():

```
TextView mTextView; // Member variable for text view in the layout
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);
    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

21.2.5 onDestroy(): distruggere un'Activity

In questo metodo non ci dovrebbe essere molto da fare, in quanto si dovrebbe aver già provveduto a chiudere quasi tutte le risorse nel metodo *onStop()*.

Potrebbe essere il posto giusto per stoppare eventuali *tracing*, nonchè per controllare che eventuali thread precedentemente creati non siano più in esecuzione, etc.

Esempio di onDestroy():

```
@Override
public void onDestroy() {
    super.onDestroy(); // Always call the superclass
    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

21.2.6 onPause(): mettere in pausa un'Activity

Se durante l'esecuzione di un'Activity ne viene avviata un'altra, nel più dei casi la prima vede richiamati i suoi metodi onPaused() e, subito dopo, onStopped().

Se però l'Activity appena lanciata non oscura totalmente la precedente Activity, allora verrà richiamato solamente il metodo onPause().

Questo avviene quando:

- La nuova Activity è trasparente (anche solo parzialmente).
- La nuova Activity non ricopre tutta l'area dell'Activity precedente.

In questo metodo è importante:

- Bloccare le azioni (es.: animazioni, etc.) che potrebbero consumare CPU.
- Rilasciare le risorse di sistema allocate nel metodo *onResumed()*.
- Salvare le modifiche, ma solo se l'utente si aspetta che ciò avvenga (es.: bozza di email).

Nel metodo onPause() è importante evitare di effettuare salvataggi di dati quando non necessario, in quanto ciò comporterebbe un rallentamento della nuova Activity che è stata lanciata. Per lo stesso motivo, tale metodo va mantenuto in genere il più "leggero" possibile.

Esempio di onPause():

```
@Override public void onPause() {
    super.onPause(); // Always call the superclass method first
    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

21.2.7 onResume(): far ripartire un'Activity

Questo metodo viene richiamato quando l'attività viene portata nello stato *Resumed*, anche quando vi entra per la prima volta.

Pertanto, i componenti qui inizializzati dovrebbero essere speculari a quelli dealloccati nel metodo *onPause()*.

Esempio di onResume():

```
@Override public void onResume() {
    super.onResume();
    // Always call the superclass method first
    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

21.2.8 onStop(): stoppare un'Activity

Lo stadio *Stopped* è importante per dare all'utente l'impressione che un'app sia sempre attiva e non perda le sue informazioni di stato.

Un'Activity in questo stato conserva la sua istanza in memoria; quando torna in stato *Resumed*, non sarà quindi necessario ricreare i *widget*, nè ricaricare eventuali dati inseriti dall'utente (ad es., in una casella di testo) in quanto ci ritroveremo tutto ciò che avevamo lasciato.

A una *Stopped* Activity è garantito che:

- L'interfaccia utente non è più visibile.
- Il *focus* dell'utente è su un'altra Activity o applicazione.

Ad esempio, un'Activity viene portata nello stadio *Stopped* quando:

- L'utente apre la finestra *Applicazioni recenti* e passa ad un'altra Activity.
- L'utente esegue un'azione nell'Activity che provoca il lancio di una nuova Activity.
- L'utente riceve una telefonata mentre usa l'Activity.

Quando viene richiamato il metodo *onStop()*, è importante rilasciare tutte le risorse che in questo stadio non servono.

Bisogna tenere conto che in particolari casi il sistema operativo, se dovesse necessitare di risorse, potrebbe distruggere un'Activity che si trova in questo stadio senza neppure richiamare il metodo *onDestroy()*.

Risulta quindi importante rilasciare le risorse che potrebbero comportare delle *memory leak*.

Dato che l'istanza dell'applicazione viene mantenuta in memoria anche quando questa passa allo stadio *Stopped*, spesso non è necessario effettuare alcuna operazione particolare nel metodo *onStop()*.

Esempio di onStop():

```
@Override
protected void onStop() {
    super.onStop(); // Always call the superclass method first
    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());
    getContentResolver().update(
        mUri, // The URI for the note to update.
        values, // The map of column names and new values to apply to them.
        null, // No SELECT criteria are used. null // No WHERE columns are used. );
}
```

21.2.9 onStart() e onRestart(): avviare/far ripartire un'Activity

Quando un'attività passa dallo stato *Stopped* allo stato *Resumed*, passa attraverso lo stato *Started*, dove si comporta come già visto (metodo *onStart()*).

Questa transizione di stato (*Stopped* -> *Resumed*) vede però venir richiamati dal sistema **due** metodi:

- onRestart();
- onStart();

Specifichiamo innanzitutto che la controparte di *onStop()* è generalmente da individuarsi nel metodo *onStart()*.

Nel metodo *onRestart()* andranno effettuate delle operazioni di *restore* solo in casi particolari: difatti *onRestart()* è raramente utilizzato.

Esempio di *onStart()*:

```
// L'activity potrebbe essere rimasta in background per parecchio tempo:
// è quindi il posto giusto dove controllare, ad esempio, che determinate risorse
// di sistema siano ancora disponibili.
@Override
protected void onStart() {
    super.onStart(); // Always call the superclass method first
    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS
     action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
```

```
}
    @Override
    protected void onRestart() {
        super.onRestart();
        // Always call the superclass method first
        // Activity being restarted from stopped state
}
```

21.2.10 Approfondimento su *onCreate():* ri-creare un'Activity

Notiamo innanzi tutto che un'Activity può essere distrutta in due modi:

- Distrutta per sua naturale terminazione:
  - l'Activity è definitivamente deceduta.
- Distrutta forzatamente dal sistema operativo per liberare risorse:
  - il sistema salva ogni View dell'attività in un **Bundle**: in questo modo, quando l'attività verrà rilanciata, il sistema le passerà questo **Bundle** contenente il suo stato al momento della terminazione, e che verrà automaticamente ripristinato.
  - ciononostante, potrebbero esservi altri dati che si desidera ripristinare (es.: variabili). Per ripristinare tali dati è però prima necessario salvarli, e questo va fatto nel metodo *onSaveInstanceState()*.
  - Nota bene: questa operazione è possibile solamente per le attività in cui ogni *View* dispone di un proprio attributo **android:id**.

Un'attività viene distrutta e ricreata ogni qualvolta l'utente ruota lo schermo, a meno che non si impostino degli appositi parametri nel Manifest. Questo per dare la possibilità all'applicazione di caricare risorse differenti (es.: layout) in base all'orientamento dello schermo, ad esempio.

In realtà vi sono altri due metodi che vengono richiamati nelle transizioni di stato:

onSaveInstanceState(): viene richiamato dal sistema nella transizione da *Paused* a *Stopped* e si occupa di salvare un **Bundle** (coppie chiave-valore) delle View dell'attività. È inoltre possibile aggiungere a tale salvataggio dello stato dell'attività anche variabili personali (ad esempio il punteggio del giocatore, etc.).
Esempio:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);
    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

onRestoreInstanceState(Bundle b): viene richiamato dal sistema dopo il metodo *onStart*() solamente nel caso in cui l'activity fosse stata terminata dal sistema. Tale metodo si occupa di ripristinare lo stato delle View precedente alla terminazione dell'activity. Se un utente ha aggiunto dei dati specifici al Bundle, può recuperarli come segue:

```
@Override
public void onRestoreInstanceState(Bundle b) {
    // Richiamo il padre per consentire il ripristino delle View:
    super(b);
    // Recupero dal Bundle le informazioni che mi interessano:
    Integer punteggioSalvato = b.get("punteggio");
    // Ripristino tali informazioni nelle mie strutture dati:
    this.punteggio = punteggioSalvato;
}
```

21.3 Layout

21.3.1 Introduzione

I layout definiscono la struttura visuale di un'UI - *User Interface,* interfaccia utente - ad esempio per una *Activity* o per un *App widget*.

Vi sono due modi per dichiare un layout:

- **Dichiarazione degli elementi dell'UI in XML:**
  - Android fornisce un vocabolario XML che corrisponde alle classi e sottoclassi di *View*, come quelle per i *widget* ed i layout.
- **Istanziazione di layout a runtime:**
  - Un'applicazione può creare oggetti *View* e *ViewGroup*, e manipolarli, direttamente da codice.

È possibile usare uno di questi metodi alla volta, o anche entrambi nella stessa UI. Ad esempio, è possibile caricare un layout XML che poi viene variato, aggiungendo o modificando elementi, a runtime.

Usare i layout in XML dà il vantaggio di separare meglio la presentazione dell'applicazione dal codice che ne controlla il comportamento. Essendo questi file esterni al codice dell'applicazione, è possibile modificare i layout senza ricompilare. Ad esempio, è possibile creare layout diversi per diversi orientamenti dello schermo, piuttosto che per lingue differenti.

21.3.2 Creazione di un XML

Ogni file di layout deve contenere un unico elemento radice, che dev'essere un oggetto *View* oppure *ViewGroup* o ancora *merge*, ed avere estensione *.xml*.

Una volta definito l'elemento *root*, è possibile aggiungere come figli oggetti di tipo layout piuttosto che dei *widget*, in modo gerarchico. L'elemento *root* deve obbligatoriamente avere il parametro **xmlns:android**.

I nomi dei tag nei layout XML sono tendenzialmente uguali ai nomi delle classi e delle loro proprietà, ma non sempre.

Esempio di layout XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Mentre la sinossi di un layout XML è la seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+][package:]id/resource_name"
    android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
    android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
    [ViewGroup-specific attributes] >
    <View
        android:id="@[+][package:]id/resource_name"
        android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
        android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
        [View-specific attributes] >
        <requestFocus/>
    </View>
    <ViewGroup >
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

21.3.3 Risorse di un layout XML

FILE LOCATION:

res/layout/filename.xml

The filename will be used as the resource ID.

COMPILED RESOURCE DATATYPE: Resource pointer to a View (or subclass) resource.

RESOURCE REFERENCE:

- In Java: R.layout.filename
- In XML: @[package:]layout/filename

21.3.4 Elementi di un layout XML

<ViewGroup> Un contenitore per gli elementi *View*. Può contenere altri *ViewGroup*.

Non bisogna assumere che un *ViewGroup* accetti qualsiasi elemento *View*: difatti, se un oggetto *ViewGroup* implementa una classe *AdapterView*, determinerà i possibili *child* tramite un *Adapter*.

Attributi:

- android:id
- anrdoid:layout_height (**obbligatorio)**
- android:layout_width (**obbligatorio)**
- ...

Esempi di layout:

- LinearLayout
- RelativeLayout
- FrameLayout

<**View**> Rappresenta un singolo componente di un'interfaccia utente, generalmente chiamato ***widget***.

Attributi:

- android:id
- anrdoid:layout_height (**obbligatorio)**
- android:layout_width (**obbligatorio)**
- ...

Esempi di layout:

- TextView
- Button
- CheckBox

<**requestFocus**> Inserito come figlio di un *widget*, fornisce allo stesso il *focus* al caricamento dell'UI. È possibile assegnarlo a un unico elemento per ogni layout.
Questo elemento non prevede attributi.

<**include**> Include un file XML di layout all'interno del layout.

Attributi:

- layout (**obbligatorio**)
- android:id
- anrdoid:layout_height
- anrdoid:layout_width
- ...

È possibile specificare solamente parametri supportati dall'elemento *root* del layout incluso. Tenere presente che gli attributi definiti per il tag *include* vanno a sovrascrivere, se presenti, i corrispondenti attributi dell'elemento *root* del layout che è stato incluso.

ViewStub Un'alternativa più leggera rispetto a *include* è il *ViewStub*. Questo elemento non carica il layout indicato fino a che tale elemento non viene "gonfiato" (*inflated*).

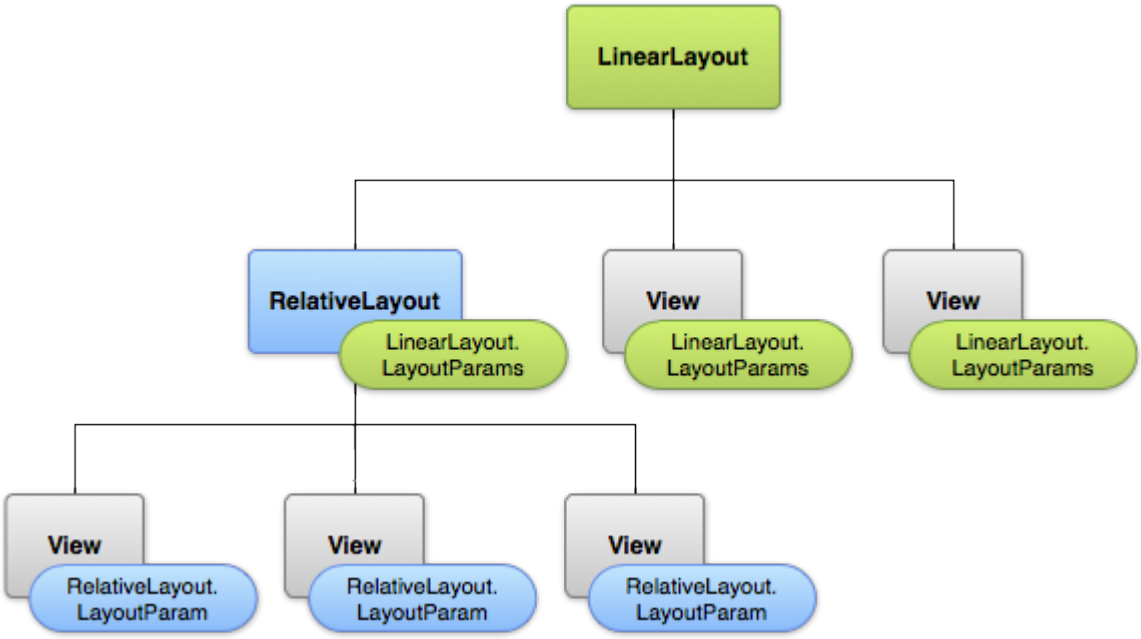
<merge> È un elemento di *root* alternativo, che non è disegnato e non compare nella gerarchia del layout.

Risulta utile quando si vuole definire un layout il cui scopo è di essere incluso in un altro layout, e questo layout da includere non richiede un differente *ViewGroup* rispetto al suo padre nel layout che effettua l'inclusione.

21.3.5 Ereditarietà dei parametri

Ogni classe *ViewGroup* implementa una sottoclasse che estende *ViewGroup.LayoutParams*. Questa sottoclasse contiene delle proprietà che definiscono dimensione e posizione, e che vengono applicate ad ogni sottoelemento del *ViewGroup*, sia che sia una *View* che un altro *ViewGroup*.

Ad esempio:



Un *ViewGroup* figlio può definire dei propri parametri, che sono preponderanti rispetto a quelli ereditati dal padre e verranno applicati anche ai figli.

21.3.6 Caricamento di un XML

Quando si compila l'applicazione, ogni file di layout XML viene compilato in una risorsa di tipo *View*.

Questa risorsa va caricata nell'Activity, all'interno della propria implementazione del metodo *onCreate()*, richiamando il metodo *setContentView(Layout myL)* con il nome del layout nella forma R.layout.file_name).

Ad esempio, dato un layout salvato come *main_layout.xml*, lo caricherò così:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

21.3.7 Posizione del layout

Ogni widget è un rettangolo, che ha come proprietà:

- Posizione relativa:
  - Coordinata **left**
  - Coordinata **top**
- Dimensioni:
  - Larghezza
  - Altezza

Entrambe le proprietà hanno come unità il pixel.

La posizione fa sempre riferimento alla **posizione relativa**, all'interno dell'oggetto padre.

21.3.8 Dimensioni, padding e margini

Ogni *View* può disporre di un padding, ovvero di uno spazio vuoto tra essa ed il padre. Non dispone invece di margini, i quali sono invece supportati dai *ViewGroup*.

La possibile presenza di un padding comporta però che le dimensioni dell'oggetto non coincideranno per forza con la sua dimensione sullo schermo; sono quindi disponibili metodi diversi per ottenere le due misure:

- getMeasuredWidth(), getMeasuredHeight()
- getWidth(), getHeight()

21.3.9 Layout definiti nel codice Java

E' anche possibile implementare i Layout nel codice Java.

In questo caso è buona norma utilizzare LayoutParameters da impostare con *setLayoutParams()* o passandoli al metodo addView.

```
LinearLayout lin = new LinearLayout(this);
lin.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;
lin.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
lin.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(lin);
```

21.3.10 Esempio

Esempio di 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.

21.4 View

Sebbene un layout possa includere diversi *ViewGroup*, principalmente per motivi di prestazioni è bene contenere il numero di layout nidificati.

È buona norma quindi lavorare con una gerarchia "larga" (tanti layout diversi) piuttosto che con una gerarchia "profonda" (più layout annidati).

21.4.1 Adapters

Quando il contenuto di un layout è dinamico o comunque non noto a priori, è possibile usare un layout che estenda *AdapterView* per popolare tale layout a runtime.

Una sottoclasse di *AdapterView* fa uso di un *Adapter* per associare i dati al suo layout. L'Adapter recupera i dati (i.e.: database, array, etc.) e converte ogni elemento in un *widget* che può essere aggiunto al layout.

Esempi di View che estendono AdapterView:

- ListView
- GridView

Caricare i dati in un adapter È possibile caricare i dati i un *AdapterView* come, ad esempio, una *ListView* associando tale *AdapterView* ad un *Adapter*.

Alcuni esempi:

ArrayAdapter

```
// Creo l'adapter:
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
myStringArray);
// Associo la AdapterView all'adapter:
ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

SimpleCursorAdapter

```
// Creo l'adapter:
String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone.NUMBER};
int[] toViews = {R.id.display_name, R.id.phone_number};
// Associo la AdapterView all'adapter:
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this, R.layout.person_name_and_number,
cursor, fromColumns, toViews, 0);
ListView listView = getListView();
listView.setAdapter(adapter);
```

Gestire gli eventi di Click in un Adapter

```
// Creo un oggetto, come una classe anonima, per gestire il click:
private OnItemClickListener mMessageClickedHandler = new OnItemClickListener() {
    public void onItemClick(AdapterView parent, View v, int position, long id) {
        // Do something in response to the click
    }
};
// Associo il click handler alla AdapterView:
listView.setOnItemClickListener(mMessageClickedHandler);
```

21.4.2 Esempio di View (Activity)

Mettendo insieme quanto visto finora (manifesto, string, layout) possiamo scrivere la seguente attività:

Listing 24: Activity Prova

```
1  package unive.dsi.android;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.InputStreamReader;
7  import java.net.HttpURLConnection;
8  import java.net.MalformedURLException;
9  import java.net.URL;
10 import java.net.URLConnection;
11 import java.util.ArrayList;
12 import android.app.Activity;
13 import android.os.Bundle;
14 import android.view.KeyEvent;
15 import android.view.View;
16 import android.widget.*;
17
18 public class Prova extends Activity {
19
20        /** Called when the activity is first created. */
21        @Override
22        public void onCreate(Bundle savedInstanceState) {
23                super.onCreate(savedInstanceState);
24                setContentView(R.layout.main);
25
26                ListView list = (ListView) findViewById(R.id.listView1);
27                final EditText text = (EditText) findViewById(R.id.editText1);
28                final ArrayList<String> strings = new ArrayList<String>();
```

```
29              final ArrayAdapter<String> aa = new ArrayAdapter<String>(this, android.R.layout.
                    simple_list_item_1, strings);
30
31              list.setAdapter(aa);
32
33              text.setOnKeyListener(new View.OnKeyListener() {
34                      public boolean onKey(View v, int keyCode, KeyEvent event) {
35                              if (event.getAction() == KeyEvent.ACTION_DOWN)
36                                      if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER
37                                        || keyCode == KeyEvent.KEYCODE_ENTER) {
38                                              try {
39                                                      URL url = new URL(text.getText().toString());
40                                                      URLConnection connection = url.openConnection()
                                                            ;
41                                                      HttpURLConnection httpConnection = (
                                                            HttpURLConnection) connection;
42                                                      int responseCode = httpConnection.
                                                            getResponseCode();
43
44                                                      if (responseCode == HttpURLConnection.HTTP_OK)
                                                            {
45                                                          InputStream in = httpConnection.
                                                                getInputStream();
46                                                          BufferedReader d = new BufferedReader(
                                                                new InputStreamReader(in));
47                                                          String res = d.readLine();
48                                                          while (res != null) {
49                                                                  strings.add(res);
50                                                                  res = d.readLine();
51                                                          }
52                                                      }
53                                                      else
54                                                              stringa.add("NO HTTP_OK");
55                                              } catch (MalformedURLException e) {
56                                                strings.add(e.getMessage());
57                                              } catch (IOException e) {
58                                                strings.add(e.getMessage());
59                                              }
60
61                                              if (strings.size() > 5)
62                                                      strings.remove(0);
63
64                                              strings.notifyDataSetChanged();
65                                              text.setText("");
66                                              return true;
67                                      }
68                              return false;
69                      }
70              });
71      }
72  }
```

21.4.3 Personalizzazione di componenti

E' possibile personalizzare i componenti :

- estendendo la classe del componente che li definisce (es. TestView)

- raggrupare più controlli in modo che si comportino come un'unico nuovo controllo estendendo una classe dei Layout
- 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;

21.4.4 Esempi di personalizzazione di un componente

Esempio TextView Vediamo un esempio di personalizzazione di un componente **TextView**:

Listing 25: Personalizzazione di 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 View Vediamo un esempio di personalizzazione di un componente **View**:

Listing 26: Personalizzazione di 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;
}
}
```

21.5 Service

21.6 Content Providers

21.7 Event Handlers

21.7.1 Introduzione

Per fare in modo che l'estensione della View sia interattiva essa deve rispondere a tutta una serie di eventi generati dall'utente.

Alcuni metodi legati agli eventi:

- onClick
- onKey
- onFocusChange
- onLongClick

- onTouch
- onCreateContextMenu

Questi metodi ritornano *true* se l'evento è stato gestito dal metodo stesso.

21.7.2 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.

21.8 Intents

21.8.1 Definizione

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.

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:

- Per gli intenti **espliciti** il nome del componente, ovvero la classe (di solito per i componenti interni all'applicazione);
- 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 chiamato.

21.8.2 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 attivazione:

- esplicita
- implicita

a seconda se vogliamo accoppiare o disaccoppiare le due attività.

21.8.3 Intenti impliciti

Ci sono due modalità di esecuzione a seconda che vogliamo o meno esaminare il valore di ritorno dell'attività invocata:

Senza esaminare il valore di ritorno:

```
Intent intent new Intent(Intent.ACTION_DIAL,URI.parse("tel:0412348457"));
startActivity(intent);
```

Ad esempio, per far partire il browser Web preinstallato nel terminale Android, sostituire con:

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(text.getText().toString()); startActivity(intent);
startActivity(intent);
```

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);
```

21.8.4 Intenti esplicti

Ci sono due modalità di esecuzione a seconda che vogliamo o meno esaminare il valore di ritorno dell'attività invocata:

1) Esplicito senza esaminare il valore di ritorno:

```
Intent intent = new Intent(this,OtherActivity.class);
startActivity(intent);
```

3) Esplicito 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.

Ad esempio:

```
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();
```

21.8.5 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.

Vediamo come definire gli intent-filter nel Manifest:

```
<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>
```

Vediamo nel dettaglio i tag utilizzati:

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 specifica 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).

21.8.6 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";
    ...
    void myMethod() {
        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);
    }
}
```

21.8.7 Broadcast receiver

Ovviamente se non ci sono ascoltatori l'evento (Broadcast Event) non genera nessun effetto.

Per definire un BroadcastReceiver bisogna effettuare due operazioni:

1. Estendere la classe BroadcastReceiver.
2. Definire il receiver, in uno dei due modi:
   - Staticamente
   - Dinamicamente

Vediamo un esempio pratico:

1) Estendere la classe BroadcastReceiver Bisogna estendere la classe *BroadcastReceiver* e sovrascrivere il metodo ***onReceive***().

Ad esempio:

```
public class CrashBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        //TODO: React to the Intent received.
    }
}
```

2) [Alternativa statica] Definire il receiver in AndroidManifest Bisogna registrare su Android il BroadcastReceiver.

È necessario aggiungere nel manifesto dell'applicazione che contiene il BroadcastReceiver che c'é un receiver e che eventi gestisce.

Ad esempio:

```
<receiver android:name=".CrashBroadcastReceiver">
    <intent-filter>
        <action android:name="it.unive.dsi.Crash.CRASH_DETECTED"/>
    </intent-filter>
</receiver>
```

2 bis)[Alternativa dinamica] Definire il receiver dinamicamente 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);
```

21.9 Notifications

A notification is a message you can display to the user outside of your application's normal UI. When you tell the system to issue a notification, it first appears as an icon in the notification area. To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.

You specify the UI information and actions for a notification in a NotificationCompat.Builder object. To create the notification itself, you call NotificationCompat.Builder.build(), which returns a Notification object containing your specifications. To issue the notification, you pass the Notification object to the system by calling NotificationManager.notify().

Although they're optional, you should add at least one action to your notification. An action allows users to go directly from the notification to an Activity in your application, where they can look at one or more events or do further work.

A notification can provide multiple actions. You should always define the action that's triggered when the user clicks the notification; usually this action opens an Activity in your application. You can also add buttons to the notification that perform additional actions such as snoozing an alarm or responding immediately to a text message; this feature is available as of Android 4.1. If you use additional action buttons, you must also make their functionality available in an Activity in your app; see the section Handling compatibility for more details.

Inside a Notification, the action itself is defined by a PendingIntent containing an Intent that starts an Activity in your application. To associate the PendingIntent with a gesture, call the appropriate method of NotificationCompat.Builder. For example, if you want to start Activity when the user clicks the notification text in the notification drawer, you add the PendingIntent by calling setContentIntent().

Starting an Activity when the user clicks the notification is the most common action scenario. You can also start an Activity when the user dismisses a notification. In Android 4.1 and later, you can start an Activity from an action button. To learn more, read the reference guide for NotificationCompat.Builder.

22 Geolocalizzazione

22.1 Introduzione

22.1.1 Definizione

La geolocalizzazione, come dice il nome, comprende tutte quelle operazioni legate alla determinazione della posizione del dispositivo.

È buona norma:

- Tracciare la locazione solo quando indispensabile;
- Avvertire l'utente quanto si sta tracciando la posizione, e come tali dati vengono usati e memorizzati;
- Permettere all'utente di disabilitare gli aggiornamenti della posizione.

22.1.2 Tipi di localizzazione

Un dispositivo mobile può essere equipaggiato di più dispositivi di localizzazione.

I più usati sono:

- **GPS**
  - Preciso.
  - Consuma molta energia.
- **Rete (GSM o WiFi)**
  - Poco preciso.
  - Consuma meno energia rispetto a GPS.
  - Può comportare dei costi aggiuntivi.

Per scegliere il dispositivo di localizzazione che si vuole utilizzare, vanno analizzate le esigenze dell'applicazione che si vuole sviluppare (precisione richiesta, etc.) per individuare il giusto compromesso.

22.1.3 Emulatore di posizioni

Per poter testare le proprie applicazioni che implementano i *Location Services*, Android fornisce un *location emulator* (emulatore di posizione), tramite il quale si possono simulare delle posizioni fittizie.

Il *location emulator* altro non fa che inviare degli oggetti *Location* fittizi ai *Location Services*. Questi oggetti vengono definiti *mock locations* (posizioni fittizie).

22.1.4 Permessi

Per poter utilizzare le posizioni del dispositivo è necessario richiedere i permessi nel file *Manifest.xml*.

- **Posizioni precise** [Es.: GPS].
  - <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
- **Posizioni approssimate** [Es.: Reti].
  - <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
- **Posizioni fittizie** [*Mock locations*].
  - <uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
  - Inoltre, va abilitata nel menù *Sviluppatore* del dispositivo l'opzione di debugging: *Allow mock locations*, .

22.2 Location Manager

Prima di poter fare qualsiasi operazioni con i *Location Provider*, è necessario recuperare l'oggetto *Location Manager*.
Vediamo come:

```
String serviceString = Context.LOCATION_SERVICE;
LocationManager locationManager = (LocationManager) getSystemService(serviceString);
```

22.3 Location Provider

Vi sono tre modalità per selezionare il *Location Provider* desiderato. Vediamole:

22.3.1 Selezione diretta

Se il programmatore conosce già il *Location Provider* più adatto, lo può selezionare direttamente.
Ad esempio:

```
LocationManager.GPS_PROVIDER
```

oppure:

```
LocationManager.NETWORK_PROVIDER
```

22.3.2 Selezione tramite elenco

È altrimenti possibile richiedere un elenco dei *Location Provider* disponibili, per poi selezionare quello che si preferisce:

```
List<String> providers = locationManager.getProviders(enabledOnly);
// Poi selezionare il provider, usandone il nome.
```

22.3.3 Selezione tramite scelta automatica

Si può anche richiedere al *Location Manager* di ritornarci il (i) *Location Provider* più adatto/i, in base a dei criteri definiti dallo sviluppatore.

Vediamo come creare un oggetto *Criteria*:

```
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);
```

Possiamo poi richiedere al *Location Manager* di ritornarci il *Provider* che maggiormente risponde alle nostre esigenze, oppure un elenco di tutti quelli che corrispondono a tali parametri.

Per richiedere l'elenco di tutti i *Provider* corrispondenti ai criteri impostati:

```
List<String> matchingProviders = locationManager.getProviders(criteria, false);
locationManager.getLastKnownLocation(provider);
```

Per ottenere il Provider che maggiormente combacia:

```
String bestProvider = locationManager.getBestProvider(criteria, true);
```

Nel caso in cui nessun Provider combaci con i criteri impostati, la ricerca continua automaticamente, "ammorbidendo" i parametri nel seguente ordine:

1. Consumo di energia;
2. Precisione della posizione;
3. Possibilità di ritornare: orientamento, velocità e altitudine.

22.4 LocationListener

Per avere un'Activity che aggiorna la posizione visualizzata quando il device cambia posizione, è necessario usare un *LocationListener* (che va poi registrato su un dispositivo di posizionamento).

LocationListener è un'interfaccia:

```
public interface android.location.LocationListener
```

tale interfaccia è così definita:

```
abstract void onLocationChanged(Location location);
abstract void onProviderDisabled(String provider);
abstract void onProviderEnabled(String provider);
abstract void onStatusChanged(String provider, int status, Bundle extras);
```

onLocationChanged() Invocato quando la posizione è cambiata.

onProviderDisabled() Invocato quando l'utente disabilita il provider.

È il caso che il provider venga disabilitato quando un'applicazione non ne ha bisogno. Ad esempio, quando un'applicazione che mostra la posizione corrente viene mandata in background.

onProviderEnabled() Invocato quando l'utente abilita il provider.

***onStatusChanged*()** Questo metodo viene richiamato quando:

- il provider non è in grado di recuperare la posizione;
- il provider è tornato disponibile dopo un periodo di indisponibilità.

22.5 Ottenere la posizione

22.5.1 getLastKnownLocation()

Quando si richiede al *Location Provider* la posizione, in realtà non gli si richiede quella corrente, bensì l'ultima posizione rilevata.

Questa può essere estremamente recente, ma anche sensibilmente datata. Difatti, ottenere l'ultima posizione rilevata e richiedere l'aggiornamento della posizione corrente sono due operazioni distinte.

Per ottenere l'ultima posizione che è stata rilevata si richiama il metodo:

```
locationManager.getLastKnownLocation(provider);
```

22.5.2 Esempio

Vediamo ora un'esempio di Activity che ottiene **l'ultima posizione rilevata** e ne mostra i vari campi che la compongono:

Listing 27: R.layout.main.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:id="@+id/myLocationText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
</LinearLayout>
```

Listing 28: MyGeoApp

```
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 MyGeoApp extends Activity {

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                String context = Context.LOCATION_SERVICE;
                LocationManager 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 = (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
```

```
38
39                      if (location.hasBearing())
40                              desc += "Dir:␣"+location.getBearing();   //in gradi (Nord=0)
41
42                      if (location.hasSpeed())
43                              desc += "Vel:␣"+location.getSpeed();   //in metri al secondo
44
45                      Bundle b = location.getExtras();
46                      if (b!=null)
47                              for (String key:b.keySet())
48                                      desc += key +":␣"+ b.get(key);   // solo satellites
49              } else {
50                      desc = "No␣location␣found";
51              }
52
53              myLocationText.setText(desc);
54      }
55 }
```

22.6 Aggiornare la posizione

22.6.1 requestLocationUpdates()

Una volta definita un'implementazione di *LocationListener*, che definisce il comportamento dell'Activity a fronte dei cambiamenti di stato del provider, è necessario associarla ad un provider.

Quest'operazione viene effettuata tramite il metodo *requestLocationUpdates*() del *LocationManager*:

locationManager.requestLocationUpdates(provider, minUpdateTime, minUpdateDistance, myLocationListener);

Come si può notare, tale metodo ha quattro parametri:

Provider il nome del *Location Provider*.

minUpdateTime il tempo minimo che deve trascorrere tra due aggiornamenti.

minUpdateDistance la distanza minima che deve essere stata coperta tra due aggiornamenti.

myLocationListener l'oggetto Listener che si vuole utilizzare.

22.6.2 removeUpdates()

Così come è possibile registrare un *LocationListener* presso un *Location Provider*, è anche possibile de-registrarlo.

Questo è possibile tramite il metodo *removeUpdates(LocationListener loclis)* del *Location Manager*:

locationManager.removeUpdates(myLocationListener);

22.6.3 Esempio

```
// Definisco i parametri
String provider = LocationManager.GPS_PROVIDER;
int minUpdateTime = 5000; // ms
int minUpdateDistance = 5; // metri
// Creo un oggetto LocationListener
LocationListener myLocationListener = new LocationListener() {
    // Questo è il metodo che verrà invocato
    // quando è disponibile una nuova posizione:
    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){
                ...
        }
};
    // Associo l'oggetto LocationListener al provider scelto,
    // utilizzando i parametri di aggiornamento desiderati
    locationManager.requestLocationUpdates(provider, minUpdateTime, minUpdateDistance, myLocationListener);
```

22.7 Proximity Alerts

22.7.1 Introduzione

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 esce 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 trova alla locazione da monitorare: in questo modo e' possibile risparmiare energia.

Per impostare un *ProximiTy Alert* è necessario indicare:

- Un punto (latitudine, longitudine) che sarà il centro del cerchio.
- Il raggio del cerchio.
- Una scadenza temporale per l'alert.

22.7.2 Gestione degli eventi

Quando si attiva un *proximity alert*, questo fa scattare degli Intenti e molto comunemente dei Broadcast Intent.

Per specificare l'intento da far 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);
```

22.7.3 Esempio

Impostiamo ora un proximity alert che non scade mai e che viene fatto scattare quando il device si allontana piu' di 10 metri dall'obiettivo.

Vediamo come:

```
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);
```

23 Geocoding

23.1 Introduzione

23.1.1 Definizione

Geocoding permette di effettuare conversioni tra indirizzi (via Rossi 1, Milano) e coordinate (latitudine, longitudine).

Le classe *Geocoder* fornisce le seguenti conversioni:

getFromLocationName() dato un indirizzo, ne ottiene le coordinate.

getFromLocation() data una coppia di coordinate, ne ottiene l'indirizzo più vicino.

23.1.2 Requisiti

La classe *Geocoder* effettua le conversioni tramite l'interrogazione di un server remoto che contiene la posizione degli indirizzi.

Sarà quindi necessario che nel *Manifest* compaia il permesso per accedere a Internet:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

23.2 Conversioni

23.2.1 Forward geocoding

Il *forward geocoding* è anche detto semplicemente *geocoding* - permette di convertire un indirizzo in una posizione.

Vediamo un esempio:

```
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 più precisa, è possibile specificare il rettangolo (*boundaries*) delimitatore della zona in cui vogliamo ricercare l'indirizzo. Questa cosa può risultare particolamente utile se vogliamo restringere la ricerca alla zona visualizzata dall'utente:

```
List<Address> locations = null;
try {
    locations = fwdGeocoder.getFromLocationName(streetAddress, 10, n, e, s, w);
} catch (IOException e) {}
```

23.2.2 Reverse geocoding

Il *reverse geocoding* permette di convertire una posizione (una coppia di coordinate) in un indirizzo.

Vediamo un esempio:

```
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.

23.3 Dettagli implementativi

23.3.1 Locale

Il costruttore di *Geocoder* prevede il passaggio del *Locale* del client. Questo è importante in quanto gli indirizzi dipendono fortemente dal paese e dalla lingua dell'utente:

```
Geocoder geocoder = new Geocoder(getApplicationContext(), Locale.getDefault());
```

23.3.2 Tipo ritornato

Entrambi i metodi ritornano una lista di oggetti di tipo *Address*.
In fase di chiamata si specifica il numero massimo di risultati che si vuole vengano restituiti.

```
List<Address> results = gc.getFromLocation(latitude, longitude, maxResults);
```

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) {
        latLongString = "No location found";
  } else {
        double lat = location.getLatitude();
        double lng = location.getLongitude();
        latLongString = "Lat:" + lat + "\nLong:" + lng;
        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) {}
  }
    myLocationText.setText("Your Current Position is:\n" + latLongString + "\n" + addres-
  sString);
}
```

23.3.3 Comportamento dei metodi

Entrambi i metodi sono asincroni, quindi bloccanti: l'esecuzione del codice non prosegue fino a che non restituiscono il risultato.

È quindi consigliabile provvedere a implementare le chiamate di tali metodi in thread secondari, al fine di non bloccare il thread principale dell'applicazione.

24 SQLite

24.1 Breve panoramica

Android fornisce supporto nativo solamente per un motore di database, che è *SQLite*.

Il motore di database non è vero e proprio *database server*; è bensì implementato tramite una libreria scritta in C che gira nello stesso processo dell'applicazione.

Se si volesse utilizzare un diverso motore di database, sarebbe necessario effettuare una delle seguenti operazioni:

- Implementare una propria API per accedere al database voluto.
- Utilizzare una libreria di terze parti che fornisca un'API per il database desiderato.

Da notare che è necessario che queste API siano state sviluppate per Android; difatti, ad esempio, JDBC non è disponibile per sistemi Android.

24.2 Query

Una query al DB viene eseguita utilizzando il metodo *query*. Tale metodo restituisce un oggetto di tipo *Cursor* (un cursore).

Gli argomenti del metodo 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.

24.3 Esempi

24.3.1 Creazione di un database con una relazione

```
SQLiteDatabase db;
db = openOrCreateDatabase("DB", Context.MODE_PRIVATE, null);
String sqlStr = "create table PROVA ( id integer primary key autoincrement, Nome text not null);";
db.execSQL(sqlStr);
```

24.3.2 Esempio completo

```
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 Content
    Values newValues = new ContentValues();
    // Assegniamo 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=?", new String[]{“alessandro”});
    db.close();
```

24.3.3 Esempio con uso di un SQL Helper

Definiamo una tabella:

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // give it an empty constructor.
    public FeedReaderContract() {}
    /* Inner class that defines the table contents */
    public static abstract class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_ENTRY_ID = "entryid";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle"; ...
  }
}
```

Aggiungiamo un metodo per creare la tabella:

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " +
    FeedEntry.TABLE_NAME + " (" + FeedEntry._ID + " INTEGER PRIMARY KEY,"
  +
    FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP + ...
    // Any other options for the CREATE command
    " )";
```

Aggiungiamo un metodo per cancellare la tabella:

```
private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Implementiamo un *SQL Helper* che faccia uso dell'oggetto creato:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version. public
  static final int
    DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";
    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
  }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
  }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
  }
}
```

Istanziamo la nostra implementazione del *SQL Helper*:

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Esempio di inserimento di dati:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();
// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);
// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert( FeedEntry.TABLE_NAME, FeedEntry.COLUMN_NAME_NULLABLE,
values);
```

Esempio di lettura di dati:

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();
// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = { FeedEntry._ID, FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ... };
// How you want the results sorted in the resulting Cursor
String sortOrder = FeedEntry.COLUMN_NAME_UPDATED + " DESC";
Cursor c = db.query(
```

```
        FeedEntry.TABLE_NAME, // The table to query
        projection, // The columns to return
        selection, // The columns for the WHERE clause
        selectionArgs, // The values for the WHERE clause
        null, // don't group the rows
        null, // don't filter by row groups
        sortOrder // The sort order );
// Per navigare i risultati, posso usare i metodi forniti dal cursore. Ad esempio:
    cursor.moveToFirst();
    long itemId = cursor.getLong ( cursor.getColumnIndexOrThrow(FeedEntry._ID) );
```

Esempio di cancellazione di dati:

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

Esempio di aggiornamento di dati:

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();
// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };
// Perform query
int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

Riferimenti bibliografici

[1] http://www.w3schools.com/

[2] http://www.dsi.unive.it/~taw/

[3] http://docs.oracle.com/javaee/7/tutorial/doc/home.htm

[4] http://docs.oracle.com/javaee/5/tutorial/doc/

[5] http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

[6] http://www.html.it/guide/guida-jsp/

[7] http://www.tutorialspoint.com/jsp/

[8] http://www.w3.org/CGI/

[9] http://www.w3.org/CGI/

[10] http://www.webopedia.com/TERM/C/CGI.html

[11] http://docs.oracle.com/javase/tutorial/deployment/applet/

[12] http://www.tutorialspoint.com/java/java_applet_basics.htm

[13] http://www.w3.org/Protocols/Specs.html

[14] http://www.ietf.org/rfc/rfc2616.txt

[15] http://www.w3schools.com/js/js_cookies.asp

[16] http://docs.oracle.com/javase/tutorial/rmi/

[17] http://www.giuseppesicari.it/articoli/java-standard-edition-remote-method-invocation/