JAVA mattone dopo mattone

DRAFT ZERO

Massimiliano Tarquini - Alessandro Ligi

www.java-net.it

Draft zero
Rilasciato il 02/05/2001

DRAFT ZERO

Titolo Originale
Java Enterprise Computing

Copyrigth per la prima edizione
© Massimiliano Tarquini

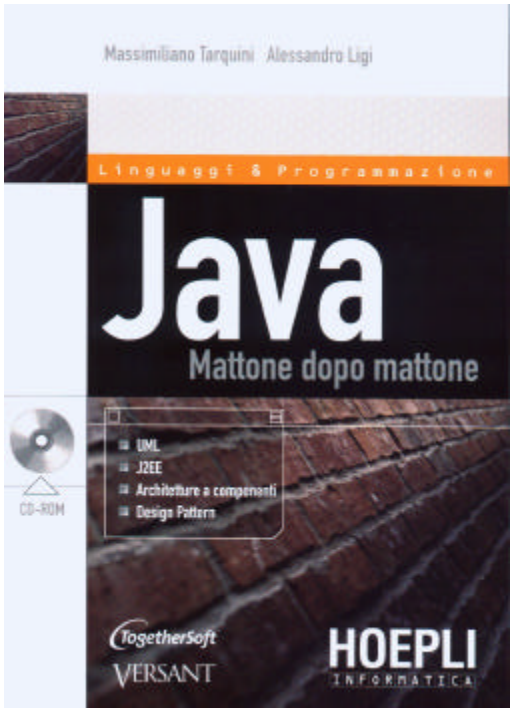
Copyrigth per l'edizione corrente
© Massimiliano Tarquini – Alessandro Ligi

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

“Se in un primo momento l'idea non è assurda, allora non c'è nessuna speranza che si realizzi”.

Albert Einstein

Il libro “Java Mattone dopo Mattone” è stato pubblicato dalla casa editrice Hoepli e presentato come novità editoriale dello SMAU 2002.



Tale edizione cartacea, di cui presentiamo la copertina, prende spunto dagli elementi trattati in questa versione elettronica che ne rappresenta il draft zero. Questo draft, scritto “a braccio” da Massimiliano Tarquini e nato come diario di lavoro e di ausilio in attività formative, è divenuto, col tempo, un vero caso:

<<oltre 200.000 copie scaricate>>

Questo successo ha condotto alla pubblicazione del libro, da parte di Hoepli, nell’ottobre 2002.

Abbiamo pensato di lasciare intatto e disponibile su Internet questo draft zero così come è stato creato, pur consapevoli delle imprecisioni e degli errori presenti, e di correggere, approfondire, dettagliare, ampliare ed esemplificare i temi trattati solo nel libro presente in libreria

In particolare, il libro contiene approfondimenti relativi a:

- linguaggio UML, che diventa lo strumenti principale di rappresentazione e modellazione concettuale di tutti i concetti espressi nel libro
- aspetti legati allo sviluppo in architetture a componenti
- gli strumenti e le tecnologie Java per lo sviluppo di applicazioni Enterprise (EJB, RMI, JSP, Servlet)
- introduzione ai Web services
- i Design Patterns ed il loro sviluppo in Java

Il libro è, inoltre, corredato di un CD-ROM ideato per agevolare l’apprendimento pratico del linguaggio, calandosi in una realtà esemplificativa tutta italiana (e quindi di più facile compresione ed ausilio) e contenente anche strumenti di produttività utili nel processo di progettazione e sviluppo.

Desideriamo ringraziare tutti coloro che hanno agevolato la diffusione su Internet di questo testo e anche tutti coloro che lo hanno scaricato: è anche grazie a loro che è stata possibile la pubblicazione cartacea.

Indice Analitico

JAVA MATTONE DOPO MATTONE 14

INTRODUZIONE 14
PREMESSE 14

CAPITOLO 1 17

INTRODUZIONE ALLA PROGRAMMAZIONE OBJECT ORIENTED 17
INTRODUZIONE 17
UNA EVOLUZIONE NECESSARIA 18
IL PARADIGMA PROCEDURALE 18
CORREGGERE GLI ERRORI PROCEDURALI 19
IL PARADIGMA OBJECT ORIENTED 21
CLASSI DI OGGETTI 21
EREDITARIETÀ 21
IL CONCETTO DI EREDITARIETÀ NELLA PROGRAMMAZIONE 22
VANTAGGI NELL'USO DELL'EREDITARIETÀ 23
PROGRAMMAZIONE OBJECT ORIENTED ED INCAPSULAMENTO 23
I VANTAGGI DELL'INCAPSULAMENTO 24
ALCUNE BUONE REGOLE PER CREARE OGGETTI 25

CAPITOLO 2 26

INTRODUZIONE AL LINGUAGGIO JAVA 26
INTRODUZIONE 26
LA STORIA DI JAVA 26
LE CARATTERISTICHE PRINCIPALI DI JAVA – INDIPENDENZA DALLA PIATTAFORMA 26
LE CARATTERISTICHE PRINCIPALI DI JAVA – USO DELLA MEMORIA E MULTI-THREADING 28
MECCANISMO DI CARICAMENTO DELLE CLASSI DA PARTE DELLA JVM 29
IL JAVA DEVELOPMENT KIT (JDK) 29
SCARICARE ED INSTALLARE IL JDK 30
IL COMPILATORE JAVA (JAVAC) 31
IL DEBUGGER JAVA (JDB) 31
L'INTERPRETE JAVA 31

CAPITOLO 3 33

INTRODUZIONE ALLA SINTASSI DI JAVA 33
INTRODUZIONE 33
VARIABILI 33
INIZIALIZZAZIONE DI UNA VARIABILE 34
VARIABILI FINAL 34
OPERATORI 34
OPERATORI ARITMETICI 36

OPERATORI RELAZIONALI 37
OPERATORI CONDIZIONALI 38
OPERATORI LOGICI E DI SHIFT BIT A BIT 38
OPERATORI DI ASSEGNAMENTO 40
ESPRESSIONI 41
ISTRUZIONI 41
REGOLE SINTATTICHE DI JAVA 41
BLOCCHI DI ISTRUZIONI 42
METODI 42
DEFINIRE UNA CLASSE 43
VARIABILI REFERENCE 44
VISIBILITÀ DI UNA VARIABILE JAVA 46
L' OGGETTO NULL 47
CREARE ISTANZE 47
L'OPERATORE PUNTO "." 48
AUTO REFERENZA ED AUTO REFERENZA ESPLICITA 49
AUTO REFERENZA IMPLICITA 49
STRINGHE 51
STATO DI UN OGGETTO JAVA 51
COMPARAZIONE DI OGGETTI 51
METODI STATICI 52
IL METODO MAIN 53
L'OGGETTO SYSTEM 53

LABORATORIO 3 55

INTRODUZIONE ALLA SINTASSI DI JAVA 55
DESCRIZIONE 55
ESERCIZIO 1 55
SOLUZIONE AL PRIMO ESERCIZIO 57

CAPITOLO 4 59

CONTROLLO DI FLUSSO E DISTRIBUZIONE DI OGGETTI 59
INTRODUZIONE 59
ISTRUZIONI PER IL CONTROLLO DI FLUSSO 59
L'ISTRUZIONE IF 60
L'ISTRUZIONE IF-ELSE 60
ISTRUZIONI IF, IF-ELSE ANNIDATE 61
CATENE IF-ELSE-IF 61
L'ISTRUZIONE SWITCH 62
L'ISTRUZIONE WHILE 64
L'ISTRUZIONE DO-WHILE 65
L'ISTRUZIONE FOR 65
ISTRUZIONE FOR NEI DETTAGLI 66
ISTRUZIONI DI RAMIFICAZIONE 66
L'ISTRUZIONE BREAK 67
L'ISTRUZIONE CONTINUE 67
L'ISTRUZIONE RETURN 68

PACKAGE JAVA 68
ASSEGNAMENTO DI NOMI A PACKAGE 68
CREAZIONE DEI PACKAGE SU DISCO 69
IL MODIFICATORE PUBLIC 70
L'ISTRUZIONE IMPORT 71

LABORATORIO 4 72

CONTROLLO DI FLUSSO E DISTRIBUZIONE DI OGGETTI 72
ESERCIZIO 1 72
ESERCIZIO 2 72
SOLUZIONE AL PRIMO ESERCIZIO 73
SOLUZIONE AL SECONDO ESERCIZIO 75

CAPITOLO 5 78

INCAPSULAMENTO 78
INTRODUZIONE 78
MODIFICATORI PUBLIC E PRIVATE 79
PRIVATE 79
PUBLIC 79
IL MODIFICATORE PROTECTED 80
UN ESEMPIO DI INCAPSULAMENTO 81
L'OPERATORE NEW 81
COSTRUTTORI 82
UN ESEMPIO DI COSTRUTTORI 83
OVERLOADING DEI COSTRUTTORI 84
RESTRIZIONE SULLA CHIAMATA AI COSTRUTTORI 85
CROSS CALLING TRA COSTRUTTORI 86

LABORATORIO 5 88

INCAPSULAMENTO DI OGGETTI 88
ESERCIZIO 1 88
SOLUZIONE DEL PRIMO ESERCIZIO 89

CAPITOLO 6 91

EREDITARIETÀ 91
INTRODUZIONE 91
DISEGNARE UNA CLASSE BASE 91
OVERLOAD DI METODI 94
ESTENDERE UNA CLASSE BASE 95
EREDITARIETÀ ED INCAPSULAMENTO 96
CONSEGUENZE DELL'INCAPSULAMENTO NELLA EREDITARIETÀ 97
EREDITARIETÀ E COSTRUTTORI 98
AGGIUNGERE NUOVI METODI 100
OVERRIDING DI METODI 101

CHIAMARE METODI DELLA CLASSE BASE 102
FLESSIBILITÀ DELLE VARIABILI REFERENCE 103
RUN-TIME E COMPILE-TIME 104
ACCESSO A METODI ATTRAVERSO VARIABILI REFERENCE 105
CAST DEI TIPI 105
L'OPERATORE INSTANCEOF 105
L'OGGETTO OBJECT 106
IL METODO EQUALS() 106
RILASCIARE RISORSE ESTERNE 108
RENDERE GLI OGGETTI IN FORMA DI STRINGA 108

LABORATORIO 6 109

INTRODUZIONE ALLA EREDITARIETÀ 109
ESERCIZIO 1 109
ESERCIZIO 2 110
SOLUZIONE AL PRIMO ESERCIZIO 111
SOLUZIONE AL SECONDO ESERCIZIO 112

CAPITOLO 7 114

ECCEZIONI 114
INTRODUZIONE 114
PROPAGAZIONE DI OGGETTI 116
OGGETTI THROWABLE 118
NULLPOINTEREXCEPTION 120
DEFINIRE ECCEZIONI PERSONALIZZATE 121
L'ISTRUZIONE THROW 121
LA CLAUSOLA THROWS 122
ISTRUZIONI TRY / CATCH 124
SINGOLI CATCH PER ECCEZIONI MULTIPLE 125
LE ALTRE ISTRUZIONI GUARDIANE. FINALLY 126

CAPITOLO 8 128

POLIMORFISMO ED EREDITARIETÀ AVANZATA 128
INTRODUZIONE 128
POLIMORFISMO : "UN'INTERFACCIA, MOLTI METODI" 128
INTERFACCE 129
DEFINIZIONE DI UNA INTERFACCIA 129
IMPLEMENTARE UNA INTERFACCIA 129
EREDITARIETÀ MULTIPLA IN JAVA 130
CLASSI ASTRATTE 131

CAPITOLO 9 133

JAVA THREADS 133
INTRODUZIONE 133

THREAD DI SISTEMA 133
LA CLASSE JAVA.LANG.THREAD 134
INTERFACCIA "RUNNABLE" 135
SINCRONIZZARE THREAD 136
LOCK 137
SINCRONIZZAZIONE DI METODI STATICI 138
BLOCCHI SINCRONIZZATI 139

LABORATORIO 9 140

JAVA THREAD 140
ESERCIZIO 1 140
ESERCIZIO 2 140
SOLUZIONE AL PRIMO ESERCIZIO 141
SOLUZIONE AL SECONDO ESERCIZIO 141

CAPITOLO 11 144

JAVA NETWORKING 144
INTRODUZIONE 144
I PROTOCOLLI DI RETE (INTERNET) 144
INDIRIZZI IP 145
COMUNICAZIONE "CONNECTION ORIENTED" O "CONNECTIONLESS" 147
DOMAIN NAME SYSTEM : RISOLUZIONE DEI NOMI DI UN HOST 148
URL 150
TRASMISSION CONTROL PROTOCOL : TRASMISSIONE CONNECTION ORIENTED 150
USER DATAGRAM PROTOCOL : TRASMISSIONE CONNECTIONLESS 152
IDENTIFICAZIONE DI UN PROCESSO : PORTE E SOCKET 152
IL PACKAGE JAVA.NET 154
UN ESEMPIO COMPLETO DI APPLICAZIONE CLIENT/SERVER 155
LA CLASSE SERVERSOCKET 157
LA CLASSE SOCKET 157
UN SEMPLICE THREAD DI SERVIZIO 158
TCP SERVER 158
IL CLIENT 160

CAPITOLO 12 162

JAVA ENTERPRISE COMPUTING 162
INTRODUZIONE 162
ARCHITETTURA DI J2EE 163
J2EE APPLICATION MODEL 165
CLIENT TIER 166
WEB TIER 167
BUSINESS TIER 168
EIS-TIER 170
LE API DI J2EE 171
JDBC : JAVA DATABASE CONNECTIVITY 171

RMI : REMOTE METHOD INVOCATION 173
JAVA IDL 174
JNDI 174
JMS 175

CAPITOLO 13 177

ARCHITETTURA DEL WEB TIER 177
INTRODUZIONE 177
L'ARCHITETTURA DEL "WEB TIER" 177
INVIARE DATI 179
SVILUPPARE APPLICAZIONI WEB 180
COMMON GATEWAY INTERFACE 180
ISAPI ED NSAPI 181
ASP – ACTIVE SERVER PAGES 181
JAVA SERVLET E JAVASERVER PAGES 182

CAPITOLO 14 183

JAVA SERVLET API 183
INTRODUZIONE 183
IL PACKAGE JAVAX.SERVLET 183
IL PACKAGE JAVAX.SERVLET.HTTP 184
CICLO DI VITA DI UNA SERVLET 185
SERVLET E MULTITHREADING 186
L'INTERFACCIA SINGLETHREADMODEL 187
UN PRIMO ESEMPIO DI CLASSE SERVLET 187
IL METODO SERVICE() 188

CAPITOLO 15 190

SERVLET HTTP 190
INTRODUZIONE 190
IL PROTOCOLLO HTTP 1.1 190
RICHIESTA HTTP 191
RISPOSTA HTTP 193
ENTITÀ 194
I METODI DI REQUEST 195
INIZIALIZZAZIONE DI UNA SERVLET 195
L'OGGETTO HTTPSERVLETRESPONSE 196
I METODI SPECIALIZZATI DI HTTPSERVLETRESPONSE 198
NOTIFICARE ERRORI UTILIZZANDO JAVA SERVLET 199
L'OGGETTO HTTPSERVLETREQUEST 199
INVIARE DATI MEDIANTE LA QUERY STRING 201
QUERY STRING E FORM HTML 202
I LIMITI DEL PROTOCOLLO HTTP : COOKIES 204
MANIPOLARE COOKIES CON LE SERVLET 204
UN ESEMPIO COMPLETO 205

SESSIONI UTENTE 206
SESSIONI DAL PUNTO DI VISTA DI UNA SERVLET 206
LA CLASSE HTTPSESSION 207
UN ESEMPIO DI GESTIONE DI UNA SESSIONE UTENTE 208
DURATA DI UNA SESSIONE UTENTE 209
URL REWRITING 209

CAPITOLO 16 211

JAVASERVER PAGES 211
INTRODUZIONE 211
JAVASERVER PAGES 211
COMPILAZIONE DI UNA PAGINA JSP 213
SCRIVERE PAGINE JSP 213
INVOCARE UNA PAGINA JSP DA UNA SERVLET 214

CAPITOLO 17 216

JAVASERVER PAGES – NOZIONI AVANZATE 216
INTRODUZIONE 216
DIRETTIVE 216
DICHIARAZIONI 217
SCRIPTLETS 217
OGGETTI IMPLICITI : REQUEST 218
OGGETTI IMPLICITI : RESPONSE 218
OGGETTI IMPLICITI : SESSION 219

CAPITOLO 18 220

JDBC 220
INTRODUZIONE 220
ARCHITETTURA DI JDBC 220
DRIVER DI TIPO 1 221
DRIVER DI TIPO 2 222
DRIVER DI TIPO 3 222
DRIVER DI TIPO 4 224
UNA PRIMA APPLICAZIONE DI ESEMPIO 224
RICHIEDERE UNA CONNESSIONE AD UN DATABASE 226
ESEGUIRE QUERY SUL DATABASE 227
L'OGGETTO RESULTSET 227

APPENDICE A 231

JAVA TIME-LINE 231
1995-1996 231
1997 231
1998 232
1999 233

2000 234

APPENDICE B **235**

GLOSSARIO DEI TERMINI **235**

BIBLIOGRAFIA **239**

Premesse

La guerra dei desktop è ormai persa, ma con Java 2 Enterprise Edition la Sun Microsystem ha trasformato un linguaggio in una piattaforma di sviluppo integrata diventata ormai standard nel mondo del "Server Side Computing".

Per anni, il mondo della IT ha continuato a spendere soldi ed energie in soluzioni proprietarie tra loro disomogenee dovendo spesso reinvestire in infrastrutture tecnologiche per adattarsi alle necessità emergenti di mercato.

Nella ultima decade di questo secolo con la introduzione di tecnologie legate ad Internet e più in generale alle reti, l'industria del software ha immesso sul mercato circa 30 application server, ognuno con un modello di programmazione specifico.

Con la nascita di nuovi modelli di business legati al fenomeno della new-economy, la divergenza di tali tecnologie è diventata in breve tempo un fattore destabilizzante ed il maggior ostacolo alla innovazione tecnologica. Capacità di risposta in tempi brevi e produttività con lunghi margini temporali sono diventate oggi le chiavi del successo di applicazioni enterprise.

Con la introduzione della piattaforma J2EE, la Sun ha proposto non più una soluzione proprietaria, ma una architettura basata su tecnologie aperte e portabili proponendo un modello in grado di accelerare il processo di implementazione di soluzioni "server-side" attraverso lo sviluppo di funzionalità nella forma di "Enterprise Java Beans" in grado di girare su qualsiasi application server compatibile con lo standard.

Oltre a garantire tutte le caratteristiche di portabilità (Write Once Run Everywhere) del linguaggio Java, J2EE fornisce:

Un modello di sviluppo semplificato per l' "enterprise computing" - La piattaforma offre ai "vendors" di sistemi la capacità di fornire una soluzione che lega insieme molti tipi di middleware in un unico ambiente, riducendo tempi di sviluppo e costi necessari alla integrazioni di componenti software di varia natura. J2EE permette di creare ambienti "server side" contenenti tutti i "middletiers" di tipo server come connettività verso database, ambienti transazionale, servizi di naming ecc.;

Una architettura altamente scalabile - La piattaforma fornisce la scalabilità necessaria allo sviluppo di soluzione in ambienti dove le applicazioni scalano da prototipo di lavoro ad architetture "24x7 enterprise wide";

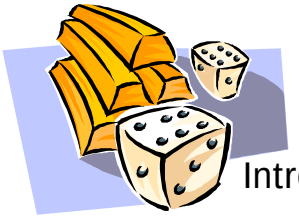
Legacy Connectivity – La piattaforma consente l'integrabilità di soluzioni pre esistenti in ambienti legacy consentendo di non reinvestire in nuove soluzioni;

Piattaforme Aperte – J2EE è uno standard aperto. La Sun in collaborazione con partner tecnologici garantisce ambienti aperti per specifiche e portabilità;

Sicurezza – La piattaforma fornisce un modello di sicurezza in grado di proteggere dati in applicazioni Internet;

Alla luce di queste considerazioni si può affermare che la soluzione offerta da Sun abbia traghettato l' "enterprise computing" in una nuova era in cui le applicazioni usufruiranno sempre più di tutti i vantaggi offerti da uno standard aperto.

Parte Prima
La programmazione
Object Oriented



Capitolo 1
Introduzione alla programmazione Object Oriented

Introduzione

Questo capitolo è dedicato al "paradigma Object Oriented" e cerca di fornire ai neofiti della programmazione in Java i concetti base necessari allo sviluppo di applicazioni Object Oriented.

In realtà le problematiche che andremo ad affrontare nei prossimi paragrafi sono estremamente complesse e trattate un gran numero di testi che non fanno alcuna menzione a linguaggi di programmazione, quindi limiterò la discussione soltanto ai concetti più importanti. Procedendo nella comprensione del nuovo modello di programmazione risulterà chiara l'evoluzione che, a partire dall'approccio orientato a procedure e funzioni e quindi alla programmazione dal punto di vista del calcolatore, porta oggi ad un modello di analisi che, partendo dal punto di vista dell'utente suddivide l'applicazione in concetti rendendo il codice più comprensibile e semplice da mantenere.

Il modello classico conosciuto come "paradigma procedurale", può essere riassunto in due parole: "Divide et Impera" ossia dividi e conquista. Difatti secondo il paradigma procedurale, un problema complesso viene suddiviso in problemi più semplici in modo che siano facilmente risolvibili mediante programmi "procedurali". E' chiaro che in questo caso, l'attenzione del programmatore è accentrata al problema.

A differenza del primo, il paradigma Object Oriented accentra l'attenzione verso dati. L'applicazione viene suddivisa in un insieme di oggetti in grado di interagire tra di loro e codificati in modo tale che la macchina sia in gradi di comprenderli.

Il primo cambiamento evidente è quindi a livello di disegno della applicazione. Di fatto l'approccio Object Oriented non è limitato a linguaggi come Java o C++ . Molte applicazione basate su questo modello sono state scritte con linguaggi tipo C o Assembler. Un linguaggio Object Oriented semplifica il meccanismo di creazione degli Oggetti allo stesso modo con cui un linguaggio procedurale semplifica la decomposizione in funzioni.

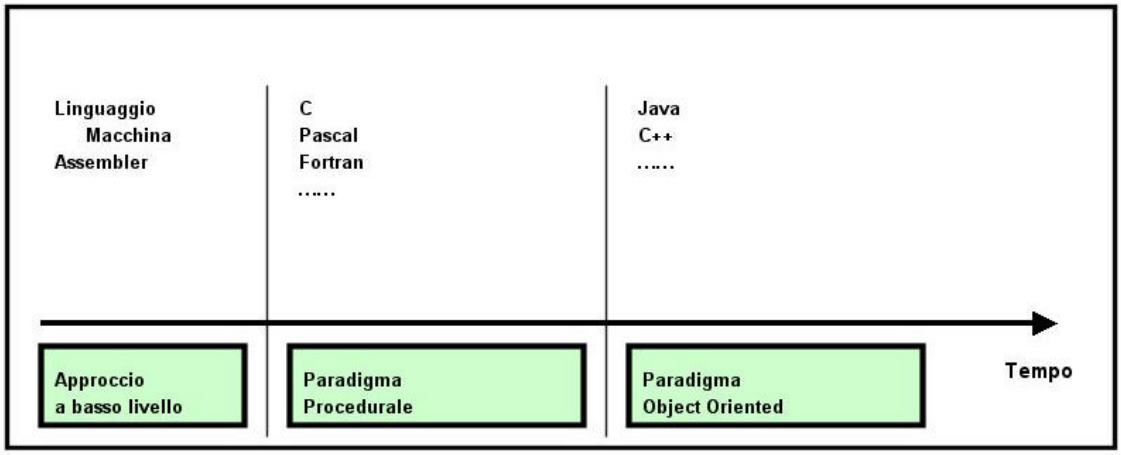


Figura 1-1 : *Evoluzione del modello di programmazione*

Una evoluzione necessaria

Quando i programmi erano scritti in assembler, ogni dato era globale e le funzioni andavano disegnate a basso livello. Con l'avvento dei linguaggi procedurali come il linguaggio C, i programmi sono diventati più robusti e semplici da mantenere inquanto il linguaggio forniva regole sintattiche e semantiche che supportate da un compilatore consentivano un maggior livello di astrazione rispetto a quello fornito dall'assembler fornendo un ottimo supporto alla decomposizione procedurale della applicazione.

Con l'aumento delle prestazione dei calcolatori e di conseguenza con l'aumento della complessità delle applicazioni, l'approccio procedurale ha iniziato a mostrare i propri limiti rendendo necessario definire un nuovo modello e nuovi linguaggi di programmazione. Questa evoluzione è stata schematizzata nella *figura 1-1*.

I linguaggi come Java e C++ forniscono il supporto ideale al disegno ad oggetti di applicazioni fornendo un insieme di regole sintattiche e semantiche che aiutano nello sviluppo di oggetti.

Il paradigma procedurale

Secondo il paradigma procedurale il programmatore analizza il problema ponendosi dal punto di vista del computer che solamente istruzioni semplici e, di conseguenza adotta un approccio di tipo divide et impera[1]. Il programmatore sa perfettamente che una applicazione per quanto complessa può essere suddivisa in step di piccola entità. Questo approccio è stato formalizzato in molti modi ed è ben supportato da molti linguaggi che forniscono al programmatore un ambiente in cui siano facilmente definibili procedure e funzioni.

Le procedure sono blocchi di codice riutilizzabile che possiedono un proprio insieme di dati e realizzano specifiche funzioni. Le funzioni una volta scritte possono essere richiamate ripetutamente in un programma, possono ricevere parametri che modificano il loro stato e possono tornare valori al codice chiamante. Una volta scritte, le procedure possono essere legate assieme a formare un applicazione.

All'interno della applicazione è quindi necessario che i dati vengano condivisi tra loro. Questo meccanismo si risolve mediante l'uso di variabili globali, passaggio di parametri e ritorno di valori.

Una applicazione procedurale tipica ed il suo diagramma di flusso è riassunta nella *Figura 1-2.* L'uso di variabili globali genera però problemi di protezione dei dati quando le procedure si richiamano tra di loro. Per esempio nella applicazione mostrata nella *Figura 1-2*, la procedura "output" esegue una chiamata a basso livello verso il terminale e può essere chiamata soltanto dalla procedura "print" la quale a sua volta modifica dati globali.

Dal momento che le procedure non sono "auto-documentanti" (self-documenting) ossia non rappresentano entità ben definite, un programmatore dovendo modificare la applicazione e non conoscendone a fondo il codice, potrebbe utilizzare la routine "Output" senza chiamare la procedura "Print" dimenticando quindi l'aggiornamento dei dati globali a carico di "Print" e producendo di conseguenza effetti indesiderati (side-effects) difficilmente gestibili.

[1] Divide et Impera ossia dividi e conquista era la tecnica utilizzata dagli antichi romani che sul campo di battaglia dividevano le truppe avversarie per poi batterle con pochi sforzi.

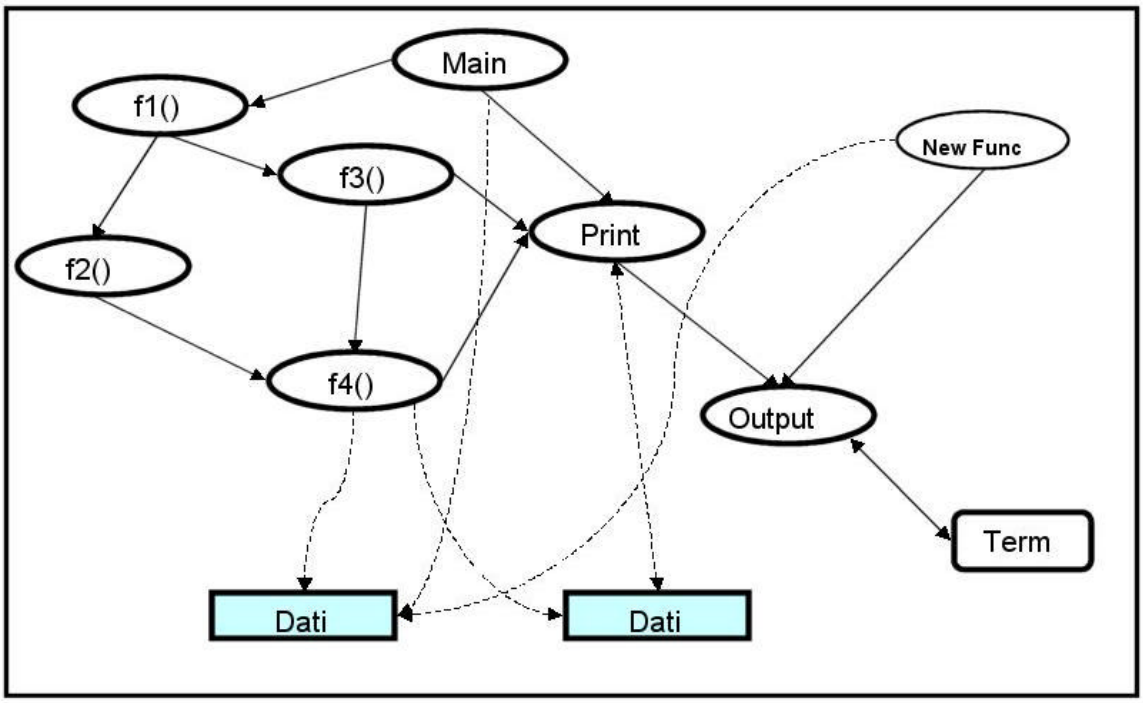


Figura 1-2 : *Diagramma di una applicazione procedurale*

Per questo motivo, le applicazioni basate sul modello procedurale sono difficili da aggiornare e controllare con meccanismi di debug. I bug derivanti da side-effects possono presentarsi in qualunque punto del codice causando una propagazione incontrollata dell'errore. Ad esempio riprendendo ancora la nostra applicazione, una gestione errata dei dati globali dovuta ad una mancata chiamata a "Print" potrebbe avere effetto su "f4()" che a sua volta propagherebbe l'errore ad "f2()" ed "f3()" fino al "main" del programma causando la terminazione anomala del processo.

Correggere gli errori procedurali

Per risolvere i problemi presentati nel paragrafo precedente i programmatori hanno fatto sempre più uso di tecniche mirate a proteggere dati globali o funzioni "nascondendone" il codice. Un modo sicuramente spartano, ma spesso utilizzato, consisteva nel nascondere il codice di routine sensibili ("Output" nel nostro esempio) all'interno di librerie contando sul fatto che la mancanza di documentazione scoraggiasse un nuovo programmatore ad utilizzare impropriamente queste funzioni.

Il linguaggio C fornisce strumenti mirati alla circoscrizione del problema come il modificatore "static" con il fine di delimitare sezioni di codice di una applicazione in grado di accedere a dati globali, eseguire funzioni di basso livello o, evitare direttamente l'uso di variabili globali.

Quando si applica il modificatore "static" ad una variabile locale, viene allocata per la variabile della memoria permanente in modo molto simile a quanto avviene per le variabili globali. Questo meccanismo consente alla variabile dichiarata static di mantenere il proprio valore tra due chiamate successive ad una funzione. A differenza di una variabile locale non statica il cui ciclo di vita (di conseguenza il valore) è limitato al tempo necessario per la esecuzione della funzione, il valore di una variabile dichiarata static non andrà perduto tra chiamate successive.

La differenza sostanziale tra una variabile globale ed una variabile locale static è che la seconda è nota solamente al blocco in cui è dichiarata ossia è una variabile globale con scopo limitato, vengono inizializzate solo una volta all'avvio del

programma e non ogni volta che si effettui una chiamata alla funzione in cui sono definite.

Supponiamo ad esempio di voler scrivere che calcoli la somma di numeri interi passati ad uno ad uno per parametro. Grazie all'uso di variabili static sarà possibile risolvere il problema nel modo seguente :

```
int _sum (int i)
{
    static int sum=0;
    sum=sum+I;
    return sum;
}
```

Usando una variabile static, la funzione è in grado di mantenere il valore della variabile tra chiamate successive evitando l'uso di variabili globali.

Il modificatore static può essere utilizzato anche con variabili globali. Difatti, se applicato ad un dato globale indica al compilatore che la variabile creata dovrà essere nota solamente alle funzioni dichiarate nello stesso file contenente la dichiarazione della variabile. Stesso risultato lo otterremmo applicando il modificatore ad una funzione o procedura.

Questo meccanismo consente di suddividere applicazioni procedurali in moduli. Un modulo è un insieme di dati e procedure logicamente correlate tra di loro in cui le parti sensibili possono essere isolate in modo da poter essere chiamate solo da determinati blocchi di codice. Il processo di limitazione dell'accesso a dati o funzioni è conosciuto come "incapsulamento".

Un esempio tipico di applicazione suddivisa in moduli è schematizzato nella *figura 1-3* nella quale è rappresentata una nuova versione del modello precedentemente proposto. Il modulo di I/O mette a disposizione degli altri moduli la funzione "Print" incapsulando la routine "Output" ed i dati sensibili.

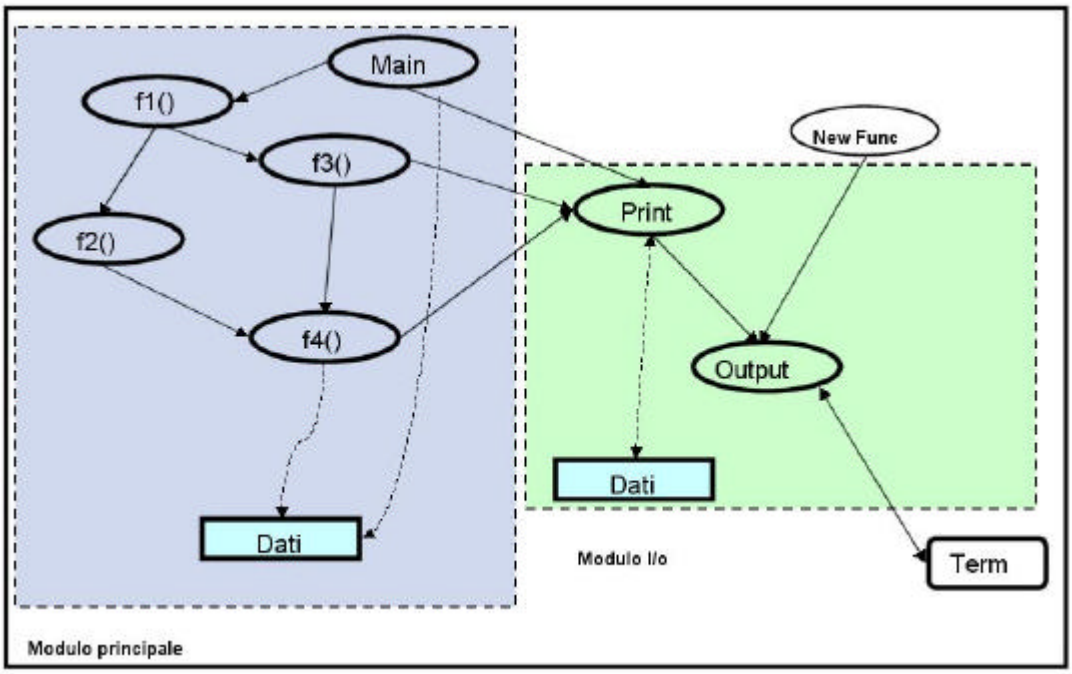


Figura 1-3 : *Diagramma di una applicazione procedurale suddiviso per moduli*

Questa evoluzione del modello fornisce numerosi vantaggi; i dati ora non sono completamente globali e risultano quindi più protetti che nel modello precedente, limitando di conseguenza i danni causati da propagazioni anomale dei bug. Inoltre il

numero limitato di procedure “pubbliche” viene in aiuto ad un programmatore che inizi a studiare il codice della applicazione. Questo nuovo modello si avvicina molto al modello proposto dall’approccio Object Oriented.

Il paradigma Object Oriented

Il paradigma Object Oriented formalizza la tecnica vista in precedenza di incapsulare e raggruppare parti di un programma. In generale, il programmatore divide le applicazioni in gruppi logici che rappresentano concetti sia a livello di utente che a livello applicativo. I pezzi che vengono poi riuniti a formare una applicazione.

Scendendo nei dettagli, il programmatore ora inizia con l’analizzare tutti i singoli aspetti concettuali che compongono un programma. Questi concetti sono chiamati oggetti ed hanno nomi legati a ciò che rappresentano. Una volta che gli oggetti sono identificati, il programmatore decide di quali attributi (dati) e funzionalità (metodi) dotare le entità. L’analisi infine dovrà includere le modalità di interazione tra gli oggetti. Proprio grazie a queste interazioni sarà possibile riunire gli oggetti a formare un applicazione.

A differenza di procedure e funzioni, gli oggetti sono “auto-documentanti” (self-documenting). Una applicazione può essere scritta a partire da poche informazioni ed in particolar modo il funzionamento interno delle funzionalità di ogni oggetto è completamente nascosto al programmatore (Incapsulamento Object Oriented).

Classi di Oggetti

Concentriamoci per qualche istante su alcuni concetti tralasciando l’aspetto tecnico del paradigma object oriented e proviamo per un istante a pensare ad un libro. Quando pensiamo ad un libro pensiamo subito ad una classe di oggetti aventi caratteristiche comuni: tutti i libri contengono delle pagine, ogni pagina contiene del testo e le note sono scritte a fondo pagina. Altra cosa che ci viene subito in mente riguarda le azioni che tipicamente compiamo quando utilizziamo un libro: voltare pagina, leggere il testo, guardare le figure etc.

E’ interessante notare che utilizziamo il termine “libro” per generalizzare un concetto relativo a qualcosa che contiene pagine da sfogliare, da leggere o da strappare ossia ci riferiamo ad un insieme di oggetti con attributi comuni, ma comunque composto da entità aventi ognuna caratteristiche proprie che rendono ognuna differente rispetto all’altra.

Pensiamo ora ad un libro scritto in francese. Ovviamente sarà comprensibile soltanto a persone in grado di comprendere questa lingua; d’altro canto possiamo comunque guardarne i contenuti (anche se privi di senso), sfogliarne le pagine o scriverci dentro. Questo insieme generico di proprietà rende un libro utilizzabile da chiunque a prescindere dalle caratteristiche specifiche (nel nostro caso la lingua).

Possiamo quindi affermare che un libro è un oggetto che contiene pagine e contenuti da guardare e viceversa ogni oggetto contenente pagine e contenuti da guardare può essere classificato come un libro.

Abbiamo quindi definito una categoria di oggetti che chiameremo classe e che, nel nostro caso, fornisce la descrizione generale del concetto di libro. Ogni nuovo libro con caratteristiche proprie apparterrà comunque a questa classe base.

Ereditarietà

Con la definizione di una classe, nel paragrafo precedente abbiamo stabilito che un libro contiene pagine che possono essere girate, scarabocchiate, strappate etc.

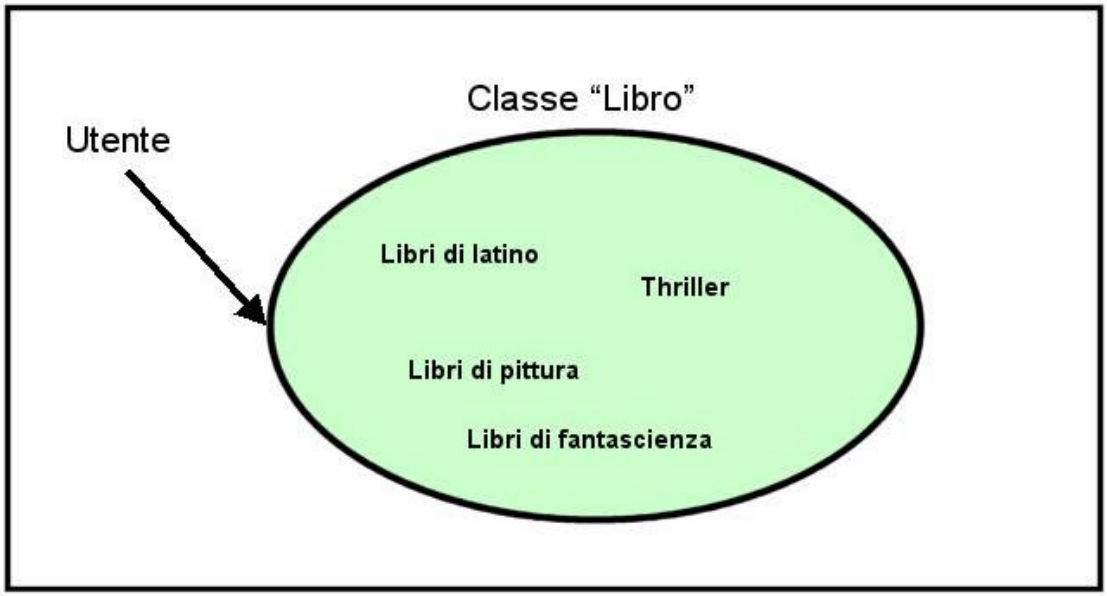


Figura 1-4 : ***Diagramma di una applicazione procedurale suddiviso per moduli***

Stabilita la classe base, possiamo creare tanti libri purché aderiscano alle regole definite (*Figura 1-4*). Il vantaggio maggiore nell'aver stabilito questa classificazione è che ogni persona deve conoscere solo le regole base per essere in grado di poter utilizzare qualsiasi libro. Una volta assimilato il concetto di pagina che può essere sfogliata, si è in grado di utilizzare qualsiasi entità classificabile come libro.

Il concetto di ereditarietà nella programmazione

Se estendiamo i concetti illustrati alla programmazione iniziamo ad intravederne i reali vantaggi. Una volta stabilite le categorie di base, possiamo utilizzarle per creare tipi specifici di oggetti ereditando e specializzando le regole base.

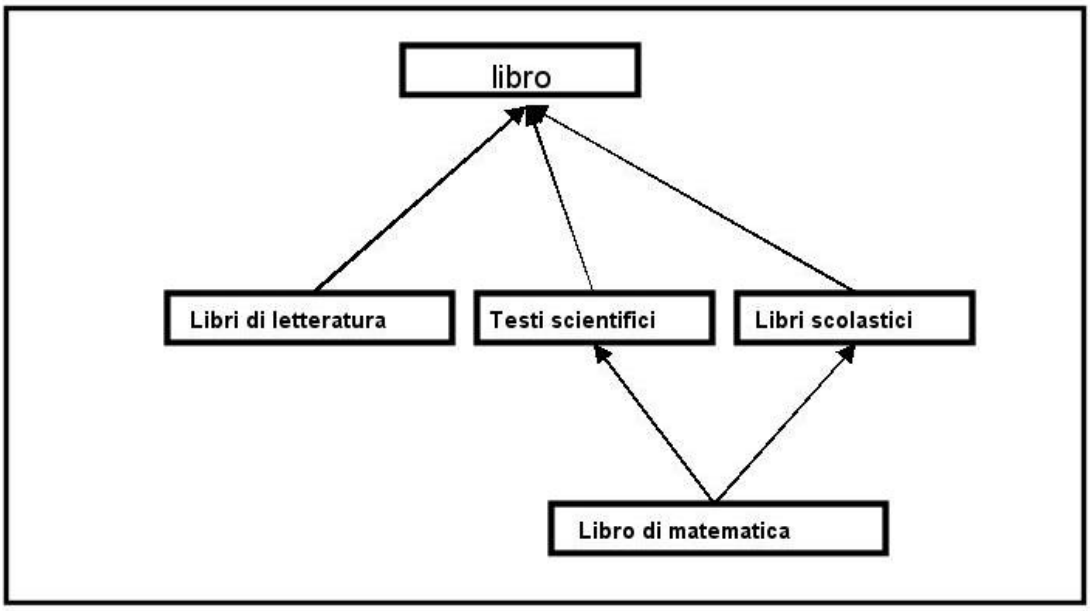


Figura 1-5 : *Diagramma di ereditarietà*

Per definire questo tipo di relazioni viene utilizzata una forma a diagramma in cui la classe generica è riportata come nodo sorgente di un grafo orientato i cui sotto nodi rappresentano categorie più specifiche e gli archi che uniscono i nodi sono orientati da specifico a generale (*Figura 1-5*).

Un linguaggio orientato ad oggetti fornisce al programmatore strumenti per rappresentare queste relazioni. Una volta definite classi e relazioni, sarà possibile mediante il linguaggio implementare applicazioni in termini di classi generiche; questo significa che una applicazione sarà in grado di utilizzare ogni oggetto specifico senza essere necessariamente riscritta, ma limitando le modifiche alle funzionalità fornite dall'oggetto per manipolare le sue proprietà.

Vantaggi nell'uso dell'ereditarietà

Come è facile intravedere, l'organizzazione degli oggetti fornita dal meccanismo di ereditarietà rende semplici le operazioni di manutenzione di una applicazione. Ogni volta che si renda necessaria una modifica, è in genere sufficiente creare un nuovo oggetto all'interno di una classe di oggetti ed utilizzarlo per rimpiazzare uno vecchio ed obsoleto.

Un altro vantaggio della ereditarietà è la re-utilizzabilità del codice. Creare una classe di oggetti per definire entità è molto di più che crearne una semplice rappresentazione: per la maggior parte delle classi l'implementazione è spesso scritta all'interno della descrizione.

In Java ad esempio ogni volta che definiamo un concetto, esso viene definito come una classe all'interno della quale viene scritto il codice necessario ad implementare le funzionalità dell'oggetto per quanto generico esso sia. Se viene creato un nuovo oggetto (e quindi una nuova classe) a partire da un oggetto (classe) esistente si dice che la nuova classe "deriva" dalla originale.

Quando questo accade, tutte le caratteristiche dell'oggetto principale diventano parte della nuova classe. Dal momento che la classe derivata eredita le funzionalità della classe predecessore, l'ammontare del codice da necessario per la nuova classe è pesantemente ridotto: il codice della classe di origine è stato riutilizzato.

A questo punto è necessario iniziare a definire formalmente alcuni termini. La relazione di ereditarietà tra classi è espressa in termini di "superclasse" e "sottoclasse". Una superclasse è la classe più generica utilizzata come punto di partenza per derivare nuove classi. Una sottoclasse rappresenta invece una specializzazione di una superclasse.

E' uso comune chiamare una superclasse "classe base"e una sottoclasse "classe derivata". Questi termini sono comunque relativi in quanto una classe derivata può a sua volta essere una classe base per una classe più specifica.

Programmazione object oriented ed incapsulamento

Come già ampiamente discusso, nella programmazione orientata ad oggetti definiamo oggetti creando rappresentazioni di entità o nozioni da utilizzare come parte di un'applicazione. Per assicurarci che il programma lavori correttamente ogni oggetto deve rappresentare in modo corretto il concetto di cui è modello senza che l'utente possa disgregarne l'integrità.

Per fare questo è importante che l'oggetto esponga solo la porzione di codice e dati che il programma deve utilizzare. Ogni altro dato e codice deve essere nascosto affinché sia possibile mantenere l'oggetto in uno stato consistente. Ad esempio se un

oggetto rappresenta uno stack[2] di dati (*figura 1-6*), l'applicazione dovrà poter accedere solo al primo dato dello stack ossia alle funzioni di Push e Pop.

Il contenitore ed ogni altra funzionalità necessaria alla sua gestione dovrà essere protetta rispetto alla applicazione garantendo così che l'unico errore in cui si può incorrere è quello di inserire un oggetto sbagliato in testa allo stack o estrapolare più dati del necessario. In qualunque caso l'applicazione non sarà mai in grado di creare inconsistenze nello stato del contenitore.

L'incapsulamento inoltre localizza tutti i possibili problemi in porzioni ristrette di codice. Una applicazione potrebbe inserire dati sbagliati nello Stack, ma saremo comunque sicuri che l'errore è localizzato all'esterno dell'oggetto.

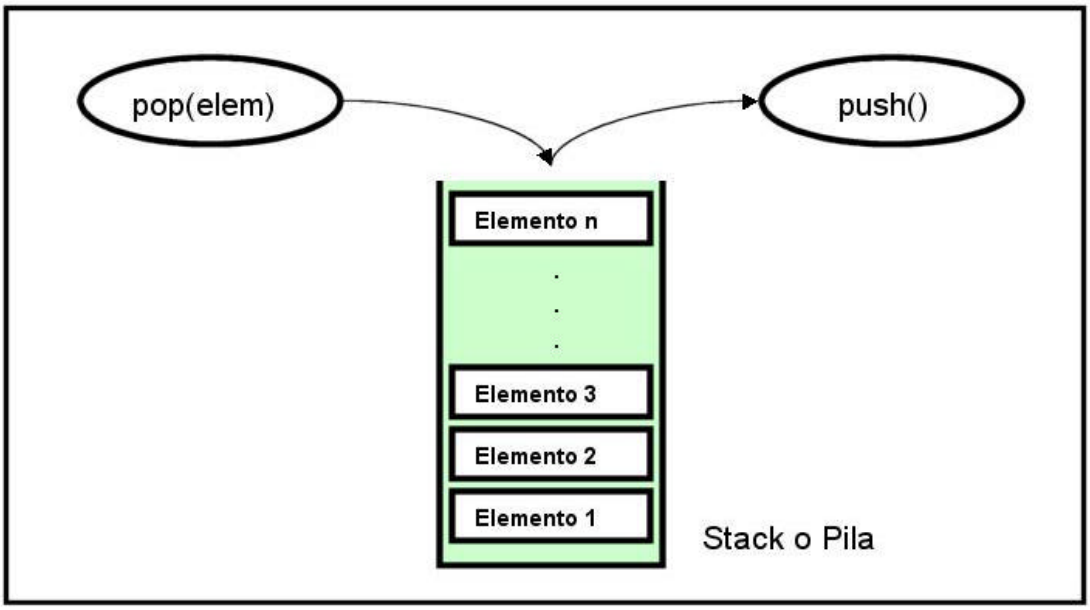


Figura 1-6 : *Stack di dati*

I vantaggi dell'incapsulamento

Una volta che un oggetto è stato incapsulato e testato, tutto il codice ed i dati associati sono protetti. Modifiche successive al programma non potranno causare rotture nelle le dipendenze tra gli oggetti inquanto non saranno in grado di vedere i legami tra dati ed entità. L'effetto principale sulla applicazione sarà quindi quello di localizzare i bugs evitando la propagazione di errori, dotando la applicazione di grande stabilità.

In un programma decomposto per funzioni, le procedure tendono ad essere interdipendenti. Ogni modifica al programma richiede spesso la modifica di funzioni condivise cosa che può propagare un errore alle componenti del programma che le utilizzano.

In un programma object oriented, le dipendenze sono sempre strettamente sotto controllo e sono mascherate all'interno delle entità concettuali. Modifiche a programmi di questo tipo riguardano tipicamente la aggiunta di nuovi oggetti, ed il meccanismo di ereditarietà ha l'effetto di preservare l'integrità referenziale delle entità componenti la applicazione.

Se invece fosse necessario modificare internamente un oggetto, le modifiche sarebbero comunque limitate al corpo dell'entità e quindi confinate all'interno dell'oggetto che impedirà la propagazione di errori all'esterno del codice.

[2] Uno Stack di dati è una struttura a pila all'interno della quale è possibile inserire dati solo sulla cima ed estrarre solo l'ultimo dato inserito.

Anche le operazioni di ottimizzazione risultano semplificate. Quando un oggetto risulta avere performance molto basse, si può cambiare facilmente la sua struttura interna senza dovere riscrivere il resto del programma, purché le modifiche non tocchino le proprietà già definite dell'oggetto.

Alcune buone regole per creare oggetti

Un oggetto deve rappresentare un singolo concetto ben definito.

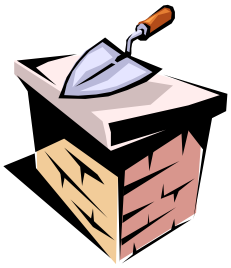
Rappresentare piccoli concetti con oggetti ben definiti aiuta ad evitare confusione inutile all'interno della applicazione. Il meccanismo dell'ereditarietà rappresenta uno strumento potente per creare concetti più complessi a partire da concetti semplici.

Un oggetto deve rimanere in uno stato consistente per tutto il tempo che viene utilizzato, dalla sua creazione alla sua distruzione.

Qualunque linguaggio di programmazione venga utilizzato, bisogna sempre mascherare l'implementazione di un oggetto al resto della applicazione. L'incapsulamento è una ottima tecnica per evitare effetti indesiderati e spesso incontrollabili.

Fare attenzione nell'utilizzo della ereditarietà.

Esistono delle circostanze in cui la convenienza sintattica della ereditarietà porta ad un uso inappropriato della tecnica. Per esempio una lampadina può essere accesa o spenta. Usando il meccanismo della ereditarietà sarebbe possibile estendere queste sue proprietà ad un gran numero di concetti come un televisore, un fon etc.. Il modello che ne deriverebbe sarebbe inconsistente e confuso.



Capitolo 2
Introduzione al linguaggio Java

Introduzione

Java è un linguaggio di programmazione object oriented indipendente dalla piattaforma, modellato a partire dai linguaggi C e C++ di cui mantiene caratteristiche. L'indipendenza dalla piattaforma è ottenuta grazie all'uso di uno strato software chiamato **Java Virtual Machine** che traduce le istruzioni dei codici binari indipendenti dalla piattaforma generati dal compilatore java, in istruzioni eseguibili dalla macchina locale (*Figura 2-1*).

La natura di linguaggio a oggetti di Java consente di sviluppare applicazioni utilizzando oggetti concettuali piuttosto che procedure e funzioni. La sintassi object oriented di Java supporta la creazione di oggetti concettuali, il che consente al programmatore di scrivere codice stabile e riutilizzabile utilizzando il paradigma object oriented secondo il quale il programma viene scomposto in concetti piuttosto che funzioni o procedure.

La sua stretta parentela con il linguaggio C a livello sintattico fa si che un programmatore che abbia già fatto esperienza con linguaggi come C, C++, Perl sia facilitato nell'apprendimento del linguaggio.

In fase di sviluppo, lo strato che rappresenta la virtual machine può essere creato mediante il comando **java** anche se molti ambienti sono in grado di fornire questo tipo di supporto. Esistono inoltre compilatori specializzati o **jit** (Just In Time) in grado di generare codice eseguibile dipendente dalla piattaforma.

Infine Java contiene alcune caratteristiche che lo rendono particolarmente adatto alla programmazione di applicazioni web (client-side e server-side).

La storia di Java

Durante l'aprile del 1991, un gruppo di impiegati della SUN Microsystem, conosciuti come "Green Group" iniziarono a studiare la possibilità di creare una tecnologia in grado di integrare le allora attuali conoscenze nel campo del software con l'elettronica di consumo.

Avendo subito focalizzato il problema sulla necessità di avere un linguaggio indipendente dalla piattaforma (il software non doveva essere legato ad un particolare processore) il gruppo iniziò i lavori nel tentativo di creare un linguaggio che estendesse il C++. La prima versione del linguaggio fu chiamata Oak e, successivamente per motivi di royalty Java.

Attraverso una serie di eventi, quella che era la direzione originale del progetto subì vari cambiamenti ed il target fu spostato dall'elettronica di consumo al world wide web. Il 23 Maggio del 1995 la SUN ha annunciato formalmente Java. Da quel momento in poi il linguaggio è stato adottato da tutti i maggiori "vendors" di software incluse IBM, Hewlett Packard e Microsoft.

In appendice è riportata la "Time line" della storia del linguaggio a partire dal maggio 1995. I dati sono stati recuperati dal sito della SUN Microsystem all'indirizzo http://java.sun.com/features/2000/06/time-line.html.

Le caratteristiche principali di Java – Indipendenza dalla piattaforma

Le istruzioni binarie di Java indipendenti dalla piattaforma sono più comunemente conosciuto come Bytecodes. Il Bytecodes di java è prodotto dal compilatore e

necessita di uno strato di software, la Java Virtual Machine (che per semplicità indicheremo con JVM), per essere eseguito (*Figura 2-1*).

La JVM è un programma scritto mediante un qualunque linguaggio di programmazione dipendente dalla piattaforma, e traduce le istruzioni Java, nella forma di Bytecodes, in istruzioni native del processore locale.

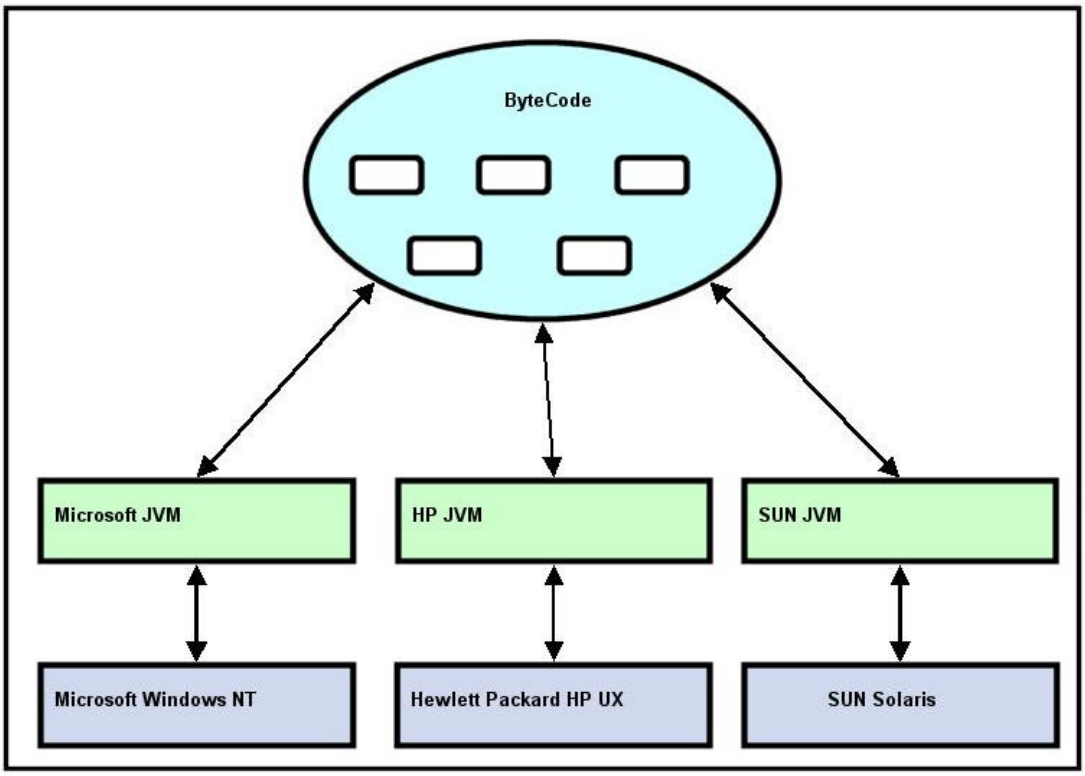


Figura 2-1: *architettura di una applicazione Java*

Non essendo il Bytecodes legato ad una particolare architettura Hardware, questo fa si che per trasferire una applicazione Java da una piattaforma ad un'altra è necessario solamente che la nuova piattaforma sia dotata di una apposita JVM. In presenza di un interprete una applicazione Java potrà essere eseguita su qualunque piattaforma senza necessità di essere ricompilata.

In alternativa, si possono utilizzare strumenti come i "Just In Time Compilers", compilatori in grado di tradurre il Bytecodes in un formato eseguibile su una specifica piattaforma al momento della esecuzione del programma Java. I vantaggi nell'uso dei compilatori JIT sono molteplici. La tecnologia JIT traduce il bytecodes in un formato eseguibile al momento del caricamento della applicazione.

Ciò consente di migliorare le performance della applicazione che non dovrà più passare per la virtual machine, e allo stesso tempo preserva la caratteristica di portabilità del codice. L'unico svantaggio nell'uso di un JIT sta nella perdita di prestazioni al momento del lancio della applicazione che deve essere prima compilata e poi eseguita.

Un ultimo aspetto interessante della tecnologia Java è quello legato agli sviluppi che la tecnologia sta avendo. Negli ultimi anni molti produttori di hardware anno iniziato a rilasciare processori in grado di eseguire direttamente il Bytecode di Java a livello di istruzioni macchina senza l'uso di una virtual machine.

Le caratteristiche principali di Java – Uso della memoria e multi-threading

Un problema scottante quando si parla di programmazione è la gestione l'uso della memoria. Uno dei problemi più complessi da affrontare quando si progetta una applicazione di fatto proprio quello legato al mantenimento degli spazi di indirizzamento del programma, con il risultato che spesso è necessario sviluppare complesse routine specializzate nella gestione e tracciamento della memoria assegnata alla applicazione.

Java risolve il problema alla radice sollevando direttamente il programmatore dall'onere della gestione della memoria grazie ad un meccanismo detto "**Garbage Collector**". Il Garbage Collector, tiene traccia degli oggetti utilizzati da una applicazione Java, nonché delle referenze a tali oggetti. Ogni volta che un oggetto non è più referenziato per tutta la durata di una specifica slide di tempo, viene rimosso dalla memoria e la risorsa liberata viene di nuovo messa a disposizione della applicazione che potrà continuare a farne uso. Questo meccanismo è in grado di funzionare correttamente in quasi tutti i casi anche se molto complessi, ma non si può dire che è completamente esente da problemi.

Esistono infatti dei casi documentati di fronte ai quali il Garbage Collector non è in grado di intervenire. Un caso tipico è quello della "referenza circolare" in cui un oggetto A referenzia un oggetto B e viceversa, ma la applicazione non sta utilizzando nessuno dei due come schematizzato nella *figura 2-2*.

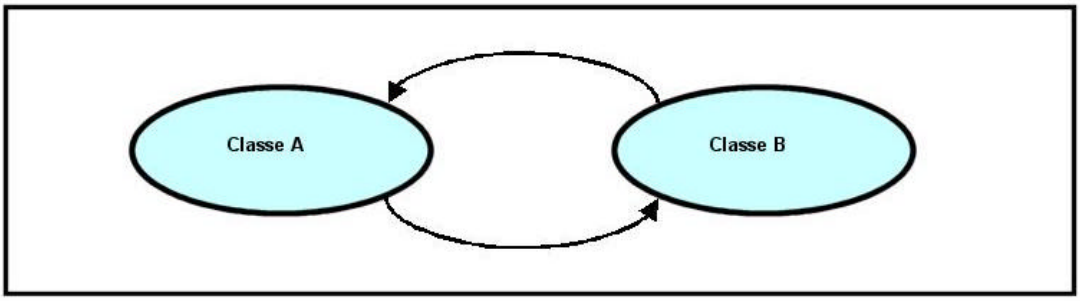


Figura 2-2: *riferimento circolare*

Java è un linguaggio multi-threaded. Il multi-threading consente ad applicazioni Java di sfruttare il meccanismo di concorrenza logica. Parti separate di un programma possono essere eseguite come se fossero (dal punto di vista del programmatore) processate parallelamente. L'uso di thread rappresenta un modo semplice di gestire la concorrenza tra processi inquanto gli spazi di indirizzamento della memoria della applicazione sono condivisi con i thread eliminando così la necessità di sviluppare complesse procedure di comunicazione tra processi.

Infine Java supporta il metodo detto di "Dynamic Loading and Linking". Secondo questo modello, ogni modulo del programma (classe) è memorizzato in un determinato file. Quando un programma Java viene eseguito, le classi vengono caricate e stanziate solo al momento del loro effettivo utilizzo. Una applicazione composta da molte classi caricherà solamente quelle porzioni di codice che debbono essere eseguite in un determinato istante.

Java prevede un gran numero di classi pre-compilate che forniscono molte funzionalità come strumenti di i/o o di networking.

Meccanismo di caricamento delle classi da parte della JVM

Quando la JVM viene avviata, il primo passo che deve compiere è quello di caricare le classi necessarie all'avvio della applicazione. Questa fase avviene secondo uno schema temporale ben preciso e schematizzato nella *figura 2-3*, ed è a carico di un modulo interno chiamato "*launcher*".

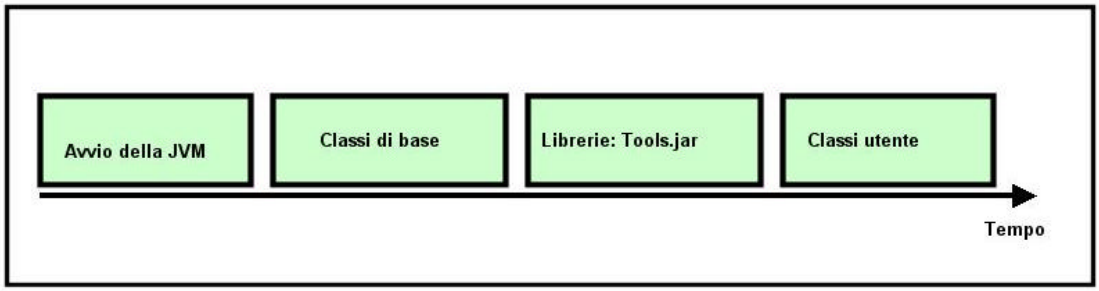


Figura 2-3: *schema temporale del caricamento delle classi della JVM*

Le prime ad essere caricate sono le classi di base necessarie alla piattaforma Java per fornire lo strato di supporto alla applicazione a cui fornirà l'ambiente di run-time. Il secondo passo è caricare le classi Java appartenenti alle librerie di oggetti messi a disposizione dalla SUN ed utilizzate all'interno della applicazione. Infine vengono caricate le classi componenti l'applicazione e definite dal programmatore.

Per consentire al launcher di trovare le librerie e le classi utente, è necessario specificare esplicitamente la loro locazione su disco. Per far questo è necessario definire una variabile di ambiente chiamata CLASSPATH che viene letta sia in fase di compilazione, sia in fase di esecuzione della applicazione.

La variabile di ambiente CLASSPATH contiene una serie di stringhe suddivise dal carattere ";" e rappresentanti ognuna il nome completo di una archivio[3] di classi (files con estensione .jar e .zip) o di una directory contenete file con estensione .class . Ad esempio, se la classe *MiaClasse.java* è memorizzata su disco nella directory *c:\java\mieclassi* , la variabile classpath dovrà contenere una stringa del tipo:

CLASSPATH=.;.\;c:\java\mieclassi\;…….

I valori "." E ".\" indicano alla JVM che le classi definite dall'utente si trovano all'interno della directory corrente o in un package definito a partire dalla directory corrente.

Il Java Development Kit (JDK)

Java Development Kit è un insieme di strumenti ed utilità ed è messo a disposizione gratuitamente da tanti produttori di software. L'insieme base o standard delle funzionalità è supportato direttamente da SUN Microsystem ed include un compilatore (javac), una Java Virtual Machine (java), un debugger e tanti altri strumenti necessari allo sviluppo di applicazioni Java. Inoltre il JDK comprende, oltre alle utilità a linea di comando, un completo insieme di classi pre-compilate ed il relativo codice sorgente.

La documentazione è generalmente distribuita separatamente, è rilasciata in formato HTML (*Figura2-4*) e copre tutto l'insieme delle classi rilasciate con il JDK a cui da ora in poi ci riferiremo come alle Java API (Application Program Interface).

[3] Gli archivi di classi verranno trattati esaustivamente nei capitoli successivi.

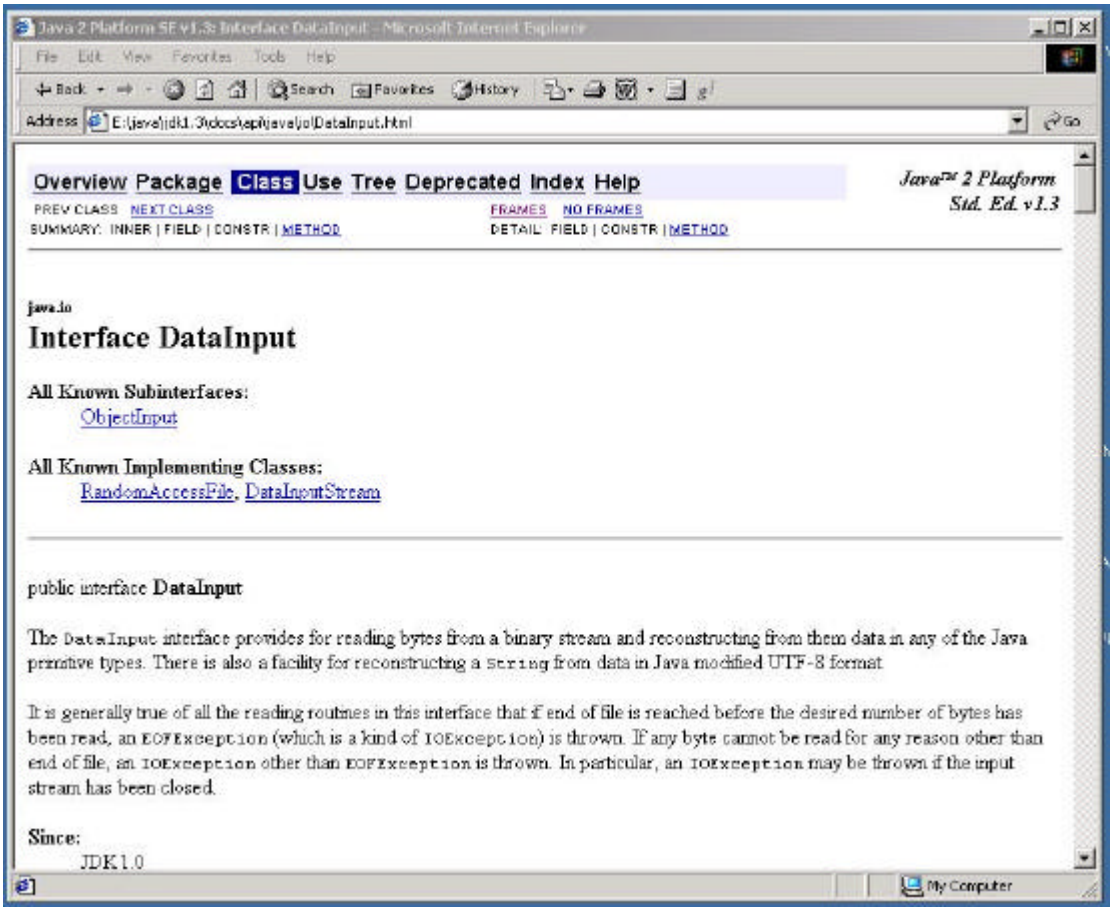


Figura 2-4: *Java Doc*

Scaricare ed installare il JDK

Questo testo fa riferimento alla ultima versione del Java Development Kit rilasciata da SUN nel corso del 2000: il JDK 1.3 . JDK può essere scaricato gratuitamente insieme a tutta la documentazione dal sito della SUN (http://www.javasoft.com), facendo ovviamente attenzione che il prodotto che si sta scaricando sia quello relativo alla piattaforma da utilizzare. L'installazione del prodotto è semplice e, nel caso di piattaforme Microsoft richiede solo la esecuzione di un file auto-installante. Per semplicità da questo momento in poi faremo riferimento a questi ambienti.

Al momento della istallazione, a meno di specifiche differenti, il JDK crea all'interno del disco principale la directory "jdk1.3" all'interno della quale istallerà tutto il necessario al programmatore. Sotto questa directory verranno create le seguenti sottodirettori:

c:\jdk1.3\bin contenente tutti i comandi java per compilare, le utility di servizio, oltre che ad una quantità di librerie *dll* di utilità varie.

c:\jdk1.3\demo contenente molti esempi comprensivi di eseguibili e codici sorgenti;

c:\jdk1.3\include contenente alcune librerie per poter utilizzare chiamate a funzioni scritte in C o C++;

c:\jdk1.3\lib contenente alcune file di libreria tra cui tools.jar contenete le API rilasciate da SUN ;

c:\jdk1.3\src contenente il codice sorgente delle classi contenute nell'archivio tools.jar.

Il compilatore Java (javac)

Il compilatore java (**javac**) accetta come argomento da linea di comando il nome di un file che deve terminare con la estensione .java. Il file passato come argomento deve essere un normale file ASCII contenente del codice Java valido.

Il compilatore processerà il contenuto del file e produrrà come risultato un file con estensione .class e con nome uguale a quello datogli in pasto contenente il Bytecodes generato dal codice sorgente ed eseguibile tramite virtual machine. I parametri più comuni che possono essere passati tramite linea di comando al compilatore sono generalmente :

-O : questa opzione attiva l'ottimizzazione del codice
-g : questa opzione attiva il supporto per il debug del codice
-classpath path : specifica l'esatto percorso su disco per trovare le librerie o gli oggetti da utilizzare.

L'ultima opzione può essere omessa purché sia definita la variabile di ambiente CLASSPATH. Ad esempio questi sono i passi da compiere per compilare una applicazione Java utilizzando il "command prompt" di windows:

C:\> **set CLASSPATH=c:\java\lib\classes.zip**
C:\> **c:\jdk1.3\bin\javac –O pippo.java**
C:\>

Il debugger Java (JDB)

Il debugger (**jdb**) è uno strumento è utilizzato per effettuare le operazioni di debug di applicazioni Java ed è molto simile al debugger standard su sistemi Unix (dbx). Per utilizzare il debugger, è necessario che le applicazioni siano compilate utilizzando l'opzione –g

C:\> **c:\jdk1.3\bin\javac –g pippo.java**

Dopo che il codice è stato compilato, il debugger può essere utilizzato direttamente chiamando attraverso linea di comando.

C:\> **c:\jdk1.3\bin\jdb pippo**

L'interprete Java

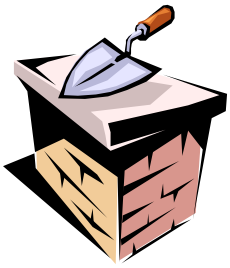
L'interprete Java (**java**) è utilizzato per eseguire applicazioni Java stand-alone ed accetta argomenti tramite linea di comando da trasmettere al metodo main() della applicazione (nei paragrafi successivi spiegheremo nei dettagli il meccanismo di passaggio di parametri ad una applicazione Java).

La linea di comando per attivare l'interprete prende la forma seguente:

C:\> **c:\jdk1.3\bin\java** ClassName arg1 arg2 …..

L'interprete va utilizzato solamente con file contenenti Bytecodes (ovvero con estensione .class) in particolare è in grado di eseguire solo file che rappresentano una applicazione ovvero contengono il metodo statico main() (anche questo aspetto verrà approfondito nei capitoli successivi).

Le informazioni di cui necessita il traduttore sono le stesse del compilatore. E' infatti necessario utilizzare la variabile di ambiente CLASSPATH o il parametro a linea di comando –classpath per indicare al programma la locazione su disco dei file da caricare.



Capitolo 3
Introduzione alla sintassi di Java

Introduzione

In questo capitolo verranno trattati gli aspetti specifici del linguaggio Java: le regole sintattiche base per la definizione di classi, l'instanziamento di oggetti e la definizione dell'entry point di una applicazione.

Per tutta la durata del capitolo sarà importante ricordare i concetti base discussi nei capitoli precedenti , in particolar modo quelli relativi alla definizione di una classe. Le definizioni di classe rappresentano il punto centrale dei programmi Java. Le classi hanno la funzione di contenitori logici per dati e codice e facilitano la creazione di oggetti che compongono la applicazione.

Per completezza il capitolo tratterà le caratteristiche del linguaggio necessarie per scrivere piccoli programmi includendo la manipolazione di stringhe e la generazione di output a video.

Variabili

Per scrivere applicazioni Java un programmatore deve poter creare oggetti, e per creare oggetti è necessario poter rappresentarne i dati. Il linguaggio mette a disposizione del programmatore una serie di primitive utili alla definizione di oggetti più complessi.

Per garantire la portabilità del Bytecodes da una piattaforma ad un'altra Java fissa le dimensioni di ogni dato primitivo . Queste dimensioni sono quindi definite e non variano se passiamo da un ambiente ad un altro, cosa che non succede con gli altri linguaggi di programmazione. I tipi numerici sono da considerarsi tutti con segno. La tabella a seguire schematizza i dati primitivi messi a disposizione da Java.

Primitiva	***Dimensione***	***Val. minimo***	***Val.Massimo***
boolean	1-bit	-	-
char	16-bit	Unicode 0	Unicode $2^{16} - 1$
byte	8-bit	-128	+127
short	16-bit	-2^{15}	$+2^{15} - 1$
int	32-bit	-2^{31}	$+2^{31} - 1$
long	64-bit	-2^{63}	$+2^{63} - 1$
float	32-bit	IEEE754	IEEE754
double	64 -bit	IEEE754	IEEE754
void	-	-	-

La dichiarazione di un dato primitivo in Java ha la seguente forma:

identificatore var_name;

dove l'identificatore è uno tra i tipi descritti nella prima colonna della tabella e var_name rappresenta il nome della variabile, può contenere caratteri alfanumerici ma deve iniziare necessariamente con una lettera. E' possibile creare più di una variabile dello stesso tipo utilizzando una virgola per separare tra di loro i nomi delle variabili:

identificatore var_name, var_name, …. ;

Per convenzione l'identificatore di una variabile deve iniziare con una lettera minuscola.

Inizializzazione di una variabile

Ogni variabile in Java richiede che al momento della dichiarazione le venga assegnato un valore iniziale. Come per il linguaggio c e C++ l'inizializzazione di una variabile in Java può essere effettuata direttamente al moneto della sua dichiarazione con la sintassi seguente:

identificatore var_name = var_value;

dove *var_value* rappresenta un valore legale per il tipo di variabile dichiarata, ed "=" rappresenta l'operatore di assegnamento. Ad esempio è possibile dichiarare ed inizializzare una variale intera con la riga di codice:

int mioPrimoIntero = 100;

In alternativa, Java assegna ad ogni variabile un valore di default al momento della dichiarazione. La tabella riassume il valore iniziale assegnato dalla JVM nel caso in cui una variabile non venga inizializzata dal programmatore:

Tipo primitivo	Valore assegnato dalla JVM
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0

Variabili final

A differenza di molti altri linguaggi, Java non consente di definire costanti. Per far fronte alla mancanza è possibile utilizzare il modificatore **final**. Una variabile dichiarata final si comporta come una costante, pertanto le deve essere assegnato il valore iniziale al momento della sua dichiarazione utilizzando l'operatore "=" di assegnamento.

final *identificatore var_name = var_value;*

Le variabili di questo tipo vengono inizializzate solo una volta al momento della dichiarazione. Qualsiasi altro tentativo di assegnamento si risolverà in un errore di compilazione.

Operatori

Una volta definito, un' oggetto deve poter manipolare i dati. Java mette a disposizione del programmatore una serie di operatori utili allo scopo. Nella tabella a seguire sono stati elencati la maggior parte degli operatori Java ordinati secondo l'ordine di precedenza; dal più alto al più basso. Gran parte degli operatori Java sono stati importati dal set degli operatori del linguaggio C, a cui ne sono stati aggiunti

nuovi allo scopo di supportare le nuove funzionalità messe a disposizione da questo linguaggio.

Tutti gli operatori funzionano solamente con dati primitivi a parte gli operatori !=, == e = che hanno effetto anche se gli operandi sono rappresentati da oggetti. Inoltre la classe String utilizza gli operatori + e += per operazioni di concatenazione.

Come in C, gli operatori di uguaglianza e disuguaglianza sono == (uguale a) e != (non uguale a). Si nota subito la disuguaglianza con gli stessi operatori come definiti dall'algebra: = e <>. L'uso dell'operatore digrafo[4] == è necessario dal momento che = è utilizzato esclusivamente come operatore di assegnamento. L'operatore != compare in questa forma per consistenza con la definizione dell'operatore logico ! (NOT).

Operatori	Funzioni
++ -- + - ?	Aritmetiche unarie e booleane
* / %	Aritmetiche
+ -	Addizione, sottrazione e concatenazione
<< >> >>>	Shift di bit
< <= > >= ***instanceof***	Comparazione
== !=	Uguaglianza e disuguaglianza
&	(bit a bit) AND
^	(bit a bit) XOR
|	(bit a bit) OR
&&	AND Logico
||	OR Logico
!	NOT Logico
expr ? expr :expr	Condizione a tre
= *= /+ %= += -= <<= >>>= n &= ^= |=	Assegnamento e di combinazione

Anche gli operatori "bit a bit" e logici derivano dal C sono completamente separati tra di loro. Ad esempio l'operatore & è utilizzato per combinare due interi bit per bit e l'operatore && è utilizzato per eseguire l'operazione di AND logico tra due espressioni booleane.

Quindi mentre (1011 & 1001) restituirà 1001 (a == a && b != b) restituirà false. La differenza con C, sta nel fatto che gli operatori logici in Java sono di tipo "**short-circuit**" ossia, se il lato sinistro di una espressione fornisce informazioni sufficienti a completare l'intera operazione, il lato destro della espressione non verrà valutato. Per esempio, si consideri l'espressione booleana

[4] Gli operatori digrafi sono operatori formati dalla combinazione di due simboli. I due simboli debbono essere adiacenti ed ordinati.

(a == a) || (b == c)

La valutazione del lato sinistro dell'espressione fornisce valore "*true*". Dal momento che si tratta di una operazione di OR logico, non c'è motivo a proseguire nella valutazione del lato sinistro della espressione, così che b non sarà mai comparato con c. Questo meccanismo all'apparenza poco utile, si rivela invece estremamente valido nei casi di chiamate a funzioni complesse per controllare la complessità della applicazione. Se infatti scriviamo una chiamata a funzione nel modo seguente :

(A == B) && (f() == 2)

dove f() è una funzione arbitrariamente complessa, f() non sarà eseguita se A non è uguale a B.

Sempre dal C Java eredita gli operatori unari di incremento e decremento ++ e --: i++ equivale a i=i+1 e i-- equivale a i=i-1.

Infine gli operatori di combinazione, combinano un assegnamento con una operazione aritmetica: i*=2 equivale a i=i*2. Questi operatori anche se semplificano la scrittura del codice lo rendono di difficile comprensione al nuovo programmatore che avesse la necessità di apportare modifiche. Per questo motivo non sono comunemente utilizzati.

Operatori aritmetici

Java supporta tutti i più comuni operatori aritmetici (somma, sottrazione, moltiplicazione, divisione e modulo), in aggiunta fornisce una serie di operatori che semplificano la vita al programmatore consentendogli, in alcuni casi, di ridurre la quantità di codice da scrivere. Gli operatori aritmetici sono suddivisi in due classi: operatori binari ed operatori unari.

Gli operatori binari (ovvero operatori che necessitano di due operandi) sono cinque e sono schematizzati nella tabella seguente:

Operatori Aritmetici Binari		
Operatore	**Utilizzo**	**Descrizione**
+	res=sx + dx	res = somma algebrica di dx ed sx
-	res= sx - dx	res = sottrazione algebrica di dx da sx
*	res= sx * dx	res = moltiplicazione algebrica tra sx e dx
/	res= sx / dx	res = divisione algebrica di sx con dx
%	res= sx % dx	res = resto della divisione tra sx e dx

Consideriamo ora le seguenti poche righe di codice:

```
int sx = 1500;
long dx = 1.000.000.000
??? res;
res = sx * dx;
```

Nasce il problema di rappresentare correttamente la variabile res affinché le si assegnare il risultato della operazione. Essendo tale risultato 1.500.000.000.000 troppo grande per essere assegnato ad una variabile di tipo int, sarà necessario utilizzare una variabile in grado di contenere correttamente il valore prodotto. Il nostro codice funzionerà perfettamente se riscritto nel modo seguente:

```
int sx = 1500;
long dx = 1.000.000.000
long res;
res = sx * dx;
```

Quello che notiamo è che se i due operandi non rappresentano uno stesso tipo, nel nostro caso un tipo int ed un tipo long, Java prima di valutare l'espressione trasforma implicitamente il tipo int in long e produce un valore di tipo long. Questo processo di conversione implicita dei tipi viene effettuato da Java secondo alcune regole ben precise. Queste regole possono essere riassunte come segue:

*Il risultato di una espressione aritmetica è di tipo **long** se almeno un operando è di tipo **long** e, nessun operando è di tipo **float** o **double**;*
*Il risultato di una espressione aritmetica è di tipo **int** se entrambi gli operandi sono di tipo **int**;*
*Il risultato di una espressione aritmetica è di tipo **float** se almeno un operando è di tipo **float** e, nessun operando è di tipo **double**;*
*Il risultato di una espressione aritmetica è di tipo **double** se almeno un operando è di tipo **double**;*

Gli operatori + e -, oltre ad avere una forma binaria hanno una forma unaria il cui significato è definito dalle seguenti regole:

*+op : trasforma l'operando op in un tipo **int** se è dichiarato di tipo **char**, **byte** o **short**;*
-op : restituisce la negazione aritmetica di op;

Non resta che parlare degli operatori aritmetici di tipo "*shortcut*" . Questo tipo di operatori consente l'incremento od il decremento di uno come riassunto nella tabella:

Forma shortcut	Forma estesa corrispondente	Risultato dopo l'esecuzione

int i=0;
int j;
j=i++;
	int i=0;
	int j;
	j=i;
	i=i+1;
		i=1
		j=0

int i=1;
int j;
j=i--;
	int i=1;
	int j;
	j=i;
	i=i-1;
		i=0
		j=1

int i=0;
int j;
j=++i;
	int i=0;
	int j;
	i=i+1;
	j=i;
		i=1
		j=1

int i=1;
int j;
j=--i;
	int i=1;
	int j;
	i=i-1;
	j=i;
		i=0
		j=0

Operatori relazionali

Gli operatori relazionali servono ad effettuare un confronto tra valori producendo come risultato di ritorno un valore booleano (**true** o **false**) come prodotto del confronto. Nella tabella sono riassunti gli operatori ed il loro significato.

Operatori Relazionali		
Operatore	**Utilizzo**	**Descrizione**
>	res=sx > dx	res = true se e solo se sx è maggiore di dx
>=	res= sx >= dx	res = true se e solo se sx è maggiore o uguale di dx
<	res= sx < dx	res = true se e solo se sx è minore di dx
<=	res= sx <= dx	res = true se e solo se sx è minore o uguale di dx
!=	res= sx != dx	res = true se e solo se sx è diverso da dx

Operatori condizionali

Gli operatori condizionali consentono di effettuare operazioni logiche su operandi di tipo booleano, ossia operandi che prendono solo valori true o false. Questi operatori sono quattro e sono riassunti nella tabella:

Operatori Condizionali		
Operatore	**Utilizzo**	**Descrizione**
&&	res=sx && dx	AND : res = true se e solo se sx vale true e dx vale true, false altrimenti.
||	res= sx || dx	OR : res = true se e solo se almeno uno tra sx e dx vale true, false altrimenti.
!	res= ! sx	NOT : res = true se e solo se sx vale false, false altrimenti.
^	res= sx ^ dx	XOR : res = true se e solo se uno solo dei due operandi vale true, false altrimenti.

Nelle tabelle successive vengono specificati tutti i possibili valori booleani prodotti dagli operatori descritti.

AND (&&)		
sx	**dx**	**res**
true	true	true
true	false	false
false	true	false
false	false	false

OR (||)		
sx	**dx**	**res**
true	true	true
true	false	true
false	true	true
false	false	false

NOT (!)	
sx	**res**
true	false
false	true

XOR (^)		
sx	**dx**	**res**
true	true	false
true	false	true
false	true	true
false	false	false

Operatori logici e di shift bit a bit

Gli operatori di shift bit a bit consentono di manipolare tipi primitivi spostandone i bit verso sinistra o verso destra secondo le regole definite nella tabella seguente

Operatori di shift bit a bit		
Operatore	**Utilizzo**	**Descrizione**
>>	sx >> dx	Sposta i bit di sx verso destra di un numero di posizioni come stabilito da dx.
<<	sx << dx	Sposta i bit di sx verso sinistra di un numero di posizioni come stabilito da dx.
>>>	sx >>> dx	Sposta i bit di sx verso sinistra di un numero di posizioni come stabilito da dx, ove dx è da considerarsi un intero senza segno.

Consideriamo ad esempio:

```
byte i = 100;
i >> 1;
```

dal momento che la rappresentazione binaria del numero decimale 100 è 01100100, lo shift verso destra di una posizione dei bit, produrrà come risultato il numero binario 00110010 che corrisponde al valore 50 decimale.

Oltre ad operatori di shift, Java consente di eseguire operazioni logiche su tipi primitivi operando come nel caso precedente sulla loro rappresentazione binaria.

Operatori logici bit a bit		
Operatore	**Utilizzo**	**Descrizione**
&	res = sx & dx	AND bit a bit
|	res = sx | dx	OR bit a bit
^	res = sx ^ dx	XOR bit a bit
~	res = ~sx	COMPLEMENTO A UNO bit a bit

Nelle tabelle seguenti sono riportati tutti i possibili risultati prodotti dalla applicazione degli operatori nella tabella precedente. Tutte le combinazioni sono state effettuate considerando un singolo bit degli operandi.

AND (&)		
sx (bit)	**dx (bit)**	**res (bit)**
1	1	1
1	0	0
0	1	0
0	0	0

OR (|)		
sx (bit)	**dx (bit)**	**res (bit)**
1	1	1
1	0	1
0	1	1
0	0	0

COMPLEMENTO (~)	
sx (bit)	**res (bit)**
1	0
0	1

XOR (^)		
sx (bit)	**dx (bit)**	**res (bit)**
1	1	0
1	0	1
0	1	1
0	0	0

Il prossimo è un esempio di applicazione di operatori logici bit a bit tra variabili di tipo **byte**:

```
byte sx = 100;
byte dx = 125;
byte res1 = sx & dx;
byte res2 = sx | dx;
byte res3 = sx ^ dx;
```

byte res4 = ~sx;

Dal momento che la rappresentazione binaria di sx e dx è rispettivamente:

variabile	decimale	binario
sx	100	01100100
dx	125	01111101

L'esecuzione del codice produrrà i risultati seguenti:

operatore	sx	dx	risultato	decimale
&	01100100	01111101	01100100	100
|	01100100	01111101	01111101	125
^	01100100	01111101	00011001	25
~	01100100	----------	10011011	155

Operatori di assegnamento

L'operatore di assegnamento "=" consente al programmatore, una volta definita una variabile, di assegnarle un valore. La sintassi da utilizzare è la seguente:

result_type res_var = espressione;

Espressione rappresenta una qualsiasi espressione che produce un valore del tipo compatibile con il tipo definito da result_type, e res_var rappresenta l'identificatore della variabile che conterrà il risultato. Se torniamo alla tabella definita nel paragrafo "*Operatori*", vediamo che l'operatore di assegnamento ha la priorità più bassa rispetto a tutti gli altri. La riga di codice Java produrrà quindi la valutazione della espressione ed infine l'assegnamento del risultato alla variabile res_var. Ad esempio:

int res1 = 5+10;

Esegue l'espressione alla destra dell'operatore e ne assegna il risultato (15) a res1.

int res1 = 5;

Assegna il valore 5 alla destra dell'operatore alla variabile res1. Oltre all'operatore "=" Java mette a disposizione del programmatore una serie di operatori di assegnamento di tipo "shortcut" (in italiano scorciatoia), definiti nella prossima tabella. Questi operatori combinano un operatore aritmetico o logico con l'operatore di assegnamento.

Operatori di assegnamento shortcut		
Operatore	**Utilizzo**	**Equivalente a**
+=	sx +=dx	sx = sx + dx;
-=	sx -=dx	sx = sx - dx;
*=	sx *=dx	sx = sx * dx;
/=	sx /=dx	sx = sx / dx;
%=	sx %=dx	sx = sx % dx;
&=	sx &=dx	sx = sx & dx;
|=	sx |=dx	sx = sx | dx;
^=	sx ^=dx	sx = sx ^ dx;
<<=	sx <<=dx	sx = sx << dx;
>>=	sx >>=dx	sx = sx >> dx;
>>>=	sx >>>=dx	sx = sx >>> dx;

Espressioni

Le espressioni rappresentano il meccanismo per effettuare calcoli all'interno della nostra applicazione, e combinano variabili e operatori producendo un singolo valore di ritorno. Le espressioni vengono utilizzate per assegnare valori a variabili o modificare, come vedremo nel prossimo capitolo, il flusso della esecuzione di una applicazione Java.

Una espressione non rappresenta una unità di calcolo completa inquanto non produce assegnamenti o modifiche alle variabili della applicazione.

Istruzioni

A differenza delle espressioni, le istruzioni sono unità eseguibili complete terminate dal carattere ";" e combinano operazioni di assegnamento, valutazione di espressioni o chiamate ad oggetti (quest'ultimo concetto risulterà più chiaro alla fine del capitolo) combinate tra loro a partire dalle regole sintattiche del linguaggio.

Regole sintattiche di Java

Una istruzione rappresenta il mattone per la costruzione di oggetti. Difatti la sintassi del linguaggio Java può essere descritta da tre sole regole di espansione:

istruzione --> expression

OR

istruzione --> {
istruzione
[istruzione]
}

OR

istruzione --> flow_control
istruzione

Queste tre regole hanno natura ricorsiva e, la freccia deve essere letta come "diventa". Sostituendo una qualunque di queste tre definizioni all'interno del lato destro di ogni espansione possono essere generati una infinità di istruzioni. Di seguito un esempio.

Prendiamo in considerazione la terza regola.

istruzione --> flow_control
Statement

E sostituiamo il lato destro utilizzando la seconda espansione ottenendo

istruzione --> flow_control **-->** flow_control
istruzione **2** {
istruzione
}

Applicando ora la terza regola di espansione otteniamo :

istruzione --> flow_control **-->** flow_control **-->** flow_control
istruzione **2** { **3** {
istruzione flow_control
} istruzione
}

Prendiamo per buona che l'istruzione **if** sia uno statement per il controllo del flusso della applicazione (flow_control), e facciamo un ulteriore sforzo accettando che la sua sintassi sia:

if(boolean_expression)

Ecco che la nostra espansione diventerà quindi :

istruzione --> **--> -->** flow_control **-->** **if(i>10)**
2 3 { **{**
flow_control **if(I==5)**
istruzione **print("Il valore di I è 5");**
} **i++;**
}

Blocchi di istruzioni

La seconda regola di espansione:

istruzione --> {
istruzione
[istruzione]
}

definisce la struttura di un blocco di istruzioni, ovvero una sequenza di una o più istruzioni racchiuse all'interno di parentesi graffe.

Metodi

Una istruzione rappresenta il mattone per creare le funzionalità di un oggetto. Nasce spontaneo chiedersi: come vengono organizzate le istruzioni all'interno di oggetti?

I metodi rappresentano il cemento che tiene assieme tutti i mattoni. I metodi sono gruppi di istruzioni riuniti a fornire una singola funzionalità. Essi hanno una sintassi

molto simile a quella della definizione di funzioni ANSI C e possono essere descritti con la seguente forma:

return_type method_name(arg_type name [,arg_type name])
{
    istruzioni
}

return_type e *arg_type* rappresentano ogni tipo di dato (primitivo o oggetto), *name* e *method_name* sono identificatori alfanumerici ed iniziano con una lettera (discuteremo in seguito come avviene il passaggio di parametri).

Se il metodo non ritorna valori, dovrà essere utilizzata la chiave speciale *void* al posto di *return_type*.

Se il corpo di un metodo contiene dichiarazioni di variabili, queste saranno visibili solo all'interno del metodo stesso, ed il loro ciclo di vita sarà limitato alla esecuzione del metodo. Non manterranno il loro valore tra chiamate differenti e, non saranno accessibili da altri metodi.

Per evitare la allocazione di variabili inutilizzate, il compilatore java prevede una forma di controllo secondo la quale un metodo non può essere compilato se esistono variabili a cui non è stato assegnato alcun valore. In questi casi la compilazione verrà interrotta con la restituzione di un messaggio di errore. Ad esempio consideriamo il metodo seguente:

```
int moltiplica_per_tre (int number)
{
    int risultato = 0;
    risultato = number*3;
    return risultato;
}
```

In questo caso la compilazione andrà a buon fine ed il compilatore Java produrrà come risultato un file contenente Bytecodes. Se invece il metodo avesse la forma:

```
int moltiplica_per_tre (int number)
{
    int risultato;
    return number*3;
}
```

il compilatore produrrebbe un messaggio di errore dal momento che esiste una variabile dichiarata e mai utilizzata.

Definire una classe

Le istruzioni sono organizzate utilizzando metodi, e i metodi forniscono funzionalità. D'altro canto, Java è un linguaggio object oriented, e come tale richiede che le funzionalità (metodi) siano organizzati in classi.

Nel primo capitolo, una classe è stata paragonata al concetto di categoria. Se trasportato nel contesto del linguaggio di programmazione la definizione non cambia, ma è importante chiarire le implicazioni che la cosa comporta. Una classe Java deve rappresentare un oggetto concettuale. Per poterlo fare deve raggruppare dati e metodi assegnando un nome comune. La sintassi è la seguente:

```
class ObjectName
{
    data_declarations
    method_declarations
}
```

I dati ed I metodi contenuti all'interno della classe vengono chiamati **membri** della classe. E' importante notare che dati e metodi devono essere rigorosamente definiti all'interno della classe. Non è possibile in nessun modo dichiarare variabili globali, funzioni o procedure. Questa restrizione del linguaggio Java, scoraggia il programmatore ad effettuare una decomposizione procedurale, incoraggiando di conseguenza ad utilizzare l'approccio object oriented.

Riprendendo la classe "libro" descritta nel primo capitolo, ricordiamo che avevamo stabilito che un libro è tale solo se contiene pagine, le pagine si possono sfogliare, strappare etc.. Utilizzando la sintassi di Java potremmo fornire una grossolana definizione della nostra classe nel modo seguente:

```
class Libro
{
    // dichiarazione dei dati

    int numero_di_pagine;
    int pagina_attuale;
    …
    …

    // dichiarazione dei metodi

    void strappaUnaPagina(int numero_della_pagina) {}
    int paginaCorrente(){}
    void giraLaPagina(){}
}
```

Variabili reference

Java fa una netta distinzione tra Classi e tipi primitivi. Una delle maggiori differenze è che un oggetto non è allocato dal linguaggio al momento della dichiarazione. Per chiarire questo punto, consideriamo la seguente dichiarazione:

```
int counter;
```

Questa dichiarazione crea una variabile intera chiamata *counter* ed alloca subito quattro byte per lo "storage" del dato. Con le classi lo scenario cambia e la dichiarazione

```
Stack s;
```

crea una variabile che referenzia l'oggetto, ma non crea l'oggetto Stack. Una variabile di referenza, è quindi una variabile speciale che tiene traccia di istanze di tipi non primitivi. Questo tipo di variabili hanno l'unica capacità di tracciare oggetti del tipo compatibile: ad esempio una referenza ad un oggetto di tipo Stack non può tracciare oggetti di diverso tipo.

Oltre che per gli oggetti, Java utilizza lo stesso meccanismo per gli array, che non sono allocati al momento della dichiarazione, ma viene semplicemente creata una variabile per referenziare l'entità. Un array può essere dichiarato utilizzando la sintassi:

int numbers[];

Una dichiarazione così fatta crea una variabile che tiene traccia di un array di interi di dimensione arbitraria.

Le variabili reference sono molto simili concettualmente ai puntatori di C e C++, ma non consentono la conversione intero/indirizzo o le operazioni aritmetiche; tuttavia queste variabili sono uno strumento potente per la creazione di strutture dati dinamiche come liste, alberi binari e array multidimensionali. In questo modo eliminano gli svantaggi derivanti dall'uso di puntatori, mentre ne mantengono tutti i vantaggi. Nella *Figura 3-1* sono riportati alcuni esempi relativi alla modalità utilizzata da Java per allocare primitive od oggetti.

Un'ultima considerazione da fare riguardo la gestione delle variabili in Java è che a differenza di C e C++ in cui un dato rappresenta il corrispondente dato-macchina ossia una variabile intera in C++ occupa 32 bit ed una variabile byte ne occupa 8, ora le variabili si "***comportano come se***".

La virtual machine Java difatti alloca per ogni dato primitivo il massimo disponibile in fatto di rappresentazione macchina dei dati. La virtual machine riserverà, su una macchina a 32 bit, 32 bit sia per variabili intere che variabili byte, quello che cambia è che il programmatore vedrà una variabile byte comportarsi come tale ed altrettanto per le altre primitive.

Questo, a discapito di un maggior consumo di risorse, fornisce però molti vantaggi: primo consente la portabilità del Bytecodes su ogni piattaforma garantendo che una variabile si comporterà sempre allo stesso modo, secondo sfrutta al massimo le capacità della piattaforma che utilizzerà le FPU (Floating Point Unit) anche per calcoli su variabili intere di piccole dimensioni.

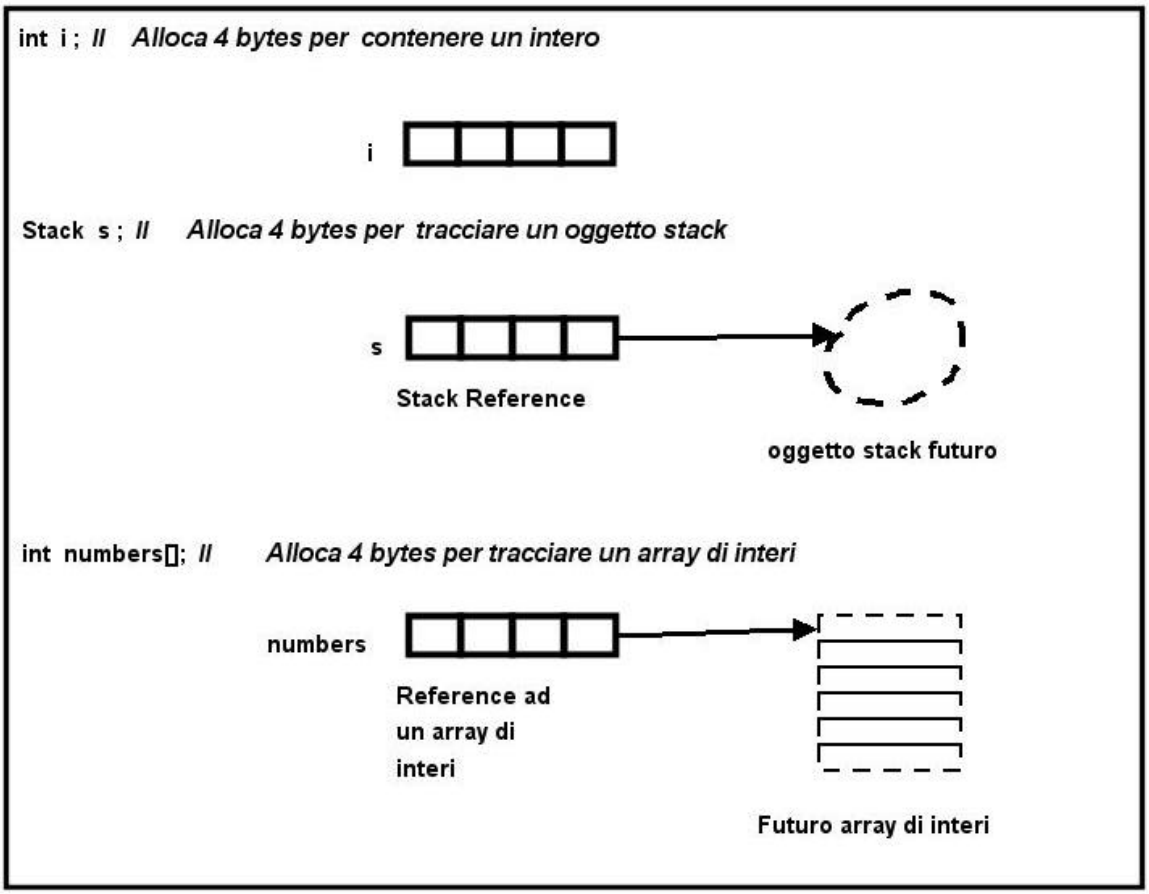


Figura 3-1 *: Variabili primitive e variabili reference*

Visibilità di una variabile Java

Come C e C++ Java consente di dichiarare variabili in qualunque punto del codice della applicazione a differenza di linguaggi come il Pascal che richiedevano che le variabili venissero dichiarate all'interno di un apposito blocco dedicato.

Dal momento che secondo questa regola potrebbe risultare possibile dichiarare più variabili con lo stesso nome in punti differenti del codice, è necessario stabilire le regole di visibilità delle variabili. I blocchi di istruzioni ci forniscono il meccanismo necessario per delimitare quello che chiameremo "scope" di una variabile Java. In generale diremo che una variabile ha "scope" limitato al blocco all'interno del quale è stata dichiarata.

Analizziamo quindi caso per caso lo scope di variabili. Quando definiamo una classe, racchiudiamo all'interno del suo blocco di istruzioni sia dichiarazioni di variabili membro che dichiarazioni di metodi membro. Secondo la regola definita, e come schematizzato nella *Figura 3-2*, i dati membro di una classe sono visibili a da tutti i metodi dichiarati all'interno della definizione dell'oggetto.

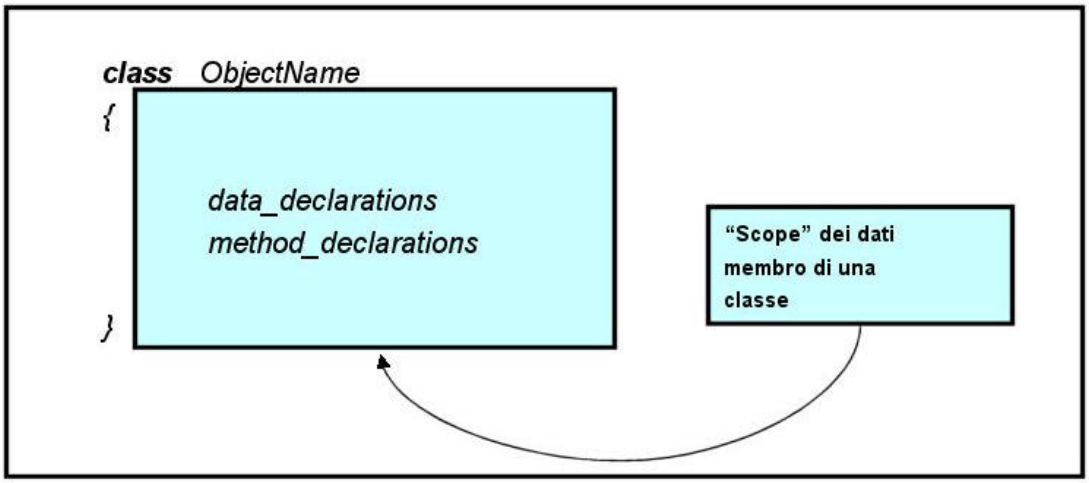


Figura 3-2 : *"scope" dei dati membro di una classe*

Nel caso di metodi membro vale a sua volta lo schema degli scope come definito nella *Figura 3-3*.

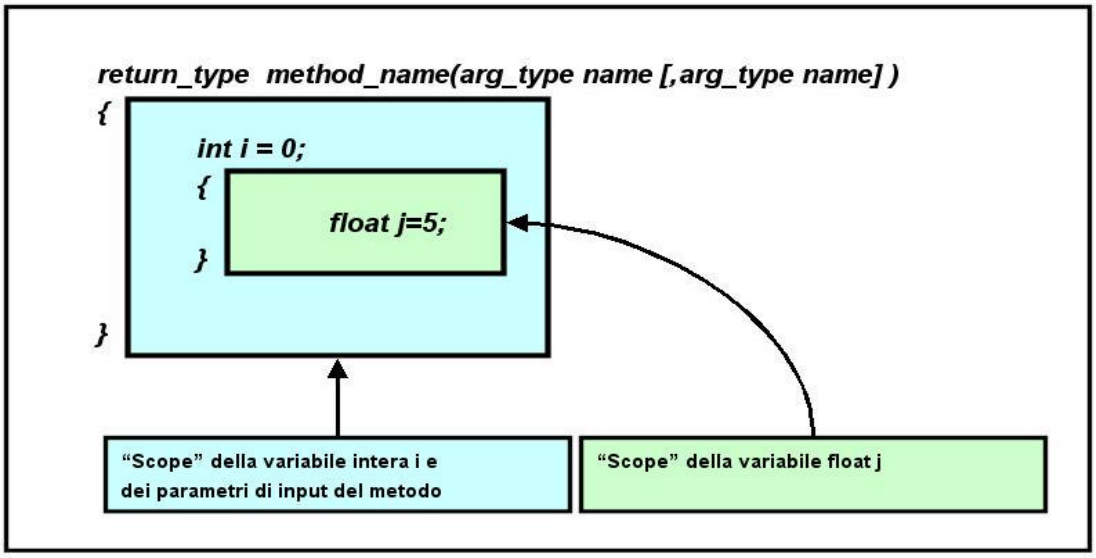


Figura 3-3 : *"scope" delle variabili in una dichiarazione di metodo*

L'oggetto null

Il linguaggio Java prevede un valore speciale per le variabili reference che non referenzia nessuna istanza di un oggetto. Il valore speciale *null* rappresenta un oggetto inesistente, e viene assegnato di default ad ogni variabile reference.

Se la applicazione esegue una chiamata ad un oggetto tramite una variabile reference non inizializzato, il compilatore Java produrrà un messaggio di errore di tipo NullPointerException[5].

Quando ad una variabile reference viene assegnato il valore *null*, l'oggetto referenziato verrà rilasciato e, se non utilizzato verrà dato in pasto alla garbage collection che si occuperà di rilasciare la memoria allocata per la entità.

Altro uso che può essere fatto dell'oggetto *null* riguarda le operazioni di comparazione come visibile nell'esempio seguente. Le poche righe di codice dichiarano un array di interi chiamato *numbers*. Mediante l'istruzione per il controllo di flusso **if**[6] controlla se l'oggetto array sia stato creato o no.

```
Stack s = null;
int numbers[];

if (numbers == null)
{
    ……..
}
```

Facciano attenzione I programmatori C, C++. Il valore *null* nel nostro caso non equivale al valore 0, ma rappresenta un oggetto nullo.

Creare istanze

Creata la variabile refence, siamo pronti a creare una istanza di un nuovo oggetto o di un array. L'operatore *new* fa questo per noi, allocando la memoria necessaria per il nostro oggetto e tornando la locazione in memoria della entità creata. Questa locazione può quindi essere memorizzata nella variabile reference di tipo appropriato ed utilizzata per accedere all'oggetto quando necessario.

Quando utilizziamo l'operatore *new* con una classe, la sintassi è la seguente:

new *class_type()* ;

Le parentesi sono necessarie ed hanno un significato particolare che sveleremo presto. *Class_type* è una classe appartenente alle API di Java oppure definita dal programmatore. Per creare un oggetto Stack utilizzeremo quindi l'istruzione:

```
Stack s = new  Stack();
```

Che dichiara una variabile s di tipo Stack(), istanzia l'oggetto utilizzando la definizione di classe e memorizza la locazione della nuova istanza nella variabile. Un risultato analogo può essere ottenuto anche nel modo seguente:

```
Stack s = null;
s = new Stack();
```

Gli array sono allocati allo stesso modo:

[5] Le eccezioni verranno discusse in un capitolo a se
[6] Le istruzioni per il controllo di flusso verranno trattate nel capitolo successivo

```
int my_array[] = new int[20];
```

o, analogamente al caso precedente

```
int my_array[] = null;
my_array = new int[20];
```

In questo caso Java dichiarerà una variabile reference intera, allocherà memoria per 20 interi e memorizzerà la locazione di memoria a *my_array*. Il nuovo array sarà indicizzato a partire da 0[7].

L'operatore punto "."

Questo operatore è utilizzato per accedere ai membri di un oggetto tramite la variabile reference. Le due definizioni di classe a seguire, rappresentano una definizione per la classe Stack ed una per la classe StackElement che rappresentano vicendevolmente il concetto di stack e quello di elemento dello stack.

```
class StackElement
{
    int val;
}

class Stack
{
    StackElement pop()
    {
            …..
    }

    void push(StackElement stack_ele)
    {
            …..
    }
}
```

La classe StackElement contiene un dato membro chiamato *val* che rappresenta un numero intero da inserire all'interno dello Stack. Le due classi possono essere legate assieme a formare un breve blocco di codice che realizza una operazione di pop() ed una di push() sullo Stack utilizzando l'operatore "punto":

```
//Creiamo un oggetto StackElement ed inizializziamo il dato membro

StackElement stack_el = new StackElement();
int value = 10;
Stack_el.val = value ;

//Creiamo un oggetto Stack
Stack s = new Stack();

//inseriamo il valore in testa allo stack
s.push(Stack_el);
```

[7] In java come in C e C++ gli array di dimensione n indicizzano gli elementi con valori compresi tra 0 e (n-1). Ad esempio my_array[0] tornerà l'indice relativo al primo elemento dell'array.

```
//Rilasciamo il vecchio valore per StackElement e eseguiamo una operazione di
//pop sullo stack
Stack_el = null;
Stack_el = s.pop();
```

Auto referenza ed auto referenza esplicita

L'operatore "punto", oltre a fornire il meccanismo per accedere a dati i metodi di un oggetto attraverso la relativa variabile reference, consente ai metodi di una classe di accedere ai dati membro della classe di appartenenza. Proviamo a fare qualche modifica alla classe StackElement del paragrafo precedente:

```
class StackElement
{
    int val;

    // questo metodo inizializza il dato membro della classe
    void setVal(int valore)
    {
            ???.val = valore;
    }

}
```

Il metodo *setVal* prende come parametro un intero ed utilizza l'operatore punto per memorizzare il dato all'interno della variabile membro *val*. Il problema che rimane da risolvere (evidenziato nell'esempio dai punti interrogativi **???**), riguarda la modalità con cui un oggetto possa referenziare se stesso.

Java prevede una modalità di referenziazione speciale identificata da *this*. Difatti il valore di *this* viene modificato automaticamente da Java in modo che ad ogni istante sia sempre referenziato all'oggetto attivo, intendendo per "oggetto attivo" l'istanza dell'oggetto in esecuzione durante la chiamata al metodo corrente. La nostra classe diventa ora:

```
class StackElement
{
    int val;

    // questo metodo inizializza il dato membro della classe
    void setVal(int valore)
    {
            this.val = valore;
    }

}
```

Questa modalità di accesso viene detta **auto referenza esplicita** ed è applicabile ad ogni tipo di dato e metodo membro di una classe.

Auto referenza implicita

Dal momento che, come abbiamo detto, ogni metodo deve essere definito all'interno di una classe, i meccanismo di auto referenza è molto comune in applicazioni Java.

Se una referenza non risulta ambigua, Java consente di utilizzare un ulteriore meccanismo detto di **auto referenza implicita**, mediante il quale è possibile accede a dati o metodi membro di una classe senza necessariamente utilizzare *this*.

```
class StackElement
{
    int val;
    void setVal(int valore)
    {
            val = valore;
    }
}
```

Dal momento che la visibilità di una variabile in Java è limitata al blocco ai sottoblocchi di codice in cui è stata effettuata la dichiarazione, Java basa su questo meccanismo la auto referenza implicita. Formalmente java ricerca una variabile non qualificata risalendo a ritroso tra i diversi livelli dei blocchi di codice.

Prima di tutto Java ricerca la dichiarazione di variabile all'interno del blocco di istruzioni correntemente in esecuzione. Se la variabile non è un parametro appartenente al blocco risale tra i vari livelli del codice fino ad arrivare alla lista dei parametri del metodo corrente. Se la lista dei parametri del metodo non soddisfa la ricerca, Java legge il blocco di dichiarazione dell'oggetto corrente utilizzando quindi implicitamente *this*. Nel caso in cui la variabile non è neanche un dato membro dell'oggetto viene generato un codice di errore dal compilatore.

Anche se l'uso implicito di variabili facilita la scrittura di codice riducendo la quantità di caratteri da digitare, un abuso della tecnica rischia di provocare ambiguità all'interno del codice. Tipicamente la situazione a cui si va incontro è la seguente:

```
class StackElement
{
    int val;
    void setVal(int valore)
    {
            int val ;
            ……..
            ……..
            val = valore;
            ……
            ……
    }
}
```

Una assegnazione di questo tipo in assenza di this provocherà la perdita del dato passato come parametro al metodo *setVal(**int**)*, dato che verrà memorizzato in una variabile visibile solo all'interno del blocco di istruzioni del metodo e di conseguenza con ciclo di vita limitato al tempo necessario alla esecuzione del metodo. Il codice per funzionare correttamente dovrà essere modificato nel modo seguente:

```
class StackElement
{
    int val;
    void setVal(int valore)
    {
            int val ;
            ……..
            ……..
            this.val = valore;
            ……
```

```
            ……
        }
    }
```

Meno ambiguo è invece l'uso della auto referenza implicita se utilizzata per la chiamata a metodi della classe. In questo caso infatti Java applicherà soltanto il terzo passo dell'algoritmo descritto per la determinazione delle variabili. Un metodo infatti non può essere definito all'interno di un altro, e non può essere passato come argomento ad un'altro metodo.

Stringhe

Come abbiamo anticipato, Java ha a disposizione molti tipi già definiti. *String* è uno di questi, ed è dotato di molte caratteristiche particolari. Le stringhe sono oggetti che possono essere inizializzato usando semplicemente una notazione con doppi apici senza l'utilizzo dell'operatore **new**:

```
String prima = "Hello";
String seconda = "world";
```

Possono essere concatenate usando l'operatore di addizione:

```
String terza = prima + seconda;
```

Hanno un membro che ritorna la lunghezza della stringa rappresentata:

```
int lunghezza = prima.lenght();
```

Stato di un oggetto Java

Gli oggetti Java rappresentano dati molto complessi il cui stato, a differenza di un tipo primitivo, non può essere definito semplicemente dal valore della variabile reference. In particolare, definiamo stato di un oggetto il valore in un certo istante di tutti i dati membro della classe. Ad esempio lo stato dell'oggetto *StackElement* è rappresentato dal valore del dato membro val di tipo intero.

Comparazione di oggetti

In Java la comparazione di oggetti è leggermente differente rispetto ad altri linguaggi e ciò dipende dal modo in cui Java utilizza gli oggetti. Una applicazione Java non usa oggetti, ma usa variabili reference "come oggetti". Una normale comparazione effettuata utilizzando l'operatore == comparerebbe il riferimento e non lo stato degli oggetti. Ciò significa che == produrrà risultato *true* solo se le due variabili reference puntano allo stesso oggetto e non se i due oggetti distinti di tipo uguale sono nello stesso stato.

```
Stack a = new Stack();
Stack b = new Stack();

(a == b) -> false
```

Molte volte però ad una applicazione Java potrebbe tornare utile sapere se due istanze separate di una stessa classe sono nello stesso stato ovvero, ricordando la definizione data nel paragrafo precedente potremmo formulare la regola secondo la quale due oggetti java dello stesso tipo sono uguali se si trovano nello stesso stato al momento del confronto.

Java prevede un metodo speciale chiamato *equals()* che confronta lo stato di due oggetti. Di fatto, tutti gli oggetti in Java, anche quelli definiti dal programmatore, possiedono questo metodo inquanto ereditato per default da una classe particolare che analizzeremo in seguito parlando di ereditarietà.

```
Stack a = new Stack();
Stack b = new Stack();
a.push(1);
b.push(1);

a.equals(b) -> true
```

Metodi statici

Finora abbiamo mostrato segmenti di codice dando per scontato che siano parte di un processo attivo: in tutto questo c'è una falla. Per tapparla dobbiamo fare alcune considerazioni: primo, ogni metodo deve essere definito all'interno di una classe (questo incoraggia ad utilizzare il paradigma object oriented). Secondo, i metodi devono essere invocati utilizzando una variabile reference inizializzata in modo che tenga traccia della istanza di un oggetto.

Questo meccanismo rende possibile l'auto referenziazione inquanto se un metodo viene chiamato senza che l'oggetto di cui è membro sia attivo, *this* non sarebbe inizializzato. Il problema quindi è che in questo scenario un metodo per essere eseguito richiede un oggetto attivo, ma fino a che non c'è qualcosa in esecuzione un oggetto non può essere istanziato.

L'unica possibile soluzione è quindi quella di creare metodi speciali che non richiedano l'attività da parte dell'oggetto di cui sono membro così che possano essere utilizzati in qualsiasi momento.

La risposta è nei **metodi statici**, ossia metodi che appartengono a classi, ma non richiedono oggetti attivi. Questi metodi possono essere creati utilizzando la parola chiave ***static*** a sinistra della dichiarazione di un metodo come mostrato nella dichiarazione di *static_method()* nell'esempio che segue:

```
1   class esempio
2   {
3           static int static_method()
4           {
5                   .......
6           }
7           int non_static_method()
8           {
9                   return static_method();
10          }
11  }
12  class altra_classe
13  {
14          void un_metodo_qualunque()
15          {
16                  int i = esempio. static_method();
17          }
18  }
```

Un metodo statico esiste sempre a prescindere dallo stato dell'oggetto; tuttavia la locazione o classe incapsulante del metodo deve sempre essere ben qualificata. Questa tecnica è chiamata "scope resolution" e può essere realizzata in svariati

modi. Uno di questi consiste nell'utilizzare il nome della classe come se fosse una variabile reference:

```
esempio.static_metod();
```

Oppure si può utilizzare una variabile reference nel modo che conosciamo:

```
esempio _ese = new esempio();
_ese.static_metod();
```

Se il metodo statico viene chiamato da un altro membro della stessa classe non è necessario alcun accorgimento. E' importante tener bene a mente che un metodo statico non inizializza l'oggetto "this"; di conseguenza un oggetto statico non può utilizzare membri non statici della classe di appartenenza.

Il metodo main

Affinché la Java Virtual Machine possa eseguire una applicazione, è necessario che abbia ben chiaro quale debba essere il primo metodo da eseguire. Questo metodo viene detto "**entry point**" della applicazione. Come per il linguaggio C, Java riserva allo scopo l'identificatore di membro *main*.

Ogni classe può avere il suo metodo *main()*, ma solo il metodo main della classe specificata verrà eseguito all'avvio del processo. Questo significa che ogni classe di una applicazione può rappresentare un potenziale entry point che può quindi essere scelto all'avvio del processo scegliendo semplicemente la classe desiderata.

Come più avanti vedremo, una classe può contenere più metodi membro aventi lo stesso nome, purché abbiano differenti parametri in input. Affinché il metodo *main()* possa essere trovato dalla virtual machine, è necessario che abbia una lista di argomenti che accetti un array di stringhe. E' proprio grazie a questo array la virtual machine è in grado di passare alla applicazioni dei valori da riga di comando.

Infine, per il metodo *main()* è necessario utilizzare il modificatore *public*[8] che accorda alla virtual machine il permesso per eseguire il metodo.

```
class prima_applicazione
{
    public static void main(String args[])
    {
            ……….
    }
}
```

Tutto questo ci porta ad una importante considerazione finale : **tutto è un oggetto**, anche un applicazione.

L'oggetto System

Un'altra delle classi predefinite in Java è la classe System. Questa classe ha una serie di metodi statici e rappresenta il sistema su cui la applicazione Java sta girando.

Due dati membro statici di questa classe sono *System.out* e *System.err* che rappresentano rispettivamente lo standard output e lo standard error dell'interprete java. Usando il loro metodo statico *println()*, una applicazione Java è in grado di inviare un output sullo standard output o sullo standard error.

[8] Capiremo meglio il suo significato successivamente

```
System.out.println("Scrivo sullo standard output");
System.err.println("Scrivo sullo standard error");
```

Il metodo statico *System.exit(**int** number)* causa la terminazione della applicazione Java.

Laboratorio 3
Introduzione alla sintassi di Java

Descrizione

In questa sezione inizieremo a comprendere la tecnica dell'uso degli oggetti Java. Prima di proseguire è però necessario capire la sintassi di una semplice istruzione if[9]

La sintassi base di if è la seguente:

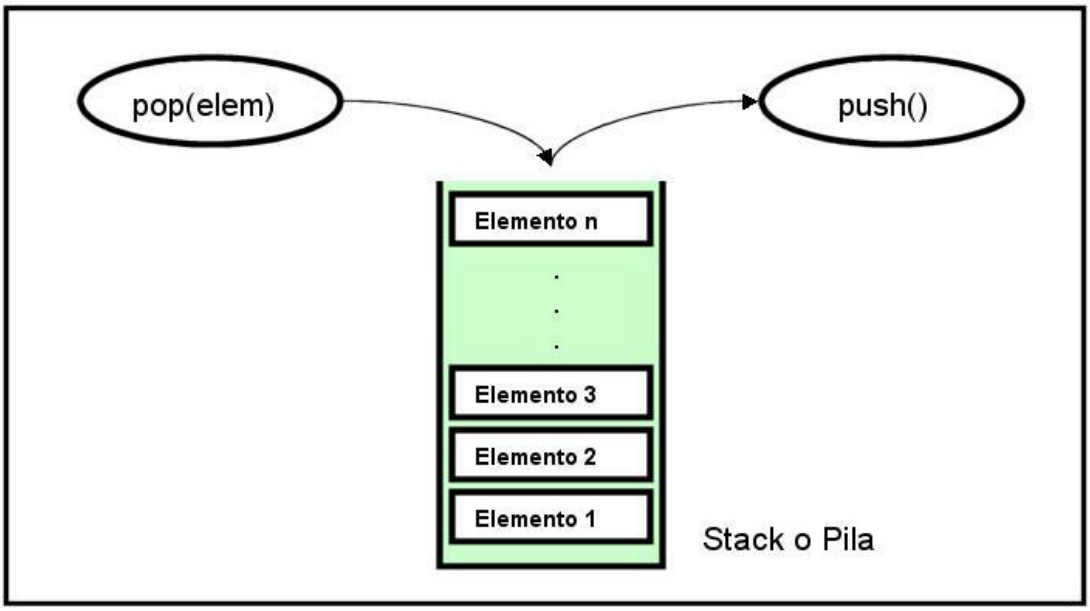
```
if (expression)
{
    istruzione
}
```

Ad esempio:

```
if(x > y)
{
     System.out.println("x is greater than y");
}
```

Esercizio 1

Uno Stack o Pila è una struttura dati gestita secondo la filosofia LIFO (Last In First Out) ovvero l'ultimo elemento ad essere inserito è il primo ad essere recuperato.



Disegnare e realizzare un oggetto Stack. L'oggetto Stack deve contenere al massimo 20 numeri interi e deve avere i due metodi:

```
void push(int)
int pop()
```

[9] I controlli di flusso verranno descritti in dettaglio nel capitolo successivo.

Il metodo push che ritorna un tipo void e prende come parametro di input un numero intero inserisce l'elemento in cima alla pila.

L'array deve essere inizializzato all'interno del metodo push(**int**). Si può controllare lo stato dello stack ricordando che una variabile reference non inizializzata punta all'oggetto *null*.

Soluzione al primo esercizio

La classe Stack dovrebbe essere qualcosa tipo:

c:\esercizi\cap3\Stack.java

```
1    class Stack
2    {
3        int data[];
4        int first;
5        void push(int i)
6        {
7            if(data == null)
8            {
9                first = 0;
10               data = new int[20];
11           }
12           if(first < 20)
13           {
14               data[first] = i;
15               first ++;
16           }
17       }
18       int pop()
19       {
20           if(first > 0)
21           {
22               first --;
23               return data[first];
24           }
25            return 0; // Bisogna tornare qualcosa
26       }
27   }
```

Le righe 3 e 4 contengono la dichiarazione dei dati membro della classe. In particolare utilizziamo un array di numeri interi per contenere i dati ed una variabile intera che mantiene traccia della prima posizione libera all'interno dell'array.

Dalla riga 5 alla riga 17 viene dichiarato il metodo ***void*** *push(**int** i)* all'interno del quale per prima cosa viene controllato se l'array è stato inizializzato (righe 7-11) ed eventualmente viene allocato come array di 20 numeri interi.

```
if(data == null)
{
    first = 0;
    data = new int[20];
}
```

Quindi viene effettuato il controllo per verificare se è possibile inserire elementi all'interno dell'array. In particolare essendo 20 il numero massimo di interi contenuti, mediante l'istruzione **if** il metodo verifica che la posizione puntata dalla variabile first sia minore di 20 (ricordiamo che in un array le posizioni sono identificate a partire da 0). Se l'espressione *first < 20* produce il valore **true**, il numero intero passato come parametro di input viene inserito nell'array nella posizione *first*. La variabile first viene quindi aggiornata in modo che punto alla prima posizione libera nell'array.

```
        if(first < 20)
        {
            data[first] = i;
            first ++;
        }
```

Le righe da 18 a 26 rappresentano la dichiarazione del metodo ***int*** *pop()* che recupera il primo elemento della pila e lo ritorna all'utente. Per prima cosa il metodo controlla all'interno di una istruzione **if** se first sia maggiore di 0 ovvero che esista almeno un elemento all'interno dello Stack. Se la condizione si verifica first viene modificata in modo da puntare all'ultimo elemento inserito il cui valore viene restituito mediante il comando **return**. In caso contrario il metodo ritorna il valore 0.

```
    int pop()
    {
        if(first > 0)
        {
            first --;
            return data[first];
        }
         return 0; // Bisogna tornare qualcosa
    }
```

A questo punto è possibile salvare il file contenente il codice sorgente e compilarlo. Alla fine del processo di compilazione avremo a disposizione un file contenente il bytecode della definizione di classe. Salviamo il file nella directory c:\esercizi\cap3\ con nome Stack.java. A questo punto possiamo quindi compilare il file:

```
C:\> set CLASSPATH=.;.\;c:\jdk1.3\lib\tools.jar;
C:\> set PATH=%PATH%;c:\jdk1.3\bin;
C:\> javac Stack.java
C:\>
```

Capitolo 4
Controllo di flusso e distribuzione di oggetti

Introduzione

Java eredita da C e C++ l'intero insieme di istruzioni per il controllo di flusso apportando solo alcune modifiche. In aggiunta Java introduce alcune nuove istruzioni necessarie alla manipolazione di oggetti.

Questo capitolo tratta le istruzioni condizionali, le istruzioni di loop, le istruzioni relative alla gestione dei package per l'organizzazione di classi e, l'istruzione import per risolvere la "posizione" delle definizioni di classi in altri file o package.

I package Java sono strumenti simili a librerie e servono come meccanismo per raggruppare classi o distribuire oggetti. L'istruzione ***import*** è una istruzione speciale utilizzata dal compilatore per determinare la posizione su disco delle definizioni di classi da utilizzare nella applicazione corrente .

Come C e C++ Java è indipendente dagli spazi ovvero l'indentazione del codice di un programma ed eventualmente l'uso di più di una riga di testo sono opzionali.

Istruzioni per il controllo di flusso

Espressioni ed istruzioni per il controllo di flusso forniscono al programmatore il meccanismo per decidere se e come eseguire blocchi di istruzioni condizionatamente a meccanismo decisionali definiti all'interno della applicazione.

Istruzioni per il controllo di flusso	
Istruzione	**Descrizione**
if	Esegue o no un blocco di codice a seconda del valore restituito da una espressione booleana.
if-else	Esegue permette di selezionare tra due blocchi di codice quello da eseguire a seconda del valore restituito da una espressione booleana.
switch	Utile in tutti quei casi in cui sia necessario decidere tra opzioni multiple prese in base al controllo di una sola variabile.
for	Esegue ripetutamente un blocco di codice.
while	Esegue ripetutamente un blocco di codice controllando il valore di una espressione booleana.
do-while	Esegue ripetutamente un blocco di codice controllando il valore di una espressione booleana.

Le istruzioni per il controllo di flusso sono riassunte nella tabella precedente ed hanno sintassi definita dalle regole di espansione definite nel terzo capitolo e riassunte qui di seguito..

istruzione --> {
istruzione
[istruzione]
}

OR

istruzione --> flow_control
istruzione

L'istruzione if

L'istruzione per il controllo di flusso **if** consente alla applicazione di decidere, in base ad una espressione booleana, se eseguire o no un blocco di codice. Applicando le regole di espansione definite, la sintassi di questa istruzione è la seguente:

```
if (boolean_expr)                    if (boolean_expr)
    istruzione1              -->     {
                                         istruzione;
                                         [istruzione]
                                     }
```

dove *boolean_expr* rappresenta una istruzione booleana valida. Di fatto, se *boolean_expr* restituisce il valore ***true***, verrà eseguito il blocco di istruzioni successivo, in caso contrario il controllo passerà alla prima istruzione successiva al blocco **if**. Un esempio di istruzione if è il seguente:

```
1   int x;
2   ....
3   if(x>10)
4   {
5           ....
6           x=0;
7           .....
8   }
9   x=1;
```

In questo caso, se il valore di x è strettamente maggiore di 10, verrà eseguito il blocco di istruzioni di **if** (righe 4-8), in caso contrario il flusso delle istruzioni salterà direttamente dalla riga 3 alla riga 9 del codice.

L'istruzione if-else

Una istruzione ***if*** può essere opzionalmente affiancata da una istruzione ***else***. Questa forma particolare dell'istruzione **if**, la cui sintassi è descritta qui di seguito, consente di decidere quale blocco di codice eseguire tra due blocchi di codice.

```
if (boolean_expr)                    if (boolean_expr)
        istruzione1          -->     {
else istruzione2                         istruzione;
                                         [istruzione]
                                     }
                                     else
                                     {
                                         istruzione;
                                         [istruzione]
                                     }
```

Se *boolean_expr* restituisce il valore ***true*** verrà eseguito il blocco di istruzioni di **if**, altrimenti il controllo verrà passato ad ***else*** e verrà eseguito il secondo blocco di istruzioni. Di seguito un esempio:

```
1   if(y==3)
2   {
3      y=12;
4   }
5   else
6   {
7      y=0;
8   }
```

In questo caso, se l'espressione booleana sulla riga 3 del codice ritorna valore true, allora verranno eseguite le istruzioni contenute nelle righe 2-4, altrimenti verranno eseguite le istruzioni contenute nelle righe 6-8.

Istruzioni if, if-else annidate

Una istruzione **if** annidata rappresenta una forma particolare di controllo di flusso in cui una istruzione **if** o **if-else** è controllata da un'altra istruzione **if** o **if-else**. Utilizzando le regole di espansione, in particolare concatenando ricorsivamente la terza con le definizioni di if e if-else:

istruzione -> flow_control *flow_control* *istruzione*	*flow_control ->if(espressione)* *istruzione*
	flow_control ->if(espressione) *istruzione* *else* *istruzione*

otteniamo la regola sintattica per costruire blocchi if annidati:

```
if(espressione)
{
    istruzione
    if (espressione2)
    {
        istruzione
        if (espressione3)
        {
            istruzione
            …….
        }
    }
}
```

Catene if-else-if

La forma più comune di if annidati è rappresentata dalla sequenza o catena ***if-else-if***. Questo tipo di concatenazione valuta una serie arbitraria di istruzioni booleane precedendo dall'alto verso il basso. Se almeno una delle condizioni restituisce il valore ***true*** verrà eseguito il blocco di istruzioni relativo. Se nessuna delle condizioni si dovesse verificare allora verrebbe eseguito il blocco ***else*** finale.

```
if(espressione)
{
    istruzione
}
else if (espressione2)
      {
         istruzione
      }
      else if (espressione3)
            {
               istruzione
               …….
            }
            else
            {
               istruzione
               …….
            }
```

Nell'esempio viene utilizzato un tipico caso in cui utilizzare questa forma di annidamento:

```
.....
int i = getUserSelection();
if(i==1)
{
    faiQualcosa();
}
else if(i==2)
      {
         faiQualcosAltro();
      }
      else
      {
         nonFareNulla ();
      }
   ....
```

Nell'esempio, la variabile di tipo int i prende il valore restituito come parametro di ritorno dal metodo *getUserSelection()*. La catena controlla quindi il valore di i ed esegue il metodo *faiQualcosa()* nel caso in cui i valga 1, *faiQualcosAltro()* nel caso in cui valga 2, *nonFareNulla()* in tutti gli altri casi.

L'istruzione switch

Java mette a disposizione una istruzione di controllo di flusso che, specializzando la catena if-else-if, semplifica la vita al programmatore.

L'istruzione ***switch*** è utile in tutti quei casi in cui sia necessario decidere tra opzioni multiple prese in base al controllo di una sola variabile. Questa istruzione può sembrare sicuramente ridondante rispetto alla forma precedente, ma sicuramente rende la vita del programmatore più semplice in fase di lettura del codice. La sintassi della istruzione è la seguente:

```
switch (espressione)
{
    case espr_costante:
                    istruzione1
                    break_opzionale
    case espr_costante:
                    istruzione2
                    break_opzionale
    case espr_costante:
                    istruzione3
                    break_opzionale
    ……..
    default:
                    istruzione4
}
```

espressione rappresenta ogni espressione valida che produca un intero e *espr_costante* una espressione che può essere valutata completamente al momento della compilazione. Quest'ultima per funzionamento può essere paragonata ad una costante. *Istruzione* è ogni istruzione Java come specificato dalle regole di espansione e ***break_opzionale*** rappresenta la inclusione opzionale della parala chiave ***break*** seguita da ";" .

Un esempio può aiutarci a comprendere il significato di questo costrutto. Consideriamo l'esempio del paragrafo precedente:

```
.....
int i = getUserSelection();
if(i==1)
{
    faiQualcosa();
}
else if(i==2)
      {
        faiQualcosAltro();
      }
      else
      {
        nonFareNulla ();
      }
    ....
```

Utilizzando l'istruzione **switch** possiamo riscrivere il blocco di codice nel modo seguente:

```
.....
int i = getUserSelection();
switch (i)
{
    case 1:
            faiQualcosa();
            break;
    case 2:
            faiQualcosAltro();
            break;
    default:
```

```
            nonFareNulla();
}
....
```

Se la variabile intera i valesse 1, il programma eseguirebbe il blocco di codice relativo alla prima istruzione ***case*** chiamando il metodo faiQualcosa(), troverrebbe l'istruzione **break** ed uscirebbe quindi dal blocco switch passando alla prossima istruzione del bytecode. Se invece i valesse 2, verrebbe eseguito il blocco contente la chiamata al metodo *faiQualcosAltro()* ed essendoci una istruzione **break** uscirebbe anche in questo caso dal blocco **switch**. Infine, per qualunque altro valore di i, l'applicazione eseguirebbe il blocco identificato dalla label **default**.

In generale, dopo che viene valutata l'espressione di **switch**, il controllo della applicazione salta al primo **case** tale che

espressione == espr_costante

ed esegue il relativo blocco di codice. Nel caso in cui il blocco sia terminato con una istruzione **break**, l'applicazione abbandona l'esecuzione del blocco switch saltando alla prima istruzione successiva al blocco, altrimenti il controllo viene eseguito sui blocchi **case** a seguire. Se nessun blocco case soddisfa la condizione ossia

espressione != espr_costante

la virtual machine controlla l'esistenza della label **default** ed esegue, se presente, solo il blocco di codice relativo ed esce da **switch**.

L'istruzione while

Una istruzione **while** permette la esecuzione ripetitiva di una istruzione utilizzando una espressione booleana per determinare se eseguire il blocco di istruzioni, eseguendolo quindi fino a che l'espressione booleana non restituisce il valore false. La sintassi per questa istruzione è la seguente:

```
while (espressione_booleana){
    istruzione
}
```

dove *espressione_booleana* è una espressione valida che restituisce un valore booleano. Per prima cosa, una istruzione while controlla il valore della espressione booleana. Se restituisce true verrà eseguito il blocco di codice di **while**. Alla fine della esecuzione viene nuovamente controllato il valore della espressione booleana per decidere se ripetere l'esecuzione del blocco di codice o passare il controllo della esecuzione alla prima istruzione successiva al blocco while.

Applicando le regole di espansione, anche in questo caso otteniamo la forma annidata:

```
while (espressione_booleana)
    while (espressione_booleana)
            istruzione
```

Il codice di esempio utilizza la forma annidata del ciclo while:

```
1       int i=0;
2       while(i<10)
3       {
4               j=10;
5               while(j>0)
6               {
7                   System.out.println("i="+i+"e j="+j);
8                   j--;
9               }
10              i++;
11      }
```

L'istruzione do-while

Una alternativa alla istruzione **while** è rappresentata dall'istruzione **do-while** a differenza della precedente, controlla il valore della espressione booleana alla fine del blocco di istruzioni. In questo caso quindi il blocco di istruzioni verrà eseguito sicuramente almeno una volta.

La sintassi di **do-while** è la seguente:

```
do {
   istruzione;
} while (espressione_booleana);
```

L'istruzione for

Quando scriviamo un ciclo, accade spesso la situazione in cui tre task distinti concorrono alla esecuzione del blocco di istruzioni. Consideriamo il ciclo di 10 iterazioni:

```
i=0;
while(i<10)
{
    faiQualcosa();
    i++;
}
```

Nel codice come prima cosa viene inizializzata una variabile per il controllo del ciclo, quindi viene eseguita una espressione condizionale per decidere sullo stato del ciclo, infine la variabile viene aggiornata in modo tale che possa determinare la fine del ciclo. Per semplificare la vita al programmatore, Java mette a disposizione l'istruzione **for** che include tutte queste operazioni nella stessa istruzione condizionale:

```
for(init_statement ; conditional_expr ; iteration_stmt){
    istruzione
}
```

che ha forma annidata:

```
for(init_statement ; conditional_expr ; iteration_stmt){
    for(init_statement ; conditional_expr ; iteration_stmt){
            istruzione
    }
}
```

init_statement rappresenta l'inizializzazione della variabile per il controllo del ciclo, conditional_expr l'espressione condizionale, iteration_stmt l'aggiornamento della variabile di controllo. In una istruzione for l'espressione condizionale viene sempre controllata all'inizio del ciclo. Nel caso in cui restituisca un valore false, il blocco di istruzioni non verrà mai eseguito. Per esempio, il ciclo realizzato nel primo esempio utilizzando il comando **while** può essere riscritto utilizzando il comando **for** :

```
for (int i=0 ; i<10 ; i++)
    faiQualcosa();
```

Istruzione for nei dettagli

L'istruzione **for** in Java come in C e C++ è una istruzione estremamente versatile inquanto consente di scrivere cicli di esecuzione utilizzando molte varianti alla forma descritta nel paragrafo precedente.

In pratica, il ciclo for consente di utilizzare zero o più variabili di controllo, zero o più istruzioni di assegnamento ed altrettanto vale per le espressioni booleane. Nella sua forma più semplice il ciclo for può essere scritto nella forma:

```
for( ;  ; ){
    istruzione
}
```

Questa forma non utilizza ne variabili di controllo, ne istruzioni di assegnamento ne tanto meno espressioni booleane. In un contesto applicativo realizza un ciclo infinito. Consideriamo ora il seguente esempio:

```
for (int i=0, j=10 ; (i<10 && j>0) ; i++, j--) {
    faiQualcosa();
}
```

Il ciclo descritto utilizza due variabili di controllo con due operazioni di assegnamento distinte. Sia la dichiarazione ed inizializzazione delle variabili di controllo che le operazioni di assegnamento utilizzando il carattere "," come separatore. Per concludere, la sintassi di questa istruzione può essere quindi descritta della regola generale:

```
for([init_stmt][,init_stmt] ; conditional_expr ; [iteration_stmt] [,iteration_stmt] ) {
    istruzione
}
```

Istruzioni di ramificazione

Il linguaggio Java consente l'uso di tre parole chiave che consentono di modificare in qualunque punto del codice il normale flusso della esecuzione della applicazione con effetto sul blocco di codice in esecuzione o sul metodo corrente. Queste parole chiave sono tre (come schematizzato nella tabella seguente) e sono dette istruzioni di "branching" o ramificazione.

Istruzioni di ramificazione	
Istruzione	**Descrizione**
break	Interrompe l'esecuzione di un ciclo evitando ulteriori controlli sulla espressione condizionale e ritorna il controllo alla istruzione successiva al blocco attuale.
continue	Salta un blocco di istruzioni all'interno di un ciclo e ritorna il controllo alla espressione booleana che ne governa l'esecuzione.
return	Interrompe l'esecuzione del metodo attuale e ritorna il controllo al metodo chiamante.

L'istruzione break

Questa istruzione consente di forzare l'uscita da un ciclo aggirando il controllo sulla espressione booleana e provocandone l'uscita immediata in modo del tutto simile a quanto già visto parlando della istruzione **switch**. Per comprenderne meglio il funzionamento torniamo per un attimo al ciclo for già visto nei paragrafi precedenti:

```
for (int i=0; i<10 ; i++) {
    faiQualcosa();
}
```

Utilizzando l'istruzione **break**, possiamo riscrivere il codice evitando di inserire controlli all'interno del ciclo for come segue:

```
for (int i=0; ; i++) {
    if(i==10) break;
    faiQualcosa();
}
```

L'esecuzione del codice produrrà esattamente gli stessi risultati del caso precedente.

L'uso di questa istruzione è tipicamente legata a casi in cui sia necessario poter terminare l'esecuzione di un ciclo a prescindere dai valori delle variabili di controllo utilizzate. Queste situazioni si verificano in quei casi in cui sia impossibile utilizzare un parametro di ritorno come operando all'interno della espressione booleana che controlla l'esecuzione del ciclo, ed è pertanto necessario implementare all'interno del blocco meccanismi specializzati per la gestione di questi casi.

Un esempio tipico è quello di chiamate a metodi che possono generare eccezioni[10] ovvero notificare errori di esecuzione in forma di oggetti. In questi casi utilizzando il comando break è possibile interrompere l'esecuzione del ciclo non appena venga catturato l'errore.

L'istruzione continue

A differenza del caso precedente questa istruzione non interrompe l'esecuzione del ciclo di istruzioni, ma al momento della chiamata produce un salto alla parentesi graffa che chiude il blocco restituendo il controllo alla espressione booleana che ne determina l'esecuzione. Un esempio può aiutarci a chiarire le idee:

[10] Le eccezioni verranno trattate in dettaglio a breve.

```
1       int i=-1;
2       int pairs=0;
3       while(I<20)
4       {
5               i++;
6               if((i%2)!=0) continue;
7               pairs ++;
8       }
```

Le righe di codice descritte calcolano quante occorrenze di interi pari ci sono in una sequenza di interi compresa tra 1 e 20 e memorizzano il valore in una variabile di tipo **int** chiamata "pairs". Il ciclo while è controllato dal valore della variabile i inizializzata a –1. La riga 6 effettua una controllo sul valore di i: nel caso in cui i rappresenti un numero intero dispari viene eseguito il comando **continue** ed il flusso ritorna alla riga 3. In caso contrario viene aggiornato il valore di pairs.

L'istruzione return

Questa istruzione rappresenta l'ultima istruzione di ramificazione e può essere utilizzata per terminare l'esecuzione del metodo corrente tornando il controllo al metodo chiamante. Return può essere utilizzata in due forme:

return *valore;*
return*;*

La prima forma viene utilizzata per consentire ad un metodo di ritornare valori al metodo chiamante e pertanto deve ritornare un valore compatibile con quello dichiarato nella definizione del metodo. La seconda può essere utilizzata per interrompere l'esecuzione di un metodo qualora il metodo ritorni un tipo **void**.

Package Java

I package sono meccanismi per raggruppare definizioni di classe in librerie, similmente ad altri linguaggi di programmazione. Il meccanismo è provvisto di una struttura gerarchica per l'assegnamento di nomi alle classi in modo da evitare eventuali collisioni in caso in cui alcuni programmatori usino lo stesso nome per differenti definizioni di classe.

Oltre a questo, sono molti i benefici nell'uso di questo meccanismo: primo, le classi possono essere mascherate all'interno dei package implementando l'incapsulamento anche a livello di file. Secondo, le classi di un package possono condividere dati e metodi con classi di altri package. Terzo, i package forniscono un meccanismo efficace per distribuire oggetti.

In questo capitolo verrà mostrato in dettaglio solamente il meccanismo di raggruppamento. Gli altri aspetti verranno trattati nei capitoli successivi.

Assegnamento di nomi a package

I package combinano definizioni di classi in un unico archivio la cui struttura gerarchica rispetta quella del file system. I nomi dei package sono separati tra loro da punto. La classe Vector ad esempio fa parte del package *java.util* archiviato nel file tools.jar .

Secondo le specifiche, il linguaggio riserva tutti i package che iniziano con *java* per le classi che sono parte del linguaggio. Questo significa che nuove classi definite da un utente devono essere raggruppate in package con nomi differenti da questo.

Le specifiche suggeriscono inoltre che package generati con classi di uso generale debbano iniziare con il nome della azienda proprietaria del codice. Ad esempio se la *pippo corporation* avesse generato un insieme di classi dedicate al calcolo statistico, le classi dovrebbero essere contenute in un package chiamato ad esempio *pippo.sta*t .

Una volta definito il nome di un package, affinché una classe possa essere archiviata al suo interno, è necessario aggiungere una istruzione *package* all'inizio del codice sorgente che definisce la classe. Per esempio all'inizio di ogni file contenente i sorgenti del package pippo.stat è necessario aggiungere la riga:

package pippo.stat;

Questa istruzione non deve assolutamente essere preceduta da nessuna linea di codice. In generale, se una classe non viene definita come appartenente ad un package, il linguaggio per definizione assegna la classe ad un particolare package senza nome.

Creazione dei package su disco

Una volta definito il nome di un package, deve essere creata su disco la struttura a directory che rappresenti la gerarchia definita dai nomi. Ad esempio, le classi appartenenti al package java.util devono essere memorizzate in una gerarchia di directory che termina con *java/util* localizzata in qualunque punto del disco (ex. C:/classes/java/util/ schematizzato nella *Figura 4-1*).

Per trovare le classi contenute in un package, Java utilizza la variabile di ambiente CLASSPATH che contiene le informazioni per puntare alla root del nome del package e non direttamente alle classi all'interno del package. Per esempio se il nome del package è java.util e le classi sono memorizzate nella directory C:/classes/java/util/*, allora CLASSPATH dovrà includere la directory C:/classes/.

Sotto sistemi microsoft, una variabile CLASSPATH ha tipicamente una forma del tipo:

CLASSPATH = c:\java\lib\tools.jar;d:\java\import;.;.\;

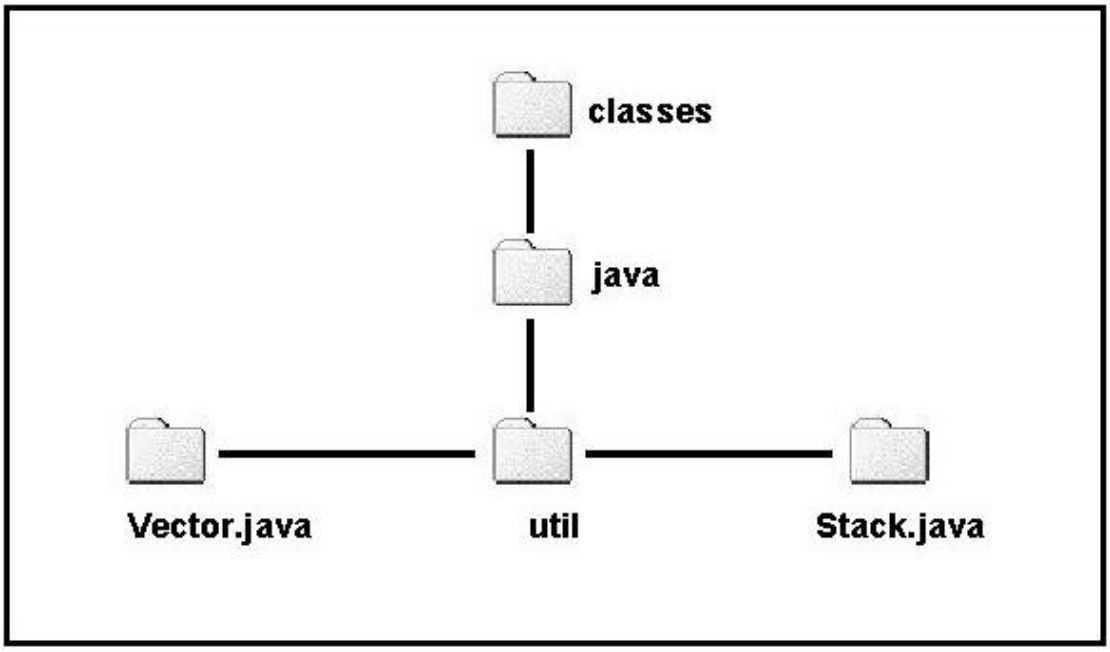


Figura 4-1 : *Struttura di un package Java*

Il modificatore public

Di default, la definizione di una classe Java può essere utilizzata solo dalle classi all'interno del suo stesso package. Per esempio, assumiamo di aver scritto una applicazione in un file Appo.java non appartenente a nessun package e, supponiamo che esista una classe Stack appartenente al package app.stack . Per definizione, Appo.java non potrà accedere alla definizione di classe di Stack.

```
package app.stack;

class Stack
{
    int data[];
    int ndata;
    void push(int i)
    {
            ……
    }
    int pop()
    {
        ……..
    }
}
```

Java richiede al programmatore di esplicitare quali classi e quali membri possano essere utilizzati all'esterno del package. A questo scopo Java riserva il modificatore **public** da utilizzare prima della dichiarazione della classe o di un membro come mostrato nella nuova versione della classe Stack detta ora classe pubblica.

```
package app.stack;

public class Stack
{
    int data[];
    int ndata;
    public void push(int i)
    {
            ……
    }
    public int pop()
    {
        ……..
    }
}
```

Le specifiche del linguaggio richiedono che il codice sorgente di classe pubblica sia memorizzata in un file avente lo stesso nome della classe (incluse maiuscole e minuscole), ma con estensione ".java". Come conseguenza alla regola, può esistere solo una classe pubblica per ogni file di sorgente. Questa regola è rinforzata dal compilatore che scrive il bytecode di ogni classe in un file avente lo stesso nome della classe (incluse maiuscole e minuscole), ma con estensione ".class".

Lo scopo di questa regola è quello di semplificare la ricerca di sorgenti e bytecode da parte del programmatore. Per esempio supponiamo di avere tre classi A, B e C in un file unico. Se A fosse la classe pubblica (solo una lo può essere), il codice sorgente di tutte e tre le classi dovrebbe trovarsi all'interno di un file A.java .

Quando A.java verrà compilato, il compilatore creerà una classe per ogni classe nel file: A.class, B.class e C.class .

Questa organizzazione per quanto contorta, ha un senso logico. Se come detto la classe pubblica è l'unica a poter essere eseguita da altre classi all'esterno del package, le altre classi rappresentano solo l'implementazione di dettagli non necessarie al di fuori del package.

Per concludere non mi resta che ricordare che, anche se una classe non pubblica può essere definita nello stesso file di una classe pubblica, questo non è strettamente necessario e sarà compito del programmatore scegliere in che modo memorizzare le definizioni delle classi all'interno di un package.

L'istruzione import

Il runtime di Java fornisce un ambiente completamente dinamico. Le classi non vengono caricate fino a che non sono referenziate per la prima volta durante l'esecuzione della applicazione. Questo consente di ricompilare singole classi senza dover ricaricare grandi applicazioni.

Dal momento che ogni classe Java è memorizzata in un suo file, la virtual machine può trovare i file binari .class appropriati cercando nelle directory specificate nelle directory definite nella variabile d'ambiente CLASSPATH. Inoltre, dal momento che le classi possono essere organizzate in package, è necessario specificare a quale package una classe appartenga pena l'incapacità della virtual machine di trovarla.

Un modo per indicare il package a cui una classe appartiene è quello di specificare il package ad ogni chiamata alla classe ossia utilizzando nomi qualificati[11]. Riprendendo la nostra classe Stack appartenente al package app.stack, il suo nome qualificato sarà app.stack.Stack .

L'uso di nomi qualificati non è sempre comodo soprattutto per package organizzati con gerarchie a molti livelli. Per venire in contro al programmatore,Java consente di specificare una volta per tutte il nome qualificato di una classe all'inizio del file utilizzando la parola chiave **import**.

L'istruzione **import** ha come unico effetto quello di identificare univocamente una classe e quindi di consentire al compilatore di risolvere nomi di classe senza ricorrere ogni volta a nomi qualificati. Grazie all'istruzione

```
import app.stack.Stack;
```

una applicazione sarà in grado di risolvere il nome di Stack ogni volta che sia necessario semplicemente utilizzando il nome di classe *Stack*. Capita spesso di dover però utilizzare un gran numero di classi appartenenti ad un unico package. Per questi casi l'istruzione **import** supporta l'uso di un carattere fantasma :

```
import app.stack.*;
```

che risolve il nome di tutte le classi pubbliche di un package (app.stack nell'esempio). Questa sintassi non consente altre forme e non può essere utilizzata per caricare solo porzioni di package. Per esempio la forma **import** app.stack.S* non è consentita.

[11] Tecnicamente, in Java un nome qualificato è un nome formato da una serie di identificatori separati da punto per identificare univocamente una classe.

Laboratorio 4
Controllo di flusso e distribuzione di oggetti

Esercizio 1

Utilizzando le API Java, scrivere un semplice programma che utilizzando la classe java.util.Date stampi a video una stringa del tipo:

Oggi è : giorno_della_settimana e sono le ore hh:mm.
Siamo nel mese di mese_corrente nell'anno anno_corrente

Esercizio 2

Utilizzando la classe stack definita in precedenza, scrivere un ciclo while che stampi tutti gli interi pari da 1 a 13 inserendoli nello stack. La definizione di classe di stack dovrà essere memorizzato nel package "esempi.lab4".

Soluzione al primo esercizio

La classe Date appartenente al package java.util definisce il concetto di “istante temporale” rappresentato in forma di millisecondi trascorsi a partire da una data di riferimento. Attraverso metodi opportuni, la classe fornisce la possibilità di interpretare l’istante rappresentato in forma di anno, giorno, mese, ora, minuto e secondo utilizzando le seguenti convenzioni:

Un anno è rappresentato da un intero y = anno_attuale-1900;
Un mese è rappresentato da un intero da 0 a 11;
Una data del mese è rappresentato da un intero da 1 a 31;
Un giorno della settimana è rappresentato da un intero da 0 a 6;
Un ora è rappresentata da un intero da 0 a 23;
Un minuto è rappresentato da un intero da 0 a 59;
Un secondo è rappresentato da un intero da 0 a 61.

La soluzione fornita di seguito, fa uso dei metodi *getDay()*, *getHours()*, *getMinutes()*, *getMonth()*, *getYear()*. Questi metodi, già esistenti dalla versione 1.1 di java sono stati “deprecati” nelle ultime versioni del JDK. Il termine deprecato sta ad indicare che i metodi sono supportati nella corrente versione del JDK, ma saranno abbandonati nelle versioni future poiché sostituiti da nuovi.

Nel caso in cui si utilizzi l’ultima versione del compilatore java, al termine del processo di compilazione comparirà un messaggio di tipo “*uses or overrides a deprecated API.*” . Tale messaggio rappresenta solo un avvertimento e non comporta la mancata generazione del bytecode.

Qui di seguito una possibile soluzione al nostro esercizio:

c:\esercizi\capitolo4\esercizio1.java

```
1       import java.util.*;
2
3       class esercizio1
4       {
5           public static void main(String args[])
6           {
7                   Date d = new Date() ;
8                   String mesi[] = new String[12]
9                   String giorni[] = new String[7];
10
11                  giorni [0] = "Lunedi'";
12                  giorni [1] = "Martedi'";
13                  giorni [2] = "Mercoledi'";
14                  giorni [3] = "Giovedi'";
15                  giorni [4] = "Venerdi'";
16                  giorni [5] = "Sabato'";
18                  giorni [6] = "Domenica'";
19
20                   mesi [0] = "Gennaio"
21                  mesi [1] = "Febbraio"
22                  mesi [2] = "Marzo"
23                  mesi [3] = "Aprile"
24                  mesi [4] = "Maggio"
25                  mesi [5] = "Giugno"
26                  mesi [6] = "Luglio"
27                  mesi [7] = "Agosto"
```

```
28                      mesi [8] = "Settembre"
29                      mesi [9] = "Ottobre"
30                      mesi [10] = "Novembre"
31                      mesi [11] = "Dicembre"
32
33                      System.out.println("Oggi e' : " + giorni[d.getDay()]
34                              + " e sono le ore " + d.getHours()+ ":"
35                              + d.getMinutes());
36                      System.out.println("Siamo nel mese di "
37                              + mesi[d.getMonth()] + " nell'anno "
38                              + (d.getYear()+1900));
39              }
40      }
```

Le righe 7,8,9 del codice sorgente rappresentano il blocco di dichiarazione degli oggetti e delle strutture dati che utilizziamo. In particolare vengono dichiarati due array di stringhe, uno di 12 elementi e l'altro di 7, che ci serviranno alla conversione da rappresentazione intera a rappresentazione estesa in forma di stringa rispettivamente del mese restituito dal metodo *getMonth()* e del giorno della settimana restituito dal metodo *getDay()*.

```
7                       Date d = new Date() ;
8                       String mesi[] = new String[12]
9                       String giorni[] = new String[7];
```

Le righe 11-31 contengono il codice di inizializzazione dei due array dichiarati in precedenza.

```
11                      giorni [0] = "Lunedi'";
12                      giorni [1] = "Martedi'";
13                      giorni [2] = "Mercoledi'";
14                      giorni [3] = "Giovedi'";
15                      giorni [4] = "Venerdi'";
16                      giorni [5] = "Sabato'";
18                      giorni [6] = "Domenica'";
19
20                       mesi [0] = "Gennaio"
21                      mesi [1] = "Febbraio"
22                      mesi [2] = "Marzo"
23                      mesi [3] = "Aprile"
24                      mesi [4] = "Maggio"
25                      mesi [5] = "Giugno"
26                      mesi [6] = "Luglio"
27                      mesi [7] = "Agosto"
28                      mesi [8] = "Settembre"
29                      mesi [9] = "Ottobre"
30                      mesi [10] = "Novembre"
31                      mesi [11] = "Dicembre"
```

Infine, le righe 33-38 contengono le chiamate alla classe System per la stampa a terminale dell'output come richiesto da specifiche.

```
33                      System.out.println("Oggi e' : " + giorni[d.getDay()]
34                              + " e sono le ore " + d.getHours()+ ":"
35                              + d.getMinutes());
36                      System.out.println("Siamo nel mese di "
37                              + mesi[d.getMonth()] + " nell'anno "
38                              + (d.getYear()+1900));
```

Siamo arrivati al momento della compilazione del codice sorgente: salviamo il codice sorgente nella directory *c:\esercizi\capitolo4* utilizzando il nome esercizio1.java e lanciamo il compilatore Java.

```
C:\>set CLASSPATH=.;.\;c:\jdk1.3\lib\tools.jar;
C:\>set PATH=%PATH%;c:\jdk1.3\bin;
C:>cd esercizi\capitolo4
C:\esercizi\capitolo4>javac esercizio1.java
Note: esercizio1.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

Infine eseguiamo la nostra applicazione:

```
C:\esercizi\capitolo4>java esercizio1
Oggi e' : Martedi' e sono le ore 21:10
Siamo nel mese di Aprile nell'anno 2001

C:\esercizi\capitolo4>
```

Soluzione al secondo esercizio

Scopo del secondo esercizio è quello di fissare il concetto di organizzazione del codice della applicazione mediante package. Procederemo quindi alla definizione del nome dei package ed alla loro creazione.

Come da specifiche la definizione di classe di stack dovrà essere memorizzato nel package “esempi.lab4. Una volta definito il nome di un package, deve essere creata su disco la struttura a directory che rappresenti la gerarchia definita dai nomi. Supponendo di lavorare all’interno della directory *c:\esercizi\capitolo4*\, il primo passo sarà quello di creare l’albero delle sottodirectory così come definito dalla nomenclatura:

```
C:\esercizi\capitolo4> mkdir esempi
C:\esercizi\capitolo4> cd esempi
C:\esercizi\capitolo4> mkdir lab4
C:\esercizi\capitolo4>
```

La classe Stack, come già definita nei capitoli precedenti e riportata di seguito mette a disposizione del programmatore i due metodi *push(**int**)* e *pop()* che rispettivamente inseriscono un intero all’interno dello stack e restituiscono l’ultimo elemento intero inserito all’interno dello stack; l’unica modifca da apportare al codice è relativa alla archiviazione del file Stack.java all’interno del package. A tal fine utilizziamo il modificatore **package** aggiungendo prima della definizione di classe la riga ***package** esempi.lab4.*

Possiamo infine salvare il codice della classe nel file Stack.java all’interno della directory *c:\esercizi\capitolo4\esempi\lab4* precedentemente creata.

c:\esercizi\capitolo4\esempi\lab4\Stack.java

```
package esempi.lab4;

/**
 * Rappresentazione dell'oggetto stack. Esercizio 2 Laboratorio 4
 * Creation date: (02/05/2001 11.52.46)
 * @author: Massimiliano Tarquini
 */
class Stack
{
	int data[];
	int first;
	int pop()
	{
		if (first > 0)
		{
			first--;
			return data[first];
		}
		return 0; // Bisogna tornare qualcosa
	}
	void push(int i)
	{
		if (data == null)
		{
			first = 0;
			data = new int[20];
		}
		if (first < 20)
		{
			data[first] = i;
			first++;
		}
	}
}
```

Il passo successivo è quello di creare la classe contenente il metodo main() della nostra applicazione contenente il ciclo while che stampi tutti gli interi pari da 1 a 13 inserendoli nello stack. Anche questa classe verrà archiviata nel package esempi.lab4 .

c:\esercizi\capitolo4\esempi\lab4\esercizio2.java

```
1	package esempi.lab4;
2
3	import esempi.lab4.*;
4
5	class esercizio2
6	{
7		public static void main(String args[])
8		{
9			Stack a = new Stack();
10			int i;
11
12			i=1;
13			while(i<=13)
14			{
15				if(i % 2 !=0)
```

```
16                      {
18                              System.out.println(i);
19                              a.push(i);
20                       }
21                       i++;
22                  }
23              }
24          }
```

E' arrivato il momento di compilare la nostra applicazione ed eseguire il nostro test.

```
C:\esercizi\capitolo4> set CLASSPATH=%CLASSPATH%;c:\esercizi\capitolo4\;
C:\esercizi\capitolo4>cd esercizi\capitolo4\esempi\lab4
C:\esercizi\capitolo4\esempi\lab4>set PATH=%PATH%;c:\jdk1.3\bin;
C:\esercizi\capitolo4\esempi\lab4>javac esercizio2.java
C:\esercizi\capitolo4\esempi\lab4>java esempi.lab4.esercizio2
1
3
5
7
9
11
13

C:\
```

Capitolo 5
Incapsulamento

Introduzione

L'incapsulamento di oggetti è il processo di mascheramento dei dettagli dell'implementazione ad altri oggetti per evitare riferimenti incrociati. I programmi scritti con questa tecnica risultano molto più leggibili e limitano i danni dovuto alla propagazione di un bug.

Una analogia con il mondo reale è rappresentata dalle carte di credito. Chiunque sia dotato di carta di credito può eseguire una serie di operazioni bancarie attraverso lo sportello elettronico. Una carta di credito, non mostra all'utente le modalità con cui si è messa in comunicazione con l'ente bancario o ha effettuato transazioni sul conto corrente, semplicemente si limita a farci prelevare la somma richiesta tramite una interfaccia utente semplice e ben definita. In altre parole, una carta di credito maschera il sistema all'utente che potrà prelevare denaro semplicemente conoscendo l'uso di pochi strumenti come la tastiera numerica ed il codice pin. Limitando l'uso della carta di credito ad un insieme limitato di operazioni si può: primo, proteggere il nostro conto corrente. Secondo, impedire all'utente di modificare in modo irreparabile dati o stati interni della carta di credito.

Uno degli scopi primari di un disegno object oriented, dovrebbe essere proprio quello di fornire all'utente un insieme di dati e metodi che danno il senso dell'oggetto in questione. Questo è possibile farlo senza esporre le modalità con cui l'oggetto tiene traccia dei dati ed implementa il corpo (metodi) dell'oggetto.

Nascondendo i dettagli, possiamo assicurare a chi utilizza l'oggetto che ciò che sta utilizzando è sempre in uno stato consistente a meno di bug dell'oggetto stesso. Uno stato consistente è uno stato permesso dal disegno di un oggetto. E' però importante notare che uno stato consistente non corrisponde sempre allo stato spettato dall'utente dell'oggetto. Se infatti l'utente trasmette all'oggetto parametri errati, l'oggetto si troverà in uno stato consistente, ma non nello stato desiderato.

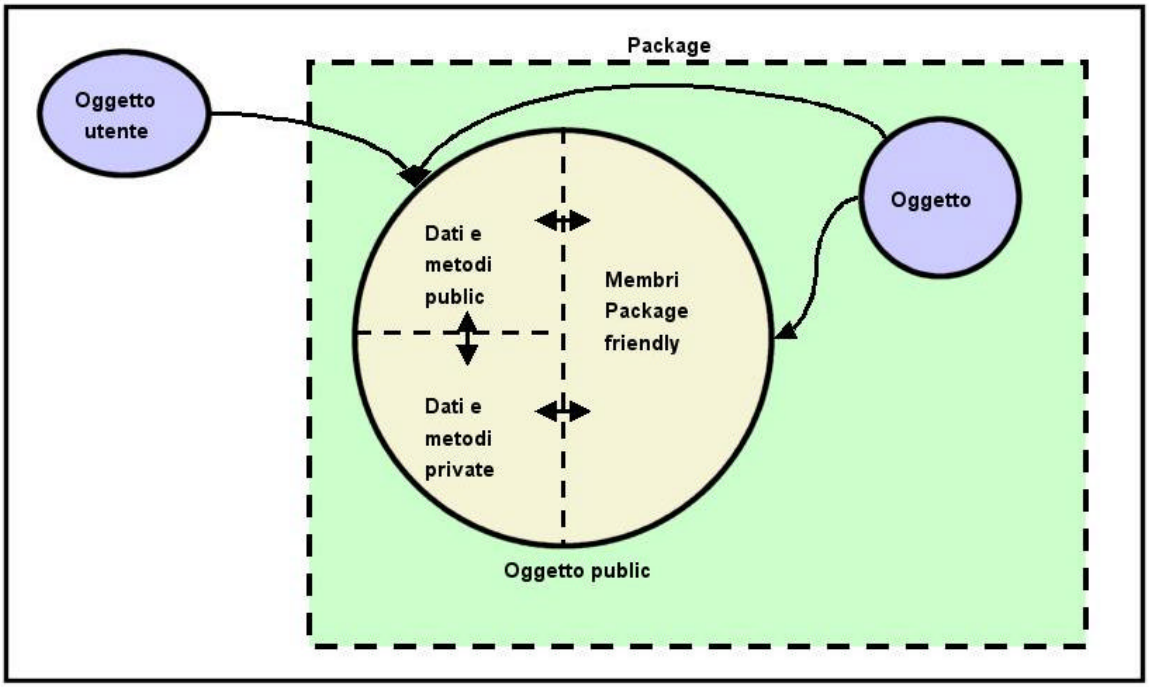


Figura 5-1 : *Modificatori public e private*

Modificatori public e private

Java fornisce supporto per l'incapsulamento a livello di linguaggio mediante i modificatori **public** e **private** da utilizzare al momento della dichiarazione di variabili e metodi. I membri di una classe o l'intera classe, definiti **public,** sono liberamente accessibili da ogni classe utilizzata nella applicazione.

I membri di una classe definiti **private** possono essere utilizzati sono dai membri della stessa classe. I membri privati mascherano i dettagli della implementazione di una classe.

Membri di una classe non dichiarati né **public**, né **private** saranno per definizione accessibili a tutte le classi dello stesso package. Questi membri o classi sono comunemente detti "**package friendly**". La regola è stata schematizzata nella *Figura 5-1*.

Private

Il modificatore **private** realizza incapsulamento a livello di definizione di classe e serve a definire membri che devono essere utilizzati solo da altri membri della stessa classe di definizione. L'intento di fatto è quello di nascondere porzioni di codice della classe che non devono essere utilizzati da altre classi.

Un membro privato può essere utilizzato da di qualsiasi membro statico, e non, della stessa classe di definizione con l'accorgimento che i membri statici possono solo utilizzare membri (dati od oggetti) statici o entità di qualunque tipo purché esplicitamente passate per parametro.

Per dichiarare un membro privato si utilizza la parola chiave ***private*** anteposta alla dichiarazione di un metodo o di un dato:

private *identificatore var_name;*

oppure nel caso di metodi:

```
private return_type  method_name(arg_type name [,arg_type name] )
{
    istruzioni
}
```

Public

Il modificatore **public** consente di definire classi o membri di una classe visibili da qualsiasi classe all'interno dello stesso package e non. Public deve essere utilizzato per definire l'interfaccia che l'oggetto mette a disposizione dell'utente. Tipicamente metodi membro **public** utilizzano membri **private** per implementare le funzionalità dell'oggetto.

Per dichiarare una classe od un membro pubblico si utilizza la parola chiave ***public*** anteposta alla dichiarazione :

public *identificatore var_name;*

nel caso di metodi:

```
public  return_type  method_name(arg_type name [,arg_type name] )
{
    istruzioni
}
```

infine nel caso di classi:

public class *ObjectName*
{
    data_declarations
    method_declarations
}

Il modificatore protected

Un altro modificatore messo a disposizione dal linguaggio Java è **protected**. Membri di una classe dichiarati protected possono essere utilizzati sia dai membri della stessa classe che da altre classi purché appartenenti allo stesso package (*Figura 5-2*).

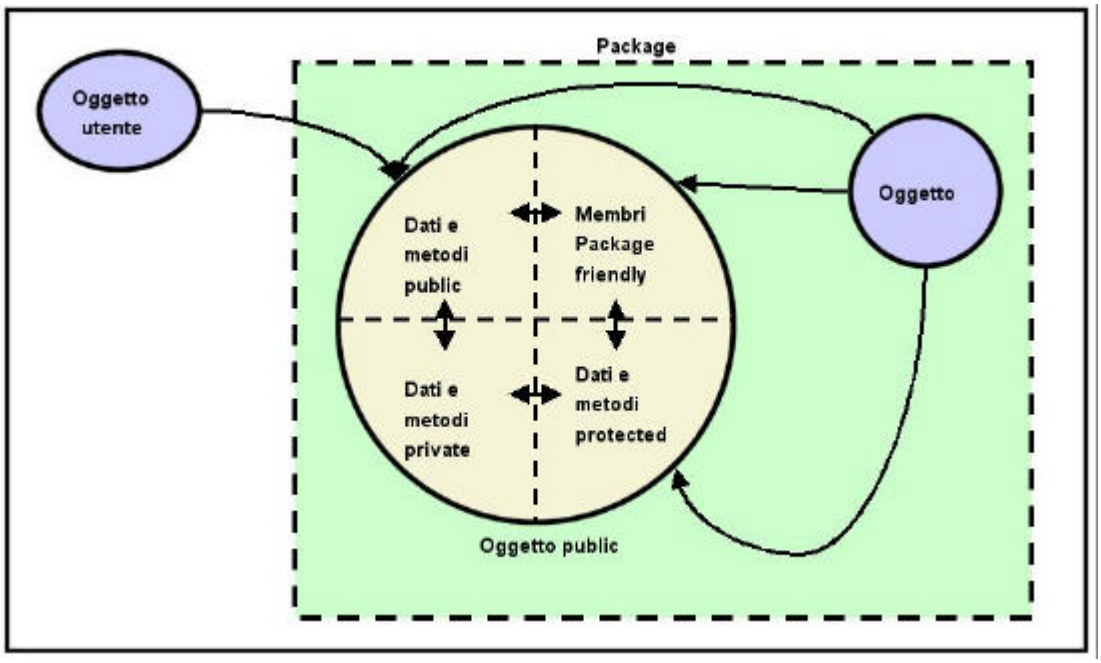


Figura 5-2 : *Modificatore protected*

Per dichiarare un membro protected si utilizza la parola chiave ***protected*** anteposta alla dichiarazione :

protected *identificatore var_name;*

nel caso di metodi:

protected *return_type method_name(arg_type name [,arg_type name])*
{
    istruzioni
}

Di questo modificatore torneremo a parlarne nei dettagli nel prossimo capitolo dove affronteremo il problema della ereditarietà.

Un esempio di incapsulamento

Nell'esempio mostrato di seguito, i dati della classe *Impiegato* (*nome*, e *affamato*) sono tutti dichiarati privati. Questo previene la lettura o peggio la modifica del valore dei dati da parte di *DatoreDiLavoro*.

D'altra parte, la classe è dotata di metodi pubblici (*haiFame()* e *nome()*), che consentono ad altre classi di accedere al valore dei dati privati. Nel codice sorgente, l'uso dei modificatori crea una simulazione ancora più realistica limitando l'azione di *DatoreDiLavoro* nella interazione *DatoreDiLavoro / Impiegato*. Ad esempio, *DatoreDiLavoro* non può cambiare il nome di *Impiegato*.

../javanet/mattone/cap5/Impiegato.java

```
package javanet.mattone.cap5;

public class Impiegato
{
    private String nome;
    private boolean affamato=true;
    public boolean haiFame()
    {
        return affamato;
    }
    public String nome()
    {
        return nome;
    }
    public void vaiAPranzo(String luogo)
    {
        // mangia
        affamato = false;
    }
}
```

../javanet/mattone/cap5/DatoreDiLavoro.java

```
package javanet.mattone.cap5;

public class DatoreDiLavoro
{
    public void siiCorrettoConImpiegato(Impiegato impiegato)
    {
        // if (person.Hungry) è una chiamata illegale perché Hungry è private
        // person.Hyngry = true  è un assegnamento illegale
        if (impiegato.haiFame())
        {
            impiegato.vaiAPranzo("Ristorante sotto l''ufficio");
        }
    }
}
```

L'operatore new

Per creare un oggetto attivo dalla sua definizione di classe, Java mette a disposizione l'operatore **new**. Questo operatore è paragonabile alla **malloc** in C, ed è identico allo stesso operatore in C++.

New oltre a generare un oggetto, consente di assegnargli lo stato iniziale ritornando un riferimento (indirizzo di memoria) al nuovo oggetto che può essere memorizzata in una variabile reference di tipo compatibile mediante l'operatore di assegnamento "=".

La responsabilità della gestione della liberazione della memoria allocata per l'oggetto non più in uso è del il garbage collector. Per questo motivo, a differenza di C++ Java non prevede nessun meccanismo esplicito per distruggere un oggetto creato.

La sintassi dell'operatore **new** prevede un tipo seguito da un insieme di parentesi. Le parentesi indicano che per creare l'oggetto verrà chiamata un metodo chiamato **costruttore**, responsabile della inizializzazione dello stato dell'oggetto (*Figura 5-3*).

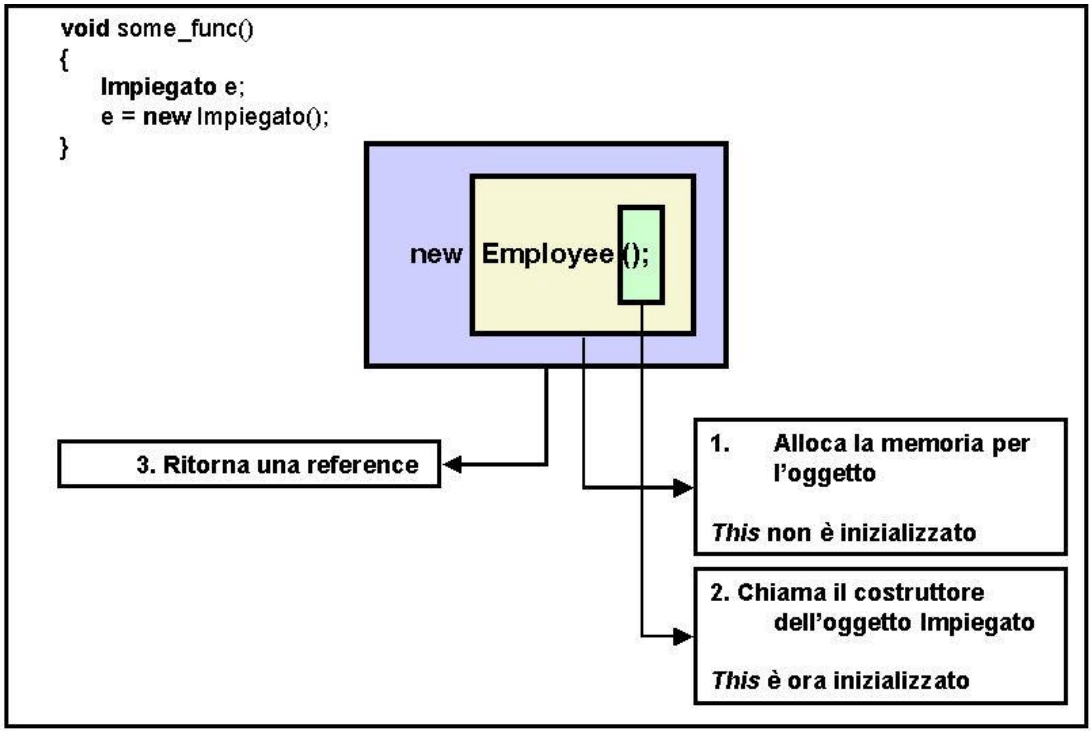


Figura 5-3 : *L'operatore new*

Costruttori

Tutti i programmatori, esperti e non, conoscono il pericolo che costituisce una variabile non inizializzata. Fare un calcolo matematico con una variabile intera non inizializzata può generare risultati errati.

In una applicazione object oriented, un oggetto è una entità molto più complessa di un tipo primitivo come **int** e, l'errata inizializzazione dello stato dell'oggetto può essere causa della terminazione prematura della applicazione o della generazione di bug intermittenti difficilmente controllabili.

In molti altri linguaggi di programmazione, il responsabile della inizializzazione delle variabili è il programmatore. In Java questo è impossibile dal momento che potrebbero essere dati membro privati dell'oggetto e quindi inaccessibili all'oggetto utente.

I costruttori sono metodi speciali chiamati quando viene creata una nuova istanza di classe e servono ad inizializzare lo stato iniziale dell'oggetto. Questi metodi hanno lo stesso nome della classe di cui sono membro e non restituiscono nessun tipo. Se una classe non è provvista di costruttore, Java ne utilizza uno speciale di default che

non fa nulla. Dal momento che il linguaggio garantisce la chiamata al costruttore ad ogni instanziamento di un oggetto, un costruttore scritto intelligentemente garantisce che tutti i dati membro vengano inizializzati. Nella nuova versione della classe Impiegato, il costruttore viene dichiarato esplicitamente dal programmatore e si occupa di impostare lo stato iniziale dei dati membro privati:

../javanet/mattone/cap5/second/Impiegato.java

```
package javanet.mattone.cap5.second;

public class Impiegato
{
    private String nome;
    private boolean affamato;

    public Impiegato haiFame()
    {
            affamato=true;
            nome="Massimiliano";
    }

    public boolean haiFame()
    {
            return affamato;
    }

    public String nome()
    {
            return nome;
    }

    public void vaiAPranzo(String luogo)
    {
            // mangia
            affamato = false;
    }
}
```

Java supporta molte caratteristiche per i costruttori, ed esistono molte regole per la loro creazione.

Un esempio di costruttori

In questo esempio, la definizione dell'oggetto Stack contiene un solo costruttore che non prende argomenti ed imposta la dimensione massima dello stack a 10 elementi.

../javanet/mattone/cap5/Stack.java

```
package javanet.mattone.cap5;
public class Stack
{
    private int maxsize;
    private int data[];
    private int first;
```

```
    public Stack)
    {
        maxsize = 10;
        data = new int[10];
        first=0;
    }

    int pop()
    {
        if (first > 0)
        {
            first--;
            return data[first];
        }
        return 0; // Bisogna tornare qualcosa
    }
    void push(int i)
    {

        if (first < maxsize)
        {
            data[first] = i;
            first++;
        }
    }

}
```

L'uso dei costruttori ci consente di inizializzare, al momento della creazione dell'oggetto tutti i dati membro (ora dichiarati privati), compreso l'array che conterrà i dati dello stack. Rispetto alla prima definizione della classe Stack fatta nel *laboratorio 3*, non sarà più necessario creare lo stack al momento della chiamata al metodo *push(int)* rendendo di conseguenza inutile il controllo sullo stato dell'array ad ogni sua chiamata:

```
if (data == null)
{
    first = 0;
    data = new int[20];
}
```

Di fatto, utilizzando il costruttore saremo sempre sicuri che lo stato iniziale della classe è correttamente impostato.

Overloading dei costruttori

Al programmatore è consentito scrivere più di un costruttore per una data classe a seconda delle necessità di disegno dell'oggetto, permettendogli di passare diversi insiemi di dati di inizializzazione. Ad esempio, un oggetto *Stack* potrebbe di default contenere al massimo 10 elementi. Un modo per generalizzare l'oggetto è quello di scrivere un costruttore che prendendo come parametro di input un intero, inizializza la dimensione massima dello *Stack* a seconda delle necessità della applicazione.

../javanet/mattone/cap5/Stack.java

```
package javanet.mattone.cap5;
public class Stack
{
```

```
        private int maxsize;
        private int data[];
        private int first;

        public Stack)
        {
                maxsize = 10;
                data = new int[10];
                first=0;
        }

        public Stack (int size)
        {
                maxsize=size;
                data = new int[size];
                first= 0;
        }

        int pop()
        {
                if (first > 0)
                {
                        first--;
                        return data[first];
                }
                return 0; // Bisogna tornare qualcosa
        }
        void push(int i)
        {

                if (first < maxsize)
                {
                        data[first] = i;
                        first++;
                }
        }

    }
```

L'utilizzo dei due costruttori della classe Stack ci consente di creare oggetti Stack di dimensioni variabili chiamando il costruttore che prende come parametro di input un intero che rappresenta le dimensioni dello stack:

```
        public Stack (int size)
        {
                maxsize=size;
                data = new int[size];
                first= 0;
        }
```

Per creare una istanza della classe Stack invocando il costruttore *Stack(int)* basterà utilizzare l'operatore **new** come segue:

```
    int dimensioni=10;
    Stack s = new Stack(dimensioni);
```

Restrizione sulla chiamata ai costruttori

Java permette una sola chiamata a costruttore al momento della referenziazione dell'oggetto. Questo significa che nessun costruttore può essere eseguito

nuovamente dopo la creazione dell'oggetto. Di fatto, il codice Java descritto di seguito produrrà un errore di compilazione sulla riga 3:

```
1  int dimensioni=10;
2  Stack s = new Stack(dimensioni);
3  s.Stack(20);
```

Cross Calling tra costruttori

Java consente ad un costruttore di chiamare altri costruttori appartenenti alla stessa defnizione di classe. Questo meccanismo è utile inquanto i costruttori generalmente hanno funzionalità simili e, un costruttore che assume uno stato di default, potrebbe chiamarne uno che prevede che lo stato sia passato come parametro, chiamandolo e passando i dati di default.

Guardando la definizione di Stack, notiamo che i due costruttori fanno esattamente la stessa cosa. Per ridurre la quantità di codice, possiamo chiamare un costruttore da un altro. Per chiamare un costruttore da un altro, è necessario utilizzare una sintassi speciale:

this*(parameter_list);*

Nella chiamata*, parameter_list* rappresenta la lista di parametri del costruttore che si intende chiamare.

Una chiamata cross-call tra costruttori, deve essere la prima riga di codice del costruttore chiamante. Qualsiasi altra cosa venga fatta prima, compresa la definizione di variabili, Java non consente di effettuare tale chiamata. Il costruttore corretto viene determinato in base alla lista dei parametri paragonando *parameter_list* con la lista dei parametri di tutti i costruttori della classe.

../javanet/mattone/cap5/Stack.java

```
package javanet.mattone.cap5;

class Stack
{
    private int data[];
    private int max; //Dimensione massima
    private int size; //Dimensione Corrente

    public Stack ()
    {
        this(10);
    }

    public Stack (int max_size)
    {
        data = new int[max_size];
        size = 0;
        max = max_size;
    }

    void push(int n)
    {
        if(size<max)
        {
            data[size]=n;
            size++;
        } else
```

```
                return;
        }
        int pop()
        {
                if(size > 0)
                {
                        size--;
                        return data[size];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

Laboratorio 5
Incapsulamento di oggetti

Esercizio 1

Definire un oggetto chiamato Set che rappresenta un insieme di interi.
L'insieme deve avere al massimo tre metodi:

boolean isMember(**int**); //Ritorna *true* se il numero è nell'insieme
void addMember(**int**); //Aggiunge un numero all'insieme
void showSet(); //stampa a video il contenuto dell'insieme nel formato:
// {1, 4, 5, 12}

Assicurarsi di incapsulare l'oggetto utilizzando i modificatori public/private appropriati.

Soluzione del primo esercizio

Scopo dell'esercizio è verificare il meccanismo dell'incapsulamento mediante i modificatori **public** e **private**.

L'incapsulamento di oggetti è il processo di mascheramento dei dettagli dell'implementazione ad altri oggetti per evitare riferimenti incrociati. Nascondendo i dettagli, possiamo assicurare a chi utilizza l'oggetto che ciò che sta utilizzando è sempre in uno stato consistente a meno di bug dell'oggetto stesso. Uno stato consistente è uno stato permesso dal disegno di un oggetto.

Il modificatore **private** realizza incapsulamento a livello di definizione di classe e serve a definire membri che devono essere utilizzati solo da altri membri della stessa classe di definizione. L'intento di fatto è quello di nascondere porzioni di codice della classe che non devono essere utilizzati da altre classi.

Un membro privato può essere utilizzato da di qualsiasi membro statico, e non, della stessa classe di definizione con l'accorgimento che i membri statici possono solo utilizzare membri (dati od oggetti) statici o entità di qualunque tipo purché esplicitamente passate per parametro.

Il modificatore **public** consente di definire classi o membri di una classe visibili da qualsiasi classe all'interno dello stesso package e non. Public deve essere utilizzato per definire l'interfaccia che l'oggetto mette a disposizione dell'utente. Tipicamente metodi membro **public** utilizzano membri **private** per implementare le funzionalità dell'oggetto.

Di seguito una possibile soluzione all'esercizio proposto:

c:\esercizi\capitolo5\Set.java

```
1     class Set
2     {
3         private int numbers[];
4         private int cur_size;
5
6         public Set()
7         {
8             cur_size=0;
9             numbers = new int[100];
10
11
12            public boolean isMember(int n)
13            {
14                int i;
15                i=0;
16                while(i < cur_size)
18                {
19                    if(numbers[i]==n) return true;
20                    i++;
21                }
22                return false;
23            }
24
25            public void addMember(int n)
26            {
27                if(isMember(n)) return;
28                if(cur_size == numbers.length) return;
29               numbers[cur_size++] = n;
30            }
31
```

```
32          public void showSet()
33          {
34              int i;
35              i=0;
36              System.out.println("{");
37              while(i < cur_size)
38              {
39                  System.out.println(numbers[i] +" , ");
40                  i++;
41              }
42              System.out.println("}");
43          }
44      }
```

Utilizzando il modificatore **private**, con le righe 3 e 4 del codice sorgente dichiariamo i dati membro della nostra classe rendendoli visibili soltanto ai metodi membro della classe. Il primo, ***int*** *numbers[]*, rappresenta un array di interi che conterrà i dati memorizzati all'interno dell'insieme. Il secondo, **int** cur_size, rappresenta la dimensione o cardinalità attuale dell'insieme.

```
3       private int numbers[];
4       private int cur_size;
```

Aver dichiarato private i due dati membro abbiamo incapsulato all'interno della classe i dettagli dell'implementazione dell'insieme evitando errori dovuti all'inserimento diretto di elementi all'interno dell'insieme. Se ciò fosse possibile, potremmo scrivere il metodo main in modo che inserisca direttamente elementi all'interno dell'array come da esempio seguente:

c:\esercizi\capitolo5\TestSet.java

```
public class TestSet
{
    public static void main(String args[])
    {
            Set mySet = new Set();
            //la prossima chiamata è illegale
            mySet.numbers[1]=0;
    }

}
```

In questo caso non verrebbe aggiornata la cardinalità dell'insieme rendendo inconsistente lo stato dell'oggetto, o ancora peggio, potremmo sovrascrivere elementi dell'insieme esistenti. Nel secondo caso lo stato dell'oggetto rimarrebbe consistente, ma l'aver cancellato dati già inseriti potrebbe causare effetti inaspettati nel corso dello svolgimento della applicazione.

Le righe 12-43 della definizione della classe rappresentano i metodi membro pubblici che la classe mette a disposizione dell'utente per manipolare i dati membro. Tali metodi rappresentano l'interfaccia che il programmatore ha a disposizione per utilizzare in modo corretto l'insieme di interi.

Capitolo 6
Ereditarietà

Introduzione

L'ereditarietà è la caratteristica dei linguaggi object oriented che consente di utilizzare classi come base per la definizione di nuovi oggetti che ne specializzano il concetto. L'ereditarietà fornisce inoltre un ottimo meccanismo per aggiungere funzionalità ad un programma con rischi minimi per le funzionalità esistenti, nonché un modello concettuale che rende un programma object oriented auto-documentante rispetto ad un analogo scritto con linguaggi procedurali.

Per utilizzare correttamente l'ereditarietà, il programmatore deve conoscere a fondo gli strumenti forniti dal linguaggio in supporto. Questo capitolo introduce al concetto di ereditarietà in Java, alla sintassi per estendere classi, all'overloading e overriding di metodi e ad una caratteristica molto importante di Java che include per default la classe Object nella gerarchia delle classi definite dal programmatore.

Disegnare una classe base

Quando disegniamo una classe, dobbiamo sempre tenere a mente che con molta probabilità ci sarà qualcuno che in seguito potrebbe aver bisogno di utilizzarla tramite il meccanismo di ereditarietà. Ogni volta che si utilizza una classe per ereditarietà ci si riferisce a questa come alla "classe base" o "superclasse". Il termine ha come significato che la classe è stata utilizzata come fondamenta per una nuova definizione. Quando definiamo nuovi oggetti utilizzando l'ereditarietà, tutte le funzionalità della classe base sono trasferite alla nuova classe detta "classe derivata" o "sottoclasse".

Quando si fa uso della ereditarietà, bisogna sempre tener ben presente alcuni concetti.

L'ereditarietà consente di utilizzare una classe come punto di partenza per la scrittura di nuove classi. Questa caratteristica può essere vista come una forma di riutilizzazione del codice: i membri della classe base sono "concettualmente" copiati nella nuova classe. Come conseguenza diretta, l'ereditarietà consente alla classe base di modificare la superclasse. In altre parole, ogni aggiunta o modifica ai metodi della superclasse sarà applicata solo alle classe derivata. La classe base risulterà quindi protetta dalla generazione di nuovi eventuali bug, che rimarranno circoscritti alla classe derivata. La classe derivata per ereditarietà, supporterà tutte le caratteristiche della classe base.

In definitiva, tramite questa tecnica è possibile creare nuove varietà di entità già definite mantenendone tutte le caratteristiche e le funzionalità. Questo significa che se una applicazione è in grado di utilizzare una classe base, sarà in grado di utilizzarne la derivata allo stesso modo. Per questi motivi, è importante che una classe base rappresenti le funzionalità generiche delle varie specializzazioni che andremo a definire.

Il modello di ereditarietà proposto da Java è un modello ad ereditarietà singola. A differenza da linguaggi come il C++ in cui una classe derivata può ereditare da molte classi base (*figura 6-1*), Java consente di poter ereditare da una sola classe base come mostrato nella *figura 6-2*.

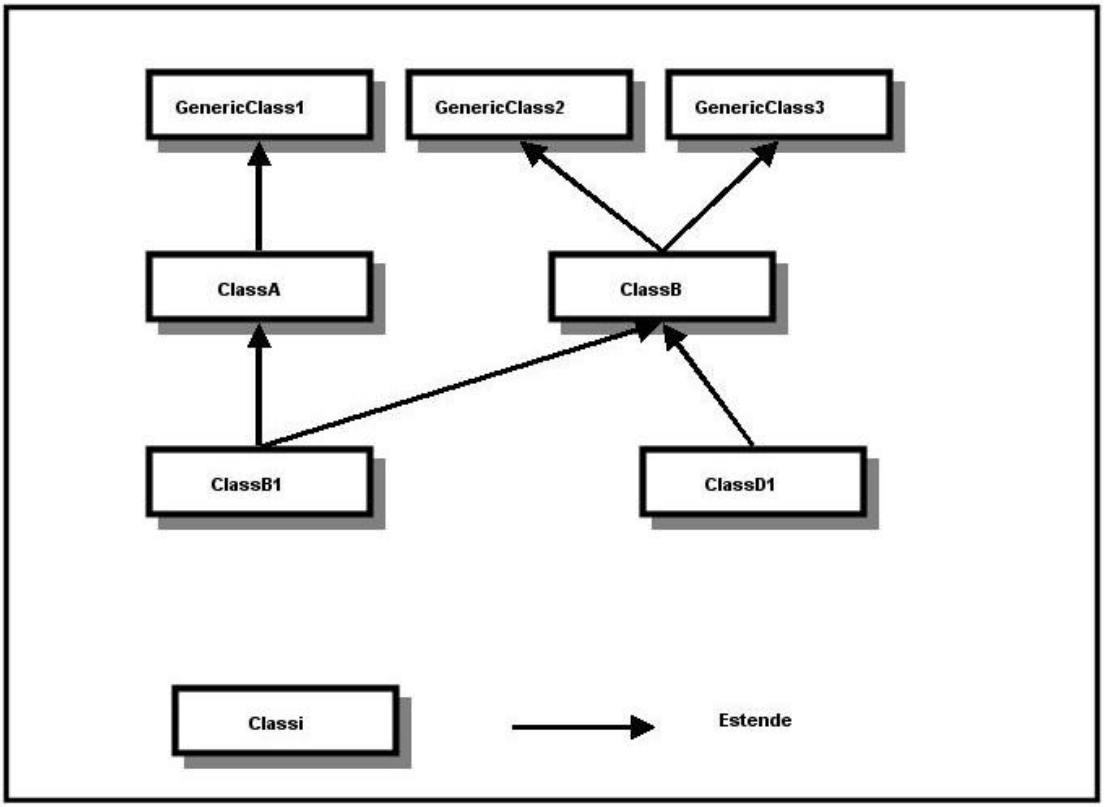


Figura 6-1 : *modello ad ereditarietà multipla*

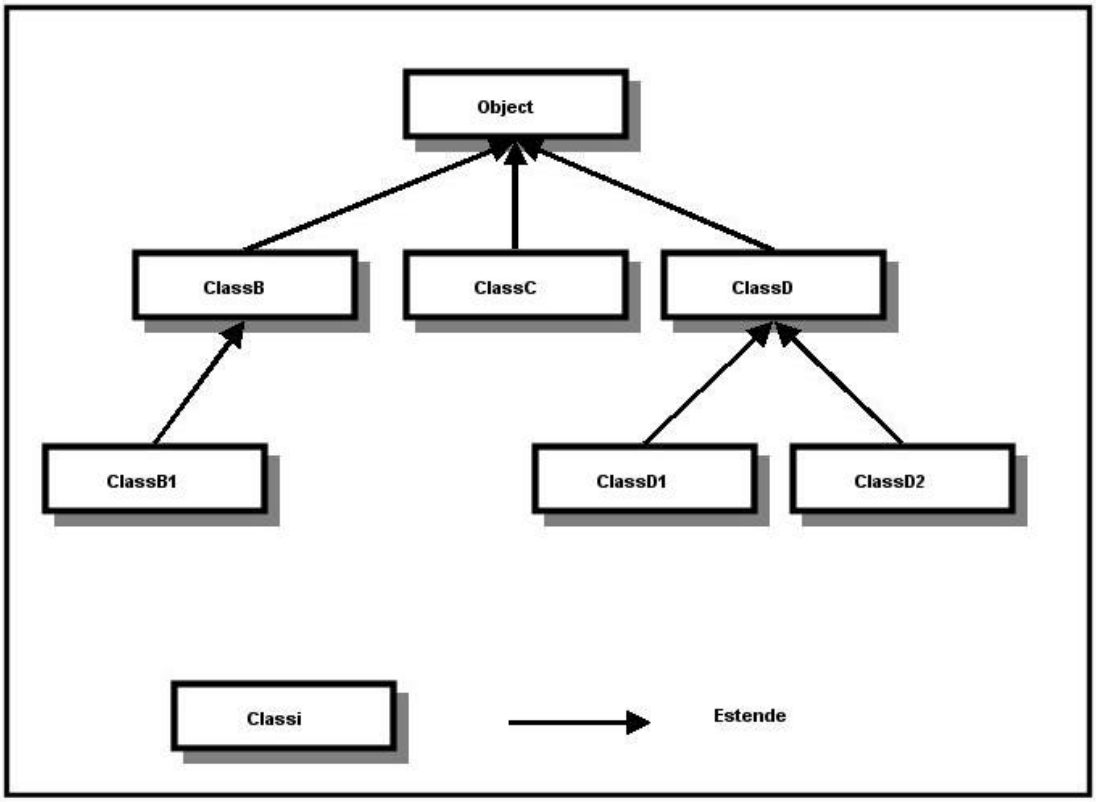


Figura 6-2 : *modello ad ereditarietà singola in Java*

Proviamo a pensare ad un veicolo generico: questo potrà muoversi, svoltare a sinistra o a destra o fermarsi. Di seguito la definizione della classe base che descrive

il concetto, e della classe contenete l'entry point (metodo main(…)) della applicazione.

../javanet/mattone/cap6/Autista.java

```
public class Autista
{
    public static void main(String args[])
    {
        Veicolo v = new Veicolo();
        v.muovi();
        v.sinistra();
        v.diritto();
        v.ferma();
    }
}
```

../javanet/mattone/cap6/Veicolo.java

```
public class Veicolo
{
    String nome;
    int velocita; //in Km/h
    int direzione;
    final static int DRITTO=0;
    final static int SINISTRA = -1;
    final static int DESTRA = 1;
    public Veicolo()
    {
        velocita=0;
        direzione = DRITTO;
        nome = "Veicolo generico";
    }
    public void muovi()
    {
        velocita=1;
        System.out.println(nome + " si sta movendo a: "+ velocita+" Kmh");
    }
    public void ferma()
    {
        velocita=0;
        System.out.println(nome + " si è fermato");
    }
    public void sinistra()
    {
        direzione =SINISTRA;
        System.out.println(nome + " ha sterzato a sinistra");
    }
    public void destra()
    {
        direzione =DESTRA;
        System.out.println(nome + " ha sterzato a destra");
    }
    public void diritto()
    {
        direzione = DRITTO;
        System.out.println(nome + " sta procedendo in linea retta");
    }
}
```

Overload di metodi

Per utilizzare a fondo l'ereditarietà, è necessario introdurre un'altra importante caratteristica di Java: quella di consentire l'overloading di metodi. Fare l'overloading di un metodo significa, in generale, dotare una classe di metodi aventi stesso nome ma con parametri differenti.

Consideriamo ad esempio il metodo *muovi()* della classe *Veicolo* nell'esempio precedente: il metodo simula la messa in moto del veicolo alla velocità di 1 Km/h. Apportiamo ora qualche modifica alla definizione di classe:

../javanet/mattone/cap6/Veicolo.java

```
public class Veicolo
{
    String nome;
    int velocita; //in Km/h
    int direzione;
    final static int DRITTO=0;
    final static int SINISTRA = -1;
    final static int DESTRA = 1;

    public Veicolo()
    {
                velocita=0;
                direzione = DRITTO;
                nome = "Veicolo";
    }
    public void muovi()
    {
        velocita=1;
        System.out.println(nome + " si sta movendo a: "+ velocita+" Kmh");
    }
    public void muovi(int quale_velocita)
    {
        velocita= quale_velocita;
        System.out.println(nome + " si sta movendo a: "+ velocita+" Kmh");
    }
    public void ferma()
    {
        velocita=0;
        System.out.println(nome + " si è fermato");
    }
    public void left()
    {
        direzione =SINISTRA;
        System.out.println(nome + " ha sterzato a sinistra");
    }
    public void destra()
    {
        direzione =DESTRA;
        System.out.println(nome + " ha sterzato a destra");
    }
    public void diritto()
    {
        direzione = DRITTO;
        System.out.println(nome + " sta procedendo in linea retta");
    }
}
```

Avendo a disposizione anche il metodo *muovi(**int** quale_velocita)* potremo migliorare la nostra simulazione facendo in modo che il veicolo possa accelerare o decelerare ad una determinata velocità.

Ecco alcune linee guida per utilizzare l'overloading di metodi. Primo, non possono esistere due metodi aventi nomi e lista dei parametri contemporaneamente uguali. Secondo, i metodi di cui si è fatto l'overloading devono implementare vari aspetti di una medesima funzionalità.

Nell'esempio, aggiungere un metodo *muovi()* che provochi la svolta della macchina non avrebbe senso.

Estendere una classe base

Definita la classe base *Veicolo*, sarà possibile definire nuovi tipi di veicoli estendendo la classe generica. La nuova classe, manterrà tutti i dati ed i metodi membro della superclasse con la possibilità di aggiungerne di nuovi o modificare quelli esistenti.

La sintassi per estendere una classe a partire dalla classe base è la seguente:

***class** nome_classe **extends** nome_super_classe*

L'esempio seguente mostra come creare un oggetto *Macchina* a partire dalla classe base *Veicolo*.

../javanet/mattone/cap6/Macchina.java

```
public class Macchina extends Veicolo
{
    public Macchina()
    {
            velocita=0;
            direzione = DRITTO;
            nome = "Macchina";
    }
}
```

../javanet/mattone/cap6/Autista.java

```
public class Autista
{
    public static void main(String args[])
    {
            Macchina fiat = new Macchina ();
            fiat.muovi();
            fiat.sinistra();
            fiat.diritto();
            fiat.ferma();
    }
}
```

Estendendo la classe *Veicolo*, ne ereditiamo tutti i dati membro ed I metodi. L'unico cambiamento che abbiamo dovuto apportare è quello di creare un costruttore ad hoc. Il nuovo costruttore semplicemente modifca il contenuto della variabile nome affinché l'applicazione stampi I messaggi corretti.

Come mostrato nel nuovo codice della classe *Autista*, utilizzare il nuovo veicolo equivale ad utilizzare il *Veicolo* generico definito nei paragrafi precedenti.

Ereditarietà ed incapsulamento

Nasce spontaneo domandarsi quale sia l'effetto dei modificatori **public**, **private**, e **protected** definiti nel capitolo precedente nel caso di classi legate tra di loro da relazioni di ereditarietà. Nella tabella seguente sono schematizzati i livelli di visibilità dei tre modificatori.

Modificatori ed ereditarietà	
Modificatore	**Visibilità con sottoclassi**
public	SI
private	NO
protected	SI

Il modificatore **public** consente di dichiarare dati e metodi membro visibili e quindi utilizzabili da una eventuale sottoclasse ed il modificatore **private** nasconde completamente dati e metodi membro dichiarati tali. E' invece necessario soffermarci sul modificatore **protected**.

Immaginiamo di aver definito una classe chiamata *ClasseBase* ed appartenente al package *packbase* nel modo seguente:

```
package packbase;
class ClasseBase
{
    protected int datoprotetto;

    protected void metodoProtetto()
    {
            ……
    }
}
```

Supponiamo quindi di utilizzare questa classe come dato membro della classe *AltraClasse* che non sia sottoclasse della prima , ma appartenga allo stesso package:

```
package packbase;
class AltraClasse
{
    ClasseBase a = new ClasseBase();

    void metodo ()
    {
            a. metodoprotetto();
    }
}
```

In questo caso sarà possibile utilizzare il metodo **protected** *metodoProtetto()* di *ClasseBase* senza che Java segnali errori. Consideriamo ora la classe *TerzaClasse* definita per ereditarietà a partire dalla classe base *ClasseBase*, ma appartenente ad un package differente :

```
import packbase;

package altropackage;
class TerzaClasse extends ClasseBase
{
    void metodoInterno (ClasseBase cb, TerzaClasse tc)
    {
            tc.metodoprotetto(); //Chiamata legale
            cb.metodoprotetto(); //Chiamata illegale
    }
}
```

In questo caso la nuova classe potrà utilizzare il metodo *metodoprotetto()* ereditato dalla superclasse anche se non appartenente allo stesso package, ma non potrà utilizzare lo stesso metodo se chiamato direttamente come metodo membro della class ClasseBase referenziata dalla variabile cb e passata come parametro al metodo *metodoInterno (ClasseBase cb, TerzaClasse tc)*.

Di fatto, il modificatore **protected** consente l'accesso ad un metodo o dato membro di una classe a tutte le sue sottoclassi o alle classi appartenenti allo stesso package. In tutti gli altri casi on sarà possibile utilizzare metodi e dati membro definiti ptotetti.

Conseguenze dell'incapsulamento nella ereditarietà

Negli esempi precedenti si nota facilmente che il codice del metodo costruttore della classe *Veicolo* è molto simile a quello del costruttore della classe *Macchina*. Conseguentemente potrebbe tornare utile utilizzare il costruttore della classe base per effettuare almeno una parte delle operazioni di inizializzazione. Facciamo qualche considerazione:

Cosa potrebbe succedere se sbagliassimo qualcosa nella definizione del costruttore della classe derivata?

Cosa succederebbe se nella classe base ci fossero dei dati privati che il costruttore della classe derivata non può aggiornare?

Consideriamo l'esempio seguente:

```
class A
{
    private int stato;

    public A()
    {
            stato=10;
    }

    public int f()
    {
            //ritorna valori basati sullo stato impostato a 10
    }
}
```

```
class B extends A
{
    private int var;
    public  B()
    {
            var=20;
            stato = 20;
    }
    public  int g()
    {
            return f();
    }
}
```

La classe base A definisce un dato membro privato di tipo intero chiamato stato e, oltre al costruttore che si occupa di inizializzare il dato membro, mette a disposizione un metodo pubblico *f()* che ritorna un valore di tipo **int** ottenuto da un algoritmo che utilizza il dato membro *stato* per effettuare i calcoli. La classe derivata B tenta, nel costruttore, di inizializzare il dato membro privato di della classe A, producendo un errore in fase di compilazione.

E' quindi necessario eliminare il problema. La classe B corretta avrà la forma seguente:

```
class B extends A
{
    private int var;
    public  B()
    {
            var=20;
    }
    public  int g()
    {
            return f(); // Problemi di runtime dal momento che A.stato
                        // non è stato inizializzato correttamente
    }
}
```

Ora la classe verrà compilata correttamente, ma il risultato fornito dalla chiamata al metodo *g()* produrrà valori inattendibili a causa della mancata inizializzazione del dato privato *stato* della classe A.

Per assicurare che un oggetto venga inizializzato ad uno stato corretto, i dati della classe devono essere inizializzati con i valori corretti, ma questo è esattamente il compito di un costruttore. In altre parole, Java applica l'incapsulamento anche a livello di ereditarietà e ciò significa che non solo Java deve consentire l'uso del costruttore della classe derivata, ma anzi deve forzare il programmatore affinché lo utilizzi.

Tornando al nostro esempio con *Macchina* e *Veicolo*, la seconda condivide tutti i dati membro con la prima, ed è quindi possibile modificarne lo stato dei dati membro mediante accesso diretto ai dati. Se la classe *Veicolo* avesse però avuto dei dati privati, l'unico modo per modificarne lo stato sarebbe stato attraverso i metodi pubblici della classe. In questo modo la classe base gode di tutti i benefici dell'incapsulamento (isolamento dei bug, facilità di tuning ecc.).

Ereditarietà e costruttori

Il meccanismo utilizzato da Java per assicurare la chiamata di un costruttore per ogni classe di una gerarchia è il seguente.

Primo, ogni classe ***deve*** avere un costruttore. Se il programmatore non ne implementa alcuno, Java assegnerà alla classe un costruttore di default con blocco del codice vuoto e senza lista di parametri:

```
public QualcheCostruttore() {
    }
```

Il costruttore di default viene utilizzato solamente in questo caso. Se il programmatore implementa un costruttore specializzato con lista di parametri di input non vuota, Java elimina completamente il costruttore di default.

Secondo, se una classe è derivata da un'altra l'utente può chiamare il costruttore della classe base immediatamente precedente nella gerarchia utilizzando la sintassi:

super*(argument_list);*

dove argument_list è la lista dei parametri del costruttore da chiamare. Una chiamata esplicita al costruttore della classe base deve essere effettuata prima di ogni altra operazione incluso la dichiarazione di variabili. Ad esempio, il costruttore *costruttoreDefinitoDaUtente()* definito di seguito chiama il costruttore della classe base che accetta un tipo **int** come parametro di input.

```
public costruttoreDefinitoDaUtente()
{
    super(23);
    int i;
}
```

Per comprendere meglio il meccanismo modifichiamo l'esempio visto nei paragrafi precedenti:

```
class A
{
    private int stato;
    public A()
    {
        stato=10;
    }
    public int f()
    {
        //ritorna valori basati sullo stato impostato a 10
    }
}

class B extends A
{
    private int var;
    public  B()
    {
        super();
        var=20;
    }
    public  int g()
    {
        return f();
    }
}
```

Il costruttore della classe B derivata da A esegue come prima istruzione la chiamata al costruttore della classe A. Consideriamo nuovamente la classe B eliminando la chiamata *super()* all'interno del costruttore.

```
class B extends A
{
    private int var;
    public  B()
    {
            var=20;
    }
    public  int g()
    {
            return f();
    }
}
```

Se l'utente non effettua una chiamata esplicita al costruttore della classe base, Java esegue implicitamente tale chiamata.

```
class B extends A
{
    private int var;
    public  B()
    {
            //Chiamata implicita a super();
            var=20;
    }
    public  int g()
    {
            return f(); //Lavora correttamente
    }
}
```

In caso di chiamata implicita a costruttore, Java utilizza una chiamata al costruttore della classe base senza passare argomenti. Se la classe base non è dotata di costruttore senza argomenti verrà generato un errore in fase di compilazione.

Se avessimo compilato il codice dell'esempio precedente, avremmo notato subito che il codice funziona correttamente ed i due oggetti sarebbero stati sempre in uno stato consistente proprio grazie alla chiamata implicita che Java effettua nel costruttore di B sul costruttore della super classe A.

Aggiungere nuovi metodi

Quando estendiamo una classe, possiamo aggiungere nuovi metodi alla classe derivata. Ad esempio, una *Macchina* generalmente possiede un clacson. Aggiungendo il metodo *suona()*, continueremo a mantenere tutte le vecchie funzionalità, ma ora la macchina è in grado di suonare il clacson.

Definire nuovi metodi all'interno di una classe derivata ci consente quindi di definire tutte le caratteristiche particolari non previsto nella definizione generica del concetto che rappresentiamo implementando la classe base.

../javanet/mattone/cap6/Macchina.java

```
public class Macchina extends Veicolo
{
    public Macchina()
    {
            velocita=0;
            direzione = DRITTO;
            nome = “Macchina”;
    }

    public void suona()
    {
            System.out.println(nome + “ ha attivato il clacson ”);
    }
}
```

../javanet/mattone/cap6/Autista.java

```
public class Autista
{
    public static void main(String args[])
    {
            Macchina fiat = new Macchina ();
            fiat.muovi();
            fiat.sinistra();
            fiat.diritto();
            fiat.ferma();
            fiat.suona();
    }
}
```

Overriding di metodi

Se un metodo ereditato non lavorasse correttamente rispetto a quanto ci aspettiamo dalla specializzazione del concetto definito nella classe base, Java ci consente di riscrivere il metodo originale. Questo significa semplicemente riscrivere il metodo coinvolto all’interno della classe derivata.

Anche in questo caso, riscrivendo nuovamente il metodo solo nella nuova classe, non c’è pericolo che la classe base venga manomessa. Il nuovo metodo verrà chiamato al posto del vecchio anche se la chiamata venisse effettuata da un metodo ereditato dalla classe base.

Modifichiamo la definizione della classe Macchina:

../javanet/mattone/cap6/Macchina.java

```
public class Macchina extends Veicolo
{
    public Macchina()
    {
                    velocita=0;
                    direzione = DRITTO;
                    nome = “ Macchina ”;
    }

    public void muovi(int quale_velocita)
    {
                    if(quale_velocita<120)
```

```
                    velocita= quale_velocita;
                else
                        velocita = 120;
                System.out.println(nome + “s i sta movendo a: ”+ velocita+” Kmh”);
    }

    public void suona()
    {
                System.out.println(nome + “ ha attivato il clacson”);
    }
}
```

../javanet/mattone/cap6/Autista.java

```
public class Autista
{
    public static void main(String args[])
    {
            Macchina fiat = new Macchina ();
            fiat.muovi(300);
            fiat.sinistra();
            fiat.diritto();
            fiat.ferma();
            fiat.suona();
    }
}
```

Quando nel metodo main verrà effettuata la chiamata al metodo *muovi(int)*, la applicazione farà riferimento al metodo della classe *Macchina* e non a quello definito all’interno della classe base *Veicolo*.

Chiamare metodi della classe base

La parola chiave **super**() può essere utilizzata anche nel caso in cui sia necessario richiamare un metodo della super classe ridefinito nella classe derivata con il meccanismo di overriding. Grazie alla parola chiave **super** il programmatore non necessariamente deve riscrivere un metodo completamente, ma è libero di utilizzare parte delle funzionalità definite all’interno del metodo della classe base. Ritorniamo ancora sulla definizione della classe *Macchina*:

```
public class Macchina extends Veicolo
{
……

    public void muovi(int quale_velocita)
    {
            if(quale_velocita<120)
                    velocita= quale_velocita;
            else
                    velocita = 120;
            System.out.println(nome + “si sta movendo a: ”+ velocita+” Kmh”);
    }

…….
}
```

il metodo *muovi(int quale_velocita)* della classe *Macchina* effettua un check della variabile di input per limitare la velocità massima del mezzo, quindi esegue un assegnamento identico a quello effettuato nel metodo analogo definito nella superclasse *Veicolo* e riportato di seguito:

```
public void muovi(int quale_velocita)
{
        velocita= quale_velocita;
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
}
```

Il metodo *muovi()* della classe macchina può quindi essere riscritto in modo da sfruttare quanto già implementato nella superclasse nel modo seguente:

```
public class Macchina extends Veicolo
{
……

    public void muovi(int quale_velocita)
    {
        if(quale_velocita<120)
            velocita= quale_velocita;
        else
            super.go(quale_velocita);
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
    }

…….
}
```

E' importante notare che a differenza della chiamata a costruttori che richiedeva solo l'uso della parola chiave **super** ed eventualmente la lista dei parametri, ora è necessario utilizzare l'operatore "." specificando il nome del metodo da chiamare. In questo caso **super** ha lo stesso significato di una variabile reference con la differenza che viene istanziato dalla JVM e referenzia sempre la superclasse della classe attiva ad ogni istante.

Flessibilità delle variabili reference

Una volta che una classe Java è stata derivata, Java consente alle variabili reference che rappresentano il tipo della classe base di referenziare ogni istanza di un oggetto derivato da essa nella gerarchia definita dalla ereditarietà.

```
Veicolo v = new Macchina();
v.muovi(10); //Chiama il metodo muovi() dichiarato nella classe Macchina
```

Il motivo alla base di questa funzionalità è che tutti gli oggetti derivati hanno sicuramente **almeno** tutti i metodi della classe base (li hanno ereditati), e quindi non ci dovrebbero essere problemi nell'utilizzarli. Nel caso in cui un metodo sia stato ridefinito mediante overriding, queste tipo di referenziamento comunque effettuerà una chiamata al nuovo metodo.

Quanto detto ci consente di definire il concetto di "compatibilità" tra variabili reference. In particolare due variabili reference sono per definizione compatibili se una referenzia un oggetto definito per ereditarietà a partire dall'oggetto referenziato dall'altra o viceversa.

Tornando al nostro esempio, dal momento che la variabile reference v di tipo *Veicolo* referenzia la superclasse per la classe *Macchina*, v può essere utilizzata per utilizzare un oggetto Macchina.

Run-time e compile-time

Prima di procedere è necessario introdurre i due concetti di run-time e compile-time. Il tipo rappresentato a **compile-time** di una espressione, è il tipo dell'espressione come dichiarato formalmente nel codice sorgente. Il tipo rappresentato a **run-time** è invece quello determinato quando il programma è in esecuzione. Il tipo a compile-time è sempre costante, mentre quello a run-time può variare.

Nel nostro esempio del *Veicolo* e della *Macchina*, una variabile reference di tipo Veicolo rappresenta il tipo Veicolo a compile-time, e il tipo Macchina a run-time.

```
Veicolo v = new Macchina();
```

Volendo fornire una regola generale, diremo che il tipo rappresentato al compile-time da una variabile è specificato nella sua dichiarazione, mentre quello a run-time è il tipo attualmente rappresentato (*Figura 6-3*). I tipi primitivi (*int, float, double* etc.) invece rappresentano lo stesso tipo sia al run-time che al compile-time.

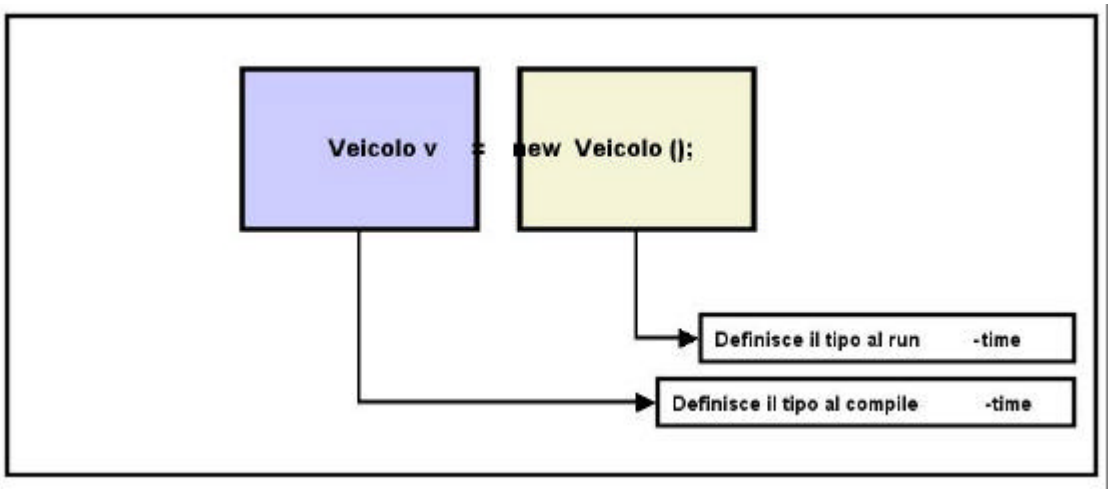


Figura 6-3 : *run-time e compile-time*

Per riassumere, ecco un breve esempio:

```
// v ha tipo a compile-time di Veicolo ed al run-time di Macchina
Veicolo v = new Macchina();

//Il tipo a run-time di v cambia in Veicolo
v = new Veicolo();

//b rappresenta al compile-time il tipo Libro, al run-time il tipo LibroDiMatematica
Libro b = new  LibroDiMatematica()

// i rappresenta un tipo int sia a run-time che a compile-time
int i;

// i è sempre un intero (4.0 viene convertito in intero)
i = 4.0;
```

E' comunque importante sottolineare che, una variabile reference potrà referenziare solo qualcosa il cui tipo sia in qualche modo compatibile con il tipo rappresentato al compile-time. Questa compatibilità è rappresentata dalla relazione di ereditarietà: tipi derivati sono sempre compatibili con le variabili reference dei predecessori.

Accesso a metodi attraverso variabili reference

Consideriamo ora le linee di codice:

```
Macchina c = new Macchina ();
Veicolo v = c;
c.suona();
v.suona();
```

Se proviamo a compilare il codice, il compilatore ci ritornerà un errore sulla quarta riga del sorgente. Questo perché, dal momento che il tipo rappresentato da una variabile al run-time può cambiare, il compilatore assumerà per definizione che la variabile reference sta referenziando l'oggetto del tipo rappresentato al compile-time. In altre parole, anche se una classe *Macchina* possiede un metodo *suona()*, questo non sarà utilizzabile tramite una variabile reference di tipo veicolo.

Cast dei tipi

Java fornisce un modo per girare intorno a questa limitazione. Il **cast** di un tipo consente di dichiarare che una variabile reference temporaneamente rappresenterà un tipo differente da quello rappresentato al compile-time. La sintassi di una cast di tipo è la seguente:

(new_type) variabile

Dove *new_type* è il tipo desiderato, e *variable* è la variabile che vogliamo convertire temporaneamente. Riscrivendo l'esempio precedente utilizzando il meccanismo di cast, il codice verrà compilato ed eseguito correttamente :

```
Macchina c = new Macchina ();
Veicolo v = c;
c.suona();
((Macchina) v).suona();
```

L'operazione di cast è possibile su tutti i tipi purché la variabile reference ed il nuovo tipo siano compatibili. Il cast del tipo di una variabile reference, ha effetto solo sul tipo rappresentato al compile-time e non sull'oggetto in se stesso. Il cast su un tipo provocherà la terminazione della applicazione se, il tipo rappresentato al run-time dall'oggetto non rappresenta il tipo desiderato al momento della esecuzione.

L'operatore instanceof

Dal momento che in una applicazione Java esistono variabili reference in gran numero, è a volte utile determinare al run-time che tipo di oggetto la variabile sta referenziando. A tal fine Java supporta l'operatore booleano **instanceof** che controlla il tipo di oggetto referenziato al run-time da una variabile reference.

La sintassi formale è la seguente:

*A **instanceof** B*

Dove A rappresenta una variabile reference, e B un tipo referenziabile. Il tipo rappresentato al run-time dalla variabile reference A verrà confrontato con il tipo definito da B. L'operatore tornerà uno tra i due possibili valori *true* o *false*. Nel primo caso (*true*) saremo sicuri che il tipo rappresentato da A al run-time consente di rappresentare il tipo rappresentato da B. *False*, indica che A referenzia l'oggetto *null* oppure che non rappresenta il tipo definito da B.

In poche parole, se è possibile effettuare il cast di A in B, **instanceof** ritornerà *true*. Consideriamo le poche righe di codice

```
Veicolo v = new Macchina();
v.suona();
```

In questo caso Java produrrà un errore di compilazione inquanto il metodo *suona()* è definito nella classe *Macchina* e non nella classe base *Veicolo*. Utilizzando l'operatore **instanceof** possiamo prevenire l'errore apportando al codice le modifiche seguenti:

```
Veicolo v = new Macchina();
if(v instanceof Macchina)
    ((Macchina)v).suona();
```

L'oggetto Object

La gerarchia delle classi in Java parte dalle definizione della classe Object che rappresenta la radice dell'albero delle gerarchie. Quando in Java viene creato un nuovo oggetto che non estende nessuna classe base, Java ne causerà l'estensione automatica dell'oggetto Object. Questo meccanismo è implementato per garantire alcune funzionalità base comuni a tutte le classi. Queste funzionalità includono la possibilità di esprimere lo stato di un oggetto in forma di stringhe (tramite il metodo ereditato toString()), la possibilità di comparare due oggetti tramite il metodo equals() e terminare l'oggetto tramite il metodo *finalize().*

Quest'ultimo metodo è utilizzato dal garbage collector nel momento in cui elimina l'oggetto rilasciando la memoria, e può essere modificato per poter gestire situazioni non gestibili dal garbage collector quali referenze circolari.

Il metodo equals()

Il metodo *equals()* ereditato dalla classe Object, è necessario dal momento che le classi istanziate vengono referenziate ossia le variabili reference "si comportano come se" e, questo significa che l'operatore di comparazione == è insufficiente inquanto opererebbe a livello di reference e non a livello di stato dell'oggetto.

Di fatto, il metodo *equals()* confronta due oggetti a livello di stato restituendo il valore true se e solo se i due oggetti rappresentano due istanze medesime della stessa definizione di classe ovvero, due oggetti di tipo compatibile si trovano nello stesso stato.

Immaginiamo di dover confrontare oggetti di tipo Integer:

../javanet/mattone/cap6/EsempioMetodoEquals.java

```
public class EsempioMetodoEquals
{
    public static void main(String args[])
    {
        Integer primointero = new Integer(1);
        Integer secondointero = new Integer(2);
        Integer terzointero = new Integer(1);

        if ( primointero.equals(secondointero) )
            System.out.println("primointero è uguale a secondointero");
        else
            System.out.println("primointero è diverso da secondointero");

        if ( primointero.equals(terzointero) )
            System.out.println("primointero è uguale a terzointero");
        else
            System.out.println("primointero è diverso da terzointero");
    }
}
```

Dopo aver compilato ed eseguito l'applicazione *EsempioMetodoEquals* l'output prodotto sarà il seguente:

```
primointero è diverso da secondointero
primointero è uguale a terzointero
```

Dal momento che il metodo *equals()* esegue un confronto a livello di stato tra due oggetti, l'implementazione definita all'interno della definizione della classe Object potrebbe non essere sufficiente. E' quindi necessario che il programmatore, implementando la definizione di una nuova classe, riscriva il metodo in questione utilizzando il meccanismo di overriding. Per capire meglio il funzionamento di questo metodo, definiamo una classe *Punto* che rappresenta un punto in uno spazio a due dimensioni ed effettua l'overriding di *equals(Object)*

../javanet/mattone/cap6/Punto.java

```
public class Punto
{
    public int x,y;
    public Punto(int xc, int yc)
    {
        x=xc;
        y=yc;
    }
    public boolean equals(Object o)
    {
        if (o instanceof Punto)
        {
            Punto p = (Punto)o;
            if (p.x == x && p.y==y) return true;
        }
        return false;
    }

    public static void main(String args[])
    {
        Point a,b;
```

```
        a= new Point(1,2);
        b= new Point(1,2);

        if (a==b)
                System.out.println("Il confronto tra variabili reference vale true");
        else
                System.out.println("Il confronto tra variabili reference vale false");

        if (a.equals(b))
                System.out.println("a è uguale a b");
        else
                System.out.println("a è diverso da b");
    }
}
```

L'esecuzione della applicazione produrrà il seguente output:

```
Il confronto tra variabili reference vale false
a è uguale a b
```

Rilasciare risorse esterne

Il metodo *finalize()* di una classe Java viene chiamato dal garbage collector prima di rilasciare l'oggetto e liberare la memoria allocata. Tipicamente questo metodo è utilizzato in quei casi in cui sia necessario gestire situazioni di referenza circolare, oppure situazione in cui l'oggetto utilizzi **metodi nativi** (metodi esterni a Java e nativi rispetto alla macchina locale) che utilizzano funzioni scritte in altri linguaggi. Dal momento che situazioni di questo tipo coinvolgono risorse al di fuori del controllo del garbage collector[12], *finalize()* viene chiamato per consentire al programmatore di implementare meccanismi di gestione esplicita della memoria.

Di default questo metodo non fa nulla.

Rendere gli oggetti in forma di stringa

Il metodi *toString()* è utilizzato per implementare la conversione in *String* di una classe. Tutti gli oggetti definiti dal programmatore, dovrebbero contenere questo metodo che ritorna una stringa rappresentante l'oggetto. Tipicamente questo metodo viene riscritto in modo che ritorni informazioni relative alla versione dell'oggetto ed al programmatore che lo ha disegnato.

L'importanza del metodo *toString()* è legata al fatto che il metodo statico *System.out.println()* utilizza *toString()* per la stampa dell'output a video.

[12] Ad esempio nel caso in cui si utilizzi una funzione C che fa uso della malloc() per allocare memoria

Laboratorio 6
Introduzione alla ereditarietà

Esercizio 1

Creare, a partire dalla classe Veicolo, nuovi tipi di veicolo mediante il meccanismo della ereditarietà: Cavallo, Nave, Aeroplano.

```
public class Veicolo
{
    String nome;
    int velocita; //in Km/h
    int direzione;
    final static int STRAIGHT=0;
    final static int LEFT = -1;
    final static int RIGHT = 1;

    public Veicolo()
    {
                velocita=0;
                direzione = STRAIGHT;
                nome = "Veicolo";
    }
    public void go()
    {
        velocita=1;
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
    }
    public void go(int quale_velocita)
    {
        velocita= quale_velocita;
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
    }
    public void stop()
    {
        velocita=0;
        System.out.println(nome + "si è fermato");
    }
    public void left()
    {
        direzione =LEFT;
        System.out.println(nome + "ha sterzato a sinistra");
    }
    public void right()
    {
        direzione =RIGHT;
        System.out.println(nome + "ha sterzato a destra");
    }
    public void diritto()
    {
        direzione = STRAIGHT;
        System.out.println(nome + "ha sterzato a sinistra");
    }
}
```

Esercizio 2

Trasformare la classe Driver in una definizione di classe completa (non più contenente solo il metodo main). La nuova definizione deve contenere un costruttore che richiede un Veicolo come parametro di input e sia in grado di riconoscere le limitazioni in fatto di velocità massima del veicolo preso in input. Aggiungere alla classe Driver il metodo sorpassa(Veicolo) e spostare il metodo main in una nuova classe. Creare alcuni autisti e veicoli ed effettuate la simulazione.

Soluzione al primo esercizio

```
public class Macchina extends Veicolo
{
    public Macchina()
    {
            velocita=0;
            direzione = STRAIGHT;
            nome = “Macchina”;
    }

    public void go(int quale_velocita)
    {
            if(quale_velocita<120)
                    velocita= quale_velocita;
            else
                    speed = 120;
            System.out.println(nome + “si sta movendo a: ”+ velocita+” Kmh”);
    }

    public void honk()
    {
            System.out.println(nome + “ha attivato il clacson”);
    }
}


public class Cavallo extends Veicolo
{
    int stanchezza; // range da 0 a 10
    public Cavallo (String n)
    {
            name = n + “il cavallo”;
            velocita=0;
            direzione = STRAIGHT;
            stanchezza = 0;
    }

    public void go(int velocita)
    {
            int velocita_massima = 20 – stanchezza;
            if(velocita > velocita_massima)
                    velocita= velocita_massima
            super.go(velocita);
            if(velocita > 10 && stanchezza <10) stanchezza++;
    }

    public void stop()
    {
            stanchezza = stanchezza /2;
            super.stop();
    }
}

public class Driver
{
    public static void main(String args[])
    {
            Macchina fiat = new Macchina ();
```

```
            fiat.go();
            fiat.left();
            fiat.diritto();
            fiat.stop();

            Cavallo furia = new Cavallo("Furia");
            furia.go(30);
            furia.left();
            furia.right();
            furia.go(15);
            furia.go(9);
            furia.go(12);
            furia.go(20);
        }
    }
```

Soluzione al secondo esercizio

```
    class Driver
    {
        String nome;
        Veicolo trasporto;

        public Driver(String nome, Veicolo v)
        {
            this.nome=nome;
            trasporto = v;
        }

        public void sorpassa(Driver altro_autista)
        {
            int velocita_sorpasso = altro_autista.trasporto.velocita+1;
            System.out.println(nome +" sta sorpassando un "+
                altro_autista.trasporto.nome + "con "+ trasporto.nome);
            trasporto.go(velocita_sorpasso);
            if(trasporto.velocita < velocita_sorpasso)
                System.out.println("Questo trasporto è troppo lento per
                    superare");
            else
                System.out.println("L'autista "+altro_autista.nome+"ha
                    mangiato la mia polvere");
        }

        public void viaggia(int velocita_di_marcia)
        {
            trasporto.diritto();
            trasporto.go(velocita_di_marcia);
        }
    }

    class Simulazione
    {
        public static void main(String args[])
        {
            Driver max = new Driver("Max", new Cavallo("Furia"));
            Driver catia = new Driver("Catia", new Macchina());
            Driver franco = new Driver("Franco", new Macchina());
```

```
            max.viaggia(15);
            franco.viaggia(30);
            catia.viaggia(40);

            max.sorpassa(franco);
            franco.sorpassa(catia);
            catia.sorpassa(max);
    }
}
```

Capitolo 7
Eccezioni

Introduzione

Le eccezioni sono utilizzate da Java in quelle situazioni in cui sia necessario gestire condizioni anomale, ed i normali meccanismi sono insufficienti ad indicare l'errore. Formalmente, una eccezione è un evento che si scatena durante la normale esecuzione di un programma, causando l'interruzione del normale flusso di esecuzione della applicazione.

Queste condizioni di errore possono svilupparsi in seguito ad una grande varietà di situazioni anomale: il malfunzionamento fisico di un dispositivo di sistema, la mancata inizializzazione di oggetti particolari quali ad esempio connessioni verso basi dati, o semplicemente errori di programmazione come la divisione per zero di un intero . Tutti questi eventi hanno la caratteristica comune di causare l'interruzione della esecuzione del metodo corrente.

Il linguaggio Java cerca di risolvere alcuni di questi problemi al momento della compilazione della applicazione tentando di prevenire ambiguità che potrebbero essere possibili cause di errore, ma non è in grado di gestire situazioni di errore complesse o indipendenti da eventuali errori di scrittura del codice.

Queste situazioni sono molto frequenti e spesso sono legate ai costruttori di classe. I costruttori sono chiamati dall'operatore **new** dopo aver allocato lo spazio di memoria appropriato all'oggetto da istanziare, non hanno valori di ritorno (dal momento che non c'è nessuno che possa catturarli) e quindi risulta molto difficile controllare casi di inizializzazione non corretta dei dati membro della classe (ricordiamo che non esistono variabili globali).

Oltre a quanto menzionato, esistono casi particolari in cui le eccezioni facilitano la vita al programmatore fornendo un meccanismo flessibile per descrivere eventi che, in mancanza delle quali, risulterebbero difficilmente gestibili. Consideriamo ancora una volta la classe *Stack* definita nel capitolo 3.

../javanet/mattone/cap7/Stack.java

```
public class Stack
{
    private int maxsize;
    private int data[];
    private int first;

    public Stack()
    {
            maxsize = 10;
            data = new int[10];
            first=0;
    }

    public Stack (int size)
    {
            maxsize=size;
            data = new int[size];
            first= 0;
    }

    int pop()
    {
            if (first > 0)
```

```
            {
                first--;
                return data[first];
            }
            return 0; // Bisogna tornare qualcosa
        }
        void push(int i)
        {

            if (first < maxsize)
            {
                data[first] = i;
                first++;
            }
        }

    }
```

Dal momento che il metodo *push()* non prevede parametri di ritorno, è necessario un meccanismo alternativo per gestire un eventuale errore causato da un overflow dell'array a seguito dell'inserimento di un ventunesimo elemento (ricordiamo che l'array può contenere solo 20 elementi).

Il metodo *pop()* a sua volta è costretto a utilizzare il valore 0 come parametro di ritorno nel caso in cui lo stack non contenga più elementi. Questo ovviamente costringere ad escludere il numero intero 0 dai valori che potrà contenere lo stack e dovrà essere riservato alla gestione della eccezione.

Un altro aspetto da considerare quando si parla di gestione degli errori è quello legato difficoltà nel descrivere e controllare situazioni arbitrariamente complesse. Immaginiamo una semplice routine di apertura, lettura e chiusura di un file. Chi ha già programmato con linguaggi come il C ricorda perfettamente i mal di testa causati dalle quantità codice necessario a gestire tutti i possibili casi di errore (*Figura 7-1*).

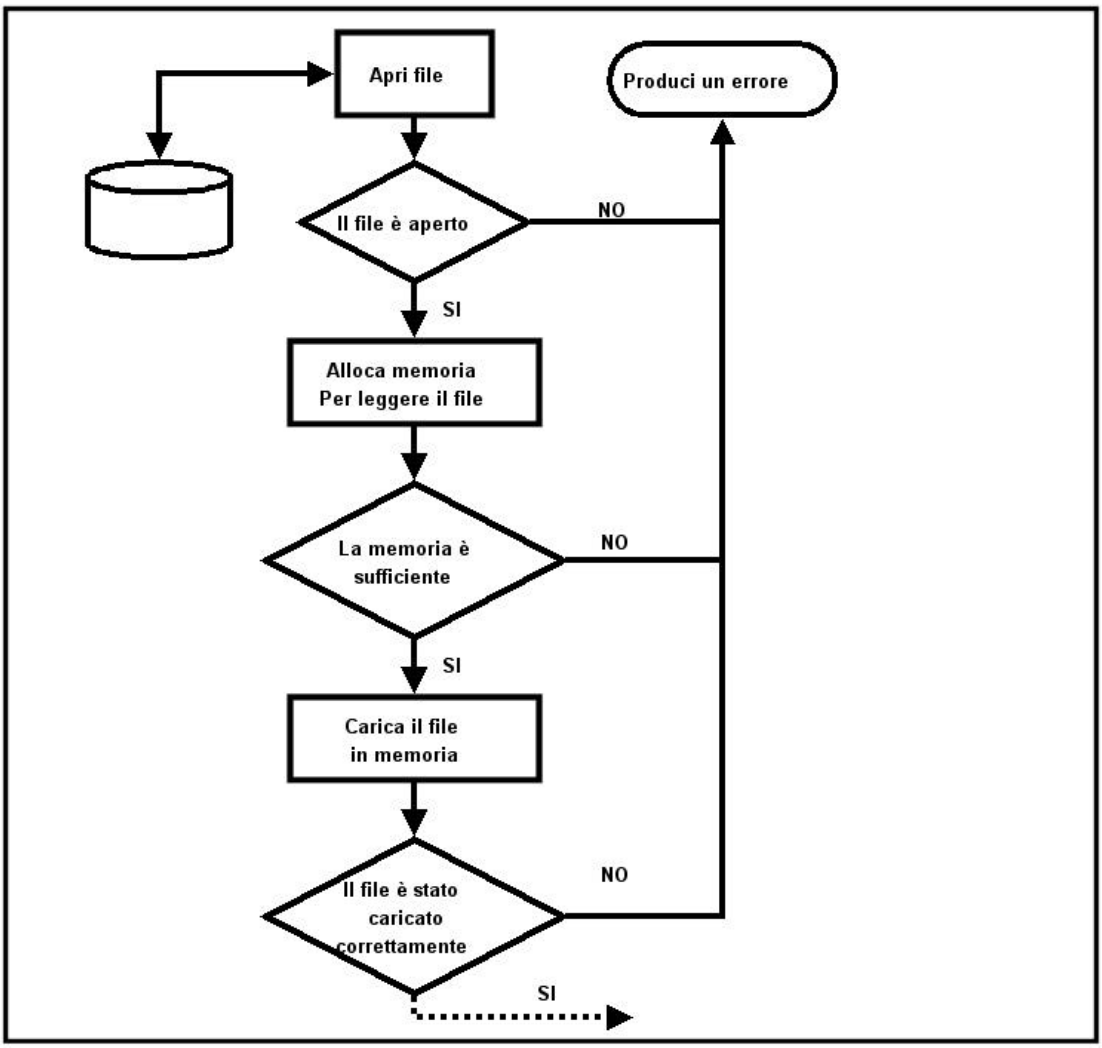


***Figura 7-1** : Schema di gestione degli errori in una routine di lettura di un file*

Grazie alla loro caratteristica di "oggetti particolari", le eccezioni si prestano facilmente alla descrizione di situazioni complicate fornendo al programmatore la capacità di descrivere e trasportare informazioni relativamente a qualsiasi tipologia di errore.

L'uso di eccezioni consente inoltre di separare il codice contenente le logiche dell'algoritmo della applicazione dal codice per la gestione degli errori.

Propagazione di oggetti

Il punto di forza del meccanismo delle eccezioni consiste nel consentire la propagazione di un oggetto a ritroso, attraverso la sequenza corrente di chiamate tra metodi. Opzionalmente, ogni metodo può fermare la propagazione e gestire la condizione di errore utilizzando le informazioni trasportate, oppure continuare la propagazione ai metodi subito adiacenti nella sequenza di chiamate. Ogni metodo che non sia in grado di gestire l'eccezione viene interrotto nel punto in cui aveva chiamato il metodo che sta propagando l'errore. Se la propagazione raggiunge l'entry point della applicazione e non viene arrestata, l'applicazione viene terminata.

Consideriamo l'esempio seguente:

```
class Example
{
    double metodo1()
    {
```

```
            double d;
            d=4.0 / metodo2();
            System.out.println(d) ;
        }

        float metodo2()
        {
            float f ;
            f = metodo3();
            //Le prossime righe di codice non vengono eseguite
            //se il metodo 3 fallisce
            f = f*f;
            return f;
        }

        int metodo3()
        {
            if(condizione)
                return espressione ;
            else
                // genera una eccezione e propaga l'oggetto a ritroso
                // al metodo2()
        }
    }
```

Questo pseudo codice Java rappresenta una classe formata da tre metodi: *metodo1()* che restituisce un tipo **double** il cui valore viene determinato sulla base del valore restituito da *metodo2()* di tipo **float**. A sua volta, *metodo2()* restituisce un valore **float** calcolato in base al valore di ritorno del *metodo3()* che, sotto determinate condizioni, genera una eccezione. Una chiamata a *metodo1()* genera quindi la sequenza di chiamate schematizzata nella *Figura 7-2*.

Se si verificano le condizioni per cui *metodo3()* genera l'eccezione, l'esecuzione del metodo corrente si blocca e l'eccezione viene propagata a ritroso verso *metodo2()* e *metodo1()* (*Figura 7-2*).

Una volta propagata, l'eccezione deve essere intercettata e gestita. In caso contrario si propagherà sino al metodo *main()* della applicazione causando la terminazione della applicazione.

Propagare un oggetto è detto "**exception throwing**" e fermarne la propagazione "**exception catching**".

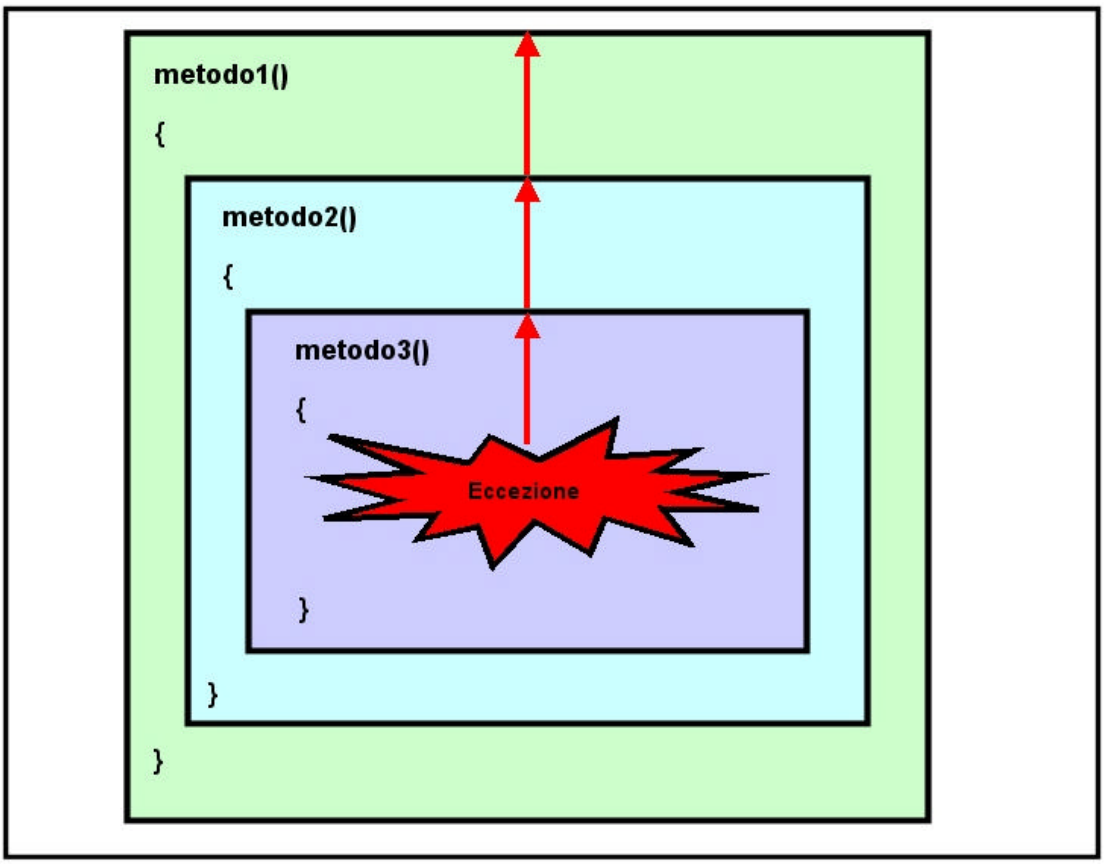


Figura 7-2 : *Propagazione di una eccezione Java*

In generale, gli oggetti da propagare come eccezioni devono derivare dalla classe base *java.lang.Exception*. A partire da questa è possibile creare per mezzo del meccanismo della ereditarietà nuovi tipi di eccezioni, specializzando il codice a seconda del caso da gestire.

Oggetti throwable

Come abbiamo detto, Java consente di propagare solo alcuni tipi di oggetti. Di fatto, tecnicamente Java richiede che tutti gli oggetti da propagare siano derivati da *java.lang.Throwable*, e questo sembra smentire quanto affermato nel paragrafo precedente in cui affermavamo che devono derivare dalla classe base *java.lang.Exception*. In realtà entrambe le affermazioni sono vere: vediamo perché.

La classe *Throwable* contiene dei metodi necessari a gestire lo "stack tracing[13]" per la propagazione dell'oggetto a ritroso lungo la sequenza corrente delle chiamate, ed ha due costruttori:

Throwable();
Throwable(String);

Entrambi i costruttori di Throwable avviano lo "stack tracing", il secondo in più inizializza un dato membro *String* con un messaggio di stato dettagliato e, accessibile attraverso il metodo *toString()* ereditato da *Object*.

[13] Al fine garantire la propagazione a ritroso, java utilizza uno stack (LIFO) per determinare la catena dei metodi chiamanti e risalire nella gerarchia.

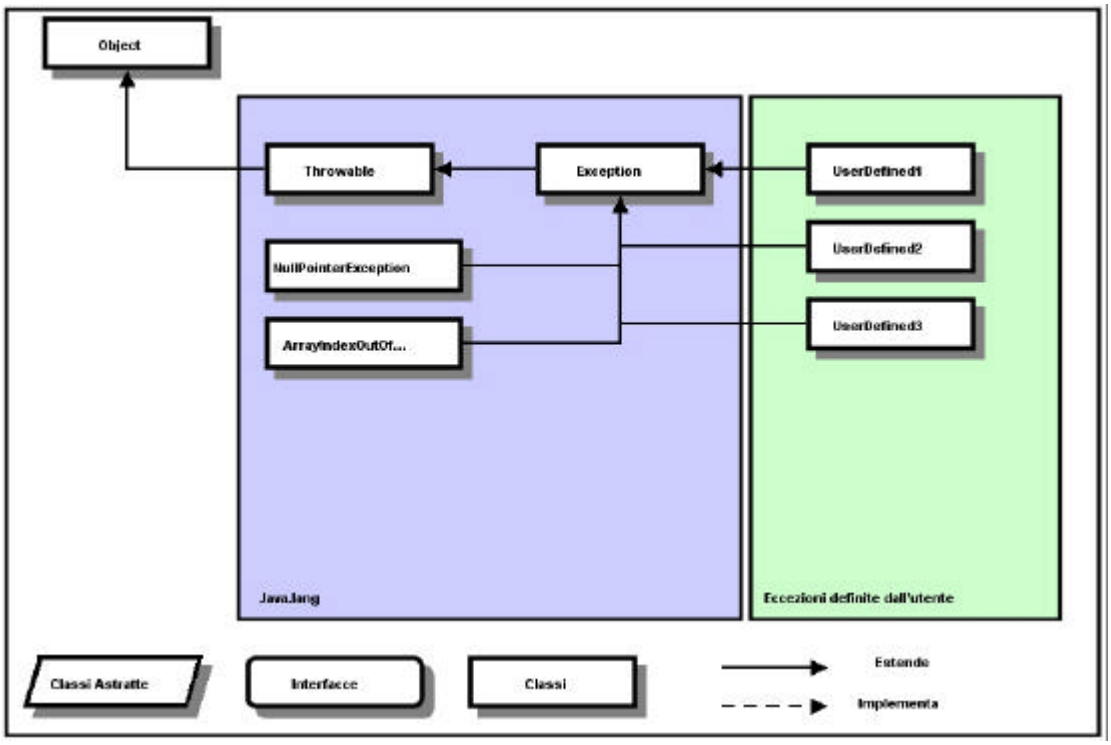


Figura 7-3 : *albero di derivazione delle eccezioni*

Per poter gestire la propagazione dell'oggetto lungo la sequenza delle chiamate, i due costruttori effettuano una chiamata al metodo ***public*** *Throwable fillInStackTrace()*, il quale registra lo stato dello stack di sistema.

Il metodo ***public void*** *printStackTrace()* consente invece di stampare sullo standard error la sequenza restituita dal metodo precedente. Consideriamo l'esempio seguente:

../javanet/mattone/cap7/ClasseEsempio.java

```
class ClasseEsempio
{
    public static void main(String[] argv)
     {
       metodo1(null);
    }
    static void metodo1 (int[] a)
     {
       metodo2 (a);
    }

    static void metodo2(int[] b)
     {
       System.out.println(b[0]);
    }
}
```

Il metodo *main()* della applicazione esegue una chiamata la *metodo1()* passandogli un array nullo, array che viene passato a sua volta al *metodo2()* che effettua la stampa a video del primo elemento. Essendo nullo l'array, al momento della chiamata al metodo system da parte del *metodo2()*, l'applicazione produce una eccezione di tipo *java.lang.NullPointerException*.

Dopo essere stata compilata ed eseguita, l'applicazione stampa a video il seguente messaggio di errore:

Exception in thread "main" java.lang.NullPointerException
at ClasseEsempio.metodo2(ClasseEsempio.java:14)
at ClasseEsempio.metodo1(ClasseEsempio.java:9)
at ClasseEsempio.main(ClasseEsempio.java:5)

Le righe 2,3,4 del messaggio identificano la sequenza delle chiamate attive, mentre la prima riga restituisce un messaggio come definito nella stringa passata al costruttore della classe.

Per convenzione invece, ogni eccezione definita dal programmatore deve derivare da *java.lang.Exception* che a sua volta deriva da *Throwable* (*Figura 7-3*). Anche questa classe ha due costruttori che coincidono con i costruttori della superclasse.

NullPointerException

Java mette a disposizione del programmatore un gran numero di eccezioni definite per ereditarietà a partire dalla classe *java.lang.Exception*, in grado di descrivere tutte le principali condizioni di errore. Nel box successivo è riportato l'albero di derivazione delle eccezioni definite a partire da Exception:

- class java.lang.**Exception**
  - class java.lang.**ClassNotFoundException**
  - class java.lang.**CloneNotSupportedException**
  - class java.lang.**IllegalAccessException**
  - class java.lang.**InstantiationException**
  - class java.lang.**InterruptedException**
  - class java.lang.**NoSuchFieldException**
  - class java.lang.**NoSuchMethodException**
  - class java.lang.**RuntimeException**
    - class java.lang.**ArithmeticException**
    - class java.lang.**ArrayStoreException**
    - class java.lang.**ClassCastException**
    - class java.lang.**IllegalArgumentException**
      - class java.lang.**IllegalThreadStateException**
      - class java.lang.**NumberFormatException**
    - class java.lang.**IllegalMonitorStateException**
    - class java.lang.**IllegalStateException**
    - class java.lang.**IndexOutOfBoundsException**
      - class java.lang.**ArrayIndexOutOfBoundsException**
      - class java.lang.**StringIndexOutOfBoundsException**
    - class java.lang.**NegativeArraySizeException**
    - class java.lang.**NullPointerException**
    - class java.lang.**SecurityException**
    - class java.lang.**UnsupportedOperationException**

L'eccezione *java.lang.NullPointerException* è sicuramente la più comune tra tutte e viene generata tutte le volte che l'applicazione tenti di fare uso di un oggetto nullo. In particolare sono sei le condizioni che possono causare la propagazione di questo oggetto:

Effettuare una chiamata ad un metodo di un oggetto nullo;
Accedere o modificare un dato membro di un oggetto null;
Richiedere la lunghezza di un array nullo;
Accedere o modificare i campi di un array nullo;
Propagare una eccezione nulla (ovvero non instanziata).

Definire eccezioni personalizzate

Oltre alle eccezioni predefinite, quando definiamo nuovi tipi di oggetti, è spesso desiderabile disegnarne nuovi tipi che li accompagnino. Come specificato nei paragrafi precedenti, un nuovo tipo di eccezione deve essere derivata da *java.lang.Exception*. Il funzionamento interno della nuova eccezione non è ristretto da nessuna limitazione.

Nell'esempio successivo definiamo una nuova eccezione di tipo *OutOfDataException*.

../javanet/mattone/cap7/OutOfDataException.java

```
class OutOfDataException extends Exception
{
    String errormessage;
    public OutOfDataException(String s)
    {
            super(s);
            errormessage = s;
    }
    public OutOfDataException()
    {
            super();
            errormessage = "OutOfDataException";
    }
    public String toString()
    {
            return errormessage;
    }
}
```

La nostra eccezione ha due costruttori :

```
public OutOfDataException()
public OutOfDataException(String s)
```

Il primo non prende parametri di input ed inizializza il dato membro *errormessage* di tipo String al valore "OutOfDataException"; il secondo prende come parametro una stringa che utilizza per inizializzare il dato membro *errormessage*.

Infine riscriviamo il metodo *toString()* ereditato dalla classe *Object* affinché restituisca la stringa rappresentata dal dato membro.

L'istruzione throw

Le eccezioni vengono propagate a ritroso attraverso la sequenza dei metodi chiamanti tramite l'istruzione **throw,** che ha sintassi:

throw *Object_Instance ;*

dove Object_Instance è una istanza dell'oggetto Throwable. E' importante tener presente che Object_Instance è una istanza creata mediante l'operatore **new** e non semplicemente un tipo di dato.

Questo metodo causa la terminazione del metodo corrente (come se fosse stata utilizzata l'istruzione **return**), ed invia l'oggetto specificato al metodo chiamante. Non c'è modo da parte del chiamante di riesumare il metodo terminato senza richiamarlo.

La clausola throws

Le eccezioni possono essere propagate solo dai metodi che ne dichiarano la possibilità. Tentare di generare una eccezione all'interno di un metodo che non ha precedentemente dichiarato di avere la capacità di propagare tali oggetti, causerà un errore in fase di compilazione.

Per far questo, è necessario utilizzare la clausola **throws** che indica al metodo chiamante che un oggetto eccezione potrebbe essere generato o propagato dal metodo chiamato.

La clausola **throws** ha sintassi:

return_type method_name (param_list) ***throws*** *Throwable_type*
{
Method Body
}

Un metodo con una clausola throws può generare un oggetto di tipo Throwable_type oppure ogni tipo derivato esso.

Se un metodo contenente una clausola **throws** viene ridefinito (overrided) attraverso l'ereditarietà, il nuovo metodo può scegliere se avere o no la clausola **throws**. Nel caso in cui scelga di avere una clausola **throws**, sarà costretto a dichiarare lo stesso tipo del metodo originale, o al massimo un tipo derivato.

Consideriamo nuovamente la classe *Stack*:

../javanet/mattone/cap7/Stack.java

```
public class Stack
{
    private int maxsize;
    private int data[];
    private int first;

    public Stack()
    {
            maxsize = 10;
            data = new int[10];
            first=0;
    }

    public Stack (int size)
    {
            maxsize=size;
            data = new int[size];
            first= 0;
    }

    int pop()
    {
            if (first > 0)
            {
```

```
                first--;
                return data[first];
        }
        return 0; // Bisogna tornare qualcosa
    }
    void push(int i)
    {

        if (first < maxsize)
        {
                data[first] = i;
                first++;
        }
    }

}
```

In particolare analizziamo nuovamente nei dettagli il metodo membro *push(**int**)*:

```
    void push(int i)
    {

        if (first < maxsize)
        {
                data[first] = i;
                first++;
        }
    }
```

Quando viene chiamato il metodo push, l'applicazione procede correttamente fino a che non si tenti di inserire un elemento all'interno della pila piena. In questo caso no è infatti possibile venire a conoscenza del fatto che l'elemento è andato perduto.

Utilizzando il meccanismo delle eccezioni è possibile risolvere il problema modificando leggermente il metodo in modo che generi una eccezione se si sta tentando di inserire un elemento nella pila piena. Per far questo utilizziamo l'eccezione definita nei paragrafi precedenti:

```
    void push(int i) throws OutOfDataException
    {

        if (first < maxsize)
        {
                data[first] = i;
                first++;
        }
        else throw new OutOfDataException("Impossibile inserire l'elemento");
   }
```

La nuova versione della nostra classe ora genererà una condizione di errore segnalando alla applicazione l'anomalia ed evitando che dati importanti vadano perduti.

Istruzioni try / catch

Una volta generata una eccezione, l'applicazione è destinata alla terminazione a meno che l'oggetto propagato non venga catturato prima di raggiungere l'entry-point del programma o direttamente al suo interno.

Questo compito spetta alla istruzione **catch**. Questa istruzione fa parte di un insieme di istruzioni dette **istruzioni guardiane**, deputate alla gestione delle eccezioni ed utilizzate per racchiudere e gestire le chiamate a metodi che le generano.

L'istruzione **catch** non può gestire da sola una eccezione, ma deve essere sempre accompagnata da un blocco **try**. Il blocco **try** è utilizzato come guardiano per il controllo di un blocco di istruzioni, potenziali sorgenti di eccezioni. **Try** ha la sintassi seguente:

```
try {
    istruzioni
}
catch (Exception var1) {
    istruzioni
}
catch (Exception var2) {
    istruzioni
}
…..
```

L'istruzione **catch** cattura solamente le eccezioni di tipo compatibile con il suo argomento e solamente quelle generate dalle chiamate a metodi racchiuse all'interno del blocco **try**. Se una istruzione nel blocco **try** genera una eccezione, le rimanenti istruzioni nel blocco non vengono eseguite. L'esecuzione di un blocco **catch** esclude automaticamente tutti gli altri.

Immaginiamo di avere definito una classe con tre metodi: *f1()*, *f2()* ed *f3()*, e supponiamo che i primi due metodi generano rispettivamente una eccezione di tipo *IOException* ed una eccezione di tipo *NullPointerException*. Il pseudo codice seguente rappresenta il blocco di istruzioni guardiane deputate ad intercettare e gestire le eccezioni propagate dai due metodi.

```
try
{
    f1();  //Una eccezione in questo punto fa saltare f2() e f3()
    f2();  //Una eccezione in questo punto fa saltare f3()
    f3();
}
catch (IOException _e) {
    System.out.println(_e.toString())
}
catch (NullPointerException _npe) {
    System.out.println(_npe.toString())
}
```

Nel caso in cui sia il metodo f1() a generare e propagare l'eccezione, il flusso della esecuzione delle istruzioni all'interno del blocco try viene interrotto ed il controllo passa al blocco catch che dichiara di gestire l'eccezione generata:

```
catch (IOException _e) {
    System.out.println(_e.toString())
}
```

Se invece è il metodo *f2()* a propagare l'eccezione, *f1()* termina correttamente l'esecuzione, *f3()* non viene eseguito ed il controllo passa la blocco catch

```
catch (NullPointerException _npe) {
    System.out.println(_npe.toString())
}
```

Infine, se nessuna eccezione viene generata I tre metodi vengono eseguiti correttamente ed il controllo passa alla prima istruzione immediatamente successiva al blocco try/catch.

Singoli catch per eccezioni multiple

Differenziare i blocchi **catch** affinché gestiscano ognuno particolari condizioni di errore identificate dal tipo della eccezione propagata consente di poter specializzare il codice affinché possa prendere decisioni adeguate alla soluzione del problema verificatosi.

Esistono dei casi in cui è però possibile trattare più eccezioni utilizzando lo stesso codice. In questi casi è necessario un meccanismo affinché il programmatore non debba replicare inutilmente linee di codice.

Con Java abbiamo la soluzione a portata di mano: abbiamo detto nel paragrafo precedente che ogni istruzione **catch**, cattura solo le eccezioni compatibili con il tipo definito dal suo argomento. Ricordando quanto detto parlando di oggetti compatibili, questo significa che una istruzione **catch** cattura ogni eccezione dello stesso tipo definito dal suo argomento o derivata dal tipo dichiarato.

D'altra parte sappiamo che tutte le eccezioni sono definite per ereditarietà a partire dalla classe base *Exception*, compatibile quindi con tutte le eccezioni.

Nell'esempio a seguire, la classe base *Exception* catturerà ogni tipo di eccezione rendendo inutile ogni altro blocco **catch** a seguire.

```
try
{
    f1();  //Una eccezione in questo punto fa saltare f2() e f3()
    f2();  //Una eccezione in questo punto fa saltare f3()
    f3();
}
catch (java.lang.Exception _e) {
    System.out.println(_e.toString())
}
catch (NullPointerException _npe) {
    //Questo codice non verrà mai eseguito
}
```

Utilizzare un tipo base con un istruzione catch, può essere utilizzato per implementare un meccanismo di catch di default. Consideriamo l'esempio seguente:

```
try
{
    f1();  //Una eccezione in questo punto fa saltare f2() e f3()
    f2();  //Una eccezione in questo punto fa saltare f3()
    f3();
}
catch (NullPointerException _npe) {
    //Questo blocco cattura NullPointerException
}
catch (java.lang.Exception _e) {
    //Questo blocco cattura tutte le altre eccezioni generate da f2() ed f3()
    System.out.println(_e.toString())
```

```
}
```

Ricordando la definizione dei metodi *f1()*, *f2()* ed *f3()* ipotizziamo ora che *f3()* possa propagare un nuovo tipo di eccezione differente da quella propagata dagli altri due metodi.

Così come abbiamo configurato il blocco guardiano, il primo blocco catch cattura tutte le eccezioni generate dal metodo *f1()*, tutte le altre sono catturate e gestite dal secondo blocco:

```
catch (java.lang.Exception _e) {
    //Questo blocco cattura tutte le altre eccezioni generate da f2() ed f3()
    System.out.println(_e.toString())
}
```

Questo meccanismo ricorda molto l'istruzione per il controllo di flusso **switch**, ed in particola modo il funzionamento del blocco identificato dalla label **default**. Nell'esempio, una eccezione di tipo NullPointerException verrà catturata da una apposita istruzione **catch**. Tutte le altre eccezioni saranno catturate dal blocco successivo.

Le altre istruzioni guardiane. Finally

Di seguito ad ogni blocco **catch**, può essere utilizzato opzionalmente un blocco **finally**, che sarà sempre eseguito prima di uscire dal blocco **try/catch**. Questo blocco vuole fornire ad un metodo la possibilità di eseguire sempre un certo insieme di istruzioni a prescindere da come il metodo manipola le eccezioni.

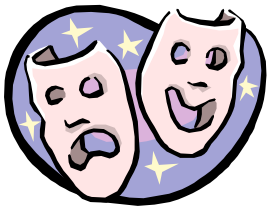
I blocchi **finally** non possono essere evitati dal controllo di flusso della applicazione. Le istruzioni **break**, **continue** o **return** all'interno del blocco **try** o all'interno di un qualunque blocco **catch** verranno eseguito solo dopo l'esecuzione del codice nel blocco **finally**.

```
try
{
    f1();  //Una eccezione in questo punto fa saltare f2() e f3()
    f2();  //Una eccezione in questo punto fa saltare f3()
    f3();
}
catch (java.lang.Exception _e) {
    //Questo blocco cattura tutte le eccezioni
    System.out.println(_e.toString())
    return;
}
finally  {
    //Questo blocco cattura tutte le altre eccezioni
    System.out.println("Questo blocco viene comunque eseguito")
}
```

Solo una chiamata del tipo *System.exit()* ha la capacità di evitare l'esecuzione del blocco di istruzioni in questione. La sintassi completa per il blocco guardiano diventa quindi:

```
try {
    istruzioni
}
catch (Exception var1) {
    istruzioni
```

```
}
catch (Exception var2) {
    istruzioni
}
finally {
    istruzioni
}
```



Capitolo 8
Polimorfismo ed ereditarietà avanzata

Introduzione

L'ereditarietà rappresenta uno strumento di programmazione molto potente; d'altra parte il semplice modello di ereditarietà presentato nel capitolo 6 non risolve alcuni problemi di ereditarietà molto comuni e, se non bastasse, crea alcuni problemi potenziali che possono essere risolti solo scrivendo codice aggiuntivo.

Uno dei limiti più comuni della ereditarietà singola è che non prevede l'utilizzo di una classe base come modello puramente concettuale, ossia priva della implementazione delle funzioni base. Se facciamo un passo indietro, ricordiamo che abbiamo definito uno Stack (pila) come un contenitore all'interno del quale inserire dati da recuperare secondo il criterio "primo ad entrare, ultimo ad uscire". Potrebbe esserci però una applicazione che richiede vari tipi differenti di Stack: uno utilizzato per contenere valori interi ed un altro utilizzato per contenere valori reali a virgola mobile. In questo caso, le modalità utilizzate per manipolare lo Stack sono le stesse, quello che cambia sono i tipi di dato contenuti.

Anche se utilizzassimo la classe Stack definita nel codice seguente come classe base, sarebbe impossibile per mezzo della semplice ereditarietà creare specializzazioni dell'entità rappresentata a meno di riscrivere una parte sostanziale del codice del nostro modello in grado di contenere solo valori interi.

```
class Stack
{
    int data[];
    int ndata;
    void push(int i) {
            ……
    }
    int pop() {
            ……..
        }
}
```

Un altro problema che non viene risolto dal nostro modello di ereditarietà è quello di non consentire **ereditarietà multipla**, ossia la possibilità di derivare una classe da due o più classi base; la parola chiave **extends** prevede solamente un singolo argomento.

Java risolve tutti questi problemi con due variazioni al modello di ereditarietà definito in precedenza: **interfacce** e **classi astratte**. Le interfacce sono entità simili a classi, ma non contengono implementazioni delle funzionalità descritte. Le classi astratte, anch'esse del tutto simili a classi normali, consentono di non implementare tutte le caratteristiche dell'oggetto rappresentato.

Interfacce e classi astratte, assieme, permettono di definire un concetto senza dover conoscere i dettagli di una classe posponendone l'implementazione attraverso il meccanismo della ereditarietà.

Polimorfismo: "un Interfaccia, molti metodi"

Polimorfismo è la terza parola chiave del paradigma ad oggetti. Derivato dal greco, significa "pluralità di forme" ed è la caratteristica che ci consente di utilizzare un'unica interfaccia per una moltitudine di azioni. Quale sia la particolare azione eseguita dipende solamente dalla situazione in cui ci si trova.

Per questo motivo, parlando di programmazione, il polimorfismo viene riassunto nell'espressione **"un'interfaccia, molti metodi".** Ciò significa che possiamo definire una interfaccia unica da utilizzare in molti casi collegati logicamente tra di loro.

Oltre a risolvere i limiti del modello di ereditarietà proposto, Java per mezzo delle interfacce fornisce al programmatore lo strumento per implementare il polimorfismo.

Interfacce

Formalmente, una interfaccia Java rappresenta un prototipo e consente al programmatore di definire lo scheletro di una classe: nomi dei metodi, tipi ritornati, lista dei parametri. Al suo interno il programmatore può definire dati membro purché di tipo primitivo con un'unica restrizione: Java considererà implicitamente questi dati come **static** e **final** (costanti). Quello che una interfaccia non consente, è la implementazione del corpo dei metodi.

Una interfaccia stabilisce il protocollo di una classe senza preoccuparsi dei dettagli di implementazione.

Definizione di una interfaccia

La sintassi per implementare una interfaccia è la seguente:

interface *identificatore {*
    corpo_dell_interfaccia
}

Interface è la parola chiave riservata da Java per lo scopo, *identificatore* è il nome interfaccia e *corpo_dell_interfaccia* è una lista di definizioni di metodi e dati membro separati da ";". Ad esempio, l' interfaccia per la nostra classe Stack sarà:

```
interface Stack {
    public void push( int i );
    public int pop();
}
```

Implementare una interfaccia

Dal momento che una interfaccia rappresenta solo il prototipo di una classe, affinché possa essere utilizzata è necessario che ne esista una implementazione che rappresenti una classe istanziabile.

Per implementare una interfaccia, Java mette a disposizione la parola chiave **implements** con sintassi:

class *nome_classe* ***implements*** *interface {*
    corpo_dell_interfaccia
}

La nostra classe Stack potrà quindi essere definita a partire da un modello nel modo seguente:

StackDef.java

```
public interface StackDef {
    public void push( int i );
    public int pop();
}
```

Stack.java

```
class Stack implements Stack {
    public void push( int i )
    {
            ……..
    }
    public int pop()
    {
            ……..
    }
}
```

Quando una classe implementa una interfaccia è obbligata ad implementarne i prototipi dei metodi definiti nel corpo. In caso contrario il compilatore genererà un messaggio di errore. Di fatto, possiamo pensare ad una interfaccia come ad una specie di contratto che il run-time di Java stipula con una classe. Implementando una interfaccia la classe non solo definirà un concetto a partire da un modello logico (molto utile al momento del disegno della applicazione), ma assicurerà l'implementazione di almeno i metodi definiti nell'interfaccia.

Conseguenza diretta sarà la possibilità di utilizzare le interfacce come tipi per definire variabili reference in grado di referenziare oggetti costruiti mediante implementazione di una interfaccia.

```
StackDef s = new Stack();
```

Varranno in questo caso tutte le regole già discusse nel capitolo sesto parlando di variabili reference ed ereditarietà.

Ereditarietà multipla in Java

Se l'operatore **extends** limitava la derivazione di una classe a partire da una sola classe base, l'operatore **implements** ci consente di implementare una classe a partire da quante interfacce desideriamo semplicemente esplicitando l'elenco delle interfacce implementate separate tra loro con una virgola.

class *nome_classe* ***implements*** *interface1, interface2,….interfacen {*
    corpo_dell_interfaccia
}

Questa caratteristica permette al programmatore di creare gerarchie di classi molto complesse in cui una classe eredità la natura concettuale di molte entità.

Se una classe implementa interfacce multiple, la classe dovrà fornire tutte le funzionalità per i metodi definiti in tutte le interfacce.

Classi astratte

Capitano casi in cui questa astrazione deve essere implementata solo parzialmente all'interno della classe base. In questi casi le interfacce sono restrittive (non si possono implementare funzionalità alcune).

Per risolvere questo problema, Java fornisce un metodo per creare classi base astratte ossia classi solo parzialmente implementate.

Le classi astratte possono essere utilizzate come classi base tramite il meccanismo della ereditarietà e per creare variabili reference; tuttavia queste classi non sono complete e, come le interfacce, non possono essere istanziate.

Per estendere le classi astratte si utilizza come nel caso di classi normali l'istruzione extends e di conseguenza solo una classe astratta può essere utilizzata per creare nuove definizioni di classe.

Questo meccanismo fornisce una alternativa alla costrizione di dover implementare tutte le funzionalità di una interfaccia all'interno di una nuova classe aumentando la flessibilità nella creazione delle gerarchie di derivazione.

Per definire una classe astratta Java mette a disposizione la parola chiave **abstract**. Questa clausola informa il compilatore che alcuni metodi della classe potrebbero essere semplicemente prototipi o astratti.

```
abstract class nome_classe
{
    data_members
    abstract_methods
    non_abstract_methods
}
```

Ogni metodo che rappresenta semplicemente un prototipo deve essere dichiarato **abstract**. Quando una nuova classe viene derivata a partire dalla classe astratta, il compilatore richiede che tutti i metodi astratti vengano definiti. Se la necessità del momento costringe a non definire questi metodi, la nuova classe dovrà a sua volta essere definita abstract .

Un esempio semplice potrebbe essere rappresentato da un oggetto *Terminale*. Questo oggetto avrà dei metodi per muovere il cursore, per inserire testo, ecc. Inoltre *Terminale* dovrà utilizzare dei dati per rappresentare la posizione del cursore e le dimensioni dello schermo (in caratteri).

```
public abstract class Terminal
{
    private int cur_row, cur_col, nrows, ncols;
    public Terminal(int rows, int cols)
    {
            nrows = rows;
            ncols = cols;
            cur_row=0;
            cur_col=0;
    }
    public abstract void move_cursor(int rows, int col);
    public abstract void insert_string(int rows, int col);
    ....
    public void clear()
    {
            int r,c;
            for(r=0 ; r<nrows ;r++)
                    for(c=0 ; c<ncols ;c++)
                    {
                            move_cursor(r , c);
```

```
                    insert_string(“ “);
                }
        }

    }
```

Il metodo che pulisce lo schermo (*clear()*) potrebbe essere scritto in termini di altri metodi prima che siano implementati. Quando viene implementato un terminale reale i metodi astratti saranno tutti implementati, ma non sarà necessario implementare nuovamente il metodo *clear()*.

Capitolo 9
Java Threads

Introduzione

Java è un linguaggio multi-thread, cosa che sta a significare che "un programma può essere eseguito logicamente in molti luoghi nello stesso momento". Ovvero, il multithreading consente di creare applicazioni in grado di utilizzare la concorrenza logica tra i processi, continuando a condividere tra i thread lo spazio in memoria riservato ai dati.

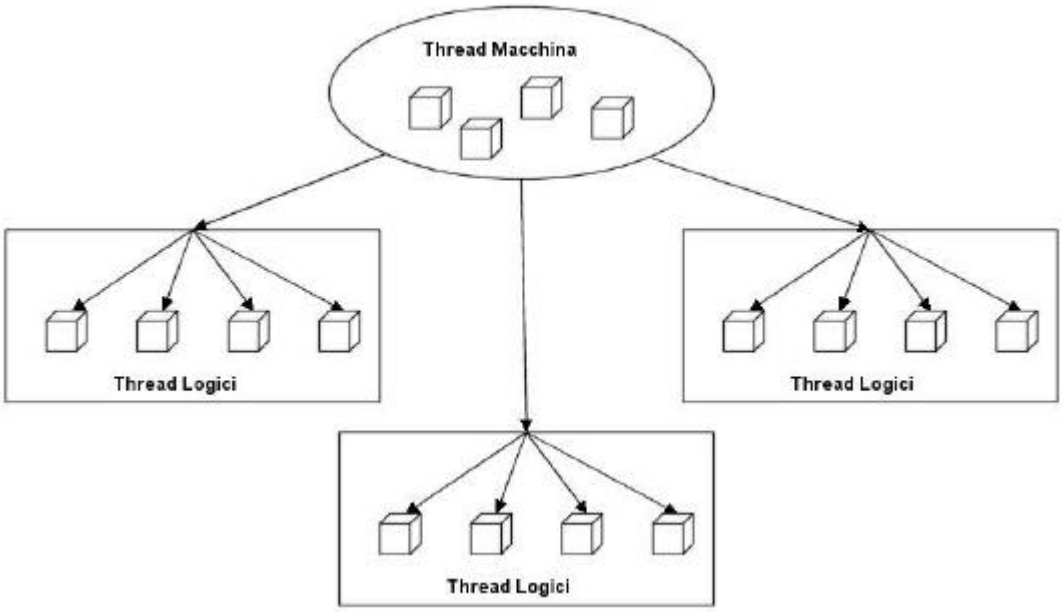


Figura 8-1

Nel diagramma nella ***Figura 8-1***, viene schematizzato l'ipotetico funzionamento della concorrenza logica. Dal punto di vista dell'utente, i thread logici appaiono come una serie di processi che eseguono parallelamente le loro funzioni. Dal punto di vista della applicazione rappresentano una serie di processi logici che, da una parte condividono la stessa memoria della applicazione che li ha creati, dall'altra concorrono con il processo principale al meccanismo di assegnazione della CPU del computer su cui l'applicazione sta girando.

Thread di sistema

In Java, esistono un certo numero di thread che vengono avviati dalla virtual machine in modo del tutto trasparente all'utente.

Esiste un thread per la gestione delle interfacce grafiche responsabile della cattura di eventi da passare alle componenti o dell'aggiornamento dei contenuti dell'interfaccia grafica.

Il garbage collection è un thread responsabile di trovare gli oggetti non più referenziati e quindi da eliminare dallo spazio di memoria della applicazione.

Lo stesso metodo main() di una applicazione viene avviato come un thread sotto il controllo della java virtual machine.

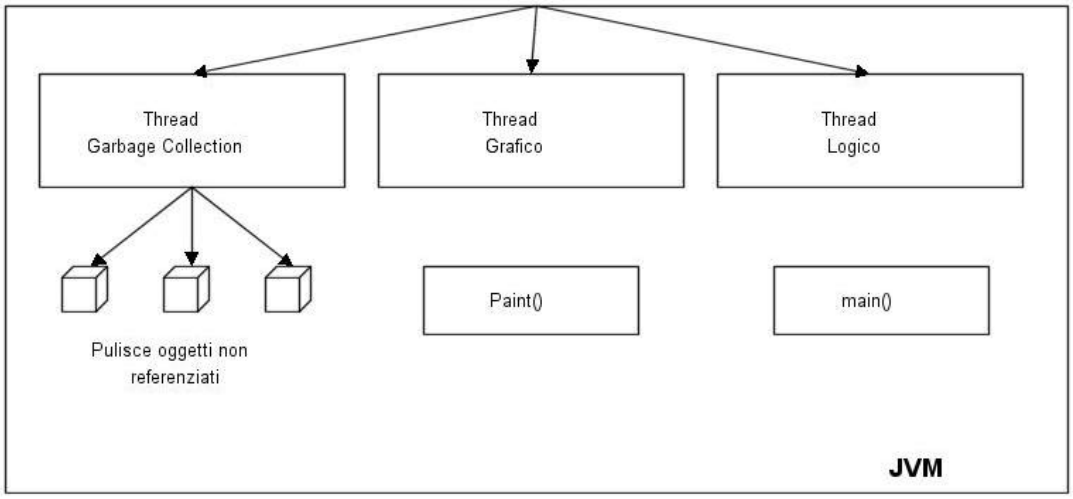


Figura 8-2

Nella *figura 8-2* viene schematizzata la struttura dei principali thread di sistema della virtual machine Java.

La classe java.lang.Thread

In Java, un modo per definire un oggetto thread è quello di utilizzare l'ereditarietà derivando il nuovo oggetto dalla classe base *java.lang.Thread* mediante l'istruzione **extends** (questo meccanismo è schematizzato nella *figura 8-3*).

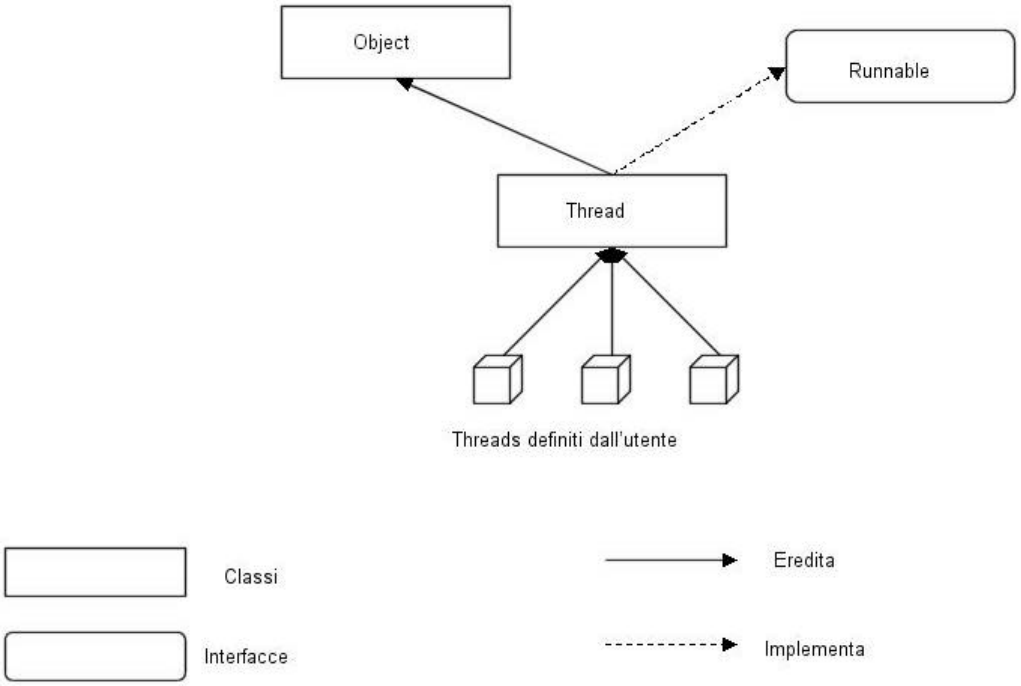


Figura 8-3

La classe *java.lang.Thread* è fornita dei metodi necessari alla esecuzione, gestione e interruzione di un thread. I tre principali:

run() : è il metodo utilizzato per implementare le funzionalità eseguite thread. In genere è l'unico metodo su cui effettuare overriding. Nel caso in cui il metodo non venga soprascritto, al momento della esecuzione eseguirà una funzione nulla;

start() : causa l'esecuzione del thread. La Java Virtual Machine chiama il metodo run() avviando il processo concorrente;

destroy() : distrugge il thread e rilascia le risorse allocate.

Un esempio di thread definito per ereditarietà dalla classe Thread è il seguente:

```
class MioPrimoThread extends Thread {
        int secondi ;
        MioPrimoThread (int secondi) {
           this. secondi = secondi;
        }

        public void run() {
                // ogni slice di tempo definito da secondi
                //stampa a video una frase
        }
```

Interfaccia "Runnable"

Ricordando i capitoli sulla ereditarietà in Java, eravamo arrivati alla conclusione che l'ereditarietà singola è insufficiente a rappresentare casi in cui è necessario creare gerarchie di derivazione più complesse, e che grazie alle interfacce è possibile implementare l'ereditarietà multipla a livello di prototipi di classe.

Supponiamo ora che la nostra classe *MioPrimoThread* sia stata definita a partire da una classe generale diversa da java.lang.Thread. A causa dei limiti stabiliti dalla ereditarietà singola, sarà impossibile creare un thread utilizzando lo stesso meccanismo definito nel paragrafo precedente.

Sappiamo però che mediante l'istruzione **implements** possiamo implementare più di una interfaccia. L'alternativa al caso precedente è quindi quella di definire un oggetto thread utilizzando l'interfaccia *Runnable* di java.lang (*Figura 8-4*).

L'interfaccia Runnable contiene il prototipo di un solo metodo:

```
public interface Runnable
{
   public void run();
}
```

necessario ad indicare l'entry-point del nuovo thread (esattamente come definito nel paragrafo precedente). La nuova versione di MioPrimoThread sarà quindi:

```
class MiaClasseBase{
        MiaClasseBase () {
           .......
        }

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

```
class MioPrimoThread extends MiaClasseBase implements Runnable{
        int secondi ;
        MioPrimoThread (int secondi) {
           this. secondi = secondi;
        }

        public void run() {
                // ogni slice di tempo definito da secondi
                //stampa a video una frase
        }
  }
```

In questo caso, affinché il threads sia attivato, sarà necessario creare esplicitamente una istanza della classe Thread utilizzando il costruttore Thread(Runnable r).

```
MioPrimoThread  miothread  = new MioPrimoThread(5);
Thread nthread = new Thread(miothread);
nthread.start() //provoca la chiamata al metodi run di MioPrimoThread ;
```

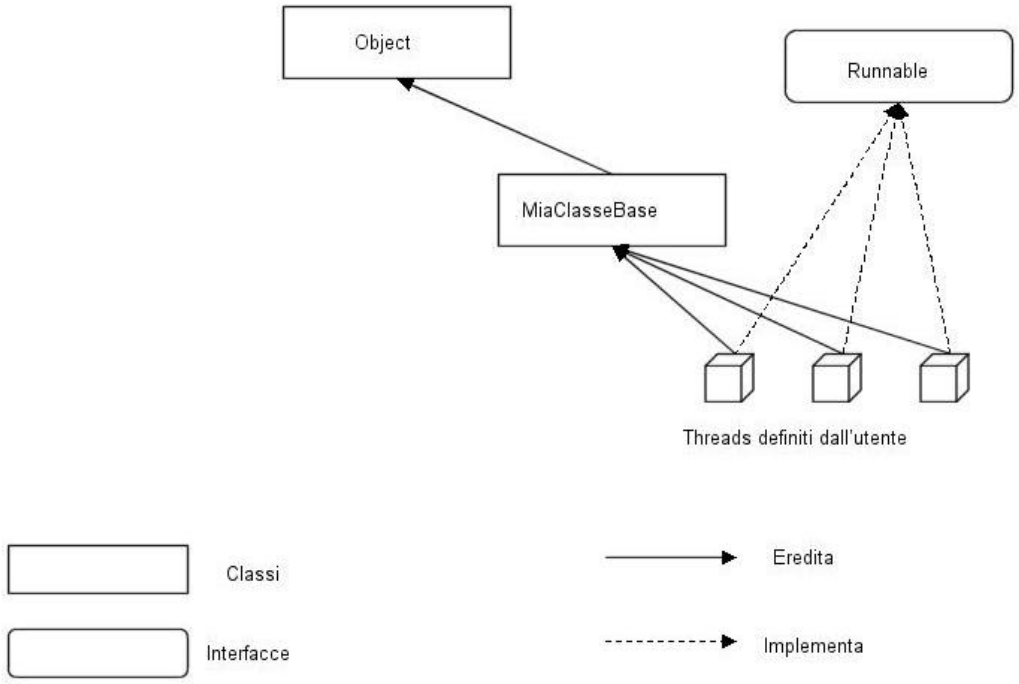


Figura 8-4

Sincronizzare thread

Quando due o più thread possono accedere ad un oggetto contemporaneamente per modificarlo, il rischio a cui si va incontro è quello della corruzione dei dati rappresentati dall'oggetto utilizzato tra i thread in regime di concorrenza. Immaginiamo ad esempio che una classe rappresenti il conto in banca della famiglia Tizioecaio e che il conto sia intestato ad entrambi i signori Tizioecaio. Supponiamo ora che l'oggetto utilizzi un dato membro intero per il saldo del conto pari a lire 1.200.000.

Se i due intestatari del conto (thread) accedessero contemporaneamente per ritirare dei soldi, si rischierebbe una situazione simile alla seguente:

Il signor Tizioecaio accede al conto chiedendo di ritirare 600.000 lire.
Un metodo membro dell'oggetto conto controlla il saldo trovando 1.200.000 lire.
La signora Tizioecaio accede al conto chiedendo di ritirare 800.000 lire.
Un metodo membro dell'oggetto conto controlla il saldo trovando 1.200.000 lire.
Il metodo membro dell'oggetto conto controlla il saldo trovando 1.200.000 lire.
Vengono addebitate 600.000 del signor Tizioecaio.
Vengono addebitate 800.000 della signora Tizioecaio.
Il conto va in scoperto di 200.000 ed i due non lo sanno.

In questi casi è quindi necessario che i due thread vengano sincronizzati ovvero che mentre uno esegue l'operazione, l'altro deve rimanere in attesa. Java fornisce un metodo per gestire la sincronizzazione tra thread mediante la parola chiave **synchronized**.

Questo modificatore deve essere aggiunto alla dichiarazione del metodo per assicurare che solo un thread alla volta sia in grado di utilizzare dati sensibili. Di fatto, indipendentemente dal numero di thread che tenteranno di accedere la metodo, solo uno alla volta potrà eseguirlo. Gli altri rimarranno in coda in attesa di ricevere il controllo. Metodi di questo tipo sono detti "Thread Safe".

Nell'esempio successivo riportiamo due versioni della classe Set di cui, la prima non utilizza il modificatore, la seconda è invece thread safe.

```
class Set // Versione non thread safe
{
    private int data[];
    …..
    boolean isMember(int n)
    {
            //controlla se l'intero appartiene all'insieme
    }
    void add(int n)
    {
            //aggiunge n all'insieme
    }
}

class Set // Versione thread safe
{
    private int data[];
    …..
    synchronized boolean isMember(int n)
    {
            //controlla se l'intero appartiene all'insieme
    }
    void add(int n)
    {
            //aggiunge n all'insieme
    }
}
```

Lock

Il meccanismo descritto nel paragrafo precedente non ha come solo effetto quello di impedire che due thread accedano ad uno stesso metodo contemporaneamente, ma impedisce il verificarsi di situazioni anomale tipiche della programmazione concorrente. Consideriamo l'esempio seguente:

```
class A
{

    …..
    synchronized int a()
    {
            return b()
    }
    synchronized b(int n)
    {
            ……..
            return a();
    }
}
```

Supponiamo ora che un thread T1 chiami il metodo b() della classe A contemporaneamente ad un secondo thread T2 che effettua una chiamata al metodo a() della stessa istanza di classe. Ovviamente, essendo i due metodi sincronizzati, il primo thread avrebbe il controllo sul metodo b() ed il secondo su a(). Lo scenario che si verrebbe a delineare è disastroso inquanto T1 rimarrebbe in attesa sulla chiamata al metodo a() sotto il controllo di T2 e, viceversa T2 rimarrebbe bloccato sulla chiamata al metodo b() sotto il controllo di T1. Questa situazione si definisce *deadlock* e necessità di algoritmi molto complessi e poco efficienti per essere gestita o prevenuta.

Java fornisce il programmatore la certezza che casi di questo tipo non avverranno mai. Di fatto, quando un thread entra all'interno di un metodo sincronizzato ottiene il lock sulla istanza dell' oggetto (non solo sul controllo del metodo). Il thread che ha ottenuto il lock sulla istanza potrà quindi richiamare altri metodi sincronizzati senza entrare in deadlock.

Ogni altro thread che proverà ad utilizzare l'istanza in lock dell'oggetto si metterà in coda in attesa di essere risvegliato al momento del rilascio della istanza da parte del thread proprietario.

Quando il primo thread avrà terminato le sue operazioni, il secondo otterrà il lock e di conseguenza l'uso privato dell'oggetto.

Sincronizzazione di metodi statici

Metodi statici accedono a dati membro statici. Dal momento che ogni classe può accedere a dati membro statici i quali non richiedono che la classe di cui sono membri sia attiva, allora un thread che effettui una chiamata ad un metodo statico sincronizzato non può ottenere il lock sulla istanza dell'oggetto. D'altra parte è necessaria una forma di prevenzione del deadlock anche in questo caso.

Java prevede l'uso di una seconda forma di lock. Quando un thread accede ad un metodo statico sincronizzato, ottiene un lock su classe, ovvero su tutte le istanza della stessa classe di cui sta chiamando il metodo. In questo scenario, nessun thread può accedere a metodi statici sincronizzati di ogni istanza di una stessa classe fino a che un thread detiene il lock sulla classe.

Questa seconda forma di lock nonostante riguardi tutte le istanze di un oggetto, è comunque meno restrittiva della precedente inquanto metodi sincronizzati non statici della classe in lock possono essere eseguiti durante l'esecuzione del metodo statico sincronizzato.

Blocchi sincronizzati

Alcune volte può essere comodo ottenere il lock su una istanza direttamente all'interno di un metodo e ristretto al tempo necessario per eseguire solo alcune istruzioni di codice. Java gestisce queste situazioni utilizzando blocchi di codice sincronizzati.

Nell'esempio a seguire, il blocco sincronizzato ottiene il controllo sull'oggetto *QualcheOggetto* e modifica il dato membro pubblico v.

```
class UnaClasseACaso
{
    public void  a(QualcheOggetto  unaistanza)
    {
            ……..
            synchronized (unaistanza); //ottiene il lock su una istanza
            {
                    unaistanza.v=23;
            }
    }
}
```

Questa tecnica può essere utilizzata anche utilizzando la parola chiave **this** come parametro della dichiarazione del blocco sincronizzato. Nel nuovo esempio, durante tutta la esecuzione del blocco sincronizzato, nessun altro thread potrà accedere alla classe attiva.

```
class UnaClasseACaso
{
    public void  a(QualcheOggetto  unaistanza)
    {
            ……..
            synchronized (this); //ottiene il lock su se stessa
            {
                    unaistanza.v=23;
            }
    }
}
```

Laboratorio 9
Java Thread

Esercizio 1

Creare una applicazione che genera tre thread. Utilizzando il metodo sleep() di java.lang.Thread, il primo deve stampare “I am thread one” una volta al secondo, il secondo “I am thread two” una volta ogni due secondi, ed il terzo “I am thread three” una volta ogni tre secondi.

Esercizio 2

Creare una applicazione che testi il funzionamento della sincronizzazione. Usando i tre thread dell’esercizio precedente, fare in modo che rimangano in attesa un tempo random da 1 a 3 secondi (non più un tempo prefissato) ed utilizzino un metodo sincronizzato statico e uno sincronizzato non statico stampando un messaggio di avviso ogni volta.

Soluzione al primo esercizio

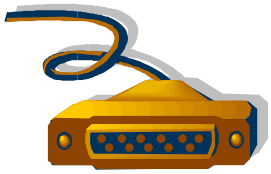
```
import java.lang.*;

public class three_threads implements Runnable
{
    String message;
    int how_long;
    Thread me;
    public three_threads (String msg, int sleep_time) {
            message = msg;
            how_long= sleep_time * 1000;
            me = new Thread(this);
            me.start();
    }
    public void run ()
    {
            while (true) {
                    try {
                       me. Sleep (how_long);
                       System.out.println(message);
                    } catch(Exception e) {}
            }
    }
    public static void main (String g[])
     {
            three_threads tmp;

            tmp= new three_threads("I am thread one",1);
            tmp= new three_threads("I am thread two",2);
            tmp= new three_threads("I am thread three",3);
    }
}
```

Soluzione al secondo esercizio

```
Import java.lang.*;

class shared_thing
{
    public static synchronized void stat_f(locks who)
    {

            System.out.println(who.name + "Locked static");
            try {
                    Who.me.sleep (who.how_long);
            } catch(Exception e) {}
            System.out.println(who.name + "Unlocked static");
    }
    public synchronized void a (locks who)
    {
            System.out.println(who.name + "Locked a");
            try {
                    who.me.sleep (who.how_long);
            } catch(Exception e) {}
            System.out.println(who.name + "Unlocked a");
```

```
  }

  public synchronized void b (locks who)
  {
        System.out.println(who.name + “Locked b”);
        try {
              Who.me.sleep (who.how_long);
        } catch(Exception e) {}
        System.out.println(who.name + “Unlocked b”);
  }
}

  public class locks implements Runnable
  {
        String name;
        int how_long;
        Shared_thing thing;
        Thread me;
        public locks(String n, shared_thing st)
        {
              Name = n ;
              Thing = st;
        }
  public static void main (String a[a])
  {
  locks lk;

        shared_thing t = new shared_thing();
        1k=new locks (“Thread one: “,t );
        1k.me =new Thread(1k);
        1k.me.start();
        1k=new locks (“Thread two: “,t );
        1k.me =new Thread(1k);
        1k.me.start();
        1k=new locks (“Thread three: “,t );
        1k.me =new Thread(1k);
        1k.me.start();
  }
  public void run ()
  {
  int which_func;

        while(true) {
        which_func;
              while(true) {
                    which_func =(int) (Math.random() * 2.5 + 1);
                    how_long = (int) (Mayh.random() * 2000.0 + 1000.0);
                    switch (which_func) {
                          case 1:
                                System.out.println(name + Trying a()”);
                                      Thing.a(this);
                                      Break;
                          case 2:
                                System.out.println(name + Trying b()”);
                                      Thing.b(this);
                                      Break;
                          case 3:
                                System.out.println(name + Trying static()”);
                                      Thing.stat_f(this);
                                      Break;
```

```
                    }
                }
            }
        }
```



Capitolo 11
Java Networking

Introduzione

La programmazione di rete è un tema di primaria importanza inquanto è spesso utile creare applicazioni che forniscano di servizi di rete. Per ogni piattaforma supportata, Java supporta sia il protocollo TCP che il protocollo UDP.

In questo capitolo introdurremo alla programmazione client/server utilizzando Java non prima però di aver introdotto le nozioni basilari su reti e TCP/IP.

I protocolli di rete (Internet)

Descrivere i protocolli di rete e più in generale il funzionamento di una rete è probabilmente cosa impossibile se fatto, come in questo caso, in poche pagine. Cercherò quindi di limitare i danni facendo solo una breve introduzione a tutto ciò che utilizziamo e non vediamo quando parliamo di strutture di rete, ai protocolli che più comunemente vengono utilizzati ed alle modalità di connessione che ognuno di essi utilizza per realizzare la trasmissione dei dati tra client e server o viceversa.

Due applicazioni client/server in rete comunicano tra di loro scambiandosi pacchetti di dati costruiti secondo un comune "*protocollo di comunicazione*" che definisce quella serie di regole sintattiche e semantiche utili alla comprensione dei dati contenuti nel pacchetto.

I dati trasportati all'interno di questi flussi di informazioni possono essere suddivisi in due categorie principali : i dati necessari alla comunicazione tra le applicazioni ovvero i dati che non sono a carico dell'applicativo che invia il messaggio (esempio: l'indirizzo della macchina che invia il messaggio e quello della macchina destinataria) ed i dati contenenti informazioni strettamente legate alla comunicazione tra le applicazioni ovvero tutti i dati a carico della applicazione che trasmette il messaggio. Risulta chiaro che possiamo identificare, parlando di protocolli, due macro insiemi : *Protocolli di rete* e *Protocolli applicativi*.

Appartengono ai "protocolli di rete" i protocolli dipendenti dalla implementazione della rete necessari alla trasmissione di dati tra una applicazione ed un'altra.

Appartengono invece ai "protocolli applicativi" tutti i protocolli che contengono dati dipendenti dalla applicazione e utilizzano i protocolli di rete come supporto per la trasmissione. Nelle *tabelle 1 e 2* vengono riportati i più comuni protocolli appartenenti ai due insiemi.

Protocollo	Descrizione
TCP	Trasmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
ICMP	Internet Control Message Protocol

***Tabella 1** – I più comuni protocolli di rete*

Protocollo	Descrizione
http	Hyper Text Trasfer Protocol
Telnet	Protocollo per la gestione remota via terminale
TP	Time Protocol
Smtp	Simple message trasfer protocollo
Ftp	File trasfer protocol

***Tabella 2** – I più comuni protocolli applicativi in Internet*

Il primo insieme, può essere a sua volta suddiviso in due sottoinsiemi: "Protocolli di trasmissione" e "Protocolli di instradamento". Per semplicità faremo comunque sempre riferimento a questi come protocolli di rete. L'unione dei due insiemi suddetti viene comunemente chiamata "TCP/IP" essendo TCP ed IP i due protocolli più noti ed utilizzati. Nella *Figura 10-1* viene riportato lo schema architetturale del TCP/IP dal quale risulterà più comprensibile la suddivisione nei due insiemi suddetti.

In pratica, possiamo ridurre internet ad un "gestore di indirizzi " il cui compito è quello di far comunicare tra di loro due o più sistemi appartenenti o no alla stessa rete.

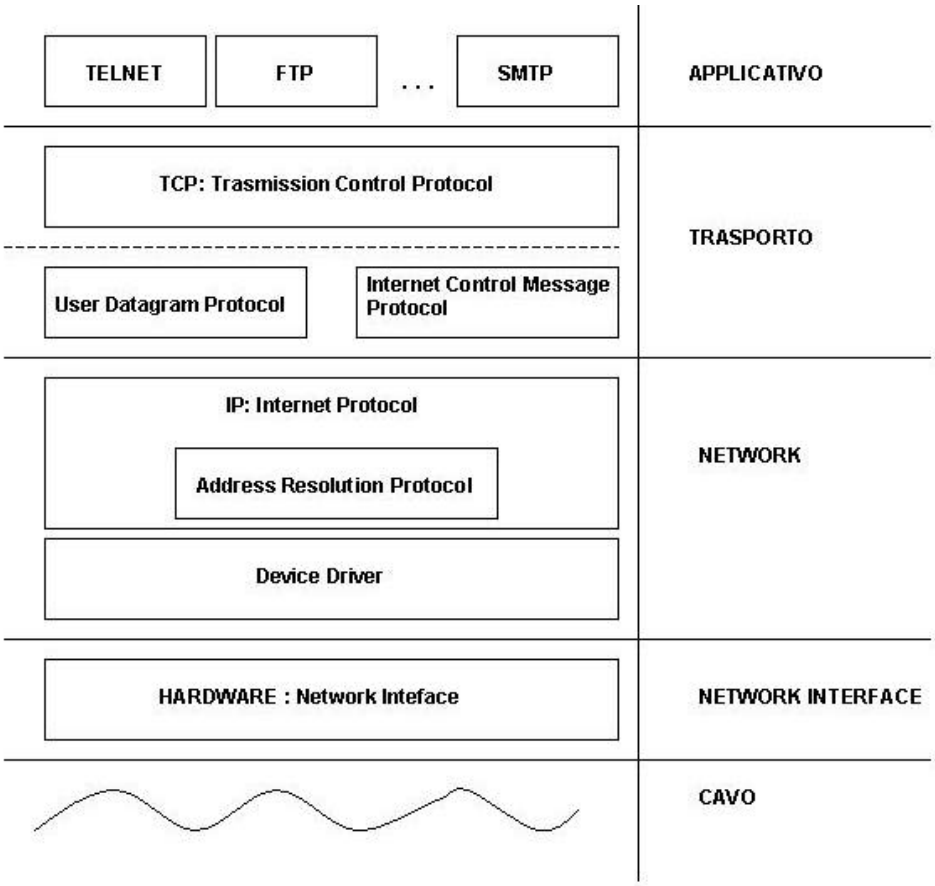


***Figura 10-1** – TCP/IP*

Indirizzi IP

Per inviare pacchetti da una applicazione all'altra è necessario specificare l'indirizzo della macchina mittente e quello della macchina destinataria. Tutti i

computer collegati ad internet sono identificati da uno o più indirizzi numerici detti IP. Tali indirizzi sono rappresentati da numeri di 32 bit e possono essere scritti informato decimale, esadecimale o in altri formati.

Il formato di uso comune è quello che usa la notazione decimale separata da punti mediante il quale l'indirizzo numerico a 32 bit viene suddiviso in quattro sequenze di 1 byte ognuna separate da punto ed ogni byte viene scritto mediante numero intero senza segno. Ad esempio consideriamo l'indirizzo IP *0xCCD499C1* (in notazione esadecimale). Dal momento che:

0xCC	204
0xD4	212
0x99	153
0xC1	193

L'indirizzo IP secondo la notazione decimale separata da punti sarà 204.212.153.193.

Gli indirizzi IP contengono due informazioni utilizzate dal protocollo IP per l'instradamento di un pacchetto dal mittente al destinatario, informazioni che rappresentano l'indirizzo della rete del destinatario e l'indirizzo del computer destinatario all'interno della rete e sono suddivisi in quattro classi differenti : A, B, C, D.

INDIRIZZO IP = INDIRIZZO RETE | INDIRIZZO HOST

Gli indirizzi di "Classe A" sono tutti gli indirizzi il cui primo byte è compreso tra 0 e 127 (ad esempio, appartiene alla "classe A" l'indirizzo IP 10.10.2.11) ed hanno la seguente struttura:

Id. Rete	Identificativo di Host		
0-127	0-255	0-255	0-255

Bit: 0 7,8 15,16 23,24 31

Dal momento che gli indirizzi di rete 0 e 127 sono indirizzi riservati, un IP di classe A fornisce 126 possibili indirizzi di rete e 2^{24} = 16.777.219 indirizzi di host per ogni rete. Appartengono alla "Classe B" gli indirizzi IP in cui il primo numero è compreso tra 128 e 191 (esempio : 129.100.1.32) ed utilizzano i primi due byte per identificare l'indirizzo di rete. In questo caso, l'indirizzo IP assume quindi la seguente struttura :

Id. Rete		Identificativo di Host	
128-191	0-255	0-255	0-255

Bit: 0 7,8 15,16 23,24 31

Notiamo che l'indirizzo IP fornisce 2^{14} = 16.384 reti ognuna delle quali con 2^{16} = 65.536 possibili host. Gli indirizzi di "Classe C" hanno il primo numero compreso tra 192 e 223 (esempio: 192.243.233.4) ed hanno la seguente struttura :

Id. Rete			Id. di Host
192-223	0-255	0-255	0-255

Bit: 0 7,8 15,16 23,24 31

Questi indirizzi forniscono identificatori per 2^{22} = 4.194.304 reti e 2^8 = 256 computer che, si riducono a 254 dal momento che gli indirizzi 0 e 255 sono indirizzi riservati. Gli indirizzi di "Classe D", sono invece indirizzi riservati per attività di "multicasting".

Indirizzo Multicast				
1110……				
Bit: 0	7,8	15,16	23,24	31

In una trasmissione Multicast, un indirizzo non fa riferimento ad un singolo host all'interno di una rete bensì a più host in attesa di ricevere i dati trasmessi. Esistono infatti applicazioni in cui è necessario inviare uno stesso pacchetto IP a più host destinatari in simultanea. Ad esempio, applicazioni che forniscono servizi di streaming video privilegiano comunicazioni multicast a unicast potendo in questo caso inviare pacchetti a gruppi di host contemporaneamente, utilizzando un solo indirizzo IP di destinazione.

Quanto descritto in questo paragrafo è schematizzato nella prossima figura (*Figura 10-2*) da cui appare chiaro che è possibile effettuare la suddivisione degli indirizzi IP nelle quattro classi suddette applicando al primo byte in formato binario la seguente regola :

1) un indirizzo IP appartiene alla "Classe A" se il primo bit è uguale a 0;
2) un indirizzo IP appartiene alla "Classe B" se i primi due bit sono uguali a 10;
3) un indirizzo IP appartiene alla "Classe C" se i primi tre bit sono uguali a 110;
4) un indirizzo IP appartiene alla "Classe D" se i primi 4 bit sono uguali a 1110.

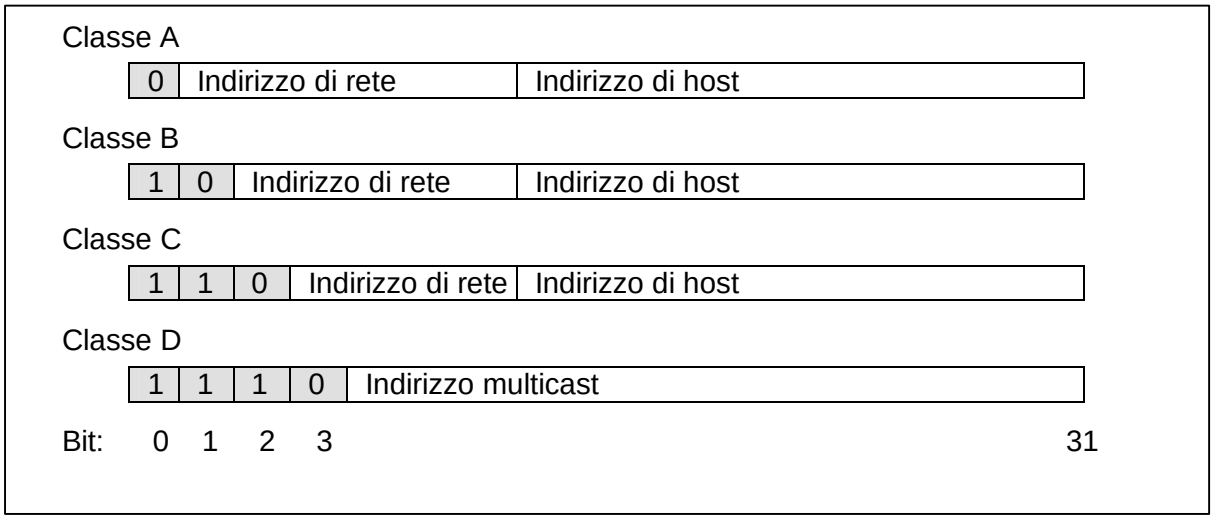


***Figura 10-2** : Le quattro forme di un indirizzo IP*

Comunicazione "Connection Oriented" o "Connectionless"

I protocolli di trasporto appartenenti al primo insieme descritto nei paragrafi precedenti si occupano della trasmissione delle informazioni tra server e client (o viceversa). Per far questo utilizzano due modalità di trasmissione dei dati rispettivamente con connessione (Connection Oriented) o senza (Connectionless).

In modalità "con connessione" il TCP/IP stabilisce un canale logico tra il computer server ed il computer client, canale che rimane attivo sino alla fine della trasmissione dei dati o sino alla chiusura da parte del server o del client. Questo tipo di

comunicazione è necessaria in tutti i casi in cui la rete debba garantire la avvenuta trasmissione dei dati e la loro integrità. Con una serie di messaggi detti di "Acknowledge" server e client verificano lo stato della trasmissione ripetendola se necessario. Un esempio di comunicazione con connessione è la posta elettronica in cui il client di posta elettronica stabilisce un canale logico di comunicazione con il server tramite il quale effettua tutte lo operazione di scarico od invio di messaggi di posta. Solo alla fine delle operazione il client si occuperà di notificare al server la fine della trasmissione e quindi la chiusura della comunicazione.

Nel secondo caso (Connectionless) il TCP/IP non si preoccupa della integrità dei dati inviati ne della avvenuta ricezione da parte del client. Per fare un paragone con situazioni reali, un esempio di protocollo connectionless ci è fornito dalla trasmissione del normale segnale televisivo via etere. In questo caso infatti il trasmettitore (o ripetitore) trasmette il suo messaggio senza preoccuparsi se il destinatario lo abbia ricevuto.

Domain Name System : risoluzione dei nomi di un host

Ricordarsi a memoria un indirizzo IP non è cosa semplicissima, proviamo infatti ad immaginare quanto potrebbe essere complicato navigare in Internet dovendo utilizzare la rappresentazione in "notazione decimale separata da punto" dell'indirizzo del sito internet che vogliamo visitare. Lo standard prevede che oltre ad un indirizzo IP un host abbia associato uno o più nomi "Host Name" che ci consentono di referenziare un computer in rete utilizzando una forma più semplice e mnemonica della precedente. Ecco perché generalmente utilizziamo nomi piuttosto che indirizzi per collegarci ad un computer sulla rete.

Per poter fornire questa forma di indirizzamento esistono delle applicazioni che traducono i nomi in indirizzi (o viceversa) comunemente chiamate "Server DNS" o "nameserver". Analizziamo in breve come funziona la risoluzione di un nome mediante nameserver, ossia la determinazione dell'indirizzo IP a partire dall' "Host Name".

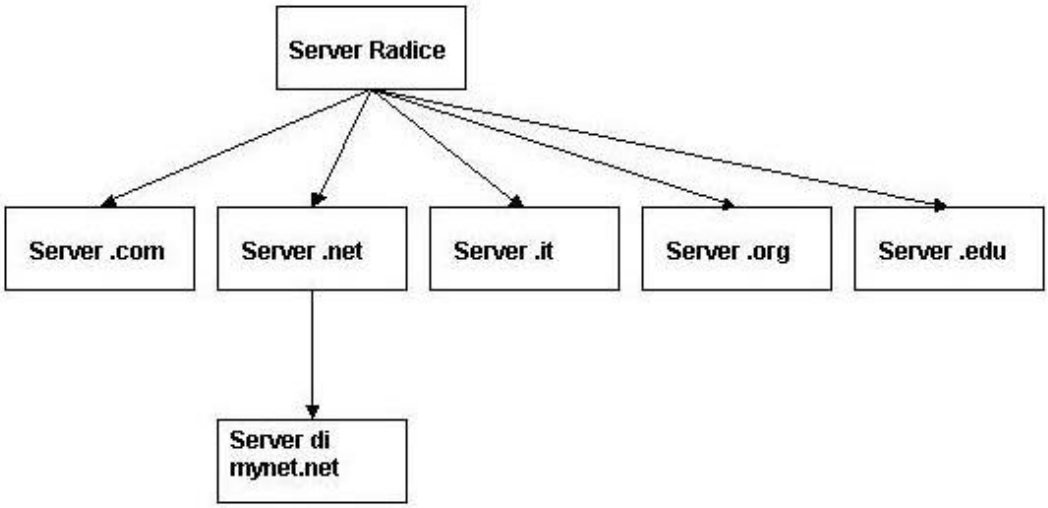


***Figura 10-3**: Gerarchia dei server DNS*

Tipicamente un nome di host ha la seguente forma :

myhost.mynet.net

Dal punto di vista dell'utente il nome va letto da sinistra verso destra ossia dal nome locale (myhost) al nome globale (net) ed ha diversi significati : myhost.mynet.net si riferisce ad un singolo host all'interno di una rete ed è per questo motivo detto "fully qualified"; mynet.net si riferisce al dominio degli host collegati alla rete dell'organizzazione "mynet". Net si riferisce ai sistemi amministrativi nella rete mondiale Internet ed è detto "nome di alto livello"

Un nome quindi definisce una struttura gerarchica, e proprio su questa gerarchia si basa il funzionamento della risoluzione di un nome. Nella *Figura 10-3* è illustrata la struttura gerarchica utilizzata dai server DNS per risolvere i nomi. Dal punto di vista di un server DNS il nome non viene letto da sinistra verso destra, ma al contrario da destra verso sinistra ed il processo di risoluzione può essere schematizzato nel modo seguente:

1. Il nostro browser richiede al proprio server DNS l'indirizzo IP corrispondente al nome **myhost.mynet.net;**
2. Il DNS interroga il proprio database dei nomi per verificare se è in grado di risolvere da solo il nome richiesto. Nel caso in cui il nome esista all'interno della propria base dati ritorna l'indirizzo IP corrispondente, altrimenti deve tentare un'altra strada per ottenere quanto richiesto.
3. Ogni nameserver deve sapere come contattare un almeno un "server radice", ossia un server DNS che conosca i nomi di alto livello e sappia quale DNS è in grado di risolverli. Il nostro DNS contatterà quindi il server radice a lui conosciuto e chiederà quale DNS server è in grado di risolvere i nomi di tipo "net";
4. Il DNS server interrogherà quindi il nuovo sistema chiedendogli quale DNS Server a lui conosciuto è in grado di risolvere i nomi appartenenti al dominio "mynet.net" il quale verrà a sua volta interrogato per risolvere infine il nome completo "myhost.mynet.net" Nella *Figura 10-4* è schematizzato il percorso descritto affinché il nostro DNS ci restituisca la corretta risposta.

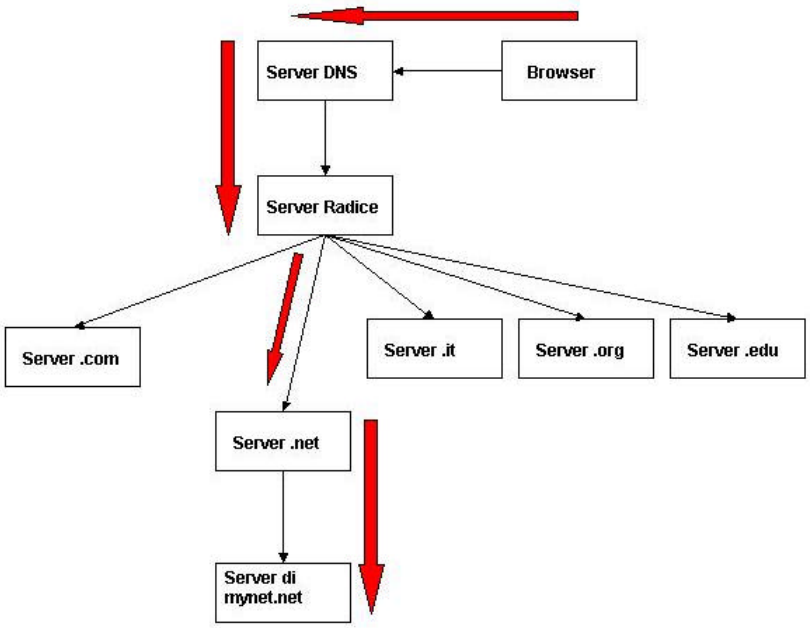


Figura 10-4: *flusso di ricerca all'interno della gerarchia dei server DNS*

URL

Parlando di referenziazione di un computer sulla rete, il nome di un Host non rappresenta ancora il nostro punto di arrivo. Con le nostre conoscenze siamo ora in grado di reperire e referenziare un computer host sulla rete, ma non siamo ancora in grado di accedere ad una risorsa di qualunque tipo essa sia.

Navigando in Internet avrete sentito spesso nominare il termine URL. URL è acronimo di "Uniform Resource Locator" e rappresenta l'indirizzo di una risorsa sulla rete e fornisce informazioni relative al protocollo necessario alla gestione della risorsa indirizzata. Un esempio di URL è il seguente:

http://www.java-net.tv/index.html

Http indica il protocollo (in questo caso Hyper Text Trasfer Protocol) e //www.java-net.tv/index.html è il nome completo della risorsa richiesta. In definitiva un URL ha la seguente struttura :

URL = Identificatore del Protocollo : Nome della Risorsa

Il "Nome della risorsa" può contenere una o più componenti tra le seguenti :

1) Host Name : Nome del computer host su cui la risorsa risiede;
2) File Name : Il path completo alla risorsa sul computer puntato da "Host Name";
3) Port Number : Il numero della porta a cui connettersi (vedremo in seguito il significato di porta);
4) Reference : un puntatore ad una locazione specifica all'interno di una risorsa.

Trasmission Control Protocol : trasmissione Connection Oriented

Il protocollo TCP appartenente allo strato di trasporto del TCP/IP (*Figura 10-1*) trasmette dati da server a client suddividendo un messaggio di dimensioni arbitrarie in frammenti o "datagrammi" da spedire separatamente e non necessariamente sequenzialmente, per poi ricomporli nell'ordine corretto. Una eventuale nuova trasmissione può essere richiesta per l'eventuale pacchetto non arrivato a destinazione o contenenti dati affetti da errori. Tutto questo in maniera del tutto trasparente rispetto alle applicazioni che trasmettono e ricevono il dato.

A tal fine, TCP stabilisce un collegamento logico o connessione tra il computer mittente ed il computer destinatario, creando una sorta di canale attraverso il quale le due applicazioni possono inviarsi dati in forma di pacchetti di lunghezza arbitraria.

Per questo motivo TCP fornisce un servizio di trasporto affidabile garantendo una corretta trasmissione dei dati tra applicazioni. Nella *Figura 10-5* è riportato in maniera schematica il flusso di attività che il TCP/IP deve eseguire in caso di trasmissioni di questo tipo. Queste operazioni, soprattutto su reti di grandi dimensioni, risultano essere molto complicate ed estremamente gravose in quanto comportano che per assolvere al suo compito il TCP debba tener traccia di tutti i possibili percorsi tra mittente e destinatario.

Un'altra capacità del TCP garantita dalla trasmissione in presenza di una connessione è quella di poter bilanciare la velocità di trasmissione tra mittente e destinatario, capacità che risulta molto utile soprattutto nel caso in cui la trasmissione avvenga in presenza di reti eterogenee.

In generale quando due sistemi comunicano tra di loro, è necessario stabilire le dimensioni massime che un datagramma può raggiungere affinché possa esservi

trasferimento di dati. Tali dimensioni sono dipendenti dalla infrastruttura di rete, dal sistema operativo della macchina host e, possono quindi variare anche per macchine appartenenti alla stessa rete.

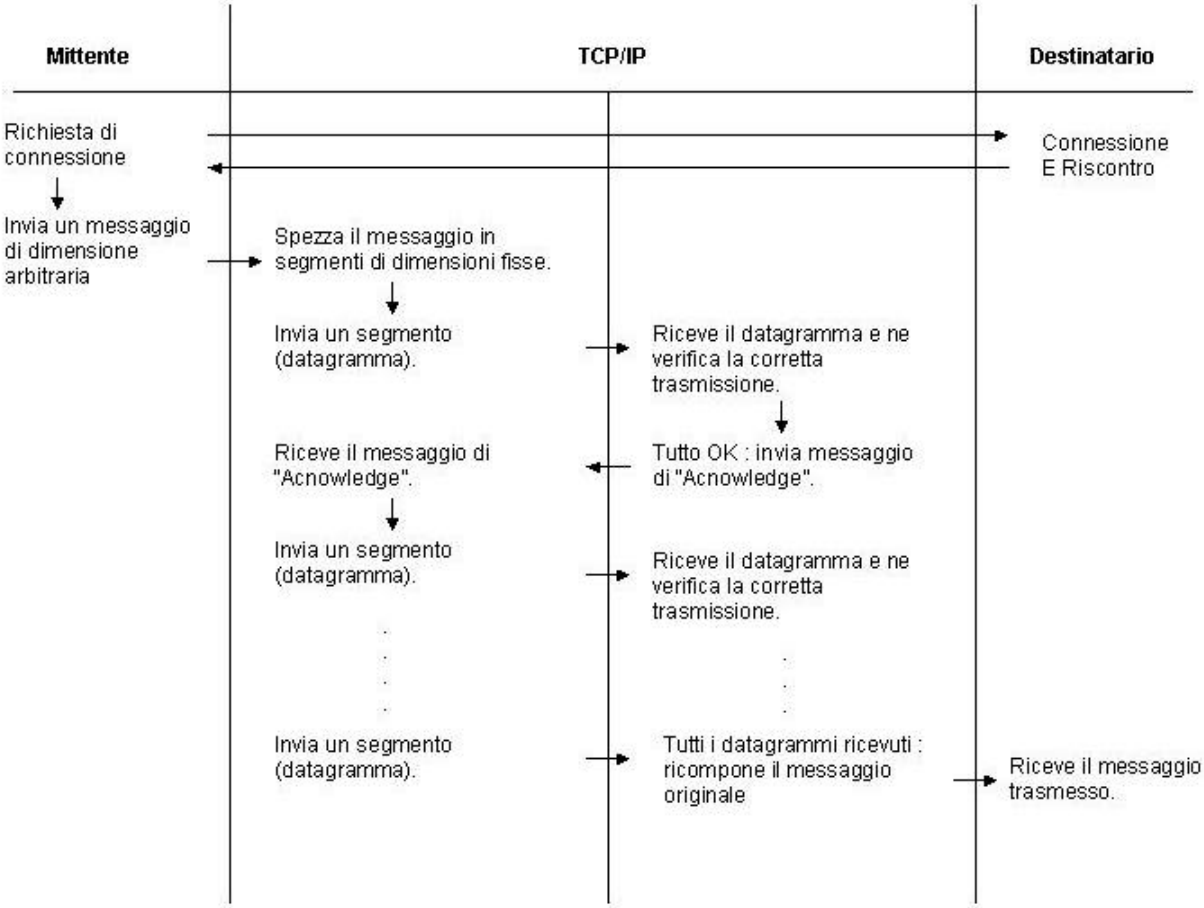


***Figura 10-5**: Time-Line di una trasmissione "Connection Oriented"*

Quando viene stabilita una connessione tra due host, il TCP/IP negozia le dimensioni massime per la trasmissione in ogni direzione dei dati. Mediante il meccanismo di accettazione di un datagramma (acknowledge) viene trasmesso di volta in volta un nuovo valore di dimensione detto "finestra" che il mittente potrà utilizzare nell'invio del datagramma successivo. Questo valore può variare in maniera crescente o decrescente a seconda dello stato della infrastruttura di rete.

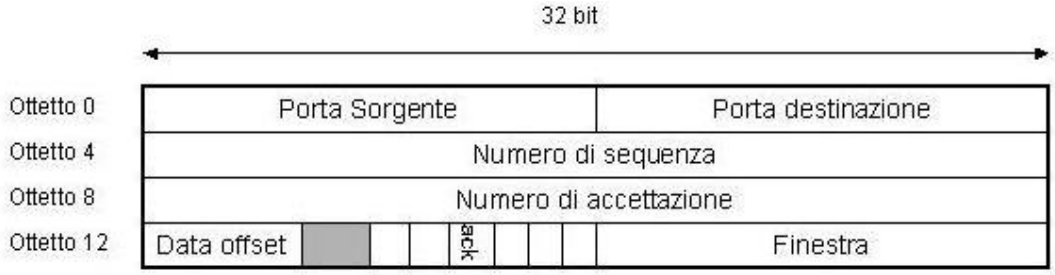


***Figura 10-6 :** I primi 16 ottetti di un segmento TCP*

Nonostante le sue caratteristiche, il TCP/IP non risulta sempre essere la scelta migliore relativamente al metodo di trasmissione di dati. Dovendo fornire tanti servizi, il protocollo in questione oltre ai dati relativi al messaggio da trasmettere deve trasportare una quantità informazioni a volte non necessarie e, che spesso possono diventare causa di sovraccarico sulla rete. In questi casi a discapito della qualità della trasmissione è necessario favorirne la velocità adottando trasmissioni di tipo "Connectionless".

User Datagram Protocol : trasmissione Connectionless

User Datagram Protocol si limita ad eseguire operazioni di trasmissione di un pacchetto di dati tra macchina mittente e macchina destinataria con il minimo sovraccarico di rete senza garanzia di consegna: il protocollo in assenza di connessione non invia pacchetti di controllo della trasmissione perdendo di conseguenza la capacità di rilevare perdite o duplicazioni di dati.

Figura 10-7 : *Datagramma User Datagram Protocol*

Nella *Figura 10-8* viene mostrato il formato del datagramma UDP. Dal confronto con l'immagine 7, appare chiaro quanto siano ridotte le dimensioni dell "header" del datagramma UDP rispetto al datagramma TCP.

Altro limite di UDP rispetto al precedente sta nella mancata capacità di calcolare la "finestra" per la trasmissione dei dati, calcolo che sarà completamente a carico del programmatore. Una applicazione che utilizza questo protocollo ha generalmente a disposizione un buffer di scrittura di dimensioni prefissate su cui scrivere i dati da inviare. Nel caso di messaggi che superino queste dimensioni sarà necessario spezzare il messaggio gestendo manualmente la trasmissione dei vari segmenti che lo compongono. Sempre a carico del programmatore sarà quindi la definizione delle politiche di gestione per la ricostruzione dei vari segmenti nel pacchetto originario e per il recupero di dati eventualmente persi nella trasmissione.

Identificazione di un processo : Porte e Socket

Sino ad ora abbiamo sottointeso il fatto che parlando di rete, esistano all'interno di ogni computer host uno o più processi che devono comunicare tra di loro tramite il TCP/IP con processi esistenti su altri computer host. Viene spontaneo domandarsi come un processo possa usufruire dei servizi messi a disposizione dal TCP/IP.

Il primo passo che un processo deve compiere per poter trasmettere o ricevere dati è quello di identificarsi rispetto al TCP/IP affinché possa essere riconosciuto dallo strato di gestione della rete. Tale identificazione viene effettuata tramite un numero detto "**Port**" o "**Port Address**". Questo numero non è però in grado di descrivere in maniera univoca una connessione (n processi su n host differenti potrebbero avere medesima "Port"). Le specifiche del protocollo definiscono una

struttura chiamata "**Association**", nient'altro che una quintupla di informazioni necessarie a descrivere univocamente una connessione :

Association = (Protocollo,
Indirizzo Locale,
Processo Locale,
Indirizzo Remoto,
Processo Remoto)

Ad esempio è una "Association" la seguente quintupla :

(TCP, 195.233.121.14, 1500, 194.243.233.4, 21)

in cui "TCP" è il protocollo da utilizzare, "195.233.121.14" è l'indirizzo locale ovvero l'indirizzo IP del computer mittente, "1500" è l'identificativo o "Port Address" del processo locale, "194.243.233.4" è l'indirizzo remoto ovvero l'indirizzo IP del computer destinatario ed infine "21" è l'identificativo o "Port Address" del processo remoto.Come fanno quindi due applicazioni che debbano comunicare tra di loro a creare una "Association"? Proviamo a pensare ad una "Association" come ad un insieme contenente solamente i 5 elementi della quintupla. Come si vede dalla *Immagine 9*, tale insieme più essere costruito mediante "unione" a partire da due sottoinsiemi che chiameremo rispettivamente "**Local Half Association**" e "**Remote Half Association**":

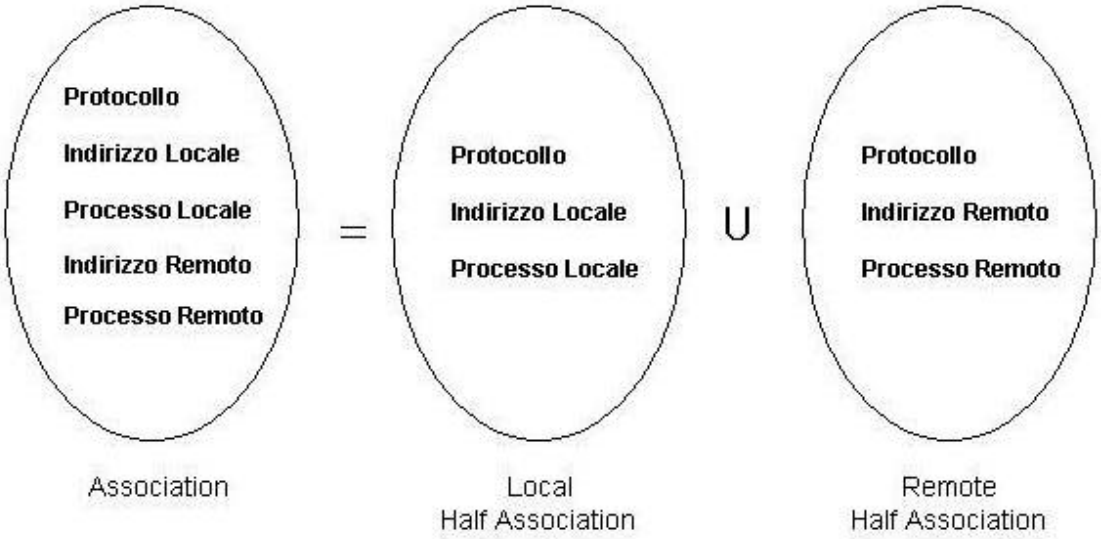


Figura 10-8 : *"Association" = "Local Half Association" U "Remote Half Association"*

Due processi (mittente o locale e destinatario o remoto) definiscono una "Association", stabilendo in modo univoco una trasmissione mediante la creazione di due "Half Association". Le "Half Association" sono chiamate "**Socket**".

Un Socket rappresenta quindi il meccanismo attraverso il quale una applicazione invia e riceve dati tramite rete con un'altra. Consideriamo ad esempio la precedente "Half Association" (TCP, 195.233.121.14, 1500, 194.243.233.4, 21). La quintupla rappresenta in modo univoco una trasmissione TCP tra due applicazioni che comunicano tramite i due Socket :

1 : (TCP, 195.233.121.14, 1500)
2 : (TCP, 194.243.233.4, 21)

Il package java.net

E' il package Java che mette a disposizione del programmatore le API necessarie per lo sviluppo di applicazioni Client/Server basata su socket. Vedremo ora come le classi appartenenti a questo package implementino quanto finora descritto solo concettualmente e come utilizzare gli oggetti messi a disposizione dal package per realizzare trasmissione di dati via rete tra applicazioni. La *Figura 10.9* rappresenta la gerarchia delle classi ed interfacce appartenenti a questo package. Scorrendo velocemente i loro nomi notiamo subito le loro analogie con la nomenclatura utilizzata nei paragrafi precedenti. Analizziamole brevemente.

Gerarchia delle classi

- class java.lang.**Object**
  - class java.net.**Authenticator**
  - class java.lang.**ClassLoader**
    - class java.security.**SecureClassLoader**
      - class java.net.**URLClassLoader**
  - class java.net.**ContentHandler**
  - class java.net.**DatagramPacket**
  - class java.net.**DatagramSocket**
    - class java.net.**MulticastSocket**
  - class java.net.**DatagramSocketImpl** (implements java.net.SocketOptions)
  - class java.net.**InetAddress** (implements java.io.Serializable)
  - class java.net.**PasswordAuthentication**
  - class java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - class java.security.**BasicPermission** (implements java.io.Serializable)
      - class java.net.**NetPermission**
    - class java.net.**SocketPermission** (implements java.io.Serializable)
  - class java.net.**ServerSocket**
  - class java.net.**Socket**
  - class java.net.**SocketImpl** (implements java.net.SocketOptions)
  - class java.lang.**Throwable** (implements java.io.Serializable)
    - class java.lang.**Exception**
      - class java.io.**IOException**
        - class java.net.**MalformedURLException**
        - class java.net.**ProtocolException**
        - class java.net.**SocketException**
          - class java.net.**BindException**
          - class java.net.**ConnectException**
          - class java.net.**NoRouteToHostException**
        - class java.net.**UnknownHostException**
        - class java.net.**UnknownServiceException**
  - class java.net.**URL** (implements java.io.Serializable)
  - class java.net.**URLConnection**
    - class java.net.**HttpURLConnection**
    - class java.net.**JarURLConnection**
  - class java.net.**URLDecoder**
  - class java.net.**URLEncoder**
  - class java.net.**URLStreamHandler**

Gerarchia delle Interfacce

- interface java.net.**ContentHandlerFactory**
- interface java.net.**DatagramSocketImplFactory**
- interface java.net.**FileNameMap**
- interface java.net.**SocketImplFactory**
- interface java.net.**SocketOptions**
- interface java.net.**URLStreamHandlerFactory**

Figura 10-9 : *Gerarchie tra le classi di Java.net*

Un indirizzo IP è rappresentato dalla classe **InetAddress** che fornisce tutti i metodi necessari a manipolare un indirizzo internet necessario all'instradamento dei dati sulla rete e consente la trasformazione di un indirizzo nelle sue varie forme :

getAllByName(String host) *ritorna tutti gli indirizzi internet di un host dato il suo nome in forma estesa (esempio: getAllByName("www.javasoft.com"));*

getByName(String host) *ritorna l'indirizzo internet di un host dato il suo nome in forma estesa;*

getHostAddress() *ritorna una stringa rappresentante l'indirizzo IP dell'host nella forma decimale separata da punto "%d.%d.%d.%d";*

getHostName() *ritorna una stringa rappresentante il nome dell'host.*

Le classi **Socket** e **ServerSocket** implementano rispettivamente un socket client e server per la comunicazione orientata alla connessione, la classe **DatagramSocket** implementa i socket per la trasmissione senza connessione ed infine la classe **MulticastSocket** fornisce supporto per i socket di tipo multicast. Rimangono da menzionare le classi **URL**, **URLConnection**, **HttpURLConnection** e **URLEencoder** che implementano i meccanismi di connessione tra un browser ed un Web Server.

Un esempio completo di applicazione client/server

Nei prossimi paragrafi analizzeremo in dettaglio una applicazione client/server basata su modello di trasmissione con connessione. Prima di entrare nei dettagli della applicazione è necessario però porci una domanda: cosa accade tra due applicazioni che comunichino con trasmissione orientata a connessione?

Il server ospitato su un computer host è in ascolto tramite socket su una determinata porta in attesa di richiesta di connessione da parte di un client (*Figura 10-10*). Il client invia la sua richiesta al server tentando la connessione tramite la porta su cui il server è in ascolto.

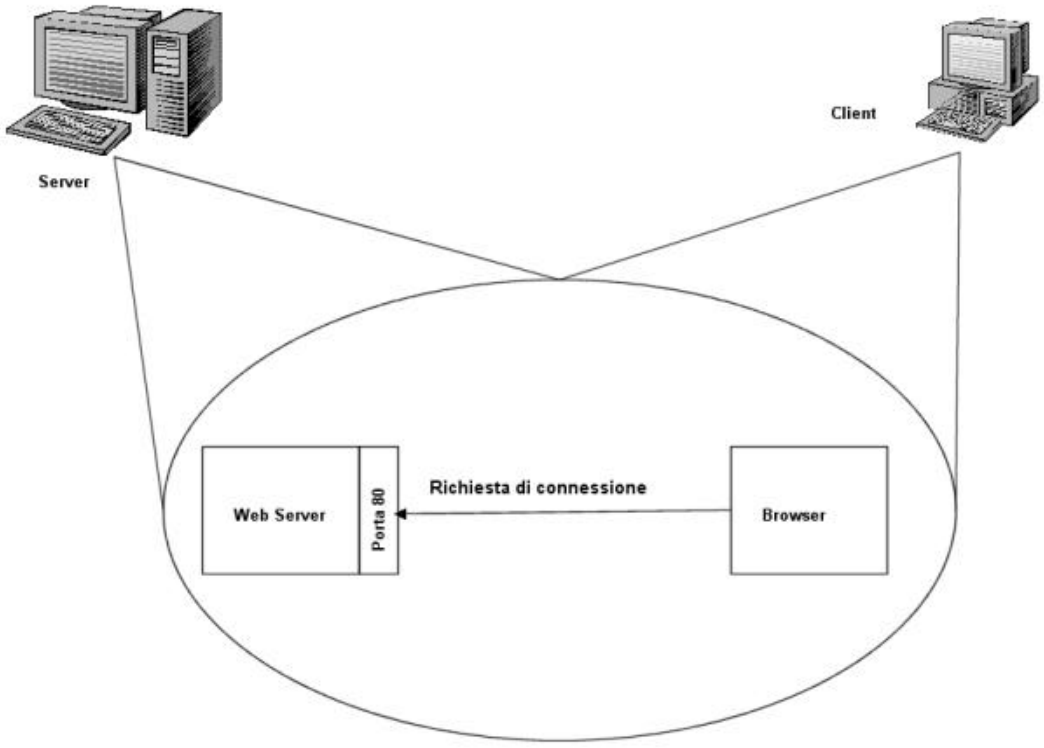


Figura 10-11: *Richiesta di connessione Client/Server*

Dobbiamo a questo punto distinguere due casi: nel primo caso il server comunica con un solo client alla volta accettando una richiesta di connessione solo se nessun'altro client è già connesso (trasmissione unicast); nel secondo caso il server è in grado di comunicare con più client contemporaneamente (trasmissione multicast).

Nel caso di trasmissione unicast, se la richiesta di connessione tra client e server va a buon fine, il server accetta la connessione stabilendo il canale di comunicazione con il client che, a sua volta, ricevuta la conferma alloca un socket su una porta locale tramite il quale trasmettere e ricevere pacchetti (*Figura 10-12*).

Figura 10-12: *Connessione Unicast*

Nel caso di trasmissione multicast, se la richiesta di connessione tra client e server va a buon fine, il server prima di creare il canale di connessione alloca un nuovo socket associato ad una nuova porta (tipicamente avvia un nuovo thread per ogni connessione) a cui cede in gestione la gestione del canale di trasmissione con il client. Il socket principale torna quindi ad essere libero di rimanere in ascolto per una nuova richiesta di connessione (*Figura 10-13*).

Un buon esempio di applicazione client/server che implementano trasmissioni multicast dono browser e Web Server. A fronte di n client (browser) connessi, il server deve essere in grado di fornire servizi a tutti gli utenti continuando ad ascoltare sulla porta 80 (porta standard per il servizio) in attesa di nuove connessioni.

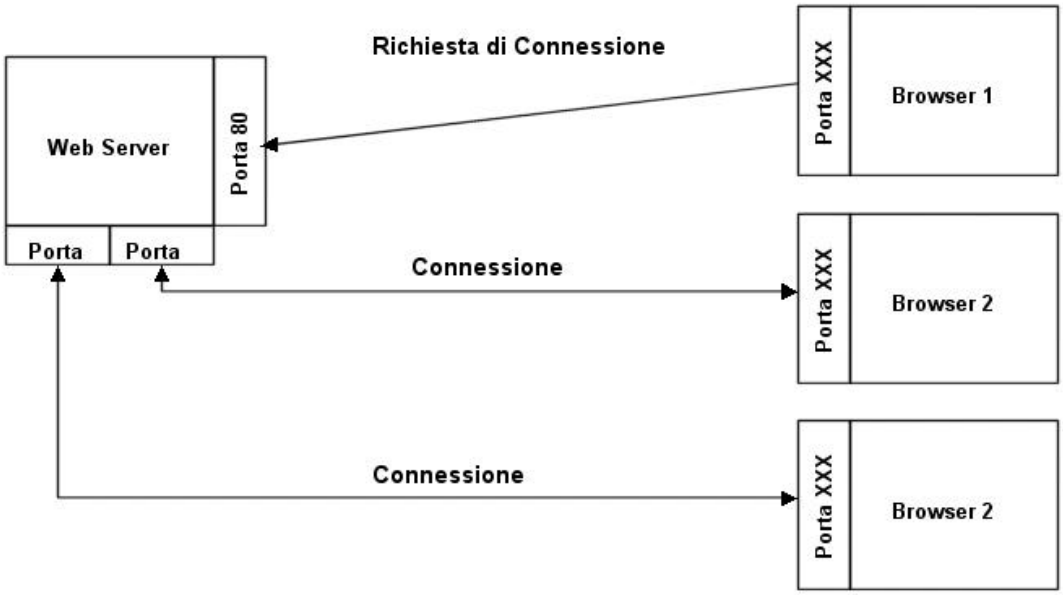


Figura 10-13 : *Connessione Multicast*

La classe ServerSocket

La classe ServerSocket rappresenta quella porzione del server che può accettare richieste di connessioni da parte del client. Questa classe deve essere istanziata passando come parametro il numero della porta su cui il server sarà in ascolto. Il codice di seguito rappresenta un prototipo semplificato della classe ServerSocket.

```
public final class ServerSocket
{
    public ServerSocket(int port) throws IOException …..
    public ServerSocket(int port, int count) throws IOException …..
    public Socket accept() throws IOException ……
    public void close()throws IOException ……
    public String toString() ……
}
```

L'unico metodo realmente necessario alla classe è il metodo *accept()*. Questo metodo mette il thread in attesa di richiesta di connessioni da parte di un client sulla porta specificata nel costruttore. Quando una richiesta di connessione va a buon fine, viene creato il canale di collegamento e il metodo ritorna un oggetto Socket connesso con il client.

Nel caso di comunicazione multicast, sarà necessario creare un nuovo thread a cui passare il socket connesso al client.

La classe Socket

La classe Socket rappresenta una connessione client/server via TCP su entrambi i lati: client e server. La differenza tra server e client sta nella modalità di creazione di un oggetto di questo tipo. A differenza del server in cui un oggetto Socket viene creato dal metodo *accept()* della classe ServerSocket, il client dovrà provvedere a creare una istanza di Socket manualmente.

```
public final class Socket
{
    public Socket (String host, int port) ……
    public int getPort() ……
    public int getLocalPort() ……
    public InputStream getInputStream() throws IOException ……
    public OutputStream getOutputStream() throws IOException ……
    public synchronized void close () throws IOException ……
    public String toString();
}
```

Quando si crea una istanza manuale della classe Socket, il costruttore messo a disposizione dalla classe accetta due parametri: il primo, rappresenta il nome dell' host a cui ci si vuole connettere; il secondo, la porta su cui il server è in ascolto. Il costruttore una volta chiamato tenterà di effettuare la connessione generando una eccezione nel caso in cui il tentativo non sia andato a buon fine.

Notiamo inoltre che questa classe ha un metodo *getInputStream()* e un metodo *getOutputStream()*. Questo perché un Socket accetta la modalità di trasmissione in entrambi i sensi (entrata ed uscita).

I metodi *getPort()* e *getLocalPort()* possono essere utilizzati per ottenere informazioni relative alla connessione, mentre il metodo *close()* chiude la connessione rilasciando la porta libera di essere utilizzata per altre connessioni.

Un semplice thread di servizio

Abbiamo detto che quando implementiamo un server di rete che gestisce più client simultaneamente, è importante creare dei thread separati per comunicare con ogni client. Questi thread sono detti “thread di servizio”. Nel nostro esempio, utilizzeremo una classe chiamata **Xfer** per definire i servizi erogati dalla nostra applicazione client/server .

```
1.  import java.net.* ;
2.  import java.lang.* ;
3.  import java.io.* ;
4.
5.  class Xfer implements Runnable
6.  {
7.      private Socket connection;
8.      private PrintStream o;
9.      private Thread me;
10.
11.     public Xfer(socket s)
12.     {
13.             connection = s;
14.             me = new Thread(this);
15.             me.start();
16.     }
17.
18.     public void run()
19.     {
20.             try
21.             {
22.                     //converte l’output del socket in un printstream
23.                     o = new PrintStream(connection.getOutPutStream());
24.             } catch(Exception e) {}
25.             while (true)
26.             {
27.                     o.println(“Questo è un messaggio dal server”);
28.                     try
29.                     {
30.                             ma.sleep(1000);
31.                     } catch(Exception e) {}
32.             }
33.     }
34. }
```

Xfer può essere creata solo passandogli un oggetto *Socket* che deve essere già connesso ad un client. Una volta istanziato, Xfer crea ed avvia un thread associato a se stesso (linee 11-16).

Il metodo run() di Xfer converte l’*OutputStream* del *Socket* in un *PrintStream* che utilizza per inviare un semplice messaggio al client ogni secondo.

TCP Server

La classe *NetServ* rappresenta il thread primario del server. E’ questo il thread che accetterà connessioni e creerà tutti gli alti thread del server.

```
1.  import java.io.* ;
2.  import java.net.* ;
3.  import java.lang.* ;
4.
5.  public class NetServ implements Runnable
6.  {
7.      private ServerSocket server;
8.      public NetServ () throws Exception
9.      {
10.             server = new ServerSocket(2000);
11.     }
12.     public void run()
13.     {
14.             Socket s = null;
15.             Xfer x;
16.
17.             while (true)
18.             {
19.                     try {
20.                             //aspetta la richiesta da parte del client
21.                             s = server.accept();
22.                     } catch (IOException e) {
23.                             System.out.println(e.toString());
24.                             System.exit(1);
25.                     }
26.                     //crea un nuovo thread per servire la richiesta
27.                     x = new xfer(s);
28.             }
29.     }
30. }
31.
32. public class ServerProgram
33. {
34.     public static void main(String args[])
35.     {
36.             NetServ n = new NetServ();
37.             (new Thread(n)).start();
38.     }
39. }
```

Il costruttore alla linea 8 crea una istanza di un oggetto ServerSocket associandolo alla porta 2000. Ogni eccezione viene propagata al metodo chiamante. L'oggetto NetServ implementa Runnable, ma non crea il proprio thread. Sarà responsabilità dell'utente di questa classe creare e lanciare il thread associato a questo oggetto.

Il metodo *run()* è estremamente semplice. Alla linea 21 viene accettata la connessione da parte del client. La chiamata *server.accept()* ritorna un oggetto socket connesso con il client. A questo punto, *NetServ* crea una istanza di *Xfer* utilizzando il Socket (linea 27) generando un nuovo thread che gestisca la comunicazione con il client.

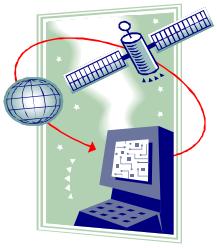
La seconda classe semplicemente rappresenta il programma con il suo metodo *main().* Nelle righe 36 e 37 viene creato un oggetto *NetServ*, viene generato il thread associato e quindi avviato.

Il client

L'oggetto *NetClient* effettua una connessione ad un host specifico su una porta specifica,e legge i dati in arrivo dal server linea per linea. Per convertire i dati di input viene utilizzato un oggetto *DataInputStream*, ed il client utilizza un thread per leggere i dati in arrivo.

```
1.  import java.net.* ;
2.  import java.lang.* ;
3.  import java.io.* ;
4.
5.  public class NetClient implements Runnable
6.  {
7.      private Socket s;
8.      private DataInputStream input;
9.      public NetClient(String host, int port) throws Exception
10.     {
11.             s = new Socket(host, port);
12.     }
13.     public String read() throws Exception
14.     {
15.             if (input == null)
16.                     input = new DataInputStream(s.getInputStream());
17.             return input.readLine();
18.     }
19.     public void run()
20.     {
40.             while (true)
41.             {
42.                     try {
43.                             System.out.println(nc.read());
44.                     } catch (Exception e) {
45.                             System.out.println(e.toString());
46.                     }
21.             }
22.     }
23. }
24.
25. class ClientProgram
26. {
27.     public static void main(String args[])
28.     {
29.             if(args.length<1) System.exit(1);
30.             NetClient nc = new NetClient(args[0], 2000);
31.             (new Thread(nc)).start();
32.     }
33. }
```

Parte Seconda
Applicazioni Server Side



Capitolo 12
Java Enterprise Computing

Introduzione

Il termine “Enterprise Computing” (che per semplicità in futuro indicheremo con EC) è sinonimo di “Distributed Computing” ovvero calcolo eseguito da un gruppo di programmi interagenti attraverso una rete. La *figura 11-1* mostra schematicamente un ipotetico scenario di “Enterprise Computing” evidenziando alcune possibili integrazioni tra componenti server-side. Appare chiara la disomogeneità tra le componenti e di conseguenza la complessità strutturale di questi sistemi.

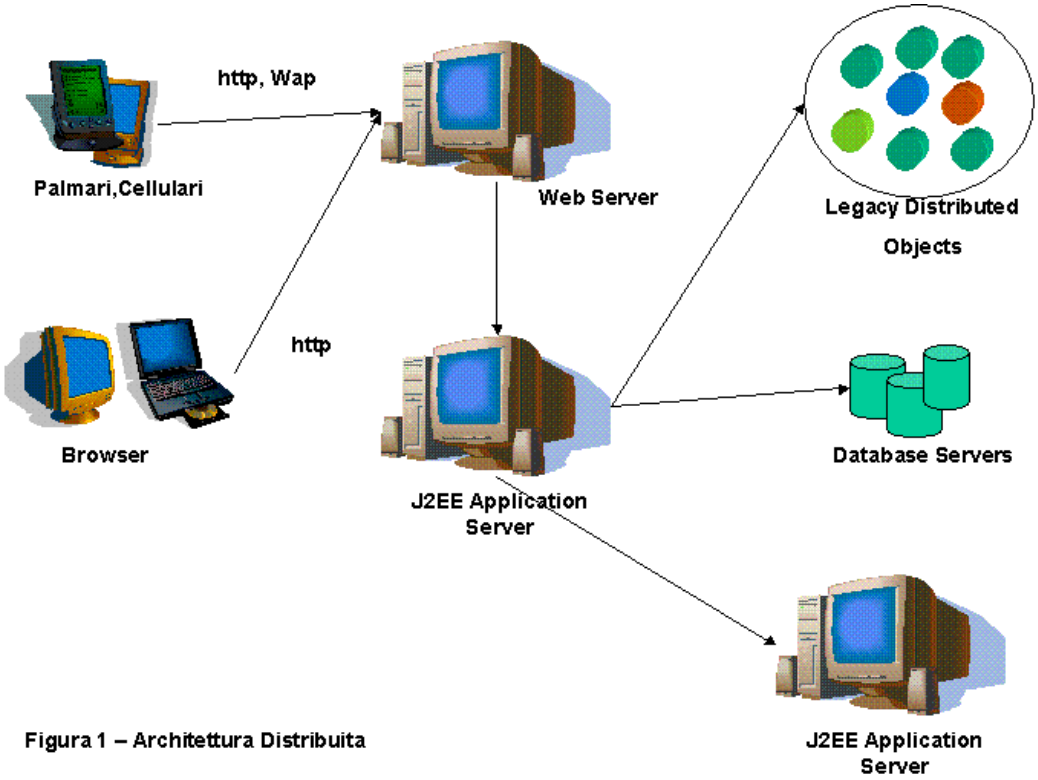


Figura 11-1

Dalla figura appaiono chiari alcuni aspetti. Primo, l’ EC generalmente è legato ad architetture di rete eterogenee in cui la potenza di calcolo è distribuita tra Main Frame, Super Computer e semplici PC. L’unico denominatore comune tra le componenti è il protocollo di rete utilizzato, in genere il TCP/IP. Secondo, applicazioni server di vario tipo girano all’interno delle varie tipologie di hardware. Ed infine l’EC comporta spesso l’uso di molti protocolli di rete e altrettanti standard differenti, alcuni dei quali vengono implementati dall’industria del software con componenti specifiche della piattaforma.

E’ quindi evidente la complessità di tali sistemi e di conseguenza le problematiche che un analista od un programmatore sono costretti ad affrontare. Java tende a semplificare tali aspetti fornendo un ambiente di sviluppo completo di API e metodologie di approccio al problem-solving. La soluzione composta da Sun di

compone di quattro elementi principali: *specifiche*, "*Reference Implementation*", *test di compatibilità* e *"Application Programming Model"*.

Le specifiche elencano gli elementi necessari alla piattaforma e le procedure da seguire per una corretta implementazione con J2EE. La reference implementation contiene prototipi che rappresentano istanze semanticamente corrette di J2EE al fine di fornire all'industria del software modelli completi per test. Include tool per il deployment e l'amministrazione di sistema, EJBs, JSPs, un container per il supporto a runtime e con supporto verso le transazioni, Java Messaging Service e altri prodotti di terzi. L' "Application Programming Model" è un modello per la progettazione e programmazione di applicazioni basato sulla metodologia "best-practice" per favorire un approccio ottimale alla piattaforma. Guida il programmatore analizzando quanto va fatto e quanto no con J2EE, e fornisce le basi della metodologia legata allo sviluppo di sistemi multi-tier con J2EE.

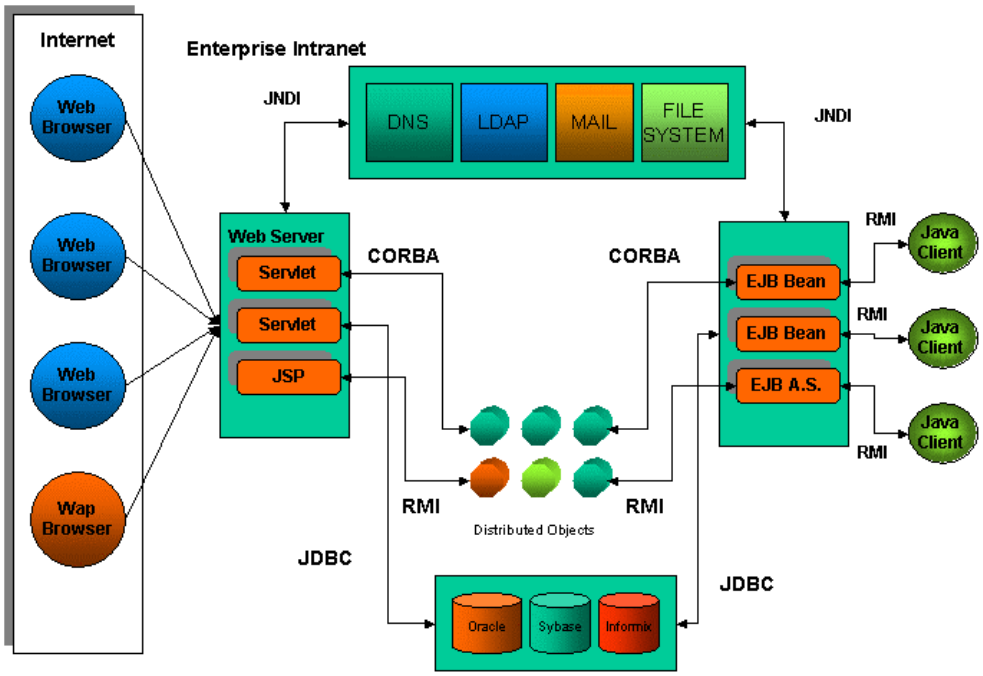


Figura 11-2

Nella *figura 11-2* viene illustrato un modello di EC alternativo al precedente in cui viene illustrato con maggior dettaglio il modello precedente con una particolare attenzione ad alcune delle tecnologie di Sun e le loro modalità di interconnessione.

Architettura di J2EE

L'architettura proposta dalla piattaforma J2EE divide le applicazioni enterprise in tre strati applicativi fondamentali (*figura 11-3*): componenti, contenitori e connettori.

Il modello di programmazione prevede lo sviluppo di soluzioni utilizzando componenti a supporto delle quali fornisce quattro tecnologie fondamentali:

Enterprise Java Beans;
Servelet ;

Java Server Pages ;
Applet.

La prima delle tre che per semplicità denoteremo con EJB fornisce supporto per la creazione di componenti server-side che possono essere generate indipendentemente da uno specifico database, da uno specifico transaction server o dalla piattaforma su cui gireranno. La seconda, servlet, consente la costruzione di servizi web altamente performanti ed in grado di funzionare sulla maggior parte dei web server ad oggi sul mercato. La terza, JavaServer Pages o JSP, permette di costruire pagine web dai contenuti dinamici utilizzando tutta la potenza del linguaggio java. Le applet, anche se sono componenti client-side rappresentano comunque tecnologie appartenenti allo strato delle componenti.

In realtà esiste una quinta alternativa alle quattro riportate per la quale però non si può parlare di tecnologia dedicate, ed è rappresentata dalle componenti server-side sviluppate utilizzando java.

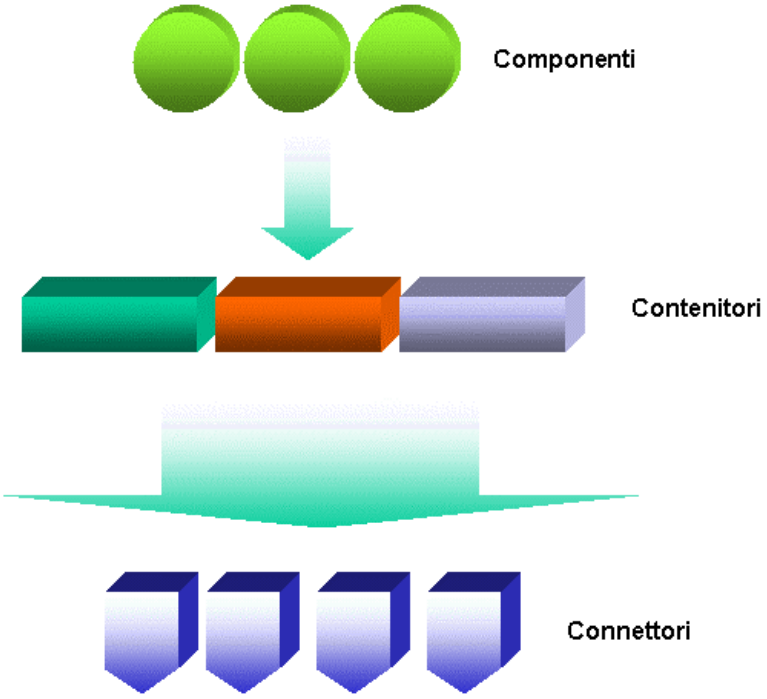


Figura 11-3

In realtà esiste una quinta alternativa alle quattro riportate per la quale però non si può parlare di tecnologia dedicate, ed è rappresentata dalle componenti server-side sviluppate utilizzando java.

Il secondo strato è rappresentato dai contenitori ovvero supporti tecnologici alle tecnologie appartenenti al primo strato logico della architettura. La possibilità di costruire contenitori rappresenta la caratteristica fondamentale del sistema in quanto fornisce ambienti scalari con alte performance.

Infine i connettori consentono alle soluzioni basate sulla tecnologia J2EE di preservare e proteggere investimenti in tecnologie già esistenti fornendo uno strato di connessione verso applicazioni-server o middleware di varia natura: dai database relazionali con JDBC fino ai server LDAP con JNDI.

Gli application-server compatibili con questa tecnologia riuniscono tutti e tre gli strati in un una unica piattaforma standard e quindi indipendente dal codice

proprietario, consentendo lo sviluppo di componenti “server-centric” in grado di girare in qualunque container compatibile con J2EE indipendentemente dal fornitore di software, e di interagire con una vasta gamma di servizi pre-esistenti tramite i connettori.

Ad appoggio di questa soluzione, la Sun mette a disposizione dello sviluppatore un numero elevato di tecnologie specializzate nella soluzione di “singoli problemi”. Gli EJBs forniscono un modello a componenti per il “server-side computing”, Servlet offrono un efficiente meccanismo per sviluppare estensioni ai Web Server in grado di girare in qualunque sistema purché implementi il relativo container. Infine Java Server Pages consente di scrivere pagine web dai contenuti dinamici sfruttando a pieno le caratteristiche di java.

Un ultima considerazione, non di secondaria importanza, va fatta sul modello di approccio al problem-solving: la suddivisione netta che la soluzione introduce tra logiche di business, logiche di client, e logiche di presentazione consente un approccio per strati al problema garantendo “ordine” nella progettazione e nello sviluppo di una soluzione.

J2EE Application Model

Date le caratteristiche della piattaforma, l’aspetto più interessante è legato a quello che le applicazioni non devono fare; la complessità intrinseca delle applicazioni enterprise quali gestione delle transazioni, ciclo di vita delle componenti, pooling delle risorse viene inglobata all’interno della piattaforma che provvede autonomamente alle componenti ed al loro supporto.

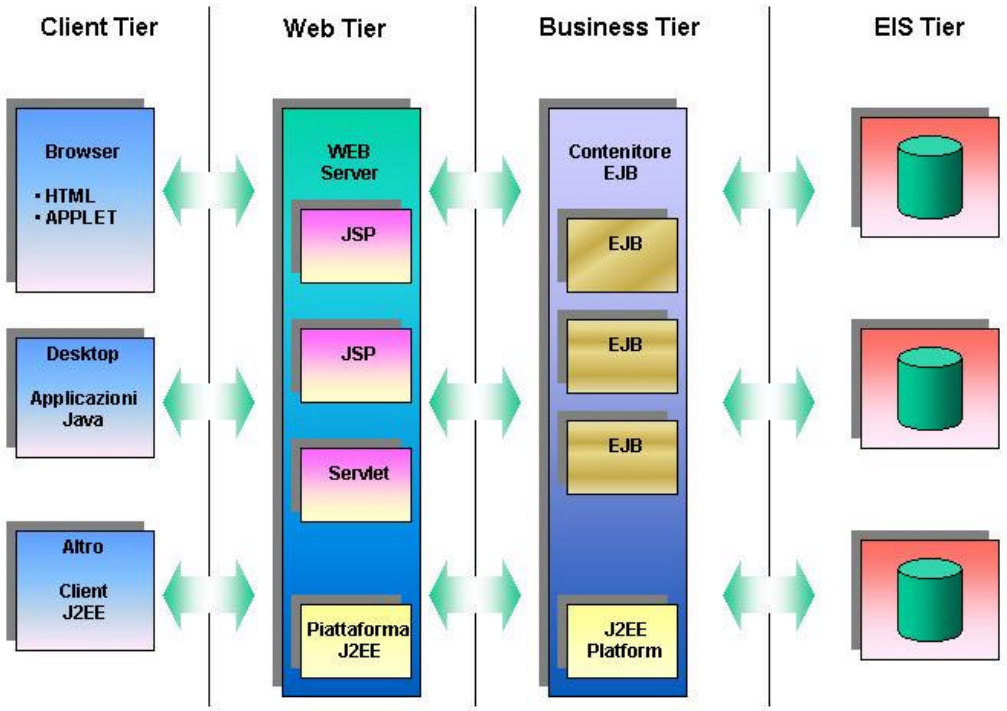


Figura 11-4

Programmatori e analisti sono quindi liberi di concentrarsi su aspetti specifici della applicazione come presentazione o logiche di business non dovendosi

occupare di aspetti la cui complessità non è irrilevante e, in ambito progettuale ha un impatto notevolissimo su costi e tempi di sviluppo.

A supporto di questo, l'Application Model descrive la stratificazione orizzontale tipica di una applicazione J2EE. Tale stratificazione identifica quattro settori principali (*figura 11-4*) : Client Tier, Web Tier, Business-Tier, EIS-Tier.

Client Tier

Appartengono allo strato client le applicazioni che forniscono all'utente una interfaccia semplificata verso il mondo enterprise e rappresentano quindi la percezione che l'utente ha della applicazione J2EE. Tali applicazioni si suddividono in due classi di appartenenza : le applicazioni web-based e le applicazioni non-web-based.

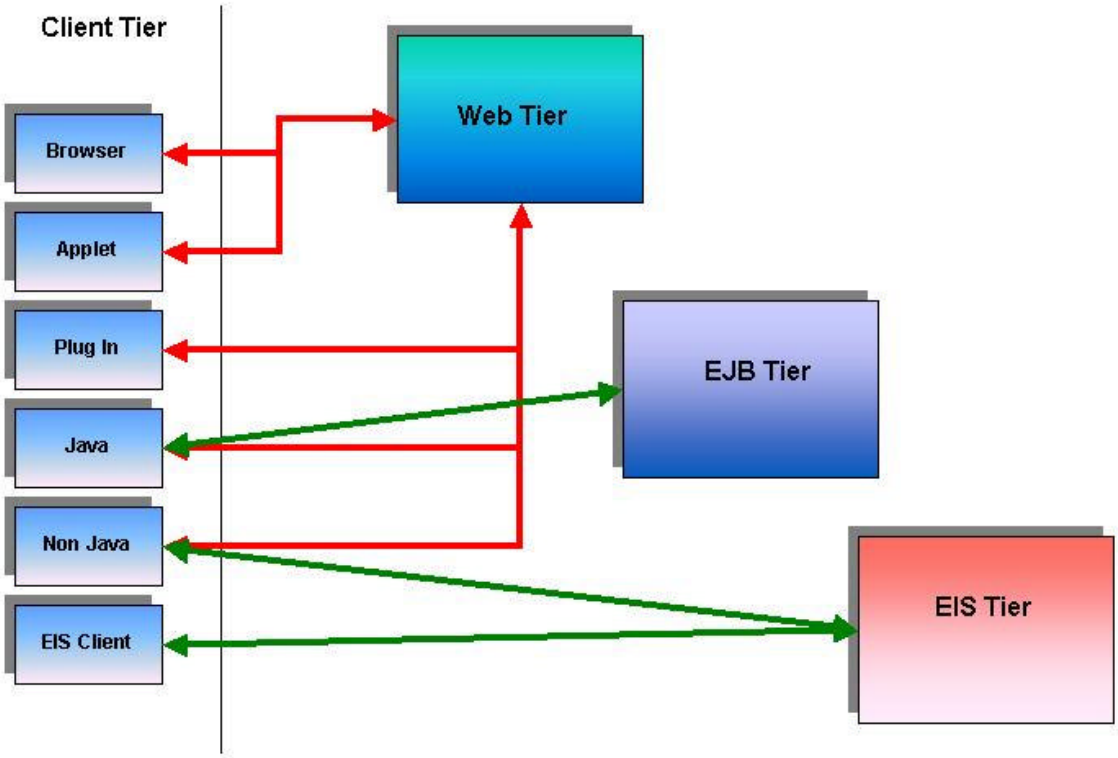


Figura 11-5

Le prime sono quelle applicazioni che utilizzano il browser come strato di supporto alla interfaccia verso l'utente ed i cui contenuti vengono generati dinamicamente da Servlet o Java Server Pages o staticamente in HTML.

Le seconde (non-web-based) sono invece tutte quelle basate su applicazioni stand-alone che sfruttano lo strato di rete disponibile sul client per interfacciarsi direttamente con la applicazione J2EE senza passare per il Web-Tier.

Nella *figura 11-5* vengono illustrate schematicamente le applicazioni appartenenti a questo strato e le modalità di interconnessione verso gli strati componenti la architettura del sistema.

Web Tier

Le componenti web-tier di J2EE sono rappresentate da pagine JSP, server-side applet e Servlet. Le pagine HTML che invocano Servlet o JSP secondo lo standard J2EE non sono considerate web-components, ed il motivo è il seguente : come illustrato nei paragrafi precedenti si considerano componenti, oggetti caricabili e gestibili all'interno di un contenitore in grado di sfruttarne i servizi messi a disposizione. Nel nostro caso, Servlet e JSP hanno il loro ambiente runtime all'interno del "Servlet-Container" che provvede al loro ciclo di vita, nonché alla fornitura di servizi quali client-request e client-response ovvero come un client appartenente allo strato client-tier rappresenta la percezione che l'utente ha della applicazione J2EE, il contenitore rappresenta la percezione che una componente al suo interno ha della interazione verso l'esterno (*Figura 11-6*).

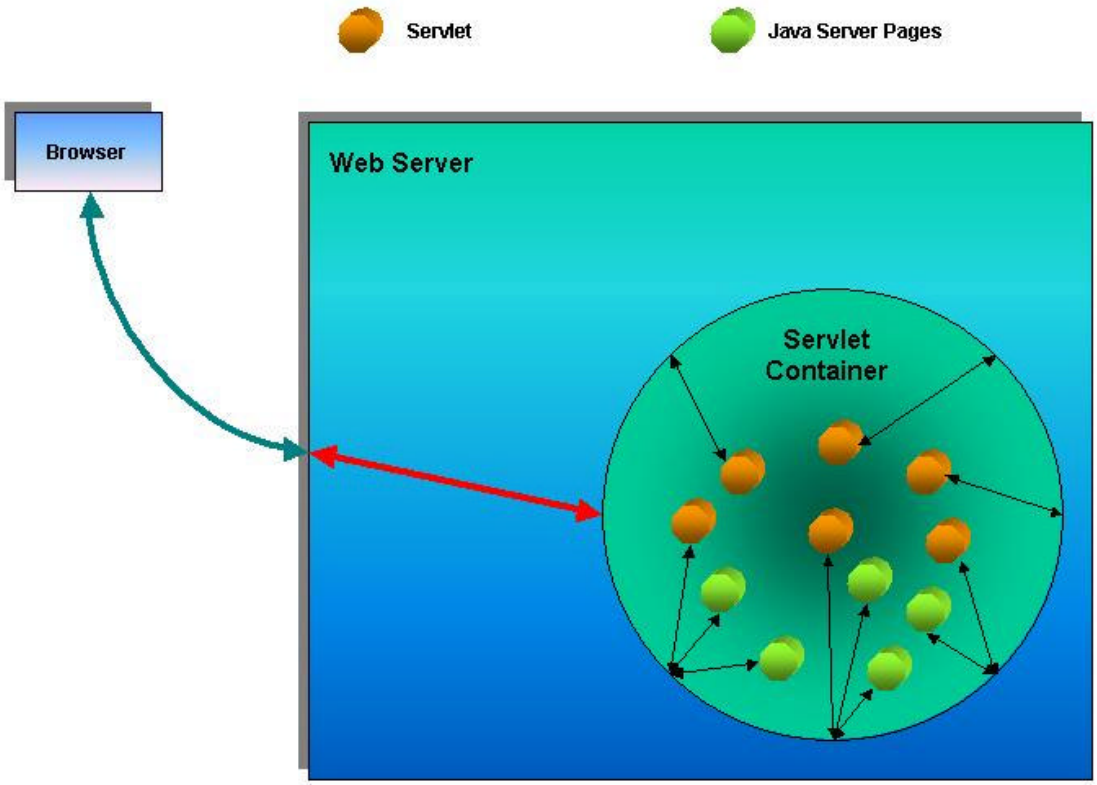


Figura 11-6

La piattaforma J2EE prevede quattro tipi di applicazioni web : "basic HTML", "HTML with base JSP and Servlet", "Servlet and JSP with JavaBeans components", "High Structured Application con componenti modulari", "Servlet ed Enterprise Beans". Di queste, le prime tre sono dette "applicazioni web-centric", quelle appartenenti al quarto tipo sono dette "applicazioni EJB Centric". La scelta di un tipo di applicazione piuttosto che di un altro dipende ovviamente da vari fattori tra cui:

Complessità del problema;
Risorse del Team di sviluppo;
Longevità della applicazione;
Dinamismo dei contenuti da gestire e proporre.

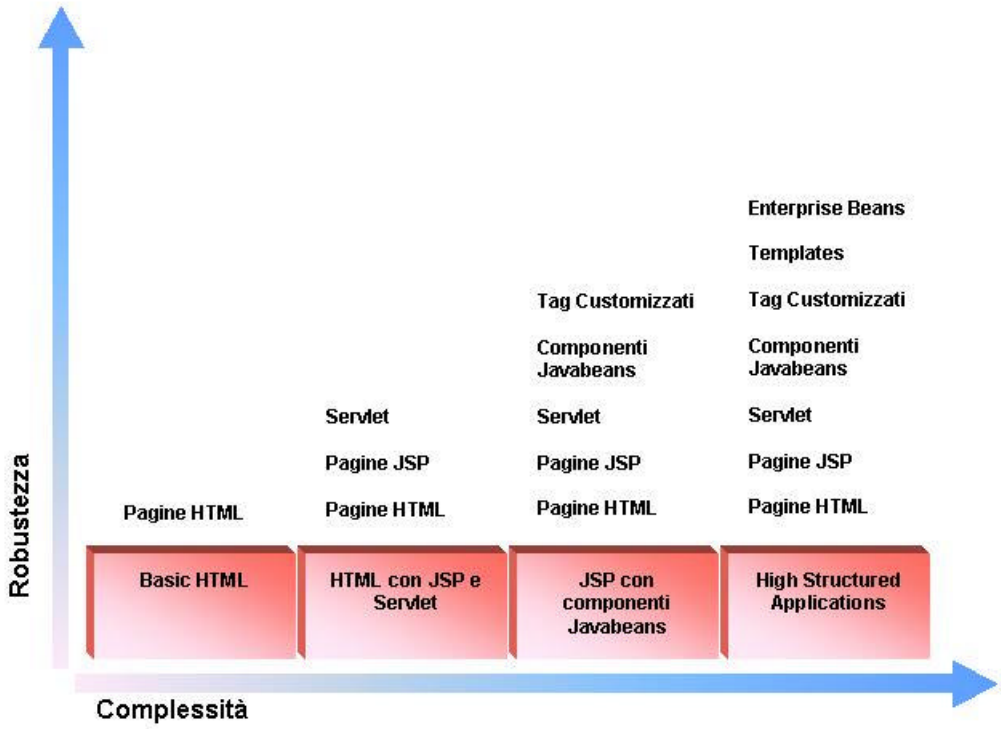


Figura 11-7

Nella *figura 11-77* viene presentata in modo schematico tutta la gamma di applicazioni web ed il loro uso in relazione a due fattori principali : complessità e robustezza.

Business Tier

Nell'ambito di una applicazione enterprise, è questo lo strato che fornisce servizi specifici: gestione delle transazioni, controllo della concorrenza, gestione della sicurezza, ed implementa inoltre logiche specifiche circoscritte all'ambito applicativo ed alla manipolazione dei dati. Mediante un approccio di tipo Object Oriented è possibile decomporre tali logiche o logiche di business in un insieme di componenti ed elementi chiamati "*business object*".

La tecnologia fornita da Sun per implementare i "business object" è quella che in precedenza abbiamo indicato come EJBs (Enterprise Java Beans). Tali componenti si occupano di:

Ricevere dati da un client, processare tali dati (se necessario), inviare i dati allo strato EIS per la loro memorizzazione su base dati;

(Viceversa) Acquisire dati da un database appartenente allo strato EIS, processare tali dati (se necessario), inviare tali dati al programma client che ne abbia fatto richiesta.

Esistono due tipi principali di EJBs : *entity beans* e *session beans.* Per session bean si intende un oggetto di business che rappresenta una risorsa "privata" rispetto al client che lo ha creato. Un entity bean al contrario rappresenta in modo univoco un

dato esistente all'interno dello strato EIS ed ha quindi una sua precisa identità rappresentata da una chiave primaria (*figura 11-8*).

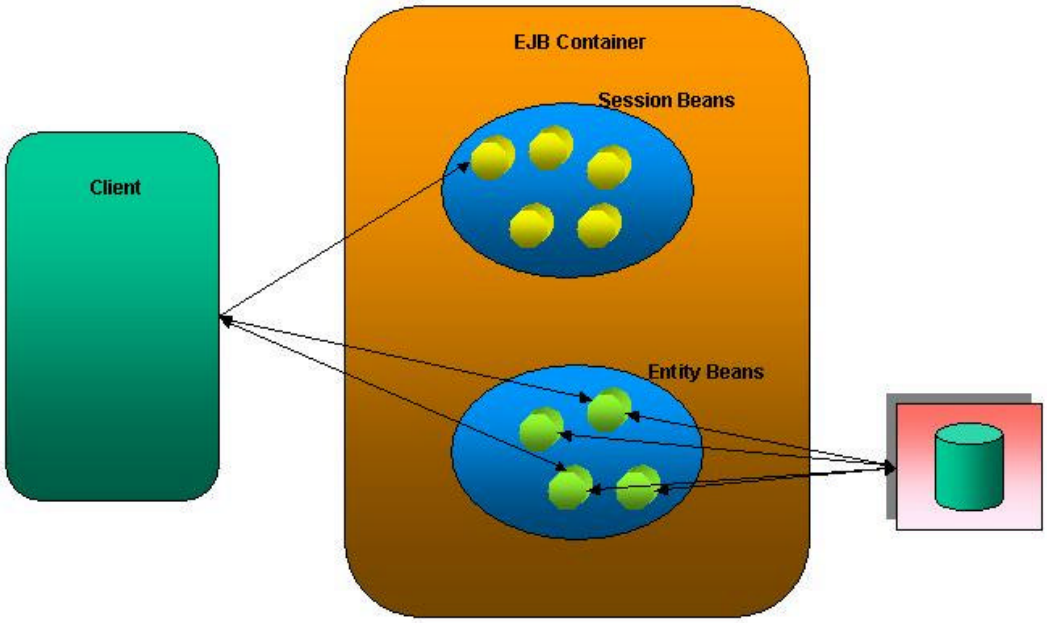


Figura 11-8

Oltre alle componenti, la architettura EJB definisce altre 3 entità fondamentali: servers, containers e client. Un Enterprise Bean vive all'interno del container che provvede al ciclo vitale della componente e ad una varietà di latri servizi quali gestione della concorrenza e scalabilità. L'EJB Container è a sua volta parte di un EJB server che fornisce tutti i servizi di naming e di directory (tali problematiche verranno affrontate negli articoli successivi).

Quando un client invoca una operazione su un EJB, la chiamata viene intergettata dal container. Questa intercessione del container tra client ed EJB a livello di "chiamata a metodo" consente al container la gestione di quei servizi di cui si è parlato all'inizio del paragrafo oppure della propagazione della chiamata ad altre componenti (Load Balancing) o ad altri container (Scalabilità) su altri server sparsi per la rete su differenti macchine.

Nella implementazione di una architettura enterprise, oltre a decidere che tipo di enterprise beans utilizzare, un programmatore deve effettuare altre scelte strategiche nella definizione del modello a componenti :

Che tipo di oggetto debba rappresentare un Enterprise Bean;

Che ruolo tale oggetto deve avere all'interno di un gruppo di componenti che collaborino tra di loro.

Dal momento che gli enterprise beans sono oggetti che necessitano di abbondanti risorse di sistema e di banda di rete, non sempre è soluzione ottima modellare tutti i business object come EJBs. In generale una soluzione consigliabile

è quella di adottare tale modello solo per quelle componenti che necessitino un accesso diretto da parte di un client.

EIS-Tier

Le applicazioni enterprise implicano per definizione l'accesso ad altre applicazioni, dati o servizi sparsi all'interno delle infrastrutture informatiche del fornitore di servizi. Le informazioni gestite ed i dati contenuti all'interno di tali infrastrutture rappresentano la "ricchezza" del fornitore, e come tale vanno trattati con estrema cura garantendone la integrità.

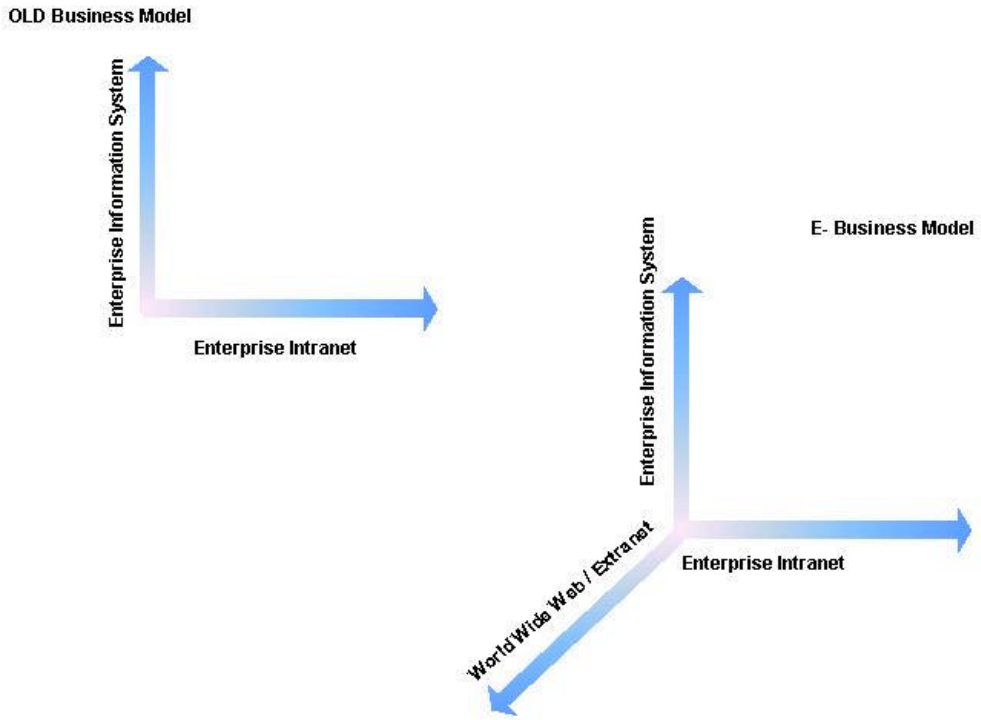


Figura 11-9

Al modello bidimensionale delle vecchie forme di business legato ai sistemi informativi (*figura 11-9*) è stato oggi sostituito da un nuovo modello (e-business) che introduce una terza dimensione che rappresenta la necessità da parte del fornitore di garantire accesso ai dati contenuti nella infrastruttura enterprise via web a: partner commerciali, consumatori, impiegati e altri sistemi informativi.

Gli scenari di riferimento sono quindi svariati e comprendono vari modelli di configurazione che le architetture enterprise vanno ad assumere per soddisfare le necessità della new-economy. Un modello classico è quello rappresentato da sistemi di commercio elettronico (*figura 11-10*) .

Nei negozi virtuali o E-Store l'utente web interagisce tramite browser con i cataloghi on-line del fornitore, seleziona i prodotti, inserisce i prodotti selezionati nel carrello virtuale, avvia una transazione di pagamento con protocolli sicuri.

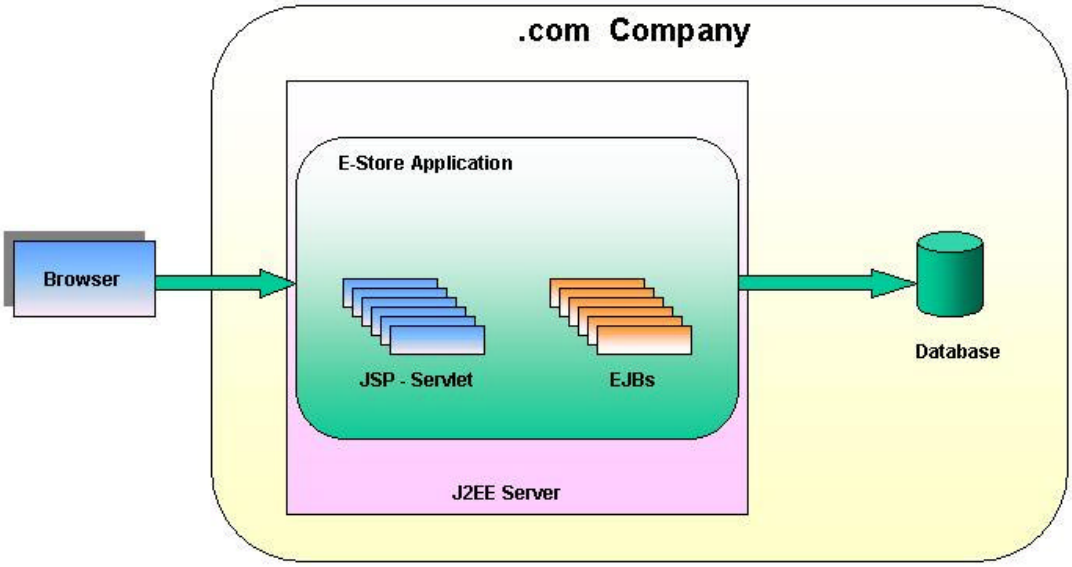


Figura 11-10

Le API di J2EE

Le API di J2EE forniscono supporto al programmatore per lavorare con le più comuni tecnologie utilizzate nell'ambito della programmazione distribuita e dei servizi di interconnessione via rete. I prossimi paragrafi forniranno una breve introduzione alle API componenti lo strato tecnologico fornito con la piattaforma definendo volta per volta la filosofia alla base di ognuna di esse e tralasciando le API relative a Servlet, EJBs e JavaServer Pages già introdotte nei paragrafi precedenti.

J2EE Technologies

Java IDL
RMI/IIOP
JDBC
JDBC 2.0 Client Parts
JNDI
Servlet
Javaserver Pages
Javamail, JavaBeans Activation Framework
JTS
EJB
JTA
JMS
Connector/Resource Mgmt. Comp.

JDBC : Java DataBase Connectivity

Rappresentano le API di J2EE per poter lavorare con database relazionali. Esse consentono al programmatore di inviare query ad un database relazionale, di effettuare dolete o update dei dati all'interno di tabelle, di lanciare stored-procedure o di ottenere meta-informazioni relativamente al database o le entità che lo compongono.

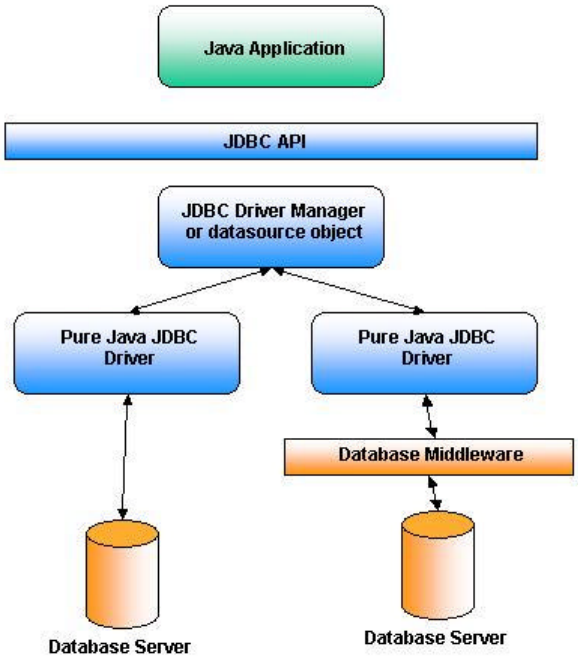


Figura 11-11

Architetturalmente JDBC sono suddivisi in due strati principali : il primo che fornisce una interfaccia verso il programmatore, il secondo di livello più basso che fornisce invece una serie di API per i produttori di drivers verso database relazionali e nasconde all'utente i dettagli del driver in uso.Questa caratteristica rende la tecnologia indipendente rispetto al motore relazionale che il programmatore deve interfacciare. I driver JDBC possono essere suddivisi in 4 tipi fondamentali come da tabella:

Tipo 4	Direct-to-Database Pure Java Driver
Tipo 3	Pure Java Driver for Database Middleware;
Tipo 2	JDBC-ODBC Bridge plus ODBC Driver;
Tipo 1	A native-API partly Java technology-enabled driver.

I driver di tipo 4 e 3 appartengono a quella gamma di drivers che convertono le chiamate JDBC nel protocollo di rete utilizzato direttamente da un server relazionale o un "middleware" che fornisca connettività attraverso la rete verso uno o più database (*figura 11-11*) e sono scritti completamente in Java.

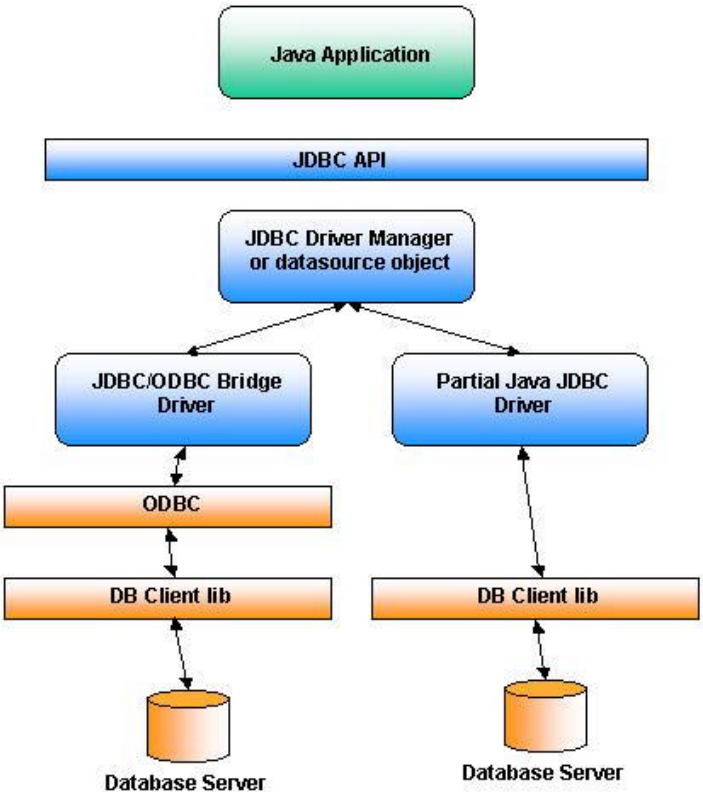


Figura 11-12

I driver di tipo 2 sono driver scritti parzialmente in Java e funzionano da interfaccia verso API native di specifici prodotti. I driver di tipo 1, anch'essi scritti parzialmente in java hanno funzioni di bridge verso il protocollo ODBC rilasciato da Microsoft. Questi driver sono anche detti JDBC/ODBC Bridge (*Figura 11-12*) e rappresentano una buona alternativa in situazioni in cui non siano disponibili driver di tipo 3 o 4.

RMI : Remote Method Invocation

Remote Method Invocation fornisce il supporto per sviluppare applicazioni java in grado di invocare metodi di oggetti distribuiti su virtual-machine differenti sparse per la rete. Grazie a questa tecnologia è possibile realizzare architetture distribuite in cui un client invoca metodi di oggetti residenti su un server che a sua volta può essere client nei confronti di un altro server.

Oltre a garantire tutte i vantaggi tipici di una architettura distribuita, essendo fortemente incentrato su Java RMI consente di trasportare in ambiente distribuito tutte le caratteristiche di portabilità e semplicità legata allo sviluppo di componenti Object Oriented apportando nuovi e significativi vantaggi rispetto alle ormai datate tecnologie distribuite (vd. CORBA).

Primo, RMI è in grado di passare oggetti e ritornare valori durante una chiamata a metodo oltre che a tipi di dati predefiniti. Questo significa che dati strutturati e complessi come le Hashtable possono essere passati come un singolo argomento senza dover decomporre l'oggetto in tipi di dati primitivi. In poche parole RMI

permette di trasportare oggetti attraverso le infrastrutture di una architettura enterprise senza necessitare di codice aggiuntivo.

Secondo, RMI consente di delegare l'implementazione di una classe dal client al server o viceversa. Questo fornisce una enorme flessibilità al programmatore che scriverà solo una volta il codice per implementare un oggetto che sarà immediatamente visibile sia a client che al server. Terzo, RMI estende le architetture distribuite consentendo l'uso di thread da parte di un server RMI per garantire la gestione ottimale della concorrenza tra oggetti distribuiti.

Infine RMI abbraccia completamente la filosofia "Write Once Run Anywhere" di Java. Ogni sistema RMI è portabile al 100% su ogni Java Virtual Machine.

Java IDL

RMI fornisce una soluzione ottima come supporto ad oggetti distribuiti con la limitazione che tali oggetti debbano essere scritti con Java. Tale soluzione non si adatta invece alle architetture in cui gli oggetti distribuiti siano scritti con linguaggi arbitrari. Per far fronte a tali situazioni, la Sun offre anche la soluzione basata su CORBA per la chiamata ad oggetti remoti.

CORBA (Common Object Request Broker Architecture) è uno standard largamente utilizzato introdotto dall'OMG (Object Managment Group) e prevede la definizione delle interfacce verso oggetti remoti mediante un IDL (Interface Definition Language) indipendente dalla piattaforma e dal linguaggio di riferimento con cui l'oggetto è stato implementato.

L'implementazione di questa tecnologia rilasciata da Sun comprende un ORB (Object Request Broker) in grado di interagire con altri ORB presenti sul mercato, nonché di un pre-compilatore IDL che traduce una descrizione IDL di una interfaccia remota in una classe Java che ne rappresenti il dato.

JNDI

Java Naming and Directory Interface è quell'insieme di API che forniscono l'accesso a servizi generici di Naming o Directory attraverso la rete. Consentono, alla applicazione che ne abbia bisogno, di ottenere oggetti o dati tramite il loro nome o di ricercare oggetti o dati mediante l'uso di attributi a loro associati.

Ad'esempio tramite JNDI è possibile accedere ad informazioni relative ad utenti di rete, server o workstation, sottoreti o servizi generici (*figura 11-13*).

Come per JDBC le API JNDI non nascono per fornire accesso in modo specifico ad un particolare servizio, ma costituiscono un set generico di strumenti in grado di interfacciarsi a servizi mediante "driver" rilasciati dal produttore del servizio e che mappano le API JNDI nel protocollo proprietario di ogni specifico servizio.

Tali driver vengono detti "Service Providers" e forniscono accesso a protocolli come LDAP, NIS, Novell NDS oltre che ad una gran quantità di servizi come DNS, RMI o CORBA Registry.

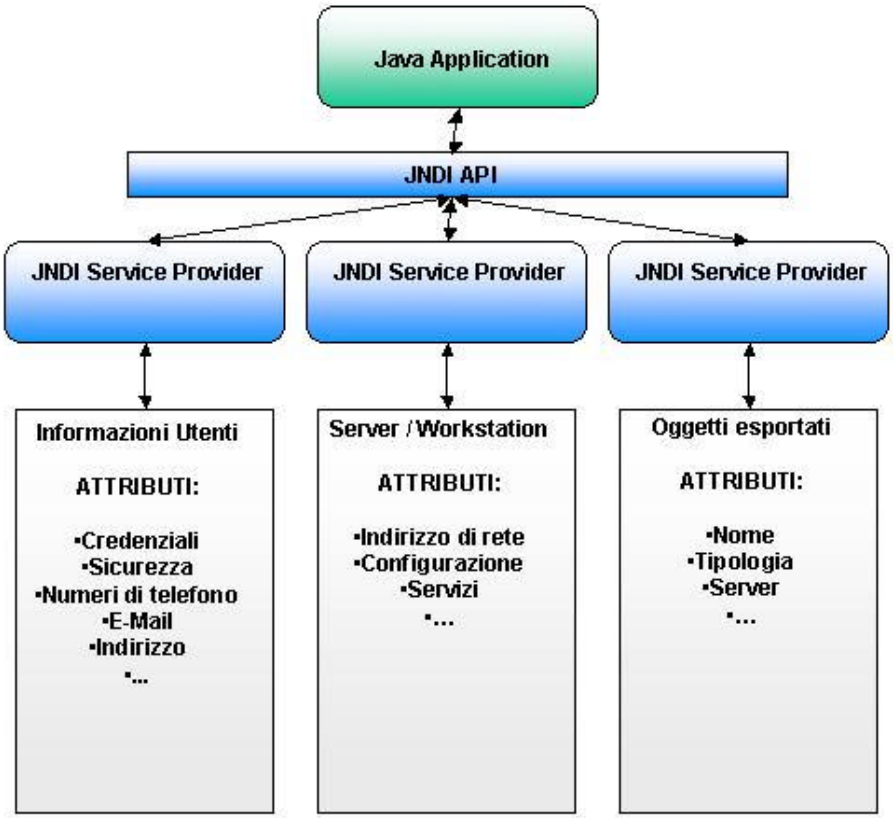


Figura 11-13

JMS

Java Message Service o Enterprise Messaging rappresentano le API forniscono supporto alle applicazioni enterprise nella gestione asincrona della comunicazione verso servizi di "messaging" o nella creazione di nuovi MOM (Message Oriented Middleware). Nonostante JMS non sia così largamente diffuso come le altre tecnologie, svolge un ruolo importantissimo nell'ambito di sistemi enterprise.

Per meglio comprenderne il motivo è necessario comprendere il significato della parola "messaggio" che in ambito JMS rappresenta tutto l'insieme di messaggi asincroni utilizzati dalle applicazioni enterprise, non dagli utenti umani, contenenti dati relativi a richieste, report o eventi che si verificano all'interno del sistema fornendo informazioni vitali per il coordinamento delle attività tra i processi. Essi contengono informazioni impacchettate secondo specifici formati relativi a particolari "eventi di business".

Grazie a JMS è possibile scrivere applicazioni di business "message-based" altamente portabili fornendo una alternativa a RMI che risulta necessaria in determinati ambiti applicativi. Nella *figura 11-14* è schematizzata una soluzione tipo di applicazioni basate su messaggi. Una applicazione di "Home Banking" deve interfacciarsi con il sistema di messagistica bancaria ed interbancaria da cui trarre tutte le informazioni relative alla gestione economica dell'utente.

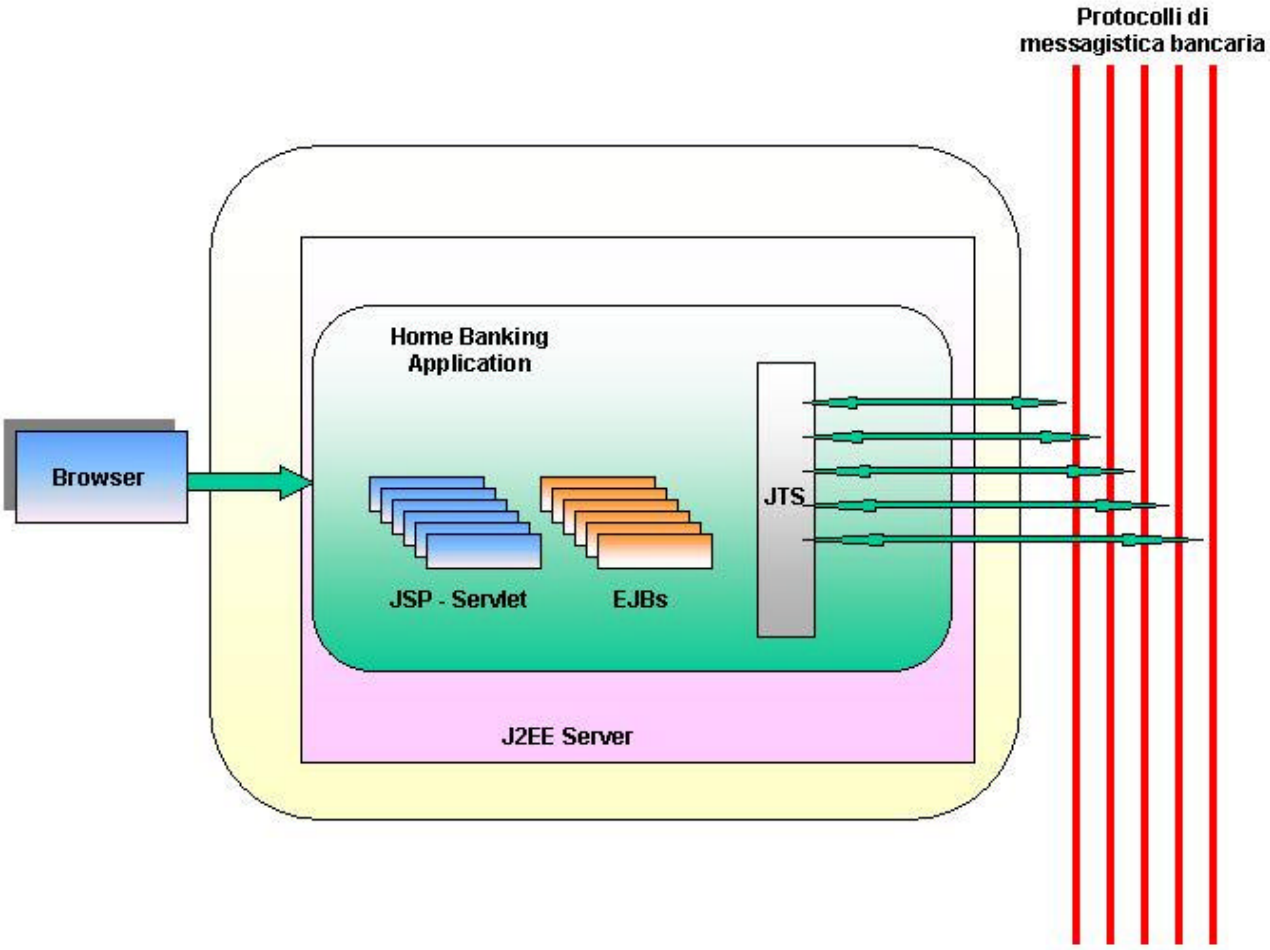


Figura 11-14

Nel modello proposto da J2EE l'accesso alle informazioni contenute all'interno di questo strato è possibile tramite oggetti chiamati "connettori" che rappresentano una architettura standard per integrare applicazioni e componenti enterprise eterogenee con sistemi informativi enterprise anche loro eterogenei.

I connettori sono messi a disposizione della applicazione enterprise grazie al server J2EE che li contiene sotto forma di servizi di varia natura (es. Pool di connessioni) e che tramite questi collabora con i sistemi informativi appartenenti allo strato EIS in modo da rendere ogni meccanismo a livello di sistema completamente trasparente alle applicazioni enterprise.

Lo stato attuale dell'arte prevede che la piattaforma J2EE consenta pieno supporto per la connettività verso database relazionali tramite le API JDBC, le prossime versioni della architettura J2EE prevedono invece pieno supporto transazionale verso i più disparati sistemi enterprise oltre che a database relazionali.

Capitolo 13
Architettura del Web Tier

Introduzione

Chi ha potuto accedere ad internet a partire dagli inizi degli anni 90, quando ancora la diffusione della tecnologia non aveva raggiunto la massa ed i costi erano eccessivi (ricordo che nel 1991 pagai un modem interno usato da 9600 bps la cifra incredibile di 200.000 lire), ha assistito alla incredibile evoluzione che ci ha portato oggi ad una diversa concezione del computer.

A partire dai primi siti internet dai contenuti statici, siamo oggi in grado di poter effettuare qualsiasi operazione tramite Web da remoto. Questa enorme crescita, come una vera rivoluzione, ha voluto le sua vittime. Nel corso degli anni sono nate e poi completamente scomparse un gran numero di tecnologie a supporto di un sistema in continua evoluzione.

Servlet e JavaServer Pages rappresentano oggi lo stato dell'arte delle tecnologie web. Raccogliendo la pesante eredità lasciata dai suoi predecessori, la soluzione proposta da SUN rappresenta quanto di più potente e flessibile possa essere utilizzato per sviluppare applicazioni web.

L'architettura del "Web Tier"

Prima di affrontare il tema dello sviluppo di applicazioni web con Servlet e JavaServer Pages è importante fissare alcuni concetti. Il web-server gioca un ruolo fondamentale all'interno di questo strato. In ascolto su un server, riceve richieste da parte del browser (client), le processa, quindi restituisce al client una entità o un eventuale codice di errore come prodotto della richiesta (*Figura 0-12*).

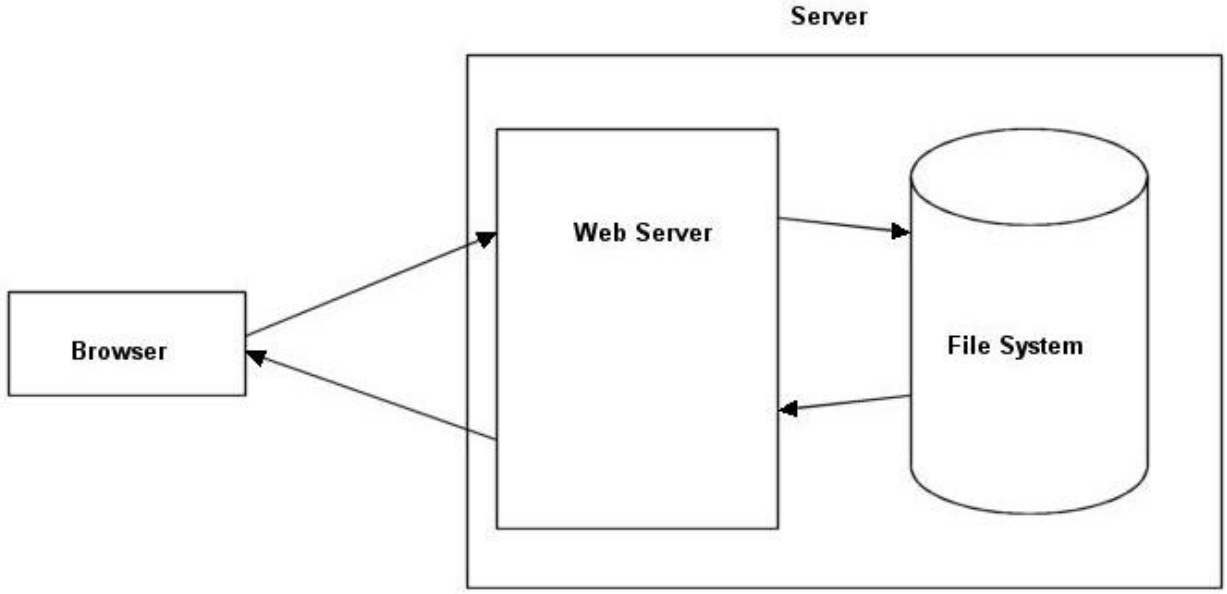
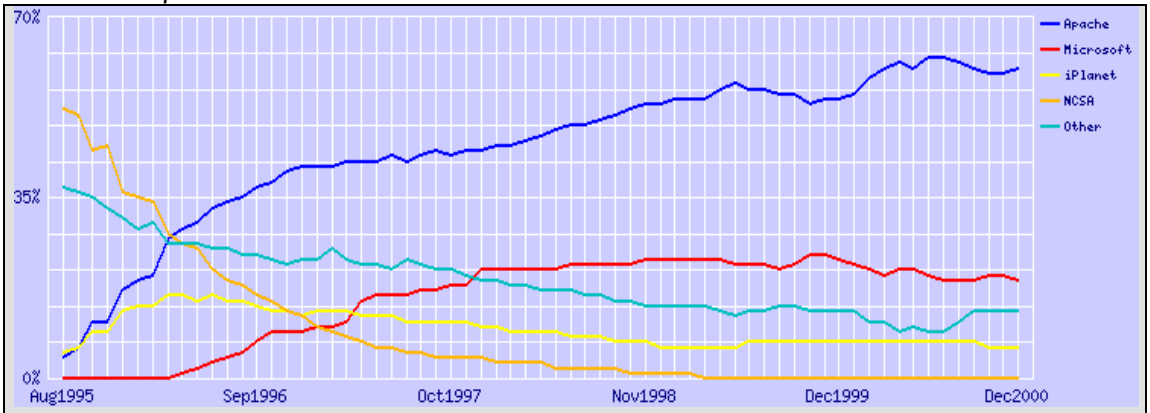


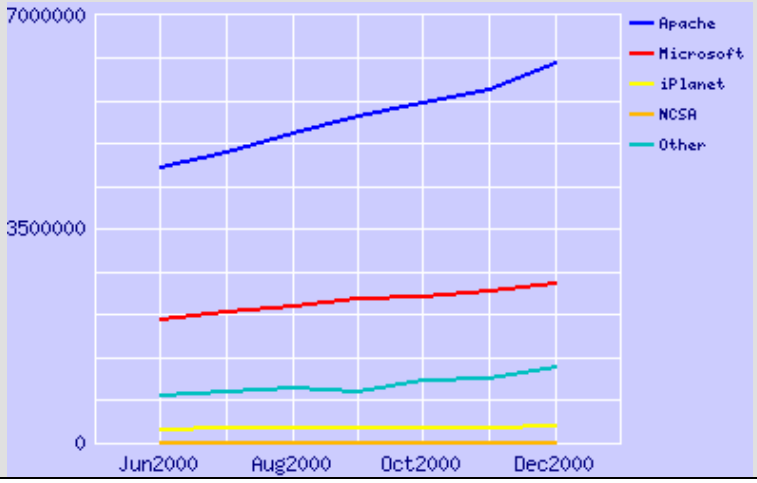
Figura 0-12

Nei **Box 1** e **2** sono riportati i dati relativi alla distribuzione dei web-server sul mercato negli ultimi anni (i dati sono disponibili nella forma completa all'indirizzo internet http://www.netcraft.com/survey).

Box 1 : *Grafici di distribuzione dei web server*



Market Share for Top Servers Across All Domains August 1995 - December 2000



Box 2 : *Distribuzione dei web server*

Top Developers

	November 2000		**December 2000**		
Apache	14193854	59.69	15414726	60.04	0.35
Microsoft	4776220	20.09	5027023	19.58	-0.51
iPlanet	1643977	6.91	1722228	6.71	-0.20

Top Servers

	November 2000		**December 2000**			
Apache	14193854	59.69	15414726	60.04	0.35	Apache
Microsoft-IIS						Microsoft-IIS
Netscape-Enterprise						Netscape-Enterprise
WebLogic	789953	3.32	890791	3.47	0.15	WebLogic
Zeus	640386	2.69	676526	2.63	-0.06	Zeus
Rapidsite	347307	1.46	365807	1.42	-0.04	Rapidsite
thttpd	226867	0.95	321944	1.25	0.30	thttpd
tigershark	120213	0.51	139300	0.54	0.03	tigershark
AOLserver	136326	0.57	125513	0.49	-0.08	AOLserver
WebSitePro	106618	0.45	110681	0.43	-0.02	WebSitePro

Le entità prodotte da un web-server possono essere entità statiche o entità dinamiche. Una entità statica è ad esempio un file html residente sul file-system del server. Rispetto alle entità statiche, unico compito del web-server è quello di recuperare la risorsa dal file-system ed inviarla al browser che si occuperà della visualizzazione dei contenuti.

Quello che più ci interessa sono invece le entità dinamiche, ossia entità prodotte dalla esecuzione di applicazioni eseguite dal web-server su richiesta del client.

Il modello proposto nella immagine 1 ora si complica in quanto viene introdotto un nuovo grado di complessità (*Figura 2-12*) nella architettura del sistema che, oltre a fornire accesso a risorse statiche dovrà fornire un modo per accedere a contenuti e dati memorizzati su una Base Dati, dati con i quali un utente internet potrà interagire da remoto.

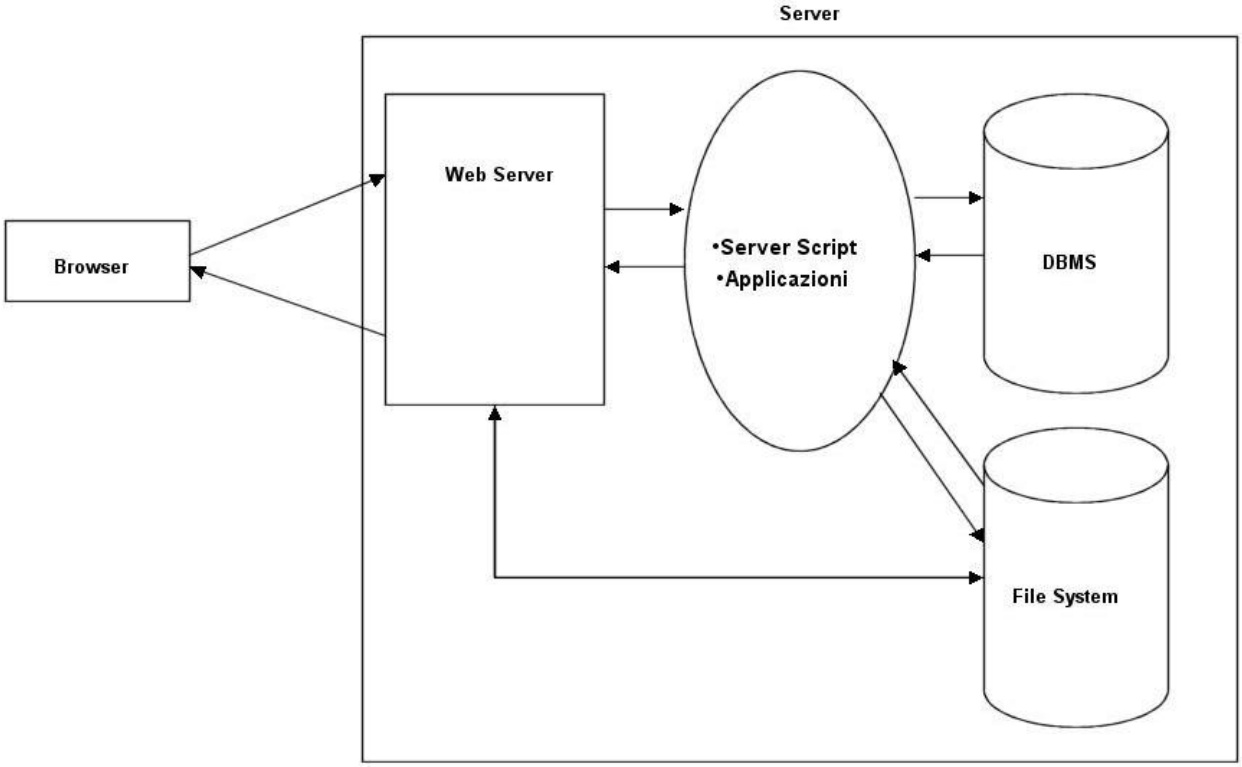


Figura 2-12

Inviare dati

Mediante form HTML è possibile inviare dati ad un web-server. Tipicamente l'utente riempie alcuni campi il cui contenuto, una volta premuto il pulsante "submit" viene inviato al server con il messaggio di richiesta. La modalità con cui questi dati vengono inviati dipende dal metodo specificato nella richiesta. Utilizzando il metodo GET, i dati vengono appesi alla request-URI nella forma di coppie *chiave=valore* separati tra di loro dal carattere "&". La stringa seguente è un esempio di campo request-URI per una richiesta di tipo GET.

http://www.java-net.tv/servelt/Hello?nome=Massimo&cognome=Rossi

"http://www.java-net.tv" rappresenta l'indirizzo del web-server a cui inviare la richiesta. I campi *"/servlet/Hello"* rappresenta la locazione della applicazione web da

eseguire. Il carattere “?” separa l’indirizzo dai dati e il carattere “&” separa ogni coppia chiave=valore.

Questo metodo utilizzato per default dai browser a meno di specifiche differenti, viene utilizzato nei casi in cui si richieda al server il recupero di informazioni mediante query su un database.

Il metodo POST pur producendo gli stessi effetti del precedente, utilizza una modalità di trasmissione dei dati differente impacchettando i contenuti dei campi di un form all’interno di un message-header. Rispetto al precedente, il metodo POST consente di inviare una quantità maggiore di dati ed è quindi utilizzato quando è necessario inviare al server nuovi dati affinché possano essere memorizzati all’interno della Base Dati.

Sviluppare applicazioni web

A meno di qualche eccezione una applicazione web è paragonabile ad una qualsiasi applicazione con la differenza che deve essere accessibile al web-server che fornirà l’ambiente di runtime per l’esecuzione. Affinché ciò sia possibile è necessario che esista un modo affinché il web-server possa trasmettergli i dati inviati dal client, che esista un modo univoco per l’accesso alle funzioni fornite dalla applicazione (entry-point), che l’applicazione sia in grado di restituire al web-server i dati prodotti in formato HTML da inviare al client nel messaggio di risposta.

Esistono molte tecnologie per scrivere applicazioni web che vanno dalla più comune “CGI” a soluzioni proprietarie come ISAPI di Microsoft o NSAPI della Netscape.

Common Gateway Interface

CGI è sicuramente la più comune e datata tra le tecnologie server-side per lo sviluppo di applicazioni web. Scopo principale di un CGI è quello di fornire funzioni di ”gateway” tra il web-server e la base dati del sistema.

I CGI possono essere scritti praticamente con tutti i linguaggi di programmazione. Questa caratteristica classifica i CGI in due categorie: CGI sviluppati mediante linguaggi script e CGI sviluppati con linguaggi compilati. I primi, per i quali viene generalmente utilizzato il linguaggio PERL, sono semplici da implementare, ma hanno lo svantaggio di essere molto lenti e per essere eseguiti richiedono l’intervento di un traduttore che trasformi le chiamate al linguaggio script in chiamate di sistema. I secondi a differenza dei primi sono molto veloci, ma risultano difficili da programmare (generalmente sono scritti mediante linguaggio C).

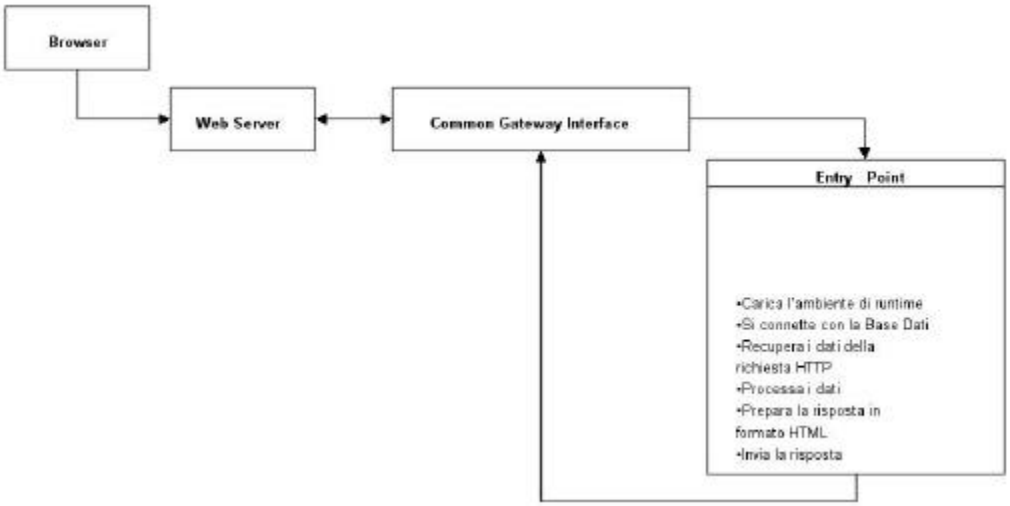


Figura 3-12

Nella *Figura 3-12* è schematizzato il ciclo di vita di un CGI. Il più grande svantaggio nell'uso dei CGI sta nella scarsa scalabilità della tecnologia infatti, ogni volta che il web-server riceve una richiesta deve creare una nuova istanza del CGI (*Figura 4-12*). Il dover stanziare un processo per ogni richiesta ha come conseguenza enormi carichi in fatto di risorse macchina e tempi di esecuzione (per ogni richiesta devono essere ripetuti i passi come nella *Figura 3-12*). A meno di non utilizzare strumenti di altro tipo è inoltre impossibile riuscire ad ottimizzare gli accessi al database. Ogni istanza del CGI sarà difatti costretta ad aprire e chiudere una propria connessione verso il DBMS.

Questi problemi sono stati affrontati ed in parte risolti dai Fast-CGI grazie ai quali è possibile condividere una stessa istanza tra più richieste http.

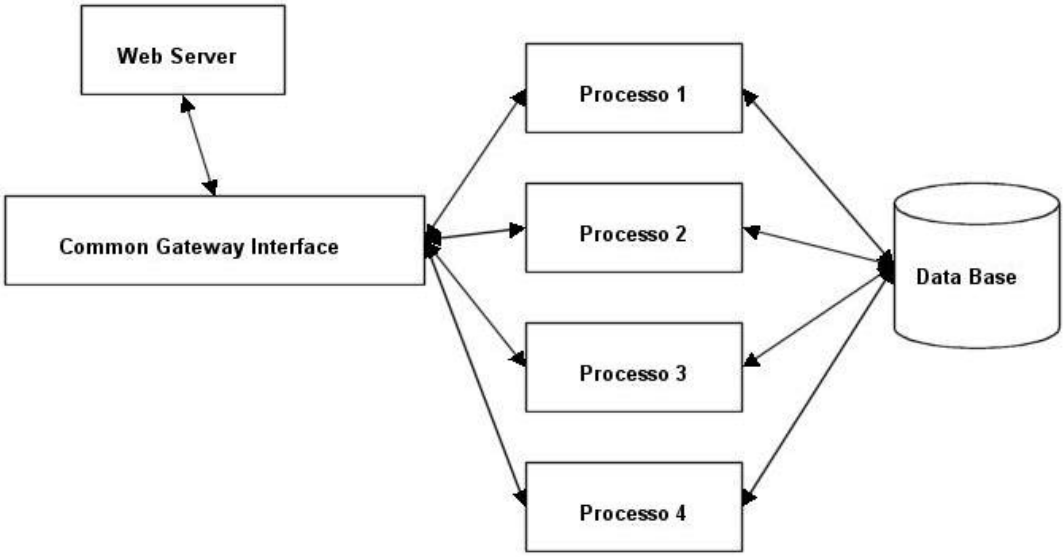


Figura 3-12: *Un CGI non è una tecnologia scalabile*

ISAPI ed NSAPI

Per far fronte ai limiti tecnologici imposti dai CGI, Microsoft e Netscape hanno sviluppato API proprietarie mediante le quali creare librerie ed applicazioni che possono essere caricate dal web-server al suo avvio ed utilizzate come proprie estensioni.

L'insuccesso di queste tecnologie è stato però segnato proprio dalla loro natura di tecnologie proprietarie e quindi non portabili tra le varie piattaforme sul mercato. Essendo utilizzate come moduli del web-server era inoltre caso frequente che un loro accesso errato alla memoria causasse il "crash" dell'intero web-server provocando enormi danni al sistema che doveva ogni volta essere riavviato.

ASP – Active Server Pages

Active Server Pages è stata l'ultima tecnologia web rilasciata da Microsoft. Una applicazione ASP è tipicamente un misto tra HTML e linguaggi script come VBScript o JavaScript mediante i quali si può accedere ad oggetti del server.

Una pagina ASP, a differenza di ISAPI, non viene eseguita come estensione del web-server, ma viene compilata alla prima chiamata, ed il codice compilato può quindi essere utilizzato ad ogni richiesta HTTP.

Nonostante sia oggi largamente diffusa, ASP come ISAPI rimane una tecnologia proprietaria e quindi in grado di funzionare solamente su piattaforma Microsoft, ma il

reale svantaggio di ASP è però legato alla sua natura di mix tra linguaggi script ed HTML. Questa sua caratteristica ha come effetto secondario quello di riunire in un unico "contenitore" sia le logiche applicative che le logiche di presentazione rendendo estremamente complicate le normali operazioni di manutenzione delle pagine.

Java Servlet e JavaServer Pages

Sun cerca di risolvere i problemi legati alle varie tecnologie illustrate, mettendo a disposizione del programmatore due tecnologie che raccogliendo l'eredità dei predecessori, ne abbattono i limiti mantenendo e migliorando gli aspetti positivi di ognuna. In aggiunta Servlet e JavaServer Pages ereditano tutte quelle caratteristiche che rendono le applicazioni Java potenti, flessibili e semplici da mantenere: portabilità del codice, paradigma object oriented.

Rispetto ai CGI, Servlet forniscono un ambiente ideale per quelle applicazioni per cui sia richiesta una massiccia scalabilità e di conseguenza l'ottimizzazione degli accessi alle basi dati di sistema.

Rispetto ad ISAPI ed NSAPI, pur rappresentando una estensione al web server, mediante il meccanismo delle eccezioni risolvono a priori tutti gli errori che potrebbero causare la terminazione prematura del sistema.

Rispetto ad ASP, Servlet e JavaServer Pages separano completamente le logiche di business dalle logiche di presentazione dotando il programmatore di un ambiente semplice da modificare o mantenere.

Capitolo 14
Java Servlet API

Introduzione

Java Servlet sono oggetti Java con proprietà particolari che vengono caricati ed eseguiti dal web server che le utilizzerà come proprie estensioni. Il web server di fatto mette a disposizione delle Servlet il container che si occuperà della gestione del loro ciclo di vita, delle gestione dell'ambiente all'interno delle quali le servlet girano, dei servizi di sicurezza. Il container ha anche funzione passare i dati dal client verso le servlet e viceversa ritornare al client i dati prodotti dalla loro esecuzione.

Dal momento che una servlet è un oggetto server-side, può accedere a tutte le risorse messe a disposizione dal server per generare pagine dai contenuti dinamici come prodotto della esecuzione delle logiche di business. E' ovvio che sarà cura del programmatore implementare tali oggetti affinché gestiscano le risorse del server in modo appropriato evitando di causare danni al sistema che le ospita.

Il package javax.servlet

Questo package è il package di base delle Servlet API, e contiene le classi per definire Servlet standard indipendenti dal protocollo. Tecnicamente una Servlet generica è una classe definita a partire dall'interfaccia Servlet contenuta all'interno del package *javax.servlet*. Questa interfaccia contiene i prototipi di tutti i metodi necessari alla esecuzione delle logiche di business, nonché alla gestione del ciclo di vita dell'oggetto dal momento del suo istanziamento, sino al momento della sua terminazione.

```
package javax.servlet;

import java.io.*;
public interface Servlet
{
    public abstract void destroy();
    public ServletConfig getServletConfig();
    public String getServletInfo();
    public void service (ServletRequest req, ServletResponse res) throws
            IOException, ServletException;
}
```

I metodi definiti in questa interfaccia devono essere supportati da tutte le servlet o possono essere ereditati attraverso la classe astratta GenericServlet che rappresenta una implementazione base di una servlet generica. Nella *Figura 14-1* viene schematizzata la gerarchia di classi di questo package.

Il package include inoltre una serie di classi utili alla comunicazione tra client e server, nonché alcune interfacce che definiscono i prototipi di oggetti utilizzati per tipizzare le classi che saranno necessarie alla specializzazione della servlet generica in servlet dipendenti da un particolare protocollo.

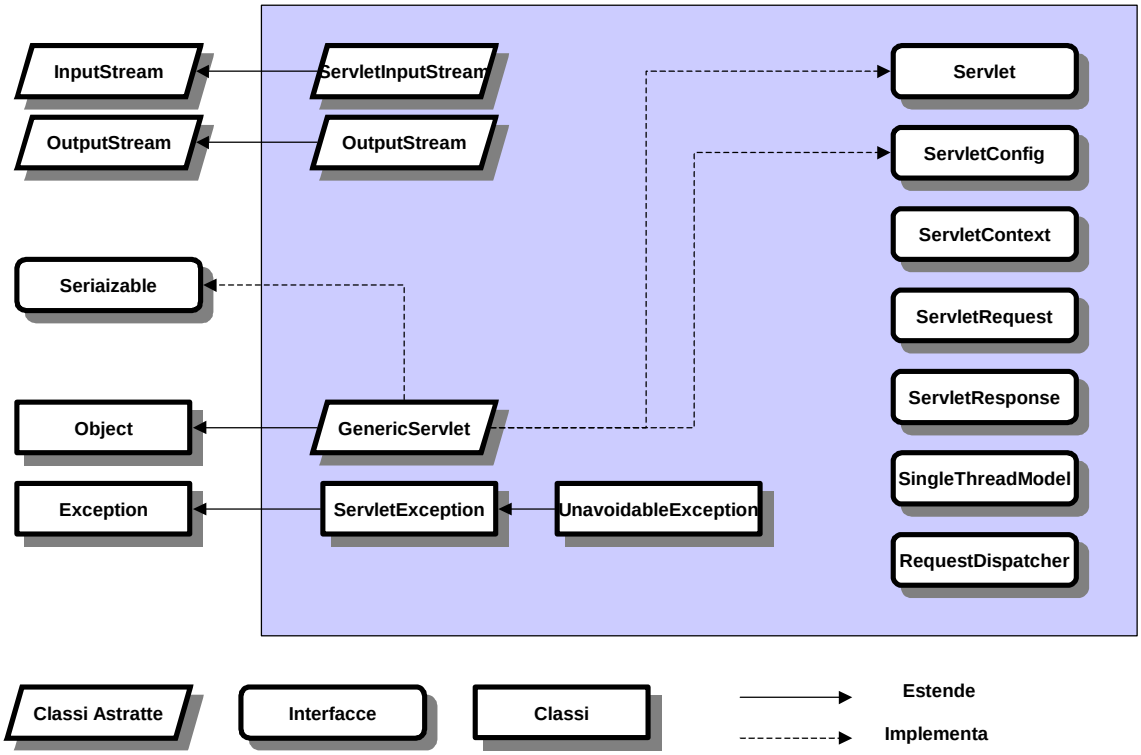


Figura 14-1: *il package javax.servlet*

Il package javax.servlet.http

Questo package supporta lo sviluppo di Servlet che specializzando la classe base astratta *GenericServlet* definita nel package javax.servlet utilizzano il protocollo http. Le classi di questo package estendono le funzionalità di base di una servlet supportando tutte le caratteristiche della trasmissione di dati con protocollo http compresi cookies, richieste e risposte http nonché metodi http (get, post head, put ecc. ecc.). Nella *Figura 14-2* è schematizzata la gerarchia del package in questione.

Formalmente quindi, una servlet specializzata per generare contenuti specifici per il mondo web sarà ottenibile estendendo la classe base astratta *javax.servlet.http.HttpServlet*.

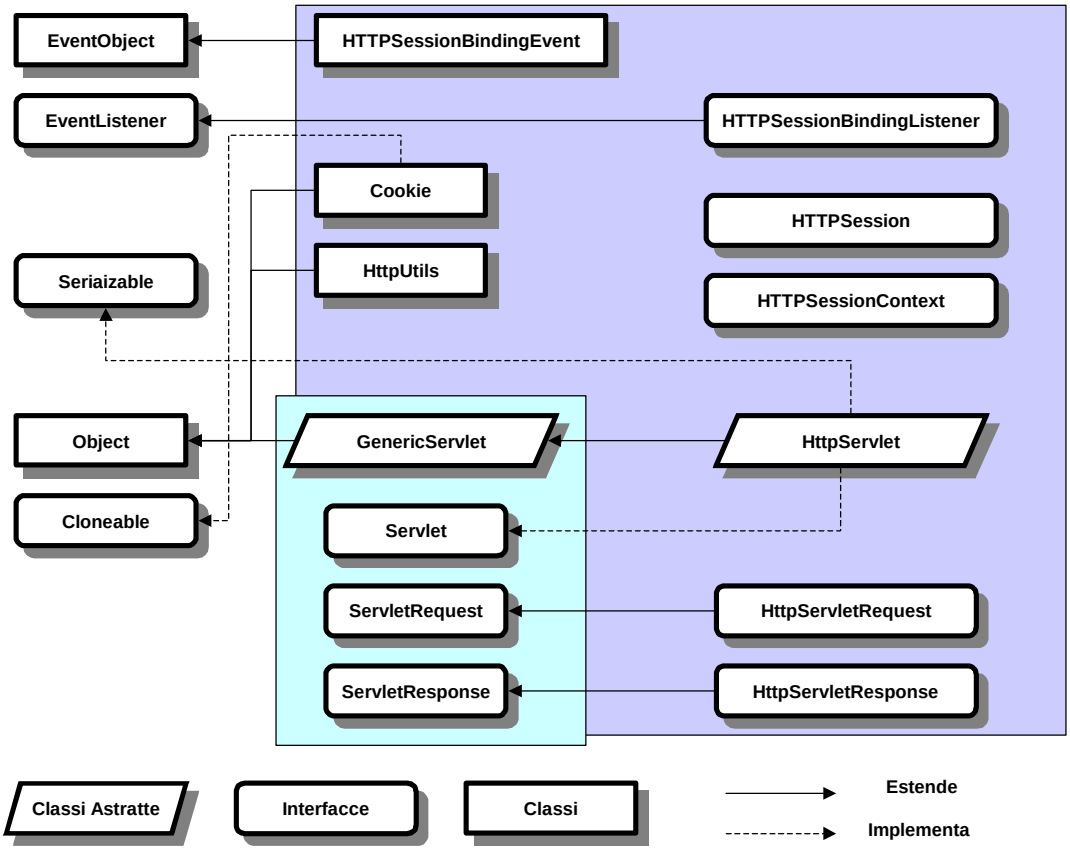


Figura 14-2 *: il package javax.servlet*

Ciclo di vita di una servlet

Il ciclo di vita di una servlet definisce come una servlet sarà caricata ed inizializzata, come riceverà e risponderà a richieste dal client ed infine come sarà terminata prima di passare sotto la responsabilità del garbage collector. Il ciclo di vita di una servlet è schematizzato nel diagramma seguente (*Figura 14- 3*).

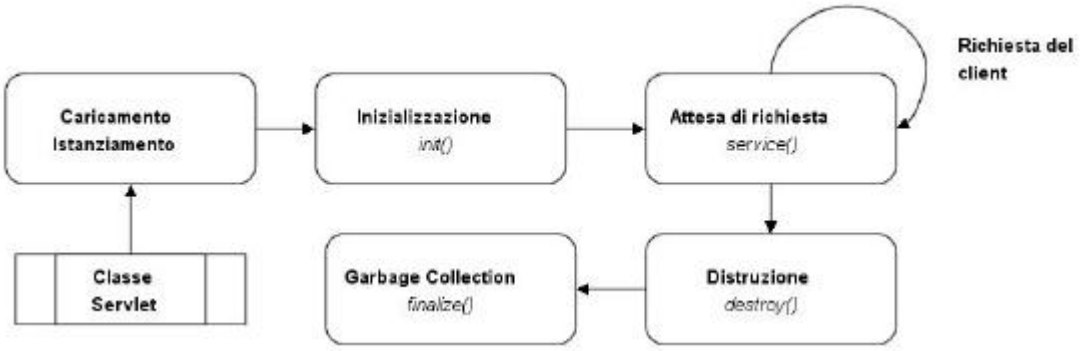


Figura 14-3: *Ciclo di vita di una servlet*

Il caricamento e l'istanziamento di una o più servlet è a carico del web server ed avviene al momento della prima richiesta http da parte di un client, o se specificato direttamente, al momento dell'avvio del servizio. Questa fase viene eseguita dal web server utilizzando l'oggetto *Class* di java.lang .

Dopo che è stata caricata, è necessario che la servlet venga inizializzata. Durante questa fase, la servlet generalmente carica dati persistenti, apre connessioni verso il database o stabilisce legami con altre entità esterne.

L'inizializzazione della servlet avviene mediante la chiamata al metodo *init()*, definito nella interfaccia *javax.servlet.Servlet* ed ereditato dalla classe base astratta javax.servlet.http.HttpServlet. Nel caso in cui il metodo non venga riscritto, il metodo ereditato non eseguirà nessuna operazione. Il metodo init di una servlet prende come parametro di input un oggetto di tipo ServletConfig che consente di accedere a parametri di inizializzazione passati attraverso il web server nella forma di coppie chiave-valore.

Dopo che la nostra servlet è stata inizializzata, è pronta ad eseguire richieste da parte di un client. Le richieste da parte di un client vengono inoltrate alla servlet dal container nella forma di oggetti di tipo HttpServletRequest e HttpServletResponse mediante passaggio di parametri al momento della chiamata del metodo *service().*

Come per *init()*, il metodo *service()* viene ereditato di default dalla classe base astratta HttpServlet. In questo caso però il metodo ereditato a meno che non venga ridefinito eseguirà alcune istruzioni di default come vedremo tra qualche paragrafo. E' comunque importante ricordare che all'interno di questo metodo vengono definite le logiche di business della nostra applicazione. Mediante l'oggetto HttpServletRequest saremo in grado di accedere ai parametri passati in input dal client, processarli e rispondere con un oggetto di tipo HttpServletResponse.

Infine, quando una servlet deve essere distrutta (tipicamente quando viene terminata l'esecuzione del web server), il container chiama di default il metodo *destroy()*. L'Overriding di questo metodo ci consentirà di rilasciare tutte le risorse utilizzate dalla servlet, ad esempio le connessioni a basi dati, garantendo che il sistema non rimanga in uno stato inconsistente a causa di una gestione malsana da parte della applicazione web.

Servlet e multithreading

Tipicamente quando n richieste da parte del client arrivano al web server, vengono creati n thread differenti in grado di accedere ad una particolare servlet in maniera concorrente (*Figura 14-4)*.

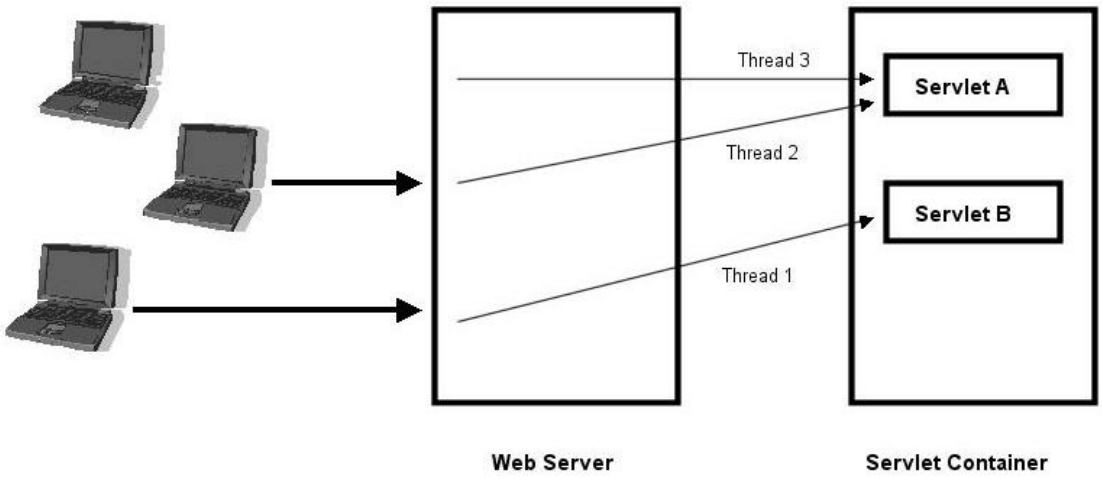


Figura 14-4 *: Accessi concorrenti a servlet*

Come tutti gli oggetti Java, servlet non sono oggetti thread-safe, ovvero è necessario che il programmatore definisca le politiche di accesso alla istanza della classe da parte del thread. Inoltre, lo standard definito da SUN Microsystem prevede

che un server possa utilizzare una sola istanza di questi oggetti (questa limitazione anche se tale aiuta alla ottimizzazione della gestione delle risorse percui ad esempio una connessione ad un database sarà condivisa tra tante richieste da parte di client).

Mediante l'utilizzo dell'operatore **synchronized**, è possibile sincronizzare l'accesso alla classe da parte dei thread dichiarando il metodo service() di tipo sincronizzato o limitando l'utilizzo del modificatore a singoli blocci di codice che contengono dati sensibili (vd. Capitolo 9).

L'Interfaccia SingleThreadModel

Quando un thread accede al metodo sincronizzato di una servlet, ottiene il lock sulla istanza dell'oggetto. Abbiamo inoltre detto che un web server per definizione utilizza una sola istanza di servlet condividendola tra le varie richieste da parte dei client. Stiamo creando un collo di bottiglia che in caso di sistemi con grossi carichi di richieste potrebbe ridurre significativamente le prestazioni del sistema.

Se il sistema dispone di abbondanti risorse, le Servlet API ci mettono a disposizione un metodo per risolvere il problema a scapito delle risorse della macchina rendendo le servlet classi "thread-safe".

Formalmente, una servlet viene considerata thread-safe se implementa l'interfaccia *javax.servlet.SingleThreadModel* . In questo modo saremo sicuri che solamente un thread avrà accesso ad una istanza della classe in un determinato istante. A differenza dell'utilizzo del modificatore **synchronized**, in questo caso il web server creerà più istanze di una stessa servlet (tipicamente in numero limitato e definito) al momento del caricamento dell'oggetto, e utilizzerà le varie istanze assegnando al thread che ne faccia richiesta, la prima istanza libera (*Figura 14-5*).

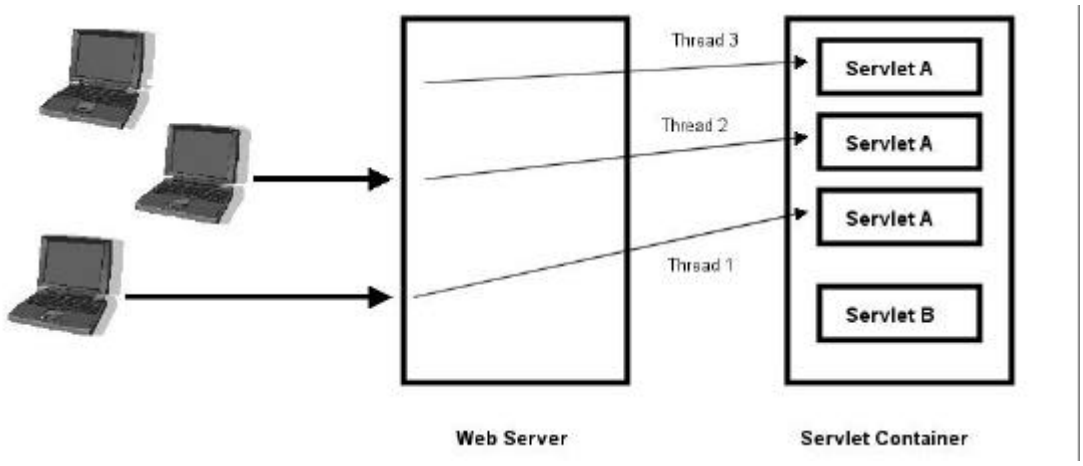


***Figura 14-5** : SingleThreadModel*

Un primo esempio di classe Servlet

Questa prima servlet di esempio fornisce la versione web della applicazione Java HelloWorld. La nostra servlet una volta chiamata ci restituirà una pagina HTML contenente semplicemente la stringa HelloWorld.

HelloWorldServlet.java

```
import javax.servlet.* ;
import javax.servlet.http.* ;
public class HelloWorldServlet extends HttpServlet
{
```

```
        public void service (HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException
        {
                res.setContentType("text/html");
                ServletOutputStream out = res.getOutputStream();
                out.println("<html>");
                out.println("<head><title>Hello World</title></head>");
                out.println("<body>");
                out.println("<h1>Hello World</h1>");
                out.println("</body></html>");
        }
    }
```

Il metodo service()

Se il metodo *service()* di una servlet non viene modificato, la nostra classe eredita di default il metodo service definito all'interno della classe astratta HttpServlet (*Figura 14-6*). Essendo il metodo chiamato in causa al momento dell'arrivo di una richiesta da parte di un client, nella sua forma originale questo metodo ha funzioni di "dispatcher" tra altri metodi basati sul tipo di richiesta http in arrivo dal client.

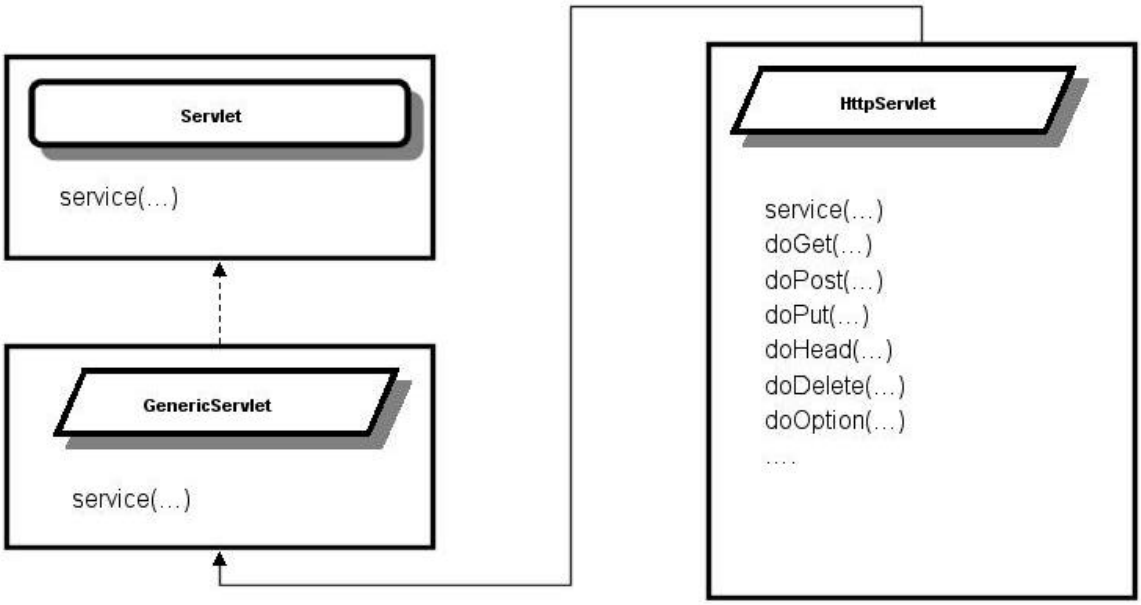


Figura 14-6: *La classe HttpServlet*

Come abbiamo già introdotto nel capitolo precedente, il protocollo http ha una varietà di tipi differenti di richieste che possono essere avanzate da parte di un client. Comunemente quelle di uso più frequente sono le richieste di tipo GET e POST.

Nel caso di richiesta di tipo GET o POST, il metodo service definito all'interno della classe astratta *HttpServlet* chiamerà i metodi rispettivamente *doGet()* o *doPost()* che conterranno ognuno il codice per gestire la particolare richiesta. In questo caso quindi, non avendo applicato la tecnica di "Overriding" sul metodo *service()* , sarà necessario implementare uno di questi metodi all'interno della nostra nuova classe a seconda del tipo di richieste che dovrà esaudire (*Figura 13-7*).

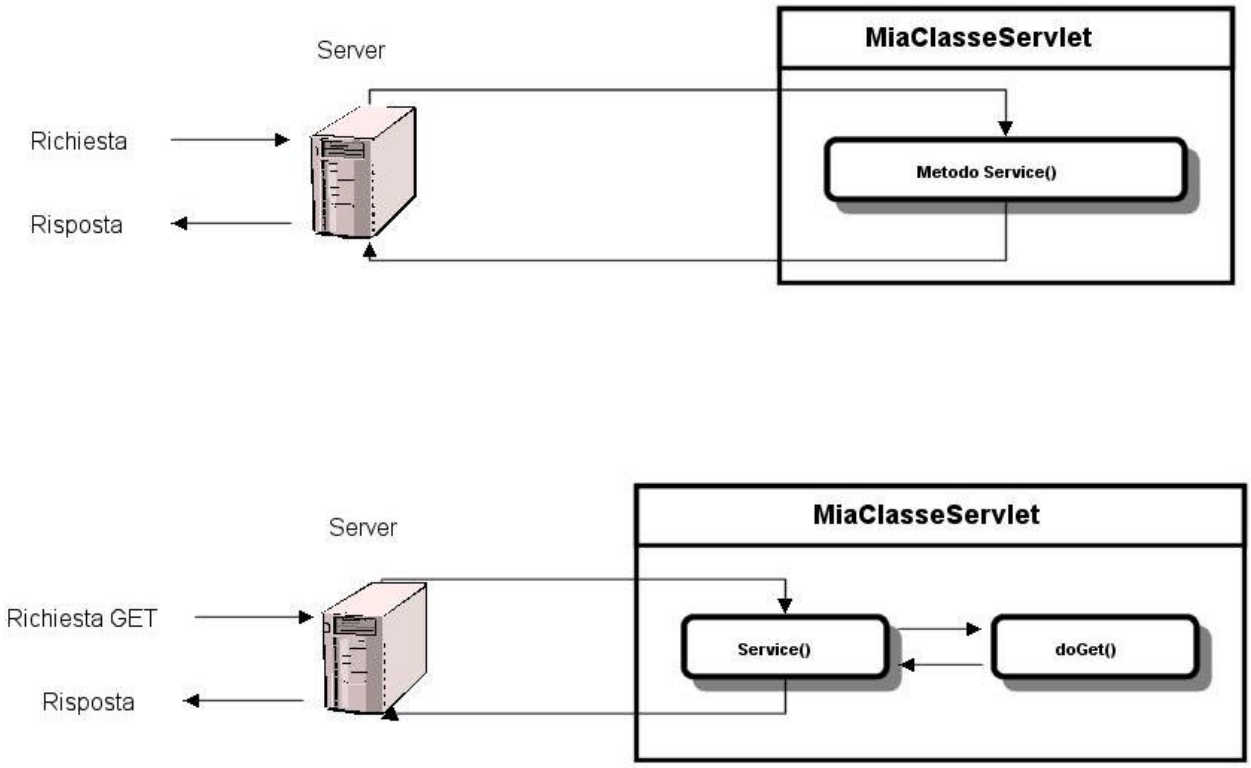


***Figura 14-7** : service() ha funzioni di bridge*

Capitolo 15
Servlet HTTP

Introduzione

Http servlet rappresentano una specializzazione di servlet generiche e sono specializzate per comunicare mediante protocollo http. Il package javax.servlet.http mette a disposizione una serie di definizioni di classe che rappresentano strumenti utili alla comunicazione tra client e server con questo protocollo, nonché forniscono uno strumento flessibile per accedere alle strutture definite nel protocollo, al tipo di richieste inviate ai dati trasportati.

Le interfacce ServletRequest e ServletResponse rappresentano rispettivamente richieste e risposte http. In questo capitolo le analizzeremo in dettaglio con lo scopo di comprendere i meccanismi di ricezione e manipolazione dei dati di input nonché di trasmissione delle entità dinamiche prodotte.

Il protocollo HTTP 1.1

Rispetto alla classificazione fornita nei paragrafi precedenti riguardo i protocolli di rete, http (HyperText Trasfer Protocol) appartiene all'insieme dei protocolli applicativi e, nella sua ultima versione, fornisce un meccanismo di comunicazione tra applicazioni estremamente flessibile dotato di capacità praticamente infinita nel descrivere possibili richieste nonché i possibili scopi per cui la richiesta è stata inviata. Grazie a queste caratteristiche http è largamente utilizzato per la comunicazione tra applicazioni in cui sia necessaria la possibilità di negoziare dati e risorse, ecco perché l'uso del protocollo come standard per i web server.

Il protocollo fu adottato nel 1990 dalla "World Wide Web Global Information Initiative" a partire dalla versione 0.9 come schematizzato nella tabella sottostante:

HTTP 0.9	Semplice protocollo per la trasmissione dei dati via internet.
HTTP 1.0	Introduzione rispetto alla versione precedente del concetto di Mime-Type. Il protocollo è ora in grado di trasportare meta-informazioni relative ai dati trasferiti.
HTTP 1.1	Estende la versione precedente affinché il protocollo contenga informazioni che tengono in considerazione problematiche a livello applicativo come ad esempio quelle legate alla gestione della cache.

Scendendo nei dettagli, http è un protocollo di tipo Request/Response: ovvero un client invia una richiesta al server e rimane in attesa di una sua risposta. Entrambe richiesta e risposta http hanno una struttura che identifica due sezioni principali: un message-header ed un message-body come schematizzato nella *figura 15-1*. Il message-header contiene tutte le informazioni relative alla richiesta o risposta, mentre il message-body contiene eventuali entità trasportate dal pacchetto (intendendo per entità i contenuti dinamici del protocollo come pagine html) e tutte le informazioni relative: ad esempio mime-type, dimensioni ecc. Le entità trasportate all'interno del message-body vengono inserite all'interno dell'entity-body.

Le informazioni contenute all'interno di un messaggio http sono separate tra di loro dalla sequenza di caratteri "CRLF" dove CR è il carattere "Carriage Return" (US ASCII 13) e LF è il carattere "Line Feed" (US ASCII 10). Tale regola non è applicabile ai contenuti dell'entity-body che rispettano il formato come definito dal rispettivo MIME.

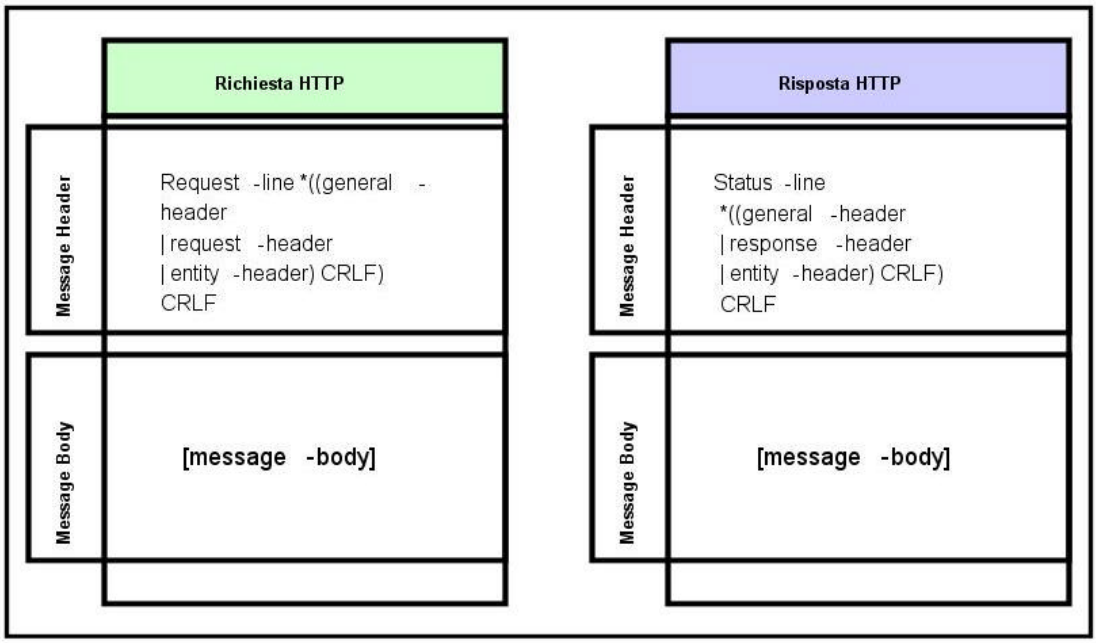


Figura 15-1 : *pacchetti http*

Per dare una definizione formale del protocollo http utilizzeremo la sintassi della forma estesa di Backup Nauer che per semplicità indicheremo con l'acronimo BNF. Secondala BNF, un messaggio http generico è definito dalla seguente regola:

HTTP-Message *= Request | Response*
Request *= generic-message*
Response *= generic-message*
generic-message *= start-line *(message-header CRLF) CRLF [message-body]*
start-line *= Request-line | Response-line*

Richiesta HTTP

Scendendo nei dettagli, una richiesta http da un client ad un server è definita dalla seguente regola BNF:

Request *= Request-line *((general-header*
| request-header
| entity-header) CRLF)
CRLF
[message-body]

La *Request-line* è formata da tre token separati dal carattere SP o Spazio (US ASCII 32) che rappresentano rispettivamente: metodo, indirizzo della risorsa o URI, versione del protocollo.

Request-line *= Method SP Request-URI SP http-version CRLF*
SP *= <US-ASCII Space (32) >*
Method *= "OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" | "DELETE" | "TRACE" | "CONNECT" | extension-method*
extension-method *= token*
Request-URI *= "*" | absoluteURI | abs-path | authority*
HTTP-version *= "HTTP" "/" 1*DIGIT "." 1*DIGIT*

DIGIT = *<any US ASCII digit "0"...."9">*

Un esempio di *Request-line* inviata da client ha server potrebbe essere :

GET /lavora.html HTTP/1.1

Il campo General-header ha valore generale per entrambi I messagi di richiesta o risposta e non contiene dati applicabili alla entità trasportata con il messaggio:

General-header = *Cache-Control*
| *Connection*
| *Date*
| *Pragma*
| *Trailer*
| *Trasfer-Encoding*
| *Upgrade*
| *Via*
| *Warning*

Senza scendere nei dettagli di ogni singolo campo contenuto nel "General-header", è importante comprenderne l'utilizzo. Questo campo è infatti estremamente importante in quanto trasporta le informazioni necessarie alla negoziazione tra le applicazioni. Ad esempio, il campo *Connection* può essere utilizzato da un server proxy per determinare lo stato della connessione tra client e server e decidere su eventuali misura da applicare.

I campi di *request-header* consentono al client di inviare informazioni relative alla richiesta in corso ed al client stesso.

Request-header = *Accept*
| *Accept-Charset*
| *Accept-Encoding*
| *Accept-Language*
| *Authorization*
| *Expect*
| *From*
| *Host*
| *If-Match*
| *If-Modified-Since*
| *If-None-Match*
| *If-Range*
| *If-Unmodified-Since*
| *Max-Forwards*
| *Proxy-Authorization*
| *Range*
| *Referer*
| *TE*
| *User-Agent*

Entity-header e *Message-Body* verranno trattati nei paragrafi seguenti.

Risposta HTTP

Dopo aver ricevuto un messaggio di richiesta, il server risponde con un messaggio di risposta definito dalla regola BNF:

Response *= Status-line*
**((general-header*
| response-header
| entity-header) CRLF)
CRLF
[message-body]

La *Status-line* è la prima linea di un messaggio di risposta http e consiste di tre token separati da spazio contenenti rispettivamente informazioni sulla versione del protocollo, un codice di stato che indica un errore o l'eventuale buon fine della richiesta, una breve descrizione del codice di stato.

Status-line *= HTTP-Version SP Status-Code SP Reason-Phrase CRLF*
SP *= <US-ASCII Space (32) >*
Status-Code *=*
"100" : Continue
| "101 : Switching Protocols
| "200" : OK
| "201" : Created
| "202" : Accepted
| "203" : Non-Authoritative Information
| "204" : No Content
| "205" : Reset Content
| "206" : Partial Content
| "300" : Multiple Choices
| "301" : Moved Permanently
| "302" : Found
| "303" : See Other
| "304" : Not Modified
| "305" : Use Proxy
| "307" : Temporary Redirect
| "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 Time-out
| "409" : Conflict
| "410" : Gone
| "411" : Length Required
| "412" : Precondition Failed
| "413" : Request Entity Too Large
| "414" : Request-URI Too Large
| "415" : Unsupported Media Type
| "416" : Requested range not satisfiable
| "417" : Expectation Failed
| "500" : Internal Server Error

| "501" : Not Implemented
| "502" : Bad Gateway
| "503" : Service Unavailable
| "504" : Gateway Time-out
| "505" : HTTP Version not supported
| extension-code
extension-code *= 3DIGIT*
Reason-Phrase *= *<TEXT, excluding CR, LF>*

Un codice di stato è un intero a 3 cifre ed è il risultato della processazione da parte del server della richiesta mentre la *Reason-Phrase* intende fornire una breve descrizione del codice di stato. La prima cifra di un codice di stato definisce la regola per suddividere i vari codici:

1xx : Informativo – Richiesta ricevuta, continua la processazione dei dati;
2xx : Successo – L'azione è stata ricevuta con successo, compresa ed accettata;
3xx : Ridirezione – Ulteriori azioni devono essere compiute al fine di completare la richiesta;
4xx : Client-Error – La richiesta è sintatticamente errata e non può essere soddisfatta;
5xx : Server-Error – Il server non ha soddisfatto la richiesta apparentemente valida.

Come per il *Request-Header*, il *Response-Header* contiene campi utili alla trasmissione di informazioni aggiuntive relative alla risposta e quindi anch'esse utili alla eventuale negoziazione :

response-header *= Accept-Ranges*
| Age
| Etag
| Location
| Proxy-Authenticate
| Retry-After
| Server
| www-autheticate

Entità

Come abbiamo visto, richiesta e risposta http possono trasportare entità a meno che non sia previsto diversamente dallo status-code della risposta o dal request-method della richiesta (cosa che vedremo nel paragrafo successivo). In alcuni casi un messaggio di risposta può contenere solamente il campo entity-header (ossia il campo descrittore della risorsa trasportata). Come regola generale, entity-header trasporta meta-informazioni relative alla risorsa identificata dalla richiesta.

entity-header *= Allow*
| Content-Encoding
| Content-Language
| Content-Length
| Content-Location
| Content-MD5
| Content-Range
| Content-Type
| Expires

| *Last-Modified*
| *extension-header*

extension-header = *message-header*

Queste informazioni, a seconda dei casi possono essere necessarie od opzionali. Il campo extension-header consente l'aggiunta di una nuove informazioni non previste, ma necessarie, senza apportare modifiche al protocollo.

I metodi di request

I metodi definiti dal campo request-method sono necessari al server per poter prendere decisioni sulle modalità di processazione della richiesta. Le specifiche del protocollo http prevedono nove metodi, tuttavia, tratteremo brevemente i tre più comuni: HEAD, GET, POST.

Il metodo GET significa voler ottenere dal server qualsiasi informazione (nella forma di entità) come definita nel campo request-URI. Se utilizzato per trasferire dati mediante form HTML, una richiesta di tipo GET invierà i dati contenuti nel form scrivendoli in chiaro nella request-URI.

Simile al metodo GET è il metodo HEAD che ha come effetto quello di indicare al server che non è necessario includere nella risposta un message-body contenente la entità richiesta in request-URI. Tale metodo ha lo scopo di ridurre i carichi di rete in quei casi in cui non sia necessario l'invio di una risorsa: ad esempio nel caso in cui il client disponga di un meccanismo di caching.

Il metodo POST è quello generalmente utilizzato lì dove siano necessarie operazioni in cui è richiesto il trasferimento di dati al server affinché siano processati. Una richiesta di tipo POST non utilizza il campo Request-URI per trasferire i parametri, bensì invia un pacchetto nella forma di message-header con relativo entity-header. Il risultato di una operazione POST non produce necessariamente entità. In questo caso uno status-code 204 inviato dal server indicherà al client che la processazione è andata a buon fine senza che però abbia prodotto entità.

Inizializzazione di una Servlet

L'interfaccia *ServletConfig* rappresenta la configurazione iniziale di una servlet. Oggetti definiti per implementazione di questa interfaccia, contengono i parametri di inizializzazione della servlet (se esistenti) nonché permettono alla servlet di comunicare con il servlet container restituendo un oggetto di tipo *ServletContext*.

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

I parametri di inizializzazione di una servlet devono essere definiti all'interno dei file di configurazione del web server che si occuperà di comunicarli alla servlet in forma di coppie chiave=valore. I metodi messi a disposizione da oggetti di tipo ServletConfig per accedere a questi parametri sono due: il metodo *getInitParametrNames()* che restituisce un elenco (enumerazione) dei nomi dei parametri, ed il metodo *getInitParameter(String)* che preso in input il nome del parametro in forma di oggetto String, restituisce a sua volta una stringa contenente il valore del parametro di inizializzazione. Nell'esempio di seguito, utilizziamo il metodo

getInitParameter(String) per ottenere il valore del parametro di input con chiave "CHIAVEPARAMETRO":

```
String key = "CHIAVE_PARAMETRO";
String val = config.getInitParameter(key);
```

Dove *config* rappresenta un oggetto di tipo ServletConfig passato come parametro di input al metodo *init(ServletConfig)* della servlet.

Il metodo *getServletContext()*, restituisce invece un oggetto di tipo *ServletContext* tramite il quale è possibile richiedere al container lo stato dell'ambiente all'interno del quale le servlet stanno girando.

Un metodo alternativo per accedere ai parametri di configurazione della servlet è messo a disposizione dal metodo *getServletConfig()* definito all'interno della interfaccia servlet di javax.servlet. Utilizzando questo metodo sarà di fatto possibile accedere ai parametri di configurazione anche all'interno del metodo *service()*.

Nel prossimo esempio viene definita una servlet che legge un parametro di input con chiave "INPUTPARAMS", lo memorizza in un dato membro della classe e ne utilizza il valore per stampare una stringa alla esecuzione del metodo service().

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

public class Test extends HttpServlet
{
    String initparameter=null;

    public void init(ServletConfig config)
    {
            initparameter = config.getInitParameter("INPUTPARAMS");
    }
    public void service (HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
    {
            res.setContentType("text/html");
            ServletOutputStream out = res.getOutputStream();
            out.println("<html>");
            out.println("<head><title>Test</title></head>");
            out.println("<body>");
            out.println("<h1>Il parametro di input vale "+ initparameter +"</h1>");
            out.println("</body></html>");
    }

}
```

L'oggetto HttpServletResponse

Questo oggetto definisce il canale di comunicazione tra la servlet ed il client che ha inviato la richiesta (browser). Questo oggetto mette a disposizione della servlet i metodi necessari per inviare al client le entità prodotte dalla manipolazione dei dati di input. Una istanza dell'oggetto ServletResponse viene definita per implementazione della interfaccia HttpServletResponse definita all'interno del package javax.servlet.http che definisce una specializzazione della interfaccia derivata rispetto al protocollo http.

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

```
public interface ServletResponse extends RequestDispatcher
{
public String getCharacterEncoding();
public ServletOutputStream getOutputStream() throws IOException;
public PrintWriter getWriter() throws IOException;
public void setContentLength(int length);
public void setContentType(String type);
}


package javax.servlet.http;

import java.util.*;
import java.io.*;
public interface HttpServletResponse extends javax.servlet.ServletResponse
{
    public static final int SC_CONTINUE = 100;
    public static final int SC_SWITCHING_PROTOCOLS = 101;
    public static final int SC_OK = 200;
    public static final int SC_CREATED = 201;
    public static final int SC_ACCEPTED = 202;
    public static final int SC_NON_AUTHORITATIVE_INFORMATION = 203;
    public static final int SC_NO_CONTENT = 204;
    public static final int SC_RESET_CONTENT = 205;
    public static final int SC_PARTIAL_CONTENT = 206;
    public static final int SC_MULTIPLE_CHOICES = 300;
    public static final int SC_MOVED_PERMANENTLY = 301;
    public static final int SC_MOVED_TEMPORARILY = 302;
    public static final int SC_SEE_OTHER = 303;
    public static final int SC_NOT_MODIFIED = 304;
    public static final int SC_USE_PROXY = 305;
    public static final int SC_BAD_REQUEST = 400;
    public static final int SC_UNAUTHORIZED = 401;
    public static final int SC_PAYMENT_REQUIRED = 402;
    public static final int SC_FORBIDDEN = 403;
    public static final int SC_NOT_FOUND = 404;
    public static final int SC_METHOD_NOT_ALLOWED = 405;
    public static final int SC_NOT_ACCEPTABLE = 406;
    public static final int SC_PROXY_AUTHENTICATION_REQUIRED = 407;
    public static final int SC_REQUEST_TIMEOUT = 408;
    public static final int SC_CONFLICT = 409;
    public static final int SC_GONE = 410;
    public static final int SC_LENGTH_REQUIRED = 411;
    public static final int SC_PRECONDITION_FAILED = 412;
    public static final int SC_REQUEST_ENTITY_TOO_LARGE = 413;
    public static final int SC_REQUEST_URI_TOO_LONG = 414;
    public static final int SC_UNSUPPORTED_MEDIA_TYPE = 415;
    public static final int SC_INTERNAL_SERVER_ERROR = 500;
    public static final int SC_NOT_IMPLEMENTED = 501;
    public static final int SC_BAD_GATEWAY = 502;
    public static final int SC_SERVICE_UNAVAILABLE = 503;
    public static final int SC_GATEWAY_TIMEOUT = 504;
    public static final int SC_HTTP_VERSION_NOT_SUPPORTED = 505;
    void addCookie(Cookie cookie);
    boolean containsHeader(String name);
    String encodeRedirectUrl(String url); //deprecated
    String encodeRedirectURL(String url);
    String encodeUrl(String url);
    String encodeURL(String url);
    void sendError(int statusCode) throws java.io.IOException;
```

```
    void sendError(int statusCode, String message) throws java.io.IOException;
    void sendRedirect(String location) throws java.io.IOException;
    void setDateHeader(String name, long date);
    void setHeader(String name, String value);
    void setIntHeader(String name, int value);
    void setStatus(int statusCode);
    void setStatus(int statusCode, String message);
}
```

Questa interfaccia definisce i metodi per impostare dati relativi alla entità inviata nella risposta (lunghezza e tipo mime), fornisce informazioni relativamente al set di caratteri utilizzato dal browser (in http questo valore viene trasmesso nell' header Accept-Charset del protocollo) infine fornisce alla servlet l'accesso al canale di comunicazione per inviare l'entità al client. I due metodi deputati alla trasmissione sono quelli definiti nella interfaccia ServletResponse. Il primo, *ServletOutputStream getOutputStream()*, restituisce un oggetto di tipo *ServletOutputStream* e consente di inviare dati al client in forma binaria (ad esempio il browser richiede il download di un file) . Il secondo *PrintWriter getWriter()* restituisce un oggetto di tipo *PrintWriter* e quindi consente di trasmettere entità allo stesso modo di una System.out. Utilizzando questo secondo metodo, il codice della servlet mostrato nel paragrafo precedente diventa quindi:

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

public class Test extends HttpServlet
{
    String initparameter=null;

    public void init(ServletConfig config)
    {
            initparameter = config.getInitParameter("INPUTPARAMS");
    }
    public void service (HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
    {
            res.setContentType("text/html");
            PrintWriter out = res. getWriter ();
            out.println("<html>");
            out.println("<head><title>Test</title></head>");
            out.println("<body>");
            out.println("<h1>Il parametro di input vale "+ initparameter +"</h1>");
            out.println("</body></html>");
    }

}
```

Nel caso in cui sia necessario utilizzare il metodo *setContentType*, sarà obbligatorio effettuarne la chiamata prima di utilizzare il canale di output.

I metodi specializzati di HttpServletResponse

Nel paragrafo precedente abbiamo analizzato i metodi di HttpServletResponse ereditati a partire dalla interfaccia ServletResponse. Come si vede dal codice riportato nel paragrafo precedente, HttpServletResponse specializza un oggetto

“response” affinché possa utilizzare gli strumenti tipici del protocollo http mediante metodi specializzati e legati alla definizione del protocollo.

I metodi definiti all’interno di questa interfaccia coprono la possibilità di modificare o aggiungere campi all’interno dell’ header del protocollo http, metodi per lavorare utilizzando i cookie, metodi per effettuare url encoding o per manipolare errori http. E’ inoltre possibile mediante il metodo sendRedirect() provocare il reindirizzamento della connessione verso un altro server sulla rete.

Notificare errori utilizzando Java Servlet

All’interno della interfaccia HttpServletResponse, oltre ai prototipi dei metodi vengono dichiarate tutta una serie di costanti che rappresentano i codici di errore come definiti dallo standard http. Il valore di queste variabili costanti può essere utilizzato con i metodi *sendError(..)* e *setStatus(…)* per inviare al browser particolari messaggi con relativi codici di errore. Un esempio è realizzato nella servlet seguente:

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

public class TestStatus extends HttpServlet
{
    String initparameter=null;

    public void init(ServletConfig config)
    {
    }
    public void service (HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
    {
            res.setContentType(“text/html”);
            ServletOutputStream out = res.getOutputStream();
            out.println(“<html>”);
            out.println(“<head><title>Test Status Code</title></head>”);
            out.println(“<body>”);
            res.sendError(HttpServletResponse. SC_OK, “Il sito e stato
                                temporaneamente sospeso”);
            out.println(“</body></html>”);
    }

}
```

L’oggetto HttpServletRequest

Come oggetti di tipo HttpServletResponse rappresentano una risposta http, oggetti di tipo HttpServletRequest rappresentano una richiesta http.

```
package javax.servlet;

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

public interface ServletRequest
{
    public Object getAttribute(String name);
    public Enumeration getAttributeNames();
```

```
    public String getCharacterEncoding();
    public int getContentLength();
    public String getContentType();
    public ServletInputStream getInputStream() throws IOException;
    public String getParameter(String name);
    public Enumeration getParameterNames();
    public String[] getParameterValues(String name);
    public String getProtocol();
    public BufferedReader getReader() throws IOException;
    public String getRealPath(String path);
    public String getRemoteAddr();
    public String getRemoteHost();
    public String getScheme();
    public String getServerName();
    public int getServerPort();
    public Object setAttribute(String name, Object attribute);
}

package javax.servlet.http;

import java.util.*;
import java.io.*;

public interface HttpServletRequest extends javax.servlet.ServletRequest {
    String getAuthType();
    Cookie[] getCookies();
    long getDateHeader(String name);
    String getHeader(String name);
    Enumeration getHeaderNames();
    int getIntHeader(String name);
    String getMethod();
    String getPathInfo();
    String getPathTranslated();
    String getQueryString();
    String getRemoteUser();
    String getRequestedSessionId();
    String getRequestURI();
    String getServletPath();
    HttpSession getSession();
    HttpSession getSession(boolean create);
    boolean isRequestedSessionIdFromCookie();
    boolean isRequestedSessionIdFromUrl();
    boolean isRequestedSessionIdFromURL();
    boolean isRequestedSessionIdValid();
}
```

Mediante i metodi messi a disposizione da questa interfaccia, è possibile accedere ai contenuti della richiesta http inviata dal client compreso eventuali parametri o entità trasportate all'interno del pacchetto http. I metodi *ServletInputStream getInputStream()* e *BufferedReader getReader()* ci permettono di accedere ai dati trasportati dal protocollo. Il metodo *getParameter(String)* ci consente di ricavare i valori dei parametri contenuti all'interno della query string della richiesta referenziandoli tramite il loro nome.

Esistono inoltre altri metodi che consentono di ottenere meta informazioni relative alla richiesta: ad esempio i metodi

```
    …..
    public String getRealPath(String path);
    public String getRemoteAddr();
```

```
public String getRemoteHost();
public String getScheme();
public String getServerName();
public int getServerPort();
….
```

Nel prossimo esempio utilizzeremo questi metodi per implementare una servlet che stampa a video tutte le informazioni relative alla richiesta da parte del client:

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

public class Test extends HttpServlet
{
    String initparameter=null;

    public void service (HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
    {
            res.setContentType("text/html");
            PrintWriter out = res. getWriter ();
            out.println("<html>");
            out.println("<head><title>Test</title></head>");
            out.println("<body>");
            out.println("<b>Carachter encoding</b>"
                        +req.getCharacterEncoding()+"<BR>");
            out.println("<b>Mime tipe dei contenuti</b>"
                        +req.getContentType()+"<BR>");
            out.println("<b>Protocollo</b>"
                        +req.getProtocol()+"<BR>");
            out.println("<b>Posizione fisica:</b>"
                        +req.getRealPath()+"<BR>");
            out.println("<b>Schema:</b>"
                        +req.getScheme()+"<BR>");
            out.println("<b>Nome del server:</b>"
                        +req.getServerName()+"<BR>");
            out.println("<b>Porta del server:</b>"
                        +req.getServerPort()+"<BR>");
            out.println("</body></html>");
            out.close();
    }

}
```

Altri metodi di questa interfaccia ci consentono di utilizzare i cookie esistenti sul client oppure ottenere meta informazioni relative al client che ha inviato la richiesta.

Inviare dati mediante la query string

Nel capitolo 13 abbiamo introdotto la trasmissione di dati tra client e applicazione web. Ricordando quanto detto, utilizzando il metodo GET del protocollo http i dati vengono appesi alla URL che identifica la richiesta nella forma di coppie nome=valore separate tra loro dal carattere "&". Questa concatenazione di coppie è detta query string ed è separata dall'indirizzo della risorsa dal carattere "?". Ad esempio la URL

http://www.java-net.tv/servelt/Hello?nome=Massimo&cognome=Rossi

contiene la query string *"?nome=Massimo&cognome=Rossi*". Abbiamo inoltre accennato al fatto che se il metodo utilizzato è il metodo POST, i dati non vengono trasmessi in forma di query string ma vengono accodati nella sezione dati del protocollo http.

Le regole utilizzare in automatiche dal browser e necessarie al programmatore per comporre la query string sono le seguenti:

La query string inizia con il carattere "?".
Utilizza coppie nome=valore per trasferire i dati.
Ogni coppia deve essere separata dal carattere "&".
Ogni carattere spazio deve essere sostituito dalla sequenza %20.
Ogni carattere % deve essere sostituito dalla sequenza %33.

I valori di una query string possono essere recuperati da una servlet utilizzando i metodi messi a disposizione da oggetti di tipo *HttpServletRequest*. In particolare nell'interfaccia HttpServletRequest vengono definiti quattro metodi utili alla manipolazione dei parametri inviati dal browser.

```
public String getQueryString ();
public Enumeration getParameterNames();
public String getParameter(String name);
public String[] getParameterValues(String name);
```

Il primo di questi metodi ritorna una stringa contenente la query string se inviata dal client (null in caso contrario), il secondo ritorna una Enumerazione dei nomi dei parametri contenuti nella query string, il terzo il valore di un parametro a partire dal suo nome ed infine il quarto ritorna un array di valori del parametro il cui nome viene passato in input. Quest'ultimo metodo viene utilizzato quando il client utilizza oggetti di tipo "checkbox" che possono prendere più valori contemporaneamente.

Nel caso in cui venga utilizzato il metodo POST il metodo *getQueryString* risulterà inutile dal momento che questo metodo non produce la query string.

Query String e Form Html

Il linguaggio HTML può produrre URL contenenti query string definendo all'interno della pagina HTML dei form. La sintassi per costruire form HTML è la seguente:

```
<FORM method=[GET/POST]  action=URL>
    <INPUT type=[text|textarea|select|hidden|option]  name=nomedelvalore
                                                        [value= valoreiniziale1]>
    <INPUT type=[text|textarea|select|hidden|option]  name=nomedelvalore2
                                                        [value= valoreiniziale2]>
    …….
    <INPUT type=[text|textarea|select|hidden|option]  name=nomedelvaloreN
                                                        [value= valoreinizialeN]>
    <INPUT type=SUBMIT value=labeldelpulsante>
</FORM>
```

Nell'istante in cui l'utente preme il pulsante SUBMIT, il browser crea una query string contenente le coppie nome=valore che identificano i dati nel form e la appende alla URL che punterà alla risorsa puntata dal campo action del tag <FORM>.

Vediamo un form di esempio:

esempio.html

```
<html>
<body>
    <FORM method=GET  action=http://www.javanet.tv/Prova>
            <INPUT type=hidden name=param value="nascosto"><br>
            Nome:<br><INPUT type=text name=nome value=""><br>
            Cognome:<br><INPUT type=text name=cognome value=""><br>
            Età:<br><INPUT type=text name=eta value=""><br>
            <INPUT type=SUBMIT value="Invia I dati">
    </FORM>
</body>
</html>
```

Nella *Figura 15-1* viene riportato il form così come il browser lo presenta all'utente:

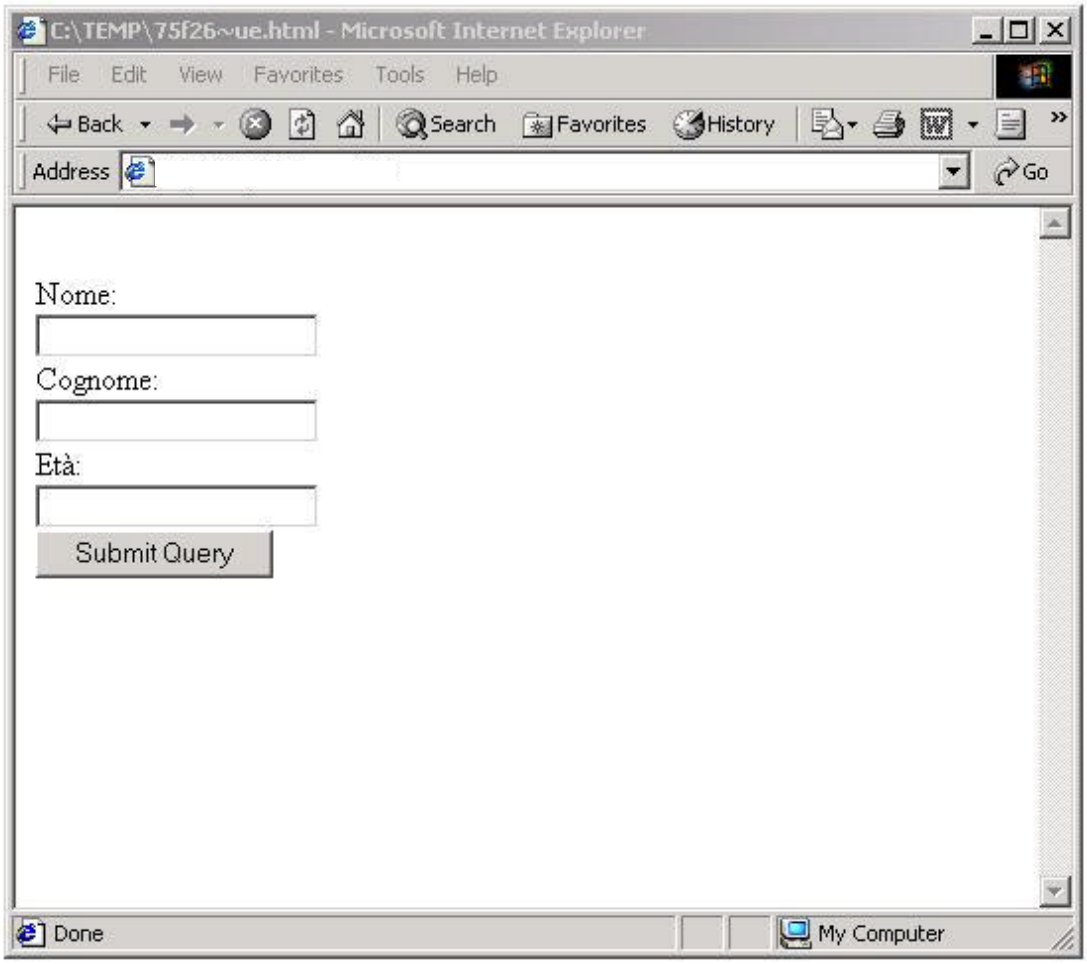


Figura 15-1 *: form html*

Dopo aver inserito i valori nei tre campi di testo, la pressione del pulsante provocherà l'invio da parte del client di una richiesta http identificata dalla URL seguente:

http://www.javanet.tv/Prova?param=nasconto&nome=nomeutente&cognome=cognomeutente&eta=etautente

I limiti del protocollo http : cookies

Il limite maggiore del protocollo http è legato alla sua natura di protocollo non transazionale ovvero non è in grado di mantenere dati persistenti tra i vari pacchetti http. Fortunatamente esistono due tecniche per aggirare il problema: i cookies e la gestione di sessioni utente. I cookies sono piccoli file contenenti una informazione scritta all'interno secondo un certo formato che vengono depositati dal server sul client e contengono informazioni specifiche relative alla applicazione che li genera.

Se utilizzati correttamente consentono di memorizzare dati utili alla gestione del flusso di informazioni creando delle entità persistenti in grado di fornire un punto di appoggio per garantire un minimo di transazionalità alla applicazione web

Formalmente i cookie contengono al loro interno una singola informazione nella forma nome=valore più una serie di informazioni aggiuntive che rappresentano:

Il dominio applicativo del cookie;
Il path della applicazione;
La durata della validità del file;
Un valore booleano che identifica se il cookie è criptato o no.

Il dominio applicativo del cookie consente al browser di determinare se, al momento di inviare una richiesta http dovrà associarle il cookie da inviare al server. Un valore del tipo www.java-web.tv indicherà al browser che il cookie sarà valido solo per la macchina www all'interno del dominio java-web.tv . Il path della applicazione rappresenta il percorso virtuale della applicazione per la quale il cookie è valido.

Un valore del tipo "/" indica al browser che qualunque richiesta http da inviare al dominio definito nel campo precedente dovrà essere associata al cookie. Un valore del tipo "/servlet/ServletProva" indicherà invece al browser che il cookie dovrà essere inviato in allegato a richieste http del tipo *http://www.java-net.tv/servlt/ServletProva*.

La terza proprietà comunemente detta "expiration" indica il tempo di durata della validità del cookie in secondi prima che il browser lo cancelli definitivamente. Mediante questo campo è possibile definire cookie persistenti ossia senza data di scadenza.

Manipolare cookies con le Servlet

Una servlet può inviare uno o più cookie ad un browser mediante il metodo *addCookie(Cookie)* definito nell'interfaccia *HttpServletResponse* che consente di appendere l'entità ad un messaggio di risposta al browser. Viceversa, i cookie associati ad una richiesta http possono essere recuperati da una servlet utilizzando il metodo getCookie() definito nella interfaccia *HttpServletRequest* che ritona un array di oggetti.

Il cookie in Java viene rappresentato dalla classe *javax.servlet.http.Cookie* il cui prototipo è riportato nel codice seguente.

Cookie.java

```
package javax.servlet.http;

public class Cookie implements Cloneable {
    public Cookie(String name, String value);
    public String getComment() ;
    public String getDomain() ;
    public int getMaxAge();
    public String getName();
    public String getPath();
```

```
        public boolean getSecure();
        public String getValue();
        public int getVersion();
        public void setComment(String purpose);
        public void setDomain(String pattern);
        public void setMaxAge(int expiry);
        public void setPath(String uri);
        public void setSecure(boolean flag);
        public void setValue(String newValue);
        public void setVersion(int v);
    }
```

Un esempio completo

Nell'esempio implementeremo una servlet che alla prima chiamata invia al server una serie di cookie mentre per ogni chiamata successiva ne stampa semplicemente il valore contenuto.

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

    public class TestCookie extends HttpServlet
    {
        private int numrichiesta=0;

        public void service (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
        {
                Cookie cookies = null;
                res.setContentType("text/html");
                PrintWriter out = res. getWriter ();
                out.println("<html>");
                out.println("<head><title>Cookie Test</title></head>");
                out.println("<body>");

                switch(numrichiesta)
                {
                        case 0:
                                //Appende 10 cookie alla risposta http
                                for (int i=0; i<10; i++)
                                {
                                        String nome="cookie"+i;
                                        String valore="valore"+i;
                                        cookies = new Cookie(nome,valore);
                                        cookies.setMaxAge(1000);
                                        res.addCookie(cookies);
                                }
                                out.println("<h1>I cookie sono stati appesi a questa
risposta<h1>");
                                numrichiesta++;
                                break;
                        default :
                                //ricavo l'array dei cookie e stampo le
                                //coppie nome=valore
                                Cookie cookies[] = req.getCookies();
                                for (int j=0; j<cookies.length; j++)
                                {
```

```
                                        Cookie appo = cookies[j];
                                        out.println("<h1>"+appo.getName()+" =
                                                    "+appo.getValue()+"<h1>");
                                }
                        }
                        out.println("</body></html>");
                        out.close();
                }

        }
```

Sessioni utente

I cookie non sono uno strumento completo per memorizzare lo stato di una applicazione web. Di fatto i cookie sono spesso considerati dall'utente poco sicuri e pertanto non accettati dal browser che li rifiuterà al momento dell'invio con un messaggio di risposta. Come se non bastasse il cookie ha un tempo massimo di durata prima di essere cancellato dalla macchina dell'utente.

Servlet risolvono questo problema mettendo a disposizione del programmatore la possibilità di racchiudere l'attività di un client per tutta la sua durata all'interno di sessioni utente.

Una servlet può utilizzare una sessione per memorizzare dati persistenti, dati specifici relativi alla applicazione e recuperarli in qualsiasi istante sia necessario. Questi dati possono essere inviati al client all'interno dell'entità prodotta.

Ogni volta che un client effettua una richiesta http, se in precedenza è stata definita una sessione utente legata al particolare client, il servlet container identifica il client e determina quale sessione è stata associata ad esso. Nel cosa in cui la servlet la richieda, il container gli mette disposizione tutti gli strumenti per utilizzala.

Ogni sessione creata dal container è associata in modo univoco ad un identificativo o ID. Due sessioni non possono essere associate ad uno stesso ID.

Sessioni dal punto di vista di una servlet

Il servlet container contiene le istanze delle sessioni utente rendendole disponibili ad ogni servlet che ne faccia richiesta (*Figura 15-2*).

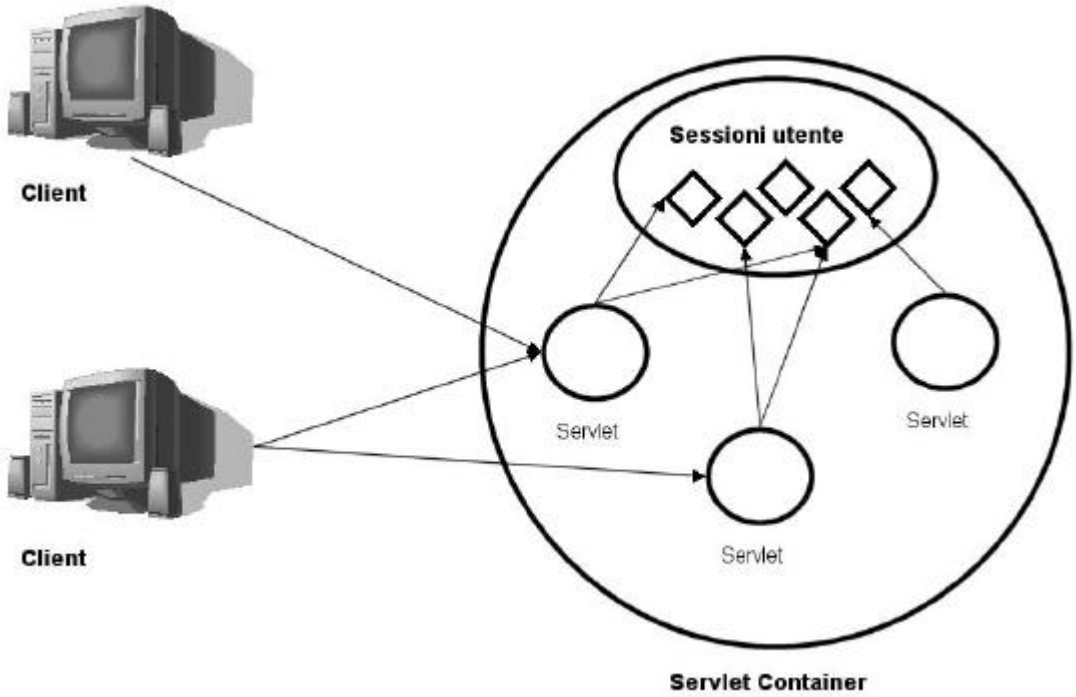


Figura 15-2 : *le sessioni sono accessibili da tutte le servlet*

Ricevendo un identificativo dal client, una servlet può accedere alla sessione associata all'utente. Esistono molti modi per consentire al client di tracciare l'identificativo di una sessione, tipicamente viene utilizzato il meccanismo dei cookie persistenti. Ogni volta che un client esegue una richiesta http, il cookie contenente l'identificativo della richiesta viene trasmesso al server che ne ricava il valore contenuto e lo mette a disposizione delle servlet. Nel caso in cui il browser non consenta l'utilizzo di cookies esistono tecniche alternative che risolvono il problema.

Tipicamente una sessione utente deve essere avviata da una servlet dopo aver verificato se già non ne esista una utilizzando il metodo *getSession(boolean)* di *HttpServletRequest*. Se il valore boolean passato al metodo vale *true* ed esiste già una sessione associata all'utente il metodo semplicemente ritornerà un oggetto che la rappresenta, altrimenti ne creerà una ritornandola come oggetto di ritorno del metodo. Al contrario se il parametro di input vale *false*, il metodo tornerà la sessione se già esistente null altrimenti.

Quando una servlet crea una nuova sessione, il container genera automaticamente l'identificativo ed appende un cookie contenente l' ID alla risposta http in modo del tutto trasparente alla servlet.

La classe HttpSession

Un oggetto di tipo *httpSession* viene restituito alla servlet come parametro di ritorno del metodo *getSession(boolean)* e viene definito dall'interfaccia *javax.servlet.http.HttpSession* .

HttpSession.java

```
package javax.servlet.http;

public interface HttpSession {
    long getCreationTime();
```

```
    String getId();
    long getLastAccessedTime();
    int getMaxInactiveInterval();
    HttpSessionContext getSessionContext();
    Object getValue(String name);
    String[] getValueNames();
    void invalidate();
    boolean isNew();
    void putValue(String name, Object value);
    void removeValue(String name);
    int setMaxInactiveInterval(int interval);
}
```

Utilizzando i metodi

getId()
long getCreationTime()
long getLastAccessedTime()
int getMaxInactiveInterval()
boolean isNew()

possiamo ottenere le meta informazioni relative alla sessione che stiamo manipolando. E' importante notare che i metodi che ritornano un valore che rappresenta un "tempo" rappresentano il dato in secondi. Sarà quindi necessario operare le necessarie conversioni per determinare informazioni tipo date ed ore. Il significato dei metodi descritti risulta chiaro leggendo il nome del metodo. Il primo ritorna l'identificativo della sessione, il secondo il tempo in secondi trascorso dalla creazione della sessione, il terzo il tempo in secondi trascorso dall'ultimo accesso alla sezione, infine il quarto l'intervallo massimo di tempo di inattività della sessione.

Quando creiamo una sessione utente, l'oggetto generato verrà messo in uno stato di **NEW** che sta ad indicare che la sessione è stata creata ma non è attiva. Dal un punto di vista puramente formale è ovvio che per essere attiva la sessione deve essere accettata dal client ovvero una sessione viene accettata dal client solo nel momento in cui invia al server per la prima volta l'identificativo della sessione. Il metodo *isNew()* restituisce un valore booleano che indica lo stato della sessione.

I metodi

void putValue(String name, Object value)
void removeValue(String name)
String[] getValueNames(String name)
Object getValue(String name)

Consentono di memorizzare o rimuovere oggetti nella forma di coppie nome=oggetto all'interno della sessione consentendo ad una servlet di memorizzare dati (in forma di oggetti) all'interno della sessione per poi utilizzarli ad ogni richiesta http da parte dell'utente associato alla sessione.

Questi oggetti saranno inoltre accessibili ad ogni servlet che utilizzi la stessa sessione utente potendoli recuperare conoscendo il nome associato.

Un esempio di gestione di una sessione utente

Nel nostro esempio creeremo una servlet che conta il numero di accessi che un determinato utente ha effettuato sul sistema utilizzando il meccanismo delle sessioni.

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

```
import java.io.* ;

public class TestSession extends HttpServlet
{

    public void service (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
            res.setContentType("text/html");
            PrintWriter out = res. getWriter ();
            out.println("<html>");
            out.println("<head><title>Test di una sessione servlet</title></head>");
            out.println("<body>");

            HttpSession sessione = req.getSession(true);
            if(session.isNew())
            {
                    out.println("<strong>Id della sessione: </strong>"
                                                    +session.getId()+"<br>");
                    out.println("<strong>Creata al tempo: </strong>"
                                            +session.creationTime()+"<br>");
                    out.println("<strong>Questa è la tua prima “
                                            +”connessione al server </strong>");
                    session.putValue("ACCESSI", new Integer(1));
            }else {
            int accessi = ((Integer)session.getValue("ACCESSI")).intValue();
            accessi++;
            session.putValue("ACCESSI", new Integer(accessi));
            out.println("<strong>Questa è la tua connessione numero: </strong>"
                                                                    +accessi);
            }


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

}
```

Durata di una sessione utente

Una sessione utente rappresenta un oggetto transiente la cui durata deve essere limitata al periodi di attività dell’utente sul server. Utilizzando i metodi

void invalidate();
int setMaxInactiveInterval(int interval)

è possibile invalidare una sessione o disporre che una volta superato l’intervallo massimo di inattività la servlet venga automaticamente resa inattiva dal container.

URL rewriting

Il browser ha la facoltà di accettare o no cookies da parte di un server web. Quando ciò accade è impossibile per il server tracciare l’identificativo della sessione e di conseguenza mettere in grado una servlet di accederle.

Un metodo alternativo è quello di codificare l’identificativo della sessione all’interno della URL che il browser invia al server all’interno di una richiesta http.

Questa metodologia deve necessariamente essere supportata dal server che dovrà prevedere l'utilizzo di caratteri speciali all'interno della URL. Server differenti potrebbero utilizzare metodi differenti.

L'interfaccia *HttpServletResponse* ci mette a disposizione il metodo *encodeURL(String)* che prende come parametro di input una URL, determina se è necessario riscriverla ed eventualmente codifica all'interno della URL l'identificativo della sessione.

Capitolo 16
JavaServer Pages

Introduzione

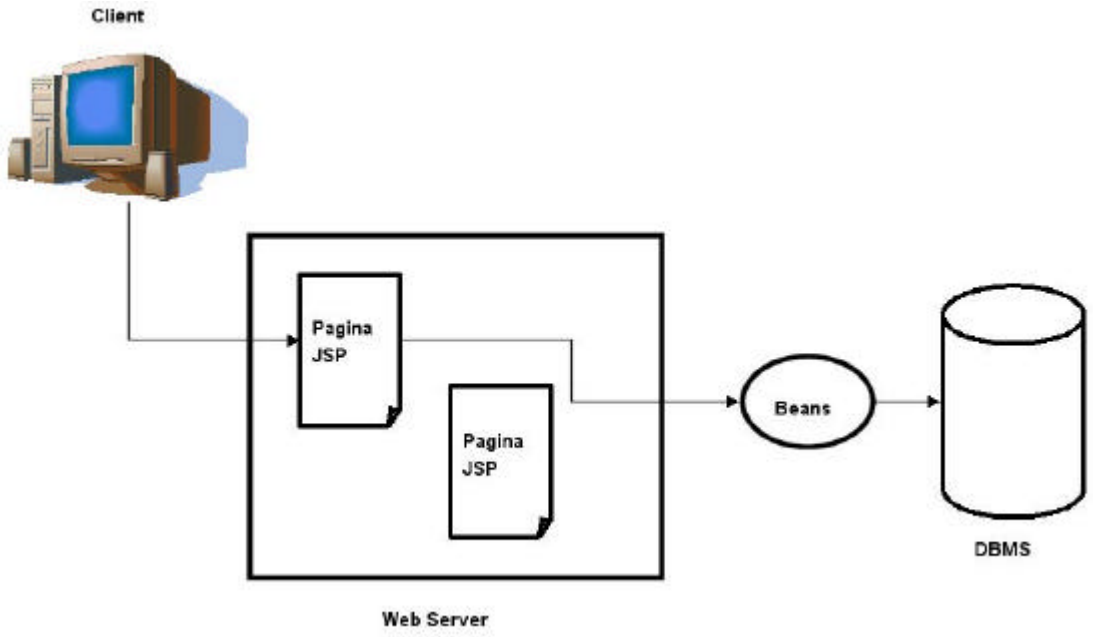
La tecnologia JavaServer Pages rappresenta un ottimo strumento per scrivere pagine web dinamiche ed insieme a servlet consente di separare le logiche di business della applicazione web (servlet) dalle logiche di presentazione.

Basato su Java, JavaServer Pages sposano il modello "write once run anywhere", consentono facilmente l'uso di classi Java, di JavaBeans o l'accesso ad Enterprise JavaBeans.

JavaServer Pages

Una pagina JSP è un semplice file di testo che fonde codice html a codice Java a formare una pagina dai contenuti dinamici. La possibilità di fondere codice html con codice Java senza che nessuno interferisca con l'altro consente di isolare la rappresentazione dei contenuti dinamici dalle logiche di presentazione. Il disegnatore potrà concentrarsi solo sulla impaginazione dei contenuti che saranno inseriti dal programmatore che non dovrà preoccuparsi dell'aspetto puramente grafico.

Da sole, JavaServer Pages consentono di realizzare applicazioni web dinamiche (*Figura 16-1*) accedendo a componenti Java contenenti logiche di business o alla base dati del sistema. In questo modello il browser accede direttamente ad una pagina JSP che riceve i dati di input, li processa utilizzando eventualmente oggetti Java, si connette alla base dati effettuando le operazioni necessarie e ritorna al client la pagina html prodotta come risultato della processazione dei dati.



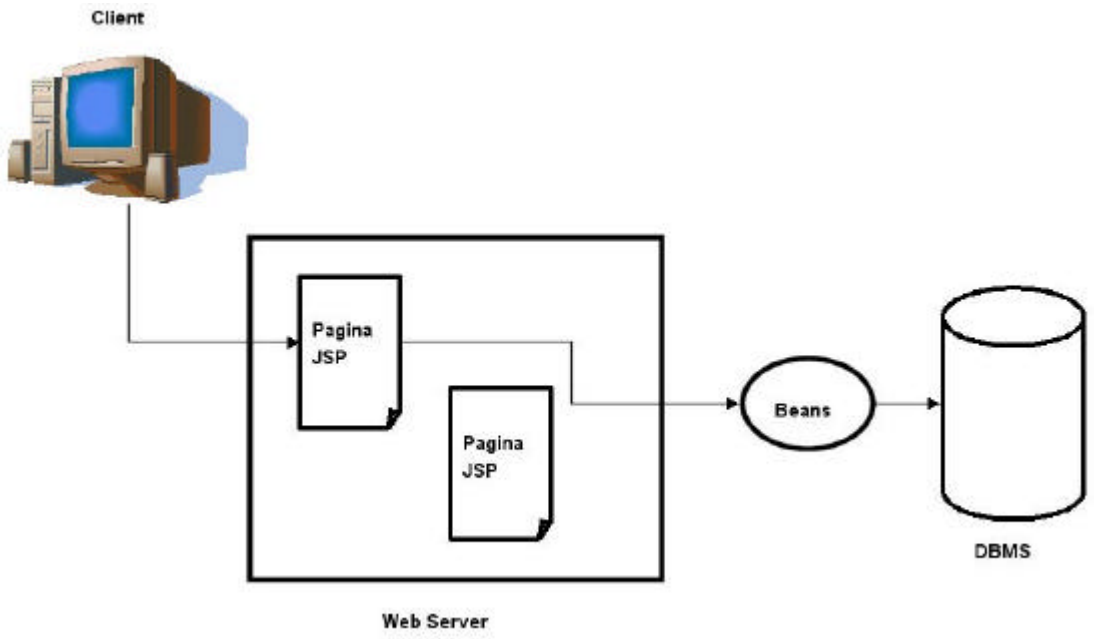


Figura 16-1 : *primo modello di accesso*

Il secondo modello, e forse il più comune, utilizza JavaServer Pages come strumento per sviluppare template demandando completamente a servlet la processazione dei dati di input (*Figura 16-2*).

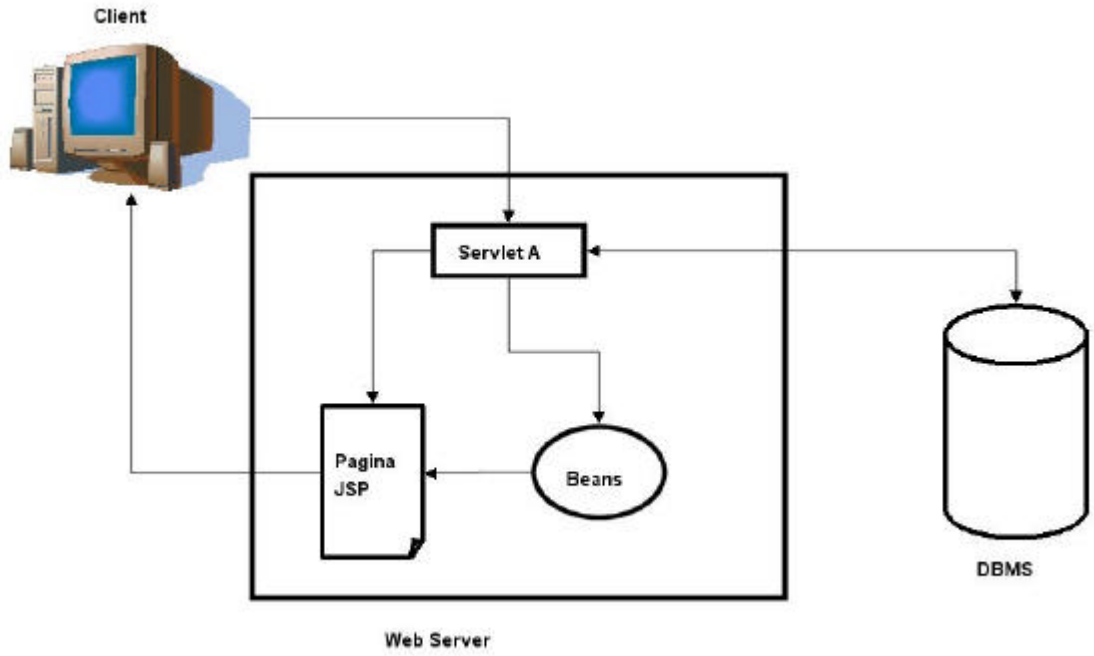


Figura 16-2 : *secondo modello di accesso*

In questo nuovo modello, il browser invia la sua richiesta ad una servlet che si preoccuperà di processare i dati di input utilizzando eventualmente via JDBC la base dati del sistema. La servlet ora genererà alcuni oggetti (non più pagine html) come prodotto della esecuzione del metodo service(…). Utilizzando gli strumenti messi a disposizione dal container, la servlet potrà inviare gli oggetti prodotti ad una pagina JavaServer Pages che si preoccuperà solamente di ricavare il contenuto di questi

oggetti inserendoli all'interno del codice html. Sara infine la pagina JSP ad inviare al client la pagina prodotta.

Compilazione di una pagina JSP

Se dal punto di vista del programmatore una pagina JSP è un documento di testo contenente tag html e codice Java, dal punto di vista del server una pagina JSP è utilizzata allo stesso modo di una servlet. Di fatto, nel momento del primo accesso da parte dell'utente, la pagina JSP richiesta viene trasformata in un file Java e compilata dal compilatore interno della virtual machine. Come prodotto della compilazione otterremo una classe Java che rappresenta una servlet di tipo HttpServlet che crea una pagina html e la invia al client.

Tipicamente il web server memorizza su disco tutte le definizioni di classe ottenute dal processo di compilazione appena descritto per poter riutilizzare il codice già compilato. L'unica volta che una pagina JSP viene compilata è al momento del suo primo accesso da parte di un client o dopo modifiche apportate dal programmatore affinché il client acceda sempre alla ultima versione prodotta.

Scrivere pagine JSP

Una pagina JSP deve essere memorizzata all'interno di un file di testo con estensione .jsp . E' questa l'estensione che il web server riconosce per decidere se compilare o no il documento. Tutte le altre estensioni verranno ignorate.

Ecco quindi un primo esempio di pagina JSP:

esempio1.jsp

```
<html>
    <body>
            <h1> Informazioni sulla richiesta http </h1>
            <br>
            Metodo richiesto : <%= request.getMethod() %>
            <br>
            URI : <%= request.getRequestURI() %>
            <br>
            Protocollo : <%= request.getProtocol() %>
            <br>
    <body>
</html>
```

Una pagina JSP come si vede chiaramente dall'esempio è per la maggior parte formata da codice html con in più un piccolo insieme di tag addizionali. Nel momento in cui avviene la compilazione della pagina il codice html viene racchiuso in istruzioni di tipo *out.println(codicehtml)* mentre il codice contenuto all'interno dei tag aggiuntivi viene utilizzato come codice eseguibile. L'effetto prodotto sarà comunque quello di inserire all'interno del codice html valori prodotti dalla esecuzione di codice Java.

I tag aggiuntivi rispetto a quelli definiti da html per scrivere pagine jsp sono tre: espressioni, scriptlet, dichiarazioni. Le espressioni iniziano con la sequenza di caratteri "<%=" e terminano con la sequenza "%>", le istruzioni contenute non devono terminare con il carattere ";" e devono ritornare un valore che può essere promosso a stringa. Ad esempio è una espressione JSP la riga di codice:

```
<%=  new Integer(5).toString() %>
```

Le scriptlet iniziano con la sequenza “<%”, terminano con la sequenza “%>“ e devono contenere codice Java valido. All’interno di questi tag non si possono scrivere definizioni di classi o di metodi, ma consentono di dichiarare variabili visibili all’interno di tutta la pagina JSP. Una caratteristica importante di questi tag è che il codice Java scritto all’interno non deve essere completo ovvero è possibile fondere blocchi di codice Java all’interno di questi tag con blocchi di codice html. Ad esempio:

```
<% for(int i=0; i<10; i++) { %>
        <strong> Il valore di I : <%=  new Integer(i).toString() %> </strong>
<% } %>
```

Infine le dichiarazioni iniziano con la sequenza “<%!” e terminano con la sequenza “%>” e possono contenere dichiarazioni di classi o di metodi utilizzabili solo all’interno della pagina, e a differenza del caso precedente, devono essere completi. Nell’esempio utilizziamo questi tag per definire un metodo StampaData() che ritorna la data attuale in formato di stringa:

```
<%!
        String StampaData()
        {
                return new Date().toString();
        }
%>
```

Nel prossimo esempio di pagina jsp, all’interno di un loop di dieci cicli effettuiamo un controllo sulla variabile intera che definisce il contatore del ciclo. Se la variabile è pari stampiamo il messaggio “Pari”, altrimenti il messaggio “Dispari”.

Esempio2.jsp

```
<html>
    <body>
            <% for(int i=0; i<10; i++) {
                    if(i%2==0) {
            %>
                    <h1>Pari</h1>
            <% } else { %>
                    <h2>Dispari</h2>
            <%
                    }
                }
            %>
    <body>
</html>
```

L’aspetto più interessante del codice nell’esempio è proprio quello relativo alla possibilità si spezzare il codice Java contenuto all’interno delle scriptlets per dar modo al programmatore di non dover fondere tag html all’interno sorgente Java.

Invocare una pagina JSP da una servlet

Nel secondo modello di accesso ad una applicazione web trattato nel paragrafo 2 di questo capitolo abbiamo definito una pagina jsp come “template” ossia meccanismo di presentazione dei contenuti generati mediante una servlet.

Affinché sia possibile implementare quanto detto, è necessario che esista un meccanismo che consenta ad una servlet di trasmettere dati prodotti alla pagina JSP. Sono necessarie alcune considerazioni: primo, una pagina JSP viene tradotta

una in classe Java di tipo HttpServlet e di conseguenza compilata ed eseguita all'interno dello stesso container che fornisce l'ambiente alle servlet. Secondo, JSP ereditano quindi tutti i meccanismi che il container mette a disposizione delle servlet compresso quello delle sessioni utente.

Lo strumento messo a disposizione dal container per rigirare una chiamata da una servlet ad una JavaServer Pages passando eventualmente parametri è l'oggetto *RequestDispatcher* restituito dal metodo

```
public ServletContext getRequestDispatcher(String Servlet_or_JSP_RelativeUrl)
```

definito nella interfaccia *javax.servlet.ServletContext*.

RequestDispatcher.java

```
package javax.servlet;


public interface RequestDispatcher {
    public void forward(ServletRequest req, ServletResponse res);
    public void include(ServletRequest req, ServletResponse res);
}
```

tramite il metodo forward(ServletRequest req, ServletResponse res) è possibile reindirizzare la chiamata alla Servlet o JavaServer Page come indicato dalla URL definita dal parametro di input di *getRequestDispatcher*. Il metodo *public void include(ServletRequest req, ServletResponse res)*, pur producendo a sua volta un reindirizzamento, inserisce il risultato prodotto dalla entità richiamata all'interno del risultato prodotto dalla servlet che ha richiamato il metodo.

Nell'esempio viene definita una servlet che trasferisce tramite requestDispatcher la richiesta http alla pagina jsp "/appo.jsp" memorizzata nella root del web server.

```
public class Redirect extends HttpServlet
{

    public void service (HttpServletRequest req, HttpServletResponse res)
                            throws ServletException, IOException
    {
            ServletContext contesto = getServletContext();
            RequestDispatcher rd = contesto.getRequestDispatcher("/appo.jsp");
            try
            {
                    rd.forward(req, res);
            }
            catch(ServletException e)
            {
                    System.out.println(e.toString());
            }
    }

}
```

Capitolo 17
JavaServer Pages – Nozioni Avanzate

Introduzione

I tag JSP possono essere rappresentati in due modi differenti: “Short-Hand” ed “XML equivalent”. Ogni forma delle due prevede i seguenti tag aggiuntivi rispetto ad html:

Tipo	Short Hand	XML
Scriptlet	<% codice java %>	<jsp :scriptlet> codice java </jsp :scriptlet>
Direttive	<%@ tipo attributo %>	<jsp:directive.tipo attributo />
Dichiarazione	<%! Dichiarazione %>	<jsp:decl> dichiarazione; </jsp:decl>
Espressione	<%= espressione %>	<jsp:expr> espressione; </jsp:expr >
Azione	NA	<jsp:useBean ….> <jsp:include ….> <jsp:getProperty ….> ecc.

Direttive

Le direttive forniscono informazioni aggiuntive sulla all’ambiente all’interno della quale la JavaServer Page è in esecuzione. Le possibili direttive sono due:

Page : informazioni sulla pagina
Include – File da includere nel documento

e possono contenere gli attributi seguenti:

Attributo e possibili valori	Descrizione
language=”Java”	Dichiara al server il linguaggio utilizzato all’interno della pagina JSP
extends=”package.class”	Definisce la classe base a partire dalla quale viene definita la servlet al momento della compilazione. Generalmente non viene utilizzato.
import=”package.*, package.class”	Simile alla direttiva import di una definizione di classe Java. Deve essere una delle prime direttive e comunque comparire prima di altri tag JSP.
session=”true|false”	Di default session vale true e significa che i dati appartenenti alla sessione utente sono disponibili dalla pagina JSP
buffer=”none|8kb|dimensione”	Determina se l’output strema della JavaServer Pages utilizza o no un buffer

	di scrittura dei dati. Di default la dimensione è di 8k. Questa direttiva va utilizzata affiancata dalla direttiva autoflush
autoFlush=”true|false”	Se impostato a true svuota il buffer di output quando risulta pieno invece di generare una eccezione
isThreadSafe=”true|false”	Di default l’attributo è impostato a true e indica all’ambiente che il programmatore si preoccuperà di gestire gli accessi concorrenti mediante blocchi sincronizzati. Se impostato a false viene utilizzata di default l’interfaccia SinglkeThreadModel in fase di compilazione.
info=”info_text”	Fornisce informazioni sulla pagina che si sta accedendo attraverso il metodo Servlet.getServletInfo().
errorPage=”error_url”	Fornisce il path alla pagina jsp che verrà richiamata in automatico per gestire eccezioni che non vengono controllate all’interno della pagina attuale.
isErrorPage=”true|false”	Definisce la pagina come una una pagina di errore.
contentType=”ctinfo”	Definisce il mime tipe della pagina prodotta dalla esecuzione.

Un esempio di blocco di direttive all’interno di una pagina JSP è il seguente:

```
<%@ page  language=”Java” session=”true” errorPage=”/err.jsp” %>
<%@ page import= “java.lang.*, java.io.*”%>
<%@ include  file=”headers/intestazione.html” %>
```

Dichiarazioni

Una dichiarazione rappresenta dal punto di vista del web server che compila la JavaServer Page il blocco di dichiarazione dei dati membro o dei metodi della classe Servlet generata. Per definire un blocco di dichiarazioni si utilizza il tag <%! Dichiarazione %>. Un esempio:

```
<%! String nome =”mionome”;
      int tipointero=1;
      private String stampaNome()
      {
            return nome;
      }
%>
```

Scriptlets

Come già definito nel capitolo precedente, le scriptlets rappresentano blocchi di codice java incompleti e consentono di fondere codice Java con codice html. Oltre a poter accedere a dati e metodi dichiarati all’interno di tag di dichiarazione consente di accedere ad alcuni oggetti impliciti ereditati dall’ambiente servlet.

Gli oggetti in questione sono otto e sono i seguenti:

request : rappresenta la richiesta del client
response : rappresenta la risposta del client
out : rappresenta lo strema di output html inviato come risposta al client
session : rappresenta la sessione utente
page : rappresenta la pagina JSP attuale
config : rappresenta i dettagli della configurazione del server
pageContext : rappresenta un container per i metodi relativi alle servlet

Oggetti impliciti : request

Questo oggetto è messo a disposizione della pagina JSP in maniera implicita e rappresenta la richiesta http inviata dall'utente ovvero implementa l'interfaccia *javax.servlet.http.HttpServletRequest*. Come tale questo oggetto mette e disposizione di una pagina JSP gli stessi metodi utilizzati all'interno di una servlet.

Nell'esempio seguente i metodi messi a disposizione vengono utilizzati implicitamente per generare una pagina http che ritorna le informazioni relative alla richiesta:

TestAmbiente.jsp

```
<%@ page  language="Java" session="true" %>
<%@ page import= "java.lang.*, java.io.*", javax.servlet.* , javax.servlet.http.*%>
<html>
<head><title>Test</title></head>
<body>
<b>Carachter encoding</b>
<%=request.getCharacterEncoding() %>
<BR>
<b>Mime tipe dei contenuti</b>
<%=request.getContentType()%>
<BR>
<b>Protocollo</b>
<%=request.getProtocol()%>
<BR>
<b>Posizione fisica:</b>
<%=request.getRealPath()%>
<BR>
<b>Schema:</b>
<%=request.getScheme()%>
<BR>
<b>Nome del server:</b>
<%=request.getServerName()%>
<BR>
<b>Porta del server:</b>
<%=request.getServerPort()%>
<BR>
</body></html>
```

Oggetti impliciti : response

Rappresenta la risposta http che la pagina JSP invia al client ed implementa *javax.servlet.http.HttpServletResponse.* Compito dei metodi messi a disposizione da

questo oggetto è quello di inviare dati di risposta, aggiungere cookies, impostare headers http e ridirezionare chiamate.

Oggetti impliciti: session

Rappresenta la sessione utente come definito per servlet. Anche questo metodo mette a disposizione gli stessi metodi di servlet e consente di accedere ai dati della sessione utente all'interno della pagina JSP.

Capitolo 18
JDBC

Introduzione

JDBC rappresentano le API di J2EE per poter lavorare con database relazionali. Esse consentono al programmatore di inviare query ad un database relazionale, di effettuare delete o update dei dati all'interno di tabelle, di lanciare stored-procedure o di ottenere meta-informazioni relativamente al database o le entità che lo compongono.

JDBC sono modellati a partire dallo standard ODBC di Microsoft basato a sua volta sulle specifiche "X/Open CLI". La differenza tra le due tecnologie sta nel fatto che mentre ODBC rappresenta una serie di "C-Level API", JDBC fornisce uno strato di accesso verso database completo e soprattutto completamente ad oggetti.

Architettura di JDBC

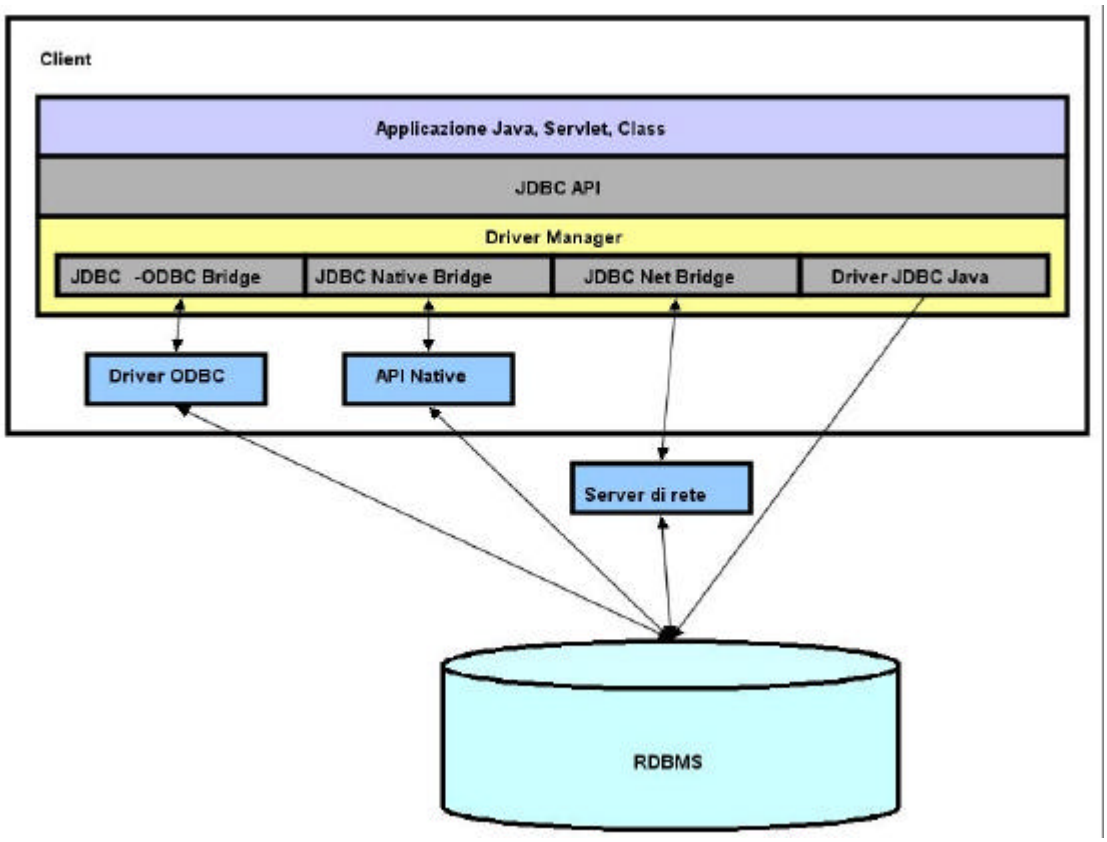


Figura 18-1: *Architettura JDBC*

Architetturalmente JDBC sono suddivisi in due strati principali : il primo che fornisce una interfaccia verso il programmatore, il secondo di livello più basso che fornisce invece una serie di API per i produttori di drivers verso database relazionali e nasconde all'utente i dettagli del driver in uso. Questa caratteristica rende la tecnologia indipendente rispetto al motore relazionale con cui il programmatore deve comunicare.

I driver utilizzabili con JDBC sono di quattro tipi: il primo è rappresentato dal bridge jdbc/odbc che utilizzano l'interfaccia ODBC del client per connettersi alla base

dati. Il secondo tipo ha sempre funzioni di bridge (ponte), ma traduce le chiamate JDBC in chiamate di driver nativi già esistenti sulla macchina. Il terzo tipo ha una architettura multi-tier ossia si connette ad un RDBMS tramite un middleware connesso fisicamente al database e con funzioni di proxy. Infine il quarto tipo rappresenta un driver nativo verso il database scritto in Java e compatibile con il modello definito da JDBC.

I driver JDBC di qualsiasi tipo siano vengono caricati dalla applicazione in modo dinamico mediante una istruzione del tipo *Class.forName(package.class)*. Una volta che il driver (classe java) viene caricato è possibile connettersi al database tramite il driver manager utilizzando il metodo *getConnection()* che richiede in input la URL della base dati ed una serie di informazioni necessarie ad effettuare la connessione. Il formato con cui devono essere passate queste informazioni dipende dal produttore dei driver.

Driver di tipo 1

Rappresentati dal bridge jdbc/odbc di SUN, utilizzano l'interfaccia ODBC del client per connettersi alla base dati (*Figura 18-2*).

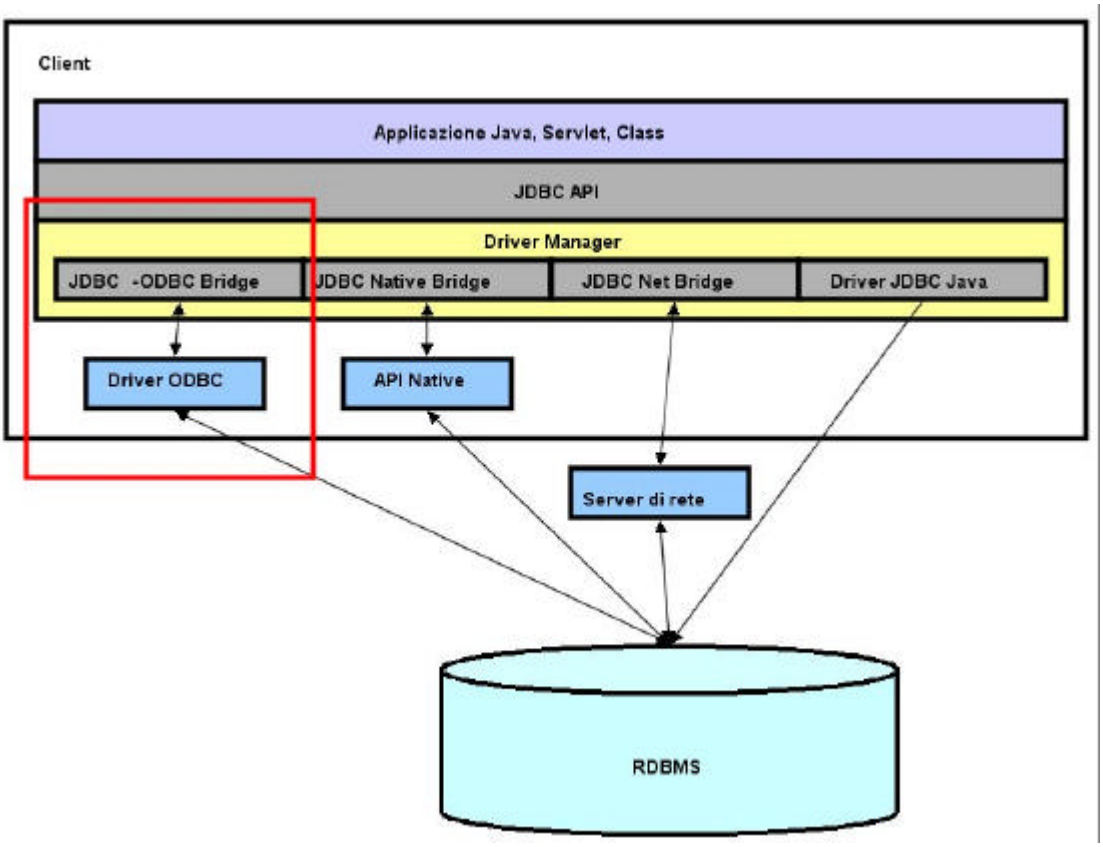


***Figura 18-2** : driver JDBC tipo 1*

Questo tipo di driver sono difficili da amministrare e dipendono dalle piattaforme Microsoft. Il fatto di utilizzare meccanismi intermedi per connettersi alla base dati (lo strato ODBC) fa si che non rappresentano la soluzione ottima per tutte le piattaforme in cui le prestazioni del sistema rappresentano l'aspetto critico.

Driver di tipo 2

I driver di tipo 2 richiedono che sulla macchina client siano installati i driver nativi del database con cui l'utente intende dialogare (*Figura 18-3*). Il driver JDBC convertirà quindi le chiamate JDBC in chiamate compatibili con le API native del server.

L'uso di API scritte in codice nativo rende poco portabili le soluzioni basate su questo tipo di driver. I driver JDBC/ODBC possono essere visti come un caso specifico dei driver di tipo 2.

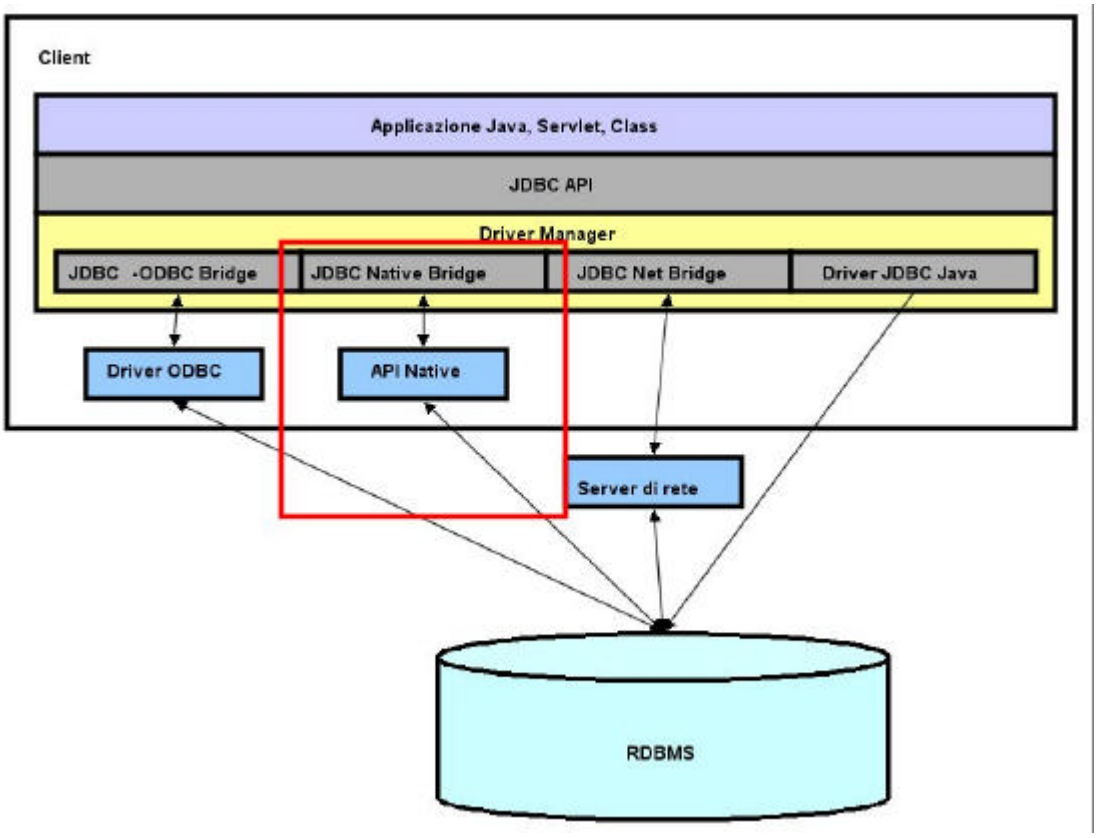


Figura 18-3 *: driver JDBC di tipo 2*

Driver di tipo 3

I driver di tipo 3 consentono di non utilizzare codice nativo sulla macchina client e quindi consentono di costruire applicazioni Java portabili.

Formalmente i driver di tipo 3 sono rappresentati da oggetti che convertono le chiamate JDBC in un protocollo di rete e comunicano con una applicazione middleware chiamata server di rete. Compito del server di rete è quello di rigirare le chiamate da JDBC instradandole verso il server database (*Immagine 4*).

Utilizzare architetture di questo tipo dette "multi-tier" comporta molti vantaggi soprattutto in quelle situazioni in cui un numero imprecisato di client deve connettersi ad una moltitudine di server database. In questi casi infatti tutti i client potranno parlare un unico protocollo di rete, quello conosciuto dal middleware, sarà compito del server di rete tradurre i pacchetti nel protocollo nativo del database con cui il client sta cercando di comunicare.

L'utilizzo di server di rete consente inoltre di utilizzare politiche di tipo caching e pooling oppure di load balancing del carico.

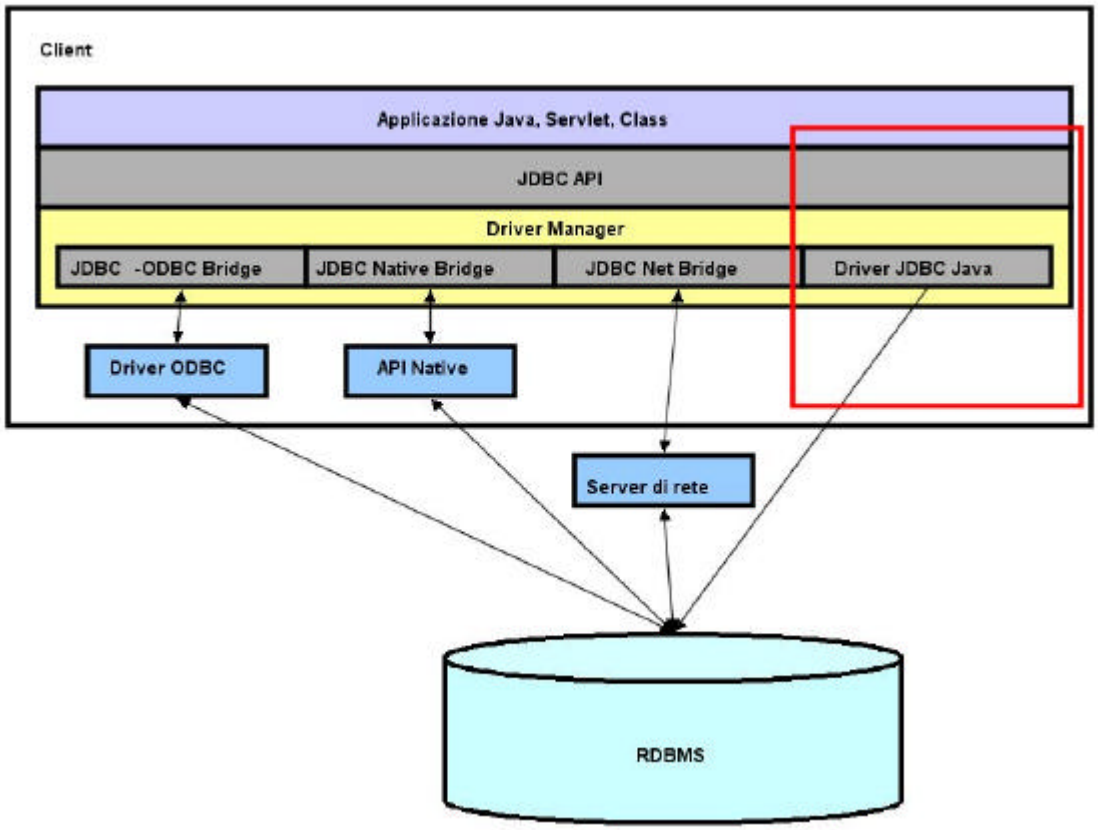


Figura 18-4 *: driver JDBC di tipo 3*

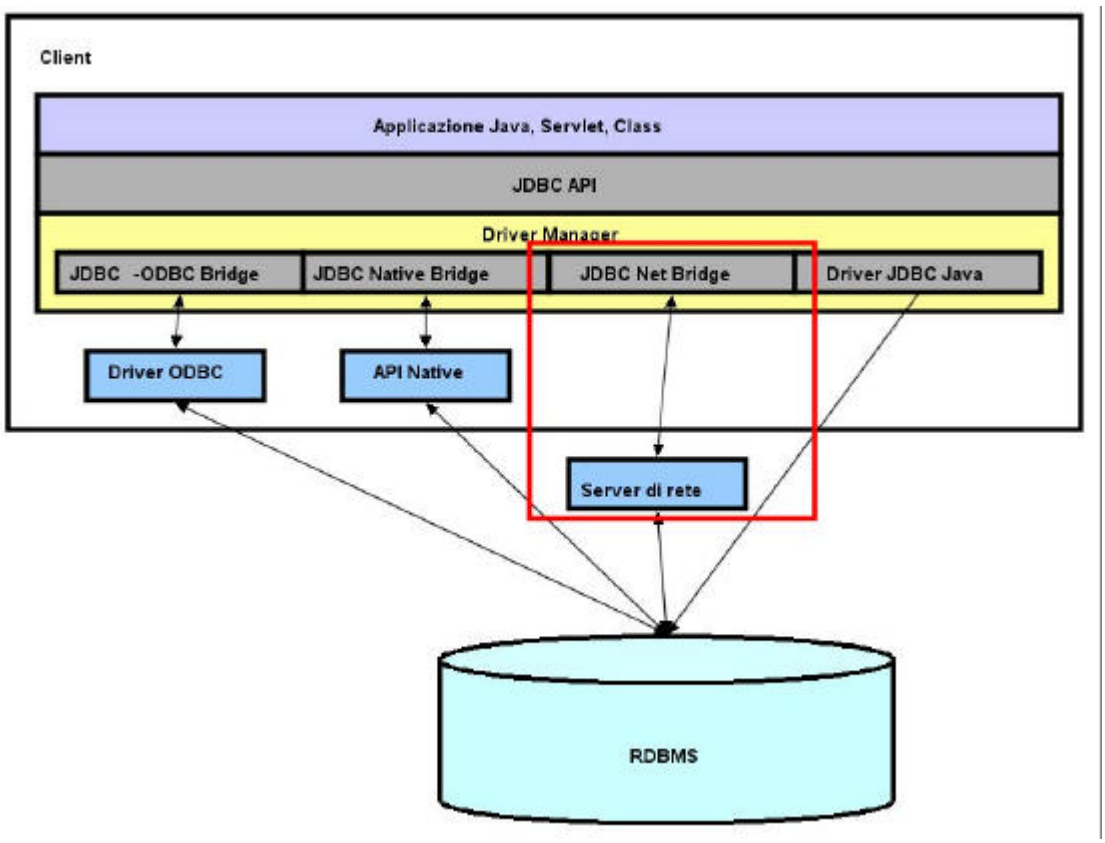


Figura 18-5 *: driver JDBC di tipo 4*

Driver di tipo 4

Un driver di tipo quattro fornisce accesso diretto ad un database ed è un oggetto Java completamente serializzabile (*Immagine 4*).Questo tipo di driver possono funzionare su tutte le piattaforme e, dal momento che sono serializzabili possono essere scaricati dal server.

Una prima applicazione di esempio

Prima di scendere nei dettagli della tecnologia JDBC soffermiamoci su una prima applicazione di esempio. La applicazione usa driver JDBC di tipo 1 per connettersi ad un database access contenente una sola tabella chiamata impiegati come mostrata nella *Figura 18-6*.

IMPIEGATI : Tabella

NOME	COGNOME	ETA	MATRICOLA
Massimiliano	Tarquini	30	0
Giovanni	Grassi	35	1
Giulia	Bongiovanni	40	2
			0

Record: 3 di 3

Figura 18-6 *: tabella access*

EsempioJDBC.java

```
import java.sql.* ;

public class EsempioJDBC
{
    public static void main(String args[])
    {
        try
        {
            Class.forname("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch(ClassNotFoundException e)
        {
            System.out.println(e.toString());
            System.out.println("Il driver non può essere caricato");
            System.exit(1);
        }
        try
        {
            Connection conn =
                DriverManager.getConnection("jdbc:odbc:impiegati","","");
            Statement stmt  = conn.createStatement();
            ResultSet rs =
                stmt.executeQuery("SELECT NOME FROM IMPIEGATI");
            while(rs.next())
            {
                System.out.println(rs.getString("NOME"));
            }
            rs.close();
            stmt.close();
            conn.close();
        }
        catch(SQLException _sql)
        {
            System.out.println(se.getMessage());
            Se.printStackTrace(System.out);
            System.exit(1);
        }
    }
```

}

Dopo aver caricato il driver JDBC utilizzando il metodo statico *forname(String)* della classe *java.lang.Class*

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

l'applicazione tenta la connessione al database utilizzando il metodo statico *getConnection(String,String,String)* dell'oggetto *DriverManager* definito nel package passando come parametro una di tipo Stringa che rappresenta la URL della base dati con una serie di informazioni aggiuntive.

Connection conn = DriverManager.getConnection("jdbc:odbc:impiegati","","")

Nel caso in cui la connessione vada a buon fine la chiamata al metodo di DriverManager ci ritorna un oggetto di tipo *Connection* definito nell'interfaccia *java.sql.Connection*. Mediante l'oggetto Connection creiamo quindi un oggetto di tipo *java.sql.Statement* che rappresenta la definizione uno statement SQL tramite il quale effettueremo la nostra query sul database.

Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT NOME FROM IMPIEGATI");

La nostra applicazione visualizzerà infine il risultato della query mediante un ciclo while che scorre gli elementi di un oggetto ResultSet definito in *java.sql.ResultSet* e ritornato come parametro dalla esecuzione del metodo *executeQuery(String)* di *java.sql.Statement*.

Richiedere una connessione ad un database

Il primo passo da compiere quando utilizziamo un driver JDBC è quello di tentare la connessione al database, sarà quindi necessario che il driver sia caricato all'interno della virtual machine utilizzando il metodo statico forName(String) dell'oggetto Class definito nel package java.lang. Una volta caricato il driver si registrerà sul sistema come driver JDBC disponibile chiamando implicitamente il metodo statico registerDriver(Driver driver) della classe DriverManager.

Questo meccanismo consente di caricare all'interno del gestore dei driver JDBC più di un driver per poi utilizzare quello necessario al momento della connessione che rappresenta il passo successivo da compiere. Per connetterci al database la classe DriverManager ci mette a disposizione il metodo statico getConnection() che prende in input tre stringhe e ritorna un oggetto di tipo java.sql.Connection che da questo momento in poi rappresenterà la connessione ad uno specifico database a seconda del driver JDBC richiamato:

Connection conn =
DriverManager.getConnection(String URL, String user, String password);

Una URL JDBC identifica un database dal punto di vista del driver JDBC. Di fatto URL per driver differenti possono contenere informazioni differenti, tipicamente iniziano con la string "jdbc", contengono indicazioni sul protocollo e informazioni aggiuntive per connettersi al database.

*JDBC URL = jdbc:**protocollo**:**other_info***

Per driver JDBC di tipo 1 (bridge JDBC/ODBC) la JDBC URL diventa

JDBC/ODBC URL = jdbc:odbc:id_odbc

Dove id_odbc è il nome ODBC definito dall'utente ed associato ad una particolare connessione ODBC.

Eseguire query sul database

Dopo aver ottenuto la nostra sezione vorremmo poter eseguire delle query sulla base dati. E' quindi necessario avere un meccanismo che ci consenta di effettuare operazioni di select, delete, update sulle tabelle della base dati.

La via più breve è quella che utilizza l'oggetto *java.sql.Statement* che rappresenta una istruzione SQL da eseguire ed è ritornato dal metodo *createStatement()* dell'oggetto *java.sql.Connection*.

```
Statement stmt = conn.createStatement();
```

I due metodi principali dell'oggetto statement sono I metodi:

```
javax.sql.ResultSet executeQuery(String sql);
int executeUpdate(String sql);
```

Il primo viene utilizzato per tutte quelle query che ritornano valori multipli (select) il secondo per tutte quelle query che non ritornano valori (update o delete). In particolar modo l'esecuzione del primo metodo produce come parametro di ritorno un oggetto di tipo java.sql.ResultSet che rappresenta il risultato della query riga dopo riga. Questo oggetto è legato allo statement che lo ha generato; se l'oggetto statement viene rilasciato mediante il metodo close() anche il ResultSet generato non sarà più utilizzabile e viceversa fino a che un oggetto ResultSet non viene rilasciato mediante il suo metodo close() non sarà possibile utilizzare Statement per inviare al database nuove query.

L'oggetto ResultSet

L'oggetto ResultSet rappresenta il risultato di una query come sequenza di record aventi colonne rappresentati dai nomi definiti all'interno della query. Ad esempio la query "SELECT NOME, COGNOME FROM DIPENDENTI" produrrà un ResultSet contenente due colonne identificate rispettivamente dalla stringa "NOME" e dalla stringa "COGNOME" (*Figura 18-6*). Questi identificativi possono essere utilizzati con i metodi dell'oggetto per recuperare i valori trasportati come risultato della select.

Nel caso in cui la query compaia nella forma "SELECT * FROM DIPENDENTI" le colonne del ResultSet saranno identificato de un numero intero a partire da 1 da sinistra verso destra (*Figura 18-7*).

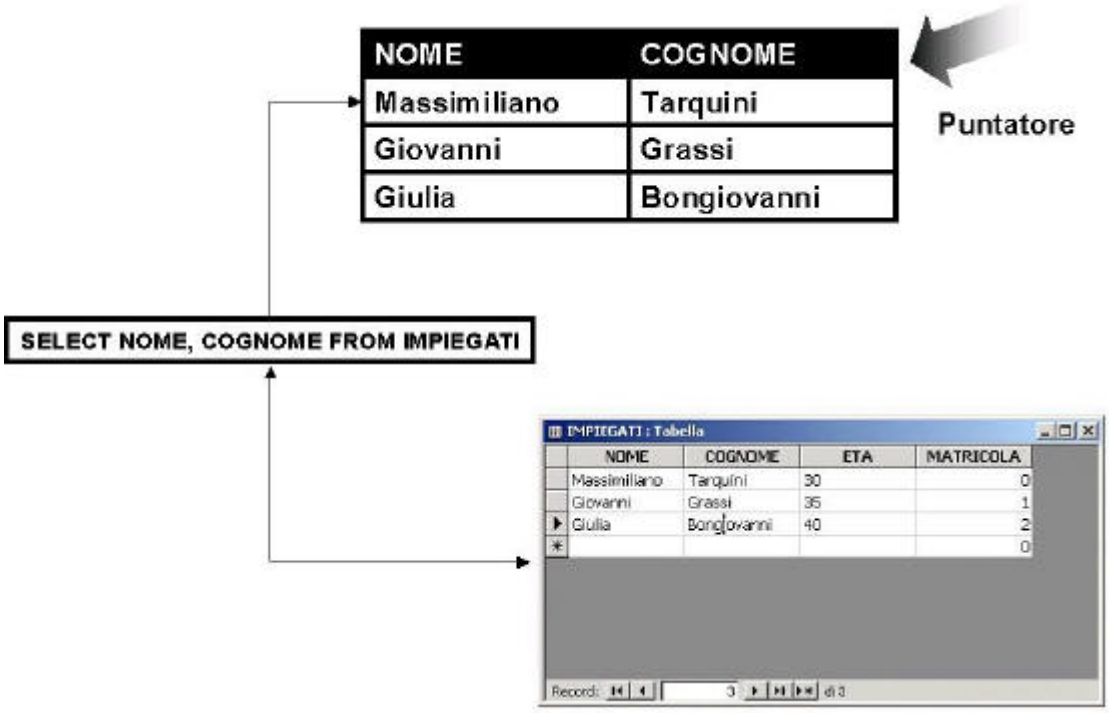


Figura 18-6 : *SELECT NOME, COGNOME ….*

Esempio2JDBC.java

```
import java.sql.* ;

public class Esempio2JDBC
{
    public static void main(String args[])
    {
        try
        {
            Class.forname("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch(ClassNotFoundException e)
        {
            System.out.println(e.toString());
            System.out.println("Il driver non può essere caricato");
            System.exit(1);
        }
        try
        {
            Connection conn =
                DriverManager.getConnection("jdbc:odbc:impiegati","","");
            Statement stmt  = conn.createStatement();
            ResultSet rs =
                stmt.executeQuery("SELECT NOME, COGNOME FROM
                IMPIEGATI");
            while(rs.next())
            {
                System.out.println(rs.getString("NOME"));
                System.out.println(rs.getString("COGNOME"));
```

```
                }
                rs.close();
                stmt.close();
                conn.close();
        }
        catch(SQLException _sql)
        {
                System.out.println(se.getMessage());
                Se.printStackTrace(System.out);
                System.exit(1);
        }
    }
}
```

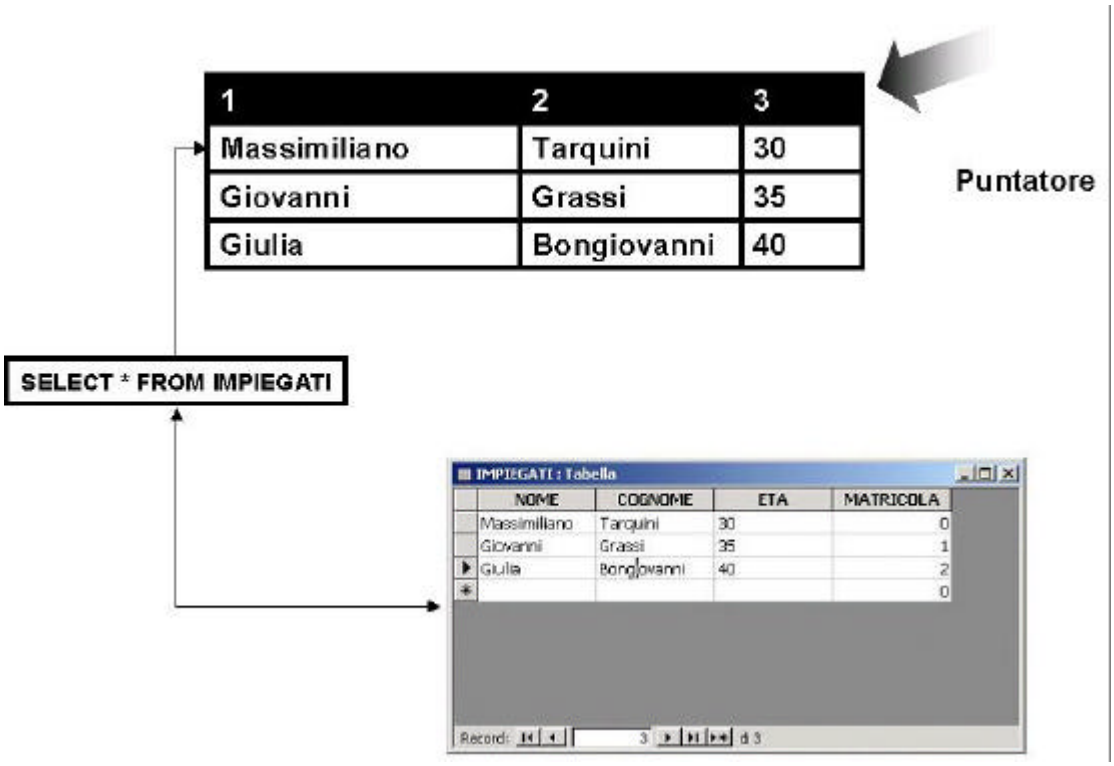


Figura 18-76 : *SELECT * FROM ….*

Esempio3JDBC.java

```
import java.sql.* ;

public class Esempio3JDBC
{
    public static void main(String args[])
    {
        try
        {
            Class.forname("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch(ClassNotFoundException e)
        {
            System.out.println(e.toString());
```

```
                System.out.println("Il driver non può essere caricato");
                System.exit(1);
        }
        try
        {
                Connection conn =
                    DriverManager.getConnection("jdbc:odbc:impiegati","","");
                Statement stmt  = conn.createStatement();
                ResultSet rs =
                    stmt.executeQuery("SELECT * FROM IMPIEGATI");
                while(rs.next())
                {
                        System.out.println(rs.getString(1));
                        System.out.println(rs.getString(2));
                        System.out.println(rs.getInt(3));
                }
                rs.close();
                stmt.close();
                conn.close();
        }
        catch(SQLException _sql)
        {
                System.out.println(se.getMessage());
                Se.printStackTrace(System.out);
                System.exit(1);
        }
    }
}
```

Appendice A
Java Time-line

1995-1996

23 Maggio 1995
Lancio della tecnologia Java

23 Gennaio 1996
Giorno del rilascio del Java Development Kit 1.0

30 Aprile 1996
I 10 maggiori vendors di software annunciano la loro intenzione di includere la tecnologia Java all'interno dei loro prodotti

29 Maggio 1996
Prima versione di JavaOne, convegno dedicato ai programmatori Java. Annunciata la tecnologia JavaBeans basata su componenti, le librerie multimediali, Java Servlet ed altre tecnologie.

10 Giugno 1996
50.000 programmatori in attesa del "Sun Java Day" in Tokio.

16 Agosto 1996
Sun Microsystem e Addison-Wesley pubblicano i "Java Tutorial" e la prima versione del "Java Language Specification"

Settembre 1996
83.000 pagine web utilizzano tecnologia Java.

Settembre 1996
Lancio del sito "Java Developer Connection".

16 Ottobre 1996
Completate le specifiche JavaBeans.

25 Ottobre 1996
La Sun Microsystem annuncia il primo compilatore JIT (Just In Time) per la piattaforma Java.

29 Ottobre 1996
Annuncio delle API JavaCard.

9 Dicembre 1996
Rilasciata la versione beta del JDK 1.1

11 Dicembre 1996
Lancio della prima iniziativa 100% Java a cui aderiscono 100 compagnie.

11 Dicembre 1996
SUN IBM e Netscape iniziano il "Java Education World Tour" in oltre 40 città.

1997

11 Gennaio 1997
SUN rilascia il JavaBeans development kit

18 Febbraio 1997
SUN rilascia il JDK 1.1

28 Febbraio 1997
Netscape annuncia che Communicator supporterà tutte le Api e le applicazioni Java.

4 Marzo 1997
Rilasciata la versione beta di Java Web Server e il Java Servlet Development Kit.

10 Marzo 1997
Introduzione della API Java "JNDI" : Java Naming and Directory Interface.

11 Marzo 1997
Raggiunti più di 220.000 download del JDK 1.1 in tre settimane.

2 Aprile 1997
JavaOne diventa la più grande conferenza per sviluppatori del mondo con più di 10.000 persone in attesa dell'evento.

2 Aprile 1997
SUN annuncia la tecnologia Enterprise JavaBeans.

2 Aprile 1997
SUN annuncia che la tecnologia Java Foundation Classes sarà inclusa nelle revisioni successive della piattaforma Java.

6 Maggio 1997
"Glasgow", software per JavaBeans viene rilasciato in visione al pubblico.

5 Giugno 1997
Java Web Server 1.0 viene rilasciato nella sua versione definitiva.

23 Luglio 1997
Annunciate le Java Accessibility API.

23 Luglio 1997
Rilasciata la piattaforma Java Card 2.0

5 Agosto 1997
Rilasciate le API Java Media e Communication.

5 Agosto 1997
Raggiunti oltre 100.000 download del JavaBeans Development Kit.

12 Agosto 1997
Più di 600 applicazioni commerciali basate sulla tecnologia Java.

23 Settembre 1997
Java Developers Connection supera i 100.000 utenti.

1998

20 Gennaio 1998
2 milioni di download del JDK 1.1

Marzo 1998
Rilascio del progetto Swing.

24 Marzo 1998
JavaOne attesa da oltre 15.000 programmatori in tutto il mondo.

24 Marzo 1998
Annunciato il prodotto Java Junpstart

31 Marzo 1998
Ericcson, Sony, Siemens, BEA, Open TV ed altri vendors licenziano la tecnologia Java.

31 Marzo 1998

SUN annuncia il porting della tecnologia Java su piattaforma WinCE.

20 Aprile 1998
Rilascio dei prodotti Java Plug-In.

3 Giugno 1998
Visa rilascia la prima smart card basata su Visa Open Platform e Java Card Technology.

16 Settembre 1998
Motorola annuncia l'introduzione della tecnologia Java.

21 Ottobre 1998
Oltre 500.000 download del software JFC/Swing.

5 Novembre 1998
Sun inizia a lavorare con Linux Community al porting della tecnologia Java sotto piattaforma Linux.

5 Novembre 1998
Completate le specifiche di EmbeddedJava.

8 Dicembre 1998
Rilascio della piattaforma Java 2.

8 Dicembre 1998
Formalizzazione del programma JCP : Java Community Process.

1999

13 Gennaio 1999
La tecnologia Java viene supportata perla produzione di televisioni digitali.

25 Gennaio 1999
Annunciata la tecnologia Jini.

1 Febbraio 1999
Rilascio della piattaforma PersonalJava 3.0.

24 Febbraio 1999
Rilascio dei codici sorgenti della piattaforma Java 2.

Febbraio 1999
Rilascio delle specifiche Java Card 2.1

4 Marzo 1999
Annunciato il supporto XML per Java.

27 Marzo 1999
Rilascio del motore Java HotSpot.

2 Giugno 1999
Rilascio della tecnologia JavaServer Pages.

15 Giugno 1999
SUN annuncia tre edizioni della piattaforma Java: J2SE, J2EE, J2ME.

29 Giugno 1999
Rilascio della "Reference Implementation" di J2EE in versione Alfa.

30 Settembre 1999
Rilascio del software J2EE in beta version.

22 Novembre 1999
Rilascio della piattaforma J2EE.

22 Novembre 1999
Rilascio della piattaforma J2SE per Linux.

2000

8 Febbraio 2000
Proposta alla comunità la futura versione di J2EE e J2SE.

29 Febbraio 2000
Rilascio delle API per XML.

Maggio 2000
Superato il 1.500.000 di utenti su Java Developer Connection.

8 Maggio 2000
Rilascio della piattaforma J2SE 1.3

26 Maggio 2000
Oltre 400 Java User Group in tutto il mondo.

Appendice B
Glossario dei termini

AGE - "Età" parlando di protocollo HTTP 1.1 indica la data di invio di una Richiesta da parte del client verso il server.
ANSI - "American National Standard Institute" stabilisce e diffonde le specifiche su sistemi e standard di vario tipo. L'istituto ANSI è attualmente formato da più di 1000 aziende pubbliche e private.
Applet - Componenti Java caricate da un browser come parte di una pagina HTML.
ARPANET - le prime quattro lettere significano: "Advanced Research Projet Agency", agenzia del Ministero della difesa U.S.A. nata nel 1969 e progenitrice dell'attuale Internet.
ADSL - Asimmetrical Digital Subscriber Line, nuova tecnologia di trasmissione dati per il collegamento ad internet. E' possibile raggiungere velocità di circa 6Mb al secondo per ricevere e 640Kb al secondo per inviare dati.
B2B - Con la sigla B2B si identificano tutte quelle iniziative tese ad integrare le attività commerciali di un'azienda con quella dei propri clienti o dei propri fornitori, dove però il cliente non sia anche il consumatore finale del bene o del servizio venduti ma sia un partner attraverso il quale si raggiungono, appunto, i consumatori finali.
B2C - B2C è l'acronimo di Business to Consumer e, contrariamente a quanto detto per il B2B, questo identifica tutte quelle iniziative tese a raggiungere il consumatore finale dei beni o dei servizi venduti.
B2E - B2E è l'acronimo di Business to Employee e riguarda un settore particolare delle attività commerciali di un'azienda, quelle rivolte alla vendita di beni ai dipendenti.
Backbone - definizione attribuita ad uno o più nodi vitali nella distribuzione e nello smistamento del traffico telematico in Internet.
Cache - "Contenitore" gestito localmente da una applicazione con lo scopo di mantenere copie di entità o documenti da utilizzare su richiesta. Generalmente il termine è largamente usato in ambito internet per indicare la capacità di un browser di mantenere copia di documenti scaricati in precedenza senza doverli scaricare nuovamente riducendo così i tempi di connessione con il server.
CGI - (Common Gateway Interface), si tratta di programmi sviluppati in linguaggio C++ o simile, per consentire alle pagine web di diventare interattive. Viene usato soprattutto in funzioni come l'interazione con database e nei motori di ricerca.
Chat - "chiacchiera" in inglese; permette il dialogo tra due o più persone in tempo reale, raggruppate o non in appositi canali, comunicando attraverso diversi protocolli tra cui il più diffuso é IRC (Internet Relay Chat).
Client - Applicazione che stabilisce una connessione con un server allo scopo di trasmettere dati.
Connessione - Canale logico stabilito tra due applicazioni allo scopo di comunicare tra loro.
CORBA - Acronimo di Common Object Request Broker identifica la architettura proposta da Object Management Group per la comunicazione tra oggetti distribuiti.
Datagram - Pacchetto di dati trasmesso via rete mediante protocolli di tipo "Connectionless"**.**
DNS - Acronimo di "Domain Name System" è un programma eseguito all'interno di un computer Host e si occupa della traduzione di indirizzi IP in nomi o viceversa.

Dominio - Sistema attualmente in vigore per la suddivisione e gerarchizzazione delle reti in Internet.
eCommerce - quel "complesso di attività che permettono ad un'impresa di condurre affari on line", di qualunque genere di affari si tratti. Se, invece di porre l'accento sull'attività svolta, si vuole porre l'accento sul destinatario dei beni o dei servizi offerti, allora la definizione può diventare più specifica, come nelle voci B2B, B2C e B2E .
EJB - Enterprise Java Beans rappresentano nella architettura J2EE lo standard per lo sviluppo di Oggetti di Business distribuiti.
Email - o posta elettronica è il sistema per l'invio di messagistica tra utenti collegati alla rete.
Entità - Informazione trasferita da un client ad un server mediante protocollo HTTP come prodotto di una richiesta o come informazione trasferita da client a server.
Una entità è formata da meta-informazioni nella forma di entity-header e contenuti nella forma di entity-body come descritto nel capitolo 2.
Extranet - rete basata su tecnologia internet, estende una Intranet al di fuori della azienda proprietaria.
FAQ - "Frequently Asked Questions" ovvero " domande più frequenti".
FTP - "File Transfer Potocol" protocollo di comunicazione precedente all' HTTP, permette il trasferimento di file tra due computer.
Gateway - periferica per collegare computer diversi in una rete. Possiede un proprio microprocessore e una propria memoria di elaborazione per gestire conversioni di protocolli di comunicazione diversi.
HDML - Handless Device Markup Language Il linguaggio per sistemi Wireless, attraverso il quale creare pagine Web navigabili dai telefoni cellulari dotati di tecnologia WAP.
Host - uno qualsiasi dei computer raggiungibili in rete. Nella architettura TCP/IP è sinonimo di End-System.
Hosting - Servizio che ospita più siti Web su una singola macchina, assegnando a ognuno di essi un IP differente. In altri termini con il servizio di hosting il sito condivide hard disk e banda disponibile con altri Website ospitati.
HTML - "Hyper Text Markup Language" (Linguaggio di Marcatura per il Testo) è il linguaggio per scrivere pagine web. Standardizzato dal W3C deriva da SGML ed è composto da elementi di contrassegno, tag e attributi.
HTTP - "Hyper Text Trasfer Protocol" protocollo di comunicazione ampiamente diffuso nel Web. HTTP utilizza TCP/IP per il trasferimento di file su Internet.
Hub - o concentratore è periferica per collegare e smistare i cavi ai computer di una rete locale, utilizzati una volta solo dalle grandi e medie aziende, oggi pare che stiano facendo la loro comparsa anche nelle piccole aziende e persino nel mondo domestico.
Hyperlink - collegamento tra parti di una stessa pagina o di documenti presenti sul Web.
IDSL -il termine indica la possibilità di fornire la tecnologia DSL a linee ISDN già esistenti. Anche se la quantità di dati trasferiti (Transfer rate) é più o meno la stessa dell'ISDN (144 Kbps contro i 128 Kbps) e l'ISDL può trasmettere solamente dati (e non anche la voce), i maggiori benefici consistono: nel poter usufruire di connessioni sempre attive, eliminando quindi i ritardi dovuti alla configurazione della chiamata, nel pagamento con tariffe che vengono definite Flat rate (cioè forfetarie, un tanto al mese, quindi non in base agli scatti effettivi) e nella trasmissione di dati su una linea dedicata solo ad essi, piuttosto che sulla normale linea telefonica.
IMAP - "Internet Message Access Protocol", protocollo anch'esso utilizzato per la ricezione delle email. L'ultima versione (IMAP4) é simile al POP3, ma supporta alcune funzioni in più. Per esempio: con IMAP4 é possibile effettuare

una ricerca per Keyword tra i diversi messaggi, quando ancora questi si trovano sul Server di email; in base all'esito della ricerca é quindi possibile scegliere quali messaggi scaricare e quali no. Come POP, anche IMAP utilizza SMTP per la comunicazioni tra il client di email ed il server.
Intranet - rete di computer (LAN) per comunicare all'interno di una medesima azienda; si basa sullo stesso protocollo di Internet (TCP/IP): utilizza i medesimi sistemi di comunicazione, di rappresentazione (pagine web) e di gestione delle informazioni. Una serie di software di protezione evita che le informazioni siano accessibili al di fuori di tale rete.
IP - é un indirizzo formato da quattro gruppi di numeri che vanno da 0,0,0,0 a 255,255,255,255; esso indica in maniera univoca un determinato computer connesso ad Internet o in alcuni casi gruppi di computer all'interno di una rete.
ISDN - "Integrated Services Digital Network", rete di linee telefoniche digitali per la trasmissione di dati ad alta velocità, che si aggira sui 64KBps; attualmente é possibile utilizzare due linee in contemporanea e raggiungere i 128KBps.
J2EE - Java 2 Enterprise Edition.
JAF - servizio necessario all'instanziamento di una componente JavaBeans.
JDBC - Java DataBase Connectivity.
JMS - (Java Message Service): API Java di Sun che fornisce una piattaforma comune per lo scambio di messaggi fra computer collegati nello stesso network, utilizzando uno dei tanti sistemi diversi di messagistica, come MQSeries, SonicMQ od altri. E' inoltre possibile utilizzare Java e XML.
JNDI - Java Naming & Directory Interface.
LAN - Acronimo di Local Area Network, una rete che connette due o più computer all'interno di piccola un'area.
LINK - alla base degli ipertesti, si tratta di un collegamento sotto forma di immagine o testo a un'altra pagina o file
MAIL SERVER - Computer che fornisce i servizi di Posta Elettronica.
MAN - Acronimo di Metropolitan Area Network.
Messaggio - Unità base della comunicazione con protocollo HTTP costituita da una sequenza di ottetti legati tra loro secondo lo standard come definito nel capitolo 2.
NBS - Acronimo di National Bureau of Standards ribattezzato recentemente in NIST.
NETIQUETTE - Galateo della rete**.**
network number - Porzione dell'indirizzo IP indicante la rete. Per le reti di classe A, l'indirizzo di rete è il primo byte dell'indirizzo IP; per le reti di classe B, sono i primi due byte dell'indirizzo IP; per quelle di classe C, sono i primi 3 byte dell'indirizzo IP. In tutti e tre i casi, il resto è l'indirizzo dell'host. In Internet, gli indirizzi di rete assegnati sono unici a livello globale.
NFS Network File System - Protocollo sviluppato da Sun Microsystems e definito in RFC 1094, che consente a un sistema di elaborazione di accedere ai file dispersi in una rete come se fossero sull'unità a disco locale.
NIC - Network Information Center : organismo che fornisce informazioni, assistenza e servizi agli utenti di una rete.
Pacchetto - E' il termine generico utilizzato per indicare le unità di dati a tutti i livelli di un protocollo, ma l'impiego corretto è nella indicazione delle unità di dati delle applicazioni.
Porta - Meccanismo di identificazione di un processo su un computer nei confronti del TCP/IP o punto di ingresso/uscita su internet.
PPP Point-to-Point Protocol - Il Point-to-Point Protocol, definito in RFC 1548, fornisce il metodo per la trasmissione di pacchetti nei collegamenti di tipo seriale da-punto-a-punto.
Protocollo - Descrizione formale del formato dei messaggi e delle regole che due elaboratori devono adottare per lo scambio di messaggi. I protocolli possono

descrivere i particolari low-level delle interfaccia macchina-macchina (cioè l'ordine secondo il quale i bit e i bytes vengono trasmessi su una linea) oppure gli scambi high level tra i programmi di allocazione (cioè il modo in cui due programmi trasmettono un file su Internet).
RA - Registration Authority italiana. Autorità con competenze per l'assegnazione di domini con suffisso .it.
RFC - Request for Comments. richiesta di osservazioni. Serie di documenti iniziata nel 1969 che descrive la famiglia di protocolli Internet e relativi esperimenti. Non tutte le RFC (anzi, molto poche, in realtà) descrivono gli standard Internet, ma tutti gli standard Internet vengono diffusi sotto forma di RFC.
RIP - Routing Information Protocol.
Router - Dispositivo che serve all'inoltro del traffico tra le reti. La decisione di inoltro è basata su informazioni relative allo stato della rete e sulle tabelle di instradamento (routing tables).
RPC - Remote Procedure Call: Paradigma facile e diffuso per la realizzazione del modello di elaborazione distribuita client-server. In generale, a un sistema remoto viene inviata una richiesta di svolgere una certa procedura, utilizzando argomenti forniti, restituendo poi il risultato al chiamante. Varianti e sottigliezze distinguono le varie realizzazioni, il che ha dato origine a una varietà di protocolli RPC tra loro incompatibili.
SERVER - Computer o software che fornisce servizi o informazioni ad utenti o computer che si collegano.
SMTP - Acronimo di "Simple Mail Transfer Protocol" è il Protocollo standard di Internet per l'invio e la ricezione della posta elettronica tra computer.
TCP/IP - standard sviluppato per le comunicazioni tra calcolatori. E' diventato il protocollo più usato per la trasmissione dei dati in Internet.

Bibliografia

✍✍David Flanagan, Jim Farley, William Crawford, Kris Magnusson - "***Java Enterprise in a nutshell***", O'Reilly, 1999;

✍✍Alan Radding - " ***Java 2 Enterprise Edition has turned the language into a total app-development environment*** ", INFORMATION WEEK On Line, 22 Maggio 2000 Sun Microsystem -"J2EE Blueprints", Sun Microsystem, 1999;

✍✍Danny Ayers, Hans Bergsten, Michael Bogovich, Jason Diamond, Mattew Ferris, Marc Fleury, Ari Halberstadt, Paul Houle, piroz Mohseni, Andrew Patzer, Ron Philips, Sing Li, Krishna Vedati, Mark Wilcox and Stefan Zeiger – "***Professional Java Server Programming***", Wrox Press, 1999;

✍✍Ruggero Adinolfi - "***Reti di Computer***", McGraw-Hill, 1999;

✍✍James Martin - "LAN – Architetture e implementazioni delle reti locali", Jackson Libri, 1999;