SCRITTO DA:

MASSIMILIANO TARQUINI
ALESSANDRO LIGI

4IT
www.4it.it

Versione 0.1.8
Rilascio del 02/05/2001

Titolo Originale
Java Enterprise Computing

Coordinamento Editoriale
Massimiliano Tarquini

Progetto Editoriale
Massimiliano Tarquini

Progetto Grafico
Riccardo Agea

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

Albert Einstein

INDICE ANALITICO

INTRODUZIONE

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 dello scorso secolo con la introduzione di tecnologie legate ad Internet e più in generale alle reti, l'industria del software ha immesso sul mercato circa trenta 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 conseguente aumento della vita del software, 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 componenti 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 transazionali, 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.

Obiettivi del libro

Questo libro, ricco di schemi e di esempi che facilitano la lettura e la comprensione, offre una panoramica delle metodologie Object Oriented e a partire dal linguaggio Java, mostra al lettore le chiavi per aiutarlo a comprendere i vantaggi e gli inconvenienti delle diverse soluzioni offerte dalla piattaforma J2EE.
Nella prima parte, il libro introduce al linguaggio Java come strumento per programmare ad oggetti partendo dal paradigma Object Oriented e identificando di volta in volta le caratteristiche del linguaggio che meglio si adattano nella'pplicazione dei principi di base della programmazione Object Oriented.
La seconda parte del libro, offre una panoramica delle tecnologie legate alla piattaforma J2EE, cercando di fornire al programmatore gli elementi per identificare le soluzioni più idonee alle proprie esigenze.
Infine, la trza parte del libro è dedicata alle tecniche di programmazione mediate l'uso dei design pattern, introducendo i ventitré pattern noti come *GoF (Gang of Four)*.
Un cdrom ricco di documentazione, strumenti per il disegno e lo sviluppo di applicazioni Java e contenente tutti gli esempi riportatie nel libro, completa il panorama didattico offerto da Java Mattone dopo Mattone.

Lasciamo quindi a voi lettori il giudizio finale e augurandovi buona lettura, ci auguriamo che il nostro lavoro vi risulti utile.

Massimiliano Tarquini
Alessandro Ligi

Parte Prima

Programmazione Object Oriented

1 INTRODUZIONE ALLA PROGRAMMAZIONE OBJECT ORIENTED

1.1 Introduzione

Questo capitolo è dedicato al "paradigma Object Oriented" ed ha lo scopo di fornire ai neofiti della programmazione in Java i concetti di base necessari allo sviluppo di applicazioni orientate ad oggetti.
In realtà, le problematiche che si andranno ad affrontare nei prossimi paragrafi sono molto 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, sarà chiara l'evoluzione che, 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 è 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 è suddivisa in un insieme di oggetti in grado di interagire tra loro, e codificati in modo tale che la macchina sia in gradi di comprenderli.

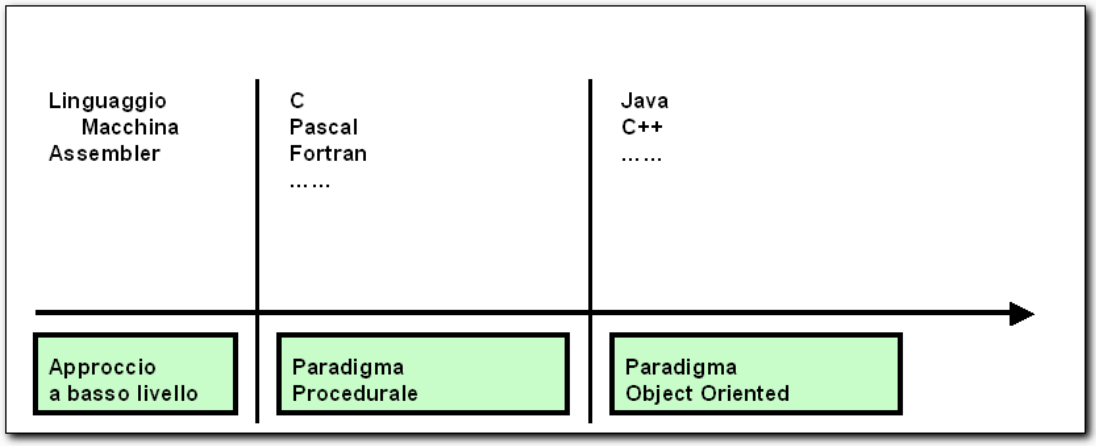


Figura 1: Evoluzione del modello di programmazione

Il primo cambiamento evidente è quindi a livello di disegno dell'applicazione. Di fatto, l'approccio Object Oriented non è limitato a linguaggi come Java o C++. Molte applicazioni 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.

1.2 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 perché 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 scomposizione procedurale dell'applicazione.
Con l'aumento delle prestazioni dei calcolatori e di conseguenza con l'aumento della complessità delle applicazioni, l'approccio procedurale ha iniziato a mostrare i propri limiti evidenziando la necessità di definire un nuovo modello e nuovi linguaggi di programmazione. Questa evoluzione è stata schematizzata nella *figura 1*.
I linguaggi come Java e C++ forniscono il supporto ideale allo sviluppo di applicazioni per componenti, fornendo un insieme di regole, sintattiche e semantiche, che aiutano nello sviluppo di oggetti.

1.3 Il paradigma procedurale

Secondo il paradigma procedurale, il programmatore analizza il problema ponendosi dal punto di vista del computer che esegue solamente istruzioni semplici e di conseguenza adotta un approccio di tipo "divide et impera[1]". Il programmatore sa perfettamente che un'applicazione per quanto complessa può essere suddivisa in procedure e funzioni di piccola entità. Quest'approccio è stato formalizzato in molti modi ed è ben supportato da un gran numero di linguaggi che forniscono al programmatore un ambiente in cui siano facilmente definibili procedure e funzioni.
Le procedure e le funzioni sono blocchi di codice riutilizzabile che possiedono un proprio insieme di dati e realizzano specifiche funzioni. Le funzioni definite possono essere richiamate ripetutamente in un programma, possono ricevere parametri che modificano il loro stato e possono tornare valori al codice chiamante. Procedure e funzioni possono essere legate assieme a formare un'applicazione, ed è quindi necessario che, all'interno di una applicazione, i dati siano condivisi tra loro.
Questo meccanismo si risolve mediante l'uso di variabili globali, passaggio di parametri e ritorno di valori.

[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.

Un'applicazione procedurale tipica, ed il suo diagramma di flusso, è riassunta nella *Figura 2.*
L'uso di variabili globali genera però problemi di protezione dei dati quando le procedure si richiamano tra loro. Per esempio nell'applicazione mostrata nella *Figura 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.
A causa del fatto che le procedure non sono "auto-documentanti" (self-documenting), ossia non rappresentano entità ben definite, un programmatore dovendo modificare l'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.

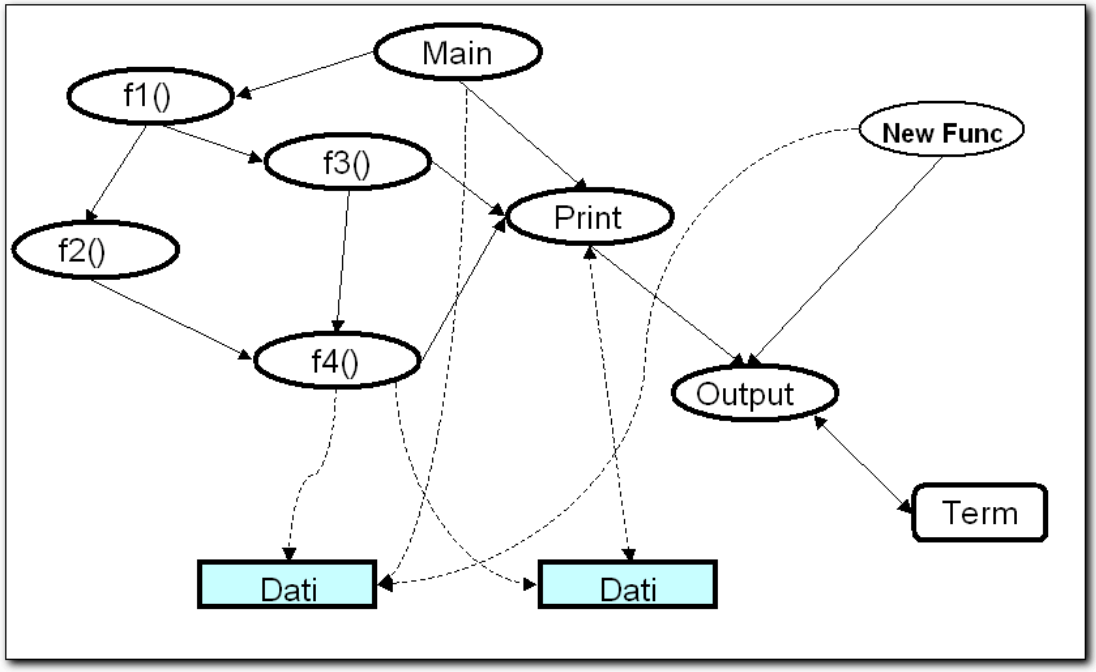


Figura 2: Diagramma di una applicazione procedurale

Per questo motivo, le applicazioni basate sul modello procedurale sono difficili da aggiornare e controllare con meccanismi di debug. Gli errori derivanti da effetti indesiderati possono presentarsi in qualunque punto del codice causando la 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.

1.4 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 all'interno dell'applicazione in grado di accedere a dati globali, eseguire funzioni di basso livello o, evitare direttamente l'uso di variabili globali: se applicato, il modificatore "static" ad una variabile locale, il calcolatore alloca 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 l'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, viene inizializzata 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 un programma 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 loro, le cui parti sensibili possono essere isolate in modo da poter essere

invocate 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 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.

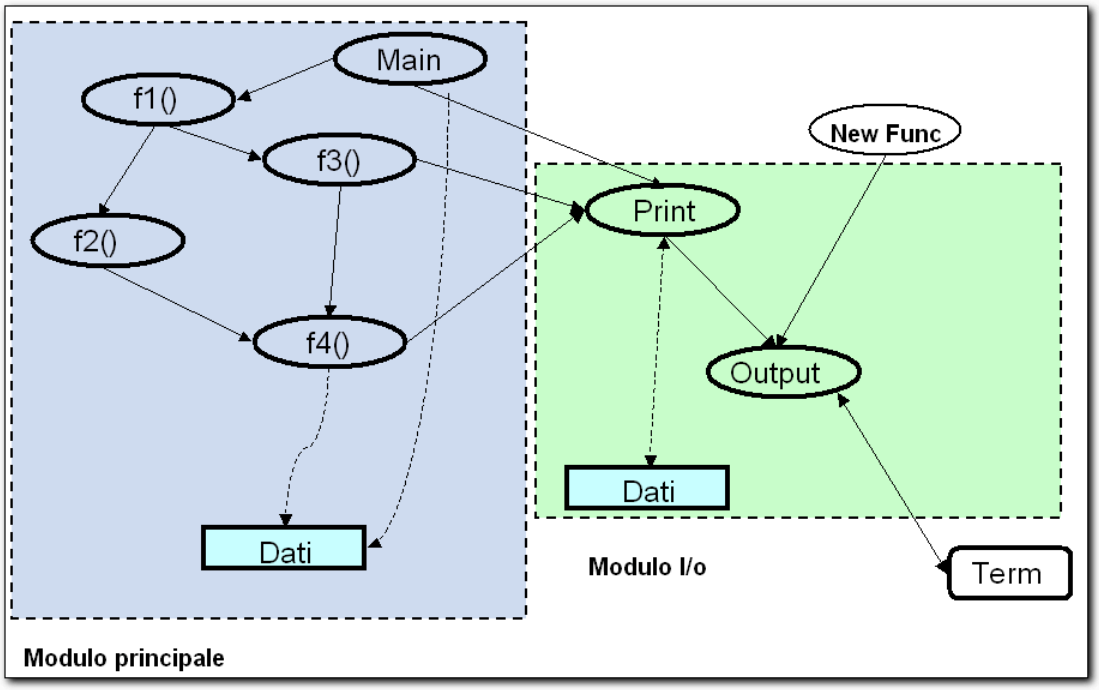


Figura 3: Diagramma di una applicazione procedurale suddiviso per moduli

Quest'evoluzione del modello fornisce numerosi vantaggi; i dati ora non sono completamente globali e sono quindi più protetti che nel modello precedente, limitando di conseguenza i danni causati da propagazioni anomale degli errori. Inoltre il numero limitato di procedure "pubbliche" viene in aiuto ad un programmatore che inizi a studiare il codice dell'applicazione. Questo nuovo modello si avvicina molto a quello proposto dall'approccio Object Oriented.

1.5 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 sia applicativo; i pezzi sono poi riuniti a formare un'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 regole 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). Un'applicazione può essere scritta avendo solo poche informazioni ed in particolare, il funzionamento interno delle funzionalità di ogni oggetto è completamente nascosto al programmatore (Incapsulamento Object Oriented).

1.6 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 all'oggetto che state utilizzando: un libro.
Pensando ad un libro è naturale richiamare alla mente un insieme di oggetti aventi caratteristiche in comune: tutti i libri contengono delle pagine, ogni pagina contiene del testo e le note sono scritte sul fondo. 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 di 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 sfogliarne i contenuti (anche se privi di senso), guardarne le illustrazioni 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.
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 di partenza.

1.7 Ereditarietà

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

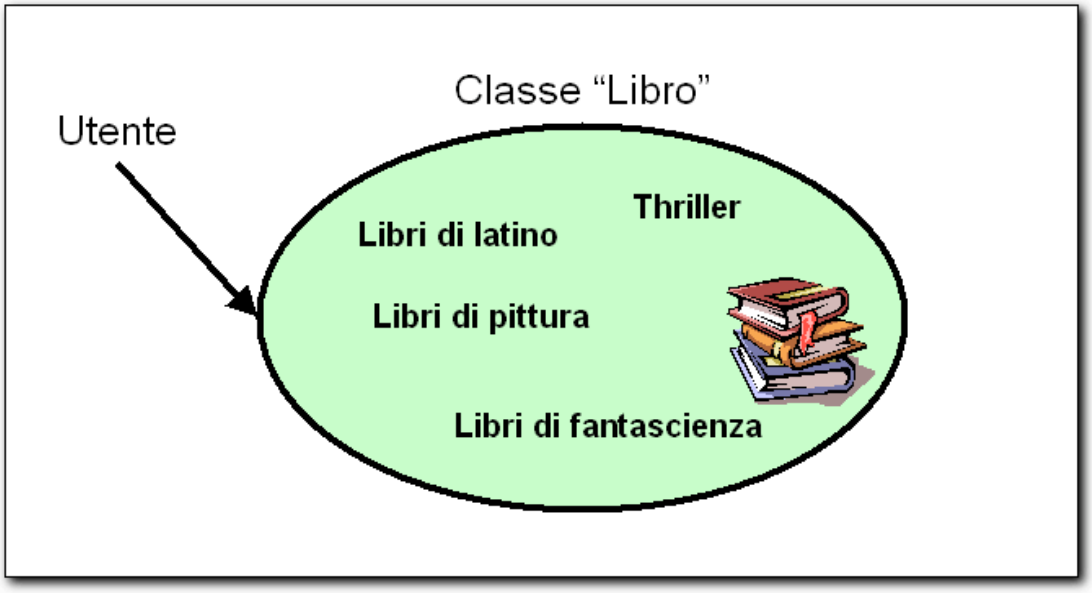


Figura 4: La classe generica "Libro" e le sue specializzazioni

1.8 Il concetto di ereditarietà nella programmazione

Estendendo 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 di base.

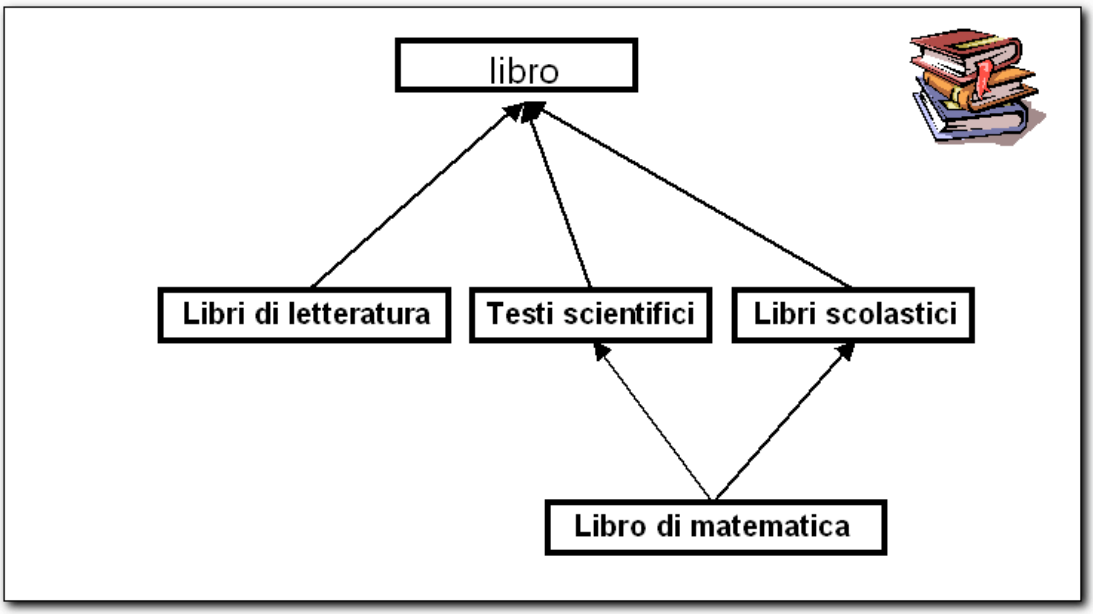


Figura 5: Diagramma di ereditarietà

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

1.9 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 un'applicazione: ogni volta che si renda necessaria una modifica, è in genere sufficiente crearne un nuovo all'interno di una classe di oggetti ed utilizzarlo per rimpiazzarne uno vecchio ed obsoleto.
Un altro vantaggio dell'ereditarietà è la capacità di fornire codice riutilizzabile. 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 inclusa all'interno della descrizione.
In Java ad esempio ogni volta che definiamo un concetto, esso è definito come una classe all'interno della quale è scritto il codice necessario ad implementare le funzionalità dell'oggetto per quanto generico esso sia. Se un nuovo oggetto è creato per mezzo del meccanismo della ereditarietà da un oggetto esistente, si dice che la nuova entità "deriva" dalla originale.
Quando questo accade, tutte le caratteristiche dell'oggetto principale diventano parte della nuova classe. Poiché l'oggetto derivato eredita le funzionalità del suo predecessore, l'ammontare del codice da implementare è notevolmente ridotto. Il codice dell'oggetto d'origine è stato riutilizzato.
A questo punto è necessario iniziare a definire formalmente alcuni termini. La relazione di ereditarietà tra oggetti 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 perché una classe derivata può a sua volta essere una classe base per una più specifica.

1.10 Programmazione Object Oriented ed incapsulamento

Come già ampiamente discusso, nella programmazione "Object Oriented" 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 devono essere nascosti affinché sia possibile mantenere l'oggetto in uno stato consistente.
Ad esempio, se un oggetto rappresenta uno "stack[2] di dati" (*figura 6*), l'applicazione dovrà poter accedere solo al primo dato dello stack od inserirne uno nuovo sulla cima, ossia alle funzioni di Push e Pop.

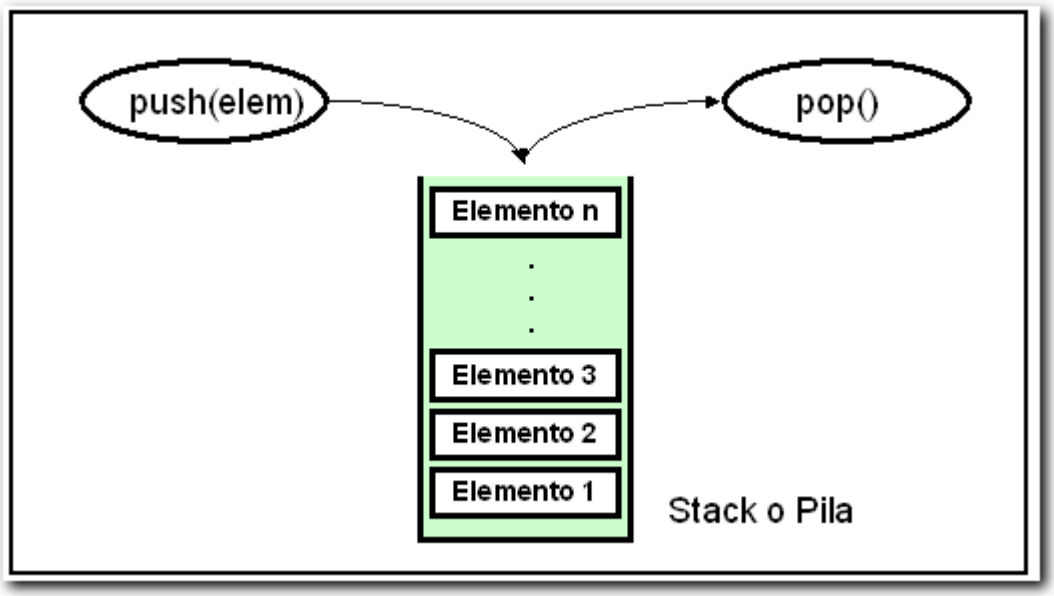


Figura 6: Stack di dati

Il contenitore, ed ogni altra funzionalità necessaria alla sua gestione, dovranno essere protetti rispetto all'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.

1.11 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

[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.

causare rotture nelle dipendenze tra gli oggetti poiché, non saranno in grado di vedere i legami tra dati ed entità. L'effetto principale sull'applicazione sarà quindi quello di localizzare gli errori evitandone la propagazione e dotando l'applicazione di grande stabilità: per sua natura, un oggetto incapsulato non produce effetti secondari.
In un programma scomposto per blocchi, le procedure e le funzioni tendono ad essere interdipendenti. Ogni variazione al programma, richiede spesso la modifica di funzioni condivise cosa che può propagare un errore ad altri membri del programma che le utilizza.
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 l'aggiunta di nuovi oggetti, ed il meccanismo di ereditarietà ha l'effetto di preservare l'integrità dei membri dell'applicazione.
Se invece fosse necessario modificare internamente un oggetto le modifiche sarebbero, in ogni caso, limitate al corpo dell'entità e quindi confinate all'interno dell'oggetto impedendo la propagazione di errori all'esterno del codice.
Anche le operazioni di ottimizzazione sono 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.

1.12 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 fono etc.. Il modello che ne deriverebbe sarebbe inconsistente e confuso.

1.13 Storia del paradigma Object Oriented

L'idea di modellare applicazioni con oggetti piuttosto che funzioni, comparve per la prima volta nel 1967 con il rilascio del linguaggio "Simula", prodotto in Norvegia da Ole-Johan Dahl e Kristen Nygaard. Simula, linguaggio largamente utilizzato per sistemi di simulazione, adottava i concetti di classe ed ereditarietà.
Il termine "Object Oriented" fu utilizzato per la prima volta solo qualche anno dopo, quando la Xerox lanciò sul mercato il linguaggio Smalltalk sviluppato nei laboratori di Palo Alto.
Se l'idea iniziale di introdurre un nuovo paradigma di programmazione non aveva riscosso grossi successi all'interno della comunità di programmatori, l'introduzione del termine "Object Oriented" stimolò la fantasia degli analisti e negli anni immediatamente a seguire videro la luce un gran numero di linguaggi di programmazione ibridi come C++ e Objective-C, oppure ad oggetti puri come Eiffel.
Il decennio compreso tra il 1970 ed il 1980 fu decisivo per la metodologia "Object Oriented". L'introduzione di calcolatori sempre più potenti e la successiva adozione di interfacce grafiche (GUI) prima da parte della Xerox e successivamente della Apple, spinse i programmatori ad utilizzare sempre di più il nuovo approccio in grado adattarsi con più semplicità alla complessità dei nuovi sistemi informativi. Fu durante gli anni 70' che la metodologia "Object Oriented" guadagnò anche l'attenzione della comunità dei ricercatori poiché il concetto di ereditarietà sembrava potesse fornire un ottimo supporto alla ricerca sull'intelligenza artificiale.
Fu proprio questo incredibile aumento di interesse nei confronti della programmazione ad oggetti che, durante gli anni successivi, diede vita ad una moltitudine di linguaggi di programmazione e di metodologie di analisi spesso contrastanti tra loro. Di fatto, gli anni 80' possono essere riassunti in una frase di Tim Rentsch del 1982:

"*...Object Oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.*" [Rentsch82]

"...la programmazione Object Oriented fu nel 1980 quello che fu nel 1970 la programmazione strutturata. Tutti ne erano a favore. Tutti i produttori di

software promuovevano le loro soluzioni ed i loro prodotti a supporto. Ogni manager, a parole, ne era favorevole. Ogni programmatore la praticava (ognuno in modo differente). Nessuno sapeva di cosa si trattasse esattamente." [Rentsch82]

A causa dei tanti linguaggi orientati ad oggetti, dei molteplici strumenti a supporto, delle tante metodologie di approccio e di analisi gli anni ottanta rappresentano quindi la torre di Babele della programmazione ad oggetti.

1.14 Conclusioni

Che cosa sia esattamente un'applicazione Object Oriented appare quindi difficile da definire. Nel 1992 Monarchi e Puhr recensirono gran parte della letteratura esistente e riguardante metodologia Object Oriented.
Dalla comparazione dei testi recensiti furono identificati tutti gli elementi in comune al fine di determinare un unico modello. I risultati di questa ricerca possono essere riassunti nel modo seguente:

- Gli oggetti:
  - Sono entità anonime;
  - Incapsulano le proprie logiche;
  - Comunicano tramite messaggi;
- Organizzazione degli oggetti:
  - L' ereditarietà è il meccanismo che consente di organizzare oggetti secondo gerarchie;
  - La gerarchia deve prevedere la possibilità di definire oggetti astratti per un maggior realismo nella definizione di un modello;
  - Gli oggetti devono essere auto-documentanti;
- Programmi
  - Un programma realizza il modello di un sistema;
  - Un cambiamento di stato del sistema si riflette in un cambiamento di stato degli oggetti;
  - Gli oggetti possono operare in concorrenza;
  - Supportano tecniche di programmazione non orientate agli oggetti lì dove risulti necessario.

2 UNIFIED MODELING LANGUAGE

2.1 Introduzione

"Possedere un martello non fa un Architetto", progettare un edificio è cosa complessa, ma ancor più complesso è coordinare tutto il personale e le attività legate alla successiva fase di messa in opera.
Analizzare e disegnare una applicazione, non è molto differente dal costruire un edificio, bisogna prima risolvere tutte la complessità delineando la struttura finale del prodotto, e coordinare poi tutti gli attori coinvolti nella fase di scrittura del codice.
Scopo dell'analisi è quindi quello di modellare il sistema rappresentando in maniera precisa la formulazione di un problema che spesso può risultare vaga od imprecisa; scopo del progetto è quello di definirne i dettagli delineando le caratteristiche portanti per la successiva fase di sviluppo.
I passi fondamentali nell'analizzare e progettare un sistema possono essere schematizzati come segue:

Analisi di un sistema	
Fase	**Dettaglio**
Analisi	1. Esamina i requisiti e ne determina tutte le implicazioni; 2. Formula il problema in maniera rigorosa; 3. Definisce le caratteristiche importanti rimanadando i dettagli alla fase successiva.
Progetto	1. Definisce i dettagli della applicazione; 2. Disegna le entità e ne definisce le relazioni; 3. Analizza gli stati di un sistema e ne disegna le varie transizioni; 4. Assegna i compiti per la successiva fase di sviluppo.

Le tecniche di analisi e progettazione hanno subito una evoluzione che, camminando pari passo alle tecnologie, a partire dalla analisi funzionale ha portato oggi alla definizione del linguaggio UML (Unified Modelling Language) diventato in breve uno standard di riferimento per il mondo dell'industria.

2.2 Dal modello funzionale alle tecniche di Booch e OMT

Gli ultimi anni hanno visto la nascita di metodologie e strumenti di analisi e disegno che, utilizzando linguaggi visuali, facilitano la scomposizione in entità semplici di entità arbitrariamente complesse.
Prima dell'affermazione del paradigma Object Oriented, la metodologia di disegno più popolare era quella chiamata "analisi strutturale" che, aiutava l'analista nella scomposizione di un'applicazione in funzioni e procedure semplici, definendone per ognuna le interazioni con le altre. Le funzioni e le

procedure venivano a loro volta scomposte secondo un meccanismo gerarchico formato da funzioni e sotto funzioni, descrivendo le trasformazioni dei valori dei dati dentro al sistema mediante diagrammi di flusso. Un diagramma di flusso è un grafo i cui nodi sono processi, e i cui archi sono flussi di dati.
A seguito della affermazione del paradigma Object Oriented, gli anni 80' e l'inizio degli anni 90' videro la nascita di numerose metodologie di analisi e disegno ognuna con le proprie regole, le proprie notazioni ed i propri simboli. Tra il 1989 ed il 1984, le metodologie identificate erano oltre cinquanta ma, solo alcune di queste si affermarono rispetto alle altre, in particolare: la metodologia OMT (Object Modelling Technique) ideata da Rumbaugh che eccelleva nella analisi dei requisiti e della transizione di stato di un sistema, la metodologia di Booch che aveva il suo punto di forza nel disegno, ed infine la metodologia di Jacobsen chiamata OOSE (Object Oriented Software Engineering) che ebbe il merito di introdurre la definizione dei "casi d'uso".
Nelle prossime immagini sono riportati alcuni esempi di diagrammi definiti utilizzando le tre differenti metodologie.

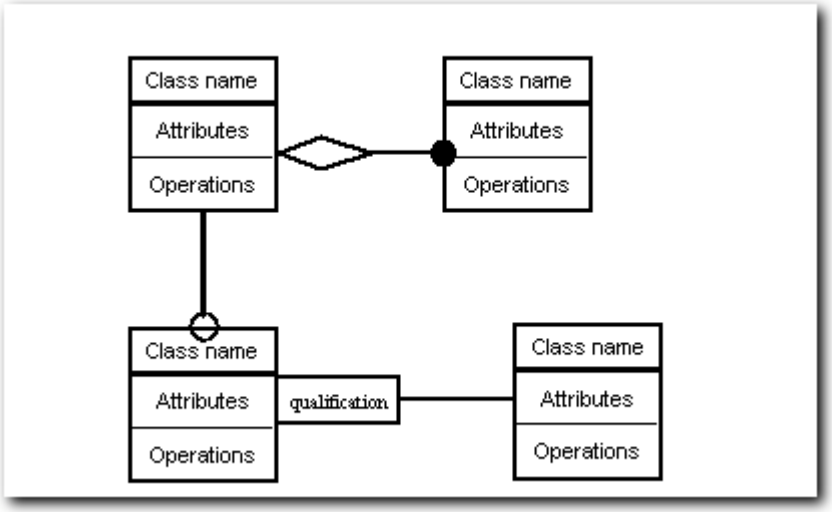


Figura 7: Diagramma OMT

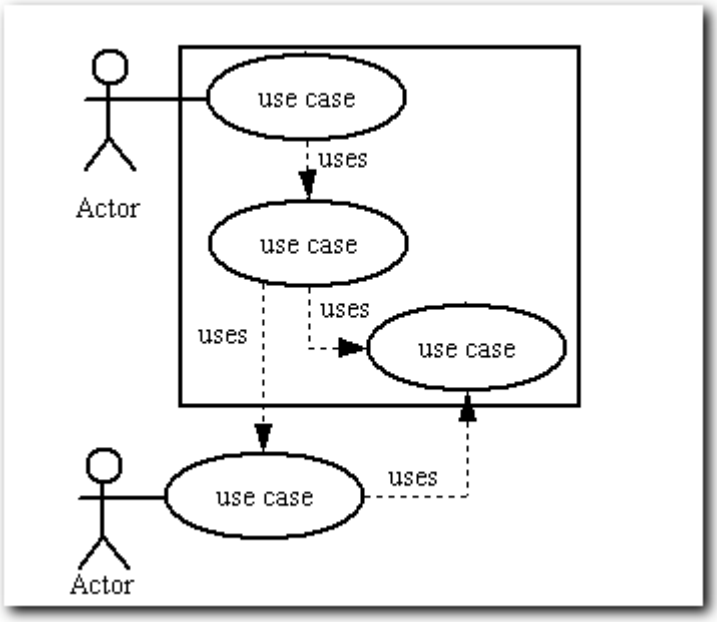


Figura 8: Diagramma dei casi-d'uso di OOSE

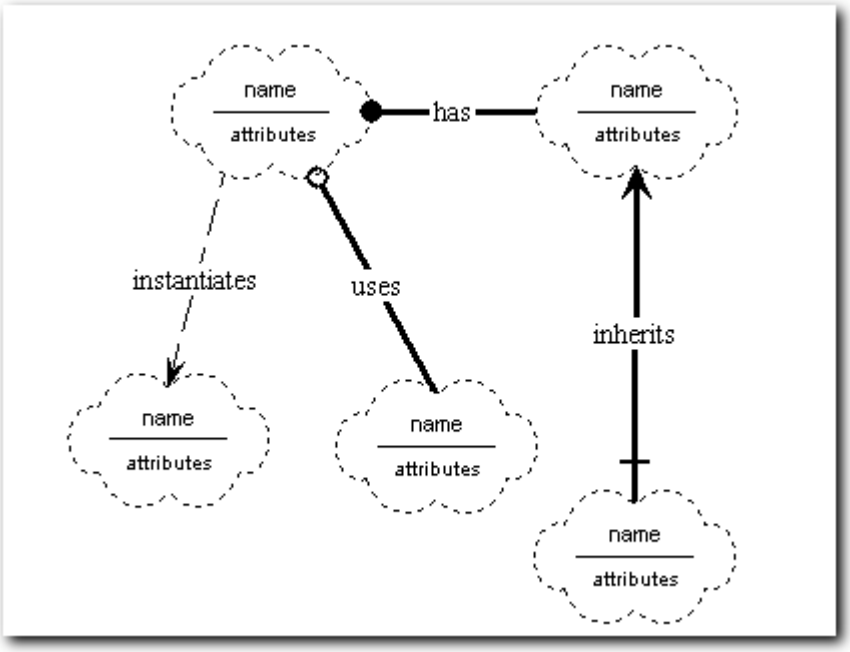
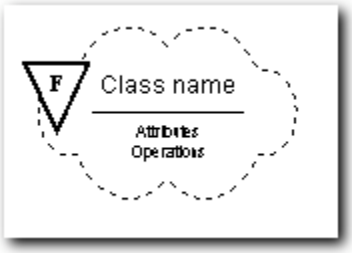
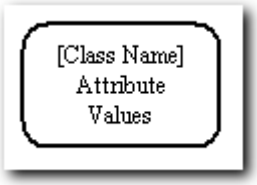
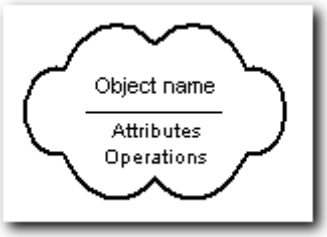
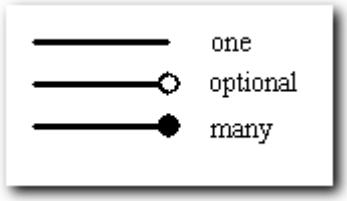
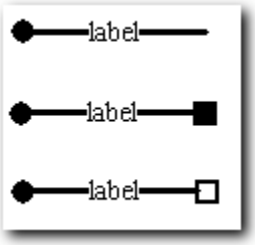
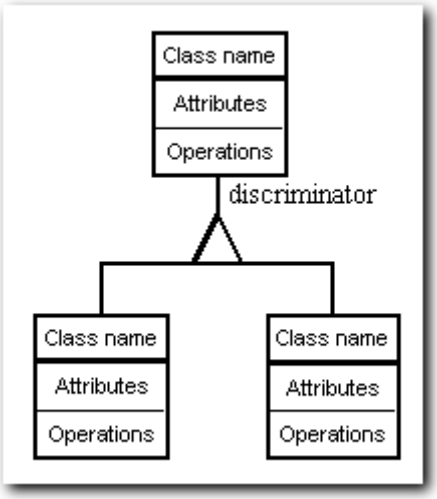
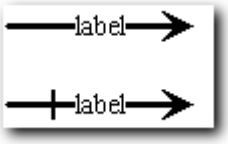


Figura 9: Diagramma di classe di Booch

Nella prossima tabella sono schematizzate alcune delle differenze tra le due notazioni di Booch ed Rumbaugh.

Comparazione tra le notazioni OMT e Booch		
Caratteristica	**OMT**	**Booch**

Diagrammi di classe	Rappresentati mediante rettangoli solidi con tre sezioni per identificare nome della classe, attributi ed operazioni. 	Rappresentati mediante nuvolette tratteggiate contenenti nome, attributi ed operazioni omettendo la suddivisione in aree. 
Visibilità di dati e metodi membro	+: Public -: Private #: Protected	null: Public |: Private ||: Protected ||| : implementazione
Classi Astratte	Identificate da molteplicità 0 nell'angolo destro in alto della rappresentazione della classe.	Parola chiave abstract utilizzata come proprietà della classe.
Oggetti (classi allocate)	Rettangolo solido contenente il nome della classe racchiusa tra parentesi. 	Nuvoletta solida con nome della classe ed attributi. 
Note	Testo racchiuso tra parentesi a fianco all'elemento destinatario.	Area rettangolare collegata con l'elemento destinatario da una linea tratteggiata.
Associazione	Linea solida tra le due classi coinvolte nella relazione ed il nome della relazione sul lato destro della linea.	Linea solida tra le due classi coinvolte nella relazione ed il nome della relazione sul lato destro della linea.

Associazione multipla	Esattamente 1 : nulla Zero o Più : Sfera solida Zero o Uno : sfera vuota 	Esattamente 1 : 1 Zero o Più : Sfera N Zero o Uno : Sfera 0..1 
Ereditarietà	Struttura ad albero che, a partire da un triangolo si sparge a ventaglio dalla superclasse alle sottoclassi. 	Freccia solida tra la sottoclasse e la superclasse. 

2.3 Unified Modelling Language

Nel 1994, Grady Booch e Jim Rumbaugh diedero vita al progetto di fusione rispettivamente tra le metodologie Booch e OMT rilasciando nell'ottobre del 1995 le specifiche del predecessore dell'attuale UML, allora chiamato UM (Unified Method). Alla fine del 1995 al gruppo si aggiunse Ivar Jacobson e dallo sforzo congiunto dei tre furono prodotte prima nel giugno 1996 ed infine ad ottobre dello stesso anno le versioni 0.9 e 0.9.1 di Unified Modelling Language (UML), ovvero Linguaggio di Modellazione Unificato.
Durante lo stesso periodo, aumentavano i consensi degli analisti e del mondo dell'industria che vedeva nel linguaggio UML uno standard prezioso nella definizione delle proprie strategie: alla fine del 1996 fu creato un consorzio di aziende che, contribuendo con l'esperienza dei propri analisti facilitò nel gennaio del 1997 il rilascio della versione 1.0 del linguaggio.

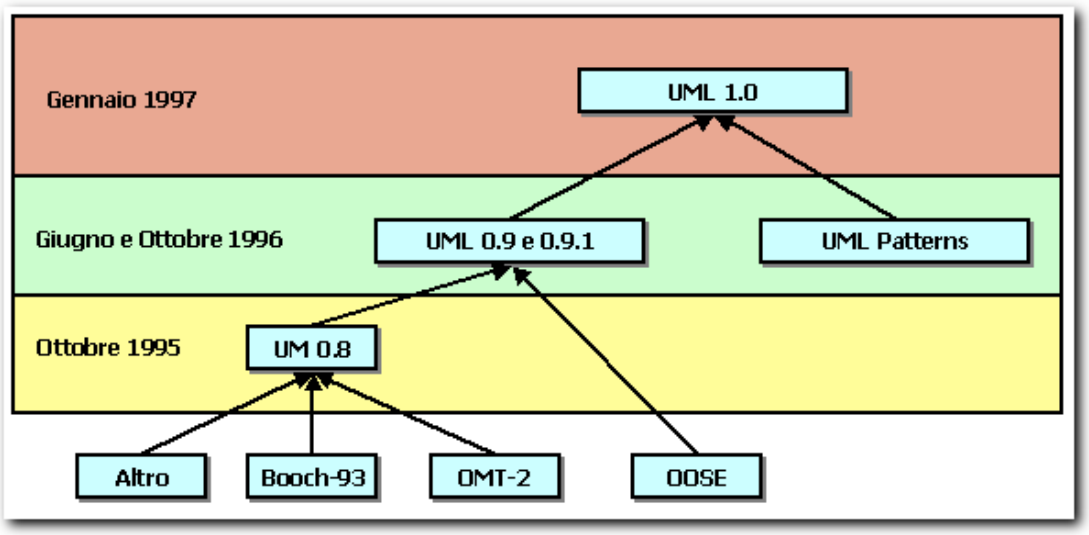


Figura 10: Time-Line del linguaggio UML

Il prodotto finale fu quindi un linguaggio ben definito, potente e facilmente applicabile a differenti realtà.
Nello stesso anno, la OMG (Object Managment Group) decise di adottare UML 1.0 come linguaggio di riferimento.

2.4 Le caratteristiche di UML

Unified Modeling Language è un linguaggio per tracciare le specifiche, disegnare e documentare la lavorazione di sistemi informativi complessi, mediante la definizione di modelli in grado di astrarre i vari aspetti del problema da risolvere. Di fatto, il linguaggio UML non intende fornire una metodologia per l'approccio al problema, ma lascia libero l'analista di decidere quale sia la strategia migliore da adottare ed il livello di astrazione di ogni modello.
I diagrammi messi a disposizione dal linguaggio UML sono i seguenti:

- Use Case Diagram;
- Class Diagram;
- Behavior Diagrams:
  - State Diagram;
  - Activity Diagram;
  - Sequence Diagram;
  - Collaboration Diagram;
- Implementation Diagrams:
  - Component Diagram;
  - Deployment Diagram.

Tutti questi diagrammi danno percezione dell'intero sistema da diversi punti di vista: in particolare, come il paradigma ad oggetti accentra l'attenzione del programmatore verso la rappresentazione di dati e non più verso le funzioni, l'analisi UML consente di disegnare il sistema defininendo le entità che lo compongono, disegnando quali oggetti concorrono alla definizione di una funzionalità ed infine tutte le possibili interazioni tra gli oggetti. Infine, ogni oggetto più essere definito in dettaglio tramite gli "Activity Diagram" o gli "State Diagram".

2.5 Genealogia del linguaggio UML

Durante il processo di unificazione, gli autori del linguaggio inserirono all'interno di UML quanto di meglio era proposto dal mondo delle scienze dell'informazione, attingendo da metodologie diverse e spesso, non collegate al paradigma Object Oriented. Di fatto, la notazione del linguaggio UML è una miscela di simboli appartenenti a diverse metodologie, con l'aggiunta di alcuni nuovi, introdotti in modo specifico a completamento del linguaggio.
L'albero genealogico del linguaggio UML è quindi veramente complesso, ed è brevemente schematizzato nella tabella seguente:

Albero genealogico di UML	
Caratteristica	**Genealogia**
Use-Case Diagram	Derivati nelle apparenze dai diagrammi della metodologia OOSE.
Class Diagram	Derivati dai *class diagram* di OMT, Booch e molte altre metodologie.
State Diagram	Basati sugli *State Diagram* di David Harel con alcune piccole modifiche.
Activity Diagram	Attingono dai diagrammi di flusso comuni a molte metodologie Object Oriented e non.
Sequence Diagram	Possono essere ritrovati sotto altro nome in molte metodologie Object Oriented e non.
Collaboration Diagram	Rappresentano una evoluzione degli *object diagram* di Booch, degli *object interation Graph* di Fusion e di altri appartenenti ad altre metodologie.
Implementation Diagrams (Component e Deployment)	Derivanti dai *module diagram* e *process diagram* di Booch.

2.6 Use Case Diagrams

Uno *Use-Case Diagram* descrive le attività del sistema dal punto di vista di un osservatore eterno, rivolgendo l'attenzione alle attività che il sistema svolge,

prescindendo da come devono essere eseguite. Di fatto, questi diagrammi definisco gli scenari in cui gli attori interagiscono tra di loro.
Consideriamo ad esempio lo scenario di una biblioteca pubblica:

"Un utente si reca in biblioteca per consultare un libro. Richiede il libro al bibliotecario che, eseguite le dovute ricerche consegna il libro al lettore. Dopo aver consultato il libro, il lettore riconsegna il libro al bibliotecario."
Attori ed azioni compiute possono essere schematizzate come segue:

La Biblioteca Pubblica	
Attori	**Azioni**
Utente	• Richiede un libro al bibliotecario; • Ritira il libro; • Legge il libro; • Restituisce il libro al bibliotecario.
Bibliotecario	• Accetta la richiesta dell'utente; • Ricerca il libro; • Consegna il libro; • Ritira il libro dall'utente;

Uno *use-case diagram* è quindi un insieme di attori ed azioni. Un attore è l'entità che scatena una azione, e rappresentano il ruolo che l'entità svolge nello scenario in analisi. Il collegamento tra attore ed azione è detto "comunicazione". Il diagramma relativo a questo scenario è quindi il seguente:

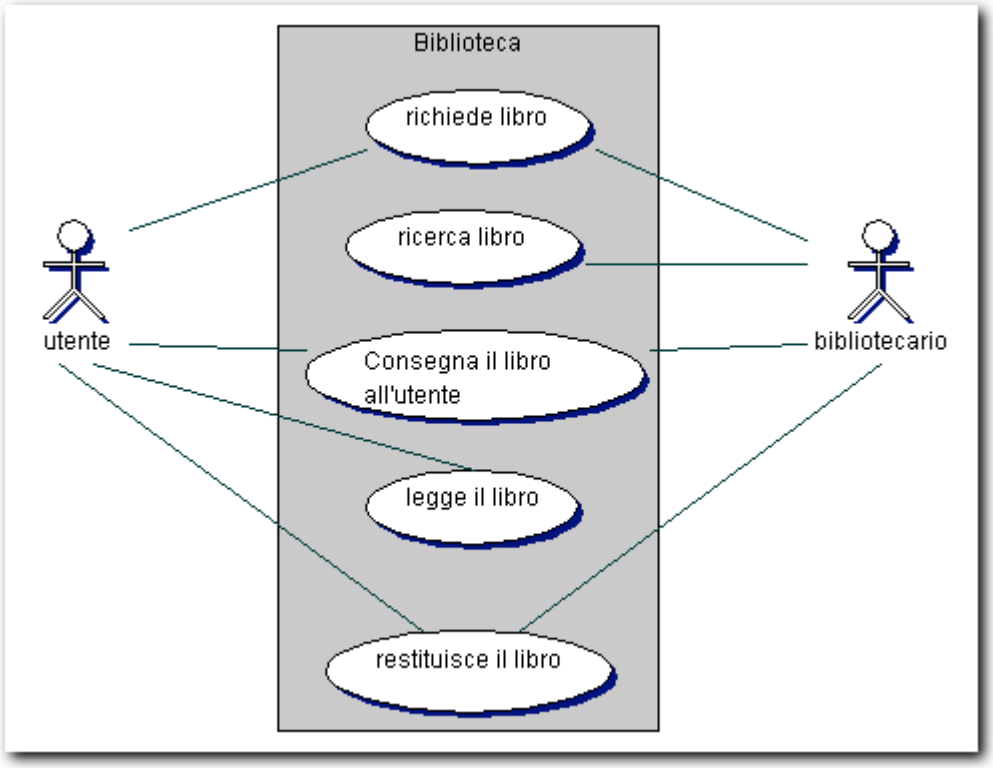
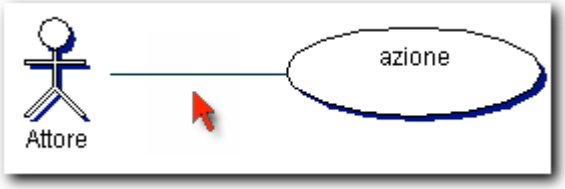


Figura 11: Use Case Diagram

I simboli utilizzati all'interno di questi diagrammi sono i seguenti:

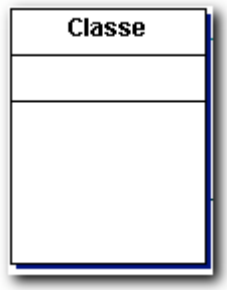
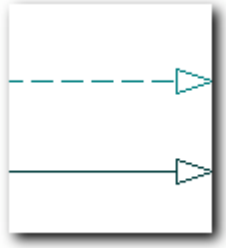
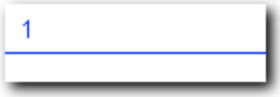
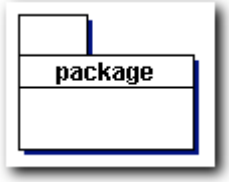
Notazione per i "*use case diagrams*"	
Simbolo	**Descrizione**
	Attore
	Azione
	Comunicazione

Per concludere, il ruolo di questi tipo di diagrammi all'interno del disegno del sistema è quello di:

1. Determinare le caratteristiche macroscopiche del sistema;
2. Generare i casi di test: la collezione degli scenari è un ottimo suggerimento per la preparazione dei casi di test del sistema.

2.7 Class Diagrams

Un *class-diagram* fornisce la fotografia del sistema, schematizzando le classi che lo compongono, e le relazioni tra di loro.
I simboli utilizzati nei *class-diagram* sono i seguenti:

Notazione per i "*class-diagram*"	
Simbolo	**Descrizione**
	Classe o Interfaccia : rappresentata da un rettangolo suddiviso in tre parti: nome, attributi ed operazioni. I nomi di classi astratte o di interfacce appaiono in corsivo.
	Relazione di generalizzazione : esprime una relazione di ereditarietà. La freccia è orientata dalla sottoclasse alla superclasse.
	Associazioni : relazione tra due oggetti se, un oggetto deve utilizzarne un altro.
	Aggregazioni : associazioni in cui una classe appartiene ad una collezione di classi di un'altra.
	Package : collezione di oggetti UML logicamente correlati. Un package può contenere altri package.

Un possibile *class-diagram* per la biblioteca descritta nel paragrafo precedente può quindi essere il seguente.

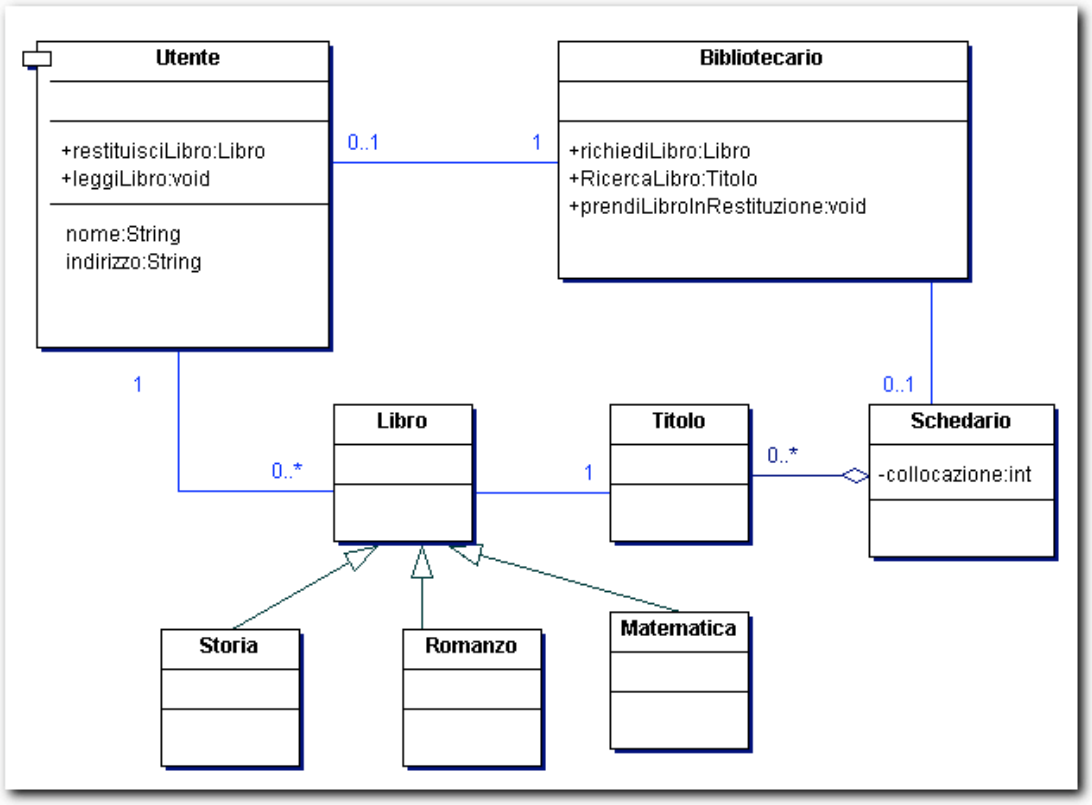


Figura 12: Class Diagram di biblioteca

Dal diagramma in figura, si nota che, una associazione (aggregazione) può essere accompagnata da un indice detto molteplicità. La molteplicità di una associazione (aggregazione) rappresenta il numero di oggetti all'altro capo del simbolo che è possibile collezionare. Nel nostro esempio, un libro può essere associato ad un solo utente, ma un utente può prelevare più di un libro. Tutte le possibili molteplicità sono schematizzate nella prossima tabella:

Molteplicità	
Simbolo	**Descrizione**
0..1 oppure	Un oggetto può collezione 0 o 1 oggetto di un altro tipo.
0..* oppure *	Un oggetto può collezione 0 o più oggetti di un altro tipo.
1	Esattamente un oggetto.
1..*	Almeno un oggetto.

Nel caso in cui sia necessario organizzare diagrammi arbitrariamente complessi, è possibile raggruppare le classi di un class-diagram all'interno di package. Ad esempio, è possibile raggruppare tutti i libri a seconda della

tipologia utilizzando un package per ogni genere letterario come mostrato nella prossima figura.

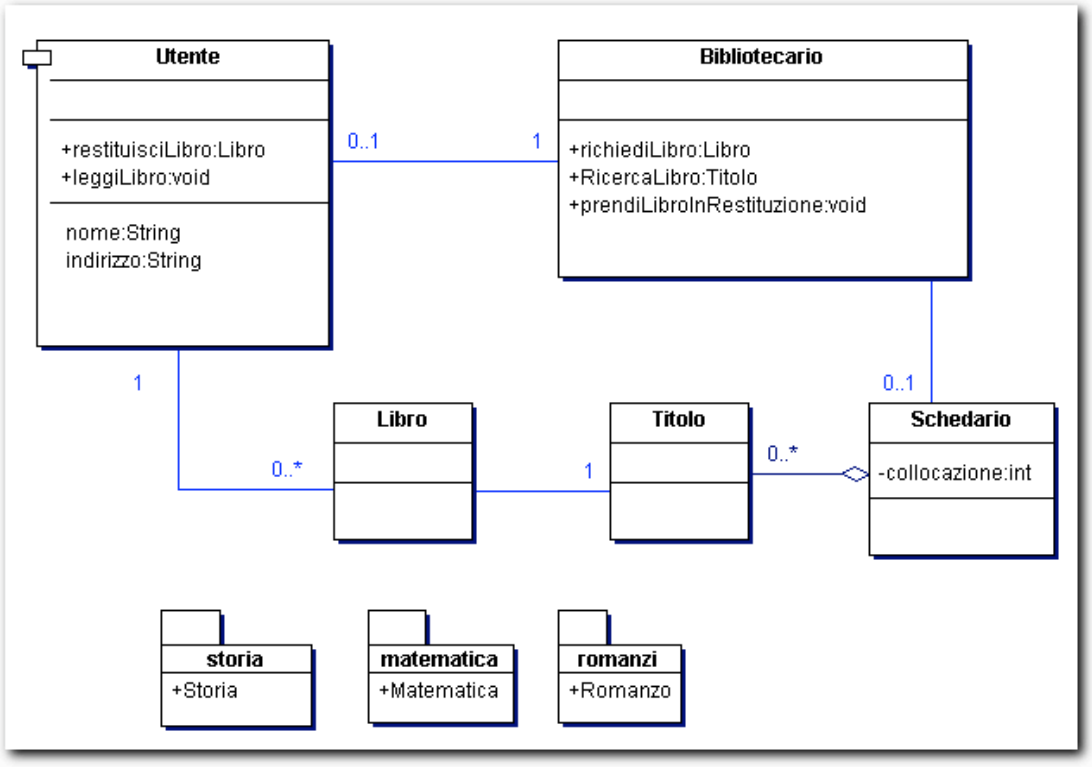


Figura 13: Organizzazione di classi mediante package

2.8 Sequence diagrams

Un *sequence diagram* descrive le interazioni tra oggetti, mettendo a fuoco l'ordine temporale con cui le varie azioni sono eseguite partendo dall'alto del diagramma e procedendo verso il basso. Ricordando le classi definite nel *class diagram* della nostra applicazione biblioteca, e lo scenario descritto nello *use-case diagram*, nelle due prossime figure sono rappresentati i *sequence diagram* che forniscono la specifica delle interazioni tra le classi necessarie ad implementare rispettivamente le due funzionalità di richiesta del libro al bibliotecario e la successiva restituzione dopo la lettura.

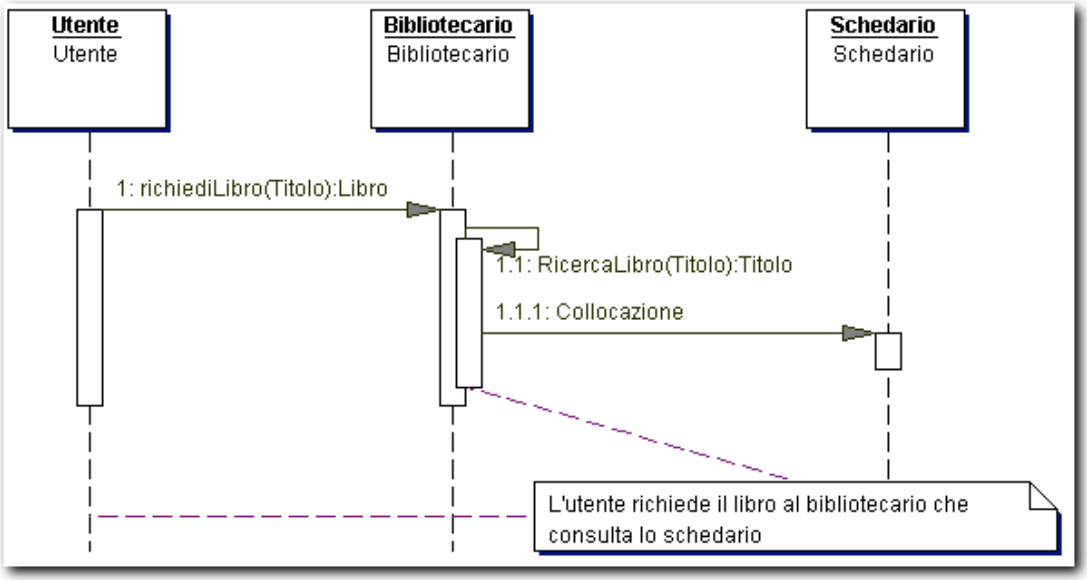


Figura 14: Richiesta di un libro

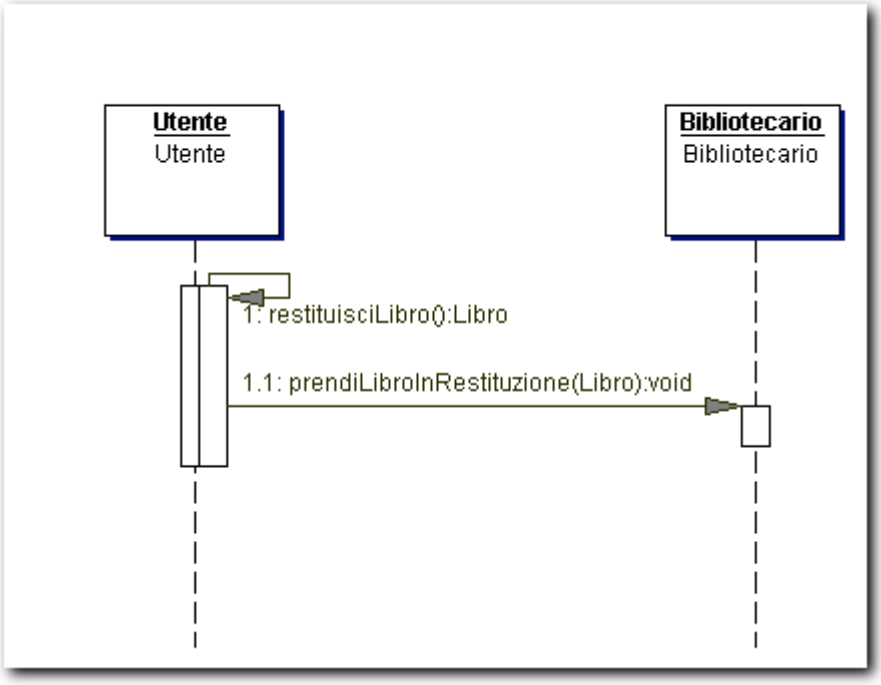
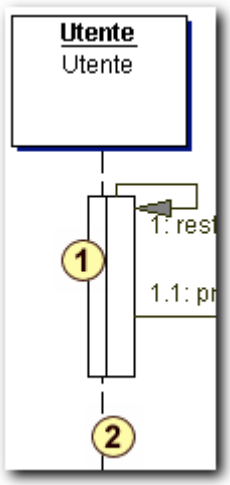
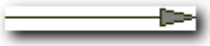


Figura 15: Lettura e restituzione del libro

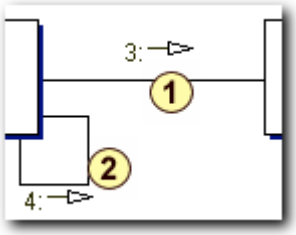
La simbologia utilizzata per realizzare *sequence diagram* è descritta nella prossima tabella:

Notazione per i "*sequence diagram*"	
Simbolo	**Descrizione**
	Oggetto: rappresenta un oggetto il cui nome è evidenziato in alto in grassetto , ed il cui tipo è riportato subito sotto il nome.
	(1) Activation bar: definisce la durata della attivazione di una funzionalità. (2) Lifetime: rappresenta il tempo di esistenza di un oggetto.
	Message Call : chiamata tra due oggetti o tra operazioni di uno stesso oggetto.

2.9 Collaboration Diagram

Come i *sequence diagram*, i *collaboration diagram* descrivono interazioni tra oggetti fornendo le stesse informazioni da un altro punto di vista. Di fatto, se nel caso precedente il soggetto del diagramma era il tempo, ora vengono messe a fuoco le regole di interazione tra gli oggetti.
I simboli utilizzati in un *collaboration diagram* sono i seguenti:

Notazione per i "*collaboration diagram*	
Simbolo	**Descrizione**
	Object-Role: o regole, sono rappresentati da aree contenenti il nome dell'oggetto, il nome della classe o entrambi con il nome della classe preceduto dal carattere ":".

	Message Call : chiamata tra due oggetti o tra operazioni di uno stesso oggetto.

I *sequenze diagram* descritti nel paragrafo precedente, possono essere espressi in forma di *collaboration diagram* nel seguente modo:

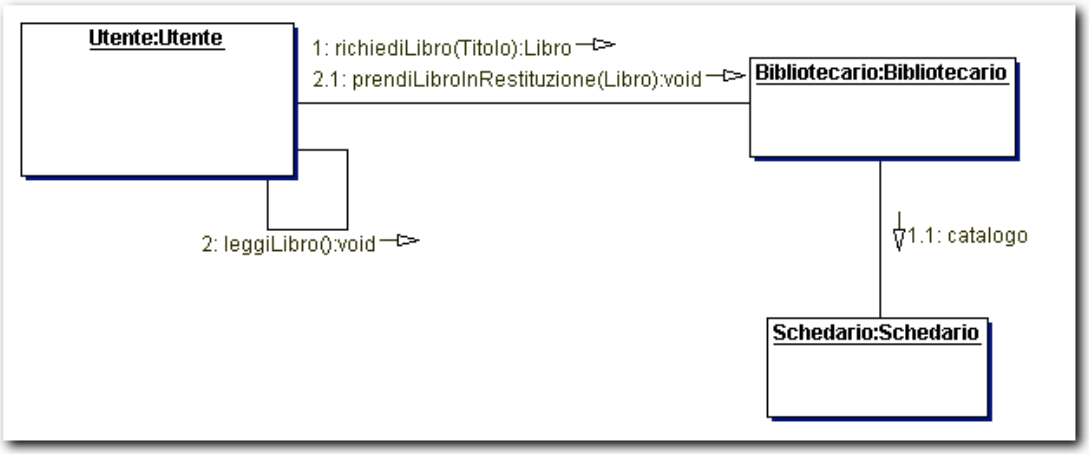


Figura 16: Collaboration Diagram di biblioteca

Come appare chiaro dal diagramma riportato in figura, in un *collaboration diagram* ogni invio di messaggio è identificato da un numero sequenziale. Le regole con cui viene assegnato il numero sequenziale sono le seguenti:

- Il messaggio di primo livello ha numero sequenziale 1;
- Messaggi sullo stesso livello hanno stesso prefisso e suffisso variabile (1,2,3…n) dipendentemente dal tempo di occorrenza del messaggio.

2.10 Statechart Diagram

Vedremo nei capitoli successivi che un oggetto, oltre ad avere un aspetto, possiede anche uno stato. Lo stato di un oggetto dipende dall'attività corrente o dalle condizioni al contorno in un determinato istante. Uno *statechart diagram* schematizza tutti i possibili stati di un oggetto e le transizioni che causano il cambiamento di stato dell'oggetto.
Nella prossima tabella sono schematizzati i simboli utilizzati per realizzare uno *statechart diagram.*

Notazione per i "*statechart diagram*	
Simbolo	**Descrizione**
	State: rappresenta lo stato di un oggetto ad un determinato istante a seguito di una transizione.
	Initial state: rappresenta lo stato iniziale dell'oggetto.
	End state: rappresenta lo stato finale dell'oggetto.
	Transition: rappresenta la transizione tra due stati o, da uno stato a se stesso (self transition)

Supponiamo che il nostro utente della biblioteca pubblica voglia effettuare una pausa desideroso di recarsi a prendere un caffe al distributore automatico. La prima azione che dovrà essere compiuta è quella di inserire le monete fino a raggiungere la somma richiesta per erogare la bevanda. Nel prossimo esempio, descriveremo il processo di accettazione delle monete da parte del distributore. Il distributore in questione accetta solo monete da 5,10 e 20 centesimi di euro, la bevanda costa 25 centesimi di euro ed il distributore non eroga resto.
Lo *statechart diagram* sarà il seguente:

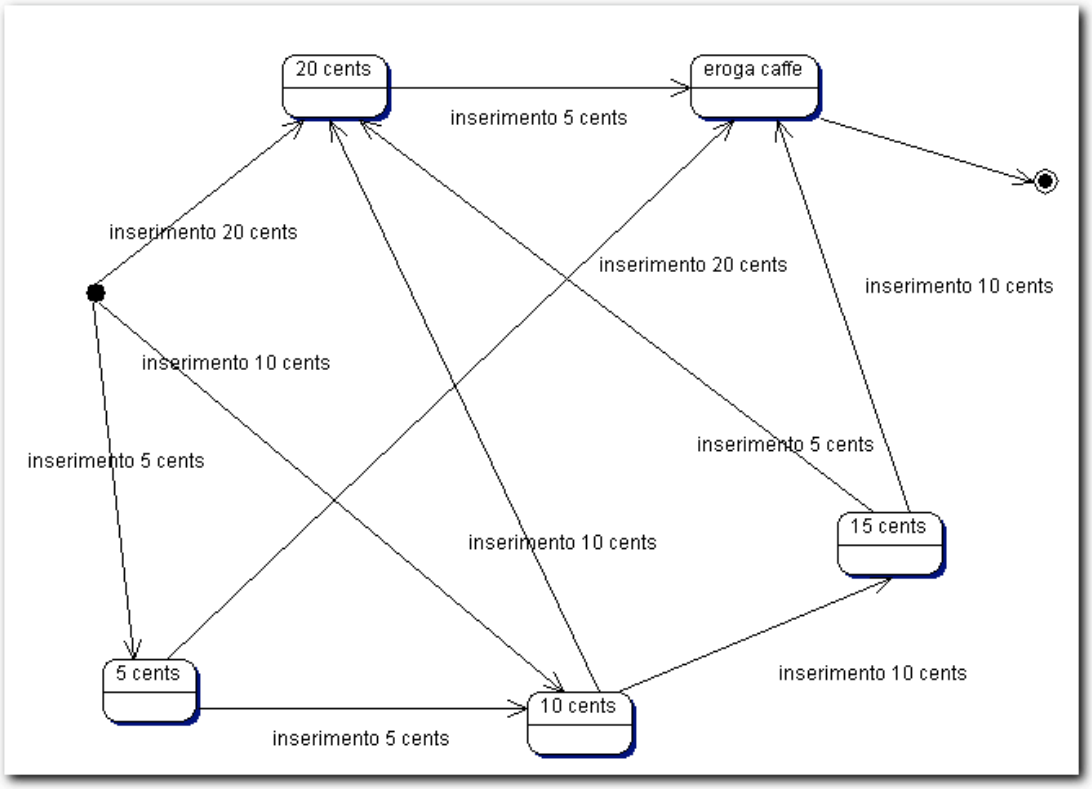


Figura 17: Statechart diagram

Nel nostro diagramma, abbiamo un solo stato iniziale ed un solo stato finale. Stato iniziale e finale sono due stati finti (dummy) in quanto sono necessari solo ad indicare l'inizio e la fine della azione descritta nel diagramma.

2.11 Activity Diagrams

Un *activity diagram* può essere paragonato ad uno *statechart diagram* con la differenza che, se uno *statechart diagram* focalizza l'attenzione dell'analisi sull'attività di un singolo oggetto appartenente ad un processo, un *activity diagram* descrive le attività svolte da tutti gli oggetti coinvolti nella realizzazione di un processo, descrivendone le reciproche dipendenze. Torniamo ad occuparci nuovamente della biblioteca comunale, ed analizziamo ancora, questa volta mediante *activity diagram* le attività compiute dalla due classi, Utente e Bibliotecario, al momento della richiesta del libro.

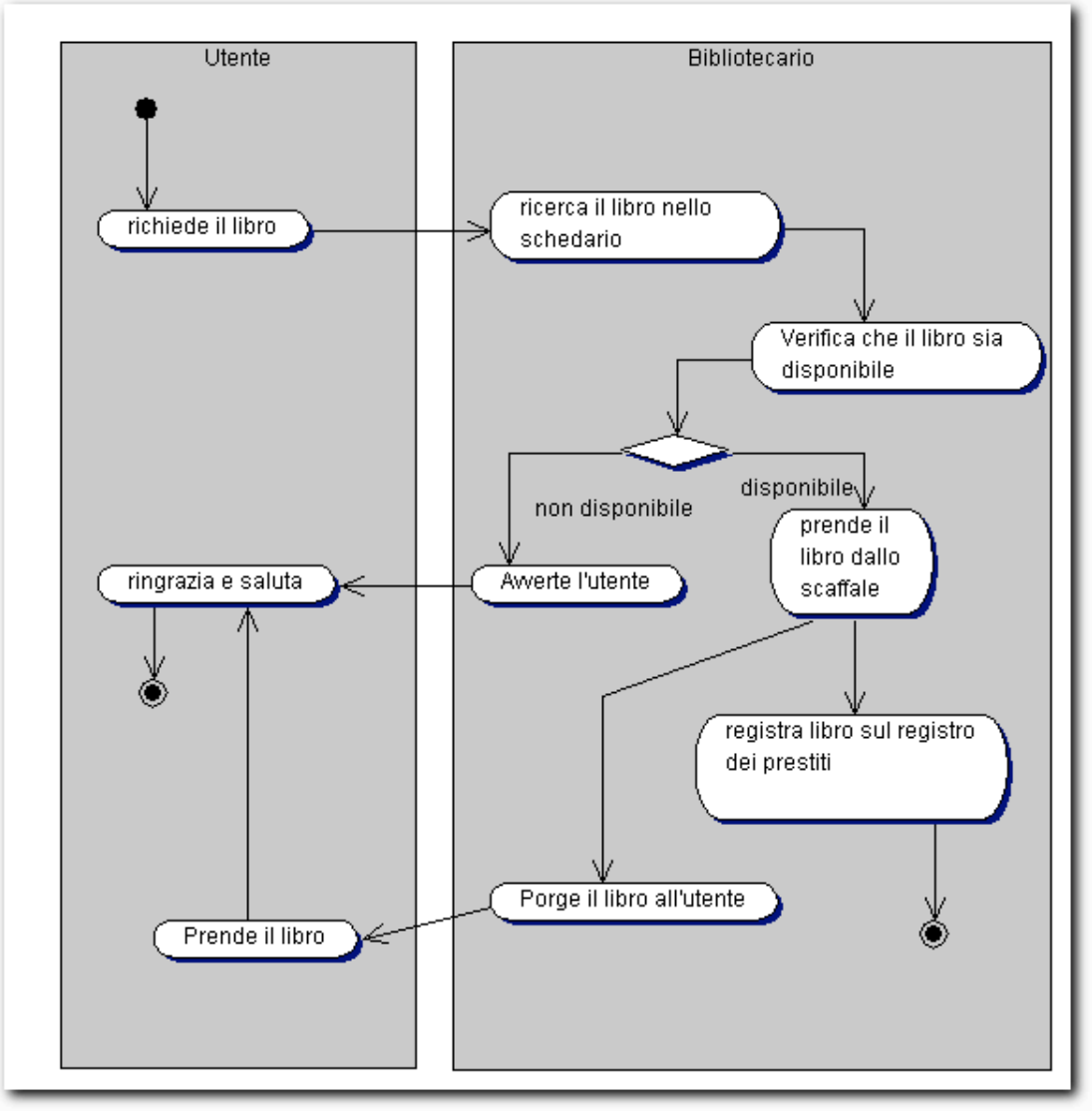
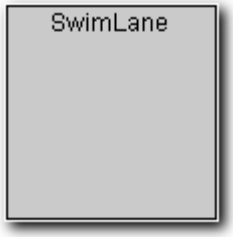
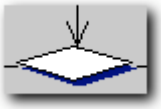


Figura 18: Richiesta del libro al bibliotecario : activity diagram

Un *activity diagram* può essere suddiviso mediante simboli detti "swimlanes" o corsie, che rappresentano l'oggetto responsabile delle attività contenute. Una attività può essere collegate ad una o più attività correlate.
I simboli utilizzati per disegnare il diagramma sono descritti nella tabella seguente:

Notazione per gli "*activity diagram*	
Simbolo	**Descrizione**
	Activity: rappresenta un a attività svolta da un oggetto ad un determinato istante a seguito di una transizione.

	Initial state: rappresenta lo stato iniziale.
	End state: rappresenta lo stato finale.
	Transition: rappresenta la relazione tra due attività.
	Swimlane: rappresenta l'oggetto che compie determinate attività.
	Merge: rappresenta una condizione che modifica il flusso delle attività.

2.12 Component Diagram e Deployment Diagram

Il linguaggio UML definisce *componente*, un modulo (od oggetto) del codice che comporrà l'applicazione finale. Entrambi, *component diagram* e *deployment diagram*, rappresentano due viste differenti delle componenti che formano il sistema finale.
Un *component diagram* rappresenta l'analogo fisico di un *class diagram,* un *deployment diagram* schematizza la configurazione fisica delle componenti software del sistema.
La prossima immagine rappresenta il *component diagram* costruito a partire dal *class diagram* della applicazione biblioteca.

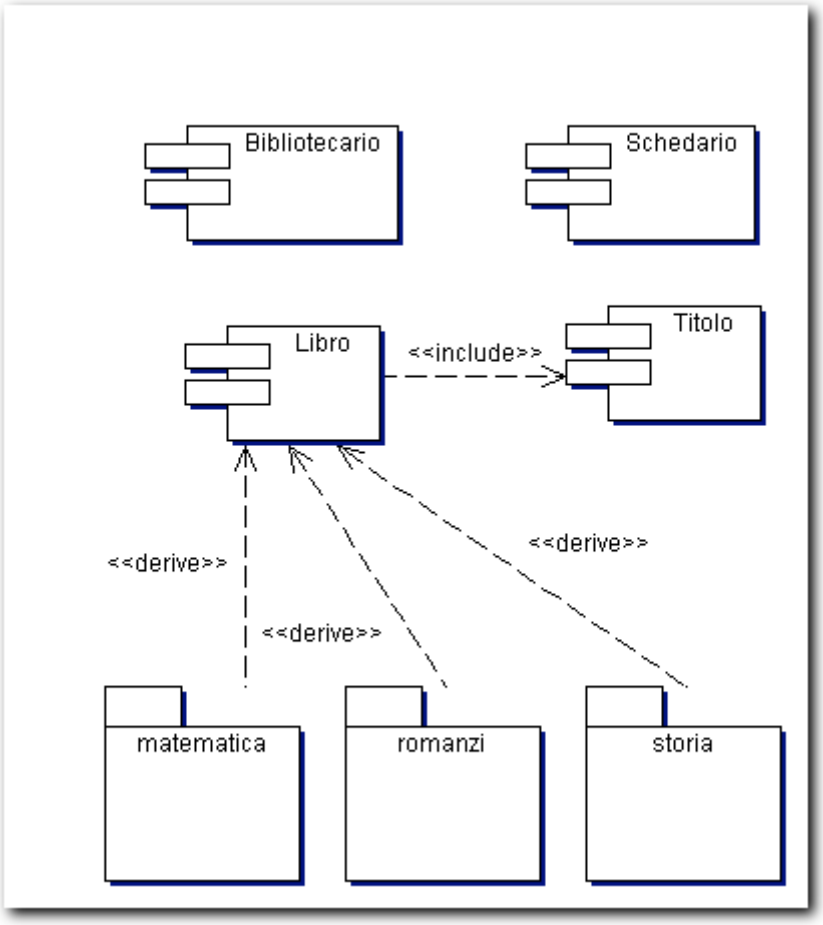


Figura 19: Component diagram di biblioteca

3 IL LINGUAGGIO JAVA

3.1 Introduzione

Java è un linguaggio di programmazione Object Oriented indipendente dalla piattaforma, modellato dal linguaggio C e C++ di cui mantiene alcune 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 e generati dal compilatore java, in istruzioni eseguibili dalla macchina locale.
La natura di linguaggio Object Oriented di Java, consente lo sviluppo applicazioni utilizzando oggetti concettuali piuttosto che procedure e funzioni. La sintassi orientata ad oggetti di Java supporta la creazione di classi concettuali, consentendo al programmatore di scrivere codice stabile e riutilizzabile per mezzo del paradigma Object Oriented, secondo il quale il programma è scomposto in concetti piuttosto che funzioni o procedure.
La sua stretta parentela con il linguaggio C a livello sintattico fa sì 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).

3.2 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: l'idea era creare una tecnologia che fosse applicabile ad ogni tipo di strumento elettronico al fine di garantire un alto grado di interattività, diminuire i tempi di sviluppo ed abbassare i costi di implementazione utilizzando un unico ambiente operativo.
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 una tecnologia partendo dal linguaggio C++. La prima versione del linguaggio fu chiamata Oak.
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. Nel 1993 il "National Center for Supercomputing

Applications (NCSA)" rilasciò Mosaic, una applicazione che consentiva agli utenti di accedere mediante interfaccia grafica ai contenuti di Internet allora ancora limitati al semplice testo.
Nell'arco di un anno, da semplice strumento di ricerca, Internet è diventato il mezzo di diffusione che oggi tutti conosciamo in grado di trasportare non solo testo, ma ogni tipo di contenuto multimediale: fu durante il 1994 che la SUN decise di modificare il nome di Oak in Java.
La prima versione del linguaggio consentiva lo sviluppo applicazioni stand-alone e di piccole applicazioni, chiamate applet, in grado di essere eseguite attraverso la rete.
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 produttori di software incluse IBM, Hewlett Packard e Microsoft.

3.3 Indipendenza dalla piattaforma

Le istruzioni binarie di Java, indipendenti dalla piattaforma, sono comunemente conosciute come Bytecode. Il Bytecode di java è prodotto dal compilatore e necessita di uno strato di software per essere eseguito. Quest'applicazione, detta interprete (*Figura 20*), è nota come Java Virtual Machine (che per semplicità indicheremo con JVM).
La JVM è un programma scritto mediante un qualunque linguaggio di programmazione, è dipendente dalla piattaforma su cui deve eseguire il Bytecode, e traduce le istruzioni Java dalla forma di Bytecode in istruzioni comprensibili dal processore della macchina che ospita l'applicazione.

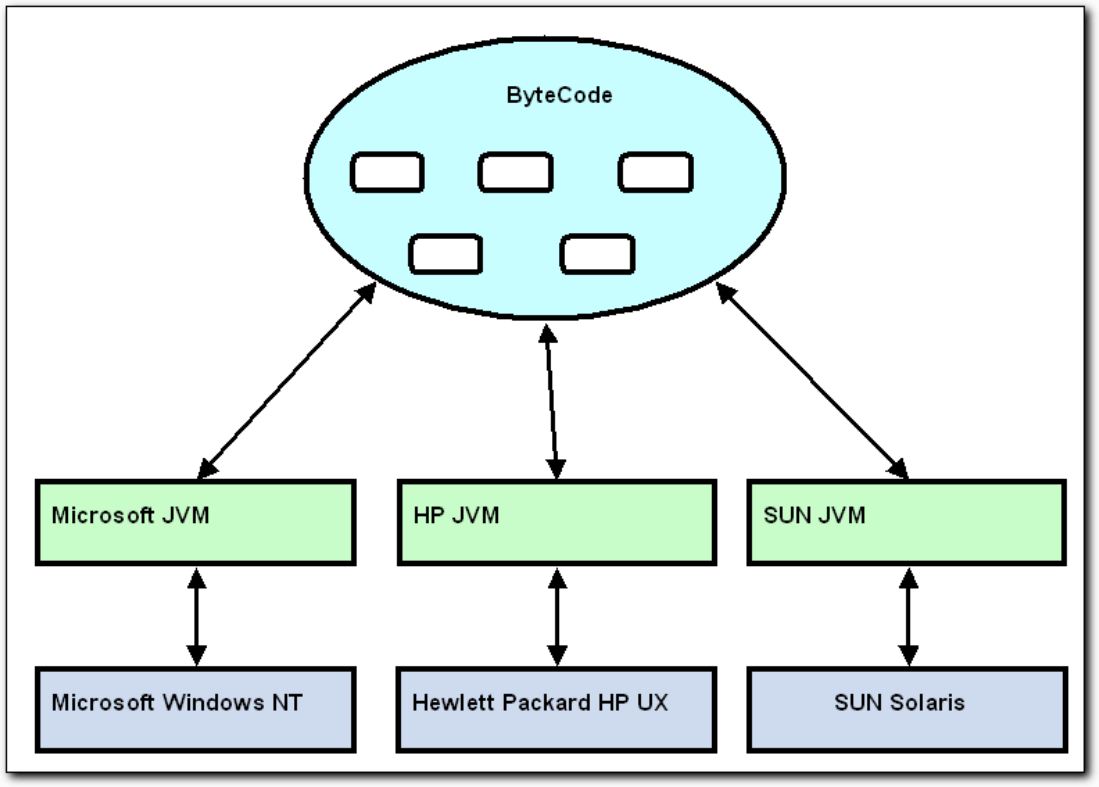


Figura 20: Architettura di una applicazione Java

Non essendo il Bytecode legato ad una particolare architettura hardware, questo fa sì che per trasferire un'applicazione Java da una piattaforma ad un'altra è necessario solamente che la nuova piattaforma sia dotata di un'apposita JVM. In presenza di un interprete un'applicazione Java potrà essere eseguita su qualunque piattaforma senza necessità di essere ulteriormente compilata.
In alternativa, si possono utilizzare strumenti come i "Just In Time Compilers", compilatori in grado di tradurre il Bytecode in un formato eseguibile per una specifica piattaforma al momento dell'esecuzione del programma Java. I vantaggi nell'uso dei compilatori JIT sono molteplici.
La tecnologia JIT traduce il Bytecode in un formato eseguibile al momento del caricamento in memoria, ciò consente di migliorare le performance dell'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 dell'esecuzione della applicazione che, deve essere prima compilata e poi eseguita.
Un ultimo aspetto interessante di Java è quello legato agli sviluppi che la tecnologia sta avendo. Negli ultimi anni, molti produttori di sistemi elettronici hanno iniziato a rilasciare processori in grado di eseguire direttamente il Bytecode a livello di istruzioni macchina senza l'uso di una virtual machine.

3.4 Uso della memoria

Un problema scottante quando si parla di programmazione è la gestione dell'uso della memoria; quando si progetta un'applicazione, è molto complesso affrontare le problematiche inerenti al mantenimento degli spazi di memoria.
Una gestione della memoria superficiale o mal pensata è spesso causa di un problema noto come "memory leak": l'applicazione alloca risorse senza riuscire a rilasciarle completamente determinando la perdita di piccole porzioni di memoria che, se sommate, possono provocare l'interruzione anomala dell'applicazione o, nel caso peggiore, all'intero sistema che la ospita.
E' facile immaginare l'instabilità di sistemi informativi affetti da questo tipo di problema. Tali applicazioni richiedono spesso lo sviluppo di complesse procedure specializzate nella gestione, nel tracciamento e nel rilascio della memoria allocata.
Java risolve il problema alla radice sollevando il programmatore dall'onere della gestione della memoria grazie ad un meccanismo detto *Garbage Collector*. Il Garbage Collector, tiene traccia degli oggetti utilizzati da un'applicazione Java, nonché delle referenze a tali oggetti. Ogni volta che un oggetto non è più referenziato (ovvero utilizzato da altre classi), per tutta la durata di uno specifico intervallo temporale, è rimosso dalla memoria e la risorsa liberata è nuovamente messa a disposizione dell'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 affermare 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 l'applicazione non sta utilizzando nessuno dei due come schematizzato nella *figura 21*.

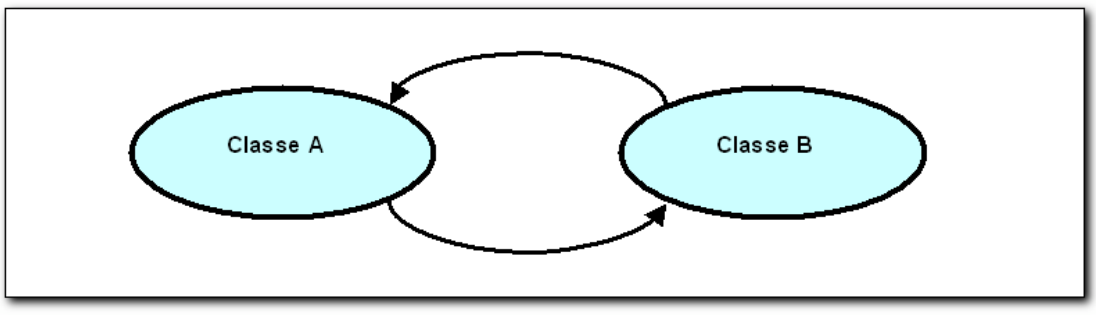


Figura 21: riferimento circolare

3.5 Multi-threading

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) eseguite parallelamente.
L'uso di thread rappresenta un modo semplice di gestire la concorrenza tra processi poiché, gli spazi di indirizzamento della memoria dell'applicazione sono condivisi con i thread, eliminando così la necessità di sviluppare complesse procedure di comunicazione tra processi.
Nei capitoli successivi torneremo a parlare in maniera approfondita di quest'aspetto del linguaggio.

3.6 Dynamic Loading and Linking

Il meccanismo tradizionale per caricare ed eseguire un'applicazione, chiamato "*static linking*", prevede il caricamento in memoria di tutte le procedure e funzioni raggruppate in un unico file eseguibile.
Questo tipo di approccio ha alcuni svantaggi:

1. I programmi contengono sempre tutto il codice ed i dati che potrebbero essere richiesti durante l'esecuzione;
2. Tutto il codice ed i dati devono essere caricati in memoria al momento dell'esecuzione dell'applicazione anche se solo una piccola parte di questi saranno realmente utilizzati;
3. L'intero codice di un'applicazione deve essere ricaricato quando una procedura o una funzione sono modificate.

Java supporta il meccanismo detto di "Dynamic Loading and Linking" secondo il quale ogni modulo del programma (classe) è memorizzato in un determinato file e quando un programma Java è eseguito, le classi sono caricate e allocate solo al momento del loro effettivo utilizzo.
I vantaggi nell'uso di questo modello sono molteplici:

1. Riduce il consumo di memoria. Un oggetto è caricato in memoria solo se richiesto da un altro membro della applicazione;
2. Le applicazioni possono controllare facilmente l'uso della memoria caricando gli oggetti solo su richiesta.

Java prevede un gran numero di classi già compilate che forniscono molte funzionalità: strumenti di scrittura e lettura da file o da altre sorgenti, meccanismi di interfacciamento con le reti, ecc. ecc..

3.7 Meccanismo di caricamento delle classi da parte della JVM

Il meccanismo di "Dynamic Loading and Linking" incide sul modo di esecuzione di un'applicazione da parte delle JVM. Di fatto, quando la JVM è avviata la prima volta, il primo passo che deve compiere è quello di caricare le classi indispensabili all'applicazione affinché possa essere eseguita. Questa fase avviene secondo uno schema temporale ben preciso a carico di un modulo interno chiamato "*launcher*".
Le prime ad essere caricate dal launcher 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 posizione sul 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.

3.8 La variabile d'ambiente CLASSPATH

La variabile di ambiente CLASSPATH, contiene l'elenco delle cartelle all'interno delle quali la JVM dovrà ricercare le definizioni delle classi java contenenti il Bytecode da eseguire.
I file contenenti le definizioni delle classi Java possono essere memorizzati all'interno di una cartella o raggruppati in archivi compressi in formato zip o jar: nel caso in cui la JVM trovi, all'interno della variabile CLASSPATH, un riferimento ad un file con estensione zip o jar, automaticamente naviga all'interno dell'archivio alla ricerca delle classi necessarie.
Il formato della variabile d'ambiente CLASSPATH varia a seconda del sistema operativo che ospita la virtual machine: Nel caso dei sistemi operativi della Microsoft (Windows 95,98,NT,2000,XP) contiene l'elenco delle cartelle e degli archivi separati dal carattere ';'. Ad esempio la variabile d'ambiente

```
CLASSPATH = .;.\;c:\jdk\lib\tools.jar:c:\src\myclasses\;
```

fa riferimento alla cartella corrente, all'archivio *tools.jar* memorizzato nella cartella c:\jdk\lib\, ed alla cartella c:\src\myclasses\.

3.9 Il Java Software Development Kit (SDK)

Java Software 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.
Il Java SDK comprende, oltre alle utilità a linea di comando, un completo insieme di classi già compilate ed il relativo codice sorgente.
La documentazione generalmente distribuita separatamente, è rilasciata in formato HTML e copre tutto l'insieme delle classi rilasciate con il SDK a cui da ora in poi ci riferiremo come alle Java Core API (Application Program Interface).

3.10 Scaricare ed installare il JDK

Questo testo fa riferimento alla versione dello Java Software Development Kit rilasciata dalla Javasoft nel corso del 2001: lo Java SDK 1.3.1. La SUN ha appena rilasciato va versione 1.4 dello SDK, ma nella trattazione, per la generalità dei suoi contenuti, si farà riferimento alla versione 1.3.1.
Il java SDK e tutta la sua documentazione possono essere scaricati gratuitamente dal sito della Javasoft all'indirizzo:

- *http://java.sun.com/j2se/1.3/download-solaris.html* per sistemi operativi Solaris;
- *http://java.sun.com/j2se/1.3/download-linux.html* per il sistema operativo Linux;
- *http://java.sun.com/j2se/1.3/download-windows.html* per tutti i sistemi operativi Microsoft.

L'installazione del prodotto è semplice e nel caso di sistemi operativi Microsoft, richiede solo l'esecuzione di un file auto-installante. Per semplicità, da questo momento in poi faremo riferimento al sistema operativo windows xp.
Al momento dell'istallazione, a meno di specifiche differenti, lo JSDK crea sul disco principale del computer la cartella "jdk1.3.1_02" all'interno della quale istallerà tutto il necessario al programmatore. Dentro questa cartella saranno create le seguenti sottocartelle:

c:\jdk1.3.1_02\bin contenente tutti i comandi java per compilare, le utility di servizio, oltre che ad una quantità di librerie *dll* di utilità varie.

c:\jdk1.3.1_02\demo contenente molti esempi comprensivi di eseguibili e codici sorgenti;

c:\jdk1.3.1_02\include contenente alcune librerie per poter utilizzare chiamate a funzioni scritte in C o C++;

c:\jdk1.3.1_02\lib contenente alcune file di libreria tra cui tools.jar contenente le Java Core API rilasciate da SUN ;

c:\jdk1.3.1_02\src contenente il codice sorgente delle classi contenute nell'archivio tools.jar.

Terminata l'installazione dello JSDK, è arrivato il momento di dedicarci alla documentazione preventivamente scaricata dal sito della Javasoft. La documentazione è archiviata in un file compresso con estensione ".zip"; sarà quindi necessario uno strumento in grado di decomprimerne il contenuto. Uno strumento molto comune per i sistemi operativi Microsoft è Winzip, che può essere scaricato dal sito www.winzip.com.
Salviamo il contenuto dell'archivio all'interno della cartella principale di installazione dello JSDK; alla fine del processo i documenti saranno salvati nella sottocartella ***c:\jdk1.3.1_02\docs***.

3.11 Il compilatore Java (javac)

Il compilatore Java (*javac*) fornito con lo JSDK, può essere trovato nella cartella ***c:\jdk1.3.1_02\bin***. Questo programma a linea di comando è un'applicazione interamente scritta in Java che, da un file con estensione ".java" produce un file con estensione ".class" contenente il Bytecode relativo alla definizione della classe che stiamo compilando. Un file con estensione ".java" è un normale file in formato ASCII contenente del codice Java valido;
il file generato dal compilatore Java potrà essere caricato ed eseguito dalla Java Virtual Machine.
Java è un linguaggio "case sensitive" ovvero sensibile al formato, maiuscolo o minuscolo, di un carattere: di conseguenza è necessario passare al compilatore java il parametro contenente il nome del file da compilare rispettandone il formato dei caratteri.
Ad esempio, volendo compilare il file "MiaPrimaApplicazione.java" la corretta esecuzione del compilatore java è la seguente:

javac MiaPrimaApplicazione.java

Qualunque altro formato produrrà un errore di compilazione, ed il processo di generazione del Bytecode sarà interrotto.
Un altro accorgimento da rispettare al momento della compilazione riguarda l'estensione del file da compilare: la regola vuole che sia sempre scritta esplicitamente su riga di comando.
Ad esempio la forma utilizzata di seguito per richiedere la compilazione di una file non è corretta e produce l'interruzione immediata del processo di compilazione:

javac MiaPrimaApplicazione

Il processo di compilazione di una file sorgente avviene secondo uno schema temporale ben definito. In dettaglio; quando il compilatore ha bisogno della definizione di una classe java per procedere nel processo di generazione del Bytecode esegue una ricerca utilizzando le informazioni contenute nella variabile di ambiente CLASSPATH. Questo meccanismo di ricerca può produrre tre tipi di risultati:

1. Trova soltanto un file ".class" : il compilatore lo utilizza;
2. Trova soltanto un file ".java" : il compilatore lo compila ed utilizza il file ".class";
3. Trova entrambi i file ".class" e ".java" : il compilatore java verifica se il file ".class" sia aggiornato rispetto al relativo file ".java" comparando le date dei due file. Se il file ".class" è aggiornato il compilatore lo utilizza e procede, altrimenti compila il file ".java" ed utilizza il file ".class" prodotto per procedere.

3.12 Opzioni standard del compilatore

Volendo fornire una regola generale, la sintassi del compilatore Java è la seguente:

javac *[opzioni] [filesorgenti] [@listadifiles]*

Oltre al file da compilare definito dall'opzione *[filesorgenti]*, il compilatore java accetta dalla riga di comando una serie di parametri opzionali (*[opzioni]*) necessari al programmatore a modificare le decisioni adottate dalla applicazione durante la fase di compilazione.
Le opzioni più comunemente utilizzate sono le seguenti:

-classpath *classpath*: ridefinisce o sostituisce il valore della variabile d'ambiente CLASSPATH. Se nessuno dei due meccanismi viene utilizzato per comunicare al compilatore la posizione della classi necessarie alla produzione del Bytecode, verrà utilizzata la cartella corrente;

-d *cartella*: imposta la cartella di destinazione all'interno della quale verranno memorizzati i file contenenti il Bytecode delle classi compilate. Se la classe fa parte di un package[3] Java, il compilatore salverà i file con estensione ".class" in una sottocartella che rifletta la struttura del package e creata a partire dalla cartella specificata dall'opzione. Ad esempio, se la classe da compilare si chiama *esercizi.primo.MiaPrimaApplicazione* e viene specificata la cartella *c:\src* utilizzando l'opzione **–d**, il file prodotto si chiamerà:

[3] Archivi di file Java compilati, di cui parleremo esaustivamente nei capitoli successivi.

c:\src\esercizi\primo\MiaPrimaApplicazione.class

Se questa opzione non viene specificata, il compilatore salverà i file contenenti il Bytecode all'interno della cartella corrente.
Se l'opzione -**sourcepath** non viene specificata, questa opzione viene utilizzata dal compilatore per trovare sia file ".class" che file ".java".

- sourcepath *sourcepath*: indica al compilatore la lista delle cartelle o dei package contenenti i file sorgenti necessari alla compilazione. Il formato di questo parametro rispecchia quello definito per la variabile di ambiente CLASSPATH.

3.13 Compilare elenchi di classi

Nel caso in cui sia necessario compilare un gran numero di file, il compilatore Java prevede la possibilità di specificare il nome di un file contenente la lista delle definizioni di classe scritte nell'elenco una per ogni riga.
Questa possibilità è utile non solo a semplificare la vita al programmatore, ma raggira il problema delle limitazioni relative alla lunghezza della riga di comando sui sistemi operativi della Microsoft.
Per utilizzare questa opzione è sufficiente indicare al compilatore il nome del file contenente l'elenco delle classi da compilare anteponendo il carattere '@'.
Ad esempio se ElencoSorgenti è il file in questione, la riga di comando sarà la seguente:

javac @ElencoSorgenti

3.14 Compilare una classe java

E' finalmente arrivato il momento di compilare la nostra prima applicazione java. Per eseguire questo esempio è possibile utilizzare qualsiasi editor di testo come notepad o wordpad per creare la definizione di classe il cui codice è riportato di seguito.

```
public class MiaPrimaApplicazione {
   public static void main(String[] argv) {
      System.out.println("Finalmente la mia prima applicazione");
   }
}
```

Creiamo sul disco principale la cartella mattone e le due sottocartelle C:\mattone\src e C:\mattone\classes.
Salviamo il file nella cartella c:\mattone\src\ facendo attenzione che il nome sia MiaPrimaApplicazione.java.
Utilizzando il menu "Avvia" di windows eseguiamo il "command prompt" e posizioniamoci nella cartella in cui abbiamo salvato il file (Figura 22).

Figura 22: Command Prompt

E' arrivato il momento di compilare il file ".java" con il comando

javac –classpath .;.\; -d c:\mattone\classes MiaPrimaApplicazione.java

dove l'opzione "*–classpath .;.\;*" indica che i file si trovano della cartella corrente e l'opzione "*–d c:\mattone\classes*" indica al compilatore di salvare il file contenente il Bytecode prodotto con il nome

c:\mattone\classes\MiaPrimaApplicazione.class

La figura 23 mostra i risultati della compilazione ed il contenuto della cartella c:\mattone\classes.

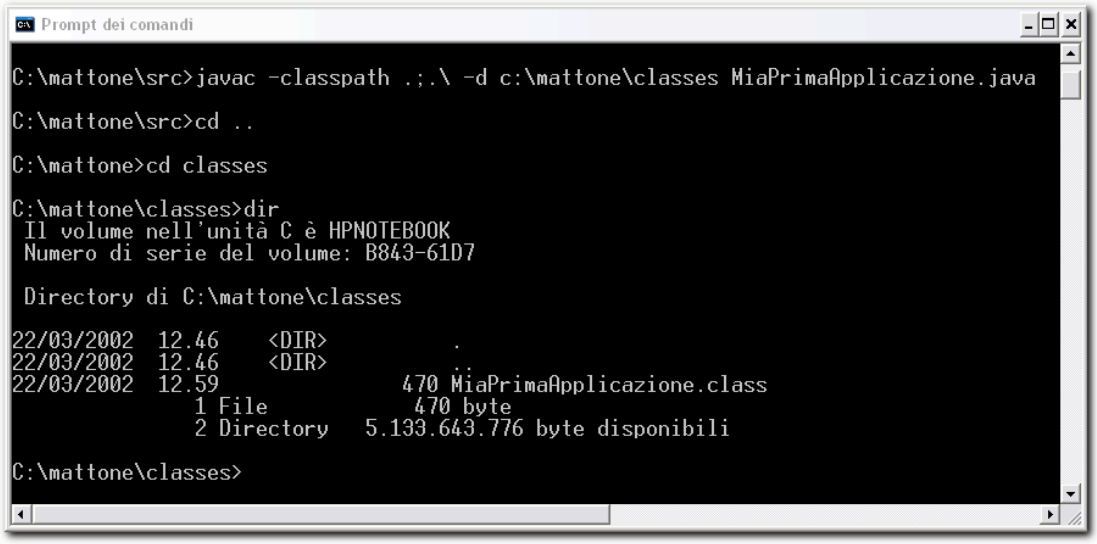


Figura 23: Contenuto della cartella c:\mattone\classes

3.15 L'interprete Java

Per eseguire l'applicazione appena compilata è necessario utilizzare l'interprete Java o JVM che si può trovare nella cartella c:\jdk1.3.1_02\bin\java.
Utilizzando il "command prompt" di window precedentemente eseguito, posizioniamoci nella cartella *c:\mattone\classes* ed eseguiamo l'applicazione digitando il comando seguente:

java MiaPrimaApplicazione

La figura seguente riporta il risultato dell'esecuzione dell'applicazione.

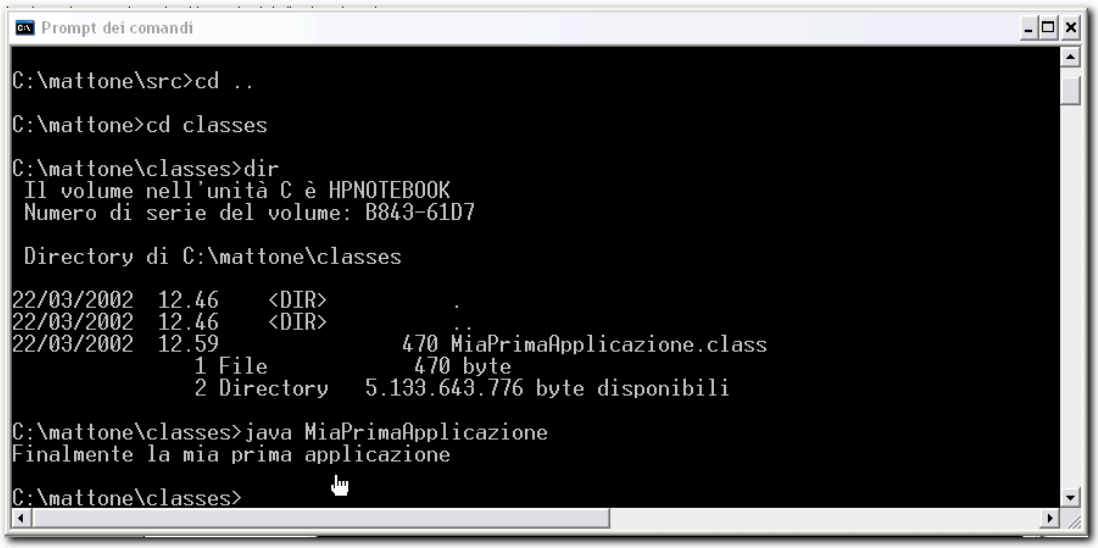


Figura 24: Esecuzione della applicazione

L'interprete Java, come il compilatore, è sensibile al formato dei caratteri ed è quindi indicare il nome dell'applicazione da eseguire rispettando il formato, maiuscolo o minuscolo, dei caratteri che compongono il nome del file.
A differenza dal compilatore, la JVM non richiede che sia specificata l'estensione del file da eseguire: la riga di comando

java MiaPrimaApplicazione.class

produrrà un errore segnalando al programmatore di non riuscire a trovare la classe MiaPrimaApplicazione/class (capiremo meglio in seguito il significato di questo messaggio di errore).

3.16 Sintassi dell'interprete java

La sintassi generale del comando java è la seguente:

java *[opzioni] class [argomenti ...]*

Le opzioni consentono al programmatore di influenzare l'ambiente all'interno del quale l'applicazione verrà eseguita. Le opzioni comunemente utilizzate sono le seguenti.

-classpath *classpath* : ridefinisce o sostituisce il valore della variabile d'ambiente CLASSPATH. Se nessuno dei due meccanismi viene utilizzato per comunicare al compilatore la posizione della classi necessarie alla esecuzione del Bytecode, verrà utilizzata la cartella corrente;
-verbose:class : Visualizza le informazioni relative ad ogni classe caricata;
-verbose:gc : Visualizza informazioni relative agli eventi scatenati dal garbage collector.
-version : Visuali le informazioni relative alla versione del prodotto ed esce;
-showversion : Visuali le informazioni relative alla versione del prodotto e prosegue;
-? | -help : Visualizza le informazioni sull'uso della applicazione ed esce;

L'interprete java consente inoltre di impostare un elenco di argomenti che possono essere utilizzati dalla applicazione stessa durante l'esecuzione.
Nel prossimo esempio viene mostrato come trasmettere argomenti ad una applicazione java.

```
public class TestArgomenti {
   public static void main(String[] argv) {
      for(int i=0; i<argv.length; i++)
         System.out.println("Argomento "+ (i+1) +" = "+ argv[i] );
   }
}
```

Dopo aver compilato la classe, eseguendo l'applicazione utilizzando la riga di comando seguente:

java TestArgomenti primo secondo terzo quarto

il risultato prodotto sarà il seguente:

Argomento 1 = primo
Argomento 2 = secondo
Argomento 3 = terzo
Argomento 4 = quarto

3.17 Inserire commenti

Commentare correttamente il codice sorgente di un'applicazione è importante primo, perché un codice ben commentato è facilmente leggibile da chiunque; secondo, perché i commenti aiutano il programmatore ad evidenziare aspetti specifici di un algoritmo riducendo sensibilmente gli errori dovuti a distrazioni in fase di scrittura; tuttavia i commenti sono spesso insufficienti e talvolta assenti.
Java consente di inserire commenti supportando entrambi i formati di C e C++: il primo include i commenti all'interno di blocchi di testo delineati dalle stringhe /* ed */, il secondo utilizza la stringa // per indicare una linea di documentazione. Nel prossimo esempio abbiamo modificato l'applicazione precedentemente scritta inserendo all'interno commenti utilizzando entrambe le forme descritte:

```
public class MiaPrimaApplicazione {

/*
        Il metodo main rappresenta il punto di ingresso
        all'interno della applicazione MiaPrimaApplicazione
*/
        public static void main(String[] argv) {

        // Visualizza sullo schermo il messaggio
        //  Finalmente la mia prima applicazione

        System.out.println("Finalmente la mia prima applicazione");
   }
}
```

Al momento della generazione del Bytecode, il compilatore Java legge il codice sorgente ed elimina le righe od i blocchi di testo contenenti commenti. Il Bytecode prodotto sarà identico a quello prodotto dalla stessa applicazione senza commenti.

3.18 Javadoc

Un errore che spesso si commette durante lo sviluppo di un'applicazione è dimenticarne o tralasciarne volutamente la documentazione tecnica. Documentare il codice del prodotto che si sta sviluppando, richiede spesso giorni di lavoro e di conseguenza costi che pochi sono disposti a sostenere: il risultato è un prodotto poco o addirittura completamente non mantenibile.
Java fonde gli aspetti descrittivo e documentale, consentendo di utilizzare i commenti inseriti dal programmatore all'interno del codice sorgente dell'applicazione per produrre la documentazione tecnica necessaria ad una corretto rilascio di un prodotto.
Oltre ai formati descritti nel paragrafo precedente, Java prevede un terzo formato che utilizza le stringhe /** e */ per delimitare blocchi di commenti

chiamati "doc comments". I commenti che utilizzano questo formato sono utilizzati da uno strumento fornito con il Java SDK per generare automaticamente documentazione tecnica di una applicazione in formato ipertestuale.
Questo strumento è chiamato *javadoc* e può essere trovato all'interno della cartella *c:\jdk1.3.1_02\bin\javadoc*.
Javadoc esegue la scansione del codice sorgente, estrapola le dichiarazioni delle classi ed i "doc comments" e produce una serie di pagine HTML contenenti informazioni relative alla descrizione della classe, dei metodi e dei dati membri più una completa rappresentazione della gerarchia delle classi e le relazioni tra loro.
Oltre ai commenti nel formato descritto, questo strumento riconosce alcune etichette utili all'inserimento, all'interno della documentazione prodotta, di informazioni aggiuntive come l'autore di una classe o la versione. Le etichette sono precedute dal carattere '@': le più comuni sono elencate nella tabella seguente:

Etichette Javadoc		
Sintassi	**Descrizione**	**Applicato**
@see *riferimento*	Aggiunge una voce "see also" con un link ad una classe descritta dal parametro.	Classi, metodi, variabili.
@author *nome*	Aggiunge una voce "Author" alla documentazione. Il parametro descrive il nome dell'autore della classe.	Classe.
@version *versione*	Aggiunge una voce "Version" alla documentazione. Il parametro contiene il numero di versione della classe.	Classe
@param *parametro*	Aggiunge la descrizione ad un parametro di un metodo .	Metodo
@return *descrizione*	Aggiunge la descrizione relativa al valore di ritorno di un metodo.	Metodo
@since *testo*	Aggiunge una voce "Since" alla documentazione. Il parametro identifica generalmente il numero di versione della classe a partire dalla quale il nuovo requisito è stato aggiunto.	Classi, Metodi, variabili
@deprecated *testo*	Imposta un requisito come obsoleto.	Classi, Metodi, variabili

@throws *classe descrizione*	Aggiunge una voce "Throws" alla definizione di una metodo contenente il riferimento ad una eccezione generata e la sua descrizione.	Metodi
@exception *classe descrizione*	Vedi ***@throws***.	Metodi

Possiamo modificare la classe MiaPrimaApplicazione per esercitarci nell'uso di questo strumento: è comunque consigliabile modificare ulteriormente l'esempio seguente con lo scopo di provare tutte le possibili combinazioni di etichette e verificarne il risultato prodotto.

```
/**
 *        La classe <STRONG>MiaPrimaApplicazione</STRONG> contiene solo il metodo
 *        main che produce un messaggio a video.
 *        @version 0.1
*/
public class MiaPrimaApplicazione {

/**
*         Il metodo main rappresenta il punto di ingresso
*         all'interno della applicazione MiaPrimaApplicazione
*        @since 0.1
*         <PRE>
*         @param String[] : array contenente la lista dei parametri di input <br> passati per
*         riga di comando.
*         </PRE>
*         @return void
*/

public static void main(String[] argv) {

          // Visualizza sullo schermo il messaggio
          //  Finalmente la mia prima applicazione
          System.out.println("Finalmente la mia prima applicazione");
   }
}
```

L'esempio evidenzia un altro aspetto dei "doc comment" di Java: di fatto, è possibile inserire parole chiavi di HTML per modificare il formato della presentazione dei documenti prodotti.
Prima di eseguire il comando sulla classe appena modificata, creiamo la cartella C:\mattone\docs\MiaPrimaApplicazione, eseguiamo il "prompt dei comandi" di windows e posizioniamoci nella cartella contenente il file sorgente dell'applicazione da documentare.
Eseguiamo il comando

javadoc -d C:\mattone\docs\MiaPrimaApplicazione MiaPrimaApplicazione.java

come mostrato nella prossima figura.

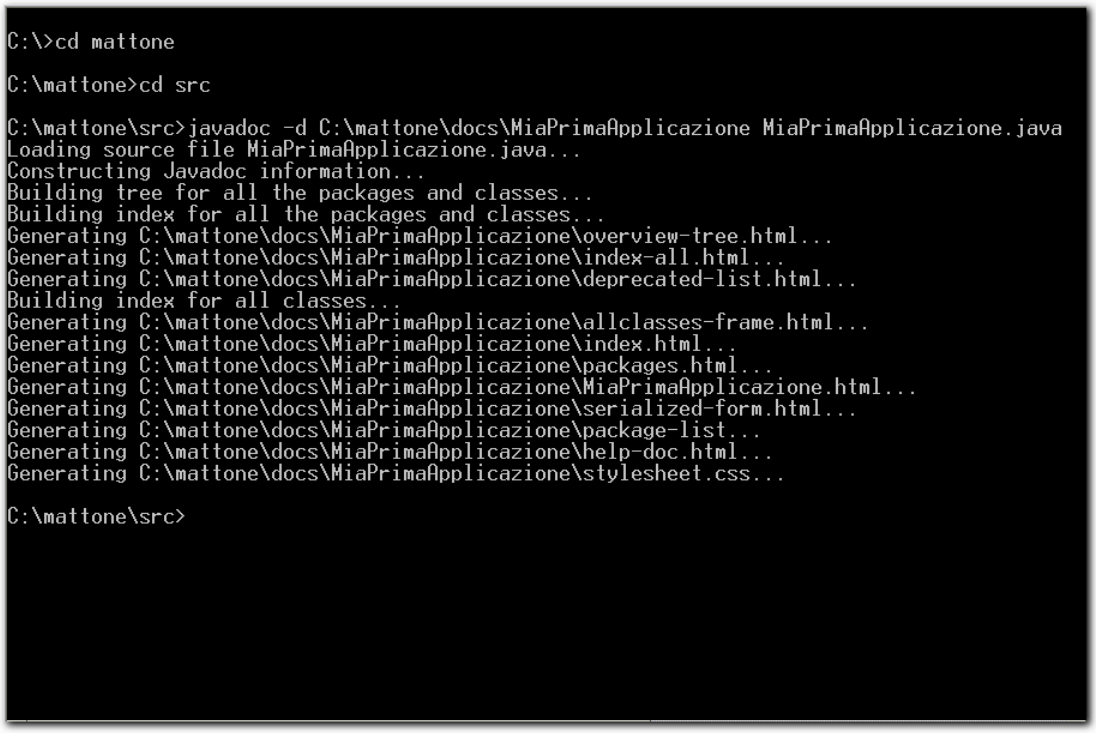


Figura 25: Esecuzione del comando javadoc

Al termine della esecuzione del comando aprite il file C:\mattone\docs\MiaPrimaApplicazione\index.html per verificarne i contenuti.

3.19 Sintassi del comando javadoc

Javadoc accetta parametri tramite la riga di comando secondo la sintassi generale:

javadoc *[opzioni] [packages] [filesorgenti] [@elenco]*

In dettaglio: le *opzioni* consentono di modificare le impostazioni standard dello strumento, *packages* rappresenta un elenco di nomi validi di packages che si desidera documentare separati da uno spazio (ad esempio java.lang o java.math), *filesorgenti* è una la lista di file sorgenti da documentare nella forma nomefile.java separati dal carattere spazio, *@elenco* rappresenta il nome di un file contenente una lista di nominativi di file sorgenti o di packages, uno per ogni riga.
Le opzioni più comuni sono elencate di seguito:

-classpath *classpath*: ridefinisce o sostituisce il valore della variabile d'ambiente CLASSPATH. Se nessuno dei due meccanismi viene utilizzato per comunicare al compilatore la posizione della classi necessarie alla esecuzione del Bytecode, verrà utilizzata la cartella corrente;
-windowtitle *titolo*: imposta il titolo della documentazione HTML prodotta;

-d *cartella*: imposta la cartella di destinazione all'interno della quale verranno memorizzati i documenti prodotti.

3.20 Java Hotspot

Il maggior punto di forza di Java, la portabilità del codice tramite Bytecode, è anche il suo tallone d'Achille. In altre parole, essendo Java un linguaggio tradotto (ovvero necessita di una virtual machine per essere eseguito), le applicazioni sviluppate con questa tecnologia hanno prestazioni inferiori rispetto ad altri linguaggi di programmazione come il C++.
Finalmente la SUN sembra aver colmato la lacuna rilasciando, insieme al nuovo Java SDK 1.3.1, Hotspot, una JVM che, utilizzando tecniche avanzate d'ottimizzazione del codice durante l'esecuzione, affiancate da una più efficiente gestione della memoria e dal supporto nativo per i thread, fornisce prestazioni paragonabili a quelle di un linguaggio non tradotto.
Nella prossima figura viene schematizzata l'architettura di Hotspot:

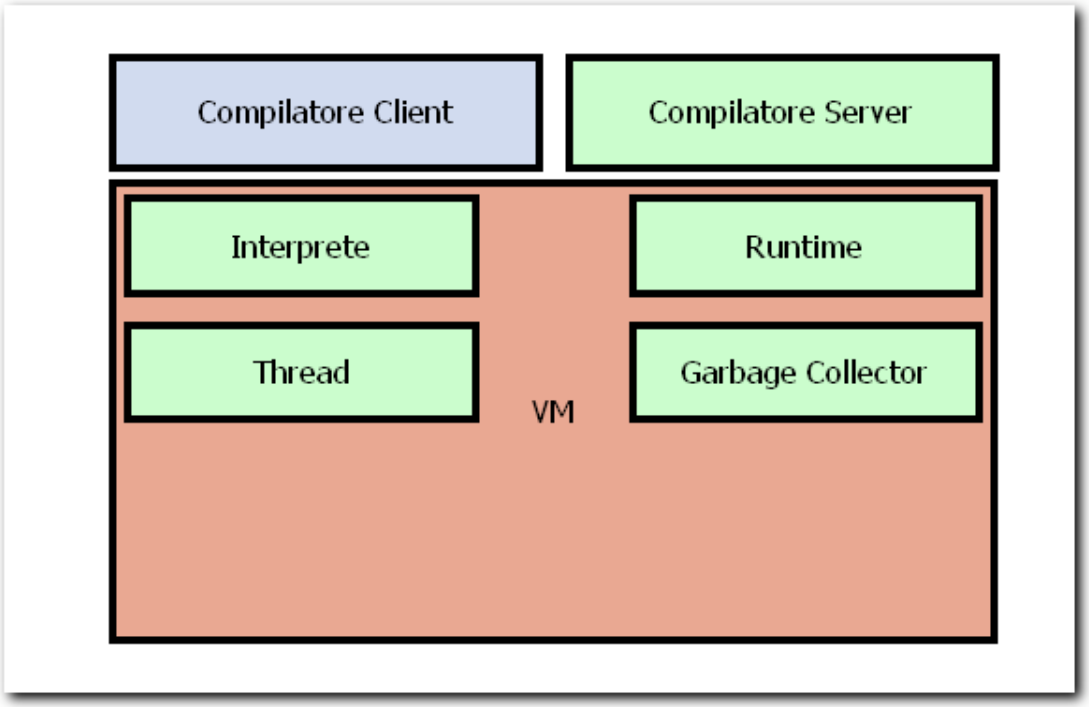


Figura 26: Architettura di Hotspot

Dallo schema appare subito evidente la prima grande differenza rispetto alle versioni precedenti: la presenza di due compilatori in grado di ottimizzare il codice a seconda che l'applicazione che si sta compilando sia di tipo client o server.
Nel primo caso, il Bytecode è ottimizzato per ridurre i tempi di caricamento e di avvio dell'applicazione ed ottimizzare l'uso della memoria; nel secondo caso, è privilegiata la velocità d'esecuzione e la robustezza dell'applicazione a discapito delle risorse di sistema.

Per selezionare il tipo di ottimizzazione da effettuare, il compilatore java accetta le due opzioni *-client* e *–server* da inviare dalla riga di comando dell'applicazione.

3.21 Il nuovo modello di memoria

Il nuovo modello per la gestione della memoria di Hotspot, è inteso a rendere più efficiente l'accesso alle istanze degli oggetti di cui l'applicazione fa uso, ed a minimizzare la quantità di memoria necessaria al funzionamento della JVM che dovrà eseguire il Bytecode.
Il primo cambiamento è stato introdotto nelle modalità con cui la JVM tiene traccia degli oggetti in memoria: un sistema di puntatori diretti alle classi in uso garantisce tempi di accesso paragonabili a quelli del linguaggio C.
Il secondo riguarda invece l'implementazione dei descrittori degli oggetti in memoria o "object header". Un descrittore di oggetto contiene tutte le informazioni relative alla sua identità, allo stato nel garbage collector, al codice hash: di fatto, a differenza della vecchia JVM che riservava tre parole macchina (32*3 bit) per il descrittore di un oggetto, Hotpost ne utilizza due per gli oggetti e tre solo per gli array, diminuendo significativamente la quantità di memoria utilizzata (la diminuzione media è stata stimata di circa 8% rispetto alla vecchia JVM).

3.22 Supporto nativo per i thread

Per rendere più efficiente la gestione della concorrenza tra thread, Hotspot demanda al sistema operativo che ospita l'applicazione la responsabilità dell'esecuzione di questi processi logici, affinché l'applicazione ne tragga benefici in fatto di prestazioni.
Per meglio comprendere quanto affermato, soffermiamoci qualche istante sul concetto di concorrenza tra processi logici.
Da punto di vista dell'utente, un thread è un processo eseguito parallelamente ad altri; dal punto di vista del calcolatore, non esistono processi paralleli.
Salvo che non possediate una macchina costosissima in grado di eseguire calcoli in parallelo, i processi si alternano tra loro con una velocità tale da dare l'impressione di essere eseguiti contemporaneamente. Questo meccanismo è detto di concorrenza: i processi attivi concorrono tra loro all'uso del processore.
Per realizzare il meccanismo di concorrenza tra processi, il sistema operativo organizza i processi attivi in una struttura dati detta "coda Round Robin[4]" (la figura 27 ne offre uno schema semplificato), da cui:

1. Il processo P5 possiede il controllo del processore della macchina per un periodo di tempo prestabilito;

[4] Round Robin è il nome dell'algoritmo più frequentemente utilizzato per realizzare la concorrenza logica tra processi o thread.

2. Alla scadenza del periodo di tempo prestabilito, il processo P5 viene inserito nella coda ed il processo P4 viene estratto dalla coda per assumere il controllo del processore;
3. Alla scadenza del periodo di tempo prestabilito, il processo P4 viene inserito nella coda ed il processo P3 viene estratto dalla coda per assumere il controllo del processore ecc. ecc..

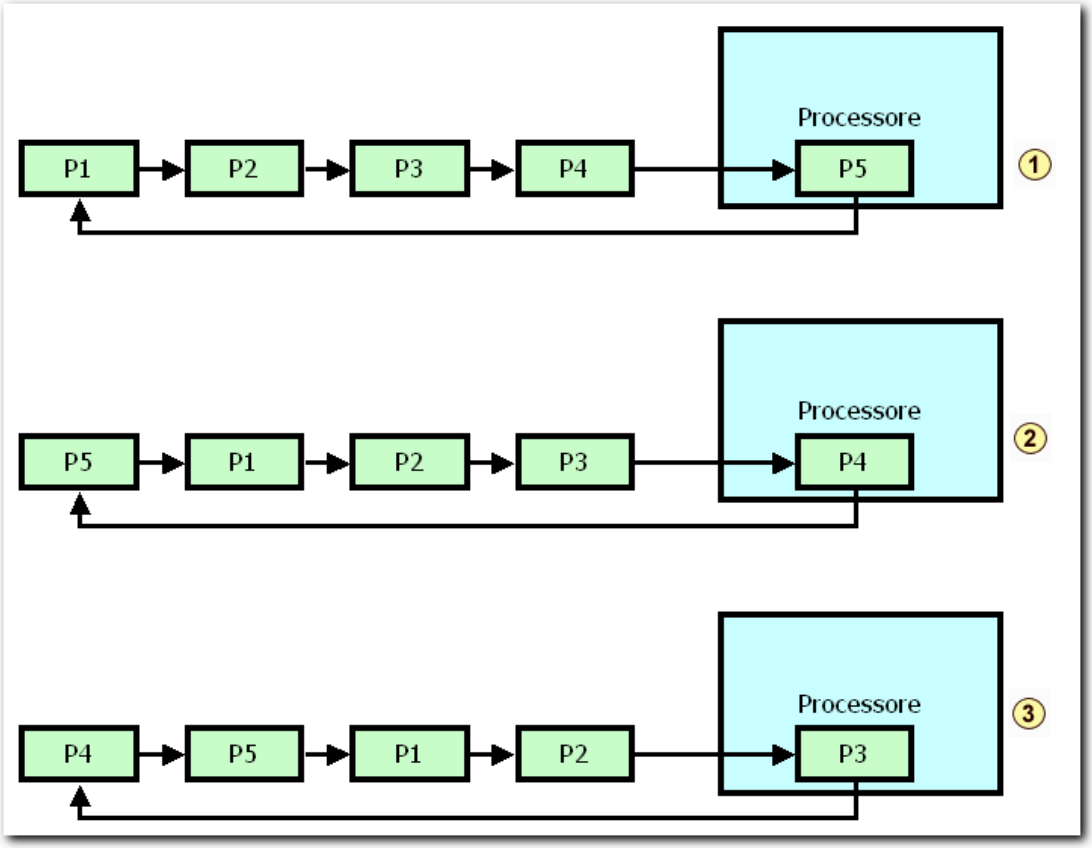


Figura 27: Concorrenza logica tra processi

Il sistema operativo è specializzato nella gestione della concorrenza tra processi e thread, ed è supportato da componenti elettroniche appositamente progettate per aumentare le prestazioni questi algoritmi.
Quando un processo prende il controllo del processore per proseguire nella esecuzione delle proprie istruzioni, necessita di una serie di dati (valori dei registri di sistema, riferimenti all'ultima istruzione eseguita, ecc. ecc.) salvati in una struttura in memoria sotto il diretto controllo del processore chiamata "stack di sistema" il cui funzionamento è schematizzato nella figura 28:

1. Il processo P3 prende il controllo del processore: il sistema operativo carica i dati di P3 nelle strutture di sistema ed il processo prosegue l'esecuzione per un intervallo di tempo prestabilito;
2. Terminato l'intervallo di tempo, il processo P3 rilascia il processore: il sistema operativo inserisce il processo in coda, estrae i dati relativi a

P3 dalle strutture di sistema e li salva per poterli recuperare successivamente;

3. Il processo P2 prende il controllo del processore: il sistema operativo carica i dati di P2 precedentemente salvati ed il processo prosegue l'esecuzione per un intervallo di tempo prestabilito.

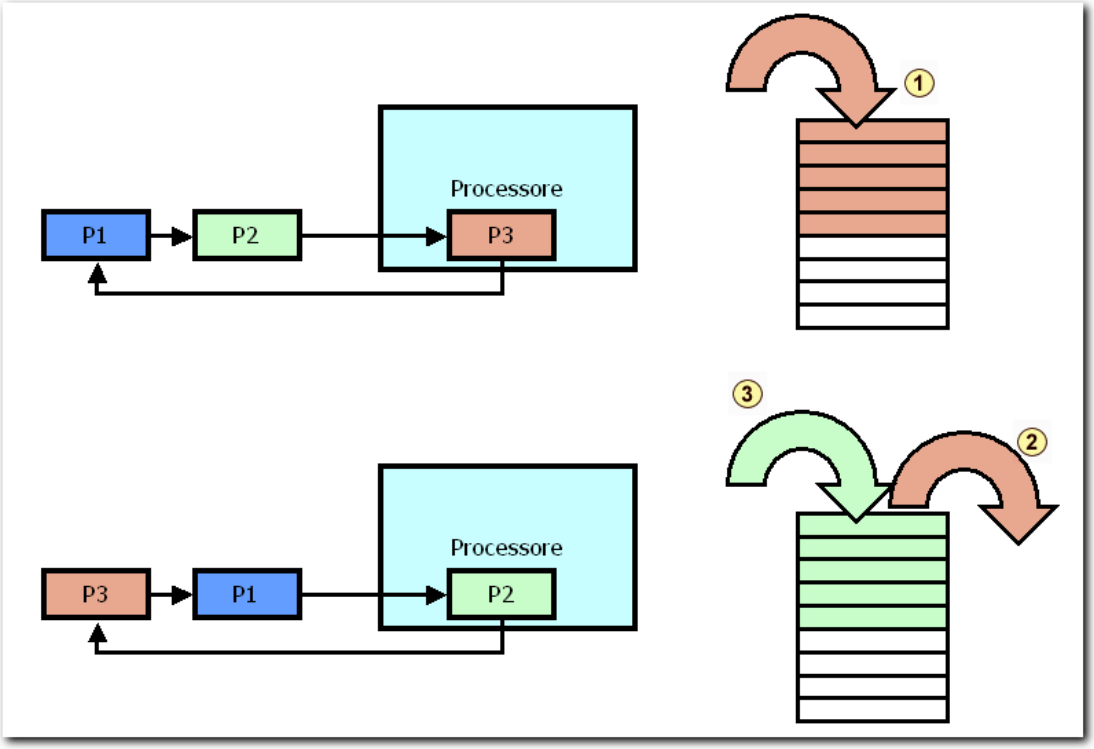


Figura 28: Stack di sistema

Questo meccanismo è a carico del sistema operativo, e rappresenta un collo di bottiglia in grado di far degradare notevolmente le prestazioni di un processo in esecuzione. Come i processi, anche i thread sono affetti da questo problema.
Anche in questo caso l'elettronica corre in aiuto del sistema operativo mettendo a disposizione componenti elettroniche appositamente progettate.
Concludendo, demandare al sistema operativo la gestione dei thread, non può che portare benefici incrementando notevolmente le prestazioni della JVM; al contrario la JVM si sarebbe dovuta far carico della implementazione e della gestione delle strutture necessarie a gestire la concorrenza tra questi processi.

3.23 Il garbage Collector di Hotspot

Il garbage collector di Hotspot è un'altra grande modica introdotta con la nuova JVM. Le caratteristiche del nuovo gestore della memoria sono tante e complesse; non essendo possibile descriverle tutte, esamineremo le più utili allo scopo che si propone questo libro.

Parlando del garbage collector, abbiamo già evidenziato alcuni limiti nel gestire e rilasciare la memoria nei casi in cui sia difficile determinare se un oggetto sia stato effettivamente rilasciato dall'applicazione o no.
Questo tipo di gestori della memoria sono comunemente detti "conservativi" o "parzialmente accurati", ed hanno il difetto di non riuscire a rilasciare completamente la memoria allocata da un'applicazione dando origine al problema del "memory leak".
Per essere più precisi, un garbage collector di tipo conservativo, non conosce esattamente dove siano posizionati tutti i puntatori agli oggetti attivi in memoria e di conseguenza, deve conservare tutti quegli oggetti che sembrano essere riferiti da altri: ad esempio potrebbe confondere una variabile intera con un puntatore e di conseguenza non rilasciare un oggetto che in realtà potrebbe essere rimosso.
Questa mancanza di accuratezza nel tener traccia dei puntatori agli oggetti allocati rende impossibile spostare gli oggetti in memoria causando un altro problema che affligge i garbage collector conservativi: la frammentazione della memoria.
Di fatto, il ciclo di esecuzione di una applicazione prevede che gli oggetti possano essere allocati o rimossi dalla memoria anche con una certa frequenza.

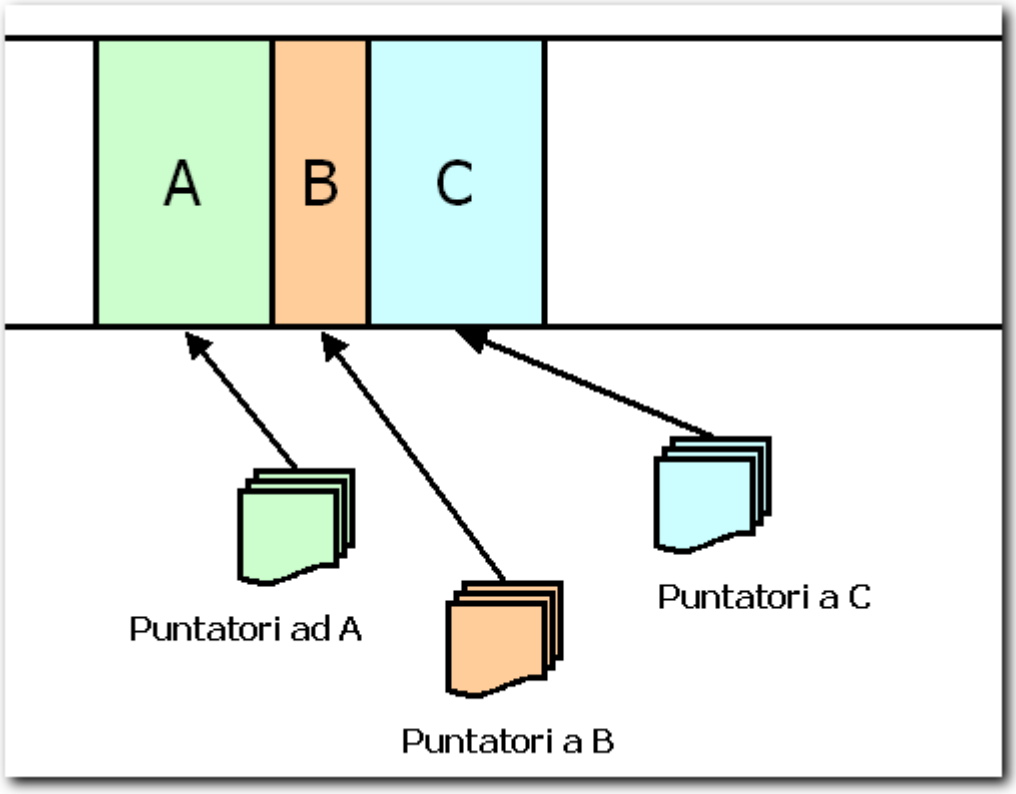


Figura 29: Allocazione della memoria per gli oggetti A,B,C

Supponiamo che ad un certo istante del ciclo di esecuzione di una applicazione risultino allocati gli oggetti *A,B,C* (Figura 29), e che il flusso delle

operazioni sia tale da rilasciare successivamente tutti i puntatori all'oggetto *B*: il garbage collector rimuoverà l'oggetto dalla memoria rilasciando le risorse utilizzate (Figura 30).

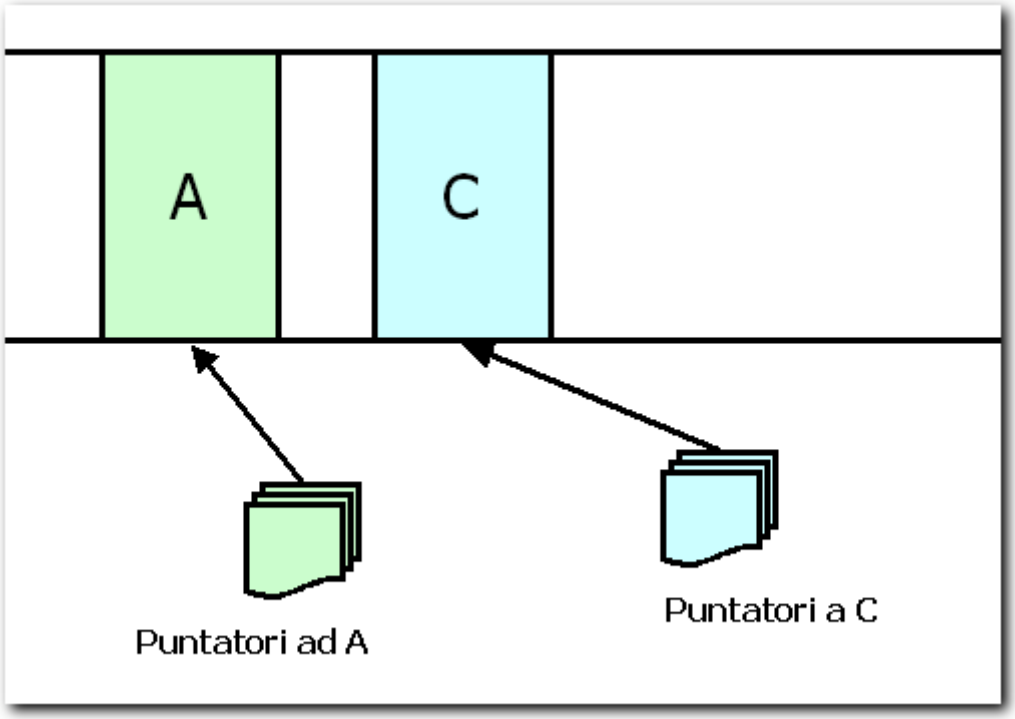


Figura 30: L'oggetto B è rimosso dalla memoria

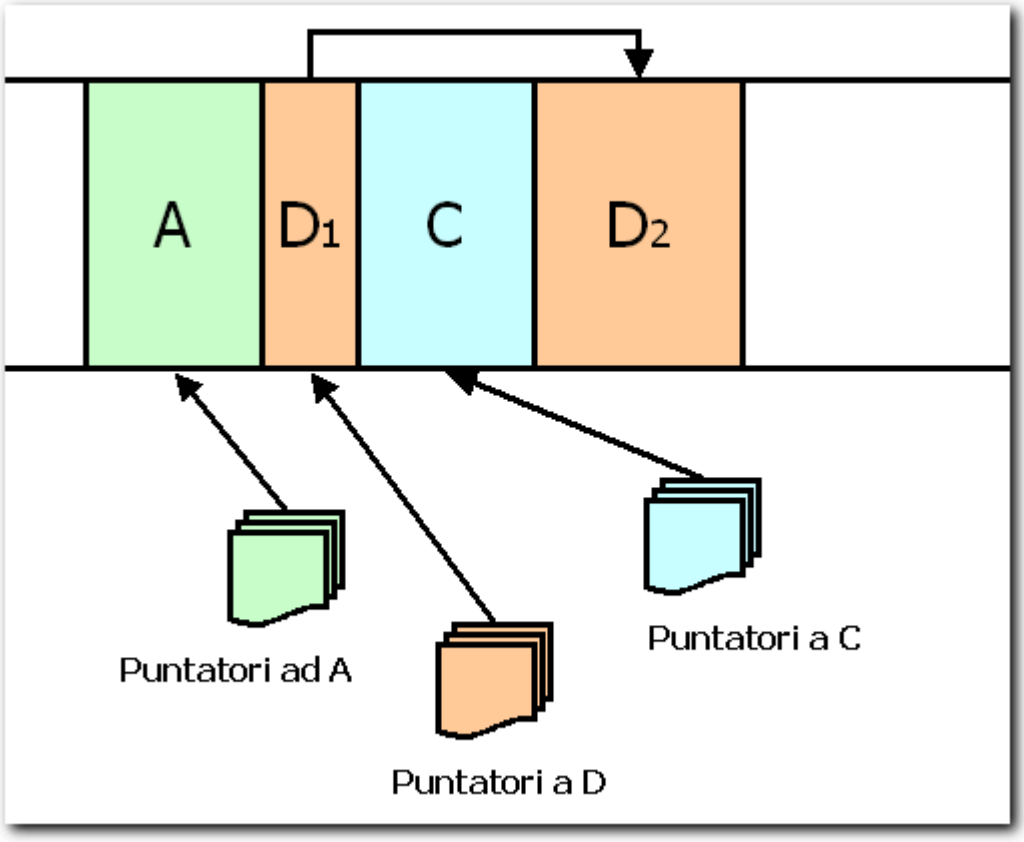


Figura 31: Frammentazione della memoria

Successivamente l'applicazione richiede risorse per allocare un nuovo oggetto *D* le cui dimensioni sono maggiori di quelle dell'oggetto appena rimosso: dal momento che gli oggetti non possono essere spostati in modo da occupare porzioni contigue di memoria, il nuovo oggetto potrebbe essere allocato utilizzando lo schema riportato nella figura 31 in cui vengono riservate due porzioni di memoria non contigue frammentando l'oggetto in più parti.
La frammentazione della memoria, nel caso di applicazioni molto complesse in cui esiste un gran numero di oggetti che interagiscono tra loro, può causare un degrado nelle prestazioni tale da necessitare il riavvio dell'applicazione.
A differenza degli altri, il garbage collector di Hotspot è "totalmente accurato". Una garbage collector di questo tipo conosce esattamente la situazione dei puntatori agli oggetti in ogni istante del ciclo di esecuzione della applicazione: questa caratteristica garantisce la corretta gestione delle risorse allocate agli oggetti e soprattutto ne consente la rilocazione evitando il problema della frammentazione.

4 LA SINTASSI DI JAVA

4.1 Introduzione

La sintassi del linguaggio Java eredita in parte quella di C e C++ per la definizione di variabili e strutture complesse, ed introduce il concetto di variabili di dimensioni fisse ovvero non dipendenti dalla piattaforma.
Nei prossimi paragrafi studieremo cosa sono le variabili, come utilizzarle e come implementare strutture dati più complesse mediante l'uso degli array.
Infine, introdurremo le regole sintattiche e semantiche specifiche del linguaggio necessarie alla comprensione del capitolo successivo.

4.2 Variabili

Per scrivere applicazioni Java, un programmatore deve poter creare oggetti. Per far questo, è necessario poterne rappresentare i dati. Il linguaggio Java mette a disposizione del programmatore una serie di "tipi semplici" o "primitivi", utili alla definizione di oggetti più complessi.
Per garantire la portabilità del Bytecode 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.

Variabili primitive 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:

tipo identificatore;

dove "*tipo*" è uno tra i tipi descritti nella prima colonna della tabella e definisce il dato che la variabile dovrà contenere, e l'identificatore rappresenta

il nome della variabile. Il nome di una 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 loro i nomi delle variabili:

tipo identificatore1, identificatore2, …. ;

Per convenzione l'identificatore di una variabile Java deve iniziare con la lettera minuscola.
L'identificatore di una variabile ci consente di accedere al valore del dato da qualunque punto del codice esso sia visibile e poiché ha valore puramente mnemonico, è bene che esso sia il più possibile descrittivo per consentire di identificare con poco sforzo il contesto applicativo all'interno del quale il dato andrà utilizzato. Ad esempio le righe di codice:

```
int moltiplicatore = 10;
int moltiplicando = 20;
int risultato;
risultato = moltiplicando* moltiplicatore;
```

rappresentano in maniera comprensibile una operazione di moltiplicazione tra due variabili di tipo intero.

4.3 Inizializzazione di una variabile

Durante l'esecuzione del Bytecode dell'applicazione, se la JVM trova la dichiarazione di una variabile, riserva spazio sulla memoria del computer per poterne rappresentare il valore, e ne associa l'indirizzo fisico all'identificatore per accedere al dato; di conseguenza, ogni variabile Java richiede che, al momento della dichiarazione, le sia assegnato un valore iniziale.
Java assegna ad ogni variabile un valore prestabilito: la tabella riassume il valore iniziale assegnato dalla JVM distinguendo secondo del tipo primitivo rappresentato.

Valori prestabiliti per le primitive	
Tipo primitivo	**Valore assegnato dalla JVM**
boolean	false
Char	'\u0000'
Byte	0
Short	0
Int	0
Long	0L
Float	0.0f
Double	0.0

In alternativa, come per il linguaggio C e C++, l'inizializzazione di una variabile Java può essere effettuata dal programmatore direttamente al momento della sua dichiarazione. La sintassi è la seguente:

tipo identificatore = valore;

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

```
int variabileintera = 100;
```

4.4 Variabili char e codifica del testo

Parlando della storia del linguaggio Java abbiamo evidenziato come la tecnologia proposta dalla SUN abbia avuto come campo principale di applicazione, Internet. Data la sua natura, era indispensabile dotarlo delle caratteristiche fondamentali a rispondere alle esigenze di internazionalizzazione proprie della più vasta rete del mondo.
La codifica ASCII, largamente utilizzata dalla maggior parte dei linguaggi di programmazione, utilizza una codifica ad otto bit idonea a rappresentare al massimo 2^8 =256 caratteri e quindi non adatta allo scopo.
Fu quindi adottato lo standard internazionale UNICODE, che utilizza una codifica a sedici bit studiata per rappresentare 2^{16}= 65536 caratteri di cui i primi 256 (UNICODE 0-255) corrispondono ai caratteri ASCII.
Per questo motivo, le variabili Java di tipo char (carattere) utilizzano sedici bit per rappresentare il valore del dato, e le stringhe a loro volta sono rappresentate come sequenze di caratteri a sedici bit.
Questa caratteristica di Java rappresenta però una limitazione per tutti i sistemi informativi non in grado di trattare caratteri codificati secondo lo standard UNICODE. Per risolvere il problema, Java utilizza la forma detta "sequenza di escape"

\u*xxxx* (xxxx=sequenza di massimo 4 cifre esadecimali)

per produrre l'output dei caratteri e delle stringhe.
La sequenza di escape descritta indica una carattere UNICODE decodificato in ASCII.

4.5 Variabili final

A differenza di altri linguaggi di programmazione, Java non consente la definizione di costanti. Questo aspetto del linguaggio non è da considerarsi

una limitazione perché è possibile simulare una costante utilizzando il modificatore **final**.
Le variabili dichiarate **final** si comportano come una costante e richiedono che sia assegnato il valore al momento della dichiarazione, utilizzando l'operatore di assegnamento:

final *tipo identificatore = valore;*

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

4.6 Operatori

Una volta definito, un oggetto deve poter manipolare i dati. Java mette a disposizione del programmatore una serie di operatori utili allo scopo.
Gli operatori Java sono elencati nella tabella seguente, ordinati secondo l'ordine di precedenza: dal più alto al più basso.

Operatori Java	
Operatore	**Funzioni**
++ -- + - ☐	Aritmetiche unarie e booleane
* / %	Aritmetiche
+ -	Addizione (o concatenazione) e sottrazione
<< >> >>>	Shift di bit
(***tipo***)	Operatore di cast
< <= > >= ***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

Gran parte degli operatori Java appartengono all'insieme degli operatori del linguaggio C, cui ne sono stati aggiunti nuovi a supporto delle caratteristiche proprie del linguaggio della SUN.
Gli operatori elencati nella tabella 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 "!=" (diverso da) di cui si nota la disuguaglianza con gli stessi operatori come definiti dall'algebra: "=" e "<>": l'uso dell'operatore digrafo[5] "==" è necessario dal momento che il carattere "=" è utilizzato esclusivamente come operatore di assegnamento, e l'operatore "!=" compare in questa forma per consistenza con la definizione dell'operatore logico "!" (NOT).
Anche gli operatori "bit a bit" e quelli logici derivano dal linguaggio C e nonostante le analogie in comune, sono completamente sconnessi tra loro. Ad esempio l'operatore "&" è utilizzato per combinare due interi operando 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, l'espressione (a == a && b != b) restituirà false.
La differenza con il linguaggio C sta nel fatto che gli operatori logici in Java sono di tipo "**short-circuit**" ovvero, se il lato sinistro di un'espressione fornisce informazioni sufficienti a completare l'intera operazione, il lato destro dell'espressione non sarà valutato.
Per esempio, si consideri l'espressione booleana

```
( 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 )
```

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

dove f() è una funzione arbitrariamente complessa, f() non sarà eseguita se A non è uguale a B.
Sempre dal linguaggio C, Java eredita gli operatori unari di incremento e decremento "++" e --:

i++ equivale a i=i+1
i-- equivale a i=i-1.

Infine gli operatori di combinazione, combinano un assegnamento con un'operazione aritmetica:

i*=2 equivale ad i=i*2.

Questi operatori anche se semplificano la scrittura del codice lo rendono di difficile comprensione, per questo motivo non sono comunemente utilizzati.

4.7 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:

tipo identificatore = espressione;

dove '*espressione*' rappresenta una qualsiasi espressione che produce un valore compatibile con il tipo definito da '*tipo*', e '*identificatore*' rappresenta la variabile che conterrà il risultato. Tornando 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.
Ad esempio:

```
int risultato = 5+10;
```

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

```
int risultato = 5;
```

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;

4.8 Operatore di cast

L'operatore di cast tra tipi primitivi consente di promuovere, durante l'esecuzione di un'applicazione, un tipo numerico in uno nuovo. La sintassi dell'operatore è la seguente:

(nuovo tipo) identificatore|espressione

dove "*nuovo tipo*" rappresenta il tipo dopo la conversione, "*identificatore*" è una qualsiasi variabile numerica o carattere, "*espressione*" un'espressione che produca un valore numerico.
Prima di effettuare un'operazione di cast tra tipi primitivi, è necessario assicurarsi che non ci siano eventuali errori di conversione.
Di fatto, il cast di un tipo numerico in un altro con minor precisione, ha come effetto di modificare il valore del tipo con precisione maggiore affinché possa essere memorizzato in quello con precisione minore: ad esempio, il tipo ***long*** può rappresentare tutti i numeri interi compresi nell'intervallo (-2^{63} , $+2^{63}-1$), mentre il tipo ***int*** quelli compresi nell'intervallo (-2^{32} , $+2^{32}-1$).
La prossima applicazione di esempio effettua tre operazioni di cast: il primo tra una variabile di tipo ***long*** in una di tipo ***int*** evidenziando la perdita del valore del tipo promosso; il secondo di un tipo ***int*** in un tipo ***long***; il terzo, di un tipo ***char*** in un tipo ***int*** memorizzando nella variabile intera il codice UNICODE del carattere 'A'.

```
public class Cast {
   public static void main(String[] argv) {
      long tipolong;
      int tipoint;
      char tipochar;

      //Cast di un tipo long in un tipo int
      tipolong = Long.MAX_VALUE;
      tipoint = (int) tipolong;
```

```
        System.out.println("La variabile di tipo long vale: "+tipolong+", La variabile di tipo int vale: "+tipoint);

        //Cast di un tipo int in un tipo long
        tipoint = Integer.MAX_VALUE;
        tipolong = (long) tipoint;
        System.out.println("La variabile di tipo long vale: "+tipolong+", La variabile di tipo int vale: "+tipoint);

        //Cast di un tipo char in un tipo int
        tipochar='A';
        tipoint = (int)tipochar;
        System.out.println("La variabile di tipo char vale: "+tipochar+", La variabile di tipo int vale: "+tipoint);
    }
}
```

Il risultato della esecuzione del codice è riportato di seguito.

```
La variabile di tipo long vale: 9223372036854775807, La variabile di tipo int vale: -1
La variabile di tipo long vale: 2147483647, La variabile di tipo int vale: 2147483647
La variabile di tipo char vale: A, La variabile di tipo int vale: 65
```

4.9 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

Esaminiamo ora le seguenti 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 si possa assegnarle il risultato dell'operazione. Essendo 1.500.000.000.000 troppo grande perché sia assegnato ad una variabile di tipo int, sarà necessario utilizzarne una di un tipo in grado di contenere correttamente il valore prodotto.
Il 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, è effettuato da Java seguendo alcune regole ben precise:

- *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*" (scorciatoia). Questo tipo di operatori consente l'incremento o il decremento di uno come riassunto nella tabella:

Operatori shortcut		
Forma shortcut	**Forma estesa**	**Risultato**

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 i=1;	i=0

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

4.10 Operatori relazionali

Gli "operatori relazionali" sono detti tali perché si riferiscono alle possibili relazioni tra valori, producendo un risultato di verità o falsità come conseguenza del confronto.
A differenza dei linguaggi C e C++ in cui vero o falso corrispondono rispettivamente con i valori 0 e ≠0 restituiti da un'espressione, Java li identifica rispettivamente con i valori **true** e **false,** detti booleani e rappresentati da variabili di tipo **boolean**.
Nella tabella seguente sono riassunti gli operatori relazionali 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.

4.11 Operatori logici

Gli "operatori logici" 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 seguente.

Operatori Condizionali		
Operatore	**Utilizzo**	**Descrizione**
&&	res=sx && dx	AND : res = true se e solo se sx e dx vagono entrambi 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.

Tutti i possibili valori booleani prodotti dagli operatori descritti possono essere schematizzati mediante le "tabelle di verità". Le "tabelle di verità" forniscono, per ogni operatore, tutti i possibili risultati secondo il valore degli operandi.

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

Ricapitolando:

- *L'operatore "&&" è un operatore binario e restituisce vero soltanto se entrambi gli operandi sono veri;*
- *L'operatore "||" è un operatore binario e restituisce vero se almeno uno dei due operandi è vero;*
- *L'operatore "!" è un operatore unario che afferma la negazione dell'opernado;*
- *L'operatore "^" è un operatore binario e restituisce vero se solo uno dei due operandi è vero;*

Come vedremo in seguito, gli operatori relazionali, agendo assieme agli operatori logici, forniscono uno strumento di programmazione molto efficace.

4.12 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 il seguente esempio:

```
public class ProdottoDivisione {
   public static void main(String args[]) {
      int i = 100;
      int j=i;
     //Applicazione dello shift bit a bit verso destra
      i=i >> 1;
      System.out.println("Il risultato di "+j+" >> 1 e': " + i);
      j=i;
      i=i >> 1;
      System.out.println("Il risultato di "+j+" >> 1 e': " + i);
      j=i;
      i=i >> 1;
      System.out.println("Il risultato di "+j+" >> 1 e': " + i);
      //Applicazione dello shift bit a bit verso sinistra
      i=100;
      j=i;
      i=i << 1;
     System.out.println("Il risultato di "+j+" << 1 e': " + i);
      j=i;
      i=i << 1;
      System.out.println("Il risultato di "+j+" << 1 e': " + i);
      j=i;
      i=i << 1;
      System.out.println("Il risultato di "+j+" << 1 e': " + i);
   }
}
```

L'esecuzione della applicazione produrrà quanto segue:

```
Il risultato di 100 >> 1 e': 50
Il risultato di 50 >> 1 e': 25
Il risultato di 25 >> 1 e': 12
```

```
Il risultato di 100 << 1 e': 200
Il risultato di 200 << 1 e': 400
Il risultato di 400 << 1 e': 800
```

Poiché la rappresentazione binaria del numero decimale 100 è 01100100, lo spostamento dei bit verso destra di una posizione, produrrà come risultato il numero binario 00110010 che corrisponde al valore 50 decimale; viceversa, lo spostamento dei bit verso sinistra di una posizione, produrrà come risultato il numero binario 11001000 che corrisponde al valore 200 decimale.
Appare evidente che le operazioni di shift verso destra o verso sinistra di 1 posizione dei bit di un numero intero, corrispondono rispettivamente alla divisione o moltiplicazione di un numero intero per 2.
Ciò che rende particolari questi operatori, è la velocità con cui sono eseguiti rispetto alle normali operazioni di prodotto o divisione: di conseguenza, questa caratteristica li rende particolarmente appetibili per sviluppare applicazioni che necessitano di fare migliaia di queste operazioni in tempo reale.
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

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

Nelle tabelle precedenti sono riportati tutti i possibili risultati prodotti dall'applicazione degli operatori nella tabella precedente. Tutte le combinazioni sono state effettuate considerando un singolo bit degli operandi. Il prossimo è un esempio di applicazione degli operatori logici bit a bit a variabili di tipo **long**:

```
public class OperatoriLogiciBitaBit {
   public static void main(String[] args) {
      long sinistro = 100;
      long destro = 125;
      long risultato = sinistro & destro;
      System.out.println("100 & 125 = "+risultato);
      risultato = sinistro | destro;
      System.out.println("100 | 125 = "+risultato);
      risultato = sinistro ^ destro;
      System.out.println("100 ^ 125 = "+risultato);
      risultato = ~sinistro;
      System.out.println("~ 125 = "+risultato);
   }
}
```

Il risultato della esecuzione della applicazione sarà il seguente:

```
100 & 125 = 100
100 | 125 = 125
100 ^ 125 = 25
~125 = -101
```

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

sx e dx in binario		
variabile	**decimale**	**binario**
sx	100	01100100
dx	125	01111101

il significato dei risultati prodotti dalla applicazione *OperatoriLogiciBitaBit* è schematizzato nella successiva tabella.

OperatoriLogiciBitaBit				
operatore	**sinistro**	**destro**	**risultato**	**decimale**
&	01100100	01111101	01100100	100
|	01100100	01111101	01111101	125
^	01100100	01111101	00011001	25
~	01100100	----------	10011011	155

Nella prossima applicazione, utilizziamo gli operatori logici bit a bit per convertire un carattere minuscolo nel relativo carattere maiuscolo. I caratteri

da convertire vengono trasmessi alla applicazione attraverso la riga di comando della Java Virtual Machine.

```
public class Maiuscolo {
   public static void main(String[] args) {
         int minuscolo = args[0].charAt(0);
         int maiuscolo = minuscolo & 223;
        System.out.println("Prima della conversione il carattere e': '"+(char)minuscolo+"' ed il suo
codice UNICODE e': "+minuscolo);
        System.out.println("Dopo la conversione il il carattere e': '"+(char)maiuscolo+"' ed il suo codice
UNICODE e':"+maiuscolo);
   }
}
```

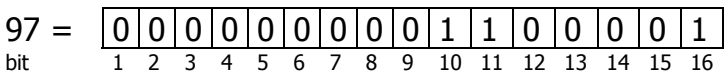
Eseguiamo l'applicazione passando il carattere 'a' come parametro di input. Il risultato che otterremo sarà:

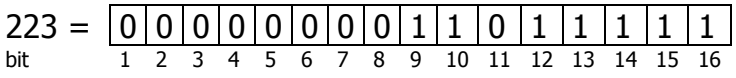
```
java ConvertiInMaiuscolo a
Prima della conversione il carattere e':  a  ed il suo codice UNICODE e': 97
Dopo la conversione il il carattere e':  A  ed il suo codice UNICODE e':  65
```

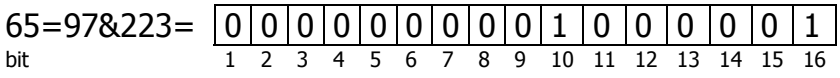
Per effettuare la conversione dei carattere nel relativo carattere minuscolo, abbiamo utilizzato l'operatore "&" per mettere a zero il sesto bit della variabile di tipo **int**, *minuscolo*, che contiene il codice UNICODE del carattere che vogliamo convertire.
Il carattere 'a' è rappresentato dal codice UNICODE 97, 'A' dal codice UNICODE 65 quindi, per convertire il carattere minuscolo nel rispettivo maiuscolo, è necessario sottrarre 32 al codice UNICODE del primo. La stessa regola vale per tutti i caratteri dell'alfabeto inglese:

UNICODE('a') - 32 = UNICODE('A')
UNICODE('b') - 32 = UNICODE('B')
UNICODE('c') - 32 = UNICODE('C')
:
UNICODE('z') - 32 = UNICODE('Z')

Dal momento che la rappresentazione binaria del numero 32 è: 0000000000100000, è ovvio che impostando a 0 il sesto bit sottraiamo 32 al valore della variabile.







4.13 Array

Java, come il C, consente l'aggregazione dei tipi base e degli oggetti mettendo a disposizione del programmatore gli array. In altre parole, mediante gli array è possibile creare collezioni di entità dette "tipi base" dell'array: i tipi base di un array possono essere oggetti o tipi primitivi ed il loro numero è chiamato "length" o lunghezza dell'array.
La sintassi per la dichiarazione di una variabile di tipo array è espressa dalla regola seguente:

tipo[] identificatore;

o, per analogia con il linguaggio C,

tipo identificatore[];

dove "*tipo*" rappresenta un tipo primitivo od un oggetto, ed "*identificatore*" è il nome che utilizzeremo per far riferimento ai dati contenuti all'interno dell'array.
Nel prossimo esempio, viene dichiarato un array il cui tipo base è rappresentato da un intero utilizzando entrambe le forme sintattiche riconosciute dal linguaggio:

```
int[] elencodiNumeriInteri;

int elencodiNumeriInteri[];
```

Dichiarare una variabile di tipo array non basta; non abbiamo ancora definito la lunghezza dell'array e sopratutto, non ne abbiamo creato l'istanza.
La creazione dell'istanza di un array in Java, deve essere realizzata utilizzando l'operatore "**new**", che discuteremo nel capitolo successivo; al momento dello stanziamento è necessario dichiararne la lunghezza. La sintassi completa è la seguente:

*identificatore = **new** tipo[lunghezza];*

dove, "*identificatore*" è il nome associato all'array al momento della dichiarazione della variabile, "*tipo*" è il tipo base dell'array e "*lunghezza*" è il numero massimo di elementi che l'array potrà contenere.
Riconsiderando l'esempio precedente, la sintassi completa per dichiarare ed allocare un array di interi di massimo 20 elementi è la seguente:

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

o, in alternativa, è possibile dichiarare e creare l'array, simultaneamente, come per le variabili di un qualsiasi tipo primitivo:

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

4.14 Lavorare con gli array

Creare un array vuole dire aver creato un contenitore vuoto, in grado di accogliere un numero massimo di elementi, definito dalla lunghezza, tutti della stessa tipologia definita dal tipo base dell'array (Figura 32). Di fatto, creando un array abbiamo chiesto alla JVM di riservare spazio di memoria sufficiente a contenerne tutti gli elementi.

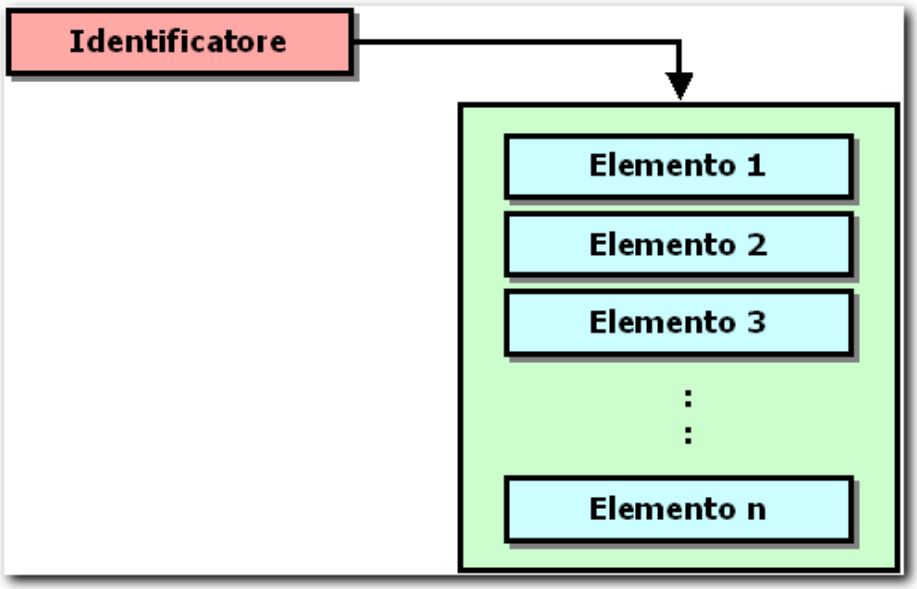


Figura 32: Schema base di un array

Come per il linguaggio C, Java accetta la notazione tra parentesi graffe per poter creare ed inizializzate l'array nello stesso momento. Nel prossimo esempio viene creato un array di interi di nome *listaDiInteri* di lunghezza 5 e contenente i numeri 1,2,3,4,5:

```
int[] listaDiInteri = {1,2,3,4,5};
```

Arrivati a questo punto siamo in grado di dichiarare, creare ed eventualmente inizializzare un array, ma come facciamo ad aggiungere, modificare o estrarre i valori memorizzati all'interno della struttura?
A tale scopo ci viene in aiuto l'operatore indice "***[]***" che, nella forma

identificatore[i]

ci consente di agire direttamente sull'i-esimo elemento contenuto nell'array.

E' importante ricordare sempre che il primo elemento in un array parte dalla posizione 0 (Figura 33) e non 1, di conseguenza un array di n elementi sarà accessibile tramite l'operatore indice *[0],[2],[3],…….[n-1]*.
Infine, è possibile ottenere la lunghezza dell'array tramite la proprietà length nel modo seguente:

lunghezza = identificatore.length

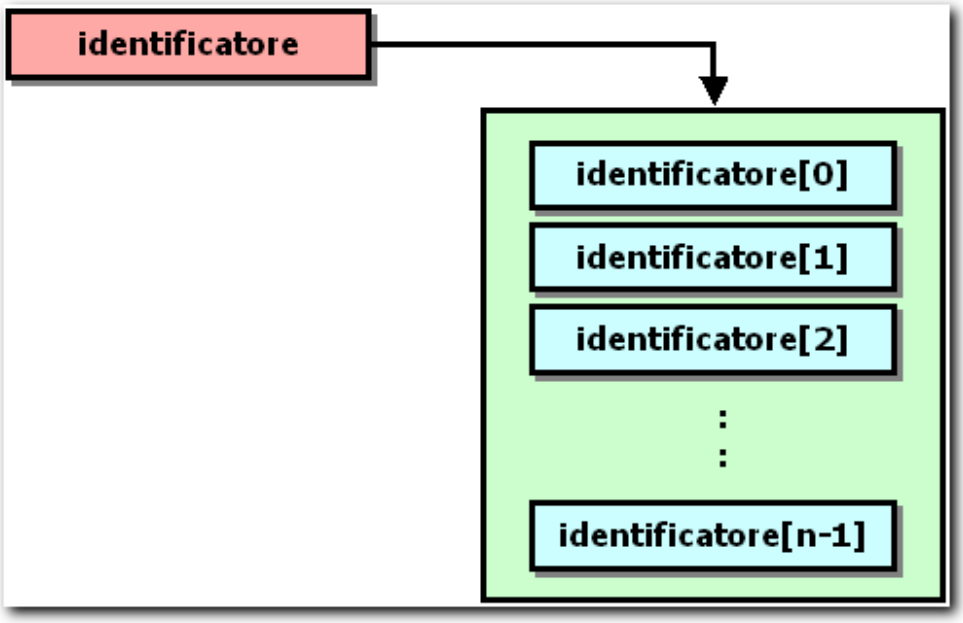


Figura 33: Operatore indice di un array

Nel prossimo esempio, creaiamo un array contenente i primo dieci caratteri dell'alfabeto italiano nel formato minuscolo ed utilizziamo un secondo array per memorizzare gli stessi caratteri in formato maiuscolo.

```
/**
 * @version 0.1
 * @see ConvertiInMaiuscolo
 */

public class MinuscoloMaiuscolo {
   public static void main(String[] args) {
      //Dichiarazione e creazione di un array di caratteri
      //contenente i primi dieci caratteri minuscoli dell'alafabeto italiano
      char[] minuscolo = {'a','b','c','d','e','f','g','h','i','l'};

      //Dichiarazione e creazione dell'array di caratteri
      //di lunghezza 10 che conterrà i caratteri maiscoli dopo
      //la copnversione

      char[] maiuscolo = new char[10];

      //Converto i caratteri in maiuscolo e li inserisco nel nuovo array
      //utilizzando due variabili int che conterranno il codice UNICODE dei
      //due caratteri
```

```
        int carattereMinuscolo;
        int carattereMaiuscolo;
        for (int i = 0; i < 10; i++) {
            carattereMinuscolo = minuscolo[i];

           //Eseguo la conversione in maiuscolo

            carattereMaiuscolo = carattereMinuscolo & 223;

             //Memorizzo il risultato nel nuovo array
           maiuscolo[i]=(char)carattereMaiuscolo;
           System.out.println("minuscolo["+i+"]='" + minuscolo[i] + "', maiuscolo ["+i+"]='" +
maiuscolo[i]+"'");

        }
    }
}
```

L'esecuzione del codice produrrà il seguente output:

```
minuscolo[0]='a', maiuscolo [0]='A'
minuscolo[1]='b', maiuscolo [1]='B'
minuscolo[2]='c', maiuscolo [2]='C'
minuscolo[3]='d', maiuscolo [3]='D'
minuscolo[4]='e', maiuscolo [4]='E'
minuscolo[5]='f', maiuscolo [5]='F'
minuscolo[6]='g', maiuscolo [6]='G'
minuscolo[7]='h', maiuscolo [7]='H'
minuscolo[8]='i', maiuscolo [8]='I'
minuscolo[9]='l', maiuscolo [9]='L'
```

4.15 Array Multidimensionali

Per poter rappresentare strutture dati a due o più dimensioni, Java supporta gli array multidimensionali o array di array. Per dichiarare un array multidimensionale la sintassi è simile a quella per gli array, con la differenza che è necessario specificare ogni dimensione, utilizzando una coppia di parentesi "[]". Un array a due dimensioni può essere dichiarato nel seguente modo:

tipo[][] *identificatore*;

in cui "*tipo*" ed "*identificatore*" rappresentano rispettivamente il tipo base dell'array ed il nome che ci consentirà di accedere ai dati in esso contenuti.
Nella realtà, Java organizza gli array multidimensionali come "array di array"; per questo motivo, non è necessario specificarne la lunghezza per ogni dimensione dichiarata, al momento della creazione: in altre parole, un array multidimensionale non deve essere necessariamente creato utilizzando una singola operazione "***new***".
Nel prossimo esempio, rappresentiamo una tavola per il gioco della dama utilizzando un array a due dimensioni il cui tipo base è l'oggetto *Pedina*.

L'inizializzazione dell'oggetto viene effettuata riga per riga, un valore *null* nella posizione [i][j] identifica una casella vuota (Figura 34).

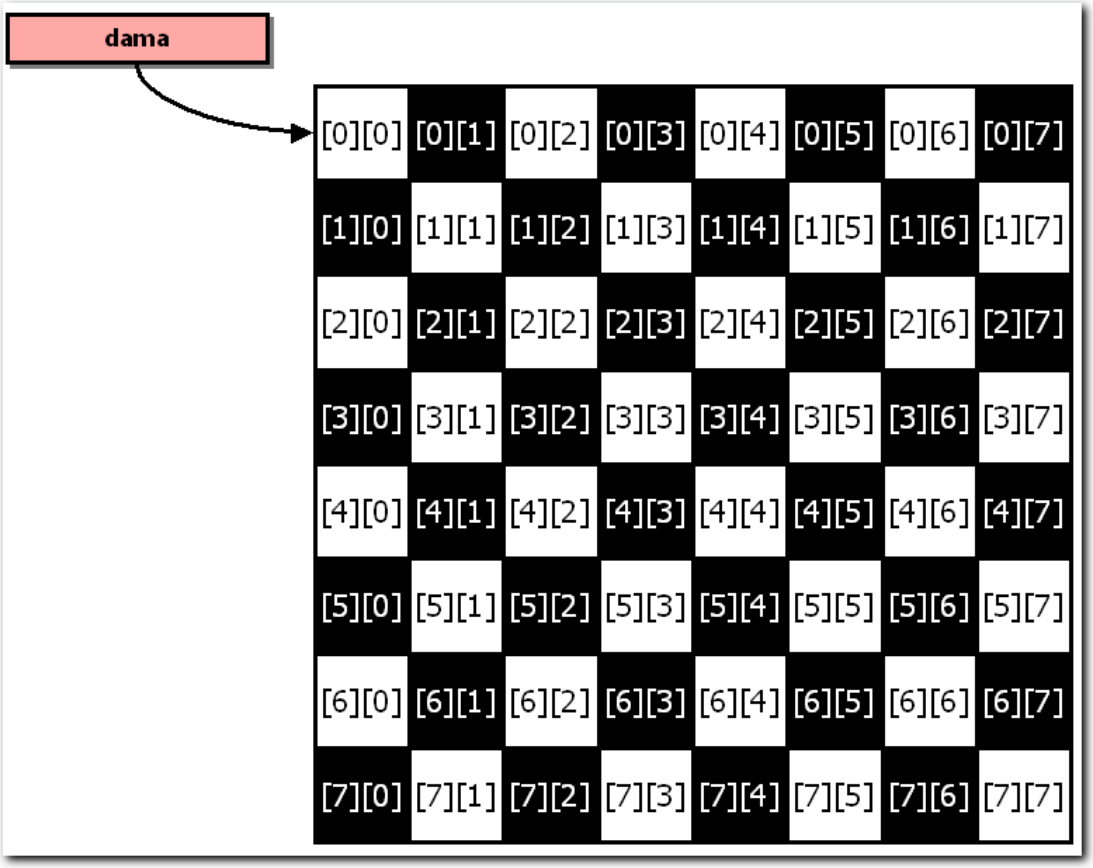


Figura 34: Gioco della dama

```
Pedina[][] dama = new Pedina[8][];
Pedina[] riga0 = {
        new Pedina("bianca"),null, new Pedina("bianca"),null,
        new Pedina("bianca"),null, new Pedina("bianca"),null
};
Pedina[] riga1 = {
        null, new Pedina("bianca"), null, new Pedina("bianca"),
        null, new Pedina("bianca"), null, new Pedina("bianca")
};
Pedina[] riga2 = {
        new Pedina("bianca"),null, new Pedina("bianca"),null,
        new Pedina("bianca"),null, new Pedina("bianca"),null
};
Pedina[] riga3 = { null, null, null, null, null, null, null, null };
Pedina[] riga4 = { null, null, null, null, null, null, null, null };
Pedina[] riga5 = {
        null, new Pedina("nera"), null, new Pedina("nera"),
        null, new Pedina("nera"), null, new Pedina("nera")
};
Pedina[] riga6 = {
        new Pedina("nera"),null, new Pedina("nera"),null,
        new Pedina("nera"),null, new Pedina("nera"),null
};
```

```
Pedina[] riga7 = {
        null, new Pedina("nera"), null, new Pedina("nera"),
        null, new Pedina("nera"), null, new Pedina("nera")
};
```

```
dama[0] = riga0;
dama[1] = riga1;
dama[2] = riga2;
dama[3] = riga3;
dama[4] = riga4;
dama[5] = riga5;
dama[6] = riga6;
dama[7] = riga7;
```

4.16 Espressioni ed istruzioni

Le espressioni rappresentano il meccanismo per effettuare calcoli all'interno della nostra applicazione; combinano variabili e operatori producendo un singolo valore di ritorno.
Le espressioni vengono utilizzate per generare valori da assegnare alle variabili o, come vedremo nel prossimo capitolo, per modificare il corso della esecuzione di una applicazione Java: di conseguenza una espressione non rappresenta una unità di calcolo completa in quanto non produce assegnamenti o modifiche alle variabili della applicazione.
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 prossimo capitolo), combinate tra loro a partire dalle regole sintattiche del linguaggio.

4.17 Regole sintattiche di Java

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

istruzione --> espressione

OPPURE

istruzione --> {
istruzione
[istruzione]
}

OPPURE

istruzione --> controllo_di_flusso

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 --> controllo_di_flusso
istruzione

E sostituiamo il lato destro utilizzando la seconda espansione ottenendo

istruzione --> controllo_di_flusso ***-->*** *controllo_di_flusso*
istruzione ***2*** *{*
istruzione
}

Applicando ora la terza regola di espansione otteniamo :

controllo_di_flusso ***-->*** *controllo_di_flusso*
{ ***3*** *{*
istruzione *controllo_di_flusso*
} *istruzione*
}

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

if(espressione_booleana)

Ecco che la nostra espansione diventerà quindi :

istruzione --> --> --> controllo_di_flusso ***--> if(i>10)***
2 ***3*** *{* ***{***
controllo_di_flusso ***if(I==5)***
istruzione ***print("I vale 5");***
} ***i++;***
}

4.18 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.

5 DEFINIZIONE DI OGGETTI

5.1 Introduzione

In questo capitolo saranno trattati gli aspetti specifici del linguaggio Java relativi alla definizione di oggetti: le regole sintattiche base per la creazione di classi, l'allocazione di oggetti e la determinazione del punto di ingresso (entry point) di un'applicazione.
Per tutta la durata del capitolo sarà importante ricordare i concetti base discussi in precedenza, in particolar modo quelli relativi alla definizione di una classe di oggetti. 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 l'applicazione.
Per completezza, il capitolo tratterà le caratteristiche del linguaggio necessarie a scrivere piccoli programmi, includendo la manipolazione di stringhe e la generazione di messaggi a video.

5.2 Metodi

Un'istruzione rappresenta il mattone per creare le funzionalità di un oggetto. Nasce spontaneo chiedersi: come sono organizzate le istruzioni all'interno degli oggetti?
I metodi rappresentano il cemento che tiene assieme tutti i mattoni e raggruppano blocchi 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:

```
tipo_di_ritorno  nome(tipo identificatore [,tipo identificatore] )
{
        istruzione
        [istruzione]
}
```

"*tipo_di_ritorno*" e "*tipo*" rappresentano ogni tipo di dato (primitivo o classe) e "*nome*" ed "*identificatore*" sono stringhe alfanumeriche, iniziano con una lettera e possono contenere caratteri numerici (discuteremo in seguito come avviene il passaggio di parametri).
La dichiarazione di una classe Java, deve sempre contenere la definizione di tipo di ritorno prodotto da un metodo: "*tipo_di_ritorno*"; se il metodo non ritorna valori, dovrà essere utilizzato il tipo speciale "*void*" .
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 all'esecuzione del metodo. Non manterranno il loro valore tra chiamate differenti e non saranno accessibili da altri metodi.

5.3 Definire una classe

Le istruzioni sono organizzate utilizzando metodi che contengono codice eseguibile che può essere invocato passandogli un numero limitato di valori come argomenti. D'altro canto, Java è un linguaggio orientato ad oggetti e come tale, richiede i metodi siano organizzati internamente alle classi.
Nel primo capitolo abbiamo associato il concetto di classe a quello di categoria; se trasportato nell'ambito del linguaggio di programmazione la definizione non cambia, ma è importante chiarire le implicazioni che ciò comporta.
Una classe Java deve rappresentare un oggetto concettuale e per poterlo fare, deve raggruppare dati e metodi assegnando un nome comune.
La sintassi è la seguente:

```
class Nome
{
        dichirazione_dei_dati
        dichirazione_dei_metodi
}
```

I dati ed i metodi contenuti all'interno della definizione sono chiamati *membri della classe* e devono essere rigorosamente definiti all'interno del blocco di dichiarazione; 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 orientato agli oggetti.
Ricordando la classe "Libro" descritta nel primo capitolo, avevamo stabilito che un libro è tale solo se contiene pagine da sfogliare, strappare ecc..
Utilizzando la sintassi di Java potremmo fornire una prima grossolana definizione della nostra classe nel modo seguente:

```
/**
 * Questa classe rappresenta una possibile definizione della classe libro.
* @version 0.1
 */

class Libro {

        //dichiarazione dei dati della classe

        int numero_di_pagine=100;
        int pagina_corrente=0;
        String lingua = "Italiano";
        String tipologia = "Testo di letteratura Italiana";

        // dichiarazione dei metodi
```

```
	/**
	* Restituisce il numero della pagina che stiamo leggendo
	* @return int : numero della pagina
	*/
	int paginaCorrente()
	{
		return pagina_corrente;
	}
	/**
	* Questo metodo restituisce il numero di pagina
	* dopo aver voltato dalla pagina attuale a quella successiva
	* @return int : numero della pagina
	*/
	int paginaSuccessiva()
	{
		pagina_corrente++;
		return pagina_corrente;
	}

	/**
	* Questo metodo restituisce il numero di pagina
	* dopo aver voltato dalla pagina attuale a quella precedente
	* @return int : numero della pagina
	*/
	int paginaPrecedente()
	{
		pagina_corrente--;
		return pagina_corrente;
	}

	/**
	* Restituisce la lingua con cui è stato scritto il libro
	* @return String : Lingua con cui il libro è stato scritto
	*/
	String liguaggio()
	{
		return lingua;
	}

	/**
	* Restituisce la tipologia del libro
	* @return String : Descrizione della tipologia del libro
	*/
	String tipologia()
	{
		return tipologia;
	}
}
```

5.4 Variabili "reference"

Java fa una netta distinzione tra classi e tipi primitivi. La differenza principale, ma anche la meno evidente, è relativa al fatto che un oggetto non è allocato dal linguaggio al momento della dichiarazione, come avviene per le variabili di tipo primitivo. Per chiarire questo punto, esaminiamo la seguente dichiarazione:

int contatore;

La JVM, quando incontra questa riga di codice, crea il puntatore ad una variabile intera chiamata "*contatore" e* contestualmente, alloca quattro byte in memoria per l'immagazzinamento del dato inizializzandone il valore a 0 (Figura 35).

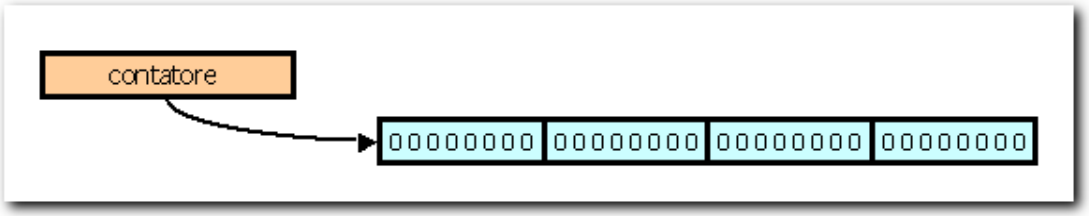


Figura 35: dichirazione di una variabile int

Con le classi lo scenario cambia: in questo caso la JVM crea una variabile che conterrà il puntatore all'oggetto in memoria, ma non alloca risorse per caricare la classe. Di fatto l'oggetto non viene creato (Figura 36).

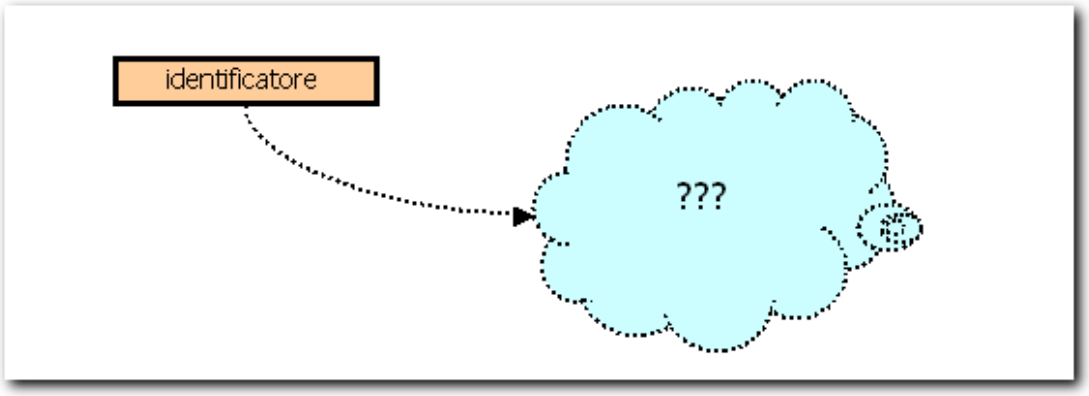


Figura 36: Dichirazione di una variabile reference

Le variabili di questo tipo sono dette "*variabili reference*" ed hanno l'unica capacità di tracciare oggetti del tipo compatibile: ad esempio una variabile "*reference*" di tipo Stack non può contenere puntatori oggetti di tipo Libro.
Oltre che per gli oggetti, Java utilizza lo stesso meccanismo per gli array; questi non sono allocati al momento della dichiarazione, ma la JVM crea una variabile che conterrà il puntatore alla corrispondente struttura dati in memoria.

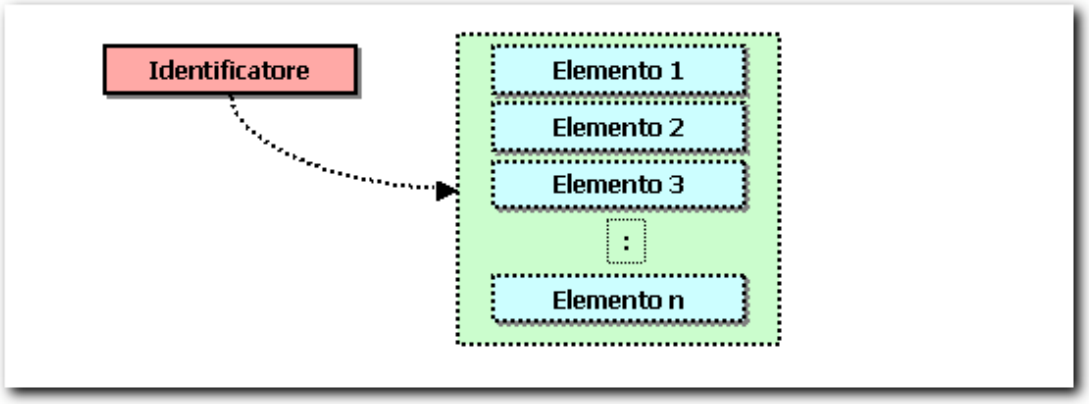


Figura 37: Dichiarazione di un array

Un array può essere dichiarato utilizzando la sintassi:

tipo *identificatore[];*

Una dichiarazione di questo genere, crea una variabile che tiene traccia di un array di dimensione arbitraria (Figura 37).
Le variabili reference sono concettualmente molto simili ai puntatori C e C++, ma non consentono la conversione intero/indirizzo o le operazioni aritmetiche; tuttavia le variabili reference possono essere ugualmente utilizzate per la creazione di strutture dati complesse come liste, alberi binari e array multidimensionali. In questo modo eliminano gli svantaggi derivanti dall'uso di puntatori, mentre ne mantengono tutti i vantaggi.
Un'ulteriore considerazione concernente la gestione delle variabili in Java, riguarda la differenza nell'allocazione della memoria per rappresentare i dati di un qualsiasi tipo primitivo.
La rappresentazione di un dato nel linguaggio C e C++ rispecchia il corrispondente dato macchina e questo comporta:

1. *Il compilatore riserva solo la memoria sufficiente a rappresentare un tipo di dato (per una variabile di tipo byte saranno riservati 8 bit e per un variabile di tipo int 16 ecc.);*
2. *Il compilatore C e C++ non garantisce che la precisione di un dato sia quella stabilita dallo standard ANSI e di conseguenza, un tipo int potrebbe essere rappresentato con 16 o 32 bit secondo l'architettura di riferimento.*

Java rappresenta i dati allocando sempre la stessa quantità di memoria, tipicamente 32 o 64 bit, indipendentemente dal tipo di dato da rappresentare: di fatto, le variabili si comportano come se: 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, garantisce la portabilità del Bytecode su ogni architettura, assicurando che una variabile si comporterà sempre allo stesso modo.

5.5 *Scope* di una variabile Java

Differentemente da linguaggi come il Pascal, in cui le variabili debbono essere dichiarate all'interno di un apposito blocco di codice, Java come il C e C++ lascia al programmatore la libertà di dichiarare le variabili in qualsiasi punto del codice del programma; altra caratteristica del linguaggio è che due o più variabili possono essere identificate dallo stesso nome.
La flessibilità offerta dal linguaggio richiede, però, che siano definite alcune regole per stabilire i limiti di una variabile evitando sovrapposizioni pericolose.
Lo "*scope*" di una variabile Java è la regione di codice all'interno della quale essa può essere referenziata utilizzando il suo identificatore e ne determina il ciclo di vita individuando quando la variabile debba essere allocata o rimossa dalla memoria.
I blocchi di istruzioni ci forniscono il meccanismo necessario a determinare i confini dello scope di una variabile: di fatto, una variabile è referenziabile solo all'interno del blocco di istruzioni che contiene la sua dichiarazione ed ai sottoblocchi contenuti.
In dettaglio, le regole di definizione dello scopo di una variabile possono essere schematizzate nel modo seguente.

Le variabili dichiarate all'interno del blocco di dichiarazione di una classe sono dette variabili membro. Lo scope di una variabile membro è determinato dall'intero blocco di dichiarazione della classe (Figura 38).

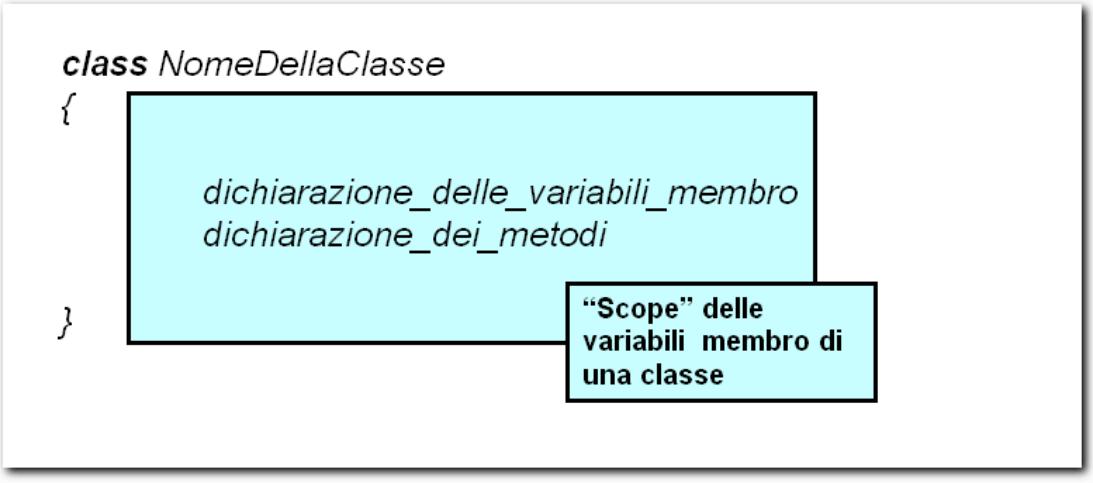


Figura 38: Scope delle variabili membro

Le variabili definite all'interno del blocco di dichiarazione di un metodo sono dette variabili locali. Lo scope di una variabile locale è determinato dal blocco

di codice all'interno della quale è dichiarata ed ai suoi sottoblocchi (Figura 39).

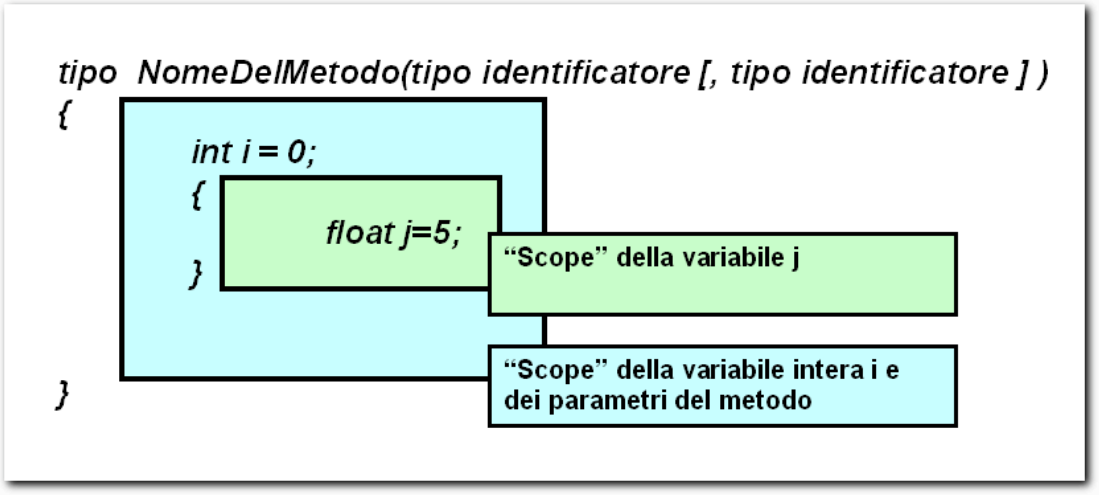


Figura 39: Scope di una variabile locale

Per esempio, il codice seguente produce un errore di compilazione poichè si tenta di utilizzare la variabile "somma" di tipo intero all'esterno del suo blocco di dichiarazione.

```
class scope1
{
        int scopeErrato()
        {
                int i;
                for (i=0 ; i<10; i++)
                {
                        int somma = somma+i;
                }
                //Questa riga di codice contiene un errore
                System.out.println("La somma vale: "+somma);
        }

        int scopeCorretto
        {
                int i;
                for (i=0 ; i<10; i++)
                {
                        int somma = somma+i;
                        //Questa riga di codice è corretta
                        System.out.println("La somma vale: "+somma);
                }

        }
}
```

Lo scope di una variabile Java ne definisce il ciclo di vita. Una variabile non esiste fino a che si entra nel blocco che ne delimita lo scope e è distrutta quando la JVM esce dal blocco. Di conseguenza, le variabili locali non possono

mantenere il loro valore tra due chiamate differenti.

```
class scope2
{
	public static void main (String[] args)
	{
		for (int i=0;i<2;i++)
		{
			int j = 5;
			System.out.println("Entro nel blocco che definisce lo scope della varabile j");
			System.out.println("La variabile J vale: "+j);
			System.out.println("Sommo 5 alla variabile j.");
			j = j+5;
			System.out.println("Ora la variabile J vale: "+j);
			System.out.println("esco dal blocco che definisce lo scope della varabile j");
			System.out.println("------------------------------------------------------------------");
		}
	}
}
```

L'esecuzione dell'esempio dimostra come la variabile *j* definita all'interno del blocco di codice dell'istruzione **for**, sia creata, modificata e distrutta perdendo il valore tra due chiamate differenti.

```
Entro nel blocco che definisce lo scope della varabile j
La variabile J vale: 5
Sommo 5 alla variabile j.
Ora la variabile J vale: 10
esco dal blocco che definisce lo scope della varabile j
------------------------------------------------------------------
Entro nel blocco che definisce lo scope della varabile j
La variabile J vale: 5
Sommo 5 alla variabile j.
Ora la variabile J vale: 10
esco dal blocco che definisce lo scope della varabile j
------------------------------------------------------------------
```

Metodi differenti possono contenere dichiarazioni di variabili con identificatore uguale. Le variabili locali possono avere lo stesso identificatore delle variabili membro. In questo caso, sarà necessario specificare esplicitamente quale variabile si voglia referenziare utilizzandone l'identificatore.

Nel prossimo esempio viene mostrato come referenziare una variabile membro o una variabile locale aventi lo stesso identificatore.

```
public class Scope3 {
	//dichiarazione della variabile membro appoggio
	int appoggio = 5;
```

```
        void somma(int valore)
        {
                //dichiarazione della variabile locale appoggio
                int appoggio = 3;

                //Sommo valore alla variabile locale
                appoggio = appoggio + valore;
                System.out.println("La variabile locale dopo la somma vale: "+appoggio);

                //sommo valore alla variabile membro
                this.appoggio = this.appoggio + valore;
                System.out.println("La variabile membro dopo la somma vale: "+this.appoggio);

                //sommo la variabile locale alla variabile membro
                this.appoggio = this.appoggio + appoggio;
                System.out.println("Dopo la somma delle due variabili, la variabile membro vale:
"+this.appoggio);
        }

        public static void main(String[] args)
        {
                Scope3 Scope = new Scope3();
                Scope.somma(5);
        }
}
```

```
La variabile locale dopo la somma vale: 8
La variabile membro dopo la somma vale: 10
Dopo la somma delle due variabili, la variabile membro vale: 18
```

Concludendo, nonostante la flessibilità messa a disposizione dal linguaggio Java, è buona regola definire tutte le variabili membro all'inizio del blocco di dichiarazione della classe e tutte le variabili locali al principio del blocco che ne delimiterà lo scope. Questi accorgimenti non producono effetti durante l'esecuzione dell'applicazione, ma migliorano la leggibilità del codice sorgente consentendo di identificare facilmente quali saranno le variabili utilizzate per realizzare una determinata funzionalità.

5.6 L'oggetto *null*

Come per i tipi primitivi, ogni variabile reference richiede che, al momento della dichiarazione, le sia assegnato un valore iniziale. Per questo tipo di variabili, il linguaggio Java prevede il valore speciale ***null*** che, rappresenta un oggetto inesistente ed è utilizzato dalla JVM come valore prestabilito: quando un'applicazione tenta di accedere ad una variabile reference contenente un riferimento a *null*, il compilatore Java produrrà un messaggio di errore in forma di oggetto di tipo NullPointerException[6].
Oltre ad essere utilizzato come valore prestabilito per le variabili reference, l'oggetto *null* gioca un ruolo importante nell'ambito della programmazione per la gestione delle risorse: quando ad una variabile reference è assegnato il

[6] Le eccezioni verranno discusse in un capitolo successivo

valore *null*, l'oggetto referenziato è rilasciato e se non utilizzato sarà inviato al garbage collector che si occuperà di rilasciare la memoria allocata rendendola nuovamente disponibile.
Altro uso che può essere fatto dell'oggetto *null* riguarda le operazioni di comparazione come mostrato nel prossimo esempio in cui creeremo la definizione di classe per una struttura dati molto comune: la "Pila" o "Stack".

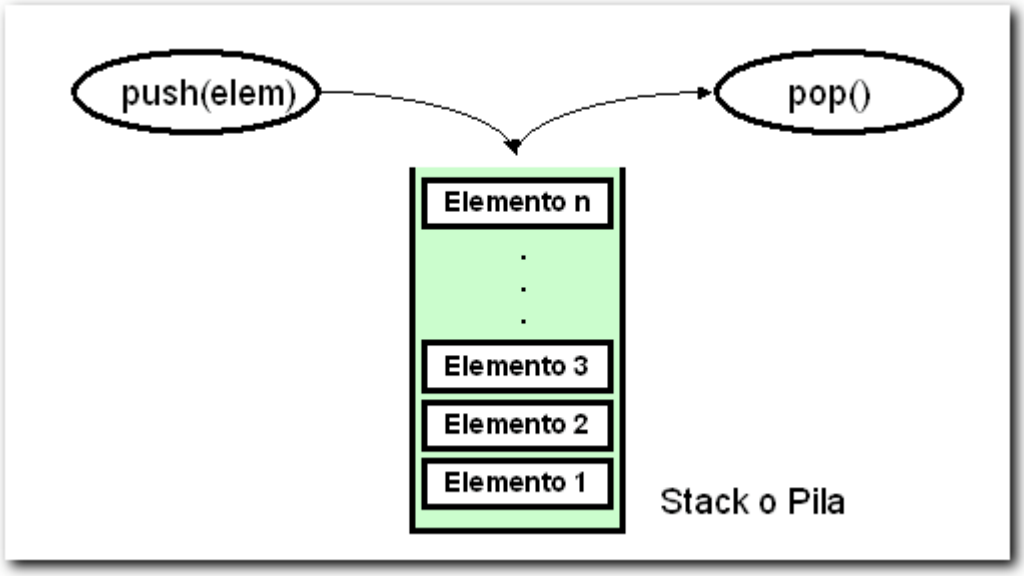


Figura 40: Pila

Una Pila è una struttura dati gestita con la metodologia LIFO (Last In First Out), in altre parole l'ultimo elemento ad essere inserito è il primo ad essere estratto (Figura 40).
L'oggetto che andremo a definire, conterrà al massimo 20 numeri interi e avrà i due metodi:

void push(int)
int pop()

Il metodo *push* ritorna un tipo **void** e prende come parametro un numero intero da inserire sulla cima della pila, il metodo *pop* non accetta parametri, ma restituisce l'elemento sulla cima della Pila.
I dati iseriti nella Pila sono memorizzati all'interno di un array che deve essere inizializzato all'interno del metodo push(**int**). Si può controllare lo stato dello stack ricordando che una variabile reference appena dichiarata contiene un riferimento all'oggetto ***null***.

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */
```

```
class Pila {
	int[] dati;
	int cima;

	/**
	* Il metodo push ritorna un tipo void e prende come
	* parametro un numero intero da inserire sulla cima della pila
	* @return void
	* @param int dato : elemento da inserire sulla cima della pila
	*/
	void push(int dato)
	{
		if(dati == null)
		{
			cima = 0;
			dati = new int[20];
		}
		if(cima < 20)
		{
			dati[cima] = dato;
			cima ++;
		}
	}

	/**
	* il metodo pop non accetta parametri e restituisce
	* l'elemento sulla cima della Pila
	* @return int : dato sulla cima della pila
	*/
	int pop()
	{
		if(cima > 0)
		{
			cima--;
			return dati[cima];
		}
		return 0; // Bisogna tornare qualcosa
	}
}
```

All'interno della definizione del metodo ***void*** *push(****int*** *i)*, per prima cosa viene controllato se l'array è stato inizializzato utilizzando l'operatore di uguaglianza con il valore ***null***, ed eventualmente è allocato come array di venti numeri interi.

```
		if(dati == null)
		{
			cima = 0;
			dati = new int[20];
		}
```

A seguire, l'algoritmo esegue un controllo per verificare se è possibile inserire elementi all'interno dell'array. In particolare, essendo venti il numero massimo di interi contenuti, mediante l'istruzione **if** il metodo accerta che la posizione puntata dalla variabile *cima* sia minore della lunghezza massima dell'array (ricordiamo che in un array le posizioni sono identificate a partire da 0): se

l'espressione "*cima < 20*" restituisce il valore **true**, il numero intero passato come parametro di input viene inserito nell'array nella posizione *cima*. La variabile *cima* viene quindi aggiornata in modo che punti alla prima posizione libera nell'array.

```
if(cima < 20)
{
        dati[cima] = dato;
        cima ++;
}
```

Il metodo ***int*** *pop()* estrae il primo elemento della pila e lo restituisce all'utente. Per far questo, il metodo controlla se il valore di *cima* sia maggiore di zero e di conseguenza, che esista almeno un elemento all'interno dell'array: se la condizione restituisce un valore di verità, *cima* è modificata in modo da puntare all'ultimo elemento inserito, il cui valore è restituito mediante il comando **return**. In caso contrario il metodo ritorna il valore zero.

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

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

5.7 Creare istanze

Definire una variabile reference non basta a creare un oggetto ma, deve essere necessariamente caricato in memoria dinamicamente. L'operatore ***new*** fa questo per noi, riservando la memoria necessaria al nuovo oggetto e restituendone il riferimento, che può quindi essere memorizzato in una variabile reference del tipo appropriato.
La sintassi dell'operatore ***new*** per creare il nuovo oggetto dalla sua definizione di classe è la seguente:

new *Nome() ;*

Le parentesi sono necessarie ed hanno un significato particolare che sveleremo presto, "*Nome*" è il nome di una definizione di classe appartenente alle API di Java oppure definita dal programmatore.
Per creare un oggetto di tipo Pila faremo uso dell'istruzione:

```
Pila Istanza_Di_Pila = new  Pila();
```

La riga di codice utilizzata, dichiara una variabile "*Istanza_Di_Pila*" di tipo Pila, crea l'oggetto utilizzando la definizione di classe e memorizza il puntatore alla memoria riservata nella variabile reference (Figura 41).
Un risultato analogo può essere ottenuto anche nel modo seguente:

```
Pila Istanza_Di_Pila = null;
Istanza_Di_Pila  = new  Pila();
```

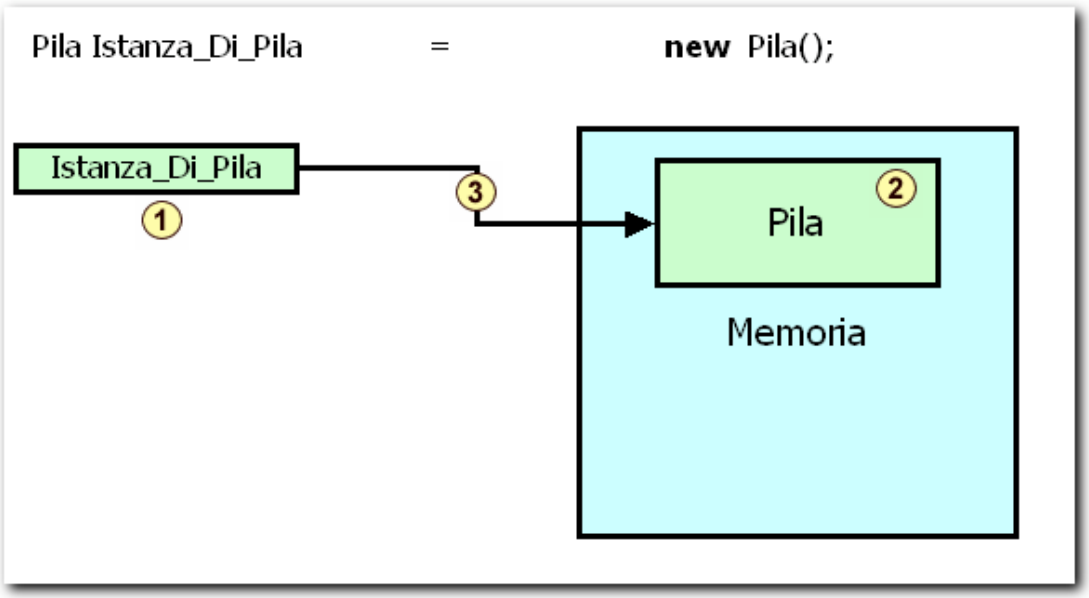


Figura 41: Ordine di esecuzione per l'operatore new

Ricordando quanto detto nel capitolo precedente, notiamo come gli array vengano creati in maniera simile alle classi Java:

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

o, analogamente al caso precedente

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

In questo caso, la JVM crea una variabile reference di tipo intero, riserva memoria per venti interi e ne memorizza il puntatore in *Array_Di_Elementi*. L'analogia con le classi è evidente e dipende dal fatto che un array Java è un oggetto particolare, costruito da una classe base che non contiene definizioni di metodi e consente l'utilizzo dell'operatore indice ***[]***. Da qui la necessità di creare un array utilizzando l'operatore **new**.

5.8 Oggetti ed Array Anonimi

Capita spesso di dover creare oggetti referenziati solo all'interno di singole istruzioni od espressioni. Nel prossimo esempio utilizziamo un oggetto di tipo Integer per convertire il valore una variabile di primitiva ***int*** in una stringa.

```
public class OggettiAnonimi
{
        public static void main(String[] args)
        {
                int i = 100;
                Integer appoggio =  new Integer(i);
                String intero = appoggio.toString();
                System.out.println(intero);
        }
}
```

L'oggetto di tipo Integer, è utilizzato soltanto per la conversione del dato primitivo e mai più referenziato. Lo stesso risultato può essere ottenuto nel modo seguente:

```
public class OggettiAnonimiNew
{
        public static void main(String[] args)
        {
                int i = 100;
                String intero = new Integer(i).toString();
                System.out.println(intero);
        }
}
```

Nel secondo caso, l'oggetto di tipo Integer è utilizzato senza essere associato a nessun identificatore e di conseguenza, è detto "oggetto anonimo". Più in generale, un oggetto è anonimo quando è creato utilizzando l'operatore ***new*** omettendo la specifica del tipo dell'oggetto ed il nome dell'identificatore.
Come gli oggetti, anche gli array possono essere utilizzati nella loro forma anonima. Anche in questo caso un array viene detto anonimo quando viene creato omettendo il nome dell'identificatore.

5.9 L'operatore punto "."

L'operatore punto, fornisce l'accesso alle variabili membro di una classe, tramite il suo identificatore. Per comprendere meglio quanto detto, esaminiamo attentamente il prossimo esempio:

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO (Last In First Out)
 * @version 0.2
 */

class Pila {
        int[] dati;
```

```
        int cima;

        /**
        * Il metodo push ritorna un tipo void e prende come
        * parametro un oggetto di tipo Elemento da  cui estrarre
        * il valore da inserire sulla cima della pila
        * @return void
        * @param Elemento dato : elemento da inserire sulla cima della pila
        */
        void push(Elemento elem)
        {
                if(dati == null)
                {
                        cima = 0;
                        dati = new int[20];
                }
                if(cima < 20)
                {
                        // Inserisco sulla cima della pila
                        // il valore intero rappresentato dal dato
                        // membro dell'oggetto Elemento.
                        dati[cima] = elem.valore;
                        cima ++;
                }
        }

        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return Elemento : dato sulla cima della pila
        */
        Elemento pop()
        {
                if(cima > 0)
                {
                        cima--;
                        //Creo una nuova istanza dell'oggetto Elemento
                        //da utilizzare per trasportare il valore sulla cima dello stack
                        Elemento elem = new Elemento();
                        //imposto il valore del dato da trasportare
                        elem.valore = dati[cima];
                        return elem;
                }
                return null;
        }
}

class Elemento
{
        int valore;
}
```

In questa nuova versione, la classe Pila è stata modificata affinché i metodi *push* e *pop* utilizzino oggetti di tipo *Elemento* invece di numeri interi: tali oggetti, la cui definizione segue quella della classe Pila, contengono un solo dato membro di tipo ***int***.

Utilizzando l'operatore punto, il metodo push estrae il valore del dato membro e lo inserisce all'interno dell'array, viceversa, il metodo pop ne imposta il valore e torna un oggetto di tipo Elemento.
Operazioni analoghe possono essere utilizzate per inserire nuovi elementi nella pila e riaverne il valore come mostrato nell'esempio seguente:

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

Elemento elem = new Elemento();
elem.valore = 10;

//Creiamo un oggetto Pila
Pila pila = new Pila();

//inseriamo il valore in cima alla Pila
pila.push(elem);

//Eseguiamo una operazione di
//pop sulla Pila
elem = s.pop();
```

Oltre ai dati membro, anche i metodi di una classe Java sono accessibili mediante l'operatore punto, come mostrato nel prossimo esempio in cui introduciamo una nuova modifica alle classi *Pila* ed *Elemento*.

```
class Elemento
{
	int valore;

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

	int restituisciIlValore()
	{
		return valore;
	}

}
```

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.3
 */

class Pila {
	int[] dati;
	int cima;

	/**
	* Il metodo push ritorna un tipo void e prende come
```

```
        * parametro un oggetto di tipo Elemento da  cui estrarre
        * il valore da inserire sulla cima della pila
        * @return void
        * @param Elemento dato : elemento da inserire sulla cima della pila
        */
        void push(Elemento elem)
        {
                if(dati == null)
                {
                        cima = 0;
                        dati = new int[20];
                }
                if(cima < 20)
                {
                        // Inserisco sulla cima della pila
                        // il valore intero rappresentato dal dato
                        // membro dell'oggetto Elemento.
                        dati[cima] = elem.restituisciIlValore();
                        cima ++;
                }
        }

        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return Elemento : dato sulla cima della pila
        */
        Elemento pop()
        {
                if(cima > 0)
                {
                        cima--;
                        //Creo una nuova istanza dell'oggetto Elemento
                        //da utilizzare per trasportare il valore sulla cima dello stack
                        Elemento elem = new Elemento();
                        //imposto il valore del dato da trasportare
                        elem. impostaIlValore(dati[cima]);
                        return elem;
                }
                return null;
        }
}
```

5.10 Auto referenza esplicita

L'operatore punto, oltre a fornire la via di accesso ai dati membro od ai metodi di un oggetto attraverso una variabile reference, consente ai metodi di accedere ai dati membro della classe di definizione.
Una nuova versione della classe Elemento, definita nel paragrafo precedente, ci può aiutare a comprendere meglio quanto affermato:

```
class Elemento
{
        int valore;

        // questo metodo inizializza il dato membro della classe
```

```
        void impostaIlValore(int valore)
        {
                ???. valore = valore;
        }

        int restituisciIlValore()
        {
                return valore;
        }

}
```

Il metodo *impostaIlValore* prende come parametro un intero, tramite il parametro identificato da *valore* ed utilizza l'operatore punto per memorizzare il dato all'interno della variabile membro *valore*. Dal momento che il riferimento alla variabile *valore* è ambiguo, è necessario chiarire il modo con cui un oggetto possa referenziare se stesso (evidenziato nell'esempio dai punti interrogativi **???**).
Java prevede un modo di auto referenza particolare identificabile con la variabile reference *this*. Di fatto, il valore di *this* è modificato automaticamente dalla Java Virtual Machine affinché, ad ogni istante, sia sempre riferito all'oggetto attivo, intendendo per "oggetto attivo" l'istanza della classe in esecuzione durante la chiamata al metodo corrente. La nostra classe diventa quindi:

```
class Elemento
{
        int valore;

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

        int restituisciIlValore()
        {
                return valore;
        }

}
```

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

5.11 Auto referenza implicita

Poiché, come abbiamo detto, ogni metodo deve essere definito all'interno di una definizione di classe, il meccanismo di auto referenza esplicita è molto comune in applicazioni Java; se però un riferimento non è ambiguo, Java

consente di utilizzare un ulteriore meccanismo detto di **auto referenza implicita**, per mezzo del quale è possibile accede a dati membro o metodi di una classe senza necessariamente utilizzare esplicitamente la variabile reference *this*.

```
class Elemento
{
        int val;

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

        int restituisciIlValore()
        {
                return val;
        }

}
```

Il meccanismo su cui si basa l'auto referenza implicita è legato alla visibilità di una variabile. Ricordando che la visibilità di una variabile in Java è limitata al blocco ed ai sottoblocchi di codice in cui è stata effettuata la sua dichiarazione, Java ricerca una variabile non qualificata risalendo a ritroso tra i diversi livelli dei blocchi di codice.
Inizialmente, Java ricerca la dichiarazione della variabile all'interno del blocco di istruzioni corrente:

1. *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.*
2. *Se neanche la lista dei parametri del metodo soddisfa la ricerca, Java legge il blocco di dichiarazione dell'oggetto corrente utilizzando implicitamente la variabile reference this.*
3. *Nel caso in cui la variabile non è neanche un dato membro dell'oggetto, un codice di errore è generato al momento della produzione del Bytecode dal compilatore.*

Anche se l'uso implicito di variabili facilita la scrittura di codice, riducendo la quantità di caratteri da digitare, abusare di questa tecnica può essere causa di ambiguità all'interno della definizione della classe.
Tipicamente, la situazione a cui si va incontro è la seguente:

```
class Elemento {
        int val;
```

```
        void impostaIlValore(int valore)
        {
                int val ;
                val = valore;
        }

        int restituisciIlValore()
        {
                return val;
        }

}
```

L'assenza dell'uso esplicito della variabile reference *this* è la causa della perdita del dato passato come parametro al metodo *impostaIlValore(**int**)*: di fatto, il valore è 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 all'esecuzione del metodo.
Meno ambiguo è invece l'uso dell'auto referenza implicita se impiegata nella chiamata ai metodi della classe. In questo caso Java applicherà soltanto il terzo punto dell'algoritmo descritto per la determinazione dei riferimenti alle variabili. Di fatto, un metodo non può essere definito all'interno di un altro, ne può essere utilizzato come argomento per il passaggio di parametri.

5.12 Stringhe

Abbiamo già anticipato che Java mette a disposizione del programmatore molti tipi predefiniti: *String* è uno di questi, definisce il concetto di stringa e come vedremo tra breve, è dotato di molte caratteristiche particolari.
A differenza del linguaggio C, in cui una stringa è gestita mediante array di caratteri terminati dal valore ***null***, Java rappresenta le stringhe come oggetti e come tali, dotati di tutte le caratteristiche previste.
Ciò che rende le stringhe oggetti particolari sono:

- *la possibilità di creare l'istanza di un oggetto String usando semplicemente una notazione con doppi apici omettendo l'operatore new;*
- *l'uso dell'operatore speciale "+" per la concatenazione come mostrato nel prossimo esempio.*

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

Il codice descritto equivale al seguente:

```
String prima = new String("Hello");
String seconda = new String("world");
```

```
String terza = prima.concat(seconda);
```

I metodi messi a disposizione dall'oggetto *String* consentono al programmatore di eseguire funzioni base quali l'individuazione di caratteri o sequenze di caratteri all'interno della stringa, la sostituzione di caratteri minuscoli con i relativi maiuscoli e viceversa, ma non consentono di modificare la stringa rappresentata.
Questa caratteristica rappresenta un'altra particolarità degli oggetti di tipo *String*: una stringa è immutabile. Una volta creata, una stringa non può essere modificata in nessun modo, a meno di utilizzare altri tipi di oggetti (*StringBuffer*) preposti alla loro modificazione.

5.13 Stato di un oggetto Java

Gli oggetti Java rappresentano spesso tipi di dati molto complessi ed 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 Java" il valore in un certo istante dei dati membro rilevanti della classe. Di fatto, non tutti i dati membro di una classe concorrono alla definizione dello stato di un oggetto bensì solo quelli sufficienti a fornire, ad un determinato istante, informazioni sufficienti a fotografarne la condizione esatta.
Ad esempio, lo stato del tipo *Elemento* è rappresentato dal valore del dato membro *valore* di tipo ***int***, mentre quello del tipo Pila dai valori dell'array che contiene gli elementi della struttura dati e dal dato membro cima di tipo ***int*** che contiene il puntatore all'elemento sulla cima della pila.

5.14 Comparazione di oggetti

La comparazione di oggetti Java è leggermente differente rispetto ad altri linguaggi di programmazione e dipende dal modo in cui Java manipola gli oggetti. Di fatto, un'applicazione Java non usa oggetti, ma variabili reference "come oggetti".
Una normale comparazione effettuata utilizzando l'operatore "==" metterebbe a confronto il riferimento agli oggetti in memoria e non il loro stato, producendo un 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.

```
String a = "Java Mattone dopo Mattone";
String b = "Java Mattone dopo Mattone";
(a == b) -> false
```

Molte volte però, ad un'applicazione Java, potrebbe tornare utile sapere se due istanze separate di una stessa classe sono uguali tra loro ovvero,

ricordando la definizione data nel paragrafo precedente, se due oggetti java dello stesso tipo si trovano nello stesso stato al momento del confronto.
Java prevede un metodo speciale chiamato *equals()* che confronta lo stato di due oggetti: tutti gli oggetti in Java, anche quelli definiti dal programmatore, possiedono questo metodo poiché ereditato da una classe base particolare che analizzeremo in seguito parlando di ereditarietà. Il confronto delle due stringhe descritte nell'esempio precedente assume quindi la forma seguente:

```
String a = "Java Mattone dopo Mattone";
String b = "Java Mattone dopo Mattone";
a.equals(b) -> true
```

5.15 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 sigillarla è necessario 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 faccia riferimento ad un oggetto in memoria.
Questo meccanismo rende possibile l'auto referenza poiché, se un metodo è invocato in assenza di un oggetto attivo, la variabile reference ***this*** non sarebbe inizializzata. Il problema è quindi 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 caricato in memoria.
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 il qualificatore ***static*** a sinistra della dichiarazione del metodo come mostrato nella dichiarazione di static_method() nell'esempio che segue:

```
class Euro {
        static double valoreInLire(){
                return 1936.27;
        }
}
```

Un metodo statico esiste sempre a prescindere dallo stato dell'oggetto; tuttavia, la locazione o classe che incapsula il 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:

```
euro.valoreInLire();
```

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

```
Euro eur = new Euro();
eur. valoreInLire();
```

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 la variabile reference ***this***; di conseguenza un oggetto statico non può utilizzare membri non statici della classe di appartenenza come mostrato nel prossimo esempio.

```
class Euro {
   double lire = 1936.27;
   static double valoreInLire()
        {
              //questa operazione non è consentita
              return lire;
        }
}
```

5.16 Il metodo main

Affinché la Java Virtual Machine possa eseguire un'applicazione, è necessario che abbia ben chiaro quale debba essere il primo metodo da eseguire. Questo metodo è detto "**entry point**" o punto di ingresso dell'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 quello della classe specificata alla Java Virtual Machine sarà eseguito all'avvio del processo. Questo significa che ogni classe di un'applicazione può rappresentare un potenziale punto di ingresso, che può quindi essere scelto all'avvio del processo scegliendo semplicemente la classe desiderata.

```
class Benvenuto
{
        public static void main(String args[])
        {
              System.out.println("Benvenuto");
        }
}
```

Vedremo in seguito come una classe possa contenere più metodi membro aventi lo stesso nome, purché abbiano differenti parametri in input. Affinché il metodo *main()* possa essere trovato dalla Java Virtual Machine, è necessario che abbia una lista di argomenti formata da un solo array di stringhe: è proprio grazie a quest'array che il programmatore può inviare ad una applicazione informazioni aggiuntive, in forma di argomenti da inserire sulla riga di comando: il numero di elementi all'interno dell'array sarà uguale al

numero di argomenti inviati. Se non vengono inseriti argomenti sulla riga di comando, la lunghezza dell'array è zero.
Infine, per il metodo *main()* è necessario utilizzare il modificatore ***public***[7] che accorda alla virtual machine il permesso per eseguire il metodo.
Tutto questo ci porta ad un'importante considerazione finale: **tutto è un oggetto**, anche un'applicazione.

5.17 Classi interne

Le classi Java possiedono di un'importante caratteristica, ereditata dai linguaggi strutturati come il Pascal: una classe può essere dichiarata a qualsiasi livello del codice e la sua visibilità è limitata al blocco di codice che ne contiene la definizione. Tali classi sono dette "classi interne" o "inner classes". Ad esempio, possiamo dichiarare la classe *Contenitore* all'interno della classe *Pila*:

```
class Pila {
        class Contenitore
        {
        ……..
        }
…….
}
```

La classe *Pila* è detta "incapsulante" per la classe *Contenitore*. Per comprendere meglio il funzionamento di una classe interna, completiamo l'esempio proposto analizzando di volta in volta gli aspetti principali di questo tipo di oggetti:

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 */
class Pila {
   int[] dati;
   int cima;

   /**
    * Dichiarazione della classe contenitore. Questa classe
    * definisce il contenitore utilizzato dalla pila per memorizzarne i dati.
    * @version 0.1
    */
   class Contenitore {
      void inserisci(int dato) {
         if (dati == null) {
            cima = 0;
            dati = new int[20];
         }
         dati[cima] = dato;
```

[7] Capiremo meglio il suo significato successivamente

```
            cima++;
        }

        int ElementoSullaCima() {
            if (cima > 0) {
                cima--;
                return dati[cima];
            }
            return 0; // Bisogna tornare qualcosa
        }

        int Cima() {
            return cima;
        }
    }

    Contenitore stack = new Contenitore();

    /**
     * Il metodo push ritorna un tipo void e prende come parametro un numero intero da inserire sulla
     * cima della pila
     * @return void
     * @param int dato : elemento da inserire sulla cima della pila
     */
    void push(int dato) {
        if (stack.Cima() < 20) {
            stack.inserisci(dato);
        }
    }

    /**
     * il metodo pop non accetta parametri e restituisce l'elemento sulla cima della Pila
     * @return int : dato sulla cima della pila
     */
    int pop() {
        return stack.ElementoSullaCima();
    }
}
```

La classe Contenitore, definita all'interno del blocco principale di definizione della classe Pila, è soggetta alle stesse regole che ne definiscono lo scope di metodi e variabili: dai metodi della classe *Pila* possiamo accedere ai dati membro ed ai metodi della classe *Contenitore*, utilizzando l'operatore ***punto*** attraverso il suo nome e viceversa, dai metodi della classe *Contenitore* possiamo utilizzare i metodi ed i dati membro della classe incapsulante.
Una caratteristica delle classi interne su cui porre l'accento è quella riguardante il loro ciclo di vita: una classe interna esiste solo se associata in maniera univoca ad un'istanza della classe incapsulante. Questa caratteristica non solo consente di poter accedere ai metodi ed ai dati membro della classe incapsulante, ma garantisce che ogni riferimento alla classe incapsulante sia relativo solo ad una determinata istanza.

Le classi interne possono essere definite a qualsiasi livello del codice, di conseguenza è possibile creare concatenazioni di classi come mostrato nel prossimo esempio:

```
class PagineGialle
{
        class ElencoAbbonati
        {
                class NumeriTelefonici
                {
                }
        }
}
```

5.18 Classi locali

Le classi interne, possono essere definite anche nei blocchi di definizione dei metodi di una classe incapsulante e sono dette classi locali.

```
class PagineGialle {
   String trovaNumeroTelefonico(final String nome, final String cognome) {
      class Abbonato {

         String numero() {
            return find(nome,cognome);
         }

         String find(String nome, String cognome)
         {
            String numero=null;
             //Trova il numero di telefono
             return numero;
         }
      }

      Abbonato abbonato = new Abbonato();
      return abbonato.numero();
   }
}
```

In accordo con le regole di visibilità, la classe ***ElencoAbbonati*** può accedere a tutte le variabili dichiarate all'interno del blocco di definizione del metodo, compresi i suoi argomenti; l'accorgimento è che tutte le variabili utilizzate dalla classe interna e condivise con un metodo della classe incapsulante debbono essere dichiarate ***final*** a causa dei possibili problemi di sincronizzazione che potrebbero verificarsi. Di fatto, poiché Java non consente l'utilizzo di vaiabili globali, l'utilizzo del modificatore ***final*** rappresenta l'unico mezzo per prevenire eventuali errori nell'uso condiviso di una variabile locale da parte di due o più classi.

5.19 Classi Interne ed autoreferenza

Parlando di autoreferenza, abbiamo definito il concetto di "istanza corrente" di una classe definendo "oggetto attivo" l'istanza della classe in esecuzione durante la chiamata ad un metodo. La natura delle classi interne estende necessariamente questo concetto; di fatto, una classe interna possiede più di un'istanza corrente. La classe *Elenco* definita nell'esercizio precedente ha due istanze correnti: la propria e quella della classe *ElencoTelefonico*.

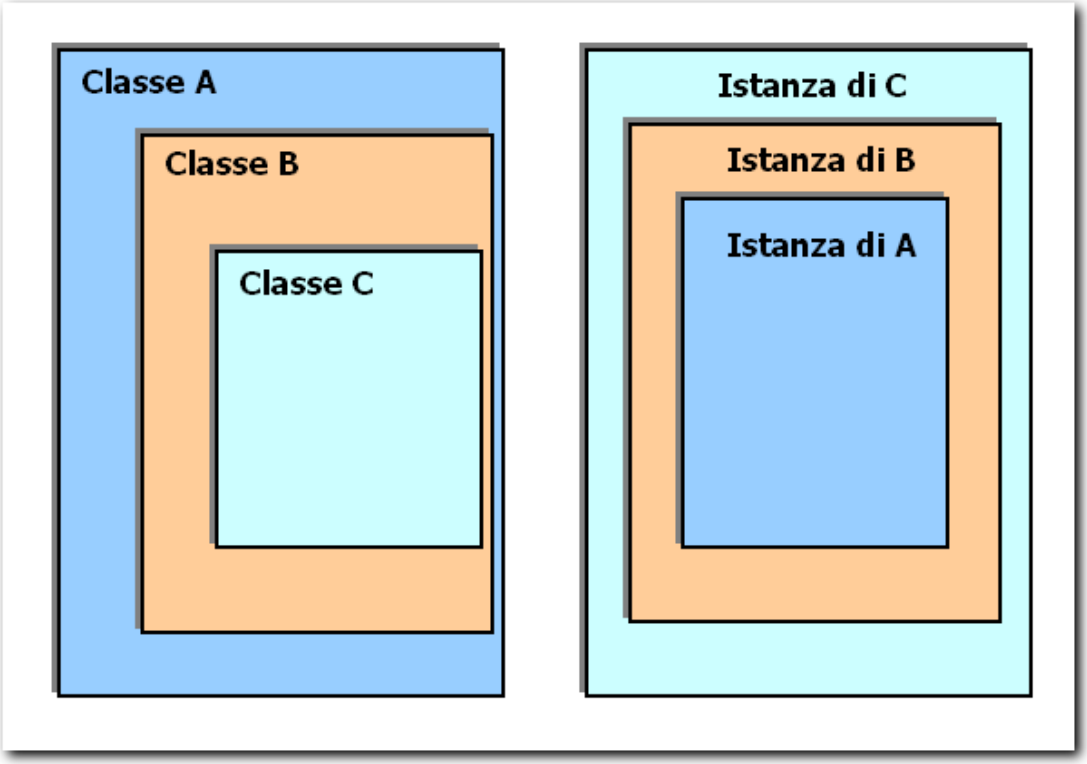


Figura 42: Istanze correnti di una classe interna

Più esplicitamente:

1. *Durante l'esecuzione del codice della classe interna C, esistono le istanze correnti di A,B,C;*
2. *Durante l'esecuzione del codice della classe interna B, esistono le istanze correnti di A,B;*
3. *Durante l'esecuzione del codice della classe A, non esistono altre istanze correnti differenti da quella dell'oggetto attivo;*
4. *Nel caso di esecuzione di un metodo statico della classe A, non esistono istanze correnti di nessun tipo.*

In generale, durante l'esecuzione di un metodo qualsiasi appartenente ad una classe interna *Ci* esistono: l'istanza corrente della di *Ci* e tutte le istanze correnti delle classi che incapsulano *Ci* sino ad arrivare alla classe di primo livello.
Detto questo, sorge spontaneo domandarsi come le classi interne influiscono sull'utilizzo della variabile reference ***this***.
Come tutte le classi, anche le classi interne utilizzano il meccanismo di autoreferenza implicita per determinare quale metodo o variabile utilizzare in caso di chiamata. Ma, se fosse necessario fare riferimento ad un particolare tipo di istanza corrente, deve essere utilizzata la variabile reference ***this*** preceduta dal nome della classe da riferire.
Nella figura precedente:

1. *Durante l'esecuzione del codice della classe interna C, B.this e A.this rappresentano i riferimenti rispettivamente alle istanze correnti delle classi A e B;*
2. *Durante l'esecuzione del codice della classe interna B, A.this rappresenta il riferimenti alla istanza correnti della classe A.*

Infine, la forma sintattica *nome_della_classe.**this*** è permessa poiché il linguaggio Java non consente di dichiarare classi interne con lo stesso nome della classe incapsulante.

5.20 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 è in esecuzione.
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 stringhe sullo standard output o sullo standard error.

```
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 producendo il codice di errore passato come argomento al metodo.
L'oggetto System fornisce anche il meccanismo per ottenere informazioni relative al sistema ospite mediante il metodo statico *System.getProperty(String)*, che ritorna il valore della proprietà di sistema richiesta o, in caso di assenza, ritorna il valore ***null***. Le proprietà, accessibili mediate questo metodo, possono variare a seconda del sistema su cui

l'applicazione è in esecuzione; la tabella seguente elenca il nome delle proprietà la cui definizione è garantita indipendentemente dalla Java Virtual Machine a partire dalla versione 1.2:

Proprietà di sistema	
Nome	**Descrizione**
file.separator	Separatore di file dipendente dalla piattaforma (Ad esempio "\" per Windows e "/" per LINUX).
java.class.path	Valore della variabile d'ambiente CLASSPATH.
java.class.version	Versione delle Java API.
java.home	Directory in cui è stato installato il Java Development Kit.
java.version	Versione dell'interprete Java.
java.vendor	Informazioni relative al produttore dell'interprete Java.
java.vendor.url	Indirizzo internet del produttore dell'interprete Java.
line.separator	Separatore di riga dipendente dalla piattaforma (Ad esempio "\r\n" per Windows e "\n" per LINUX).
os.name	Nome del sistema operativo
os.arch	Nome dell'architettura
os.version	Versione del sistema operativo
path.separator	Separatore di PATH dipendente dalla piattaforma (Ad esempio ";" per Windows e ":" per LINUX).
user.dir	Cartella di lavoro corrente.
user.home	Cartella "Home" dell'utente corrente.
user.name	Nome dell'utente connesso.

Nel prossimo esempio, utilizziamo il metodo in esame per ottenere tutte le informazioni disponibili relative al sistema che stiamo utilizzando:

```
class Sistema {
   public static void main(String[] argv) {

      System.out.println("file.separator= "+System.getProperty("file.separator") );
      System.out.println("java.class.path= "+System.getProperty("java.class.path") );
      System.out.println("java.class.version= "+System.getProperty("java.class.version") );
      System.out.println("java.home= "+System.getProperty("java.home") );
      System.out.println("java.version= "+System.getProperty("java.version") );
      System.out.println("java.vendor= "+System.getProperty("java.vendor") );
      System.out.println("java.vendor.url= "+System.getProperty("java.vendor.url") );
      System.out.println("os.name= "+System.getProperty("os.name") );
      System.out.println("os.arch= "+System.getProperty("os.arch") );
      System.out.println("os.version= "+System.getProperty("os.version") );
      System.out.println("path.separator= "+System.getProperty("path.separator") );
      System.out.println("user.dir= "+System.getProperty("user.dir") );
      System.out.println("user.home= "+System.getProperty("user.home") );
      System.out.println("user.name= "+System.getProperty("user.name") );
```

```
    }
}
```

Dopo l'esecuzione della applicazione sul mio computer personale, le informazioni ottenute sono elencate di seguito:

```
file.separator= \
java.class.path= E:\Together6.0\out\classes\mattone;E:\Together6.0\lib\javax.jar;
java.class.version= 47.0
java.home= e:\Together6.0\jdk\jre
java.version= 1.3.1_02
java.vendor= Sun Microsystems Inc.
java.vendor.url= http://java.sun.com/
os.name= Windows 2000
os.arch= x86
os.version= 5.1
path.separator= ;
user.dir= D:\PROGETTI\JavaMattone\src
user.home= C:\Documents and Settings\Massimiliano
user.name= Massimiliano
```

Per concludere, il metodo descritto fornisce un ottimo meccanismo per inviare informazioni aggiuntive ad una applicazione senza utilizzare la riga di comando come mostrato nel prossimo esempio:

```
class Argomenti1 {
    public static void main(String[] argv) {
        System.out.println("user.nome= "+System.getProperty("user.nome") );
        System.out.println("user.cognome= "+System.getProperty("user.cognome") );
    }
}
```

```
java -Duser.nome=Massimiliano -Duser.cognome=Tarquini Argomenti1

user.nome= Massimiliano
user.cognome= Tarquini
```

6 CONTROLLO DI FLUSSO E DISTRIBUZIONE DI OGGETTI

6.1 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 cicliche, quelle 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 un'istruzione speciale utilizzata dal compilatore per determinare la posizione su disco delle definizioni di classi da utilizzare nell'applicazione corrente.
Come C e C++, Java è un linguaggio indipendente dagli spazi, in altre parole, l'indentazione del codice di un programma ed eventualmente l'uso di più di una riga di testo sono opzionali.

6.2 Istruzioni per il controllo di flusso

Espressioni booleane ed istruzioni per il controllo di flusso forniscono al programmatore il meccanismo per comunicare alla Java Virtual Machine se e come eseguire blocchi di codice, condizionatamente ai meccanismi decisionali.

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	Determina quale tra due blocchi di codice sia 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]
}

oppure

istruzione --> controllo_di_flusso
istruzione

6.3 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 *(condizione)*
istruzione1;
istruzione2;

-->

if *(condizione)*
{
istruzione;
[istruzione]
}
istruzione;
[istruzione]

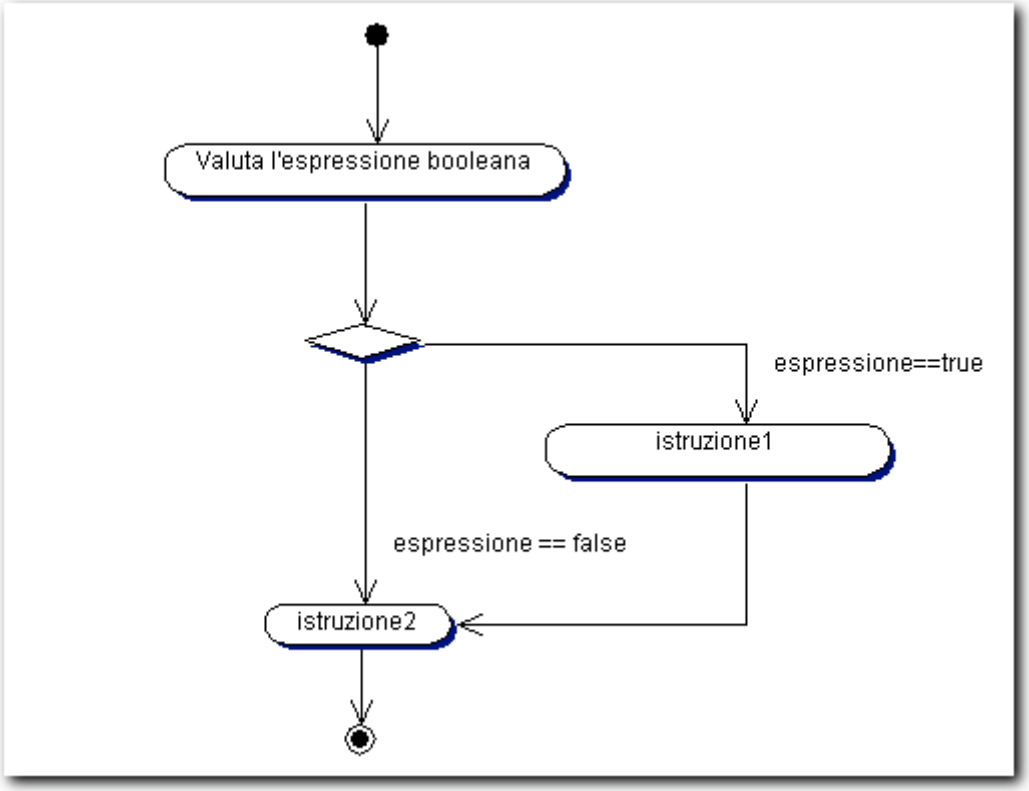


Figura 43: Diagramma di attività dell'istruzione IF

Nella regola definita, *condizione* rappresenta un'istruzione booleana valida. Di fatto, se l'espressione restituisce il valore ***true***, sarà eseguito il blocco di istruzioni immediatamente successivo, in caso contrario il controllo passerà alla prima istruzione successiva al blocco **if**, come schematizzato nella *Figura 43*. Un esempio di istruzione if è il seguente:

```
int x;
.....
if(x>10)
{
        x=0;
}
x=1;
```

Nell'esempio, se il valore di x è strettamente maggiore di 10, verrà eseguito il blocco di istruzioni di **if** ed il valore di x verrà impostato a 0 e successivamente ad 1. In caso contrario, il flusso delle istruzioni salterà direttamente al blocco di istruzioni immediatamente successivo al blocco if ed il valore della variabile x verrà impostato direttamente a 1.

6.4 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 di seguito, consente di decidere quale, tra due blocchi di codice, eseguire.

```
if (condizione)                    if (condizione)
        istruzione1          -->   {
else istruzione2                       istruzione;
istruzione3                            [istruzione]
                                   }
                                   else
                                   {
                                       istruzione;
                                       [istruzione]
                                   }
                                   istruzione;
                                   [istruzione]
```

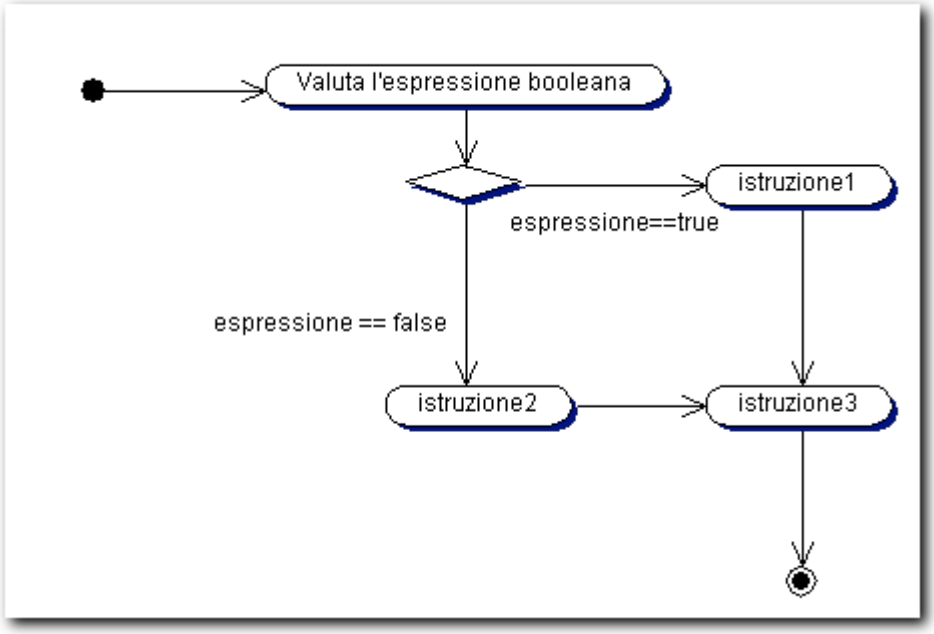


Figura 44: Diagramma di attività dell'istruzione IF-ELSE

Se *condizione* restituisce il valore ***true,*** sarà eseguito il blocco di istruzioni di **if**, altrimenti il controllo sarà passato ad ***else*** e sarà eseguito il secondo blocco di istruzioni. Al termine, il controllo di flusso passa alla *istruzione3*.
Di seguito un esempio:

```
if(y==3)
{
```

```
   y=12;
}
else
{
   y=0;
}
```

In questo caso, se l'espressione booleana ritorna valore ***true***, allora saranno eseguite le istruzioni contenute nel blocco di istruzioni di ***if***, altrimenti saranno eseguite le istruzioni contenute nel blocco di ***else***.

6.5 Istruzioni if, if-else annidate

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

istruzione -> controllo_di_flusso
controllo_di_flusso
istruzione

controllo_di_flusso ->if(condizione)
istruzione

oppure

controllo_di_flusso ->if(condizione)
istruzione
else
istruzione

Da cui deriviamo una possibile regola sintattica per costruire blocchi ***if*** annidati:

```
if(condizione1)
{
        istruzione1
        if (condizione2)
        {
             istruzione2
            if (condizione3)
           {
                istruzione3
           }
```

```
    }
}
istruzione4
```

La figura 45, schematizza l'esecuzione del blocco definito:

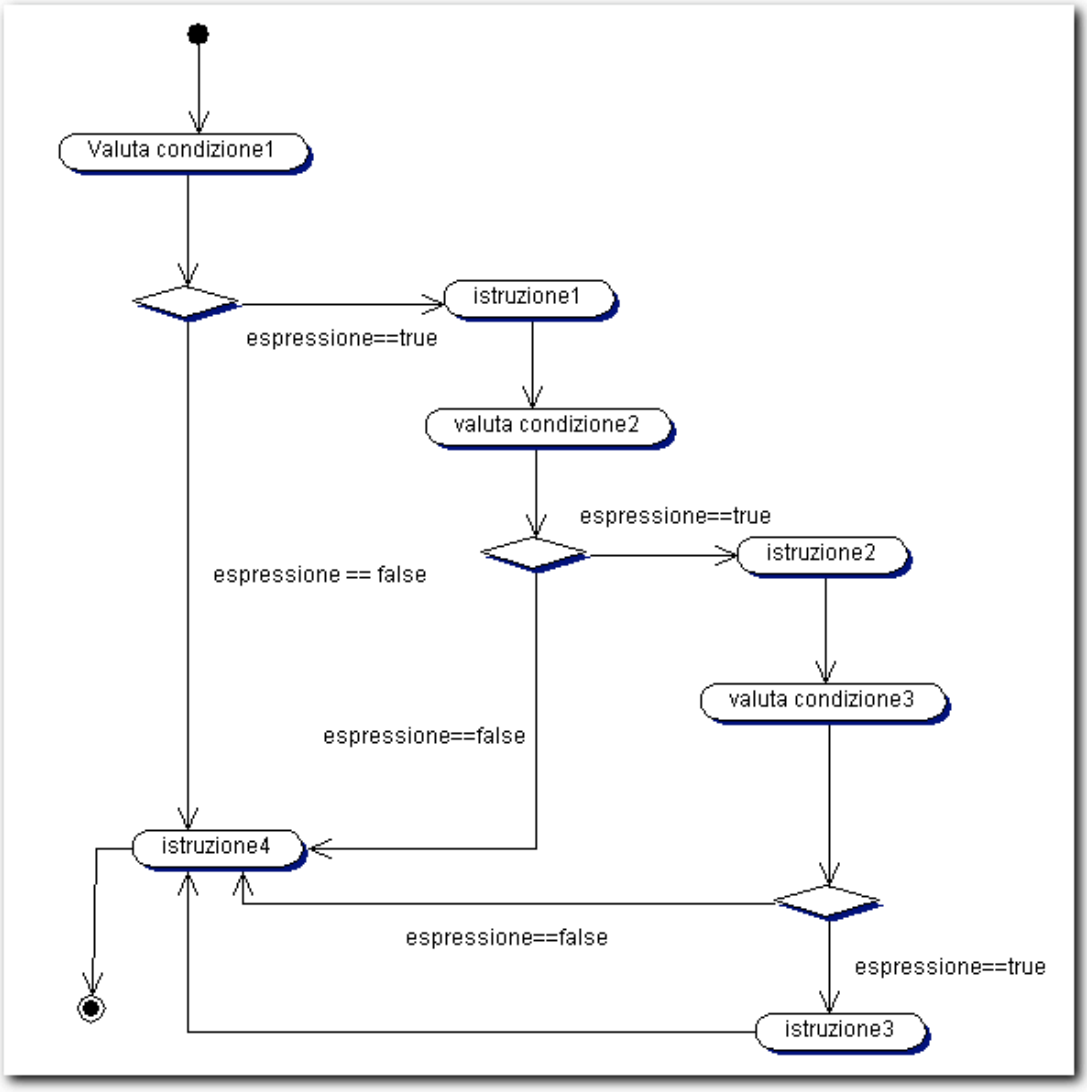


Figura 45: Istruzioni If annidate

6.6 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 procedendo dall'alto verso il basso: se almeno una delle condizioni restituisce il valore ***true*** sarà eseguito il blocco di istruzioni relativo.

Se nessuna delle condizioni si dovesse verificare, allora sarebbe eseguito il blocco ***else*** finale.

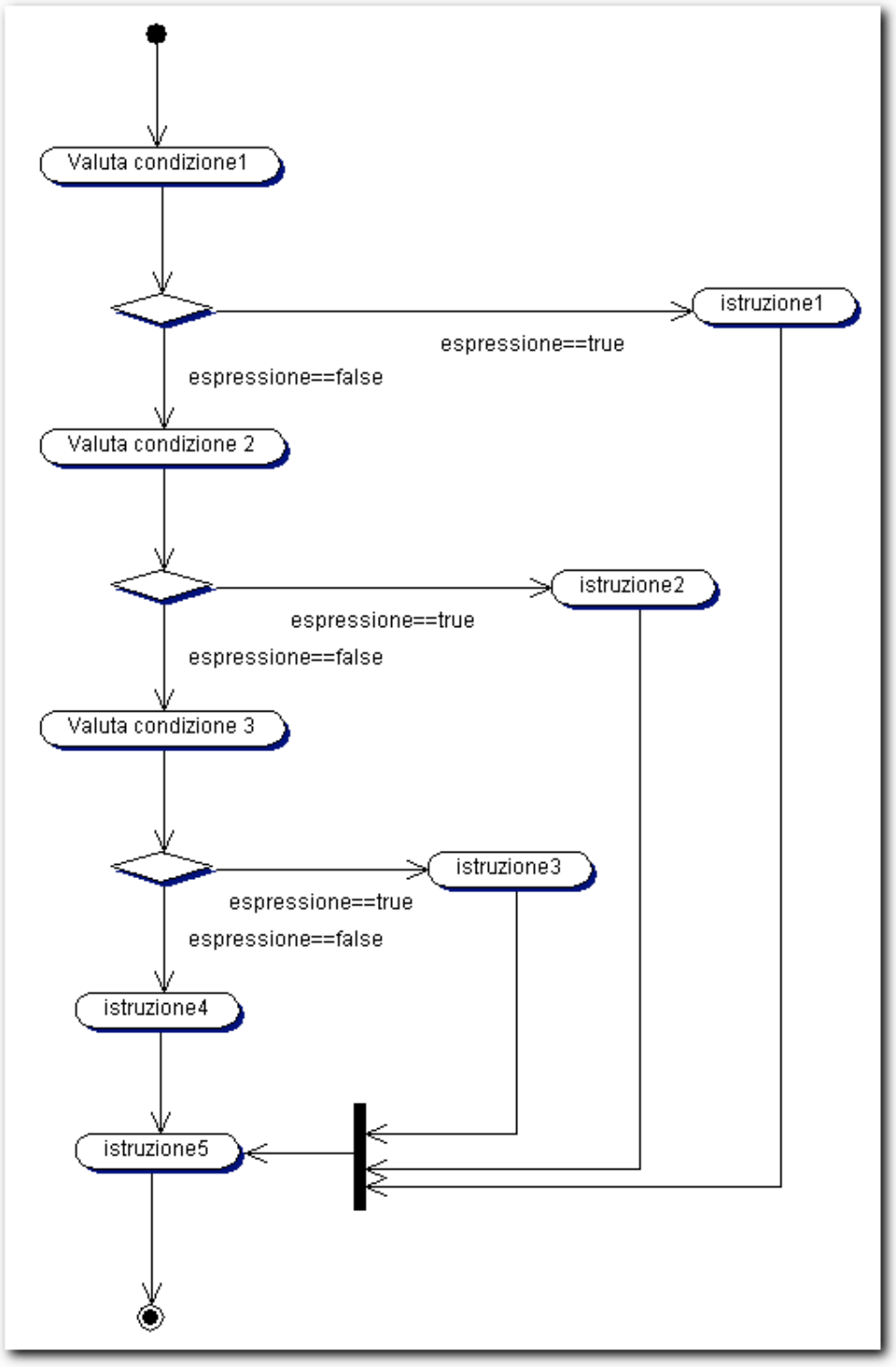


Figura 46: Catene If-Else-If annidate

La figura 46 schematizza la sequenza delle attività svolte dalla seguente catena if-else-if:

```
if(condizione1)
{
        istruzione1
}
else if (condizione2)
         {
            istruzione2
       }
          else if (condizione3)
                 {
                    istruzione3
              }
             else
             {
                istruzione4
             }
istruzione5
```

Nel prossimo esempio, utilizziamo una catena if-else-if per stampare a video il valore di un numero intero da zero a dieci in forma di stringa. Il valore da stampare è passato all'applicazione dalla riga di comando. Nel caso in cui il numero sia maggiore di dieci, l'applicazione stampa a video la stringa: "Impossibile stampare un valore maggiore di dieci"

```
/**
 * Questa classa stampa a video il valore del numero intero passato
 * sulla riga di comando in forma di stringa.
 */
class ValueOf {
   public static void main(String[] argv) {
      int intero = Integer.parseInt(argv[0]);
      if (intero == 0) {
         System.out.println("Zero");
      }
      else if (intero == 1) {
         System.out.println("Uno");
      }
      else if (intero == 2) {
         System.out.println("Due");
      }
      else if (intero == 3) {
         System.out.println("Tre");
      }
      else if (intero == 4) {
         System.out.println("Quattro");
```

```
        }
        else if (intero == 5) {
            System.out.println("Cinque");
        }
        else if (intero == 6) {
            System.out.println("Sei");
        }
        else if (intero == 7) {
            System.out.println("Sette");
        }
        else if (intero == 8) {
            System.out.println("Otto");
        }
        else if (intero == 9) {
            System.out.println("Nove");
        }
        else if (intero == 10) {
            System.out.println("Dieci");
        }
        else {
            System.out.println("Impossibile stanpare un valore maggiore di dieci");
        }
    }
}
```

6.7 L'istruzione switch

L'esempio del paragrafo precedente rende evidente quanto sia complesso scrivere o, leggere, codice java utilizzando catene if-else-if arbitrariamente complesse. Per far fronte al problema, Java fornisce al programmatore un'istruzione di controllo di flusso che, specializzando la catena if-else-if rende più semplice la programmazione di un'applicazione.
L'istruzione ***switch*** è utile in tutti quei casi in cui sia necessario decidere tra scelte multiple, prese in base al controllo di una sola variabile. La sintassi dell'istruzione è la seguente:

```
switch (espressione)
{
        case espressione_costante1:
                        istruzione1
                        break_opzionale
        case espressione_costante2:
                        istruzione2
                        break_opzionale
        case espressione_costante3:
                        istruzione3
                        break_opzionale
        default:
                        istruzione4
}
```

istruzione5

Nella forma di Backus-Naur definita, *espressione* rappresenta ogni espressione valida che produca un intero ed *espressione_costante* un'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 l'inclusione opzionale della parola chiave ***break*** seguita da ";" .

Nella Figura 47, è schematizzati il diagramma delle attività della istruzione switch.

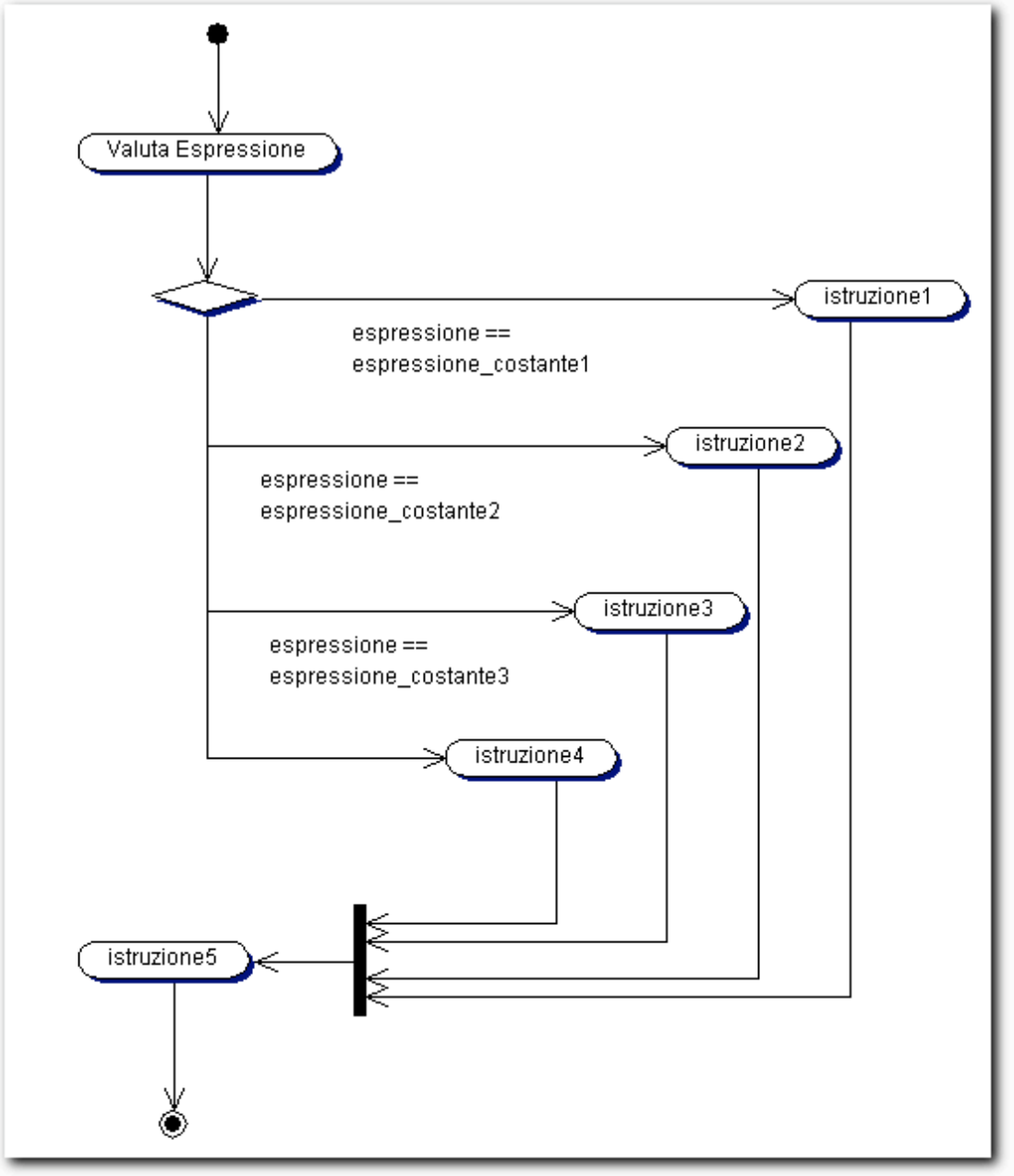


Figura 47: L'istruzione switch

In generale, dopo la valutazione di *espressione*, il controllo dell'applicazione salta al primo blocco **case** tale che

espressione == espressione_costante

ed esegue il relativo blocco di codice. Nel caso in cui il blocco termini 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** successivi. 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'esempio del paragrafo precedente può essere riscritto nel modo seguente:

```
/**
 * Questa classa stampa a video il valore del numero
 * intero passato sulla riga di comando in forma di stringa utilizzando
 * l'istruzione switch in sostituzione di una catena if-else-if.
 */
class ValueOf2 {
   public static void main(String[] argv) {
      int intero = Integer.parseInt(argv[0]);
      switch (intero) {
         case 0:
            System.out.println("Zero");
            break;
         case 1:
            System.out.println("Uno");
            break;
         case 2:
            System.out.println("Due");
            break;
         case 3:
            System.out.println("Tre");
            break;
         case 4:
            System.out.println("Quattro");
            break;
         case 5:
            System.out.println("Cinque");
            break;
         case 6:
            System.out.println("Sei");
            break;
         case 7:
            System.out.println("Sette");
            break;
         case 8:
            System.out.println("Otto");
            break;
         case 9:
            System.out.println("Nove");
            break;
         case 10:
            System.out.println("Dieci");
            break;
         default:
            System.out.println("Impossibile stanpare un valore maggiore di dieci");
      }
   }
}
```

6.8 L'istruzione while

Un'istruzione **while** permette l'esecuzione ripetitiva di un blocco di istruzioni, utilizzando un'espressione booleana per determinare se eseguirlo o no, eseguendolo quindi fino a che l'espressione booleana non restituisce il valore ***false***. La sintassi per questa istruzione è la seguente:

```
while (espressione){
        istruzione1
}
istruzione2
```

dove, *espressione* è una espressione valida che restituisce un valore booleano.

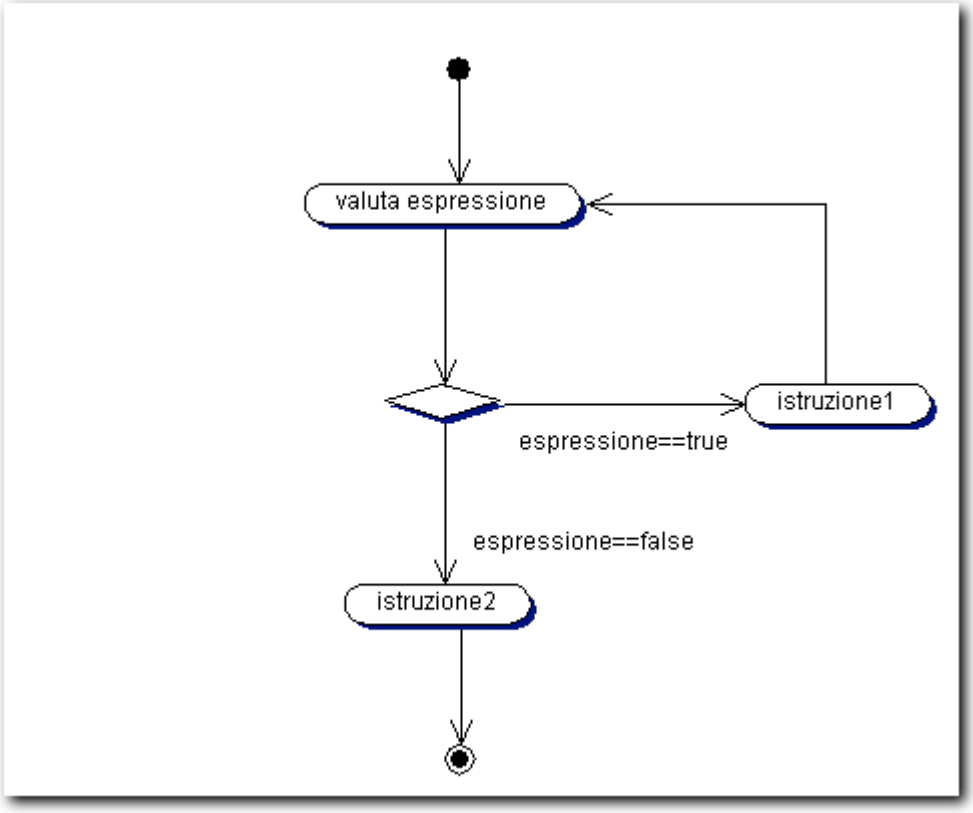


Figura 48: L'istruzione while

In dettaglio, un'istruzione **while** controlla il valore dell'espressione booleana: se il risultato restituito è **true** sarà eseguito il blocco di codice di **while**. Alla fine dell'esecuzione è nuovamente controllato il valore dell'espressione

booleana, per decidere se ripetere l'esecuzione del blocco di codice o passare il controllo dell'esecuzione alla prima istruzione successiva al blocco while. Applicando le regole di espansione, anche in questo caso otteniamo la forma annidata:

while *(espressione)*
    while *(espressione)*
        istruzione

Il codice di esempio utilizza la forma annidata dell'istruzione **while**:

```
int i=0;
while(i<10)
{
        j=10;
        while(j>0)
        {
            System.out.println("i="+i+"e j="+j);
            j--;
        }
        i++;
}
```

6.9 L'istruzione do-while

Un'alternativa all'istruzione **while** è rappresentata dall'istruzione **do-while** che, a differenza della precedente, controlla il valore dell'espressione booleana alla fine del blocco di istruzioni. La sintassi di **do-while** è la seguente:

do *{*
    istruzione1;
} ***while*** *(espressione);*
istruzione2;

A differenza dell'istruzione **while**, ora il blocco di istruzioni sarà eseguito sicuramente almeno una volta, come appare evidente nella prossima figura.

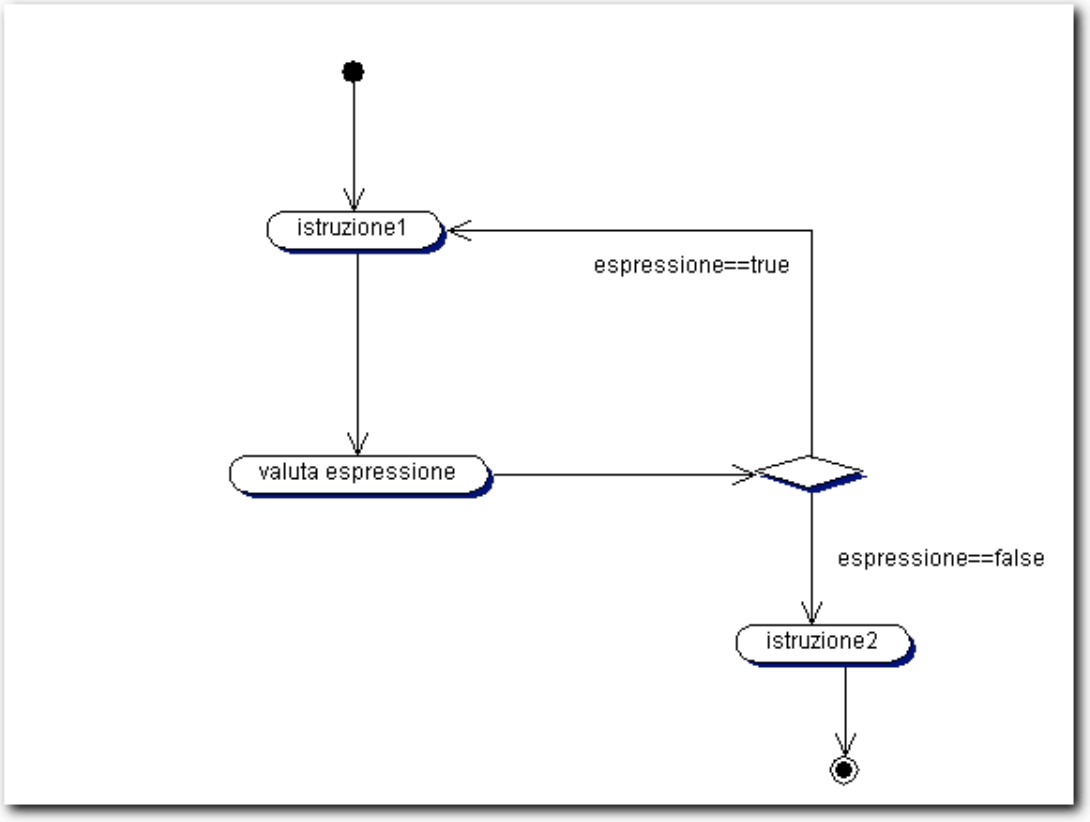


Figura 49: L'istruzione do-while

6.10 L'istruzione for

Quando definiamo un ciclo di istruzioni, accade spesso la situazione in cui tre operazioni distinte concorrono all'esecuzione del blocco di istruzioni. Consideriamo il ciclo di 10 iterazioni:

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

Nell'esempio, come prima operazione, l'applicazione inizializza una variabile per il controllo del ciclo, quindi viene eseguita un'espressione condizionale per decidere se eseguire o no il blocco di istruzioni dell'istruzione **while**, infine la variabile è aggiornata in modo tale che possa determinare la fine del ciclo.
Tutte queste operazioni sono incluse nella stessa istruzione condizionale dell'istruzione **for**:

for*(inizializzazione ; condizione ; espressione){*
istruzione1

}
istruzione2

che ha forma annidata:

for*(inizializzazione ; condizione ; espressione){*
*****for****(inizializzazione ; condizione ; espressione){*
istruzione1
}
}
istruzione2

dove: *inizializzazione* rappresenta la definizione della variabile per il controllo del ciclo, *condizione* l'espressione condizionale che controlla l'esecuzione del blocco di istruzioni ed *espressione* contiene le regole per l'aggiornamento della variabile di controllo.

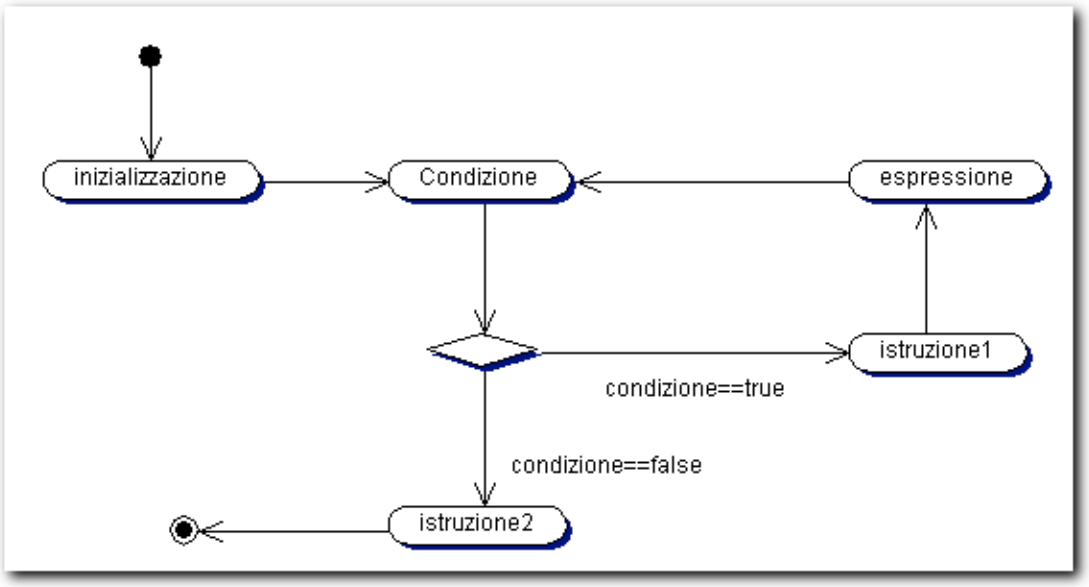


Figura 50: L'istruzione for

In una istruzione **for**, la condizione viene sempre controllata all'inizio del ciclo. Nel caso in cui restituisca un valore **false**, il blocco di istruzioni non verrà mai eseguito altrimenti, viene eseguito il blocco di istruzioni, viene aggiornato il valore della variabile di controllo e infine viene nuovamente valutata la condizione come mostrato nella Figura 50.
Il ciclo realizzato nel precedente esempio utilizzando il comando **while**, può essere riscritto utilizzando il comando **for** nel seguente modo:

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

6.11 Istruzione *for* nei dettagli

L'istruzione **for**, in Java come in C e C++, è un'istruzione molto versatile poiché 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( ;  ; ){
        istruzione1
}
istruzione2
```

Questa forma non utilizza né variabili di controllo, né istruzioni di assegnamento né 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 quest'istruzione può essere quindi descritta della regola:

```
for([inizializzazione][, inizializzazione] ; [condizione]; [espressione] [,espressione] ) {
        istruzione1
}
istruzione2
```

6.12 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 di esecuzione dell'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.

6.13 L'istruzione break

L'istruzione **break** consente di forzare l'uscita da un ciclo aggirando il controllo sull'espressione booleana e provocandone l'uscita immediata, in modo del tutto simile a quanto già visto parlando dell'istruzione **switch**. Per comprenderne meglio il funzionamento, esaminiamo il prossimo esempio:

```
int controllo =  0;
while(controllo<=10)
{
        controllo ++;
}
```

Nell'esempio, è definito il semplice ciclo **while** utilizzato per sommare 1 alla variabile di controllo stessa, fino a che il suo valore non sia uguale a 10. Lo stesso esempio, può essere riscritto nel seguente modo utilizzando l'istruzione **break**:

```
int controllo =  0;
while(true)
{
        controllo ++;
        if(controllo==10)
                break;
}
```

L'uso di quest'istruzione, tipicamente, è legato a casi in cui sia necessario poter terminare l'esecuzione di un ciclo a prescindere dai valori delle variabili di controllo utilizzate. Queste situazioni occorrono in quei casi in cui è impossibile utilizzare un parametro di ritorno come operando all'interno dell'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[8], 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 sia catturato l'errore.

6.14 L'istruzione continue

A differenza del caso precedente, l'istruzione **continue** 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 all'espressione booleana che ne determina l'esecuzione. Un esempio può aiutarci a chiarire e idee:

```
int i=-1;
int pairs=0;
while(i<20)
{
        i++;
        if((i%2)!=0) continue;
        pairs ++;
}
```

L'esempio calcola quante occorrenze di numeri pari ci sono in una sequenza di interi compresa tra 1 e 20 memorizzando il risultato in una variabile di tipo **int** chiamata "pairs". Il ciclo while è controllato dal valore della variabile *i* inizializzata a –1. L'istruzione riportata sulla sesta riga del codice, effettua un 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 tre. In caso contrario viene aggiornato il valore di pairs.
Nella prossima immagine viene schematizzato il diagramma delle attività del blocco di codice descritto nell'esempio.

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

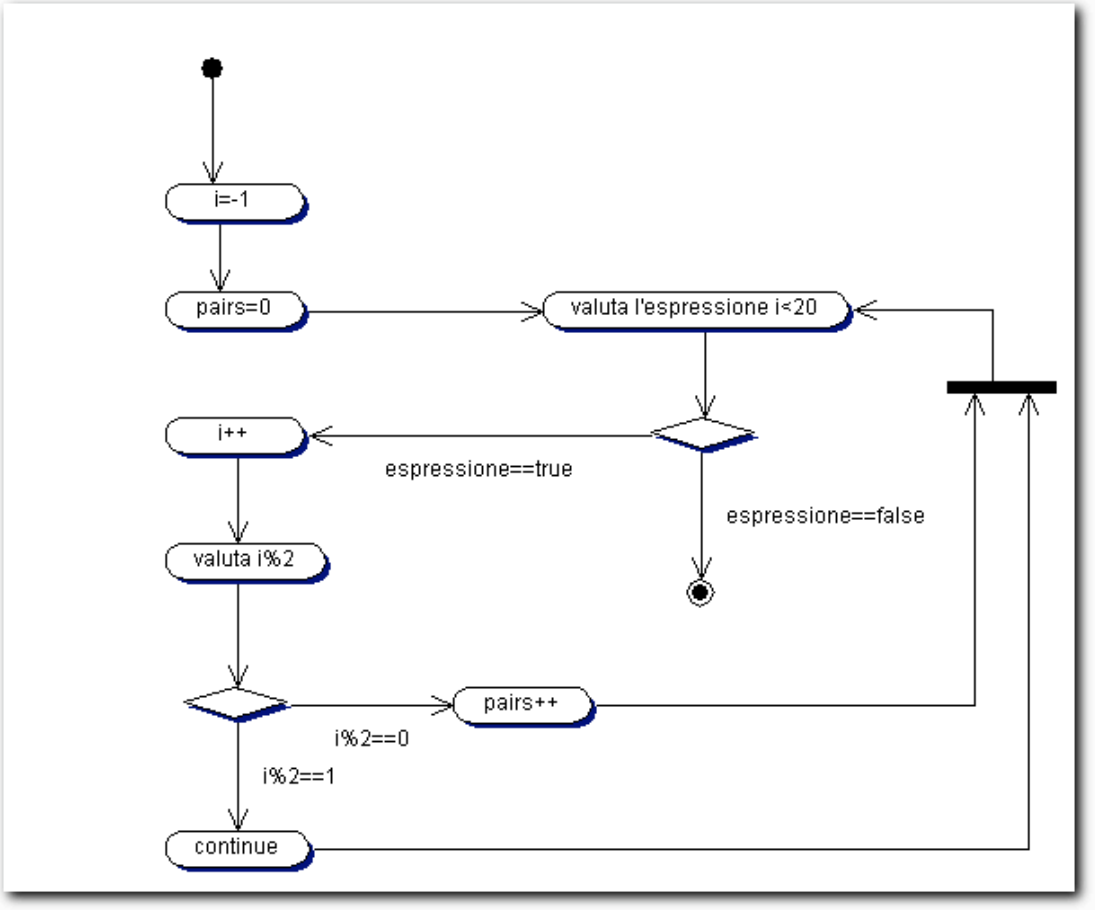


Figura 51: L'operazione continue

6.15 L'istruzione return

Return, rappresenta l'ultima istruzione di ramificazione ed è 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 è 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**.

6.16 Package Java

I package sono raggruppamenti di definizioni di classi sotto uno stesso nome e rappresentano il meccanismo utilizzato da Java per localizzare le definizioni di classi, durante la compilazione o durante l'esecuzione di un'applicazione.

Per questo motivo, i package Java possono essere paragonati a quelle che, per altri linguaggi di programmazione, sono chiamate librerie.
I package Java organizzano le classi secondo una struttura gerarchica per l'assegnamento dei nomi: questa caratteristica impedisce che due programmatori assegnino lo stesso nome a due differenti definizioni di classe. Di fatto, utilizzare i package comporta molti vantaggi:

1. *le classi possono essere mascherate all'interno dei package di appartenenza, facilitando l'incapsulamento anche a livello di file;*
2. *le classi di un package possono condividere dati e metodi con classi di altri package;*
3. *i package forniscono un meccanismo efficace per distribuire oggetti.*

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

6.17 Assegnamento di nomi a package

Nei capitoli precedenti abbiamo affermato che, la definizione della classe *MiaClasse* deve essere salvata nel file *MiaClasse.java* e ogni file con estensione ".java" deve contenere una sola classe. Ogni definizione di classe è detta unità di compilazione.
I package combinano unità di compilazione in un unico archivio, la cui struttura gerarchica rispetta quella del file system del computer: il nome completo di una unità di compilazione appartenente ad un package è determinato dal nome della classe, anteceduto dai nomi dei package appartenenti ad un ramo della gerarchia, separati tra loro dal carattere punto. Ad esempio, se la classe MiaClasse appartiene alla gerarchia definita nella figura seguente, il suo nome completo è:

esempi.capitolosei.classes.MiaClasse.

Le specifiche del linguaggio Java identificano alcuni requisiti utili alla definizione dei nomi dei package:

1. *I nomi dei package debbono contenere solo caratteri minuscoli;*
2. *Tutti i package che iniziano con "java." Contengono le classi appartenenti alle Java Core API;*
3. *Package conteneti classi di uso generale debbono iniziare con il nome della azienda proprietaria del codice.*

Una volta definito il nome di un package, affinché un'unità di compilazione possa essere archiviata al suo interno, è necessario aggiungere un'istruzione **package** all'inizio del codice sorgente della definizione di classe contenente il

nome completo del package che la conterrà. Per esempio, all'inizio di ogni file contenente la definizione di una classe del package esempi.capitolosei.classes, è necessario aggiungere la riga:

package esempi.capitolosei.classes;

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.

6.18 Creazione dei package

Dopo aver definito il nome di un package, deve essere creata su disco la struttura di cartelle che rappresenti la gerarchia definita dai nomi. Ad esempio, le classi appartenenti al package esempi.capitolosei.classes devono essere memorizzate in una gerarchia di directory che termina con esempi/capitolosei/classes localizzata in qualunque punto del disco (es. C:/esempi/capitolosei/classes schematizzato nella prossima figura).

Figura 52: definizione di un package

Mediante la variabile di ambiente CLASSPATH sarà possibile comunicare alla Java Virtual Machine come poter trovare il package definito inserendo il percorso fino alla cartella di primo livello nella gerarchia dei package. Nel nostro esempio:

CLASSPATH =c:\

6.19 Distribuzione di classi

Organizzare classi Java mediante package non ha come unico beneficio quello di organizzare definizioni di oggetti secondo una struttura logica, piuttosto i package rappresentano un meccanismo efficace per la distribuzione di un'applicazione Java. Per capire le ragioni di quest'affermazione, dobbiamo fare un salto indietro nel tempo, a quando gli analisti del Green Group tracciarono le prime specifiche del linguaggio Java.
Essendo Java un linguaggio nato per la creazione di applicazioni internet, si rese necessario studiare un meccanismo in grado di ridurre al minimo i tempi di attesa dell'utente durante il trasferimento delle classi dal server al computer dell'utente. Le prime specifiche del linguaggio, stabilirono che i package Java potessero essere distribuiti in forma di archivi compressi in formato zip, riducendo drasticamente i tempi necessari al trasferimento dei file attraverso la rete. Se all'inizio doveva essere uno stratagemma per ridurre le dimensioni dei file, con il passare del tempo la possibilità di distribuire applicazioni Java di tipo enterprise mediante archivi compressi si è rivelata una caratteristica alquanto vantaggiosa, tanto da stimolare la SUN nella definizione di un formato di compressione chiamato "JAr" o "Java Archive" che, basandosi sull'algoritmo zip, inserisce all'interno dell'archivio compresso un file contente informazioni relative all'utilizzo delle definizioni di classe.
Creare un archivio secondo questo formato, è possibile utilizzando l'applicazione jar.exe che può essere trovata nella cartella bin all'interno della cartella di installazione dello Java SDK.
La applicazione jar.exe può essere eseguita tramite la riga di comando ed ha la seguente sintassi:

jar *[opzioni] [file_manifest] destinazione file_di_input [file_di_input]*

Oltre ai file da archiviare definiti dall'opzione *[file_di_input]*, l'applicazione jar accetta dalla riga di comando una serie di parametri opzionali (*[opzioni]*) necessari al programmatore a modificare le decisioni adottate dalla applicazione durante la creazione dell'archivio compresso. Non essendo scopo del libro quello di scendere nei dettagli di questa applicazione, esamineremo solo le opzioni più comuni, rimandando alla documentazione distribuita con il Java SDK la trattazione completa del comando in questione.

c: crea un nuovo archivio vuoto;

v: Genera sullo standard error del terminale un output molto dettagliato.

f: L'argomento *destinazione* del comando jar si riferisce al nome dell'archivio jar che deve essere elaborato. Questa opzione indica alla applicazione che *destinazione* si riferisce ad un archivio che deve essere creato.

x: estrae tutti i file contenuti nell'archivo definito dall'argomento identificato da *destinazione*. Questa opzione indica alla applicazione che *destinazione* si riferisce ad un archivio che deve essere estratto.

Passiamo quindi ad un esempio pratico. Volendo comprimere il package Java creato nell'esempio del capitolo precedente, il comando da eseguire dovrà essere il seguente:

```
C:\>jar cvf mattone.jar esempi
aggiunto manifesto
aggiunta in corso di: esempi/(in = 0) (out= 0)(archiviato 0%)
aggiunta in corso di: esempi/capitolosei/(in = 0) (out= 0)(archiviato 0%)
aggiunta in corso di: esempi/capitolosei/classes/(in = 0) (out= 0)(archiviato 0%)
```

Per concludere, è necessaria qualche altra informazione relativamente all'uso della variabile d'ambiente CLASSPATH. Nel caso in cui una applicazione Java sia distribuita mediante uno o più package Java compressi in formato jar, sarà comunque necessario specificare alla JVM il nome dell'archivio o degli archivi contenenti le definizioni di classi necessarie. Nel caso dell'esempio precedente, la variabile di ambiente CLASSPATH dovrà essere impostata nel modo seguente:

```
CLASSPATH =c:\mattone.jar
```

6.20 Il modificatore public

Le specifiche del linguaggio Java stabiliscono che un oggetto possa essere utilizzato solo da oggetti appartenenti al suo stesso package. Per comprendere meglio il concetto, utilizziamo un esempio concreto definendo due classi: la prima, chiamata *Autore* appartenente al package *mattone.esempi.provapublic.autori*. La seconda, chiamata EsempioPublic appartenente al package *mattone.esempi.provapublic*.
Nella figura seguente è riportato il *component diagram* che descrive la struttura dei package che utilizzeremo nel nostro esempio.

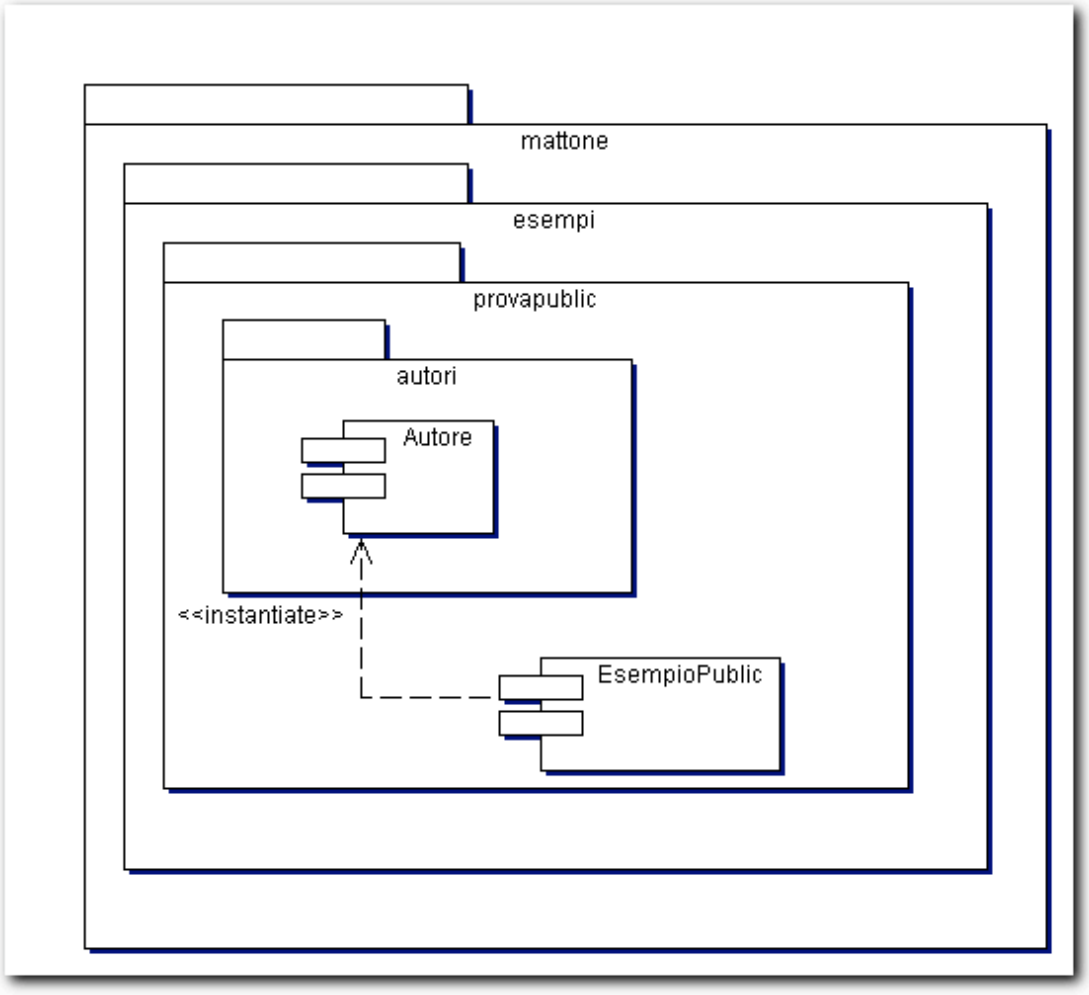


Figura 53: Component Diagram del package mattone.esempi.provapublic

Le due definizioni di classe sono riportate di seguito.

```
package mattone.esempi.provapublic.autori;

class Autore {
   String Nome()
   {
      return "Mario";
   }
    String Cognome()
   {
      return "Rossi";
   }
}
```

```
package mattone.esempi.provapublic;

public class EsempioPublic {
   mattone.esempi.provapublic.autori.Autore AutoreDelLibro = new Autore();
   public void StampaNomeAutore()
   {
```

```
        System.out.println(AutoreDelLibro.Nome());
    }
}
```

La classe *EsempioPublic* alloca un oggetto di tipo Autore e nel metodo *StampaNomeAutore()* visualizza il valore della stringa tornata dal metodo *Nome()* della classe allocata. Se tentassimo di eseguire il compilatore java, il risultato della esecuzione sarebbe il seguente:

```
javac -classpath .;.\; mattone/esempi/provapublic/EsempioPublic.java
mattone/esempi/provapublic/EsempioPublic.java:5: mattone.esempi.provapublic.autori.Autore is
not public in mattone.esempi.provapublic.autori; cannot be accessed from outside package
import mattone.esempi.provapublic.autori.Autore;
                                  ^
mattone/esempi/provapublic/EsempioPublic.java:7: mattone.esempi.provapublic.autori.Autore is
not public in mattone.esempi.provapublic.autori; cannot be accessed from outside package
    Autore AutoreDelLibro = new Autore();
    ^
mattone/esempi/provapublic/EsempioPublic.java:7: mattone.esempi.provapublic.autori.Autore is
not public in mattone.esempi.provapublic.autori; cannot be accessed from outside package
    Autore AutoreDelLibro = new Autore();
                            ^
mattone/esempi/provapublic/EsempioPublic.java:7: Autore() is not public in
mattone.esempi.provapublic.autori.Autore; cannot be accessed from outside package
    Autore AutoreDelLibro = new Autore();
                          ^
mattone/esempi/provapublic/EsempioPublic.java:10:    Nome() is not public in
mattone.esempi.provapublic.autori.Autore; cannot be accessed from outside package
        System.out.println(AutoreDelLibro.Nome());
                                  ^
5 errors
```

Le righe evidenziate segnalano che si sono verificati degli errori, dovuti al fatto che si sta tentando di utilizzare una classe inaccessibile al di fuori del suo package di definizione.
Modifichiamo ora la classe Autore aggiungendo il modificare ***public*** prima della dichiarazione del nome della classe e dei due metodi membro.

```
package mattone.esempi.provapublic.autori;

public class Autore {
    public String Nome()
    {
        return "Mario";
    }
     public String Cognome()
    {
        return "Rossi";
    }
}
```

La compilazione ora avrà buon fine: la classe autore è ora visibile anche al di fuori del suo package di definizione.

Come già accennato, le specifiche del linguaggio richiedono che il codice sorgente di una classe sia memorizzato in un file avente lo stesso nome della classe (incluse maiuscole e minuscole), ma con estensione ".java".
Per essere precisi, la regola completa deve essere enunciata come segue:

*"Il codice sorgente di una classe **pubblica** deve essere memorizzato 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 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 da parte del programmatore, e quella del bytecode da parte della JVM. Supponiamo, come esempio, di aver definito le 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 del file A.java.
Al momento della compilazione del file A.java, il compilatore creerà una classe per ogni definizione contenuta nel file: A.class, B.class e C.class .
Questa impostazione, per quanto bizarra 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 classi B e C rappresentano solo l'implementazione di dettagli e quindi, 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 solo non è strettamente necessario, ma è dannoso dal punto di vista della leggibilità del codice sorgente. Sarà comunque compito del programmatore scegliere in che modo memorizzare le definizioni delle classi all'interno di un package.

6.21 L'istruzione import

Il runtime di Java fornisce un ambiente completamente dinamico. Di fatto, le classi non sono caricate fino a che non sono referenziate per la prima volta durante l'esecuzione dell'applicazione. Questo meccanismo consente di ricompilare singole classi senza dover necessariamente ricaricare intere applicazioni.
Poiché il bytecode di ogni definizione di classe Java è memorizzato in un unico file avente lo stesso nome della classe, ma con estensione ".class", la virtual machine può trovare i file binari appropriati cercando nelle cartelle specificate nella variabile di ambiente CLASSPATH; inoltre, poiché 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 di appartenenza di una classe, è quello di specificarne il nome ad ogni chiamata alla classe. In questo caso diremo che stiamo utilizzando nomi qualificati[9], come mostrato nel codice sorgente della classe EsempioPublic che fa riferimento alla classe Autore utilizzandone il nome qualificato *mattone.esempi.provapublic.autori.Autore*.

```
package mattone.esempi.provapublic;

public class EsempioPublic {
   mattone.esempi.provapublic.autori.Autore AutoreDelLibro = new Autore();
   public void StampaNomeAutore()
   {
      System.out.println(AutoreDelLibro.Nome());
   }
}
```

L'uso di nomi qualificati non è sempre comodo, soprattutto quando i package sono organizzati con gerarchie a molti livelli. Per venire incontro al programmatore, Java consente di specificare una volta per tutte il nome qualificato di una classe all'inizio del file contenente la definizione di classe, 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. Utilizzando questa istruzione, la classe *EsempioPublic* può essere riscritta nel modo seguente:

```
import mattone.esempi.provapublic.autori.Autore;

package mattone.esempi.provapublic;

public class EsempioPublic {
   Autore AutoreDelLibro = new Autore();
   public void StampaNomeAutore()
   {
      System.out.println(AutoreDelLibro.Nome());
   }
}
```

La classe sarà in grado di risolvere il nome di Autore ogni volta che sia necessario, semplicemente utilizzando il nome di classe *Autore*.
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 del carattere fantasma '*' che identifica tutte le classi pubbliche appartenenti ad un package.

```
import mattone.esempi.provapublic.autori.*;
```

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

```
package mattone.esempi.provapublic;

public class EsempioPublic {
   Autore AutoreDelLibro = new Autore();
   public void StampaNomeAutore()
   {
      System.out.println(AutoreDelLibro.Nome());
   }
```

La sintassi dell'istruzione **import**, non consente altre forme e non può essere utilizzata per caricare solo porzioni di package. Per esempio, la forma

```
import mattone.esempi.provapublic.autori.Au*;
```

non è consentita.

7 INCAPSULAMENTO

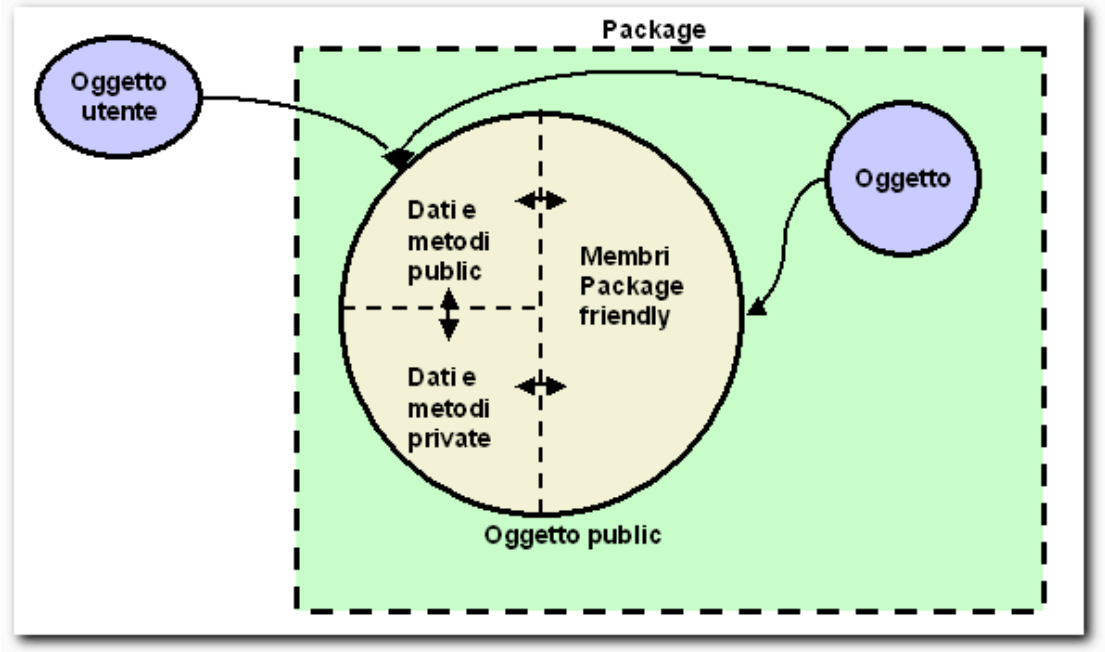
7.1 Introduzione

L'incapsulamento di oggetti è il processo di mascheramento dei dettagli implementativi di un oggetto ad altri oggetti, con lo scopo di proteggere porzioni di codice o dati critici. I programmi scritti con questa tecnica, risultano molto più leggibili e limitano i danni dovuti alla propagazione di errori od anomalie all'interno dell'applicazione.
Un'analogia con il mondo reale è rappresentata dalle carte di credito. Chiunque sia dotato di carta di credito può eseguire determinate operazioni bancarie attraverso lo sportello elettronico. Una carta di credito, non mostra all'utente gli automatismi necessari a mettersi in comunicazione con l'ente bancario o quelli necessari ad effettuare transazioni sul conto corrente, semplicemente si limita a farci prelevare la somma richiesta tramite un'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 i dati della carta di credito o addirittura dell'intero sistema.
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 errori di programmazione 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 a quanto aspettato dall'utente dell'oggetto. Se infatti l'utente trasmette all'oggetto parametri errati, l'oggetto si troverà in uno stato consistente, ma non in quello desiderato.

7.2 Modificatori public e private

Java fornisce supporto per l'incapsulamento a livello di linguaggio, tramite i modificatori **public** e **private** da utilizzare al momento della dichiarazione di variabili e metodi. I membri di una classe, o l'intera classe, se definiti **public** sono liberamente accessibili da ogni oggetto componente l'applicazione.
I membri di una classe definiti **private** possono essere utilizzati sono dai membri della stessa classe. I membri privati mascherano i dettagli dell'implementazione di una classe.
Membri di una classe non dichiarati **public** o **private** sono per definizione accessibili solo alle classi appartenenti allo stesso package. Questi membri o classi sono comunemente detti "**package friendly**".



Modificatori public e private

7.3 Il modificatore 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. Di fatto, l'intento è di nascondere porzioni di codice della classe che non devono essere utilizzati da altre classi.

Un membro privato può essere utilizzato da un qualsiasi membro statico, e non, della stessa classe di definizione con l'accorgimento che i membri statici possono accedere solamente ad altri membri statici od oggetti di qualunque tipo purché esplicitamente passati per parametro.
Per dichiarare un membro privato si utilizza la parola chiave ***private*** anteposta alla dichiarazione di un metodo o di un dato:

private *tipo nome;*

Oppure, nel caso di metodi:

private *tipo_di_ritorno nome(tipo identificatore [,tipo identificatore])*
{
    istruzione
    [istruzione]
}

7.4 Il modificatore public

Il modificatore **public** consente di definire classi o membri di una classe visibili a qualsiasi oggetto definito all'interno dello stesso package e non. Questo modificatore 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 :

private *tipo nome;*

Oppure, nel caso di metodi:

private *tipo_di_ritorno nome(tipo identificatore [,tipo identificatore])*
{
    istruzione
    [istruzione]
}

Infine, nel caso di classi:

public class *Nome*
{
    dichirazione_dei_dati
    dichirazione_dei_metodi

```
}
```

7.5 Il modificatore protected

Un altro modificatore messo a disposizione dal linguaggio Java è **protected**. I membri di una classe dichiarati **protected** possono essere utilizzati sia dai membri della stessa classe che da altre classi purché appartenenti allo stesso package.
Per dichiarare un membro protected si utilizza la parola chiave ***protected*** anteposta alla dichiarazione :

protected *tipo nome;*

Oppure, nel caso di metodi:

protected *tipo_di_ritorno nome(tipo identificatore [,tipo identificatore])*
{
    istruzione
    [istruzione]
}

Nonostante possa sembrare un modificatore ridondante rispetto ai precedenti due, tuttavia di questo modificatore torneremo a parlarne nei dettagli nel prossimo capitolo dove affronteremo il problema della ereditarietà.

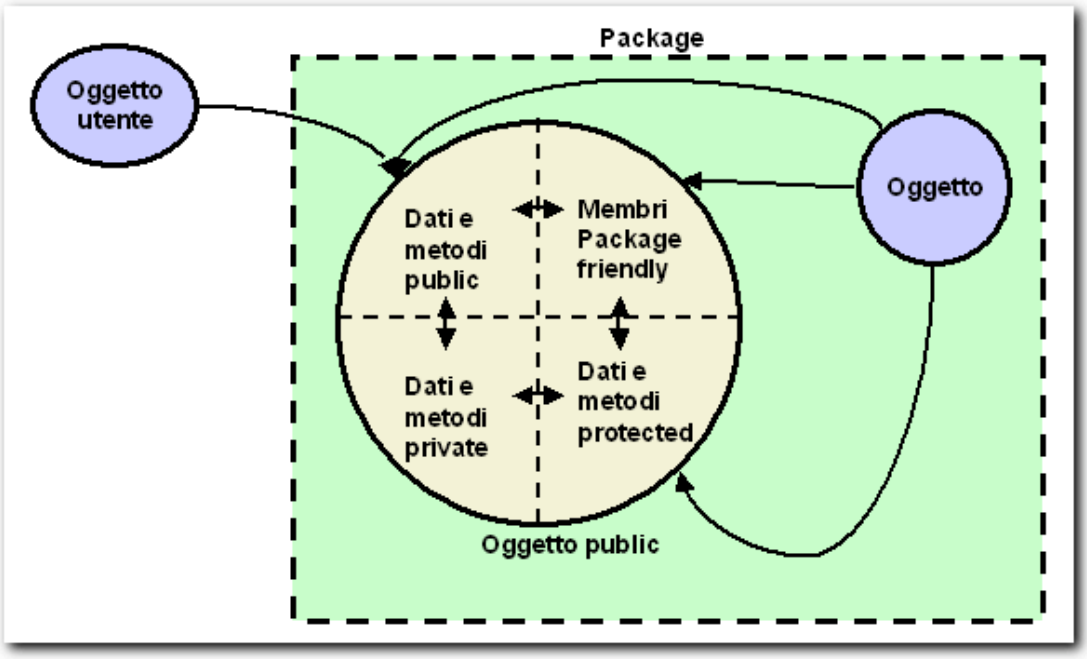


Figura 54: Modificatore protected

7.6 Un esempio di incapsulamento

Le classi che utilizzeremo nel prossimo esempio sono rappresentate nella figura successiva che ne rappresenta il *class-diagram*. La classe *Impiegato* contiene la definizione di due dati membro privati, chiamati *nome* e *affamato,* e di tre metodi pubblici *chiamati haiFame(), nome(), vaiAPranzo()*. La classe *DatoreDiLavoro* contiene l'unica definizione del metodo membro *siiCorrettoConImpiegato()* di tipo public.
Per identificare la visibilità di un metodo od un dato membro, il linguaggio UML antepone al nome, i caratteri '-' e '+' rispettivamente per identificare membri **private** e **public**. Il nome di ogni membro è seguito dal carattere ':' e di seguito il tipo del membro.

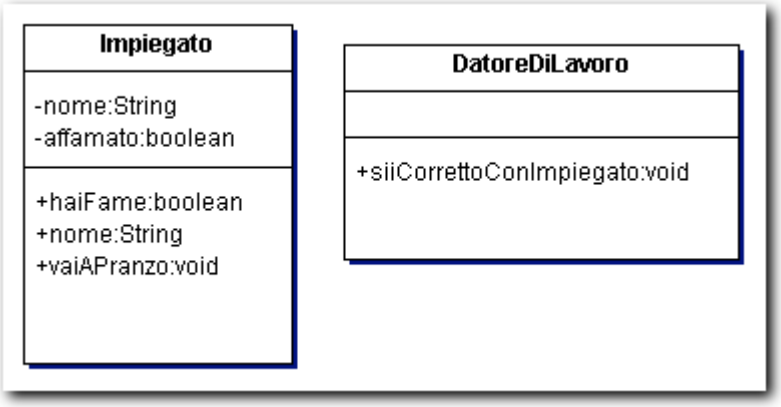


Figura 55: Class Diagram Datore-Impiegato

```
/**
 * Definisce il concetto di Impiegato.
 * Un datore di lavoro non può cambiare il nome di un  ipèiegato ne,
 * può evitare che, se affamato, l'piegato vada a pranzo
 */

package src.esercizi.modificatori;

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

```
        {
                // mangia
                affamato = false;
        }

}
```

```
/**
 * Un datore di lavoro che voglia
 * essere corretto con un suo impiegato,
 * deve mandarlo a pranzo quando quest'ultimo è affamato.
 */

package src.esercizi.modificatori;

public class DatoreDiLavoro
{
        public void siiCorrettoConImpiegato(Impiegato impiegato)
        {
                if (impiegato.haiFame())
                {
                        impiegato.vaiAPranzo("Ristorante sotto l''ufficio");
                }
        }
}
```

Aver definito i dati *nome* e *affamato*, membri privati della classe *Impiegato*, 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* alle attività schematizzate nel prossimo *sequenze-diagram*: un datore di lavoro non può cambiare il nome di un impiegato, ne può convincerlo di non avere fame arrivata l'ora di pranzo.

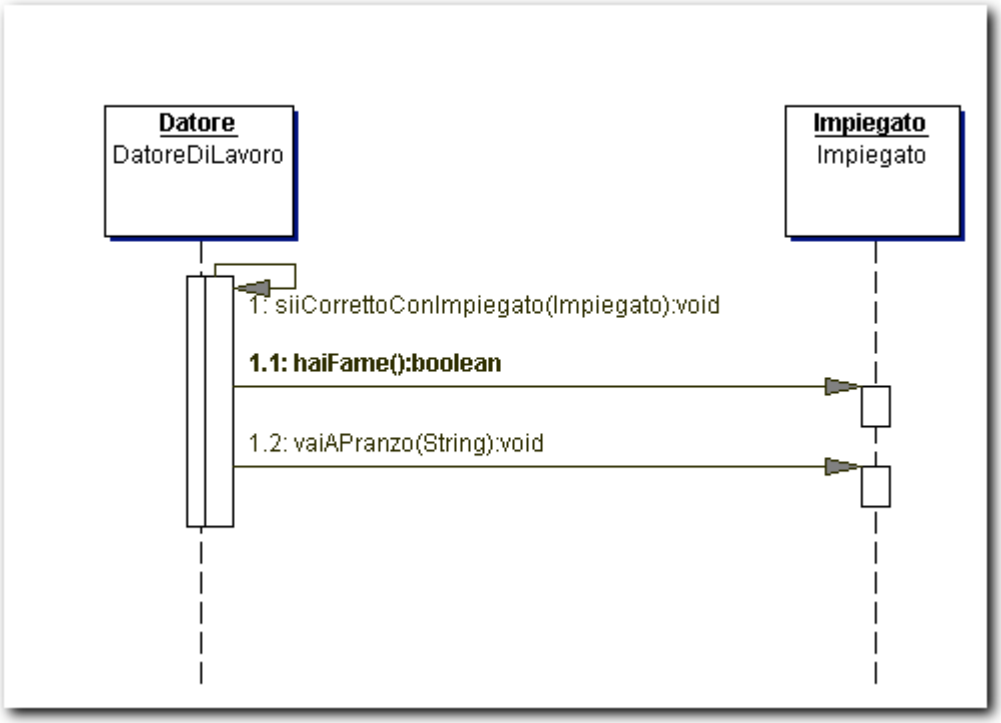


Figura 56: Sequenze Diagram Datore Impiegato

7.7 L'operatore new

Abbiamo già accennato che, per creare un oggetto dalla sua definizione di classe, Java mette a disposizione l'operatore **new** responsabile del caricamento dell'oggetto in memoria e della successiva creazione di un riferimento (indirizzo di memoria), che può essere memorizzato in una variabile di tipo "reference" ed utilizzato per accederne ai membri. Quest'operatore è quindi paragonabile alla **malloc** in C, ed è identico al medesimo operatore in C++.
La responsabilità del rilascio 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.
Quello che non abbiamo detto è che, questo operatore ha la responsabilità di consentire l'assegnamento dello stato iniziale dell'oggetto allocato.
Di fatto, la sintassi dell'operatore **new** prevede un tipo seguito da un insieme di parentesi. Le parentesi indicano che, al momento della creazione in memoria dell'oggetto verrà chiamato un metodo speciale detto **costruttore**, responsabile proprio della inizializzazione del suo stato.
Le azioni compiute da questo operatore, schematizzate nella prossima figura, sono le seguenti:

1. *Richiede alla JVM di caricare la definizione di classe utilizzando le informazioni memorizzate nella variabile d'ambiente CLASSPATH.*

*Terminata quest'operazione, stima la quantità memoria necessaria a contenere l'oggetto e chiede alla JVM di riservarla. La variabile reference **this** non è ancora inizializzata.*

2. *Esegue il metodo costruttore dell'oggetto caricato per consentirne l'inizializzazione dei dati membro. Inizializza la variabile reference **this**.*
3. *Restituisce il riferimento alla locazione di memoria allocata per l'oggetto. Utilizzando l'operatore di assegnamento è possibile memorizzare il valore restituito in una variabile reference dello stesso tipo dell'oggetto caricato.*

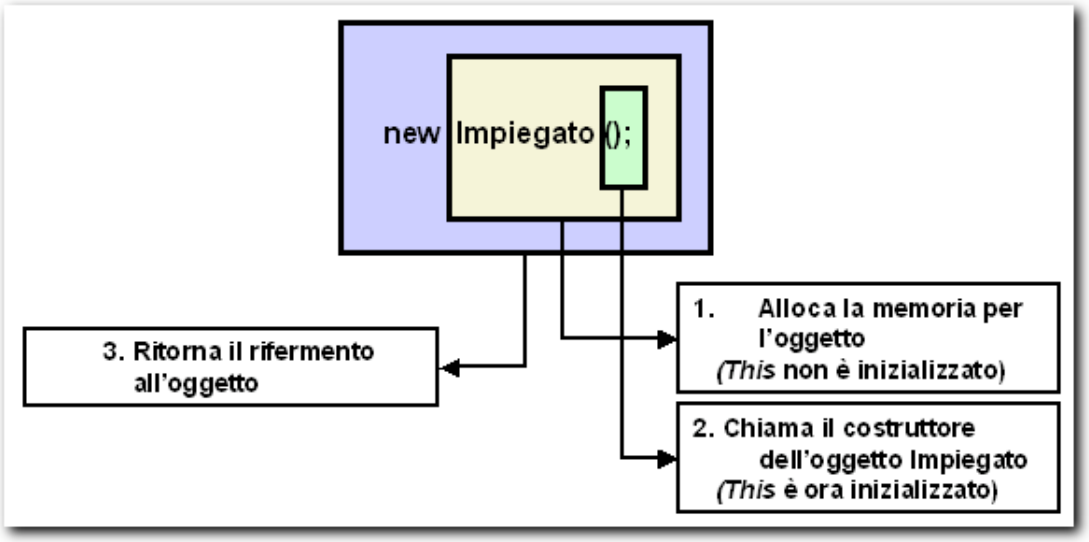


Figura 57: L'operatore new

7.8 Metodi costruttori

Tutti i programmatori, esperti e non, conoscono il pericolo costituito da una variabile non inizializzata. In un'applicazione Object Oriented, un oggetto è un'entità più complessa di un tipo primitivo e l'errata inizializzazione dello stato di un oggetto può essere causa della terminazione prematura dell'applicazione o della generazione di errori intermittenti difficilmente controllabili.
In molti altri linguaggi di programmazione, il responsabile dell'inizializzazione delle variabili è il programmatore. In Java, questo è impossibile poiché potrebbero essere membri privati di un oggetto, e quindi inaccessibili all'utente.
I costruttori sono metodi speciali chiamati in causa dall'operatore **new** al momento della creazione di un nuovo oggetto e servono ad impostarne lo stato iniziale. Questi metodi, hanno lo stesso nome della classe di cui sono membri e non restituiscono nessun tipo (**void** compreso). Sfogliando a ritroso tra gli esempi precedenti, dovrebbe sorgere spontaneo chiedersi come mai non abbiamo mai definito il metodo costruttore delle classi. Di fatto, una

classe non è mai sprovvista di costruttore. Nel caso in cui il costruttore non sia definito dall'utente, uno costruttore speciale che non fa nulla, verrà aggiunto dal compilatore Java al momento della creazione del bytecode.
Dal momento che Java garantisce l'esecuzione del metodo costruttore di un nuovo 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:

```
/**
 * Definisce il concetto di Impiegato.
 * Un datore di lavoro non può cambiare il nome di un  ipèiegato ne,
 * può evitare che, se affamato, l'piegato vada a pranzo
 */

package src.esercizi.modificatori;

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

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

        public boolean haiFame()
        {
                return affamato;
        }

        public String nome()
        {
                return nome;
        }

        public void vaiAPranzo(String luogo)
        {
                afamato = false;
        }
}
```

7.9 Un esempio di costruttori

In quest'esempio, modificheremo la definizione dell'oggetto Pila inserendo il metodo costruttore che, assegna il valore 10 al dato membro privato '*dimensionemassima*' che rappresenta il numero massimo di elementi contenuti nella Pila, inizializza il dato che tiene traccia della cima della pila, ed infine crea l'array che conterrà i dati inseriti.

```
package src.esercizi.costruttori;

/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */

public class Pila {
	private int[] dati;
	private int cima;
	private int dimensionemassima;

	/**
	* Il metodo costruttore della classe Pila, non ritorna nessun tipo
	* di dato, imposta la dimensione massima dell'array che
	* conterrà i dati della pila e crea l'array.
	*/

	public Pila() {
		dimensionemassima = 10;
		dati = new int[dimensionemassima];
		cima=0;
	}

	/**
	* Il metodo push ritorna un tipo void e prende come
	* parametro un numero intero da inserire sulla cima della pila
	* @return void
	* @param int dato : elemento da inserire sulla cima della pila
	*/

	public void push(int dato)
	{
		if(cima < dimensionemassima)
		{
			dati[cima] = dato;
			cima ++;
		}
	}

	/**
	* il metodo pop non accetta parametri e restituisce
	* l'elemento sulla cima della Pila
	* @return int : dato sulla cima della pila
	*/

	public int pop()
	{
		if(cima > 0)
```

```
                {
                        cima--;
                        return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

Rispetto alla prima definizione della classe Pila, non è più necessario creare l'array di interi al momento della prima 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.

7.10 Overloading dei costruttori

Java supporta molte caratteristiche per i costruttori, ed esistono molte regole per la loro creazione. In particolare, al programmatore è consentito scrivere più di un costruttore per una data classe, secondo le necessità di disegno dell'oggetto. Questa caratteristica permette di passare all'oggetto diversi insiemi di dati di inizializzazione, consentendo di adattarne lo stato ad una particolare situazione operativa.
Nell'esempio precedente, abbiamo ridefinito l'oggetto *Pila* affinché contenga un massimo di dieci elementi. Un modo per generalizzare l'oggetto è definire un costruttore che, prendendo come parametro un intero, inizializza la dimensione massima della *Pila* secondo le necessità dell'applicazione. La nuova definizione della classe potrebbe essere la seguente:

```
package src.esercizi.costruttori;

/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */

public class Pila {
        private int[] dati;
        private int cima;
        private int dimensionemassima;

        /**
        * Il metodo costruttore della classe Pila, non ritorna nessun tipo
        * di dato, imposta la dimensione massima dell'array che
```

```
        * conterrà i dati della pila e crea l'array.
        */

        public Pila() {
                dimensionemassima = 10;
                dati = new int[dimensionemassima];
                cima=0;
        }

        /**
        * Questo metodo costruttore della classe Pila,
        * consente di impostare la dimensione massima della Pila
        * accettando un parametro di tipo int
        */

        public Pila(int dimensionemassima) {
                this.dimensionemassima = dimensionemassima;
                dati = new int[dimensionemassima];
                cima=0;
        }

        /**
        * Il metodo push ritorna un tipo void e prende come
        * parametro un numero intero da inserire sulla cima della pila
        * @return void
        * @param int dato : elemento da inserire sulla cima della pila
        */

        public void push(int dato)
        {
                if(cima < dimensionemassima)
                {
                        dati[cima] = dato;
                        cima ++;
                }
        }

        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return int : dato sulla cima della pila
        */

        public int pop()
        {
                if(cima > 0)
                {
                        cima--;
                        return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

L'utilizzo dei due costruttori, ci consente di creare oggetti Pila di dimensioni fisse utilizzando il primo costruttore, o variabili utilizzando il costruttore che prende come parametro di input un tipo **int**.

```
/**
* Questo metodo costruttore della classe Pila,
* consente di impostare la dimensione massima della Pila
* accettando un parametro di tipo int
*/

 public Pila(int dimensionemassima) {
          this.dimensionemassima = dimensionemassima;
          dati = new int[dimensionemassima];
          cima=0;
}
```

Per creare una istanza della classe Pila invocando il nuovo costruttore, basterà utilizzare l'operatore **new** come segue:

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

In definitiva, la sintassi completa dell'operatore **new** è la seguente:

new *Nome([lista_parametri]) ;*

7.11 Restrizione sulla chiamata ai costruttori

Java consente una sola chiamata al costruttore di una classe. Di fatto, un metodo costruttore può essere invocato solo dall'operatore **new** al momento della creazione di un oggetto. Nessun metodo costruttore può essere eseguito nuovamente dopo la creazione dell'oggetto. Il frammento seguente di codice Java, produrrà un errore di compilazione.

```
int dimensioni=10;
Pila s = new Pila(dimensioni);
//Questa chiamata è illegale
Pila.Stack(20);
```

7.12 Chiamate incrociate tra costruttori

Un metodo costruttore ha la possibilità di effettuare chiamate ad altri costruttori appartenenti alla stessa definizione di classe. Questo meccanismo è utile perché i costruttori generalmente hanno funzionalità simili e un

costruttore che assegna all'oggetto uno stato comune, potrebbe essere richiamato da un altro per sfruttare il codice definito nel primo.
Per chiamare un costruttore da un altro, è necessario utilizzare la sintassi speciale:

***this**(lista_dei_parametri);*

dove, *lista_dei_parametri* rappresenta la lista di parametri del costruttore che si intende chiamare.
Una chiamata incrociata tra costruttori, deve essere la prima riga di codice del costruttore chiamante. Qualsiasi altra cosa sia fatta prima, compresa la definizione di variabili, non consente di effettuare tale chiamata.
Il costruttore corretto è determinato in base alla lista dei parametri. Java paragona *lista_dei_parametri* con la lista dei parametri di tutti i costruttori della classe.
Tornando alla definizione di Pila, notiamo che i due costruttori eseguono operazioni simili. Per ridurre la quantità di codice, possiamo chiamare un costruttore da un altro come mostraton el prossimo esempio.

```
package src.esercizi.costruttori;

/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */

public class Pila {
	private int[] dati;
	private int cima;
	private int dimensionemassima;

	/**
	* Il metodo costruttore della classe Pila, non ritorna nessun tipo
	* di dato, imposta la dimensione massima dell'array che
	* conterrà i dati della pila e crea l'array.
	*/

	public Pila() {
		this(10);
	}

	/**
	* Questo metodo costruttore della classe Pila,
	* consente di impostare la dimensione massima della Pila
	* accettando un parametro di tipo int
	*/

	public Pila(int dimensionemassima) {
		this.dimensionemassima = dimensionemassima;
		dati = new int[dimensionemassima];
		cima=0;
	}
```

```
        /**
        * Il metodo push ritorna un tipo void e prende come
        * parametro un numero intero da inserire sulla cima della pila
        * @return void
        * @param int dato : elemento da inserire sulla cima della pila
        */

        public void push(int dato)
        {
                if(cima < dimensionemassima)
                {
                        dati[cima] = dato;
                        cima ++;
                }
        }

        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return int : dato sulla cima della pila
        */

        public int pop()
        {
                if(cima > 0)
                {
                        cima--;
                        return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

7.13 Un nuovo esempio

Nel prossimo esempio, definiremo un oggetto di tipo Insiema che, rappresenta un insieme di numeri interi assicurandoci di incapsulare l'oggetto utilizzando i modificatori **public** o **private** appropriati.

L'insieme deve avere al massimo tre metodi:

1. **boolean** membro(**int valore**): ritorna *true* se il numero è nell'insieme;
2. **void** aggiungi (**int valore**) : aggiunge un numero all'insieme ;
3. **void** stampa(): stampa a video il contenuto dell'insieme nel formato {elemento, elemento, ... , elemento}

Di seguito, una possibile soluzione all'esercizio proposto.

```
package src.esercizi.costruttori;

/**
 * Questa classe e' la definizione
 * dell'oggetto che rappresenta
```

```
 * un insieme di numeri interi.
 */

public class Insieme {
	public Insieme() {
		setDimensioneCorrente(0);
		elementi = new int[100];
	}

	private int getDimensioneCorrente(){
		return dimensioneCorrente;
	}

	private void setDimensioneCorrente(int dimensioneCorrente){
		this.dimensioneCorrente = dimensioneCorrente;
	}

	/**
	* Verica l'appartenenza del numero intero passato come
	* parametro all'insieme corrente, tornando true se elemento
	* appartiene all'insieme.
	* @param int elemento : elemento di cui verificare l'appartenenza all'insieme.
	* @return boolean : true se elemento appartiene all'insieme, false altrimenti
	*/

	public boolean membro(int elemento) {
		int indice = 0;
		while(indice < getDimensioneCorrente())
		{
			 if(elementi[indice]==elemento) return true;
			indice++;
		}
		return false;
	}




	/**
	* Produce la stampa dell'insieme su terminale
	* nella foema {elemento, elemento, ...., elemento}
	* @return void
	*/

	public void stampa() {
		int indice = 0;
		System.out.print("{");
		while(indice < getDimensioneCorrente())
		{
			System.out.println(elementi[indice] +" , ");
			indice++;
		}
		System.out.print("}");
	}

	 /**
	* Aggiunge un elemento all'insieme corrente.
	* L'elemento non deve essere contenuto
	* nell'insieme corrente.
```

```
	* @return void
	* @param int elemento : elemento da ggiungere all'insieme corrente.
	                        L'elemento viene inserito solo se non risulta
	                        già contenuto all'interno dell'insieme.
	*/

	public void aggiungi(int elemento) {
		//Verifica l'appartenenza dell'elemento all'insieme.
		//Se l'elemento appartiene all'insieme, il metodo
		//ritorna senza eseguire alcuna operazione.
		if (!membro(elemento))
		{
			//Se l'array è pieno il metodo ritorna senza
			//eseguire alcuna operazione.

			if(getDimensioneCorrente() == elementi.length) return;

			//Aggiorno l'elemento all'insieme

			elementi[getDimensioneCorrente()] = elemento;

			//Aggiorno il valore di DimensioneCorrente

			setDimensioneCorrente(getDimensioneCorrente()+1);
		}
	}

	/**
	* Array contenente la lista degli
	* elementi inseriti all'interno dell'insieme.
	*/

	private int[] elementi;

	/**
	* Proprietà che rappresenta le
	* dimensioni correnti dell'insieme
	*/
	private int dimensioneCorrente;
}
```

Utilizzando il modificatore **private**, dichiariamo i due dati membro della nostra classe rendendoli visibili soltanto ai metodi membri della classe.
Il primo, ***int*** *elemento[]*, rappresenta un array di interi che conterrà i dati memorizzati all'interno dell'insieme. Il secondo, **int** *dimensioneCorrente*, rappresenta la dimensione o cardinalità attuale dell'insieme.
Focalizziamo per qualche istante l'attenzione del dato membro *dimensioneCorrente*. All'interno della definizione della classe Insieme, sono definiti i due metodi

```
private int getDimensioneCorrente(){
	return dimensioneCorrente;
}
```

```
private void setDimensioneCorrente(int dimensioneCorrente){
        this.dimensioneCorrente = dimensioneCorrente;
}
```

detti rispettivamente metodi "getter" e "setter". Il primo, è responsabile della restituzione del valore rappresentato dal dato membro *dimensioneCorrente;* il secondo è invece responsabile dell'aggiornamento del valore del dato.
In generale, un dato affiancato da due metodi "getter" e "setter" è detto proprietà di una classe Java. Le regole per definire la proprietà di una classe sono le seguenti:

1. *La proprietà è un qualsiasi dato che rappresenta una classe od un tipo semplice. In genere, tali membri sono dichiarati come privati.*
2. *Il nome del metodo "getter" è determinato dal nome del dato anteposto dalla stringa "get";*
3. *Il nome del metodo "setter" dal nome del dato anteposto dalla stringa "set".*
4. *I metodi "getter" e "setter" possono essere dichiarati pubblici o privati.*

In un *class-diagram* UML, la proprietà di una classe è rappresentata dal nome della variabile. I metodi "getter" e "setter" non compaiono all'interno del simbolo che rappresenta la classe.

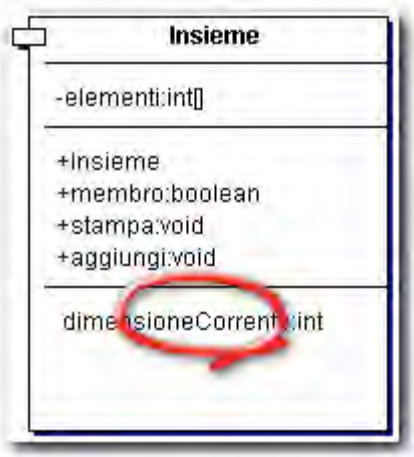


Figura 58: Class Diagram della classe Insieme

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'array. Se, infatti, ciò fosse possibile, potremmo scrivere il metodo main dell'applicazione affinché inserisca direttamente gli elementi dell'insieme all'interno dell'array come da esempio seguente:

```
public class TestInsieme
```

```
{
        public static void main(String args[])
        {
                Insieme insieme = new Insieme ();
                //la prossima chiamata è illegale
                Insieme.elementi[1]=0;
        }

}
```

In questo caso, non sarebbe aggiornata la cardinalità dell'insieme rendendo inconsistente lo stato dell'oggetto, o ancora peggio, potremmo sovrascrivere elementi dell'insieme già esistenti.
I tre metodi dichiarati pubblici, rappresentano l'interfaccia che il programmatore ha a disposizione per utilizzare in modo corretto l'insieme di interi.

8 EREDITARIETÀ

8.1 Introduzione

L'ereditarietà è la caratteristica dei linguaggi Object Oriented che consente di utilizzare definizioni di classe come base per la definizione di nuove che ne specializzano il concetto. L'ereditarietà inoltre, offre un ottimo meccanismo per aggiungere funzionalità ad un programma con rischi minimi nei confronti di quelle già 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 in supporto dal linguaggio. Questo capitolo introduce al concetto di ereditarietà in Java, alla sintassi per estendere classi, all'*overloading* e *overriding* di metodi. Infine, in questo capitolo introdurremo ad una particolarità rilevante del linguaggio Java che, include sempre la classe Object nella gerarchia delle classi definite dal programmatore.

8.2 Disegnare una classe base

Disegnando 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".
Facendo uso del meccanismo della ereditarietà, è necessario tener sempre 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 riciclaggio 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 alla classe derivata. La classe base sarà quindi protetta dalla generazione di nuovi eventuali errori, 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 detto di "ereditarietà singola". A differenza da linguaggi come il C++ in cui una classe derivata può ereditare da molte classi base (ereditarietà multipla), Java consente di poter ereditare da una sola classe base come mostrato. Nelle due prossime figure sono illustrati rispettivamente i due modelli di ereditarietà multipla e singola.

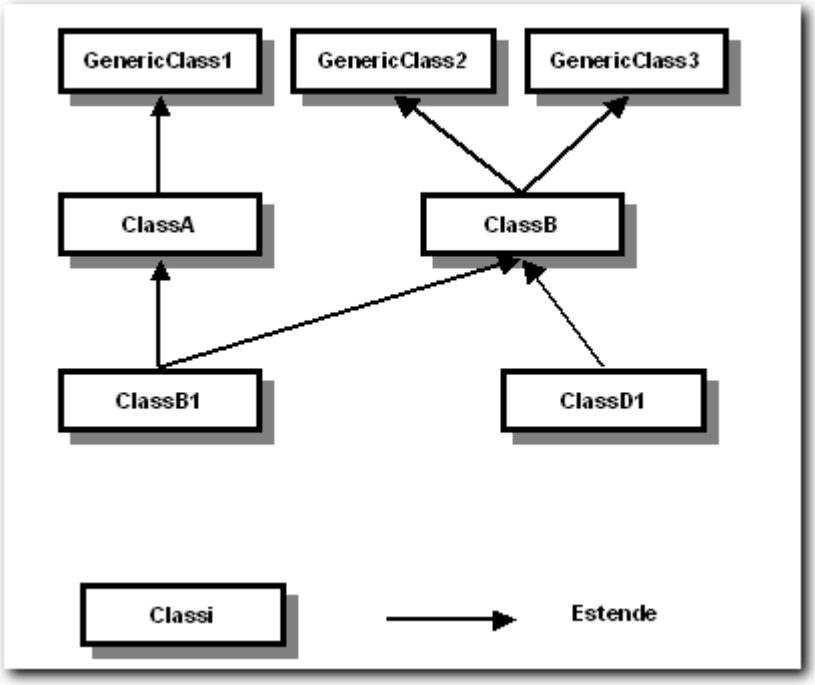


Figura 59: *Modello* ***ad ereditarietà multipla***

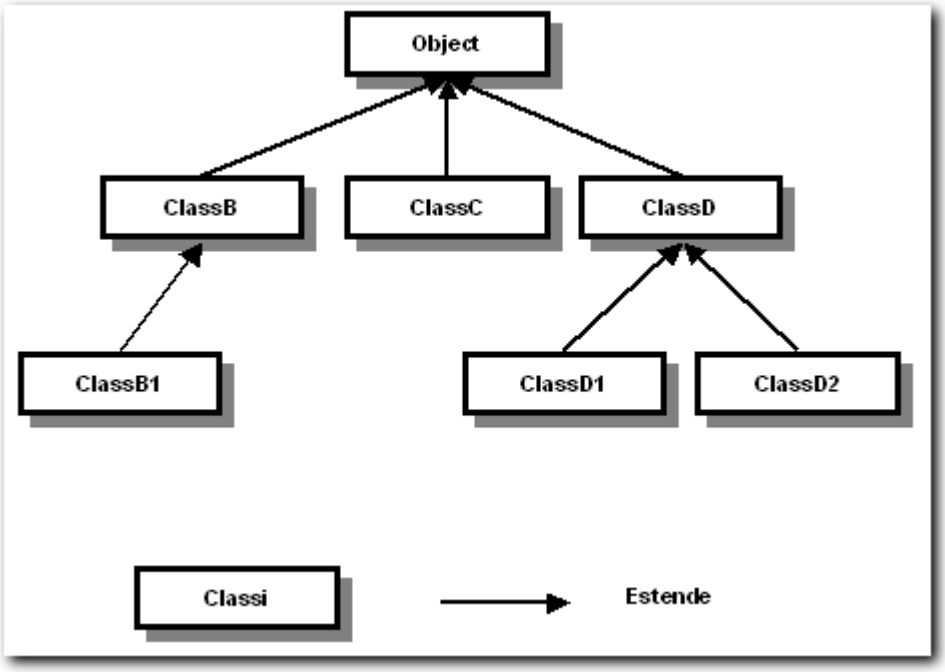


Figura 60: Modello ad ereditarietà singola in Java

Proviamo a disegnare una classe base e per farlo, pensiamo ad un veicolo generico: questo potrà muoversi, svoltare a sinistra o a destra o fermarsi. Di seguito sono riportate le definizioni della classe base *Veicolo*, e della classe contenete il metodo main dell'applicazione *Autista*. Nelle due prossime figure sono schematizzati rispettivamente il class-diagram della applicazione ed il relativo sequenze-diagram.

Figura 61: Class-Diagram di autista

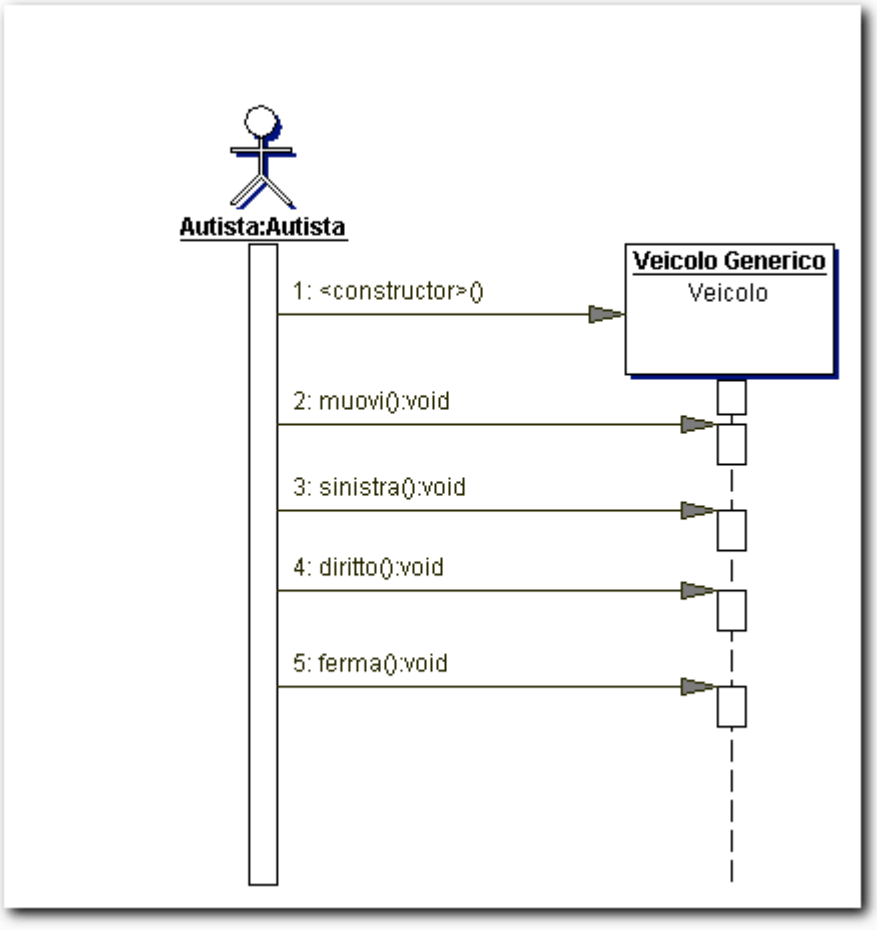


Figura 62: Sequenze-Diagram di autista

```
package src.esercizi.ereditarieta;

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

```
package src.esercizi.ereditarieta;

public class Veicolo {
	/**
	* Nome del veicolo
	*/
	String nome;
```

```
/**
* velocità  del veicolo espressa in Km/h
*/
int velocita;

/**
* Direzione di marcia del veicolo
*/
int direzione;

/**
* Costante che indica la direzione "dritta" di marcia del veicolo
*/
final static int DRITTO = 0;

/**
* Costante che indica la svola a sinistra del veicolo
*/
final static int SINISTRA = -1;

/**
* Costante che indica la svola a destra del veicolo
*/
final static int DESTRA = 1;

public Veicolo() {
        velocita = 0;
        direzione = DRITTO;
        nome = "Veicolo generico";
}

/**
* Simula la messa in marcia del veicolo alla velocità di 1 km/h
*/
public void muovi() {
        velocita = 1;
        System.out.println(nome + " si sta movendo a: " + velocita + " Kmh");
}

/**
* Simula la frenata del veicolo
*/
public void ferma() {
velocita = 0;
System.out.println(nome + " si è fermato");
}

/**
* Simula la svola a sinistra del veicolo
*/
public void sinistra() {
        direzione = SINISTRA;
        System.out.println(nome + " ha sterzato a sinistra");
}

/**
* Simula la svola a destra del veicolo
*/
public void destra() {
```

```
                direzione = DESTRA;
                System.out.println(nome + " ha sterzato a destra");
        }

        /**
        * Simula la conclusione di una svola a sinistra o a destra del veicolo
        */
        public void diritto() {
                direzione = DRITTO;
                System.out.println(nome + " sta procedendo in linea retta");
        }
}
```

8.3 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.
Esaminiamo il metodo *muovi()* della classe *Veicolo* definito 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:

```
package src.esercizi.ereditarieta;

public class Veicolo {
        /**
        * Nome del veicolo
        */
        String nome;

        /**
        * velocità  del veicolo espressa in Km/h
        */
        int velocita;

        /**
        * Direzione di marcia del veicolo
        */
        int direzione;

        /**
        * Costante che indica la direzione "dritta" di marcia del veicolo
        */
        final static int DRITTO = 0;

        /**
        * Costante che indica la svola a sinistra del veicolo
        */
        final static int SINISTRA = -1;

        /**
        * Costante che indica la svola a destra del veicolo
        */
        final static int DESTRA = 1;
```

```
	public Veicolo() {
		velocita = 0;
		direzione = DRITTO;
		nome = "Veicolo generico";
	}

	/**
	* Simula la messa in marcia del veicolo alla velocità di 1 km/h
	*/
	public void muovi() {
		velocita = 1;
		System.out.println(nome + " si sta movendo a: " + velocita + " Kmh");
	}

	/** Simula la messa in marcia del veicolo specificando la velocità in km/h
	* @return void
	* @param int velocita : velocita del veicolo
	*/

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

	/**
	* Simula la frenata del veicolo
	*/
	public void ferma() {
	velocita = 0;
	System.out.println(nome + " si è fermato");
	}

	/**
	* Simula la svola a sinistra del veicolo
	*/
	public void sinistra() {
		direzione = SINISTRA;
		System.out.println(nome + " ha sterzato a sinistra");
	}

	/**
	* Simula la svola a destra del veicolo
	*/
	public void destra() {
		direzione = DESTRA;
		System.out.println(nome + " ha sterzato a destra");
	}

	/**
	* Simula la conclusione di una svola a sinistra o a destra del veicolo
	*/
	public void diritto() {
		direzione = DRITTO;
		System.out.println(nome + " sta procedendo in linea retta");
	}
}
```

Avendo a disposizione anche il metodo *muovi(**int** velocita)*, possiamo migliorare la nostra simulazione facendo in modo che il veicolo possa accelerare o decelerare ad una determinata velocità.
L'overloading di metodi è possibile poiché, il nome di un metodo, non definisce in maniera univoca un membro di una classe. Ciò che consente di determinare in maniera univoca quale sia il metodo correntemente chiamato di una classe Java è la sua firma (signature). In dettaglio, si definisce firma di un metodo, il suo nome assieme alla lista dei suoi parametri.
Per concludere, ecco alcune linee guida per utilizzare correttamente l'overloading di metodi. Primo, come conseguenza diretta della definizione precedente, 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.

8.4 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(lista_parametri) **extends** nome_super_classe*

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

```
package src.esercizi.ereditarieta;

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

}
```

```
package src.esercizi.ereditarieta;

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 specializzato per la nuova classe. Il nuovo costruttore semplicemente modifica il contenuto della variabile nome affinché l'applicazione stampi i messaggi corretti.
Come mostrato nel codice della nuova definizione di classe per *Autista*, utilizzare il nuovo veicolo equivale ad utilizzare il *Veicolo* generico definito nei paragrafi precedenti.

8.5 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 loro da relazioni di ereditarietà. Nella tabella seguente sono schematizzati i livelli di visibilità dei tre modificatori.

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

Le direttive che regolano il rapporto tra i tre modificatori e l'ereditarietà sono le seguenti: il modificatore **public** consente di dichiarare dati e metodi membro visibili e quindi utilizzabili da un'eventuale sottoclasse; il modificatore **private** nasconde completamente dati e metodi membro dichiarati tali. E' invece necessario soffermarci sul modificatore **protected** al quale è dedicato il prossimo esempio.
Creiamo il package *src.esercizi.modificatori.protect* la cui struttura è schematizzata nel prossimo *class-diagram*.

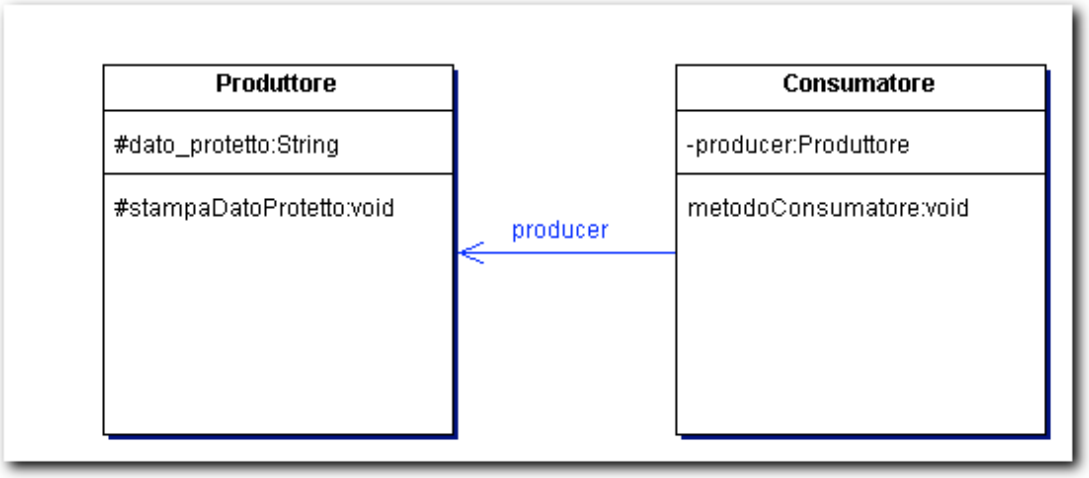


Figura 63: Modificatore protected

Il package contiene le due seguenti definizioni di classe:

```
package src.esercizi.modificatori.protect;

public class Produttore {

	protected String dato_protetto ="Questo dato e di tipo protected";

	protected void stampaDatoProtetto()
	{
		System.out.println(dato_protetto);
	}
}
```

```
package src.esercizi.modificatori.protect;

public class Consumatore {
	/**
	* @label base
	*/
	private Produttore base = new ClasseBase();
	void metodoConsumatore()
	{
		base.stampaDatoProtetto();
	}

}
```

La definizione della classe *Produttore* contiene due membri di tipo **protected**, identificati nel *class-diagram* dal nome dei membri anteposti dal carattere '#'. Il primo, è un attributo di tipo *String* chiamato *dato_protetto*, il secondo un metodo chiamato *stampaDatoProtetto* che, torna un tipo **void** e produce la stampa a terminale del valore del primo membro.

La classe *Consumatore*, ha un riferimento alla classe precedente, tramite il dato membro producer e ne effettua la chiamata al metodo *stampaDatoProtetto*.
Poiché le due classi appartengono allo stesso package, alla classe *Consumatore* è consentito l'utilizzo del metodo **protected** *stampaDatoProtetto()* di *Produttore* senza che il compilatore Java segnali errori. Analogamente, i membri **protected** della classe *Produttore* saranno visibili alla classe *NuovoProduttore*, definita per mezzo della ereditarietà all'interno dello stesso package, come schematizzato nella prossima figura.

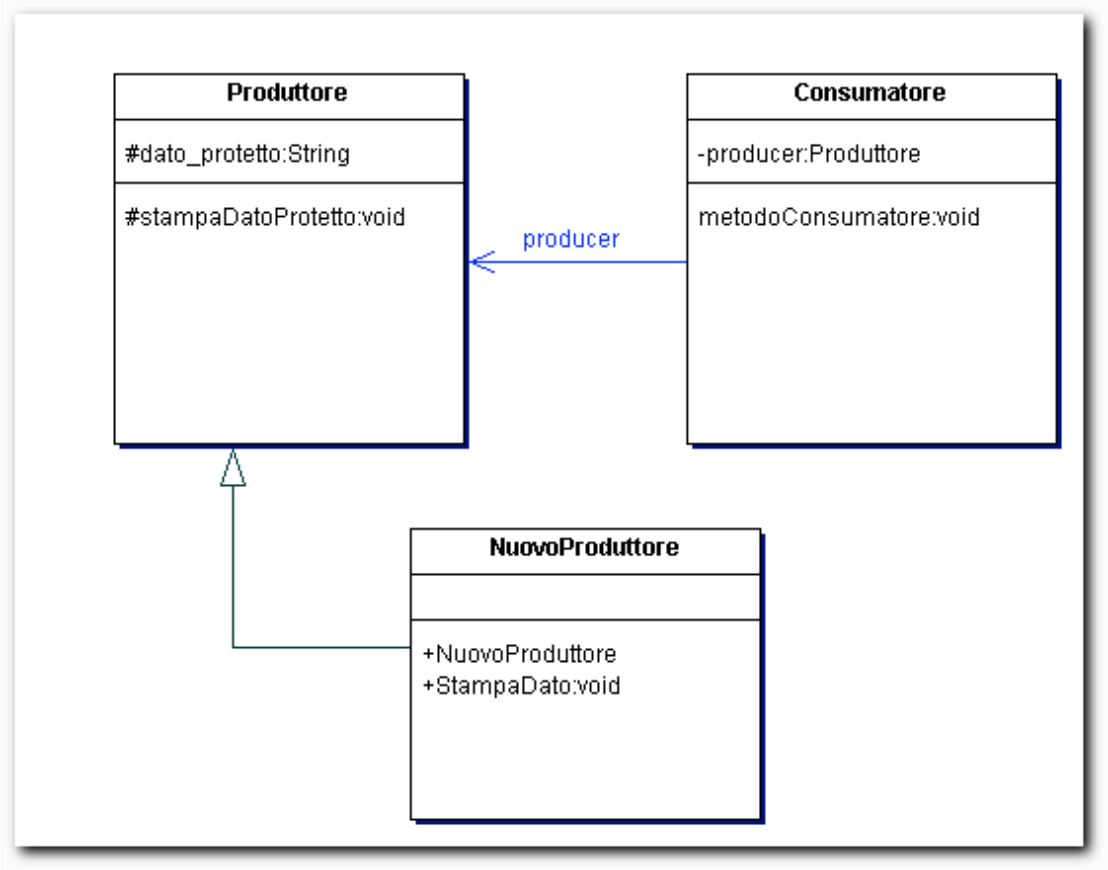


Figura 64: Modificatore protected ed ereditarietà

```
package src.esercizi.modificatori.protect;

public class NuovoProduttore extends Produttore {
	public NuovoProduttore() {
		dato_protetto = "Il valore di dato_protetto viene modificato";

	}

	public void StampaDato()
	{
		stampaDatoProtetto();
	}
}
```

Consideriamo ora la classe *TerzoProduttore*, definita per ereditarietà a partire dalla classe base *ClasseBase*, ma appartenente ad un sotto package di quello corrente:

```
package src.esercizi.modificatori.protect.sub;

import src.esercizi.modificatori.protect.Produttore;

public class TerzoProduttore extends Produttore {
        public TerzoProduttore() {
                dato_protetto = "Il valore di dato_protetto viene modificato";

        }

        public void StampaDato(Produttore producer)
        {
                stampaDatoProtetto(); //Questa chiamata è legale
                producer.stampaDatoProtetto(); //Questa chiamata è illegale

        }
}
```

In questo caso, la nuova classe potrà utilizzare il metodo *stampaDatoProtetto()* ereditato dalla superclasse, anche se non appartenente al medesimo package, ma non potrà utilizzare lo stesso metodo se chiamato direttamente come metodo membro dell'oggetto di tipo *Produttore* referenziato dalla variabile *producer* passata come parametro al metodo *stampaDato*. Se, infatti, provassimo a compilare la classe precedente, il compilatore produrrebbe il seguente messaggio di errore:

```
src\esercizi\modificatori\protect\sub\TerzoProduttore.java:15: stampaDatoProtetto() has protected access in src.esercizi.modificatori.protect.Produttore
        producer.stampaDatoProtetto();
                 ^
1 error
*** Compiler reported errors
```

Di fatto, il modificatore **protected** consente l'accesso ad un metodo o dato membro di una classe:

1. *a tutte le sue sottoclassi, definite o no all'interno dello stesso package;*
2. *alle sole classi appartenenti allo stesso package, se riferite tramite una variabile reference.*

In tutti gli altri casi non sarà possibile utilizzare metodi e dati membro definiti **protected**.

8.6 Conseguenze dell'incapsulamento nella ereditarietà

Negli esempi precedenti si nota facilmente che, la definizione del metodo costruttore della classe *Veicolo* è molto simile a quella 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 succederebbe se sbagliassimo qualche cosa 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:

```
package src.esercizi.incapsulamento;

/**
 * Definizione della figura geometrica cerchio
 */
public class Cerchio {

	public Cerchio() {
		PiGreco = 3.14;
	}

	public double getRaggio() { return raggio; }

	public void setRaggio(double raggio) { this.raggio = raggio; }

	/**
	* Calcola la circonferenza del cerchio il
	* cui raggio è rappresentato dalla proprietà "raggio"
	* @return double : circonferenza del cerchio
	*/
	public double circonferenza() {
		return (2*PiGreco)*getRaggio();
	}

	/**
	* Calcola ll'area del cerchio il
	* cui raggio è rappresentato dalla proprietà "raggio"
	* @return double : area del cerchio
	*/
	public double area() {
		return (getRaggio()*getRaggio())*PiGreco;
	}

	private double PiGreco;
	private double raggio;
}
```

La classe base Cerchio, contiene la definizione di un dato membro privato di tipo **double** che rappresenta la costante matematica Π e mette a disposizione due metodi che consentono di calcolare rispettivamente la lunghezza della circonferenza e l'area del cerchio il cui raggio può essere impostato utilizzando il metodo "setter" della proprietà *raggio*. Il costruttore della classe, ha la responsabilità di inizializzare al valore corretto il dato membro PiGreco.
Definiamo ora la classe *Ruota* per mezzo del meccanismo di ereditarietà a partire dalla classe *Cerchio*.

```
package src.esercizi.incapsulamento;

/**
 * Una ruota può essere definita
 * a partire da un Cerchio generico
 */
public class Ruota extends Cerchio {
        public Ruota(raggio) {
                setRaggio(raggio);
                //La prossima riga produce un errore di compilazione
                PiGreco = 3.14;
        }
}
```

Il metodo costruttore della nuova classe, oltre ad impostare il raggio della ruota utilizzando il metodo *setRaggio* di *Cerchio* avendolo ereditato dalla classe base, tenta di impostare il valore del dato privato *PiGreco* causando un errore durante la compilazione. E' quindi necessario eliminare il problema. La classe *Ruota* corretta avrà la forma seguente:

```
package src.esercizi.incapsulamento;

/**
 * Una ruota può essere definita
 * a partire da un Cerchio generico
 */
public class Ruota extends Cerchio {
        public Ruota(raggio) {
                setRaggio(raggio);
        }
}
```

La nuova definizione di classe verrà compilata correttamente, ma i risultati forniti dalla esecuzione dei metodi *circonferenza* e *area* potrebbero essere valori inattendibili a causa della mancata inizializzazione del dato privato *PiGreco* della classe *Cerchio*.
In altre parole, Java applica l'incapsulamento anche a livello di ereditarietà. In questo modo la classe base gode di tutti i benefici derivanti da un uso corretto dell'incapsulamento; d'altra parte, poiché l'interfaccia di *Cerchio* non

prevede metodi pubblici in grado di modificare il valore di *PiGreco*, l'unico modo per assegnarle il valore corretto è utilizzare il costruttore della classe base. Java deve quindi poter garantire chiamate a costruttori risalendo nella catena delle classi di una gerarchia.

8.7 Ereditarietà e costruttori

Il meccanismo utilizzato da Java per assicurare la chiamata di un costruttore per ogni classe di una gerarchia, si basa su alcuni principi di base.
Primo, ogni classe ***deve*** avere un costruttore. Se il programmatore non ne implementa alcuno, Java per definizione, assegnerà alla classe un costruttore vuoto e senza lista di parametri che chiameremo per comodità costruttore nullo. Il costruttore nullo viene attribuito automaticamente dal compilatore; la classe

```
public class ClasseVuota {

}
```

equivale alla classe:

```
public class ClasseVuota {

        public ClasseVuota(){
        }
}
```

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

super*(lista_degli_argomenti);*

dove lista_degli_*argomenti* rappresenta 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.
Per comprendere meglio il meccanismo, modifichiamo l'esempio visto nei paragrafi precedenti aggiungendo alla classe *Cerchio* un metodo costruttore che ci consente di impostare il valore del raggio della circonferenza al momento della creazione dell'oggetto:

```
package src.esercizi.incapsulamento;

/**
 * Definizione della figura geometrica cerchio
 */
public class Cerchio {
```

```
	public Cerchio() {
		PiGreco = 3.14;
	}

	public Cerchio(double raggio) {
		PiGreco = 3.14;
		setRaggio(raggio);
	}

	public double getRaggio() { return raggio; }

	public void setRaggio(double raggio) { this.raggio = raggio; }

	/**
	* Calcola la circonferenza del cerchio il
	* cui raggio è rappresentato dalla proprietà "raggio"
	* @return double : circonferenza del cerchio
	*/
	public double circonferenza() {
		return (2*PiGreco)*getRaggio();
	}

	/**
	* Calcola l'area del cerchio il
	* cui raggio è rappresentato dalla proprietà "raggio"
	* @return double : area del cerchio
	*/
	public double area() {
		return (getRaggio()*getRaggio())*PiGreco;
	}

	private double PiGreco;
	private double raggio;
}
```

Inseriamo all'interno del costruttore della classe *Cerchio*, la chiamata esplicita al costruttore senza argomenti della classe base:

```
package src.esercizi.incapsulamento;

/**
 * Una ruota può essere definita
 * a partire da un Cerchio generico
 */
public class Ruota extends Cerchio {
	public Ruota() {
		super();
		setRaggio(20);
	}
}
```

Ora siamo sicuri che lo stato della classe base è inizializzato in modo corretto, compreso il valore dell'attributo *PiGreco*, come mostrato dalla applicazione *Circonferenze*.

```
package src.esercizi.incapsulamento;

public class Circonferenze {
	public static void main(String[] argv) {
		Cerchio c = new Cerchio();
		c.setRaggio(20);
		System.out.println("L'area di un cerchio di raggio 20 è: "+c.area());
		System.out.println("La circonferenza di un cerchio di raggio 20 è:
			"+c.circonferenza());

		Ruota r = new Ruota();
		System.out.println("L'area di una ruota di raggio 20 è: "+r.area());
		System.out.println("La circonferenza di una ruota di raggio 20 è:
			"+r.circonferenza());

   }
}
```

```
java src.esercizi.incapsulamento.Circonferenze

L'area di un cerchio di raggio 20 è: 1256.0
La circonferenza di un cerchio di raggio 20 è: 125.60000000000001
L'area di una ruota di raggio 20 è: 1256.0
La circonferenza di una ruota di raggio 20 è: 125.60000000000001
```

In generale, la chiamata esplicita al costruttore senza argomenti è soggetta alle seguente restrizione:

Se, la classe base non contiene la definizione del costruttore senza argomenti ma, contiene la definizione di costruttori specializzati con liste di argomenti, la chiamata esplicita ***super****() provoca errori al momento della compilazione.*

Torniamo ora alla prima versione dell'oggetto *Ruota*, ed eseguiamo nuovamente l'applicazione *Circonferenze*.

```
package src.esercizi.incapsulamento;

/**
 * Una ruota può essere definita
 * a partire da un Cerchio generico
 */
public class Ruota extends Cerchio {
	public Ruota() {
		setRaggio(20);
	}
}
```

```
java src.esercizi.incapsulamento.Circonferenze

L'area di un cerchio di raggio 20 è: 1256.0
La circonferenza di un cerchio di raggio 20 è: 125.60000000000001
L'area di una ruota di raggio 20 è: 1256.0
La circonferenza di una ruota di raggio 20 è: 125.60000000000001
```

Notiamo subito che, nonostante il metodo costruttore di *Ruota* non contenga la chiamata esplicita al costruttore della superclasse, il risultato dell'esecuzione di *Circonferenze* è identico al caso analizzato in precedenza. L'anomalia è dovuta al fatto che, se il programmatore non effettua una chiamata esplicita al costruttore senza argomenti della classe base, Java esegue implicitamente tale chiamata.
Infine, possiamo modificare ulteriormente il costruttore della classe *Ruota*, affinché utilizzi il nuovo costruttore della classe *Cerchio* evitando di chiamare esplicitamente il metodo "setter" per impostare il valore della proprietà *raggio*.

```
/**
 * Una ruota può essere definita
 * a partire da un Cerchio generico
 */
public class Ruota extends Cerchio {
        public Ruota() {
                super(20);
        }
}
```

8.8 Aggiungere nuovi metodi

Quando estendiamo una classe base, possiamo aggiungere nuovi metodi alla classe derivata. Ad esempio, una *Macchina* generalmente possiede un avvisatore acustico. Aggiungendo il metodo *segnala()*, continueremo a mantenere tutte le vecchie funzionalità, ma ora la macchina è in grado di emettere segnali acustici.
Definire nuovi metodi all'interno di una classe derivata ci consente quindi di definire quelle caratteristiche particolari non previste nella definizione generica del concetto, e necessarie a specializzare le nuove classi.

```
package src.esercizi.ereditarieta;

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

        /**
         * Simula l'attivazione del segnalatore acustico della macchina.
         * @return void
         */
        public void segnala()
        {
                System.out.println(nome + "ha attivato il segnalatore acustivo ");
```

```
        }

}
```

```
package src.esercizi.ereditarieta;

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

8.9 Overriding di metodi

Capita spesso che, un metodo ereditato da una classe base non sia adeguato rispetto alla specializzazione della classe. Per ovviare al problema, Java ci consente di ridefinire il metodo originale semplicemente riscrivendo il metodo in questione, nella definizione della classe derivata.
Anche in questo caso, definendo nuovamente il metodo nella classe derivata, non si corre il pericolo di manomettere la superclasse. Il nuovo metodo sarà eseguito al posto del vecchio, anche se la chiamata fosse effettuata da un metodo ereditato dalla classe base.
La definizione della classe *Macchina*, potrebbe ridefinire il metodo *muovi(**int**)* affinché controlli che la velocità, passata come attributo, non superi la velocità massima di 120 Km/h consentiti al mezzo. Modifichiamo la definizione della classe *Macchina*:

```
package src.esercizi.ereditarieta;

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

        /**
         * Simula l'attivazione del segnalatore acustico della macchina.
         * @return void
        */
        public void segnala()
        {
                System.out.println(nome + "ha attivato il segnalatore acustivo ");
        }
```

```
        /** Simula la messa in marcia del veicolo consentendo di specificare
        * la velocità in km/h.
        * Questo metodo ridefinisce il metodo analogo
        * definito nella classse base Veicolo.
        * Per adattare il metodo al nuovo oggetto, il metodo
        * verifica che la velocità passata come argomento
        * sia al massimo pari alla velocità massima della macchina.
        * @return void
        * @param int velocita : velocita del veicolo
        */
         public void muovi(int velocita) {
                if(velocita<120)
                        this.velocita= velocita;
                else
                        this.velocita = 120;
                System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
         }

}
```

```
package src.esercizi.ereditarieta;

public class Autista
{
        public static void main(String args[])
        {
                Macchina fiat = new Macchina();
                fiat.muovi();
                //L'autista accelera fino al massimo della velocità
                fiat.muovi(120);
                //L'autista deve effettuare una serie di curve.
                //Prima di curvare frena la macchina
                fiat.muovi(60);
                fiat.sinistra();
                fiat.diritto();
                fiat.segnala();
                fiat.ferma();

        }
}
```

Nel metodo main della applicazione *Autista*, quando viene eseguito metodo *muovi(int)*, l'applicazione farà riferimento al metodo della classe *Macchina* e non a quello definito all'interno della classe base *Veicolo*.

8.10 Chiamare metodi della classe base

La parola chiave **super**, può essere utilizzata anche nel caso in cui sia necessario eseguire un metodo della superclasse, evitando che la JVM esegua il rispettivo metodo ridefinito nella classe derivata con il meccanismo di overriding.

Grazie alla parola chiave **super**, il programmatore non deve necessariamente riscrivere un metodo completamente, ma è libero di utilizzare parte delle funzionalità definite all'interno del metodo della classe base.
Esaminiamo ancora la definizione della classe *Macchina* ed in particolare sulla definizione del metodo *muovi(int)*:

```
/** Simula la messa in marcia del veicolo consentendo di specificare
* la velocità in km/h.
* Questo metodo ridefinisce il metodo analogo
* definito nella classse base Veicolo.
* Per adattare il metodo al nuovo oggetto, il metodo
* verifica che la velocità passata come argomento
* sia al massimo pari alla velocità massima della macchina.
* @return void
* @param int velocita : velocita del veicolo
*/
 public void muovi(int velocita) {
        if(velocita<120)
                this.velocita= velocita;
        else
                velocita = 120;
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
 }
```

Questo nuova versione del metodo implementata nella definizione della classe *Macchina*, effettua il controllo del valore della variabile passata come argomento, per limitare la velocità massima del mezzo. Al termine della operazione, esegue un assegnamento identico a quello effettuato nello stesso metodo definito nella superclasse *Veicolo* e riportato di seguito:

```
/** Simula la messa in marcia del veicolo specificando la velocità in km/h
        * @return void
        * @param int velocita : velocita del veicolo
        */

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

Il metodo *muovi()* della classe *Macchina*, può quindi essere riscritto affinchè sfrutti quanto già implementato nella superclasse nel modo seguente:

```
        /** Simula la messa in marcia del veicolo consentendo di specificare
        * la velocità in km/h.
        * Questo metodo ridefinisce il metodo analogo
        * definito nella classse base Veicolo.
        * Per adattare il metodo al nuovo oggetto, il metodo
        * verifica che la velocità passata come argomento
        * sia al massimo pari alla velocità massima della macchina.
        * @return void
```

```
* @param int velocita : velocita del veicolo
*/
 public void muovi(int velocita) {
        if(velocita<120)
                super.muovi(velocita);
        else
                super.muovi(120);
        System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
}
```

E' importante notare che, a differenza della chiamata ai metodi costruttori della superclasse, da eseguire facendo uso solo della parola chiave **super** ed eventualmente la lista degli argomenti trasmessi metodo, 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, come **this** è creato dalla JVM al momento della creazione dell'oggetto, e con la differenza che **super** fa sempre riferimento alla superclasse della classe attiva ad un determinato istante.

8.11 Compatibilità tra 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 oggetto derivato da essa, nella gerarchia definita dalla ereditarietà.

```
Veicolo veicolo = new Macchina();

//Eseguo il metodo muovi(int) definito nella classe Macchina
veicolo.muovi(10);
```

La ragione alla base di questa funzionalità è che, 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, questo tipo di riferimento effettuerà una chiamata al nuovo metodo.
Quanto detto ci consente di definire il concetto di "compatibilità" e "compatibilità stretta" tra variabili reference.

Una variabile reference A si dice strettamente compatibile con la variabile reference B se, A fa riferimento ad un oggetto definito per ereditarietà dall'oggetto riferito da B.

Viceversa, la variabile reference B si dice compatibile con la variabile reference A se, B fa riferimento ad un oggetto che rappresenta la classe base per l'oggetto riferito da A.

Nell'esempio successivo, la variabile reference *ferrari* di tipo *Macchina*, è strettamente compatibile con la variabile reference *carro* di tipo *Veicolo*, dal momento che *Macchina* è definita per ereditarietà da *Veicolo*. Viceversa, la variabile reference *carro* di tipo *Veicolo* è compatibile con la variabile reference *ferrari* di tipo *Macchina*.

```
Macchina ferrari = new Macchina();
Veicolo carro = new Veicolo();
```

8.12 Run-time e compile-time

Prima di procedere oltre, è necessario fermarci qualche istante per introdurre i due concetti di *tipo a run-time* e *tipo a compile-time*.

Il tipo a compile-time di una espressione, è il tipo dell'espressione come dichiarato formalmente nel codice sorgente.

Il tipo a run-time è il tipo dell'espressione determinato durante l' esecuzione della applicazione.

I tipi primitivi (int, float, double etc.) rappresentano sempre lo stesso tipo sia al run-time che al compile-time.

Il tipo a *compile-time* è sempre costante, quello a *run-time* variabile. Ad esempio, la variabile reference *veicolo* di tipo *Veicolo*, rappresenta il tipo *Veicolo* a *compile-time*, e il tipo *Macchina* a *run-time*.

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

Volendo fornire una regola generale, diremo che:

Il tipo rappresentato al compile-time da una espressione è specificato nella sua dichiarazione, mentre quello a run-time è il tipo attualmente rappresentato.

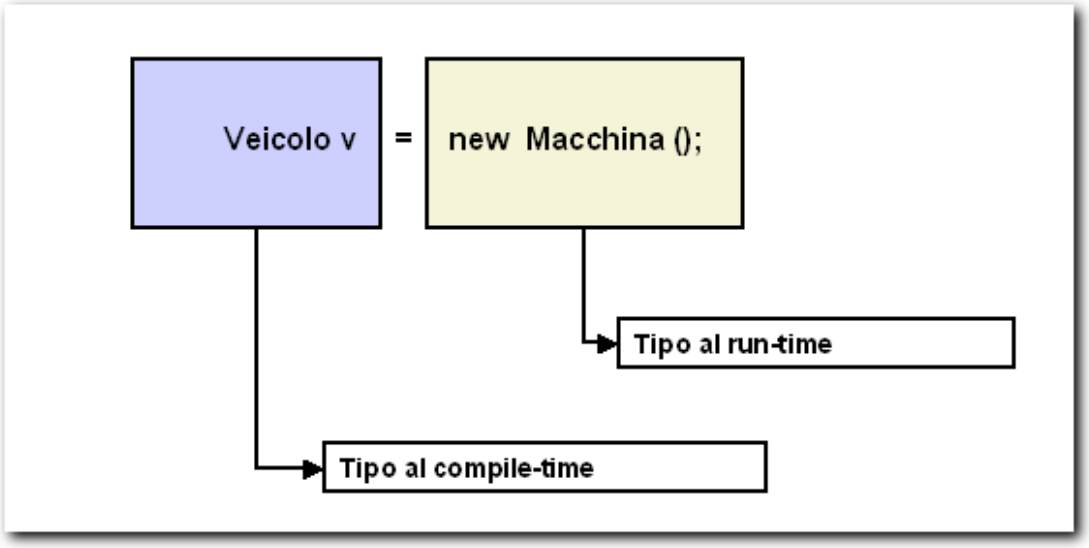


Figura 65: run-time e compile-time

Per riassumere, ecco alcuni esempi:

1. V ha tipo Veicolo a *compile-time* e Macchina al *run-time*

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

2. Il tipo a *run-time* di v cambia in *Veicolo*

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

3. i rappresenta un tipo **int** sia a *run-time* che a *compile-time*

```
int i=0;
```

E' comunque importante sottolineare che, una variabile reference potrà fare riferimento solo ad oggetti il cui tipo è in qualche modo compatibile con il tipo rappresentato al *compile-time*. Questa compatibilità, come già definito nel paragrafo precedente, è rappresentata dalla relazione di ereditarietà: tipi derivati sono sempre compatibili con le variabili reference dei predecessori.

8.13 Accesso a metodi attraverso variabili reference

Tutti i concetti finora espressi, hanno un impatto rilevante sulle modalità di accesso ai metodi di un oggetto. Consideriamo ad esempio la prossima applicazione:

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
        {
```

```
                Macchina fiat = new Macchina();
                Veicolo v = fiat;
                fiat.segnala();
                //Questa è una chiamata illegale
                v.segnala();


        }
}
```

Se proviamo a compilare il codice, il compilatore produrrà un messaggio di errore relativo alla riga otto del codice sorgente. Dal momento che, il tipo rappresentato da una variabile al *run-time* può cambiare, il compilatore assumerà che la variabile reference *v* sta facendo riferimento all'oggetto del tipo rappresentato al *compile-time* (*Veicolo*).
In altre parole, anche se una classe *Macchina* possiede un metodo *suona()*, questo non sarà utilizzabile tramite una variabile reference di tipo *Veicolo*.
La situazione non si verifica se modifichiamo l'applicazione nel modo seguente:

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
        {
                Veicolo v = new Veicolo();
                Macchina fiat = v;
                fiat.muovi(120);

        }
}
```

Le conclusioni che possiamo trarre, possono essere riassunte come segue:

1. *Se A è una variabile reference strettamente compatibile con la variabile reference B, allora A avrà accesso a tutti i metodi di entrambe le classi;*

2. *Se B è una variabile reference compatibile con la variabile reference A, allora B potrà accedere solo ai metodi che la classe riferita da A ha ereditato.*

8.14 Cast dei tipi

Java, fornisce un modo per girare intorno alle limitazioni imposte dalle differenze tra tipo al *run-time* e tipo al *compile-time*, permettendo di risolvere il problema verificatosi con la prima versione della applicazione *MacchinaVSVeicolo*.

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
```

```
        {
                Macchina fiat = new Macchina();
                Veicolo v = fiat;
                fiat.segnala();
                //Questa è una chiamata illegale
                v.segnala();


        }
}
```

Il **cast** di un tipo, è una tecnica che consente di dichiarare alla JVM che una variabile reference temporaneamente rappresenterà un tipo differente da quello rappresentato al compile-time. La sintassi necessaria a realizzare un'operazione di cast di tipo è la seguente:

(nuovi_tipo) nome

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

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
        {
                Macchina fiat = new Macchina();
                Veicolo v = fiat;
                fiat.segnala();
                //Questa è una chiamata legale
                ((Macchina)v).segnala();


        }
}
```

L'operazione di cast è possibile su tutti i tipi purché il tipo della variabile reference ed il nuovo tipo siano almeno *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 dell'applicazione se, il tipo rappresentato al *run-time* dall'oggetto non rappresenta il tipo desiderato al momento dell'esecuzione.

8.15 L'operatore instanceof

Poiché, in un'applicazione Java esistono un gran numero di variabili reference, è a volte utile determinare al *run-time* il tipo di oggetto cui una variabile sta facendo riferimento. 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 sarà 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 consente di rappresentare il tipo rappresentato da B. *False*, indica che A fa riferimento all'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** restituirà *true*.
Modifichiamo l'applicazione *MacchinaVSVeicolo*:

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
        {
                Veicolo v = new Macchina();
                v.segnala();
        }
}
```

In questo caso, Java produrrà un errore di compilazione in quanto, il metodo *segnala()* 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:

```
public class MacchinaVSVeicolo
{
        public static void main(String args[])
        {
                Veicolo v = new Macchina();
                if(v instanceof Macchina)
                        ((Macchina)v).segnala();
        }
}
```

8.16 L'oggetto Object

La gerarchia delle classi delle Java Core API, parte dalle classe Object che, ne rappresenta la radice dell'albero, come mostrato nella prossima figura.

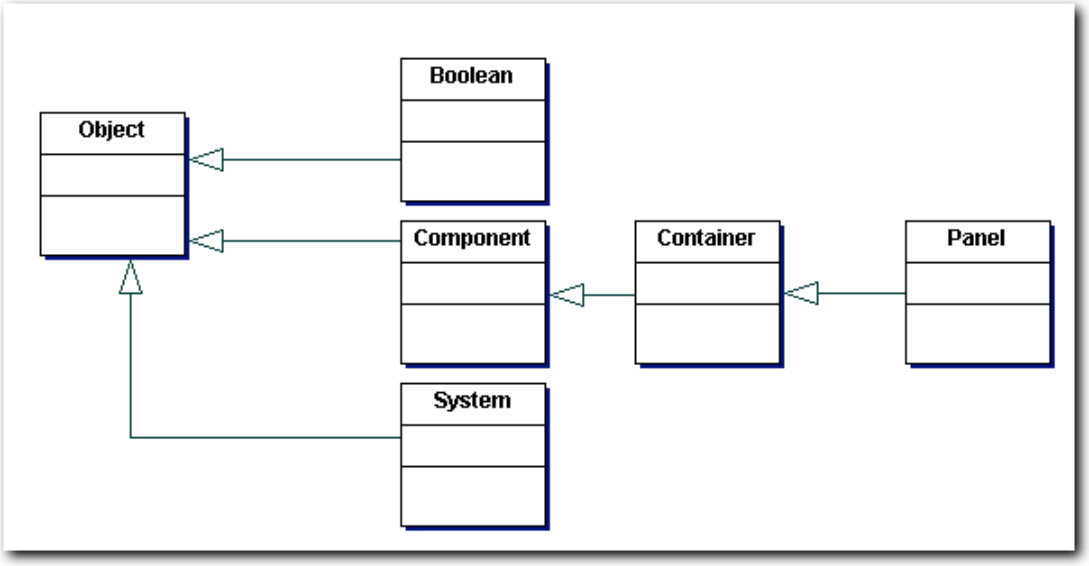


Figura 66: La classe Object

In realtà, ogni volta che definiamo una nuova classe Java senza utilizzare il meccanismo dell'ereditarietà ovvero, la classe non estende nessuna classe base, Java ne provocherà automaticamente l'estensione dell'oggetto Object.
Questa caratteristica degli oggetti Java, nasce principalmente allo scopo di garantire alcune funzionalità base comuni a tutte le classi, includendo la possibilità di esprimere lo stato di un oggetto in forma di stringa, la possibilità di comparare due oggetti o terminarne uno tramite il metodo ereditato *finalize()*.
Quest'ultimo metodo, è utilizzato dal garbage collector nel momento in cui elimina l'oggetto rilasciando la memoria e può essere modificato per gestire situazioni non controllabili dal garbage collector come i riferimenti circolari.

8.17 Il metodo equals()

Abbiamo già anticipato che, l'operatore di booleano di uguaglianza '==', non è sufficiente a confrontare due oggetti poiché opera a livello di variabili referenze e di conseguenza, confronta due puntatori a locazioni di memoria producendo il valore *true* se e solo se le due variabili reference puntano allo stesso oggetto. Abbiamo inoltre detto che, due oggetti sono uguali se è uguale il loro stato.
Il metodo *equals()*, ereditato dalla classe *Object*, 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, se due oggetti di tipo compatibile si trovano nello stesso stato.
Immaginiamo di dover confrontare oggetti di tipo Integer:

package src.esercizi.confronto;

```
/**
 * Questa applicazione esegue il confromtro
 * tra oggetti di tipo Integer utilizzando
 * il metodo equals ereditato da Object
 */
public class MetodoEquals {
	public static void main(String[] argv) {
		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,il risultato prodotto prodotto sarà il seguente:

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

Proviamo ora a definire una nuova classe che, rappresenta un punto sul piano:

```
package src.esercizi.confronto;

public class Punto {
	public int x, y;

	public Punto(int xc, int yc) {
		x = xc;
		y = yc;
	}

	public static void main(String args[]) {
		Punto a, b;
		a = new Punto(1, 2);
		b = new Punto(1, 2);

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

		//Uguaglianza mediante metodo equals
		if (a.equals(b))
			System.out.println("a è uguale a b");
		else
		System.out.println("a è diverso da b");
```

```
        }
}
```

Il confronto tra variabili reference vale false
a è diverso da b

Lo stato dell'oggetto Punto, è rappresentato dal valore dei due attributi *x* e *y* che rappresentano rispettivamente le ascisse e le coordinate del punto su un piano cartesiano. All'interno del metodo main dell'applicazione, vengono creati due oggetti tra loro compatibili ed aventi lo stesso stato ma, il confronto *a.equals(b)* restituisce il valore *false* contrariamente a quanto ci aspettavamo. Dal momento che il metodo *equals()* esegue un confronto a livello di stato tra due oggetti, a causa dei diversi aspetti che le definizioni di classe possono assumere, l'implementazione definita all'interno della definizione della classe *Object* potrebbe non essere sufficiente. E' quindi necessario che il programmatore riscriva il metodo in questione utilizzando il meccanismo di overriding. Modificando la classe Punto inserendo la nuova definizione del metodo object al suo interno, il risultato della applicazione sarà finalmente quello atteso.

```
package src.esercizi.confronto;

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[]) {
                Punto a, b;
                a = new Punto(1, 2);
                b = new Punto(1, 2);

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

                //Uguaglianza mediante metodo equals
                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 risultato:

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

8.18 Rilasciare risorse esterne

Quando un oggetto viene creato mediante l'operatore **new**, il linguaggio Java ci assicura che, verrà comunque eseguito un metodo costruttore affinché lo stato iniziale dell'oggetto sia correttamente impostato. Analogamente, quando un oggetto viene rilasciato, il garbage collector Java ci assicura che, prima di rilasciare l'oggetto e liberare la memoria allocata sarà eseguito il metodo *finalize()* della classe.
Tipicamente, questo metodo è utilizzato in quei casi in cui sia necessario gestire situazioni di riferimenti circolari di tra oggetti, oppure circostanze 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[10], *finalize()* può essere utilizzato per consentire al programmatore di implementare meccanismi di gestione esplicita della memoria.
Il metodo ereditato dalla classe *Object* non fa nulla.

8.19 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 oppure, viene utilizzato per trasformare in forma di stringa un particolare dato rappresentato dall'oggetto. Ad esempio, nel caso dell'oggetto *Integer*, il metodo *toString()*, viene utilizzatio per convertire in forma di stringa il numero intero rappresentato dall'oggetto:

```
String valore_in_forma_di_stringa;
Integer intero_in_forma_di_oggetto = new Integer(100);
valore_in_forma_di_stringa = intero_in_forma_di_oggetto.toString();
```

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

L'importanza del metodo *toString()* è legata al fatto che, il metodo statico *System.out.println()* utilizza *toString()* per la stampa dell'output a terminale come mostrato nel prossimo esempio:

```
package src.esercizi.confronto;

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 String toString() {
		return "("+x+","+y+")";
	}

	public static void main(String args[]) {
		Punto a, b;
		a = new Punto(1, 2);
		b = new Punto(1, 2);
		c = new Punto(10, 3);

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

		//Uguaglianza mediante metodo equals
		if (a.equals(b))
			System.out.println(a+" è uguale a "+b);
		else
			System.out.println(a+" è diverso da "+b);
		if (a.equals(c))
			System.out.println(a+" è uguale a "+c);
		else
			System.out.println(a+" è diverso da "+c);
	}
}
```

```
Il confronto tra variabili reference vale false
(1,2) è uguale a (1,2)
(1,2) è diverso da  (10,3)
```

8.20 Giochi di simulazione

Tutti i linguaggi ad oggetti si prestano facilmente alla creazione di simulazioni anche molto complesse. Nel prossimo esercizio, a partire dalla classe *Veicolo*, definiremo nuovi tipi di veicolo attraverso il meccanismo della ereditarietà e trasformeremo la applicazione *Autista* in una definizione di classe completa (non più contenente solo il metodo main).
La nuova definizione conterrà un costruttore che richiede un *Veicolo* come attributo e sarà in grado di riconoscerne i limiti di velocità. Aggiungeremo alla classe *Autista* il metodo *sorpassa(Veicolo)*.
Infine, utilizzeremo una nuova applicazione che, creati alcuni oggetti di tipo *Autista* e *Veicolo*, completa il gioco di simulazione.
Il *class-diagram* relativo all'esercizio è mostrato nella prossima figura:

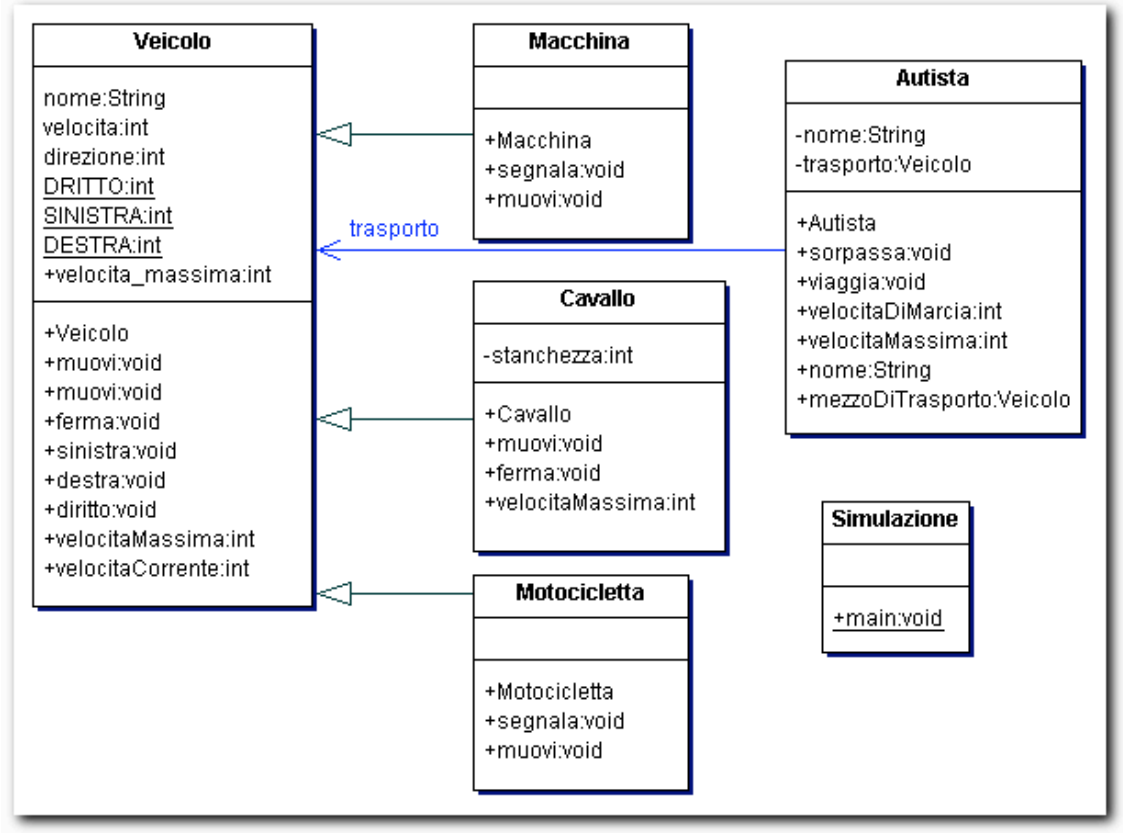


Figura 67: Class-diagram della applicazione Simulazione

```
package src.esercizi.ereditarieta;
public class Veicolo {
        /** Nome del veicolo */
        String nome;

        /** Rappresenta la velocità del veicolo espressa in Km/h */
        int velocita;
```

```
/** Direzione di marcia del veicolo */
int direzione;

/** Costante che indica la direzione "dritta" di marcia del veicolo */
final static int DRITTO = 0;

/** Costante che indica la svola a sinistra del veicolo */
final static int SINISTRA = -1;

/** Costante che indica la svola a destra del veicolo */
final static int DESTRA = 1;
public int velocita_massima;

public Veicolo() {
        velocita = 0;
        direzione = DRITTO;
        nome = "Veicolo generico";
}

/** Simula la messa in marcia del veicolo alla velocità di 1 km/h */
public void muovi() {
        velocita = 1;
        System.out.println(nome + " si sta movendo a: " + velocita + " Kmh");
}

/** Simula la messa in marcia del veicolo specificando la velocità in km/h
* @return void
* @param int velocita : velocita del veicolo
*/
public void muovi(int velocita) {
        this.velocita = velocita;
        System.out.println(nome + " si sta movendo a: " + velocita + " Kmh");
}

/** Simula la frenata del veicolo
*@return void
*/
public void ferma() {
        velocita = 0;
        System.out.println(nome + " si è fermato");
}

/** Simula la svola a sinistra del veicolo
*@return void
*/
public void sinistra() {
        direzione = SINISTRA;
        System.out.println(nome + " ha sterzato a sinistra");
}

/** Simula la svola a destra del veicolo
*@return void
*/
public void destra() {
        direzione = DESTRA;
        System.out.println(nome + " ha sterzato a destra");
}

/** Simula la conclusione di una svola a sinistra o a destra del veicolo
```

```
	*@return void
	*/
	public void diritto() {
		direzione = DRITTO;
		System.out.println(nome + " sta procedendo in linea retta");
	}

	/** Restituisce la velocità massima del veicolo
	*@return int velocità massima del veicolo in Km/h
	*/
	public int velocitaMassima() {
		return velocita_massima;
	}

	/** Restituisce la velocità di marcia del veicolo
	*@return int velocità massima del veicolo in Km/h
	*/
	public int velocitaCorrente() {
		return velocita;
	}
}
```

La classe base *Veicolo* è stata modificata rispetto alla versione precedente. Ora, è possibile impostare una velocità massima per il veicolo e grazie a due appositi metodi è possibile conoscere la velocità di marcia del mezzo di trasporto nonché la sua velocità massima.

```
package src.esercizi.ereditarieta;
public class Cavallo extends Veicolo {
	private int stanchezza; // range da 0 a 10

	public Cavallo(String nome) {
		velocita = 0;
		velocita_massima = 20;
		direzione = DRITTO;
		this.nome = nome;
		stanchezza = 0;
	}

	/**
	* La marcia di un cavallo è soggetta non solo a restrizioni
	* rispetto alla velocità massima dell'animale,
	* ma dipende dalla stanchezza del cavallo.
	* @return void
	* @param int velocita : velocita del cavallo
	*/
	public void muovi(int velocita) {
		if (velocita > velocitaMassima())
			velocita = velocitaMassima();
		super.muovi(velocita);
		if (velocita > 10 && stanchezza < 10)
			stanchezza++;
	}

	/**
	* Quando il cavallo si ferma, la stanchezza dimezza
	* poiché il cavallo può riposarsi.
```

```
        * @return void
        */
        public void ferma()
        {
                stanchezza = stanchezza /2;
                super.ferma();
        }

        public int velocitaMassima()
        {
                return velocita_massima - stanchezza;
        }
}
```

La classe *Cavallo*, rappresenta un tipo particolare di mezzo di trasporto in quanto, la velocità massima dipende da fattori differenti quali la stanchezza dell'animale. Per simulare questa caratteristica, la classe contiene un dato membro di tipo intero che rappresenta la stanchezza del cavallo, può contenere solo valori da 0 a 10. Il valore dell'attributo *stanchezza* viene incrementato nel metodo *muovi* e dimezzato nel metodo *ferma*.

```
package src.esercizi.ereditarieta;
public class Macchina extends Veicolo {
        public Macchina(String marca)
        {
                velocita_massima = 120;
                velocita=0;
                direzione = DRITTO;
                nome = marca;
        }

        /**
        * Simula l'attivazione del segnalatore acustico della macchina.
        * @return void
        */
        public void segnala()
        {
                System.out.println(nome + "ha attivato il segnalatore acustivo ");
        }

        /** Simula la messa in marcia del veicolo consentendo di specificare
        * la velocità in km/h.
        * Questo metodo ridefinisce il metodo analogo
        * definito nella classse base Veicolo.
        * Per adattare il metodo al nuovo oggetto, il metodo
        * verifica che la velocità passata come argomento
        * sia al massimo pari alla velocità massima della macchina.
        * @return void
        * @param int velocita : velocita del veicolo
        */
        public void muovi(int velocita) {
                if(velocita< velocita_massima)
                        this.velocita= velocita;
                else
                        this.velocita = velocita_massima;
                System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
```

```
        }
}
```

```
package src.esercizi.ereditarieta;
public class Motocicletta extends Veicolo {
        public Motocicletta(String marca)
        {
                velocita_massima = 230;
                velocita=0;
                direzione = DRITTO;
                nome = marca;
        }

        /**
        * Simula l'attivazione del segnalatore acustico della macchina.
        * @return void
        */
        public void segnala()
        {
                System.out.println(nome + "ha attivato il segnalatore acustivo ");
        }

        /** Simula la messa in marcia del veicolo consentendo di specificare
        * la velocità in km/h.
        * Questo metodo ridefinisce il metodo analogo
        * definito nella classse base Veicolo.
        * Per adattare il metodo al nuovo oggetto, il metodo
        * verifica che la velocità passata come argomento
        * sia al massimo pari alla velocità massima della macchina.
        * @return void
        * @param int velocita : velocita del veicolo
        */
        public void muovi(int velocita) {
                if(velocita< velocita_massima)
                        this.velocita= velocita;
                else
                        this.velocita = velocita_massima;
                System.out.println(nome + "si sta movendo a: "+ velocita+" Kmh");
        }
}
```

La prossima definizione di classe rappresenta l'autista generico che dovrà guidare uno dei mezzi sopra definiti.
La definizione di classe, contiene una serie di metodi necessari a determinare lo stato del mezzo guidato dall'autista rappresentato nonché, ad impostare o modificare lo stato iniziale dell'oggetto ma, il metodo centrale è quello che simula il sorpasso tra due automobilisti.
Il metodo *sorpassa(Autista)*, le cui attività sono schematizzate nel prossimo *activity-diagram*, primo richiede la velocità del veicolo da sorpassare e calcola la eventuale nuova velocità enecessaria ad effettuare il sorpasso. Secondo, tente di accelerare fino alla nuova velocità calcolata. Solo nel caso in cui la velocità finale sarà maggiore del veicolo da superare, allora il sorpasso potrà ritenersi concluso.

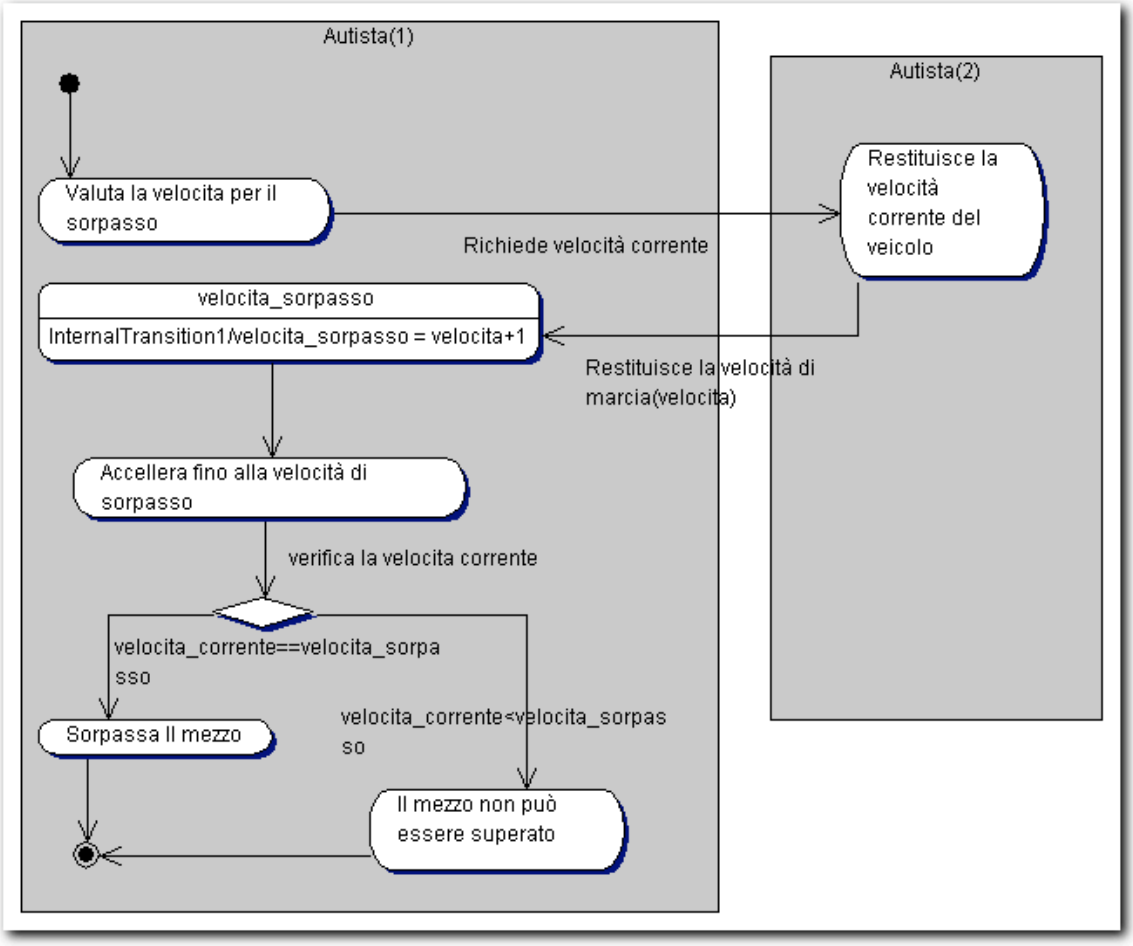


Figura 68: Sorpasso tra due autisti

```
package src.esercizi.ereditarieta;
public class Autista {
	/**
	* Nome dell'autista del veicolo
	*/
	private String nome;

	/**
	* Veicolo utilizzato dall'autista
	*/
	private Veicolo trasporto;

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

	/**
	* Simula il tentativo di sorpasso di un altro autista
	* @return void
	* @param Autista : autista da superare
	*/
	public void sorpassa(Autista altro_autista) {
```

```
                int velocita_sorpasso = altro_autista.mezzoDiTrasporto().velocitaCorrente()+1;
                System.out.println(nome +"sta sorpassando "+ altro_autista.trasporto.nome +"con"+
                        trasporto.nome);
                trasporto.muovi(velocita_sorpasso);
                if (trasporto.velocitaCorrente() < 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");
        }

        /**
        * inizia la marcia del veicolo
        * @return void
        * @param int : velocità di marcia del veicolo
        */
        public void viaggia(int velocita_di_marcia) {
                trasporto.diritto();
                trasporto.muovi(velocita_di_marcia);
        }

        /**
        * Restituisce la velocità di marcia del veicolo
        * @return int : velocità di marcia del veicolo in uso
        */
        public int velocitaDiMarcia() {
                return trasporto.velocitaCorrente();
        }
        /**
        * Restituisce la velocità massima del veicolo
        * @return int : velocità massima del veicolo in uso
        */
        public int velocitaMassima() {
                return trasporto.velocitaMassima();
        }
        /**
        * Restituisce il nome dell'autista
        * @return String : nome dell'autista
        */
        public String nome() {
                return nome;
        }
        /**
        * Restituisce il veicolo usato dall'autista
        * @return Veicolo : veicolo in uso
        */
        public Veicolo mezzoDiTrasporto()
        {
                return trasporto;
        }

}
```

La prossima definizione di classe contiene il metodo main per l'esecuzione della simulazione. La sequenza delle operazioni compiute all'interno del metodo main della applicazione, sono rappresentate nel prossimo *seuquence-diagram*:

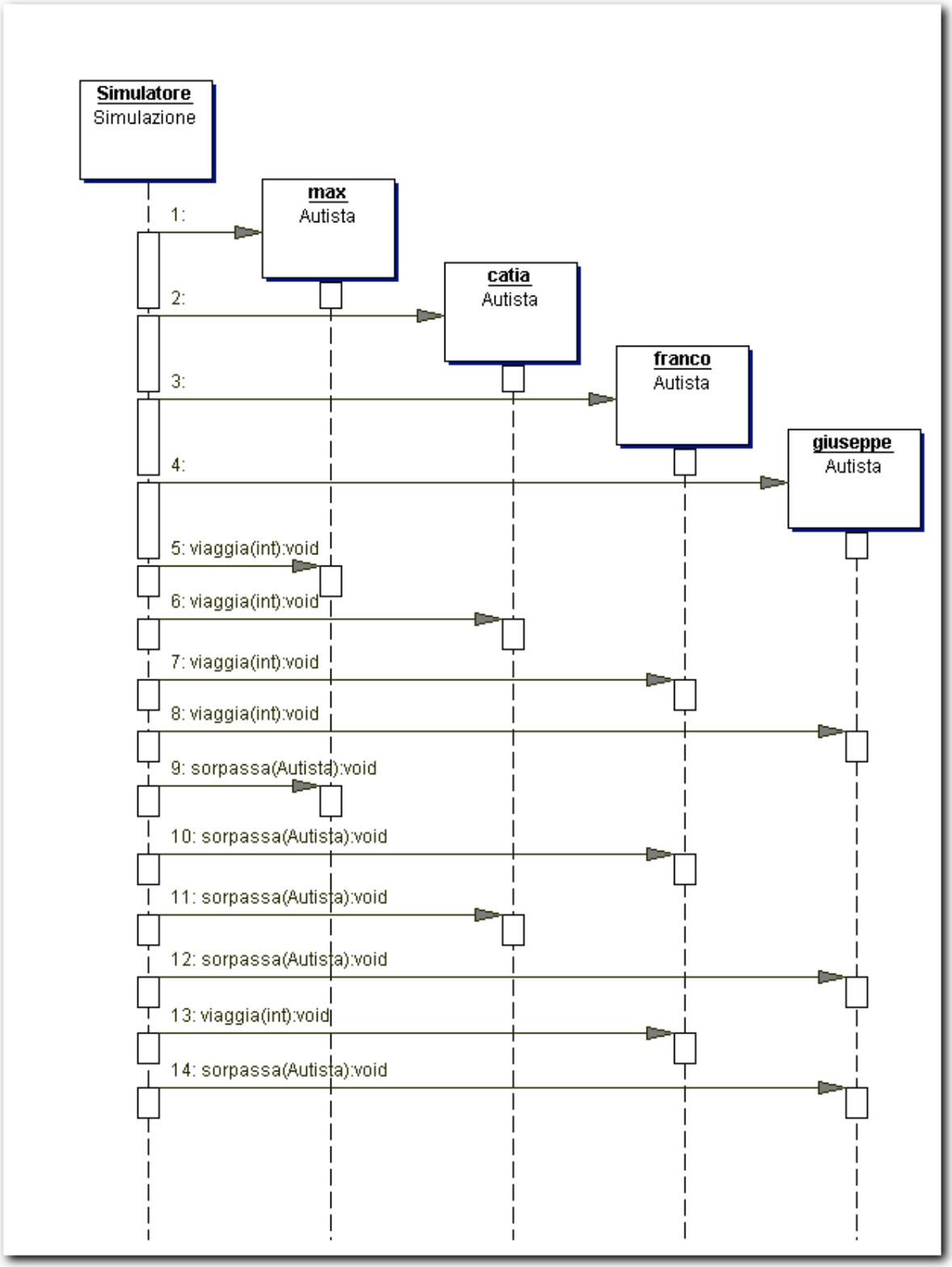


Figura 69: Sequenze-Diagram per Simulazione

```
package src.esercizi.ereditarieta;
public class Simulazione {
	public static void main(String[] argv) {
		Autista max = new Autista("Max", new Cavallo("Furia"));
		Autista catia = new Autista("Catia", new Macchina("Fiat 600"));
		Autista franco = new Autista("Franco", new Motocicletta("Suzuky XF 650"));
		Autista giuseppe = new Autista("Giuseppe", new Motocicletta("Honda CBR 600"));

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

		max.sorpassa(franco);
		franco.sorpassa(catia);
		catia.sorpassa(max);
		giuseppe.sorpassa(franco);

		franco.viaggia(franco.mezzoDiTrasporto().velocitaMassima());
		giuseppe.sorpassa(franco);

	}
}
```

Il risultato della esecuzione della applicazione Simulazione è il seguente:

```
java src.esercizi.ereditarieta.Simulazione

Furia sta procedendo in linea retta
Furia si sta movendo a: 15 Kmh
Suzuky XF 650 sta procedendo in linea retta
Suzuky XF 650 si sta movendo a: 30 Kmh
Fiat 600 sta procedendo in linea retta
Fiat 600si sta movendo a: 40 Kmh
Honda CBR 600 sta procedendo in linea retta
Honda CBR 600 si sta movendo a: 40 Kmh
Maxsta sorpassando Suzuky XF 650conFuria
Furia si sta movendo a: 19 Kmh
Questo trasporto è troppo lento per superare
Francosta sorpassando Fiat 600conSuzuky XF 650
Suzuky XF 650 si sta movendo a: 41 Kmh
L'autista Catia ha mangiato la mia polvere
Catiasta sorpassando FuriaconFiat 600
Fiat 600si sta movendo a: 20 Kmh
L'autista Max ha mangiato la mia polvere
Giuseppesta sorpassando Suzuky XF 650conHonda CBR 600
Honda CBR 600 si sta movendo a: 42 Kmh
L'autista Franco ha mangiato la mia polvere
Suzuky XF 650 sta procedendo in linea retta
Suzuky XF 650 si sta movendo a: 230 Kmh
Giuseppesta sorpassando Suzuky XF 650conHonda CBR 600
```

Honda CBR 600 si sta movendo a: 231 Kmh
Questo trasporto è troppo lento per superare

9 ECCEZIONI

9.1 Introduzione

Le eccezioni Java sono utilizzate in quelle situazioni in cui sia necessario gestire condizioni anomale, ed i normali meccanismi si dimostrano insufficienti ad indicare completamente una condizione di errore o, un'eventuale anomalia. Formalmente, un'eccezione è un evento che si scatena durante l'esecuzione di un programma, causando l'interruzione del normale flusso delle operazioni. Queste condizioni di errore possono svilupparsi in seguito ad una gran 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 dell'esecuzione del metodo corrente.
Il linguaggio Java, cerca di risolvere alcuni di questi problemi al momento della compilazione del codice sorgente, tentando di prevenire ambiguità che potrebbero essere possibili cause di errori, ma non è in grado di gestire situazioni 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 allocare, 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.
Torniamo ancora una volta a prendere in considerazione la classe *Pila*:

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */

class Pila {
        int[] dati;
        int cima;

        /**
        * Il metodo push ritorna un tipo void e prende come
        * parametro un numero intero da inserire sulla cima della pila
        * @return void
        * @param int dato : elemento da inserire sulla cima della pila
        */
         public void push(int dato) {
```

```
                if(cima < dati.length)
                {
                        dati[cima] = dato;
                        cima ++;
                }

        }

        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return int : dato sulla cima della pila
        */
        int pop()
        {
                if(cima > 0)
                {
                        cima--;
                        return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

Dal momento che il metodo *push()* non prevede parametri di ritorno, è necessario un meccanismo alternativo per gestire un eventuale errore causato da un trabocco dell'array, a seguito dell'inserimento di un ventunesimo elemento (ricordiamo che l'array può contenere solo 20 numeri interi).
Il metodo *pop()* a sua volta, è costretto ad utilizzare il valore zero come parametro di ritorno nel caso in cui lo stack non possa contenere più elementi. Questo ovviamente costringere ad escludere il numero intero zero, dai valori che potrà contenere la pila e dovrà essere riservato alla gestione dell'eccezione.
Un altro aspetto da considerare quando si parla di gestione degli errori, è quello legato alla difficoltà nel descrivere e controllare situazioni arbitrariamente complesse. Immaginiamo una semplice funzione 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.
Grazie alla loro caratteristica di "oggetti particolari", le eccezioni si prestano facilmente alla descrizione di situazioni complicate, fornendo al programmatore la capacità di rappresentare 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.

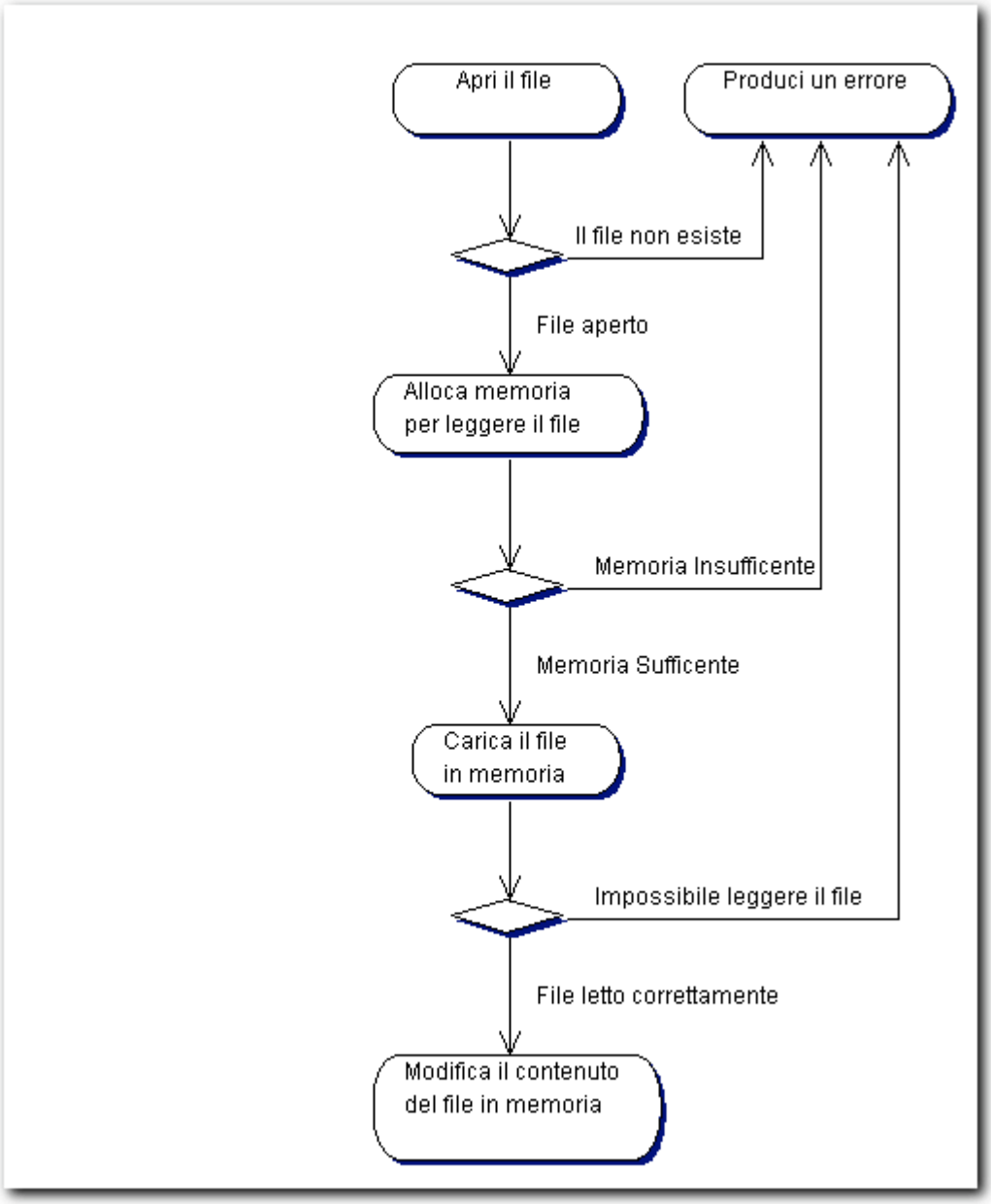


Figura 70: Schema di gestione degli errori in una funzione di lettura di un file

9.2 Propagazione di oggetti

Il punto di forza delle eccezioni, consiste nel permettere la propagazione di un oggetto a ritroso, attraverso la sequenza corrente di chiamate tra metodi. Opzionalmente, ogni metodo può:

1. *Catturare l'oggetto per gestire la condizione di errore utilizzando le informazioni trasportate, terminandone la propagazione;*
2. *Prolungare la propagazione ai metodi subito adiacenti nella sequenza di chiamate.*

Ogni metodo, che non sia in grado di gestire l'eccezione, è interrotto nel punto in cui aveva chiamato il metodo che sta propagando l'errore. Se la propagazione di un'eccezione raggiunge il metodo main dell'applicazione senza essere arrestata, l'applicazione termina in maniera incontrollata.
Consideriamo l'esempio seguente:

```
class Eccezioni
{
        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 è determinato sulla base di quello restituito da *metodo2()* di tipo **float**. A sua volta, *metodo2()* restituisce un valore **float** calcolato in base a quello di ritorno del *metodo3()* che, sotto determinate condizioni, genera un'eccezione.
L'esecuzione del *metodo1()* genera quindi la sequenza di chiamate schematizzata nella *Figura 72*.
Se si verificano le condizioni a seguito delle quali *metodo3()* genera l'eccezione, l'esecuzione del metodo corrente si blocca e l'eccezione viene propagata a ritroso verso *metodo2()* e *metodo1()* (*Figura 72*).
Una volta propagata, l'eccezione deve essere intercettata e gestita. In caso contrario si propagherà sino al metodo *main()* dell'applicazione causando la terminazione dell'applicazione.
Propagare un oggetto è detto "**exception throwing**" e fermarne la propagazione "**exception catching**".

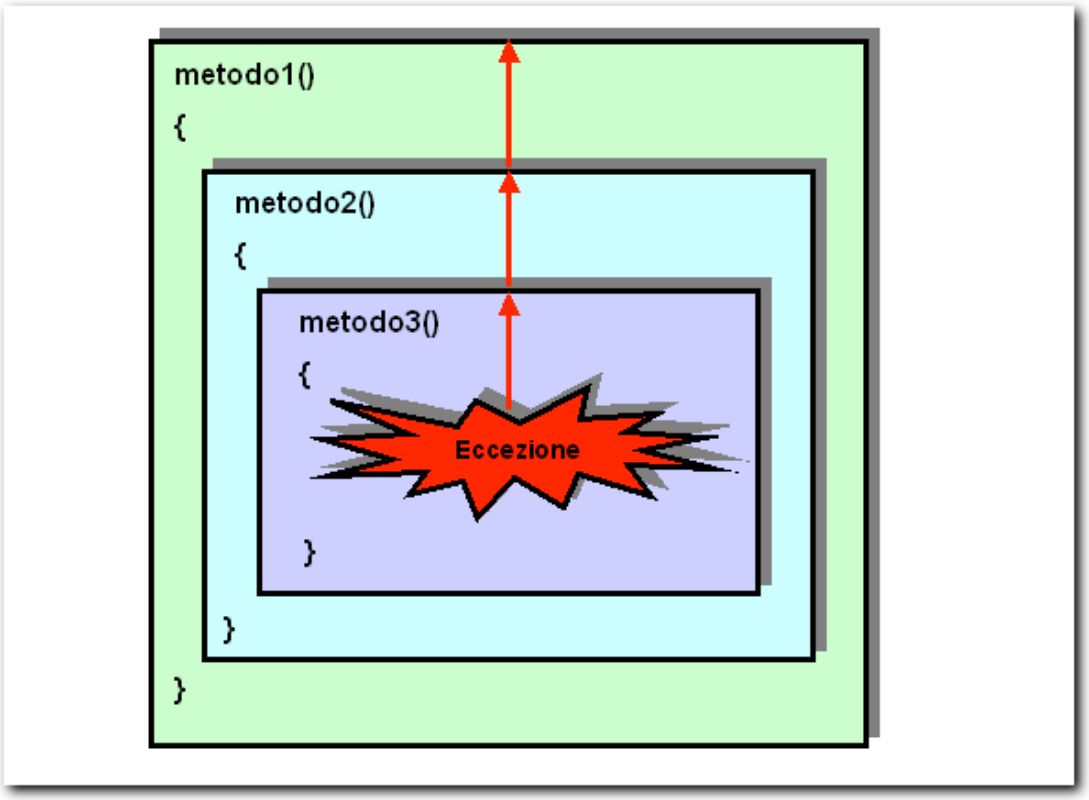


Figura 71: Propagazione di una eccezione Java

In generale, gli oggetti da propagare come eccezioni devono derivare dalla classe base *java.lang.Exception* appartenente alle Java Core API. A partire da questa, è possibile creare nuovi tipi di eccezioni per mezzo del meccanismo dell'ereditarietà, specializzando il codice secondo il caso da gestire.

9.3 Oggetti throwable

Come abbiamo detto, Java consente di propagare solo alcuni tipi di oggetti. Di fatto, Java richiede che tutti gli oggetti da propagare siano derivati dalla classe base *java.lang.Throwable*.
Nonostante questo sembri 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.
Tecnicamente, tutte le eccezioni generate dal linguaggio Java derivano dalla classe *java.lang.Throwable*, per convenzione invece, ogni eccezione definita dal programmatore deve derivare da *java.lang.Exception* che a sua volta deriva da *Throwable* (*Figura 72*).

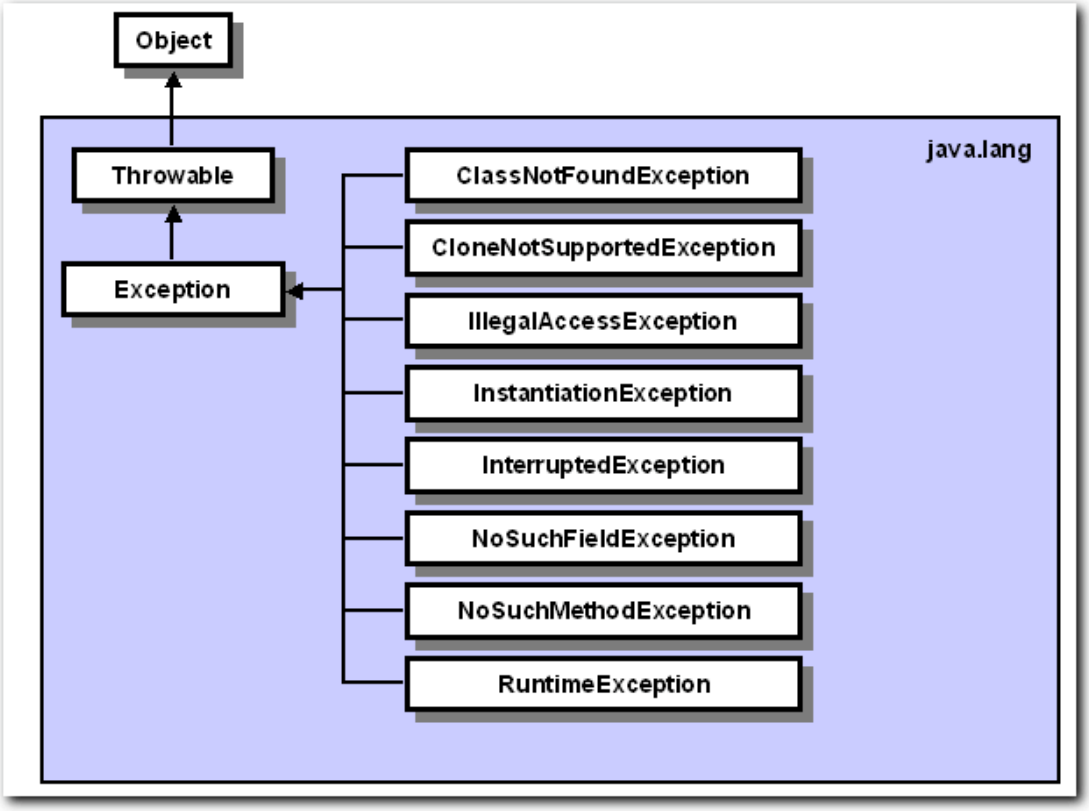


Figura 72: Albero di derivazione delle eccezioni

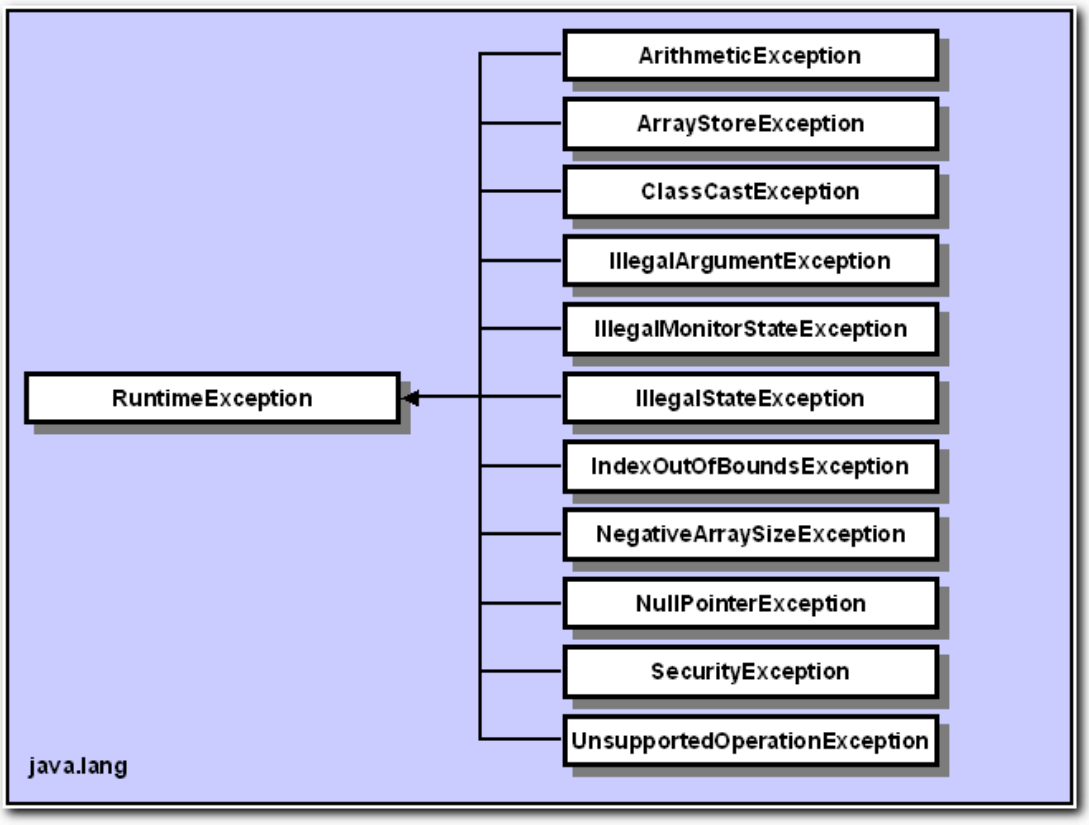


Figura 73: Albero di derivazione delle eccezioni incontrollate

La classe *Throwable* contiene i metodi necessari a gestire lo "stack tracing[11]", ovvero la sequenza delle chiamate tra metodi, per la propagazione dell'oggetto a ritroso lungo la sequenza corrente delle chiamate. Questa classe ha due costruttori:

Throwable();
Throwable(String);

Entrambi i costruttori di Throwable, hanno la responsabilità di mettere in moto il meccanismo di propagazione, il secondo, in più, imposta un dato membro di tipo *String* racchiudente un messaggio contenente lo stato dettagliato dell'errore al momento della generazione.
Il messaggio di errore trasportato da un oggetto Throwable, è accessibile grazie al metodo *toString()* ereditato da *Object*.
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.

[11] Al fine garantire la propagazione a ritroso, java utilizza uno stack (LIFO) per determinare la catena dei metodi chiamanti e risalire nella gerarchia.

Il metodo ***public void*** *printStackTrace()* consente invece di stampare sullo standard error la sequenza restituita dal metodo precedente. Consideriamo l'esempio seguente:

```
public 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 a *metodo1()* passandogli come argomento un array nullo, array che viene passato a sua volta al *metodo2()* che tenta di visualizzarne sul terminale il valore del primo elemento.
Essendo nullo l'array, nel momento in cui*metodo2()* tenta di leggere l'elemento sulla cima dell'array, l'applicazione produce una eccezione di tipo *java.lang.NullPointerException*.
Dopo essere stata compilata ed eseguita, l'applicazione visualizzerà 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.

9.4 Eccezioni controllate ed eccezioni incontrollate

Le eccezioni Java sono suddivise in due categorie distinte: le eccezioni controllate o *"checked exceptions"* e quelle incontrollate o *"unchecked exceptions"*. Le prime, rappresentano la maggior parte delle eccezioni a livello applicativo (comprese quelle definite dall'utente) e hanno bisogno di essere gestite esplicitamente (da qui la dicitura *"controllate"*). Le eccezioni di questo tipo:

1. *Possono essere definite dal programmatore;*
2. *Devono essere allocate mediante operatore **new**;*
3. *Hanno la necessità di essere esplicitamente propagate;*
4. *Richiedono di essere esplicitamente gestite dal programmatore.*

Ricordando la sequenza di apertura e lettura da un file, schematizzata nella *Figura 70*, una eccezione controllata potrebbe ad esempio rappresentare un errore durante il tentativo di apertura del file, a causa di un inserimento errato del nome.
Le eccezioni derivate dalla classe *java.lang.RuntimeException* (*Figura 73*), appartengono invece alla seconda categoria di eccezioni. Le eccezioni incontrollate, sono generate automaticamente dalla Java Virtual Machine e sono relative a tutti quegli errori di programmazione che, tipicamente, non sono controllati dal programmatore a livello applicativo: memoria insufficiente, riferimento ad oggetti nulli, accesso errato al contenuto di un array ecc. ecc.
Un'eccezione incontrollata di tipo *java.lang.ArrayIndexOutOfBoundException* sarà automaticamente generata se, tentassimo di inserire un elemento all'interno di un array, utilizzando un indice maggiore di quelli consentiti dalle dimensioni dell'oggetto.

```
package src.esercizi.eccezioni;

public class ArrayOutOfBounds {
	public static void main(String[] argv) {
		byte[] elenco = new byte[20];
		//Indici consentiti 0..19
		elenco[20]=5;
	}
}
```

```
java src.esercizi.eccezioni.ArrayOutOfBounds

java.lang.ArrayIndexOutOfBoundsException
	at src.esercizi.eccezioni.ArrayOutOfBounds.main(ArrayOutOfBounds.java:8)
Exception in thread "main"
```

Le eccezioni appartenenti alla seconda categoria, per loro natura sono difficilmente gestibili, non sempre possono essere catturate e gestite, causano spesso la terminazione anomala dell'applicazione.
Java dispone di un gran numero di eccezioni predefinite di tipo incontrollato, in grado di descrivere le principali condizioni di errore a livello di codice sorgente. Esaminiamo le più comuni.
L'eccezione *java.lang.NullPointerException* è sicuramente la più comune tra queste, ed è generata tutte le volte che l'applicazione tenti di fare uso di un oggetto nullo. In particolare sono cinque le condizioni che possono causare la propagazione di quest'oggetto:

1. *Effettuare una chiamata ad un metodo di un oggetto nullo;*
2. *Accedere o modificare un dato membro pubblico di un oggetto nullo;*

3. *Richiedere la lunghezza di un array nullo;*
4. *Accedere o modificare i campi di un array nullo;*
5. *Propagare un'eccezione nulla (ovvero non allocata mediante operatore new).*

Di fatto, questo tipo di eccezione viene generata automaticamente ogni volta si tenti di effettuare un accesso illegale ad un oggetto **null**.
L'eccezione *java.lang.ArrayIndexOutOfBoundException*, già introdotta in questo paragrafo, rappresenta invece un specializzazione della classe base *java.lang.IndexOutOfBoundException* (*Figura 74*), utilizzata per controllare tutte le situazioni in cui si tenti di utilizzare indici errati per accedere a dati contenuti in strutture dati ordinate.

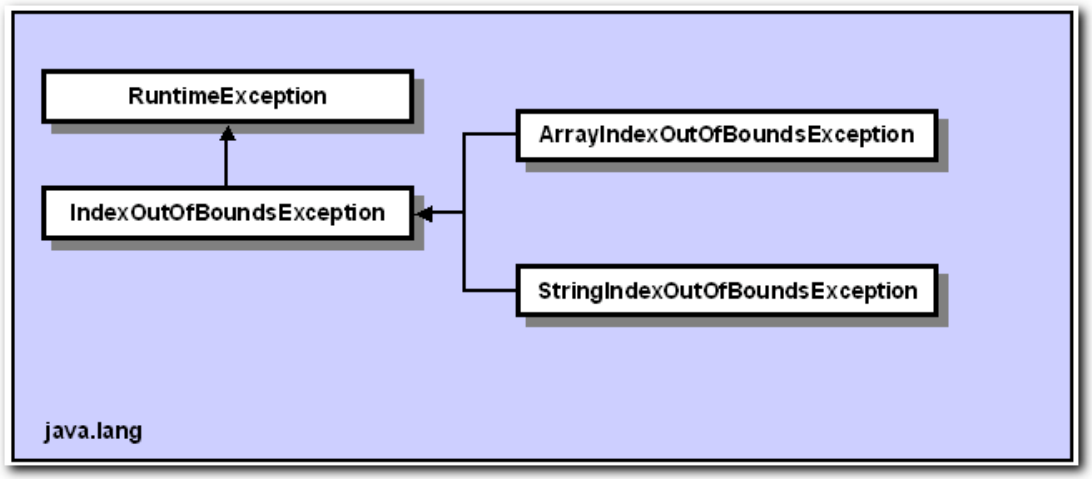


Figura 74: Sottoclassi di IndexOutOfBoundException

La seconda le sottoclasse di *java.lang.IndexOutOfBoundException* è la classe *java.lang.StringIndexOutOfBoundException*, generata da alcuni metodi dell'oggetto *String*, per indicare che un indice è minore di zero oppure maggiore o uguale alla dimensione della stringa rappresentata (una stringa in Java è rappresentata mediante un array di byte).

```
package src.esercizi.eccezioni;

public class StringOutOfBounds {
        public static void main(String[] argv) {
                String nome = "massimiliano";
                System.out.println("La lunghezza di nome e': "+nome.length());
                //La prossima operazione genera una eccezione
                System.out.println("Il carattere "+nome.length()+" e': "
                        +nome.charAt(nome.length()));
   }
}
```

```
java src.esercizi.eccezioni.StringOutOfBounds

La lunghezza di nome e': 12
java.lang.StringIndexOutOfBoundsException: String index out of range: 12
        at java.lang.String.charAt(String.java:516)
        at src.esercizi.eccezioni.StringOutOfBounds.main(StringOutOfBounds.java:10)
Exception in thread "main"
```

Infine, la classe *java.lang.ArithmeticException*, è generata tutte le volte che si tenti di effetuare una operazione aritmetica non consentita, come ad esempio la divisione di un numero per zer.

```
package src.esercizi.eccezioni;

public class ArithmeticErroCondition {
        public static void main(String[] argv) {
                int i = 100;
                int j = 0;
                int risultato = i/j;
        }
}
```

```
java src.esercizi.eccezioni.ArithmeticErroCondition

java.lang.ArithmeticException: / by zero
        at src.esercizi.eccezioni.ArithmeticErroCondition.main(ArithmeticErroCondition.java:9)
Exception in thread "main"
```

9.5 Errori Java

Oltre alle eccezioni di primo e secondo tipo, il package *java.lang* appartenente alle Java Core API contiene la definizione di un altro tipo particolare di classi chiamate "errori". Queste classi, definite anche loro per ereditarietà dalla superclasse *java.lang.Throwable*, rappresentano errori gravi della Java Virtual Machine che causano l'interruzione anomala dell'applicazione. A causa della loro natura, questo tipo di eccezioni non sono mai catturate per essere gestite: se un errore accade, la JVM stampa un messaggio di errore su terminale ed esce.
La gerarchia di queste classi è schematizzata in parte nella prossima immagine:

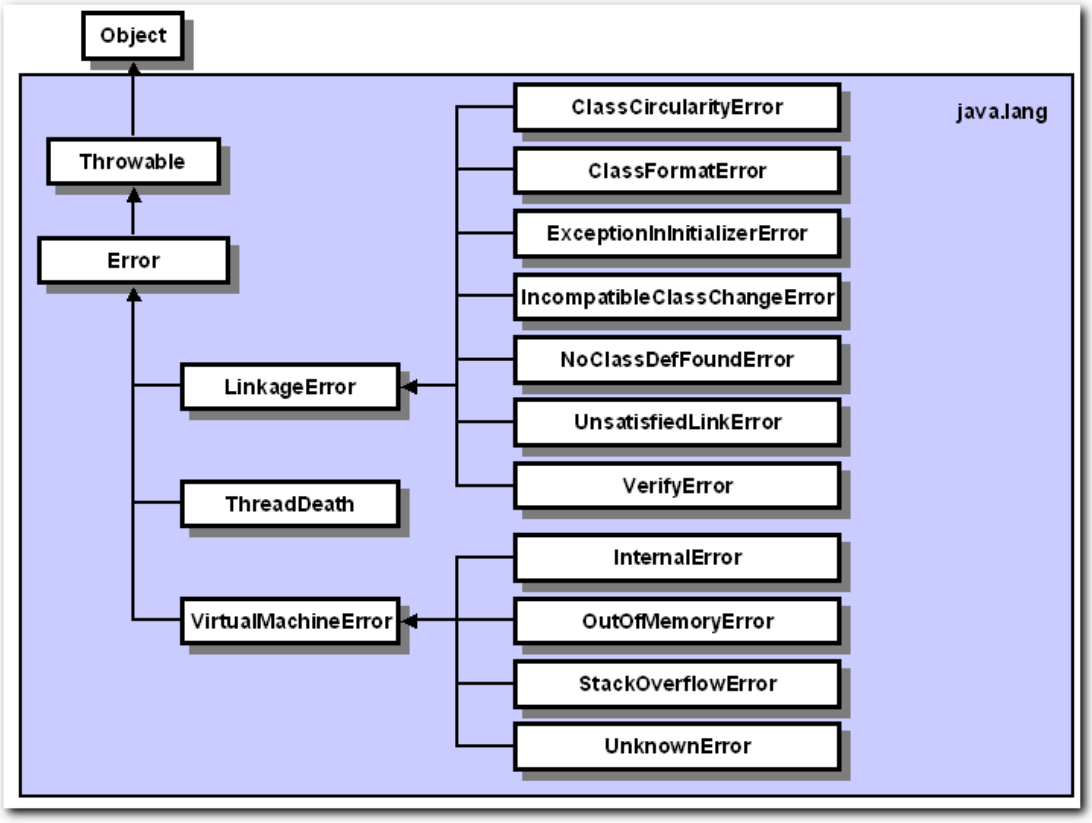


Figura 75: Java Errors

Un errore molto comune, soprattutto per chi sviluppa applicazioni Java da poco tempo, è quello definito dalla class *java.lang.NoClassDefFoundError*. Questa eccezione, è generata dal "*classloader*" della Java Virtual Machine quando l'applicazione richiede di caricare un oggetto, la cui definizione non è indicata all'interno della variabile d'ambiente CLASSPATH.

9.6 Definire eccezioni personalizzate

Quando definiamo nuovi oggetti, è spesso desiderabile disegnare nuovi tipi di eccezioni che li accompagnino.
Come specificato nei paragrafi precedenti, un nuovo tipo di eccezione deve essere derivata da *java.lang.Exception*, ed appartiene alla prima categoria di eccezioni: quelle controllate. Il funzionamento interno della nuova eccezione non è ristretto da nessuna limitazione.
Ad esempio, potremmo creare una eccezione di tipo *StackOutOfBoundException*, per seganalare che la Pila definita in precedenza, ha raggiunto la capienza massima.

```
package src.esercizi.eccezioni;

/**
```

```
 * Questa eccezione è generata dalla classe Pila
 * quando l'array raggiunge la massima capienza e
 * si tenta di inserire nuovi elementi all'interno.
 */
public class StackOutOfBoundException extends Exception {
	/**
	 * Questo costruttore permette di inserire
	 * un messaggio di errore personalizzato.
	 * @param String s : messaggio di errore
	 */
	public StackOutOfBoundException(String s) {
		super(s);
	}

	/**
	 * Questo costruttore deve essere utilizzato se,
	 * si vuole utilizzare un messaggio di errore
	 * comune a tutte le eccezioni di questo tipo.
	 */
	public StackOutOfBoundException() {
		super("StackOutOfDataException: la pila ha raggiunto la capienza massima");
	}

}
```

La nuova classe ha due costruttori :

```
public StackOutOfBoundException ()
public StackOutOfBoundException (String s)
```

Il primo, non accetta argomenti, ed inizializza il messaggio di errore trasportato dalla eccezione con un valore standard; il secondo, accetta come argomento una stringa che rappresenta il messaggio di errore da trasportare. Poiché una eccezione è un oggetto come altri, potremmo includere all'interno qualsiasi altro tipo di informazione che ritenessimo necessario trasportare. Ad esempio, se volessimo trasportare la dimensione massima della Pila all'interno della eccezione, potremmo modificare la classe nel modo seguente:

```
package src.esercizi.eccezioni;

/**
 * Questa eccezione è generata dalla classe Pila
 * quando l'array raggiunge la massima capienza e
 * si tenta di inserire nuovi elementi all'interno.
 */
public class StackOutOfBoundException extends Exception {

	int capienza = 0;

	/**
	 * Questo costruttore permette di inserire
	 * un messaggio di errore personalizzato.
	 * @param String s : messaggio di errore
	 */
	public StackOutOfBoundException(String s) {
```

```
		super(s);
	}

	/**
	* Questo costruttore permette di inserire
	* un messaggio di errore personalizzato e
	* di impostare il valore la capienza massima della pila
	* @param String s : messaggio di errore
	* @param int capienza : dimensione massima della pila
	*/
	public StackOutOfBoundException(String s, int capienza) {
		super(s);
		this.capienza=capienza;
	}

	/**
	* Questo costruttore deve essere utilizzato se,
	* si vuole utilizzare un messaggio di errore
	* comune a tutte le eccezioni di questo tipo.
	*/
	public StackOutOfBoundException() {
		super("StackOutOfDataException: la pila ha raggiunto la capienza massima");
	}

	/**
	* Questo costruttore permette di assegnare
	* un valore alla capienza massima della pila
	* @param int capienza : dimensione massima della pila
	*/
	public StackOutOfBoundException(int capienza) {
		super("StackOutOfDataException: la pila ha raggiunto la capienza massima");
		this.capienza = capienza;
	}

	/**
	* Restituisce il valore impostato per la capienza
	* massima della Pila. SE non esplicitamente
	* impostato, il valore restituito è zero.
	* @return int : capienza massima della pila.
	*/
	public int capienza() {
		return capienza;
	}

}
```

9.7 L'istruzione throw

La definizione di un oggetto di tipo *Throwable*, non è sufficiente a completare il meccanismo di propagazione dell'oggetto. Nei paragrafi precedenti abbiamo affermato che, la propagazione di un'eccezione controllata, deve essere gestita esplicitamente dall'origine della propagazione dell'oggetto, fino alla cattura e successiva gestione del medesimo.
Le eccezioni, vengono propagate a ritroso attraverso la sequenza dei metodi chiamanti tramite l'istruzione **throw,** che ha sintassi:

throw *Oggetto ;*

dove *Oggetto* è una istanza valida dell'oggetto Throwable. E' importante tener presente che *Oggetto* rappresenta un oggetto valido creato mediante l'operatore **new** e non semplicemente un tipo di dato.
L'istruzione **throw,** 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 riprendere il metodo terminato senza effettuare una nuova chiamata. Anche in questo caso, il metodo non riprenderà dal ounto in cui è stato interrotto.

9.8 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 dichiarare che un metodo ha la capacità di causare eccezioni, 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:

tipo nome (argomenti) ***throws*** *tipo_Throwable{[,tipo_Throwable]}*
{
Corpo del metodo
}

Un metodo con una clausola **throws**, può generare una eccezione del tipo dichiarato da *tipo_Throwable* oppure, ogni tipo derivato da esso.
Se un metodo contenente una clausola **throws** viene ridefinito (overrided) attraverso l'ereditarietà, il nuovo metodo può scegliere se contenere o no la clausola **throws**. Nel caso in cui scelga di contenerla, sarà costretto a dichiarare lo stesso tipo del metodo originale o, al massimo, un tipo derivato.
Analizziamo nuovamente nei dettagli il metodo membro *push(**int**)* della classe Pila:

```
/**
* Il metodo push ritorna un tipo void e prende come
* parametro un numero intero da inserire sulla cima della pila
* @return void
* @param int dato : elemento da inserire sulla cima della pila
*/
void push(int dato)
{
        if(cima < dati.length)
        {
```

```
                dati[cima] = dato;
                cima ++;
        }
}
```

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, non è 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 affinché generi un'eccezione quando si tenti di inserire un elemento nella pila piena. Per far questo, utilizziamo l'eccezione definita nei paragrafi precedenti:

```
/**
* Inserisce un elemento sulla cima della pila
* @param int dato : dato da inserire sulla cima della struttura dati
* @return void
* @exception StackOutOfBoundException
*/
void push(int dato) throws StackOutOfBoundException {
        if (cima < dati.length) {
                dati[cima] = dato;
                cima++;
        }
        else {
                throw new StackOutOfBoundException("La pila ha raggiunto la dimensione massima.
                Il valore inserito è andato perduto +[" + dato + "]", dati.length);
        }
}
```

La nuova versione della classe Pila, genererà una condizione di errore, segnalando alla applicazione l'anomalia, ed evitando che dati importanti vadano perduti.

9.9 Istruzioni try / catch

A questo punto siamo in grado di generare e propagare un'eccezione. Consideriamo però la prossima applicazione:

```
package src.esercizi.eccezioni;

public class StackOutOfBound {
        public static void main(String[] argv) {
                int capienza = 10;
                Pila stack = new Pila(capienza);
                for(int i=0; i<=capienza; i++)
                {
                        System.out.println("Inserisco "+i+" nella pila");
                        stack.push(i);
                        System.out.println("Il dato è stato inserito");
                }
```

```
        }
}
```

Se provassimo a compilare la nuova classe, il risultato sarebbe il seguente:

```
D:\PROGETTI\JavaMattone\src\src\esercizi\eccezioni\StackOutOfBound.java:12: unreported exception
src.esercizi.eccezioni.StackOutOfBoundException; must be caught or declared to be thrown
            stack.push(i);
                ^
1 error
*** Compiler reported errors
```

Il compilatore allerta il programmatore avvertendo che, l'eccezione eventualmente propagata dal metodo *push* della classe Pila, deve essere obbligatoriamente catturata e gestita.
Una volta generata un'eccezione, l'applicazione è destinata alla terminazione salvo che l'oggetto propagato sia catturato prima di raggiungere il metodo main del programma o, direttamente al suo interno.
Di fatto, Java obbliga il programmatore a catturare un'eccezione al più all'interno del metodo main di un'applicazione. Questo compito spetta all'istruzione **catch**.
Questa istruzione fa parte di un insieme di istruzioni dette "**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 un'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 {
        istruzione;
}
catch (Exception nome) {
        istruzioni
}
catch (Exception nome) {
        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 un'istruzione nel blocco **try** genera un'eccezione, le rimanenti istruzioni nel blocco non sono eseguite. L'esecuzione di un blocco **catch** esclude automaticamente tutti gli altri.

Immaginiamo di avere definito una classe con tre metodi: *f1()*, *f2()* e *f3()*, e supponiamo che i primi due metodi generano rispettivamente un'eccezione di tipo *IOException* ed un'eccezione di tipo *NullPointerException*.
Lo pseudo codice seguente, rappresenta un esempio di blocco di guardia, responsabile di intercettare e poi 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, l'esecuzione del blocco di guardia (**try**) passerebbe il controllo della esecuzione blocco catch che dichiara di gestire l'eccezione generata da *f1()*:

```
catch (IOException _e) {
        System.out.println(_e.toString())
}
```

Se invece è il metodo *f2()* a propagare l'eccezione, *f1()* termina correttamente, *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**.
Per concludere, una versione corretta della applicazione *StackOutOfBound* potrebbe essere la seguente:

```
package src.esercizi.eccezioni;

public class StackOutOfBound {
        public static void main(String[] argv) {
                int capienza = 10;
                Pila stack = new Pila(capienza);
                for(int i=0; i<=capienza; i++)
                {
                        System.out.println("Inserisco "+i+" nella pila");
                        try {
```

```
                                stack.push(i);
                                System.out.println("Il dato è stato inserito");
                        }
                        catch (StackOutOfBoundException _error) {
                                System.out.println(_error);
                        }
                }
        }
}
```

Il risultato della esecuzione della applicazione sarebbe il seguente:

```
.....
Il dato è stato inserito
Inserisco 5 nella pila
Il dato è stato inserito
Inserisco 6 nella pila
Il dato è stato inserito
Inserisco 7 nella pila
Il dato è stato inserito
Inserisco 8 nella pila
Il dato è stato inserito
Inserisco 9 nella pila
Il dato è stato inserito
Inserisco 10 nella pila
src.esercizi.eccezioni.StackOutOfBoundException: La pila ha raggiunto la dimensione massima. Il valore inserito è andato perduto +[10]
```

9.10 Singoli catch per eccezioni multiple

Differenziare i blocchi **catch**, affinché gestiscano ognuno particolari condizioni di errore identificate dal tipo dell'eccezione propagata, consente di poter specializzare il codice affinché possa prendere decisioni adeguate per la soluzione del problema verificatosi.
Esistono dei casi in cui è 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.
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 un'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*.
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
}
```

Un tipo base con un istruzione catch, può essere utilizzato per implementare un meccanismo di gestione dell'errore che, specializza le istruzioni quando necessario o, utilizza blocchi di codice comuni a tutte le eccezioni che non richiedano una gestione particolare. 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 di guardia, 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 particolarmodo il funzionamento del blocco identificato dalla **default**.

9.11 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 di istruzioni, offre la possibilità di eseguire sempre un certo insieme di istruzioni a prescindere da come i blocchi guardiani catturano e gestiscono le condizioni di errore.

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 delle istruzioni contenute 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  {
        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.
Per concludere, la sintassi completa per il blocco guardiano diventa quindi:

```
try {
        istruzione;
}
catch (Exception nome) {
        istruzioni
}
catch (Exception nome) {
        istruzioni
}
.....
finally {
        istruzioni
}
```

10 POLIMORFISMO ED EREDITARIETÀ AVANZATA

10.1 Introduzione

L'ereditarietà rappresenta uno strumento di programmazione molto potente; d'altra parte il semplice modello di ereditarietà presentato nell'ottavo capitolo 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 di un modello di ereditarietà singola è che, non prevede l'utilizzo di una classe base come modello puramente concettuale, ossia priva dell'implementazione delle funzioni base. Se facciamo un passo indietro, ricordiamo che abbiamo definito una Pila (pila) come un contenitore all'interno del quale inserire dati da recuperare secondo il criterio "primo ad entrare, ultimo ad uscire". Potrebbe esserci però un'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 regole da utilizzare per manipolare lo Stack sono le stesse, quello che cambia sono i tipi di dato contenuti.
Anche se utilizzassimo la classe Pila, riportata di seguito, 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.

```
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.1
 */

class Pila {
        int[] dati;
        int cima;

        /**
        * Il metodo push ritorna un tipo void e prende come
        * parametro un numero intero da inserire sulla cima della pila
        * @return void
        * @param int dato : elemento da inserire sulla cima della pila
        */
         public void push(int dato) {
                if(cima < dati.length)
                {
                        dati[cima] = dato;
                        cima ++;
                }

        }
```

```
        /**
        * il metodo pop non accetta parametri e restituisce
        * l'elemento sulla cima della Pila
        * @return int : dato sulla cima della pila
        */
        int pop()
        {
                if(cima > 0)
                {
                        cima--;
                        return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }
}
```

Un altro problema che non è risolto dal modello ad 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: **interfacce** e **classi astratte**. Le interfacce, sono entità simili a classi, ma non contengono implementazioni delle funzionalità descritte. Le classi astratte, anch'esse simili a classi normali, consentono di implementare solo parte delle 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 dell'ereditarietà.

10.2 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 può essere riassunto nell'espressione **"un'interfaccia, molti metodi".** Ciò significa che, possiamo definire un'interfaccia unica da utilizzare in molti casi collegati logicamente tra loro.
Oltre a risolvere i limiti del modello di ereditarietà, Java realizza il polimorfismo per mezzo delle interfacce.

10.3 Interfacce

Formalmente, un'interfaccia Java rappresenta un prototipo e consente al programmatore di definire lo scheletro di una classe: nomi dei metodi, tipi

ritornati, lista degli argomenti. 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).
Le interfacce, sono molto utili per definire gruppi di classi aventi un insieme minimo di metodi in comune, senza fornirne però una implementazione comune. Ad esempio, se volessimo generalizzare la definizione di Pila, potremmo definire un'interfaccia contenente i prototipi dei metodi *push* e *pop* comuni a tutti gli oggetti di questo tipo. Poiché, il tipo di dato gestito da questi oggetti dipende dall'implementazione della pila, ogni classe costruita da quest'interfaccia, avrà la responsabilità di gestire in modo appropriato i propri dati. Quello che un'interfaccia non consente, è, infatti, l'implementazione del corpo dei metodi.
Di fatto, un'interfaccia stabilisce il protocollo di una classe, senza preoccuparsi dei dettagli di implementazione.

10.4 Definizione di un'interfaccia

Per consentire al programmatore la dichiarazione di interfacce, Java dispone della parola chiave **interface**. La sintassi necessaria a definire una interfaccia è la seguente:

*[**public** | **abstract**]**interface** identificatore [**extends** tipo]{*
corpo_dell_interfaccia
}

Gli elementi necessari a dichiarare questo tipo particolare di classi, sono quindi elencati di seguito:

1. *I modificatori opzionali public o private per definire gli attributi della classe;*
2. *La parola chiave **interface**;*
3. *Un nome che identifica la classe;*
4. *Opzionalmente, la clausola extends se l'interfaccia è definita a partire da una classe base;*
5. *Le dichiarazioni dei dati membro della classe, e dei prototipi dei metodi.*

Proviamo, ed esempio, a definire un'interfaccia che ci consenta di generalizzare la definizione della classe Pila. Per poterlo fare, è necessario ricordare che l'oggetto Object, in quanto padre di tutte le classi Java, è compatibile con qualsiasi tipo definito o, definibile. Questa caratteristica è alla base di ogni generalizzazione in Java, in quanto, ogni oggetto può essere ricondotto ad Object e viceversa, da Object possono essere derivati tutti gli

altri oggetti. L'interfaccia generalizzata di un oggetto generico Pila è quindi la seguente:

```
package src.esercizi.eccezioni;

public interface Stack {
        /**
        * Inserisce un elemento sulla cima della pila
        * @param int dato : dato da inserire sulla cima della struttura dati
        * @return void
        * @exception <{StackOutOfBoundException}>
        */
        void push(Object dato) throws StackOutOfBoundException;

        /**
        * Ritorna il primo elemento della pila
        * @return int : primo elemento sulla pila
        */
        Object pop();
}
```

Consentitemi ora una breve digressione. Dall'analisi della definizione dell'interfaccia *Stack*, potrebbe a questo punto sorgere il seguente dubbio: avendo utilizzato un oggetto per definire i prototipi dei metodi della classe, come facciamo a rappresentare pile i cui elementi sono rappresentati mediante tipi primitivi (**int**, **long**, **double** ecc. ecc.)?
La risposta è immediata. Di fatto, nonostante Java disponga di dati di tipo primitivo, essendo un linguaggio orientato ad oggetti, deve poter rappresentare anche i tipi primitivi in forma di oggetto.
Nella prossima figura, è stato riportato parte del contenuto del package *java.lang* appartenente alle Java Core API.

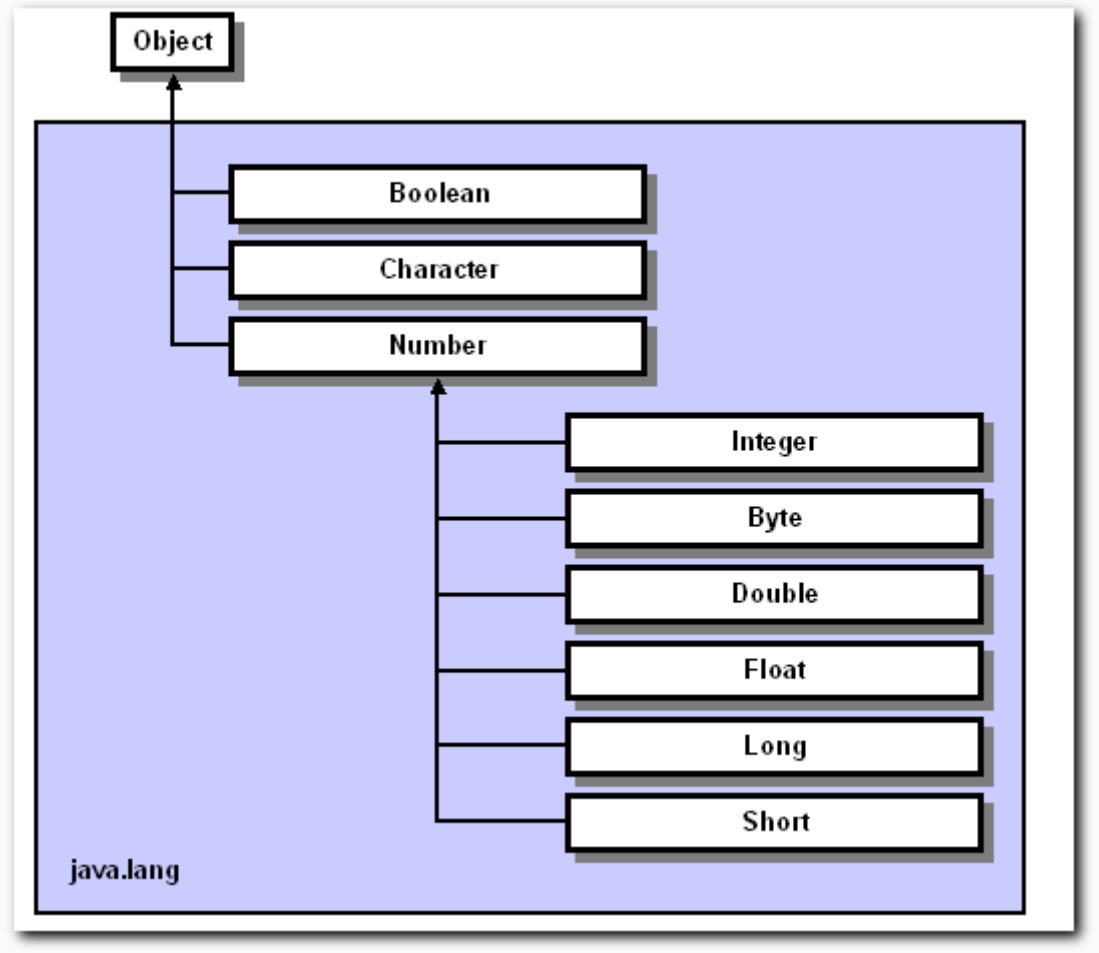


Figura 76: Tipi primitivi come oggetti

La Pila di interi, potrebbe ad esempio essere sostituita da una Pila di oggetti di tipo *java.lang.Integer*.

10.5 Implementare una interfaccia

Poiché un'interfaccia rappresenta solo il prototipo di una classe, affinché possa essere utilizzata è necessario che ne esista un'implementazione che rappresenti una classe allocabile.
Per implementare un'interfaccia, Java mette a disposizione la parola chiave **implements** con sintassi:

class *nome* ***implements*** *interfaccia{*
corpo_della_classe
}

La nostra classe Stack potrà quindi essere definita da un modello nel modo seguente:

```
package src.esercizi.eccezioni.stack;
```

```
import src.esercizi.eccezioni.StackOutOfBoundException;

/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.5
 */
public class PilaDiInteri implements Stack {
	/** Contenitore per i dati della Pila */
	Integer[] dati;

	/** Punta al primo elmento corrente nella struttura dati */
	int cima;

	/** Costruisce una pila contenente al massimo 10 elementi */
	public PilaDiInteri() {
		dati = new Integer[20];
		cima = 0;
	}

	/**
	* Costruisce una Pila, consentendo di impostarne la dimensione massima
	* @param int dimensione : dimensione massima della pila
	*/
	public PilaDiInteri(int dimensione) {
		dati = new Integer[dimensione];
		cima = 0;
	}

	/**
	* Inserisce un elemento sulla cima della pila
	* @param int dato : dato da inserire sulla cima della struttura dati
	* @return void
	* @exception <{StackOutOfBoundException}>
	*/
	public void push(Object dato) throws StackOutOfBoundException {
		if (cima < dati.length) {
			dati[cima] = (Integer)dato;
			cima++;
		}
		else {
			throw new StackOutOfBoundException("La pila ha raggiunto la dimensione
			massima. Il valore inserito è andato perduto +[" +		dato + "]",
			dati.length);
		}
	}

	/**
	* Ritorna il primo elemento della pila
	* @return int : primo elemento sulla pila
	*/
	public Object pop() {
		if (cima > 0) {
			cima--;
			return dati[cima];
		}
		return null; // Bisogna tornare qualcosa
	}
}
```

In un *class-diagram*, questo rapporto tra classi e interfacce, è rappresentato in modo analogo alla relazione di ereditarietà tra classi Java. La relazione tra l'interfaccia Stack e la classe PilaDiInteri in un *class-diagram* è mostrata nella figura 77.

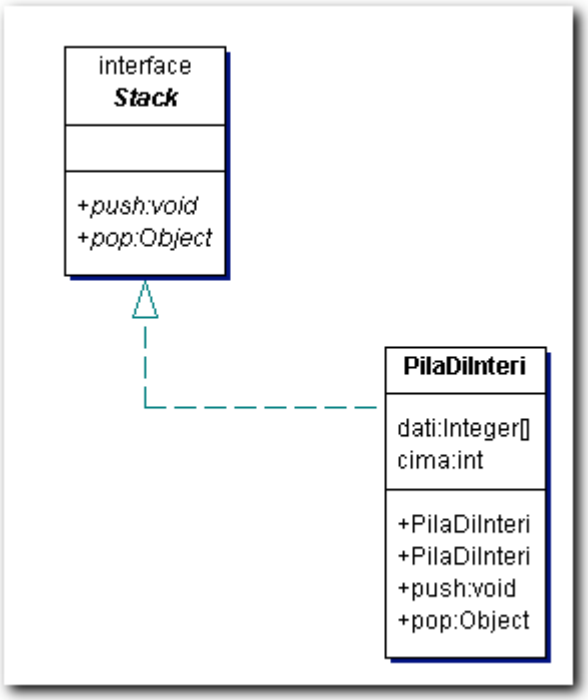


Figura 77: Relazione di ereditarietà Classe-Interfaccia

Quando una classe implementa un'interfaccia, è obbligata a fornirne un'implementazione di tutti i prototipi dei metodi. In caso contrario, il compilatore genererà un messaggio di errore. Di fatto, possiamo pensare ad un'interfaccia come ad una specie di contratto che l'ambiente di Java stipula con una classe. Implementando un'interfaccia, la classe, non si limita a definire un concetto da un modello logico (molto utile al momento del disegno dell'applicazione), ma assicurerà l'implementazione di almeno i metodi definiti nell'interfaccia.
La relazione che intercorre tra un'interfaccia ed una classe Java, è anch'essa una forma di ereditarietà (da qui la similitudine tra le due simbologie UML). Se l'interfaccia dovesse contenere definizioni di dati membro, le stesse saranno ereditate dalla classe costruita da essa mediante la clausola **implements**.
Conseguenza diretta di quest'affermazione, è la possibilità di utilizzare le interfacce come tipi per definire variabili reference in grado di far riferimento ad oggetti definiti mediante implementazione di un'interfaccia.

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

Varranno in questo caso tutte le regole già discusse nel capitolo settimo parlando di variabili reference ed ereditarietà.

10.6 Ereditarietà multipla in Java

Se l'operatore **extends** limitava la derivazione di una classe da una sola classe base, l'operatore **implements** ci consente di implementare una classe da quante interfacce desideriamo, semplicemente elencando le interfacce da implementare, separate tra loro con una virgola.

class *nome* ***implements*** *interfaccia{[, interfaccia]} {*
    corpo_della_classe
}

Questa caratteristica permette al programmatore di creare gerarchie di classi molto complesse in cui, una classe eredita la natura concettuale di molte entità. Se una classe implementa interfacce multiple, dovrà fornire tutte le funzionalità per i metodi definiti in tutte le interfacce.

10.7 Classi astratte

Capitano casi in cui l'astrazione offerta dalle interfacce, eccede rispetto alle necessità del programmatore (le interfacce 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 normali classi base e rispettano le definizioni fornite per la compatibilità tra classi; tuttavia, queste classi non sono complete e come le interfacce, non possono essere allocate direttamente.
Per estendere le classi astratte, si utilizza la clausola **extends** e di conseguenza, solo una classe astratta può essere utilizzata per creare nuove definizioni di classe.
Questo meccanismo fornisce la scappatoia alla costrizione imposta dalle interfacce, di doverne implementare tutte le funzionalità all'interno di un'eventuale nuova definizione di classe, aumentando la flessibilità del linguaggio, 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*
{
    attributi
    metodi_astratti

metodi_non_astratti
}

Ogni metodo che rappresenta semplicemente un prototipo, deve essere dichiarato **abstract,** utilizzando la sintassi seguente:

*[**private**|**public**] **abstract** tipo_di_ritorno nome(argomento [,argomento])*
{
istruzione
[istruzione]
}

Quando una classe deriva da una classe base astratta, il compilatore richiede che tutti i metodi astratti siano definiti. Se la necessità del momento costringe a non definire questi metodi, la nuova classe dovrà a sua volta essere definita abstract.
Un esempio d'uso del modificatore **abstract** potrebbe essere rappresentato da una classe che rappresenta un terminale generico. Un terminale ha alcune funzionalità comuni quali lo spostamento di un cursore, la stampa di una stringa, consente di spostare il cursore da un punto ad un altro oppure di pulire l'area visibile da ogni carattere. Le proprietà di un terminale sono sicuramente: il numero di righe e colonne da cui è composto e la posizione corrente del cursore.
Ovviamente, tra le funzionalità di un terminale, quelle indipendenti dalle specifiche di basso livello del sistema potranno essere implementate già a livello di oggetto generico, le altre si potranno implementare solo all'interno di classi specializzate.
Nel prossimo esempio, viene mostrata una possibile definizione generica di un terminale. Utilizzando le classi astratte, sarà possibile definire i prototipi di tutti i metodi che dovranno essere implementati all'interno di classi specializzate, fornendo però la definizione completa di quelle funzionalità di più alto livello, comuni a tutti i terminali.

```
package src.esercizi.polimorfismo;

/**
 * Definizione generica di un terminale a video.
 */
public abstract class Terminale {
```

```
	private int rigacorrente, colonnacorrente, numrighe, numcolonne;

	/**
	* Crea un terminale le cui dimensioni sono specificate
	* dal numero massimo di righe e di colonne visibili.
	* @return void
	* @param int rows : numero massimo di righe
	* @param int col : numero massimo di colonne
	*/
	public Terminale(int rows, int cols) {
		numrighe = rows;
		numcolonne = cols;
		rigacorrente = 0;
		colonnacorrente = 0;
	}

	/**
	* Sposta il cursone alle coordinate precisate
	* @return void
	* @param int rows : indice della riga
	* @param int col : indice della colonna
	*/
	public abstract void spostaIlCursore(int rows, int col);

	/**
	* Inserisce una stringa di caratteri alle coordinate spcificate
	* @return void
	* @param int rows : indice della riga
	* @param int col : indice della colonna
	* @param String buffer : sequenza di caratteri da stampare a terminale
	*/
	public abstract void inserisci(int rows, int col, String buffer);

	/**
	* IPulisce il terminale
	* @return void
	*/
	public void pulisci() {
		int r, c;
		for (r = 0; r < numrighe; r++)
			for (c = 0; c < numcolonne; c++) {
				spostaIlCursore(r, c);
				inserisci(r, c, " ");
			}
	}
}
```

11 JAVA THREADS

11.1 Introduzione

Java è un linguaggio multithread, cosa che sta a significare che:

"un programma può essere eseguito logicamente in molti luoghi nello stesso momento".

Il multithreading, consente di creare applicazioni in grado di utilizzare la concorrenza logica tra i processi, con il vantaggio di poter consentire ai thread di continuare a condividere lo spazio di memoria riservato ai dati.

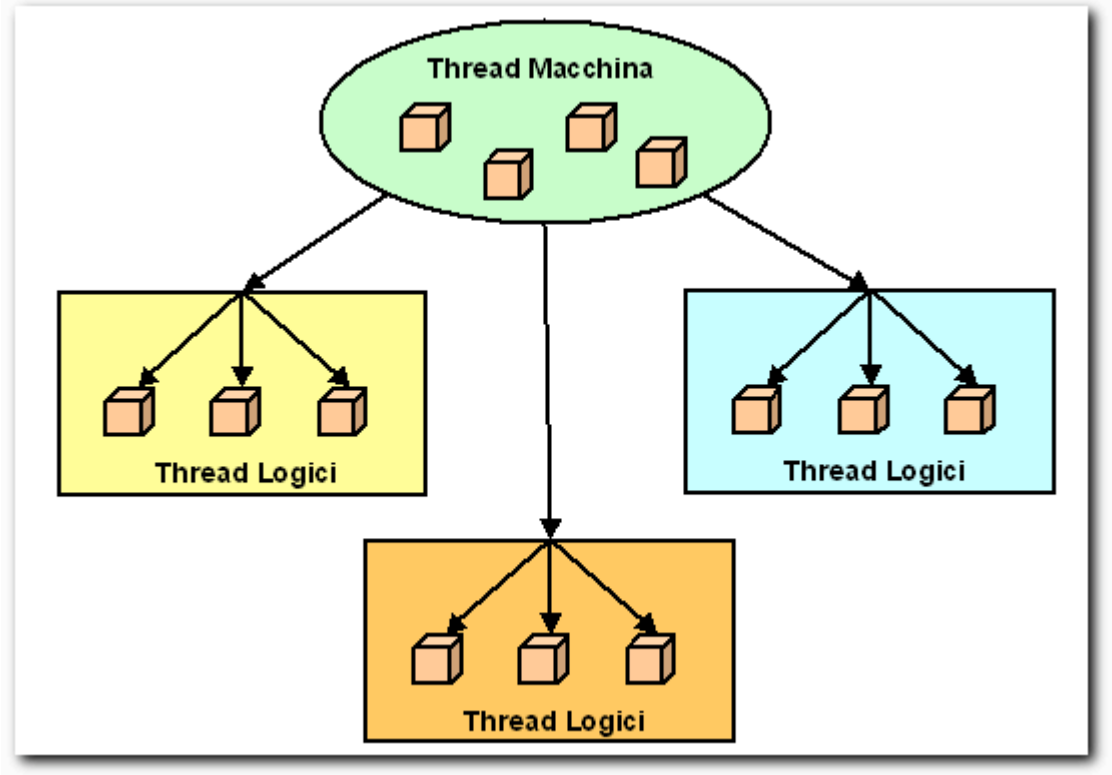


Figura 78: Thread Macchina e Thread Logici

Nel diagramma nella *Figura 79*, 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 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 del processore su cui l'applicazione è in esecuzione.

11.2 Thread e processi

Obiettivo di questo capitolo è descrivere come programmare applicazioni multithread utilizzando il linguaggio Java. Prima di procedere, è importante capire a fondo il significato di programmazione concorrente, come scrivere applicazioni utilizzando questa tecnica, ed infine la differenza tra thread e processi.

"Cara, vai comprare le aspirine in farmacia, mentre misuro la febbre al bambino"

Nonostante possa sembrare bizzarro, quello che per noi può sembrare un comportamento banale, dal punto di vista di un calcolatore, rappresenta un problema gravoso. Eseguire contemporaneamente, due azioni logicamente correlate, comporta uno sforzo non indifferente dovendo gestire problematiche complesse quali:

1. *La condivisione dei dati tra le entità che dovranno concorrere a risolvere il problema;*
2. *L'accesso concorrente alle risorse condivise del sistema (stampanti, supporti per i dati, periferiche, porte di comunicazione);*

Le tecniche di programmazione più comuni, fanno uso di "processi multipli" o thread.
Un processo, è un'applicazione completa con un proprio spazio di memoria. Molti sistemi operativi, consentono l'esecuzione contemporanea di tanti processi, supportando la condivisione o la comunicazione di dati mediante rete, memoria condivisa, pipe, ecc...
Il sistema operativo UNIX ad esempio, dispone della funzione *fork()* che, consente ad un processo di sdoppiarsi in un processo figlio (*Figura 79*).

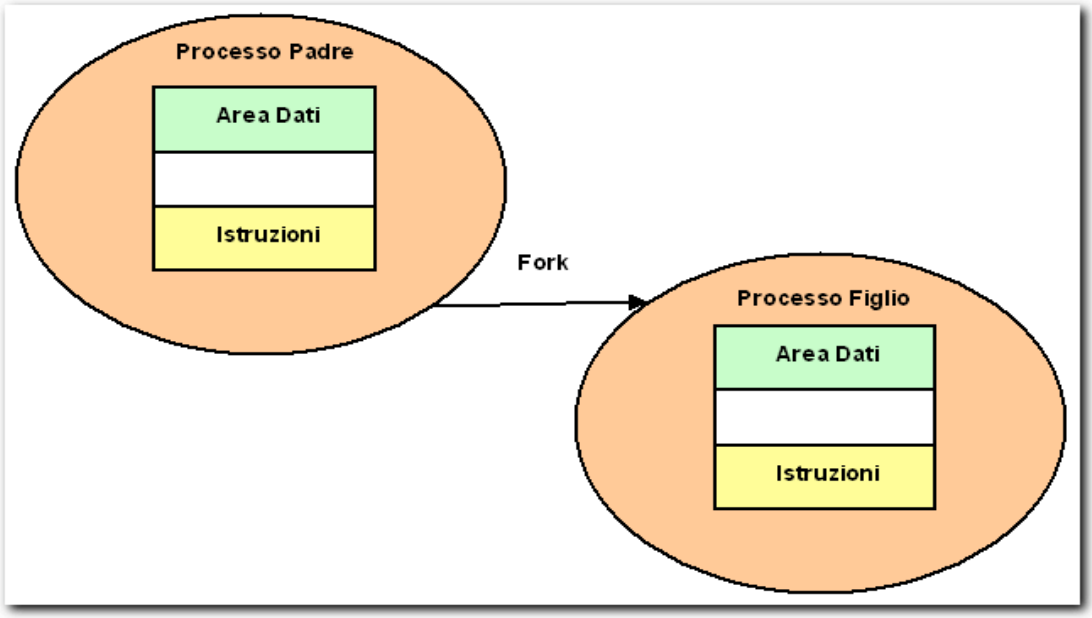


Figura 79: Fork UNIX

Il risultato finale saranno due processi distinti, ognuno con una propria area dati.

I thread, conosciuti anche come "ligth-weight processes" o "processi leggeri" (LWP), vivono all'interno di uno stesso processo con il quale condividono istruzioni ed area dati (*Figura 80*).

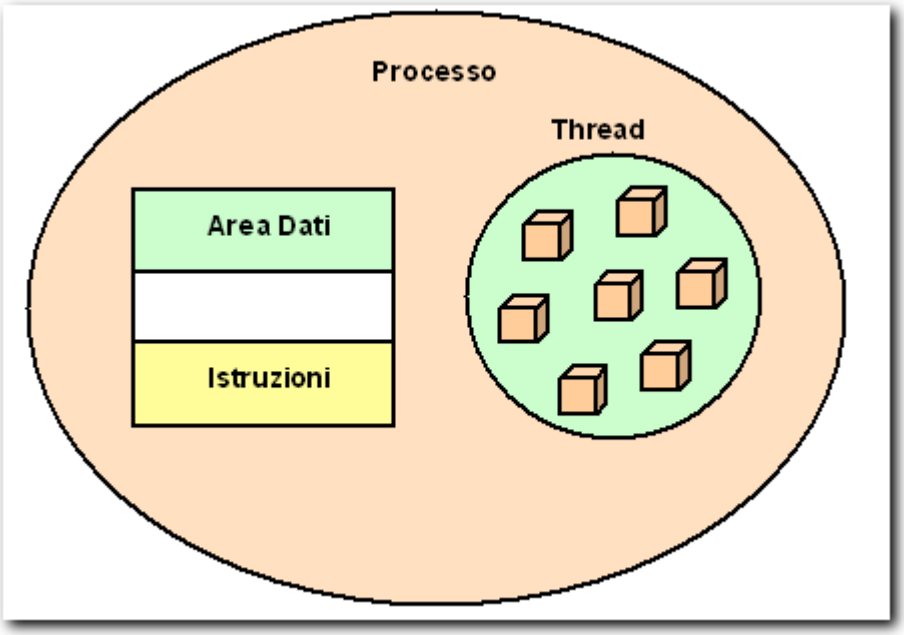


Figura 80: Thread

I thread, in definitiva, consentono di eseguire più parti di codice di una stessa applicazione, nello stesso istante.

11.3 Vantaggi e svantaggi nell'uso di thread

La scelta tra la prima e la seconda soluzione proposte nel paragrafo precedente, non è così scontata come può sembrare. Di fatto, i thread non rappresentano sempre la tattica migliore nell'approccio ad un problema di programmazione in concorrenza.
In questo paragrafo, cercheremo di comprendere quali sono i vantaggi e gli svantaggi che l'uso di thread comporta rispetto a quello di processi multipli.

1. *I thread sono molto efficienti nell'utilizzo della memoria.*

L'uso di processi multipli, necessita di una quantità di memoria maggiore rispetto a quella richiesta dai thread. Di fatto, ogni processo per essere eseguito ha bisogno di una propria area dati ed istruzioni. I thread, differentemente dai processi, utilizzano la stessa area dati ed istruzioni del processo che li ha generati.

2. *I thread comunicano mediante memoria condivisa.*

Poiché i thread condividono la stessa area di memoria, la comunicazione tra loro può avvenire utilizzando semplicemente puntatori a messaggi. La comunicazione tra processi implica invece la trasmissione di un messaggio da un'applicazione ad un'altra. La comunicazione tra processi di conseguenza, può essere causa della duplicazione di grandi quantità di dati.

3. *La concorrenza tra thread è molto efficiente.*

Processi attivi e thread concorrono tra loro all'utilizzo del processore. Poiché il contesto di un thread è sicuramente minore di quello di un processo, è minore anche la quantità di dati da salvare sullo stack di sistema. L'alternanza tra thread risulta quindi più efficiente di quella tra processi.

4. *I thread sono parte del codice della applicazione che li genera.*

Per aggiungere nuovi thread ad un processo, è necessario modificare e compilare tutto il codice della applicazione con tutti i rischi che comporta questa operazione. Viceversa, i processi sono entità dinamiche indipendenti tra loro. Aggiungere un processo significa semplicemente eseguire una nuova applicazione.

5. *I thread possono accedere liberamente ai dati utilizzati da altri thread sotto il controllo del medesimo processo.*

Condividere la stessa area di memoria con il processo padre e con gli altri thread figli, significa avere la possibilità di modificare dati vitali per gli altri thread o, nel caso peggiore per il processo stesso. Viceversa, un processo non ha accesso alla memoria riservata da un altro.
Concludendo, i thread sono più efficienti dei processi e consentono di sviluppare facilmente applicazioni complesse e molto efficienti. Il costo di quest'efficienza è però alto poiché, è spesso necessario implementare complesse procedure per la gestione dell'accesso concorrente ai dati condivisi con gli altri thread.

11.4 La classe java.lang.Thread

Il primo metodo per creare un thread è estendere la classe base *java.lang.Thread* mediante la direttiva **extends** (*figura 81*). La classe *java.lang.Thread*, non è una classe astratta e incapsula il codice necessario al funzionamento del thread : avvio, esecuzione, interruzione.
Tra i metodi definiti, il metodo

public void *run()*

deve essere utilizzato per implementare le operazioni eseguite dal thread riscrivendone la definizione nella nuova classe mediante il meccanismo di overriding. In genere, *run()* è l'unico metodo su cui effettuare overriding. Nel caso in cui il metodo non sia riscritto, al momento dell'avvio del thread, eseguirà la funzione nulla.

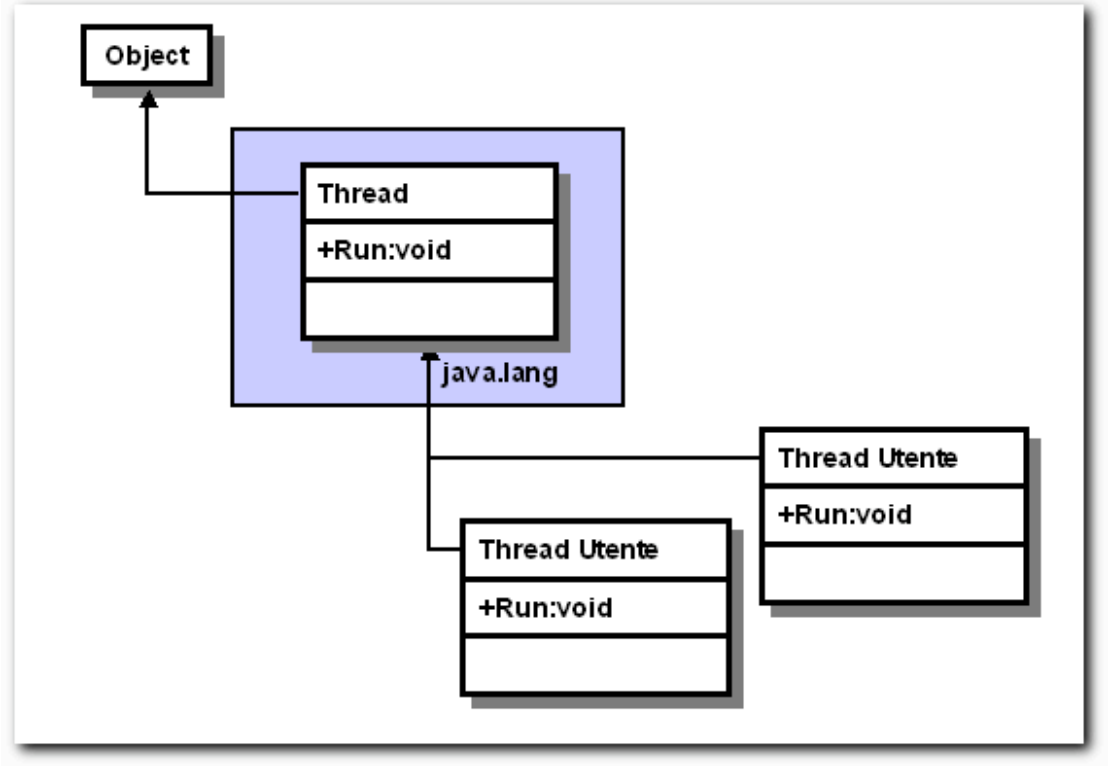


Figura 81: La classe java.lang.Thread

La classe java.lang.Thread è fornita dei metodi necessari alla esecuzione, gestione e interruzione di un thread. I tre principali:
Un esempio di thread definito per ereditarietà dalla classe Thread è il seguente:

```
package src.esercizi.thread;

public class MioPrimoThread extends Thread {
	public void run() {
		try {
			int i=0;
			while(true)
			{
				sleep(1000);
				i++;
				 System.out.println("Il valore di i: "+i);
			}
		}
		catch(InterruptedException _ie) {
			System.out.println(_ie);
		}
	}
}
```

In quest'esempio abbiamo definito una classe che, estende *Thread* e ne ridefinisce il metodo *run()*. Il metodo *run()*, contiene un ciclo infinito che, ad

ogni iterata si addormenta per un secondo e successivamente aggiorna il valore di una variabile intera visualizzandolo sul terminale.
Di fatto, tutti i thread contengono cicli infiniti poiché, un thread cessa la propria esecuzione nel momento in cui metodo *run()* termina ed esce.
Nel prossimo esempio, cessiamo l'esecuzione del thread provocando l'interruzione del ciclo mediante l'istruzione **break**, quando la variabile intera assume il valore dieci.

```
package src.esercizi.thread;

public class MioPrimoThread extends Thread {
	public void run() {
		try {
			int i=0;
			while(true)
			{
				sleep(1000);
				i++;
				 System.out.println("Il valore di i: "+i);
				//Se i vale 10, il thread viene terminato
				//provocando l'interruzione del ciclo infinito.
				if(i==10) break;
			}
		}
		catch(InterruptedException _ie) {
			System.out.println(_ie);
		}
	}
}
```

E' importante notare che ogni thread, una volta entrato all'interno del ciclo infinito, ogni tanto deve necessariamente addormentarsi per qualche istante. Utilizzando il metodo *sleep(**long**)* ereditato dalla superclasse, è possibile ottenere l'effetto desiderato congelando il thread per il tempo specificato dall'argomento del metodo espresso in millisecondi. In caso contrario, il thread consumerà tutto il tempo di cpu impedendo ad altri thread o applicazioni di proseguire nell'esecuzione.

11.5 L'interfaccia "Runnable"

L'ereditarietà singola di Java impedisce ad una classe di avere più di una classe padre, sebbene possa implementare un numero arbitrario di interfacce. In tutte le sitazioni in cui non sia possibile derivare una classe da *java.lang.Thread*, può essere utilizzata l'interfaccia *java.lang.Runnable*, come schematizzato in *figura 82*.
L'interfaccia Runnable contiene il prototipo di un solo metodo:

```
public interface Runnable
{
	public void run();
```

```
}
```

necessario ad indicare il metodo che dovrà contenere il codice del nuovo thread (esattamente come definito nel paragrafo precedente). La classe *MioPrimoThread* potrebbe essere riscritta come mostrato nel prossimo esempio.

```
package src.esercizi.thread;

public class MioPrimoThread implements Runnable {
	public void run() {
		try {
			int i=0;
			while(true)
			{
				Thread.sleep(1000);
				i++;
				 System.out.println("Il valore di i: "+i);
			}
		}
		catch(InterruptedException _ie) {
			System.out.println(_ie);
		}
	}
}
```

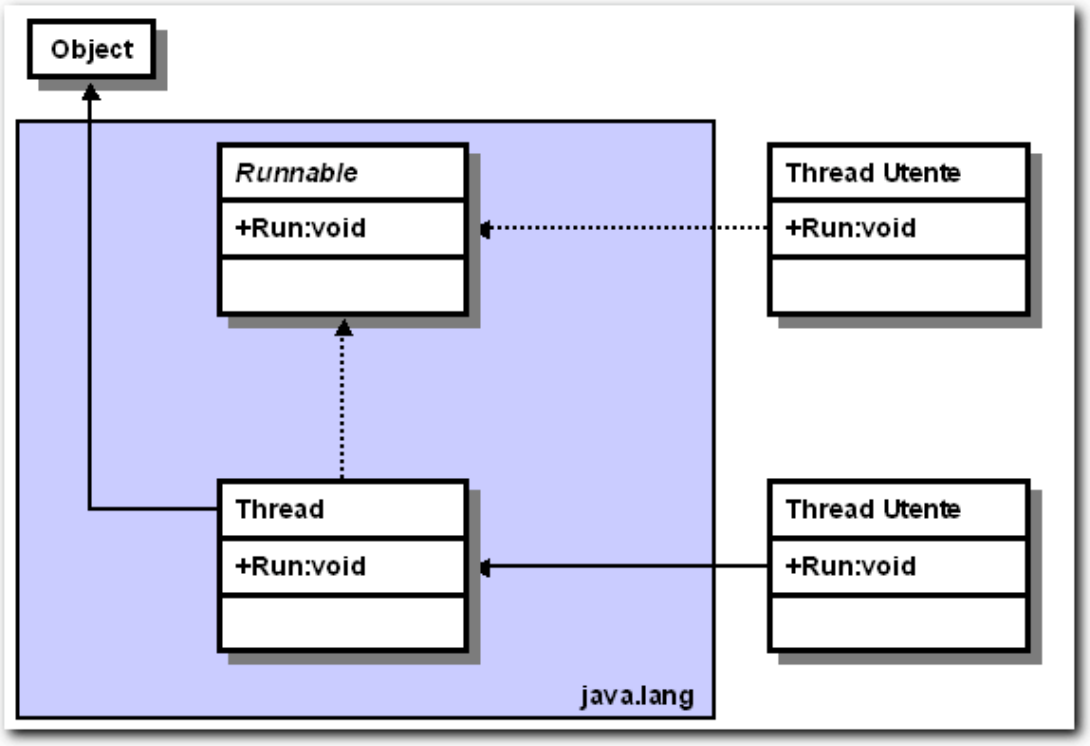


Figura 82: L'interfaccia Runnable

In generale, tutti i thread sono creati utilizzando l'interfaccia *Runnable* (anche la classe *java.lang.Thread* implementa quest'interfaccia come schematizzato nella *figura 82*), e questo è dovuto alla flessibilità offerta dalle interfacce rispetto alle classi, nel disegno delle gerarchie di ereditarietà. D'altra parte, un'interfaccia rappresenta solo un modello. Ricordiamo, infatti, che un'interfaccia può contenere solo metodi astratti e variabili di tipo **static final**, di conseguenza un interfaccia non può essere utilizzata per creare un oggetto, in altre parole, non potremmo scrivere:

*Runnable mioThread = **new** Runnable()*

In definitive quindi, implementare la classe *Runnable* non è sufficiente a creare un oggetto thread, semplicemente, abbiamo definito una classe che assomiglia ad un thread.
Nel paragrafo precedente, abbiamo detto che, molto del lavoro di un thread è svolto dalla classe *java.lang.Thread* che contiene la definizione dei metodi necessari a gestirne il ciclo di vita completo.
Per creare un thread da un oggetto di tipo *Runnable* possiamo utilizzare la classe *java.lang.Thread* dotata di un costruttore che accetta come argomento un tipo Runnable. Nel prossimo esempio utilizzando le due versioni della classe *MioPrimoThread* creeremo due applicazioni uguali che, creano un Thread partendo rispettivamente da un tipo *Thread* e da un tipo *Runnable*.

```
package src.esercizi.thread;

public class MioPrimoThread extends Thread {
	public void run() {
		try {
			int i=0;
			while(true)
			{
				Thread.sleep(1000);
				i++;
				 System.out.println("Il valore di i: "+i);
			}
		}
		catch(InterruptedException _ie) {
			System.out.println(_ie);
		}
	}

	public static void main(String[] args)
	{
		//Crea un thread semplicemente creando un oggetto di tipo MioPrimoThread
		MioPrimoThread miothread = new MioPrimoThread();
	}
}
```

Il metodo main di questa prima versione della applicazione, crea un oggetto thread semplicemente creando un oggetto di tipo *MioPrimoThread*.

```
package src.esercizi.thread;

public class MioPrimoThread implements Runnable {
	public void run() {
		try {
			int i=0;
			while(true)
			{
				sleep(1000);
				i++;
				 System.out.println("Il valore di i: "+i);
			}
		}
		catch(InterruptedException _ie) {
			System.out.println(_ie);
		}
	}

	public static void main(String[] args)
	{
		//Crea un thread creando un oggetto di tipo Thread
		//utilizzando il costruttore che accetta un oggetto Runnable
		MioPrimoThread t= new MioPrimoThread();
		Thread miothread = new Thread(t);
	}
}
```

Il metodo main della seconda versione della applicazione, crea un oggetto thread creando: prima un oggetto di tipo *MioPrimoThread*, poi un oggetto di tipo Thread passando il primo oggetto come argomento del metodo costruttore del secondo.

11.6 Ciclo di vita di un Thread

Creare un thread non è sufficiente poiché, è necessario che sia in esecuzione il metodo *run()* affinché il thread possa ritenersi attivo. La responsabilità di avviare l'esecuzione del thread è affidata al metodo pubblico *start()* della classe Thread, il cui funzionamento è schematizzato nel *sequence-diagram* in *figura 83*.

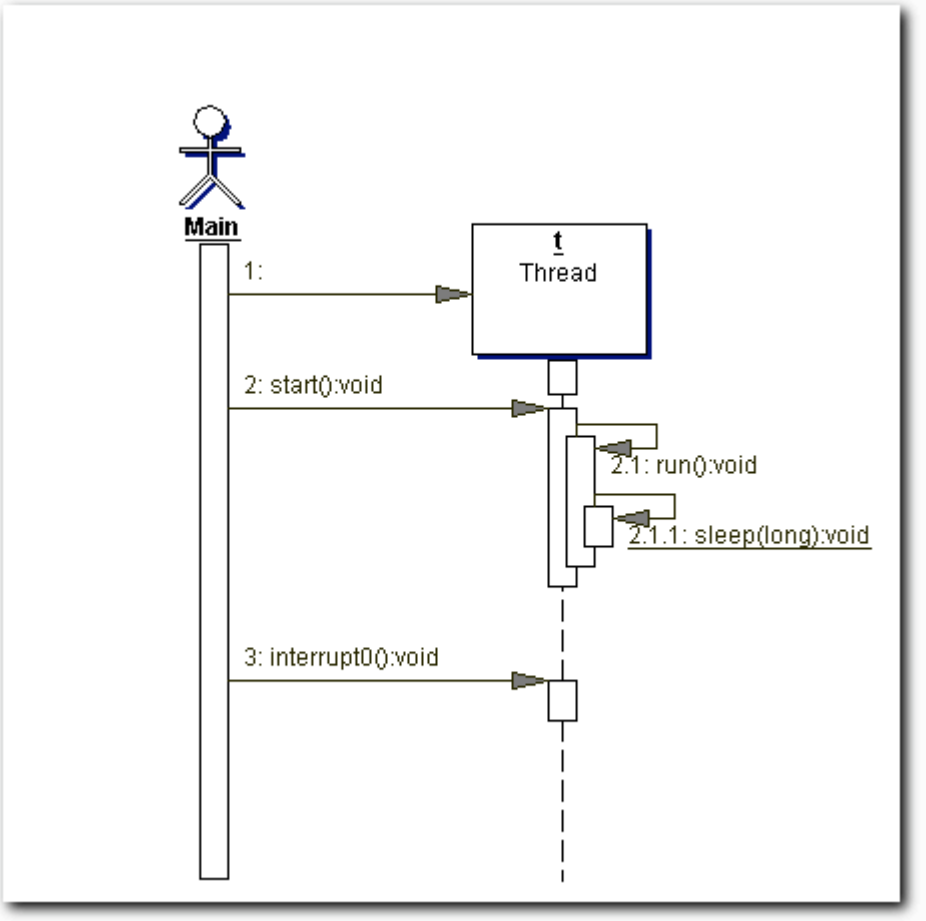


Figura 83: Ciclo di vita di un thread

Una volta in esecuzione, un thread può essere fermato utilizzando in metodo *interrupt()*. Quando un thread è fermato mediante chiamata al metodo *interrupt()*, non può in nessun modo essere riavviato mediante il metodo *start()*. Nel caso in cui sia necessario sospendere e poi riprendere l'esecuzione del thread, è necessario utilizzare i due metodi *suspend()* e *resume()* che oppure, come già detto in precedenza, possiamo utilizzare il metodo *sleep(**long**)*.
Il metodo *sleep(long)* consente di interrompere l'esecuzione di un thread per un periodo di tempo determinato. I due metodi *suspend()* e *resume()*, il cui funzionamento è descritto nel sequence-diagram in *figura 84*, rispettivamente: interrompe l'esecuzione del thread a tempo indeterminato, riprende l'esecuzione di un thread, interrotta dal metodo *suspend()*.

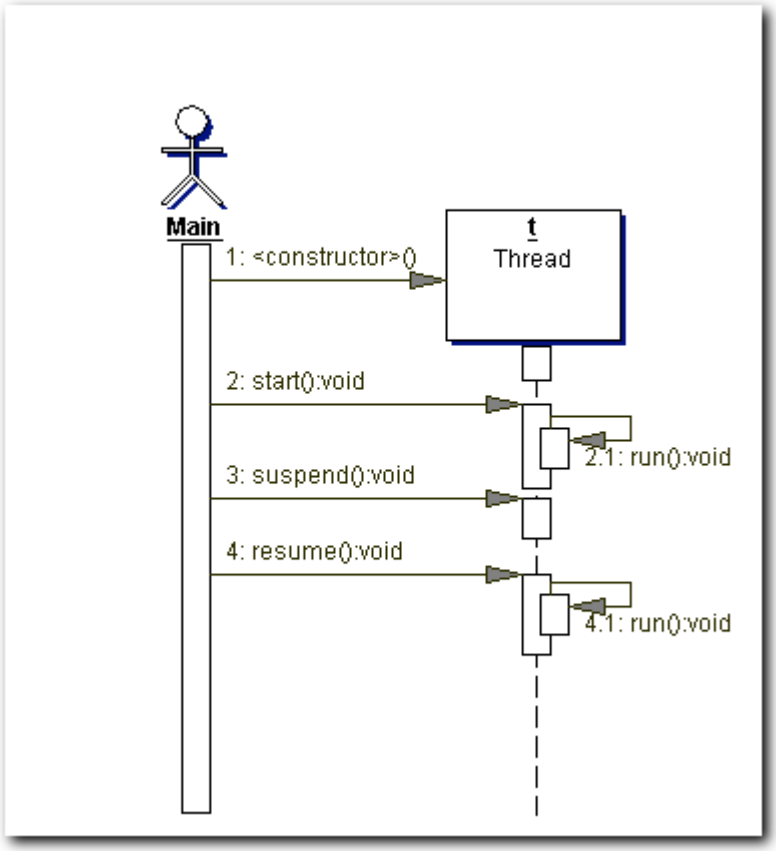


Figura 84: Sospendere e riprendere l'esecuzione

11.7 L'applicazione Orologio

Per fissare i concetti introdotti nei paragrafi precedenti, sviluppiamo una breve applicazione di esempio che implementa un orologio di sistema un po' particolare, il cui pannello principale è mostrato nella prossima figura.

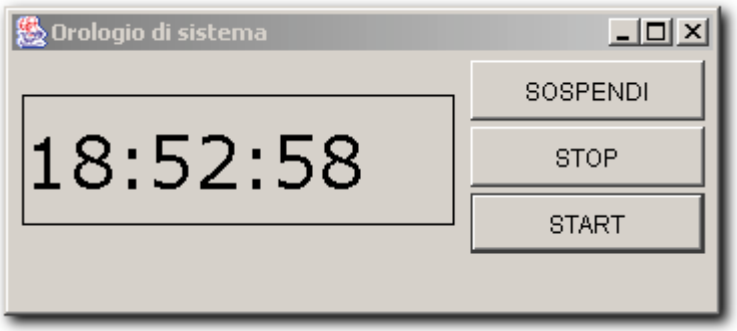


Figura 85: Pannello principale di orologio

All'avvio della applicazione, viene mostrato un pannello contenente i tre pulsanti sulla destra del pannello e l'area contenente l'ora di sistema vuota. Premendo il pulsante 'start' (*Figura 86*), l'applicazione inizia a visualizzare l'ora di sistema aggiornandone il valore ogni secondo.

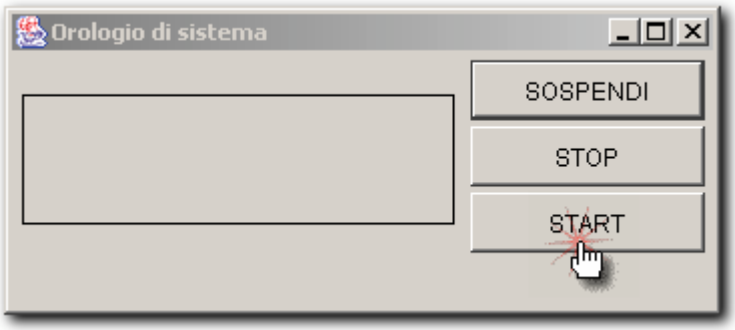


Figura 86: Avvio della applicazione orologio

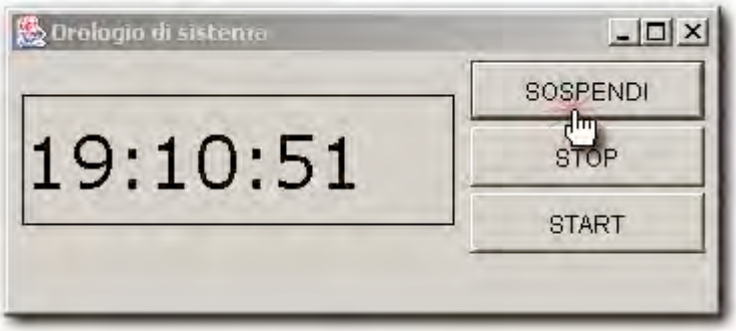


Figura 87: Interruzione dell'esecuzione

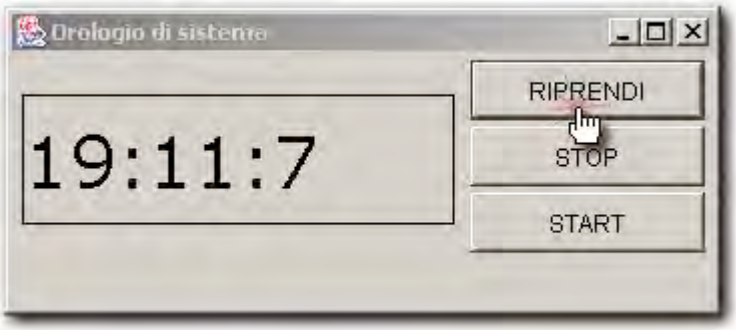


Figura 88: Ripresa della esecuzione

Premendo il pulsante 'sospendi', è possibile sospendere, e successivamente riprendere, l'aggiornamento dell'ora nel pannello della applicazione (*figura 87,88*). Il pulsante 'stop' interrompe definitivamente l'esecuzione dell'applicazione. In nessun modo sarà possibile riavviarne l'esecuzione.
L'applicazione è formata dalle tre classi schematizzate nel *class-diagram* in *figura 89*.

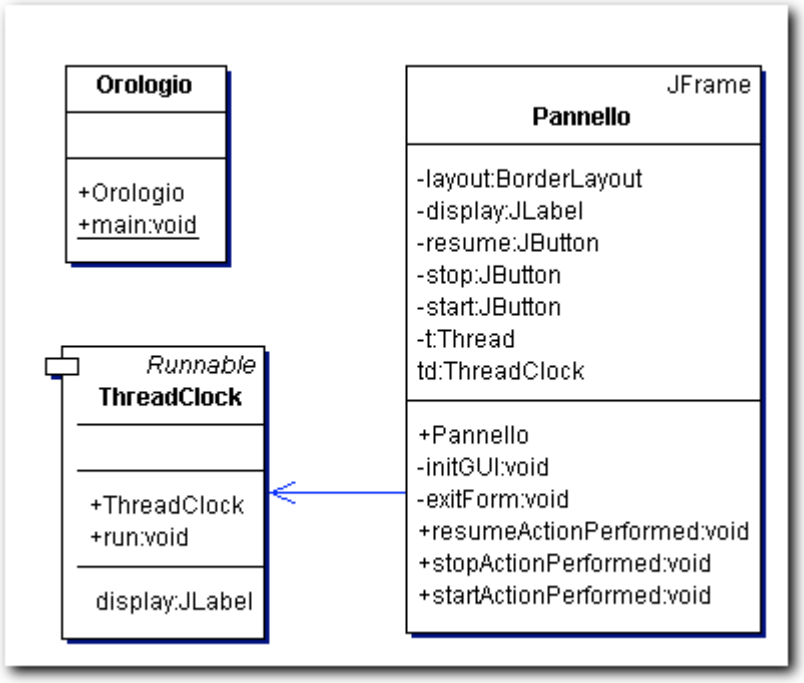


Figura 89: Class-Diagram di orologio

La classe orologio, il cui codice sorgente è riportato di seguito, contiene semplicemente il metodo *main* dell'applicazione il quale, crea il pannello principale e ne imposta le dimensioni iniziali e la posizione sullo schermo.

```
package src.esercizi.thread;

import javax.swing.UIManager;
import java.awt.Dimension;
import java.awt.Toolkit;
public class Orologio {
	public Orologio() {
		Pannello frame = new Pannello();
		//Centra il frame sullo schermo
		Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
		Dimension frameSize = frame.getSize();
		frameSize.height = ((frameSize.height > screenSize.height) ? screenSize.height :
		frameSize.height);
		frameSize.width = ((frameSize.width > screenSize.width) ? screenSize.width :
		frameSize.width);
		frame.setLocation((screenSize.width - frameSize.width) / 2, (screenSize.height –
		frameSize.height) / 2);
		frame.setVisible(true);
	}


	public static void main(String[] argv) {
		try {
			UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
		}
		catch (Exception e) {
			e.printStackTrace();
```

```
                }
                new Orologio();
        }
}
```

Il pannello principale è rappresentato dalla classe *Pannello*. Notiamo che, questa classe è stata sviluppata utilizzando le librerie *swing* di Java. Con *swing* è identificata la collezione di classi appartenenti alle Java Core API, necessarie allo sviluppo di interfacce grafiche. Non essendo obiettivo di questo libro introdurre alle librerie *swing*, rimandiamo la trattazione dell'argomento alla documentazione ufficiale della SUN Microsystem che, può essere scaricata da internet dal sito ufficiale di Java: www.javasoft.com.

```
package src.esercizi.thread;

import javax.swing.JFrame;
import java.awt.event.WindowEvent;
import java.awt.Dimension;
import java.awt.BorderLayout;
import javax.swing.JPanel;
import java.awt.Rectangle;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Pannello extends JFrame {
        /** Crea un nuovo JFrame */
        public Pannello() {
                initGUI();
                pack();
                td = new ThreadClock(display);
                t = new Thread(td, "Orologio");
        }

        /** Questo metodo è chiamato dal costruttore per inizializzare la fionestra. */
        private void initGUI() {
                start.setText("START");
                start.setBounds(new java.awt.Rectangle(229, 70, 117, 30));
                start.setActionCommand("START");
                start.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) { startActionPerformed(e); }});
                stop.setText("STOP");
                stop.setBounds(new java.awt.Rectangle(229, 37, 117, 30));
                stop.setActionCommand("SOSPENDI");
                stop.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) { stopActionPerformed(e); }});
                getContentPane().setLayout(layout);
                JPanel content = new JPanel();
                content.setPreferredSize(new java.awt.Dimension(350, 127));
                content.setLayout(null);
                content.setSize(new java.awt.Dimension(350, 150));
                content.add(display);
                content.add(resume);
```

```
                content.add(stop);
                content.add(start);
                getContentPane().add(content, BorderLayout.CENTER);
                setTitle("Orologio di sistema");
                setSize(new java.awt.Dimension(354, 150));
                addWindowListener(
                        new java.awt.event.WindowAdapter() {
                                public void windowClosing(java.awt.event.WindowEvent evt) {
                                        exitForm(evt);
                                }
                        });
                display.setText("");
                display.setBounds(new java.awt.Rectangle(5, 21, 216, 65));
                display.setFont(new java.awt.Font("Verdana", java.awt.Font.PLAIN, 36));
                display.setBorder(javax.swing.BorderFactory.createLineBorder(new java.awt.Color(0,
                        0, 0), 1));
                resume.setText("SOSPENDI");
                resume.setBounds(new java.awt.Rectangle(229, 4, 117, 30));
                resume.setActionCommand("SOSPENDI");
                resume.addActionListener(
                        new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                        resumeActionPerformed(e); }
                        });
        }

        /** Chiude l'applicazione */
        private void exitForm(WindowEvent evt) {
                System.exit(0);
        }

        public void resumeActionPerformed(ActionEvent e) {
                if (resume.getText().equals("SOSPENDI")) {
                        t.suspend();
                        resume.setText("RIPRENDI");
                }
                else {
                        t.resume();
                        resume.setText("SOSPENDI");
                }
        }

        public void stopActionPerformed(ActionEvent e) {
                t.interrupt();
        }

        public void startActionPerformed(ActionEvent e) {
                t.start();
        }

        private BorderLayout layout = new BorderLayout();
        private JLabel display = new JLabel();
        private JButton resume = new JButton();
        private JButton stop = new JButton();
        private JButton start = new JButton();
        private Thread t;
        private ThreadClock td;
}
```

Infine, la classe *ThreadClock*, implementa l'interfaccia *Runnable* e all'interno del metodo *run()* contiene le istruzioni necessarie a calcolare l'ora corrente di sistema per poi trasformarla in forma di stringa. Il metodo costruttore della classe accetta come argomento un oggetto di tipo Jlabel, che rappresenta la componente dell'interfaccia che visualizza la stringa contenente l'ora di sistema.

```
import javax.swing.JLabel;
import java.util.Date;

public class ThreadClock implements Runnable {
	public ThreadClock(JLabel pannello) {
		setDisplay(pannello);
	}

	public void run() {
		try {
			int i = 0;
			while (true) {
				Date data = new Date();
				String ora = data.getHours()+":"+
					data.getMinutes()+":"+data.getSeconds();
				getDisplay().setText(ora);
				Thread.sleep(1000);

			}
		}
		catch (InterruptedException _ie) {
			System.out.println(_ie);
		}
	}

	private JLabel getDisplay(){ return display; }
	private void setDisplay(JLabel display){ this.display = display; }
	private JLabel display;
}
```

Il metodo costruttore della classe pannello, crea un oggetto di tipo *ThreadClock* e lo utilizza come argomento del metodo costruttore di un oggetto di tipo Thread. Tramite i pulsanti forniti dall'interfaccia, è possibile interagire con il thread creato. In questo modo sarà possibile provare i vari stati in cui un thread può trovarsi durante il suo ciclo di vita. In particolare:

1. *Il pulsante 'start' avvia il thread. Viene eseguito il metodo run() di ThreadClock e l'applicazione inizia ad aggiornare l'ora di sistema. Premere questo pulsante, equivale ad eseguire il metodo start() della classe java.lang.Thread;*

2. *Il pulsante 'stop' ferma il thread e l'orologio non viene più aggiornato. Premere questo pulsante, equivale ad eseguire il metodo interrupt() della classe java.lang.Thread. Da questo momento in poi, non sarà più possibile riavviare il thread;*
3. *Il pulsante 'sospendi/riavvia' alternamente sospende e riavvia il thread utilizzando i due metodi suspend() e resume() di java.lang.Thread.*

11.8 Priorità di un thread

Nel terzo capitolo, parlando di supporto nativo per i thread, abbiamo introdotto il concetto di 'code Round Robin' per la gestione della concorrenza tra i processi che si candidano per utilizzo del processore. D'altra parte abbiamo anche affermato che i thread sono anch'essi processi e come tale concorrono tra loro all'assegnamento delle risorse della macchina.
Nel modello introdotto, tutti i processi gareggiano su uno stesso piano. Non esistono processi privilegiati e soprattutto, ogni processo utilizza le risorse della macchina per uno spazio di tempo (tecnicamente 'slice') uguale per tutti.
I moderni calcolatori utilizzano meccanismi più complessi in grado di stabilire la priorità di un processo rispetto agli altri. Un processo con priorità più alta avrà più possibilità di altri di ottenere le risorse della macchina e soprattutto, per una slice maggiore.
La classe *java.lang.Thread* dispone dei metodi

***public final void** setPriority(**int** newPriority);*
***public final int** getPriority();*

che, rispettivamente, imposta la priorità di un thread e ritorna la priorità del thread. I valori ammessi per la priorità di un thread sono compresi tra *java.lanag.Thread.MAX_PRIORITY* e *java.lanag.Thread.MIN_PRIORITY* che valgono rispettivamente 1 e 10. La priorità di base di un thread è *java.lanag.Thread.NORM_PRIORITY* che vale 5.
Nel prossimo esempio, l'applicazione *Priorita* esegue due thread identici aventi però diverse priorità. Il metodo *run()* dei thread contiene un ciclo infinito che stampa sul terminale una stringa che lo identifichi.

```
package src.esercizi.thread;

public class Priorita extends Thread {
        public Priorita(String nome) {
                setNome(nome);
        }

        public String getNome(){ return nome; }
        public void setNome(String nome){ this.nome = nome; }

        public void run() {
                while(true)
```

```
                System.out.println(nome);
        }

        public static void main(String[] argv) {
                Priorita T1 = new Priorita("Thread con priorità alta");
                Priorita T2 = new Priorita("Thread con priorità bassa");
                T1.setPriority(Thread.MAX_PRIORITY);
                T2.setPriority(Thread.MIN_PRIORITY);
                T1.start();
                T2.start();
        }

        private String nome;
}
```

Eseguendo l'applicazione, il risultato evidenzia chiaramente come, il thread T1 con priorità più alta sia privilegiato rispetto a quello con priorità più bassa.

```
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità bassa
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità bassa
.....
```

Capita spesso che thread con priorità molto alta facciano uso di risorse di sistema, necessarie al funzionamento di altri thread. In queste situazioni, i thread con minore priorità potrebbero trovarsi a dover lavorare in mancanza di risorse.
In questi casi può essere utilizzato il metodo *yeld()* affinché, il thread con priorità più alta, diminuisca temporaneamente la propria priorità consentendo ad altri thread di farne uso.
Infine, esistono dei thread particolari, con priorità molto bassa detti "thread daemons" o servizi. Questi thread, provvedono ai servizi di base di un'applicazione attivandosi solo quando la macchina è al minimo delle attività. Un'esempio di thread di questo tipo è il garbage collector della JVM. La classe *java.lanag.Thread* è dotata del metodo

***public final void** setDaemon(**boolean** on)*

che imposta il thread corrente affinchè si comporti come un servizio. Il metodo

public final boolean *isDaemon()*

può essere utilizzato per determinare lo stato del thread.

11.9 Sincronizzare thread

Finora abbiamo analizzato thread molto semplici, esistono però situazioni in cui due o più thread possono trovarsi a dover accedere ad un oggetto per modificarlo in regime di concorrenza. Il rischio cui si va incontro in questi casi, è quello dell'alterazione dello stato dell'oggetto condiviso.
Proviamo ad immaginare una classe che rappresenti il conto in banca della famiglia Tizioecaio e che il conto sia intestato ad entrambi i signori Tizioecaio. Supponiamo ora che sul conto siano depositati 600 euro.
Se i due intestatari del conto (thread) accedessero contemporaneamente all'oggetto per ritirare dei soldi, si rischierebbe una situazione simile alla seguente:

1. *Il signor Tizioecaio accede al conto chiedendo di ritirare 400 euro.*
2. *Un metodo dell'oggetto conto controlla il saldo trovando 600 euro.*
3. *La signora Tizioecaio accede al conto chiedendo di ritirare 400 euro.*
4. *Un metodo dell'oggetto conto controlla il saldo trovando 600 euro.*
5. *Sono addebitati sul conto i 400 euro del signor Tizioecaio.*
6. *Sono addebitati sul conto i 400 euro del signor Tizioecaio.*
7. *Il conto va in scoperto di 200 euro ed i due non lo sanno.*
8. *Quando i signori Tizioecaio scoprono lo scoperto chiudono il conto presso quella banca.*

La classe *ContoCorrente* potrebbe essere così definita:

```
package src.esercizi.thread;

/**
 * Rappresenta un generico conto corrente bancario.
 * Fornisce i metodi per depositare o ritirare soldi dal conto.
 */
public class ContoCorrente {
        public float saldo(){ return saldo; }

        /**
        * Ritira la cifra specificata dal conto
        * @param float ammontare : ammontare della cifra da depositare
        * @return void
        */
        public void ritira(float ammontare) {
```

```
                if(saldo()>=ammontare)
                        saldo = saldo-ammontare;
        }

        /**
        * Deposita la cifra specificata sul conto
        * @param float ammontare : ammontare della cifra da depositare
        * @return void
        */
        public void deposita(float ammontare) {
                saldo = saldo+ammontare;
        }

        private float saldo=600;
}
```

In questi casi è necessario che i due thread siano sincronizzati ovvero, mentre uno esegue l'operazione, l'altro deve rimanere in attesa. Java fornisce il metodo per gestire la sincronizzazione tra thread mediante la parola chiave synchronized.
Questo modificatore deve essere aggiunto alla dichiarazione del metodo da sincronizzare ed assicura che, solo un thread alla volta è in grado di eseguire il metodo.

Di fatto, indipendentemente dal numero di thread che tenteranno di accedere al metodo sincronizzato, solo uno potrà eseguirlo. Gli altri, saranno accodati in attesa di riceverne il controllo. Metodi di questo tipo sono detti "*Thread Safe*". La versione thread safe della classe controcorrente è di conseguenza:

```
package src.esercizi.thread;

/**
 * Rappresenta un generico conto corrente bancario.
 * Fornisce i metodi per depositare o ritirare soldi dal conto.
 */
public class ContoCorrente {
        public float saldo(){ return saldo; }

        /**
        * Ritira la cifra specificata dal conto
        * @param float ammontare : ammontare della cifra da depositare
        * @return void
        */
        public synchronized void ritira(float ammontare) {
                if(saldo()>=ammontare)
                        saldo = saldo-ammontare;
        }

        /**
        * Deposita la cifra specificata sul conto
        * @param float ammontare : ammontare della cifra da depositare
        * @return void
        */
```

```
        public synchronized void deposita(float ammontare) {
                saldo = saldo+ammontare;
        }

        private float saldo=600;
}
```

11.10 Lock su oggetto

Il meccanismo descritto nel paragrafo precedente non ha come unico 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
{
        public synchronized int a()
        {
                return b()
        }
        public synchronized int b(int n)
        {
                return a();
        }
}
```

Supponiamo ora che un thread T1 esegua il metodo b() della classe A e contemporaneamente, un secondo thread T2 effettua una chiamata al metodo a() dello stesso oggetto. 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: 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. Il risultato finale sarebbe lo stallo di entrambi i thread.
Questa situazione, detta *deadlock,* è difficilmente rilevabile e necessita di algoritmi molto complessi e poco efficienti per essere gestita o prevenuta.
Java garantisce al programmatore la certezza che casi di questo tipo non avvengono mai.
Di fatto, quando un thread entra all'interno di un metodo sincronizzato ottiene il lock sull'oggetto (non solo sul metodo). Ottenuto il lock, il thread potrà richiamare altri metodi sincronizzati dell'oggetto senza entrare in deadlock.
Ogni altro thread che proverà ad utilizzare l'oggetto in lock, si metterà in coda in attesa di essere risvegliato al momento del rilascio della risorsa da parte del thread proprietario.
Quando il primo thread termina le sue operazioni, il secondo otterrà il lock e di conseguenza l'uso privato dell'oggetto.
Formalmente, lock di questo tipo sono chiamati "*lock su oggetto*".

11.11 Lock di classe

Differenti dai "*lock su oggetto*" sono invece i "*lock su classe*". Vediamo di cosa si tratta.
Sappiamo che metodi statici accedono solo a dati membro statici e che un metodo o un attributo statico non richiedono un oggetto attivo per essere eseguiti.
Di conseguenza, un thread che effettui una chiamata ad un metodo statico sincronizzato non può ottenere il lock sull'istanza di un oggetto inesistente. D'altra parte, è necessaria una forma di prevenzione dal *deadlock* anche in questo caso.
Java prevede una seconda forma di lock: il "*lock su classe*". Quando un thread accede ad un metodo statico sincronizzato, ottiene un lock su classe, vale a dire, su tutti gli oggetti attivi del tipo utilizzato. In questo scenario, nessun thread potrà accedere a metodi statici sincronizzati di ogni oggetto 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 perché metodi sincronizzati non statici della classe in lock possono essere eseguiti durante l'esecuzione del metodo statico sincronizzato.

11.12 Produttore e Consumatore

Quando un thread deve spettare che accada una determinata condizione esterna per proseguire nell'esecuzione, il modificatore **synchronized** non è più sufficiente. Sincronizzare un metodo significa garantire che due thread non accedano contemporaneamente ad un oggetto, tuttavia, **synchronized** non prevede la possibilità di coordinare più thread mediante scambio di messaggi. Il problema del Produttore del Consumatore rappresenta un caso tipico in cui il modificatore **synchronized** non è sufficiente a rendere consistente la concorrenza.
Il Produttore produce un numero intero compreso tra 0 e 9 o lo rende disponibile al Consumatore, il quale potrà utilizzare il numero intero una ed una sola volta (*figura 90*).

Figura 90: Usecase-duagram per Produttore Consumatore

Nella nostra implementazione, produttore e consumatore sono rappresentati da due thread che condividono i dati mediante la classe *Magazzino* (*Figura 91*). Per rendere la simulazione più reale, faremo in modo che il *Produttore* si fermi per un periodo compreso tra 0 e 100 millisecondi prima di generare un nuovo elemento. Il codice delle tre definizioni di classe è il seguente.

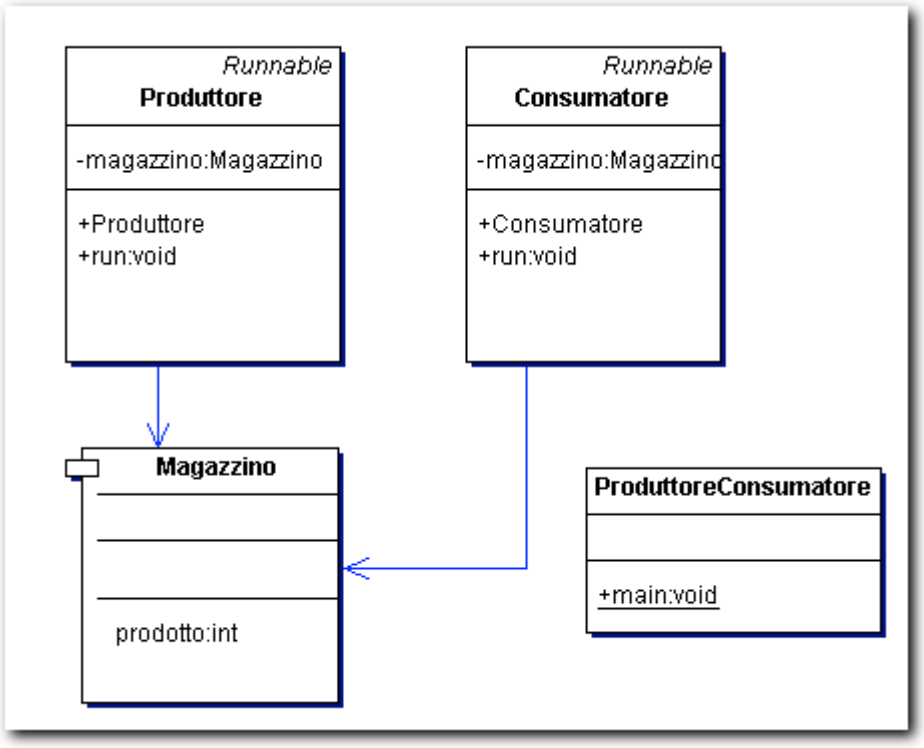


Figura 91: Class-diagram per Produttore / Consumatore

```
/**
```

```
 * CLASSE M A G A Z Z I N O
 */
package src.esercizi.thread;
public class Magazzino {
	public synchronized int getProdotto(){
		return prodotto;
	}

	public synchronized void setProdotto(int prodotto){
		this.prodotto = prodotto;
	}

	private int prodotto;
}



/**
 * CLASSE P R O D U T T O R E
 */
package src.esercizi.thread;
public class Produttore implements Runnable {
	private Magazzino magazzino;
	public Produttore(Magazzino magazzino) {
		this.magazzino = magazzino;
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			magazzino.setProdotto(i);
			System.out.println("Produttore ha inserito: " + i);
			try {
				Thread.sleep((int)(Math.random() * 100));
			} catch (InterruptedException e) { }
		}

	}
}
```

```
/**
 * CLASSE C O N S U M A T O R E
 */
package src.esercizi.thread;

public class Consumatore implements Runnable {
	private Magazzino magazzino;

	public Consumatore(Magazzino magazzino) {
		this.magazzino = magazzino;
	}

	public void run() {
		int prodotto=0;
		for (int i = 0; i < 10; i++) {
			prodotto = magazzino.getProdotto();
			System.out.println("Consumatore ha estratto: " + prodotto);
		}
	}
```

```
}
```

La classe magazzino, nonostante sia sincronizzata affinché *Produttore* e *Consumatore* non le accedano contemporaneamente, non consente di sincronizzare i due thread affinché:

1. *Il consumatore aspetti che il produttore abbia generato un nuovo numero intero;*
2. *Il produttore aspetti che il consumatore abbia utilizzato l'ultimo intero generato.*

Quanto detto sarà ancora più evidente dopo che avremo eseguito l'applicazione *ProduttoreConsumatore* il cui codice è il seguente:

```
package src.esercizi.thread;

public class ProduttoreConsumatore {
        public static void main(String[] argv) {
                Magazzino magazzino = new Magazzino();
                Thread produttore = new Thread(new Produttore(magazzino));
                Thread consumatore = new Thread(new Consumatore(magazzino));
                produttore.start();
                consumatore.start();
        }
}
```

```
java src.esercizi.thread.ProduttoreConsumatore

Produttore ha inserito: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Produttore ha inserito: 1
Produttore ha inserito: 2
Produttore ha inserito: 3
Produttore ha inserito: 4
……
```

Condizioni di questo tipo possono essere risolte solo se il *Produttore* segnala al *Consumatore* che ha generato un nuovo elemento ed è disponibile per essere utilizzato, viceversa, il *Consumatore* segnala al *Produttore* che è in attesa di un nuovo elemento da utilizzare.
La classe *Object* dispone dei metodi *wait*, *notify* e *notifyAll* che consentono di indicare ad un oggetto che deve aspettare in attesa di una condizione o, di segnalare che una condizione si è verificata.

11.13 Utilizzare i metodi wait e notify

Di seguito, una nuova versione della classe *Magazzino* i cui metodi *getProdotto* e *setProdotto* sono stati modificati utilizzando i metodi *wait()* e *notify()* ereditati dalla classe *Object*.

```
package src.esercizi.thread;

/** CLASSE M A G A Z Z I N O */
public class Magazzino {
	public synchronized int getProdotto() {
		while (isDisponibile() == false) {
			try {
				// aspetta che il produttore abbia
				// generato un nuovo elemento
				wait();
			} catch (InterruptedException e) {
			}
		}
		setDisponibile(false);
		// notifica al produttore di aver
		// utilizzato l'ultimo elemento generato
		notifyAll();
		return prodotto;
	}

	public synchronized void setProdotto(int prodotto) {
		while (isDisponibile() == true) {
			try {
				// aspetta che il consumatore abbia
				// utilizzato l'ultimo elemento
				wait();
			} catch (InterruptedException e) {
			}
		}
		this.prodotto = prodotto;
		setDisponibile(true);
		// notifica al consumatore di aver
		// ugenerato un nuovo elemento
		notifyAll();
	}

	public boolean isDisponibile() { return disponibile; }

	public void setDisponibile(boolean disponibile) { this.disponibile = disponibile; }

	private int prodotto;
	private boolean disponibile;
}
```

Le attività svolte dai metodi *getProdotto()* e *setProdotto(**int**)* sono schematizzate in *figura 92*.

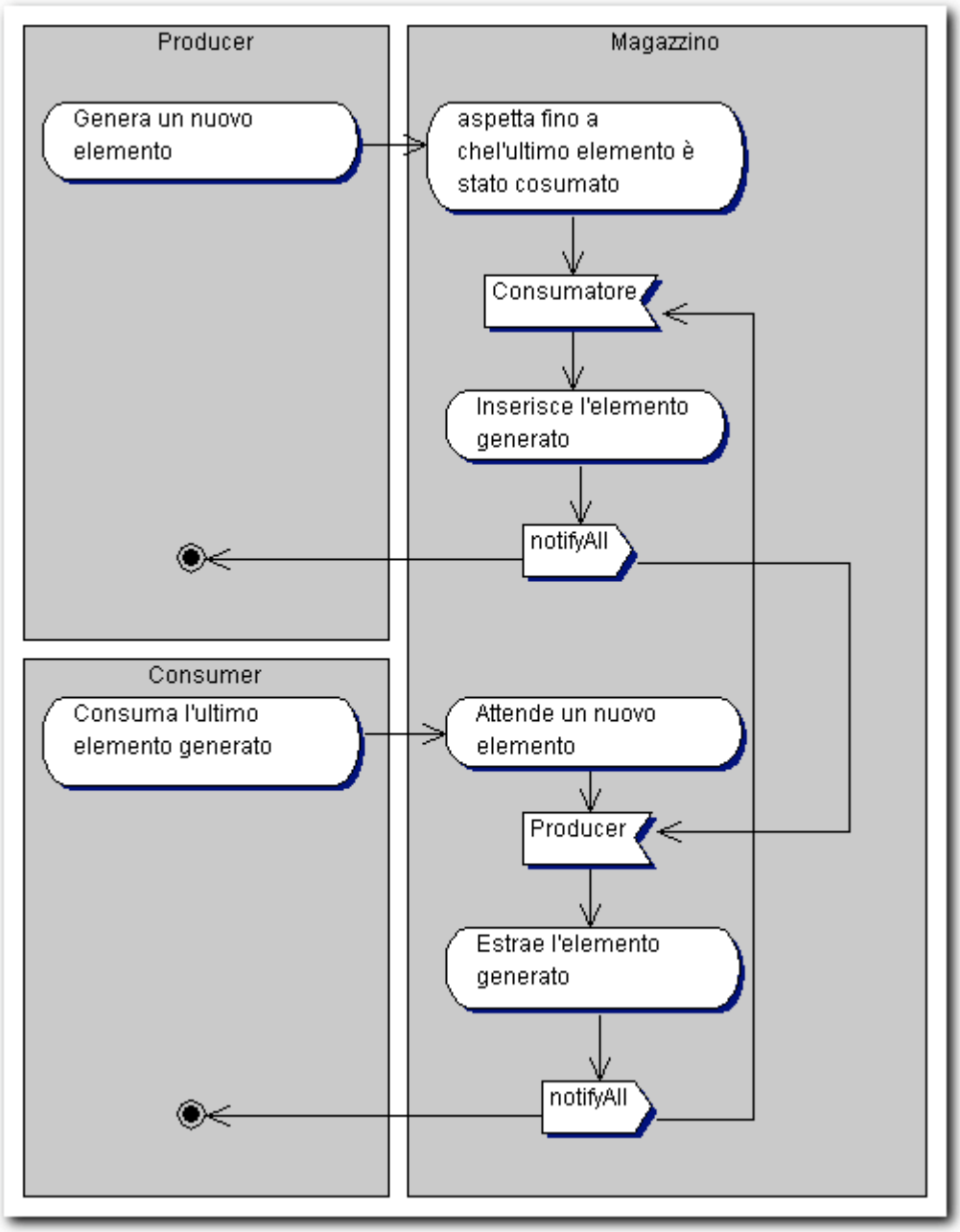


Figura 92: Activity-Diagram di Produttore / Consumatore

Il metodo *getProdotto()* entra in un ciclo attendendo che il produttore abbia generato un nuovo elemento. Ad ogni ciclo, viene eseguito il metodo *wait()* che, acquisisce il lock sull'oggetto generato dalla classe *Consumatore* (il metodo *getProdotto()* è sincronizzato) ed attende un segnale dalla classe *Produttore*. Così facendo, la classe *Produttore* può ottenere il lock sull'oggetto ed inserire il nuovo elemento generato. Terminata l'operazione, notifica allla

classe *Consumatore* che può riprendere il controllo dell'oggetto e continuare con l'esecuzione del metodo per acquisire l'ultimo intero generato.
Il metodo *setProdotto(**int**)* ha un funzionamento analogo e contrario.
Eseguendo nuovamente l'applicazione *ProduttoreConsumatore*, il risultato finale è esattamente quello che ci aspettavamo.

```
Produttore ha inserito: 0
Consumatore ha estratto: 0
Produttore ha inserito: 1
Consumatore ha estratto: 1
Produttore ha inserito: 2
Consumatore ha estratto: 2
Produttore ha inserito: 3
Consumatore ha estratto: 3
Produttore ha inserito: 4
Consumatore ha estratto: 4
.....
```

11.14 Blocchi sincronizzati

Nei paragrafi precedenti abbiamo visto come i metodi sincronizzati acquisiscano un lock su un oggetto o, nel caso di lock su classe, su tutti quelli di un determinato tipo.
Se il metodo sincronizzato viene eseguito molte volte e da molti thread, può rappresentare un collo di bottiglia in grado di deteriorare pesantemente le prestazioni di tutta l'applicazione. In generale, è quindi buona regola limitare le funzioni sincronizzate al minimo indispensabile.
Riprendiamo nuovamente la classe *Pila*, nella versione definita nel capitolo dedivato alle eccezioni e supponiamo che esista un numero non definito di thread che accede ai metodi *push* e *pop* dell'oggetto.
Possiamo facilmente proteggere l'oggetto sincronizzando i due metodi in questione.

```
package src.esercizi.thread;

import src.esercizi.eccezioni.StackOutOfBoundException;
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
* @version 0.5
 */
public class PilaDiInteri {
        /** Contenitore per i dati della Pila */
        Integer[] dati;

        /** Punta al primo elmento corrente nella struttura dati */
        int cima;

        /** Costruisce una pila contenente al massimo 10 elementi */
        public PilaDiInteri() {
                dati = new Integer[20];
                cima = 0;
```

```
	}

	/**
	* Costruisce una Pila, consentendo di impostarne la dimensione massima
	* @param int dimensione : dimensione massima della pila
	*/
	public PilaDiInteri(int dimensione) {
		dati = new Integer[dimensione];
		cima = 0;
	}

	/**
	* Inserisce un elemento sulla cima della pila
	* @param int dato : dato da inserire sulla cima della struttura dati
	* @return void
	* @exception <{StackOutOfBoundException}>
	*/
	public synchronized void push(Object dato) throws StackOutOfBoundException {
		if (cima < dati.length) {
			dati[cima] = (Integer)dato;
			cima++;
		}
		else {
			throw new StackOutOfBoundException("La pila ha raggiunto la dimensione
			massima. Il valore inserito è andato perduto +[" + dato + "]", dati.length);
		}
	}

	/**
	* Ritorna il primo elemento della pila
	* @return int : primo elemento sulla pila
	*/
	public synchronized  Object pop() {
		if (cima > 0) {
			cima--;
			return dati[cima];
		}
		return null;
	}
}
```

Anche se risolve il problema della concorrenza, la sincronizzazione dei metodi *push* e *pop* rende il codice poco efficiente. Analizzando bene la classe, notiamo che per poter salvaguardare i dati sulla pila, sarebbe sufficente proteggere l'array che contiene fisicamente i dati. La versione efficiente della classe *PilaDiInteri* diventa quindi:

```
package src.esercizi.thread;

import src.esercizi.eccezioni.StackOutOfBoundException;
/**
 * Questa classe definisce un tipo pila, una struttura dati di tipo LIFO
 * @version 0.5
 */
public class PilaDiInteri {
```

```
	/** Contenitore per i dati della Pila */
	Integer[] dati;

	/** Punta al primo elmento corrente nella struttura dati */
	int cima;

	/** Costruisce una pila contenente al massimo 10 elementi */
	public PilaDiInteri() {
		dati = new Integer[20];
		cima = 0;
	}

	/**
	* Costruisce una Pila, consentendo di impostarne la dimensione massima
	* @param int dimensione : dimensione massima della pila
	*/
	public PilaDiInteri(int dimensione) {
		dati = new Integer[dimensione];
		cima = 0;
	}

	/**
	* Inserisce un elemento sulla cima della pila
	* @param int dato : dato da inserire sulla cima della struttura dati
	* @return void
	* @exception <{StackOutOfBoundException}>
	*/
	public void push(Object dato) throws StackOutOfBoundException {
		if (cima < dati.length) {
			synchronized(dati){
				dati[cima] = (Integer)dato;
				cima++;
			}
		}
		else {
			throw new StackOutOfBoundException("La pila ha raggiunto la dimensione
			massima. Il valore inserito è andato perduto +[" + dato + "]", dati.length);
		}
	}

	/**
	* Ritorna il primo elemento della pila
	* @return int : primo elemento sulla pila
	*/
	public Object pop() {
		if (cima > 0) {
			cima--;
			return dati[cima];
		}
		return null;
	}
}
```

Quando un thread accede al metodo *push* della classe, ottiene il lock solo sull'array che contiene i dati e solo per il periodo limitato all'esecuzione del blocco di codice sincronizzato.

In generale, se non si può fare a meno di sincronizzare oggetti, è buona norma cercare di limitare la sincronizzazione a semplici blocchi di codice. In questo caso genereremo solo lock di tipo dinamico, ossia lock che acquisiamo e rilasciamo in continuazione all'interno di un singolo metodo.

12 JAVA NETWORKING

12.1 Introduzione

La programmazione client-server è un tema di primaria importanza poiché, è spesso necessario creare applicazioni che forniscano servizi di rete. Per ogni piattaforma supportata, Java supporta sia il protocollo TCP sia 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.

12.2 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 sono utilizzati, alle modalità di connessione che ognuno di loro utilizza per realizzare la trasmissione dei dati tra client e server o, viceversa.
Due applicazioni (client e server) connesse tramite una rete, comunicano tra 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:

1. *I dati necessari alla comunicazione tra le applicazioni ovvero quelli 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);*
2. *I dati contenenti informazioni strettamente legate alla comunicazione tra le applicazioni ovvero tutti i dati a carico dell'applicazione che trasmette il messaggio;*

Risulta chiaro che, parlando di protocolli, possiamo identificare due macro insiemi : *Protocolli di rete* e *Protocolli applicativi*.
Appartengono ai "protocolli di rete" i protocolli dipendenti dalla implementazione dello strato fisico e 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 prossime due tabelle, sono citati i più comuni protocolli appartenenti ai due insiemi.

Protocolli di rete	
Nome	**Descrizione**
TCP	Trasmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
ICMP	Internet Control Message Protocol

Protocolli applicativi	
Nome	**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

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 93* è riportato lo schema architetturale del TCP/IP dal quale risulterà più comprensibile la suddivisione nei due insiemi suddetti.

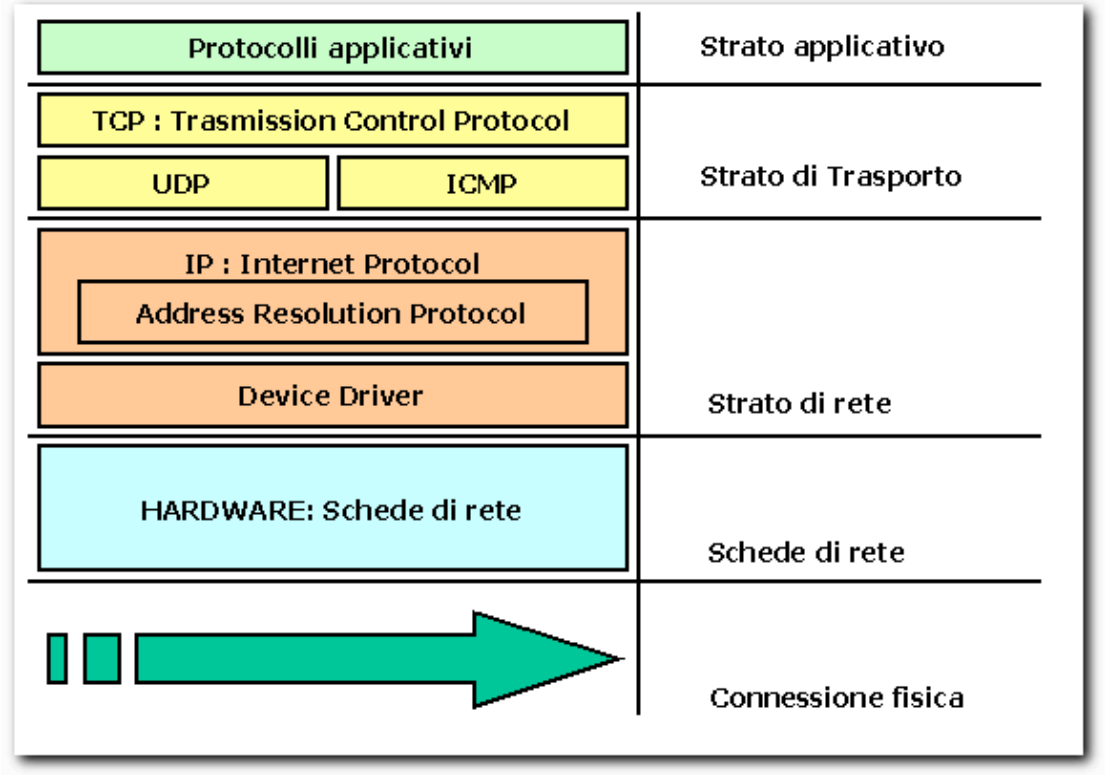


Figura 93: Architettura di TCP/IP

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.

12.3 Indirizzi IP

Per inviare pacchetti da un'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 in formato decimale, esadecimale o in altri formati.
Il formato più comune è quello che usa la notazione decimale separata da punti, con il quale l'indirizzo numerico a 32 bit è suddiviso in quattro sequenze di un byte, ognuna separate da punto, ed ogni byte è scritto mediante numero intero senza segno. Ad esempio, consideriamo l'indirizzo IP *0xCCD499C1* (in notazione esadecimale). Poiché:

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. Queste informazioni rappresentano l'indirizzo della rete del destinatario e l'indirizzo del computer destinatario all'interno di una rete. Gli indirizzi IP 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 rappresenta 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 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.

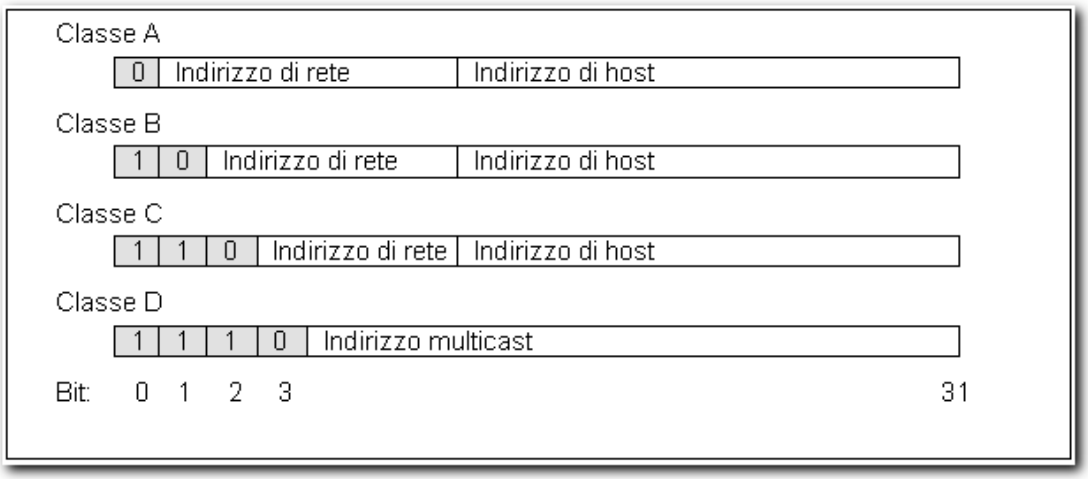


Figura 94: Le quattro forme di un indirizzo IP

12.4 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 utilizzata in tutti i casi in cui la rete debba garantire l'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 stabilisce un canale logico di comunicazione con il server. Per mezzo di questo canale, il client effettua tutte le 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 dell'integrità dei dati inviati, né dell'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.

12.5 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" degli indirizzi che ci interessaano. Oltre ad un indirizzo IP, ad un host è associato uno o più nomi chiamati "*Host Name*" che ci consentono di far riferimento ad 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".

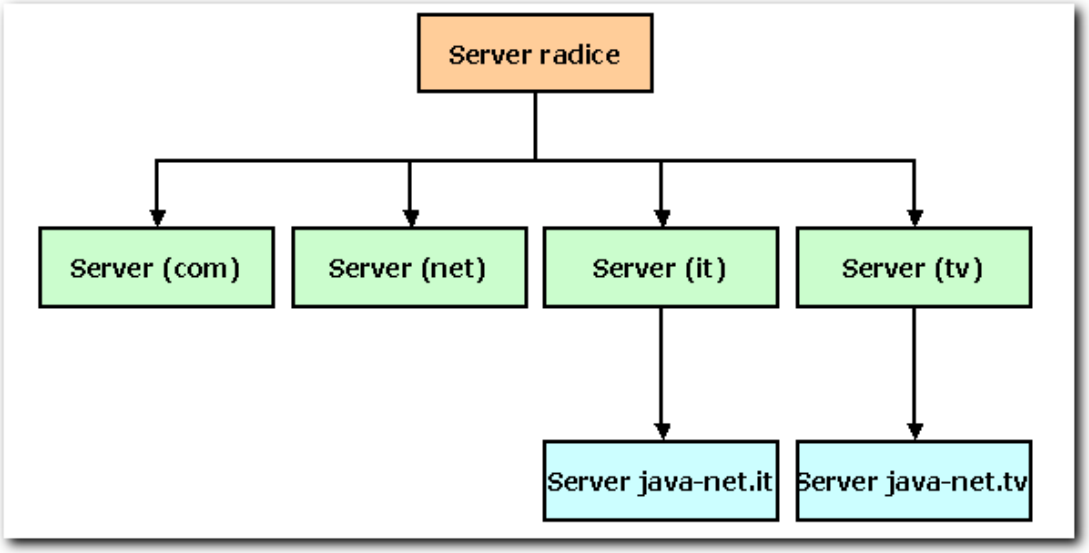


Figura 95: Gerarchia dei server DNS

Tipicamente un nome di host ha la seguente forma:

server.java-net.it

Dal punto di vista dell'utente il nome va letto da sinistra verso destra ossia dal nome locale (server) a quello globale (it) ed ha diversi significati: *server.java-net.it* si riferisce ad un singolo host all'interno di una rete ed è per questo motivo detto "*fully qualified*"; *java-net.it* si riferisce al dominio degli host collegati alla rete dell'organizzazione "*java-net*". Infine, *it* si riferisce ai sistemi amministrativi nella rete mondiale Internet ed è detto "nome di alto livello"
Un nome quindi definisce una struttura gerarchica. Proprio su questa gerarchia si basa il funzionamento della risoluzione di un nome. Nella *Figura 95* è illustrata la struttura gerarchica utilizzata dai server DNS per risolvere i nomi. Dal punto di vista di un server DNS il nome non è 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* ***server.java-net.it;***

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 "it";*

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 "java-net.it" il quale verrà a sua volta interrogato per risolvere infine il nome completo "server.java-net.it" Nella Figura 96 è schematizzato il percorso descritto, affinché il nostro DNS ci restituisca la corretta risposta.*

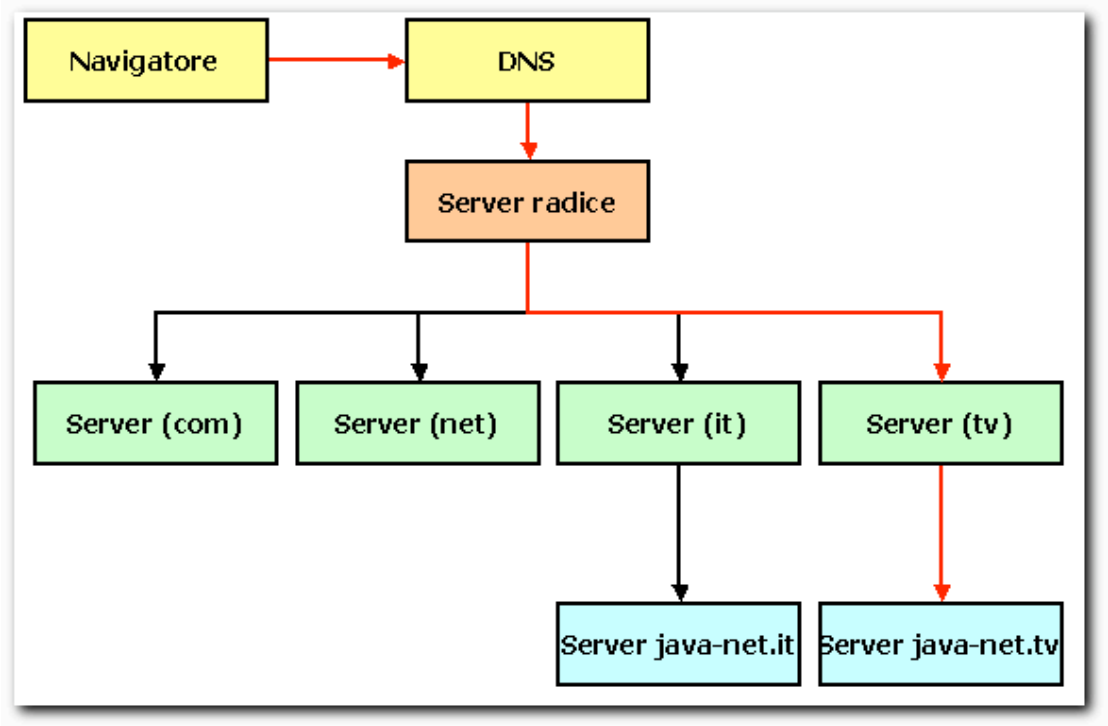


Figura 96: flusso di ricerca all'interno della gerarchia dei server DNS

12.6 URL

Parlando di indirizzamento all'interno di una rete, il nome di un Host non rappresenta ancora il nostro punto di arrivo. Con le nostre conoscenze siamo ora in grado di trovare ed indirizzare un host, ma non siamo ancora in grado di accedere ad una risorsa, di qualunque tipo essa sia.
Navigando in Internet sentiamo spesso nominare il termine URL. URL è acronimo di "*Uniform Resource Locator*", 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.it/index.html

in cui:

1. *Http indica il protocollo (in questo caso Hyper Text Trasfer Protocol)*
2. *//www.java-net.tv/index.html è il nome completo della risorsa richiesta.*

In definitiva una 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 percorso completo della risorsa sul computer indirizzato 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.

12.7 Trasmission Control Protocol : trasmissione Connection Oriented

Il protocollo TCP, appartenente allo strato di trasporto del TCP/IP (*Figura 93*) trasmette dati da server a client, e viceversa, suddividendo un messaggio di dimensioni arbitrarie in frammenti o "*datagrammi*" da spedire separatamente e non necessariamente in maniera sequenziale, per poi ricomporli nell'ordine corretto; una eventuale nuova trasmissione può essere richiesta per il pacchetto non arrivato a destinazione o contenente dati affetti da errore.
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 specie di canale attraverso il quale le due applicazioni possono inviarsi dati in forma di pacchetti di lunghezza arbitraria.
Per questo motivo, il protocollo TCP fornisce un servizio di trasporto affidabile, garantendo una corretta trasmissione dei dati tra applicazioni.
Nella *Figura 97* 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.

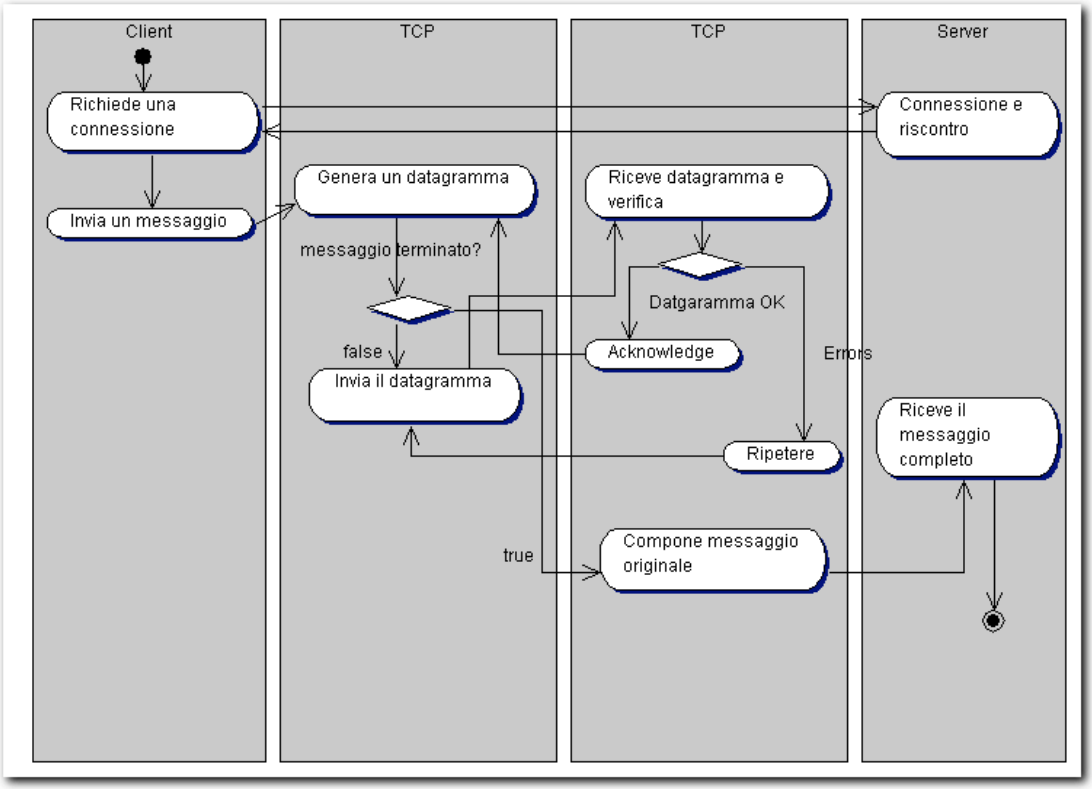


Figura 97: Time-Line di una trasmissione "Connection Oriented"

Un'altra facoltà del TCP, è quella di poter bilanciare la velocità di trasmissione tra mittente e destinatario, cosa molto utile soprattutto nel caso in cui la trasmissione avvenga attraverso reti eterogenee.
In generale, quando due sistemi comunicano tra loro, è necessario stabilire le dimensioni massime che un datagramma può raggiungere affinché possa esservi trasferimento di dati. Tali dimensioni sono dipendenti dall'infrastruttura di rete, dal sistema operativo della macchina host e possono quindi variare anche tra computer appartenenti alla stessa rete.
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), di volta in volta il prtocollo trasmette 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.
Nonostante le sue caratteristiche, dovendo scegliere il protocollo per la trasmissione di dati, il TCP/IP non risulta sempre essere la scelta migliore.

Dovendo fornire tanti servizi, oltre ai dati relativi al messaggio da trasmettere il protocollo deve trasportare una quantità informazioni, a volte non necessarie, che spesso possono diventare causa di sovraccarico sulla rete. Nella prossima figura è stata schematizzata la struttura dei primi sedici ottetti (byte) di un datagramma TCP. In questi casi, è necessario favorire la velocità a discapito della qualità della trasmissione, adottando trasmissioni di tipo "Connectionless".

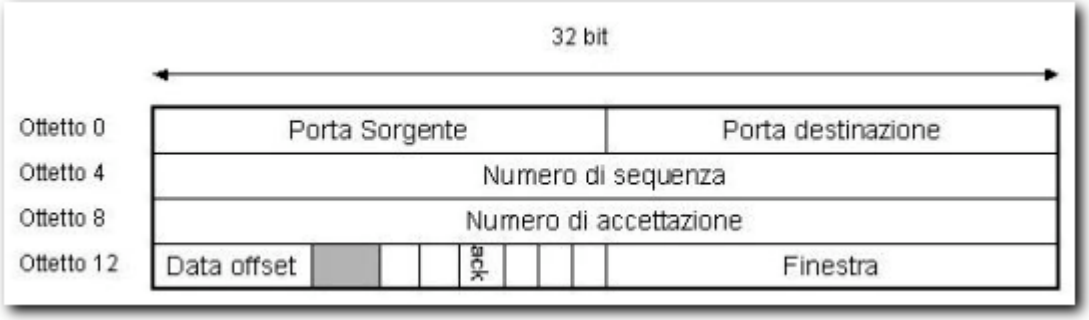


Figura 98: ***I primi 16 ottetti di un segmento TCP***

12.8 User Datagram Protocol : trasmissione Connectionless

User Datagram Protocol si limita ad eseguire operazioni di trasmissione di un pacchetto di dati tra mittente e destinatario con il minimo sovraccarico di rete, senza però garanzia di consegna. Di fatto, questo 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 99: ***Segmento di un datagramma UDP***

Nella *Figura 99* viene mostrato il formato del datagramma UDP. Dal confronto con l'immagine precedente, appare chiaro quanto siano ridotte le dimensioni di un 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 del pacchetto originario, e per il recupero di dati eventualmente persi nella trasmissione.

12.9 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 loro o con applicazioni esistenti su altri computer host tramite il TCP/IP. 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 trasmettere o ricevere dati, è quello di identificarsi al TCP/IP, affinché possa essere riconosciuto dallo strato di gestione della rete. Il riconoscimento di un processo nei confronti della rete, viene effettuato tramite un numero detto *porta* 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 la medesima *porta*), di conseguenza è necessario introdurre un nuova definizione.

Una "Association", è una quintupla formata dalle informazioni necessarie a descrivere univocamente una connessione :

Association = (Protocollo,Indirizzo Locale,Processo Locale,Indirizzo Remoto,Processo Remoto)

Ad esempio, è una "Association" la 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 porta del processo locale, "194.243.233.4" è l'indirizzo IP del computer destinatario, ed infine "21" è l'identificativo o porta del processo remoto.
Come fanno 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 cinque elementi della quintupla. Come si vede dalla *Figura 9*, questo insieme più essere costruito mediante unione a partire da due sottoinsiemi, che chiameremo rispettivamente *Local Half Association* e *Remote Half Association*:

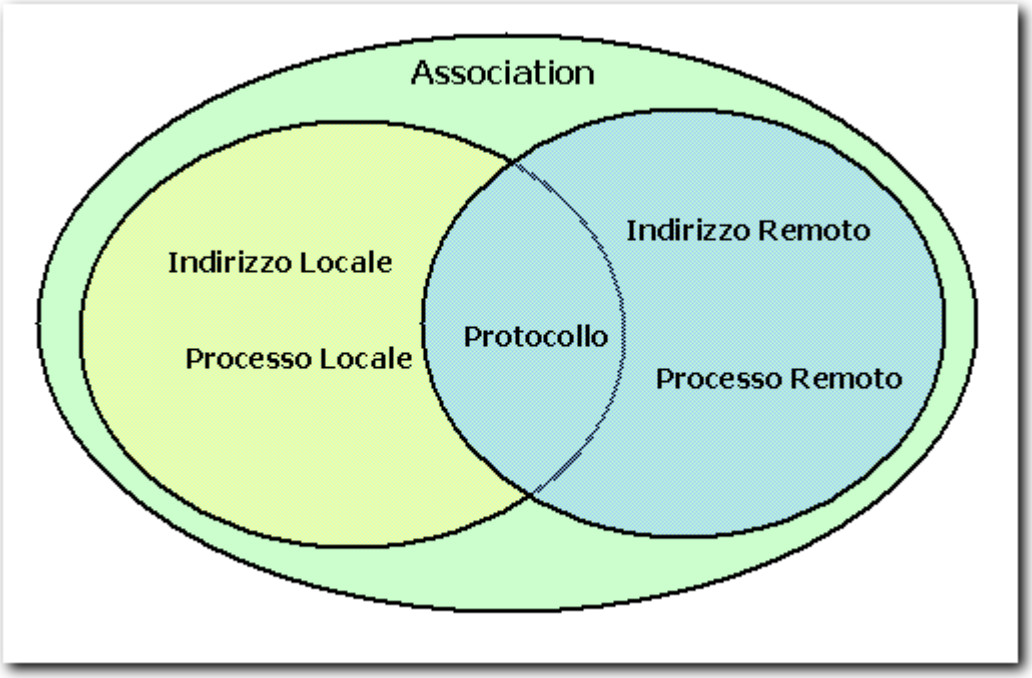


Figura 100: "Association" = "Local Half Association" U "Remote Half Association"

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. 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)

12.10 Il package java.net

E' il *package* Java che contiene le definizioni di classe necessarie per lo sviluppo di applicazioni client/server basate su *socket*.
Nei prossimi paragrafi analizzeremo le principali classi di questo package e come utilizzare gli oggetti per realizzare trasmissione di dati via rete tra applicazioni Java.
La prossima tabella, contiene la gerarchia delle classi e delle interfacce appartenenti a questo package. Scorrendo velocemente i loro nomi, notiamo subito le 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**

Un indirizzo IP è rappresentato dalla classe *InetAddress*, che fornisce tutti i metodi necessari a manipolare un indirizzo per instradare dati sulla rete, e consente la trasformazione di un indirizzo nelle sue varie forme:

1. *getAllByName(String host)* ritorna tutti gli indirizzi internet di un host dato il suo nome in forma estesa (esempio: getAllByName("www.java-net.it"));

2. *getByName(String host)* ritorna l'indirizzo internet di un host dato il suo nome in forma estesa;

3. *getHostAddress()* ritorna una stringa rappresentante l'indirizzo IP dell'host nella forma decimale separata da punto "%d.%d.%d.%d";

4. *getHostName()* ritorna una stringa rappresentante il nome dell'host.

Le classi *Socket* e *ServerSocket* implementano rispettivamente un socket client ed uno 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.

12.11 Un esempio completo di applicazione client/server

Nei prossimi paragrafi analizzeremo in dettaglio un'applicazione client/server basata su modello di trasmissione con connessione. Prima di entrare nei dettagli dell'applicazione, è necessario però porci una domanda: cosa accade tra due applicazioni che comunicano con trasmissione orientata a connessione?
Il server, eseguito da un computer host, è in ascolto tramite socket su una determinata porta, in attesa di richiesta di connessione da parte di un client (*Figura 101*). Il client invia la sua richiesta al server tentando la connessione tramite la porta su cui il server è in ascolto.

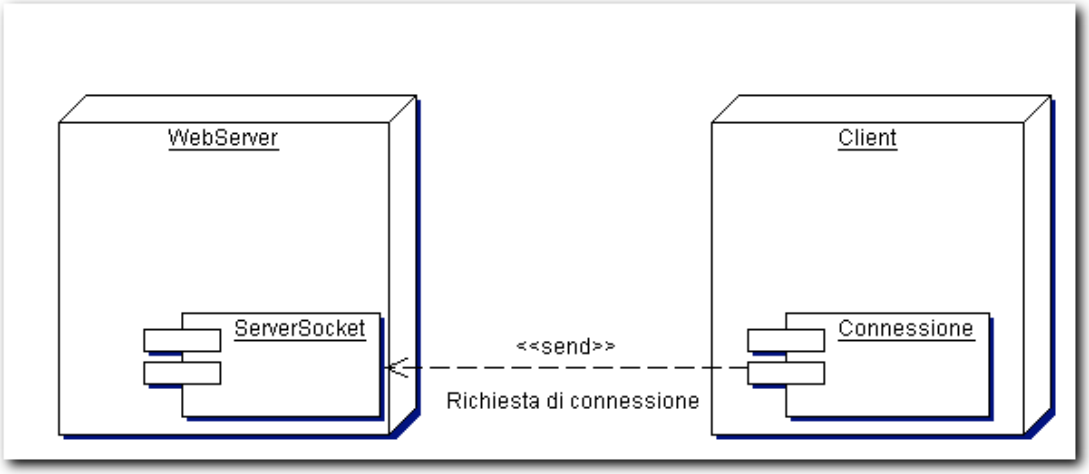


Figura 101: Richiesta di connessione Client/Server

Dobbiamo a questo punto distinguere due casi:

1. *il server comunica con un solo client alla volta;*
2. *il server deve poter comunicare con più client contemporaneamente.*

Nel primo caso, il server accetta 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 crea fisicamente un socket su una porta locale tramite il quale trasmettere e ricevere pacchetti (*Figura 102*).

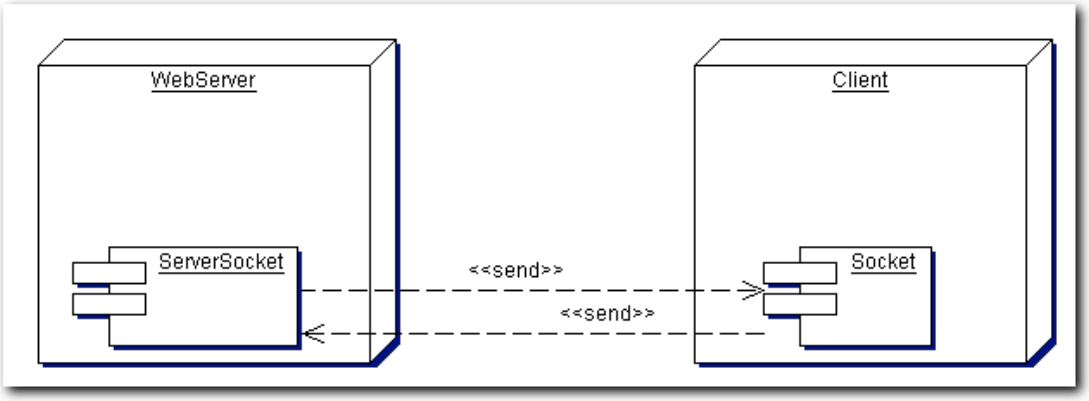


Figura 102: Connessione Unicast

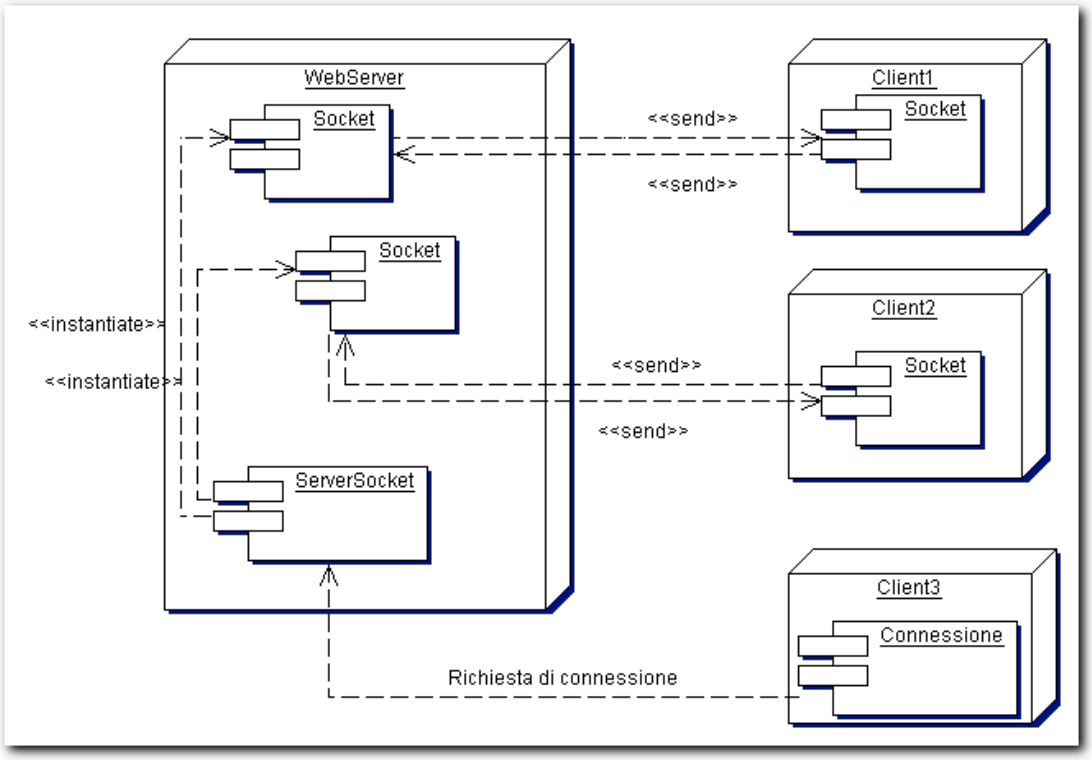


Figura 103: Connessione Multicast

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 crea un nuovo socket associato ad una nuova porta (tipicamente avvia un nuovo thread per ogni connessione), a cui cede in gestione il 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 103*).
Un buon esempio di applicazione client/server che implementano trasmissioni multicast sono browser internet ed il Web Server. A fronte di un numero indefinito di client connessi (browser), 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.

12.12 La classe ServerSocket

La classe *ServerSocket* rappresenta quella porzione del server che può accettare richieste di connessioni da parte del client. Quest'oggetto deve essere creato passando come attributo al metodo costruttore, 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()*, che blocca il thread in attesa di una richiesta di connessioni da parte di un client sulla porta specificata nel costruttore. Quando una richiesta di connessione va a buon fine, il metodo crea il canale di collegamento e restituisce un oggetto Socket già connesso con il client.
Nel caso di comunicazione multicast, sarà necessario creare un nuovo thread cui passare l'oggetto restituito, contenente il codice di gestione della trasmissione e gestione dei dati.

12.13 La classe Socket

La classe *Socket* rappresenta una connessione client/server via TCP su entrambi i lati. 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 crearlo 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 argomenti: 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 metodo costruttore è responsabile del completamento della connessione con il server. Nel caso in cui il tentativo non vada a buon fine, il metodo costruttore genererà una eccezione per segnalare l'errore.
Notiamo 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.

12.14 Un semplice thread di servizio

Abbiamo detto nei paragrafi precedenti che, quando implementiamo un server di rete che gestisce più client simultaneamente, è importante creare dei thread separati per comunicare con ognuno. 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.

12.15 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.

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

public class NetServ implements Runnable
{
        private ServerSocket server;
        public NetServ () throws Exception
        {
                server = new ServerSocket(2000);
        }
        public void run()
        {
                Socket s = null;
                Xfer x;

                while (true)
                {
                        try {
                                //aspetta la richiesta da parte del client
                                s = server.accept();
                        } catch (IOException e) {
                                System.out.println(e.toString());
                                System.exit(1);
                        }
                        //crea un nuovo thread per servire la richiesta
                        x = new xfer(s);
                }
        }
}
```

```
public class ServerProgram
{
        public static void main(String args[])
        {
                NetServ n = new NetServ();
                (new Thread(n)).start();
        }
}
```

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.

12.16 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.

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

public class NetClient implements Runnable
{
        private Socket s;
        private DataInputStream input;
        public NetClient(String host, int port) throws Exception
        {
                s = new Socket(host, port);
        }
        public String read() throws Exception
        {
                if (input == null)
                        input = new DataInputStream(s.getInputStream());
                return input.readLine();
        }
        public void run()
        {
                while (true)
                {
                        try {
                                System.out.println(nc.read());
                        } catch (Exception e) {
                                System.out.println(e.toString());
                        }
                }
        }
}
```

```
class ClientProgram
{
        public static void main(String args[])
        {
                if(args.length<1) System.exit(1);
                NetClient nc = new NetClient(args[0], 2000);
                (new Thread(nc)).start();
```

```
        }
}
```

Parte Seconda

Java 2 Enterprise Edition

13 JAVA ENTERPRISE COMPUTING

13.1 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 104* mostra schematicamente un ipotetico scenario di "Architettura Enterprise" evidenziando alcune possibili integrazioni tra componenti server-side. Appare chiara la disomogeneità tra le componenti e di conseguenza la complessità strutturale di questi sistemi.

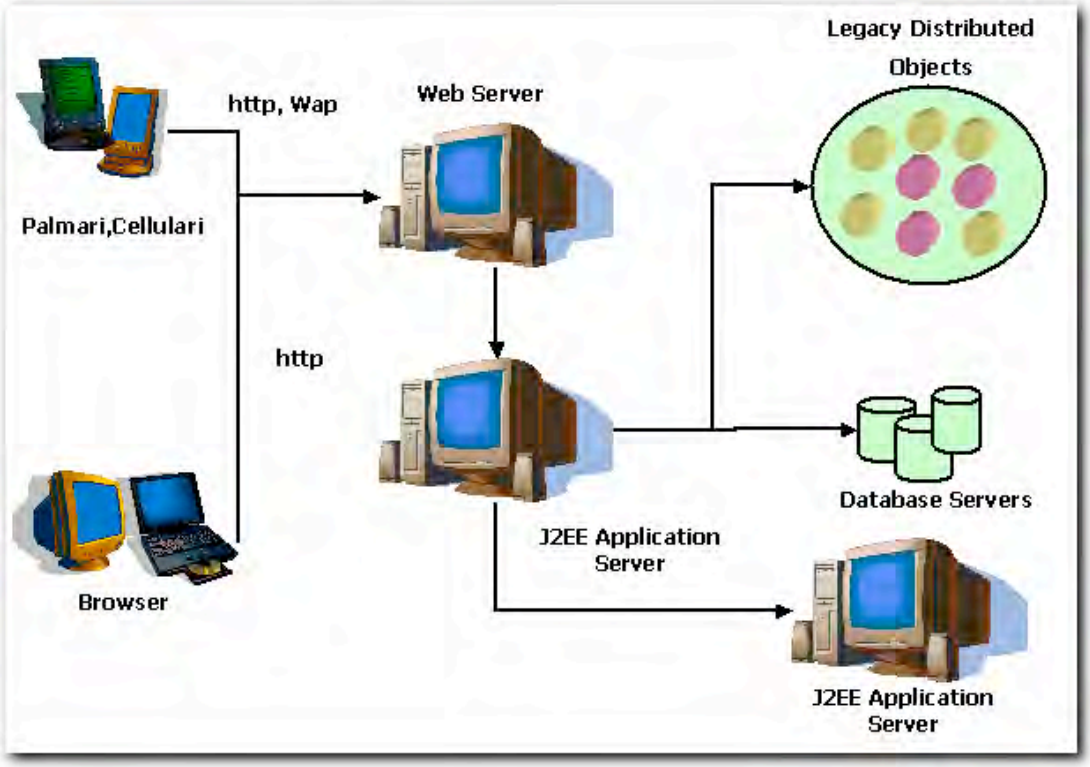


Figura 104: Architetture enterprise

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.

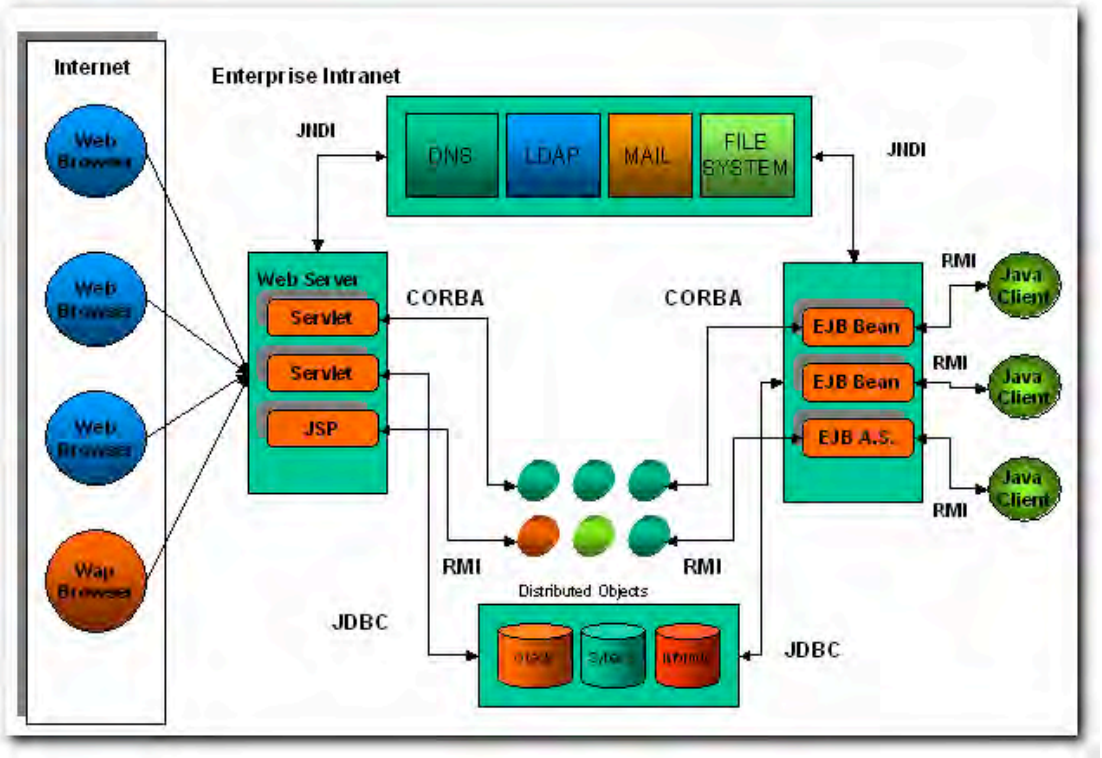


Figura 105: Modello distribuito basato su java

Nella *figura 105* 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.

13.2 Architettura di J2EE

L'architettura proposta dalla piattaforma J2EE divide le applicazioni enterprise in tre strati applicativi fondamentali (*figura 106*): 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;
Servlet;
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 verranno eseguiti. 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.

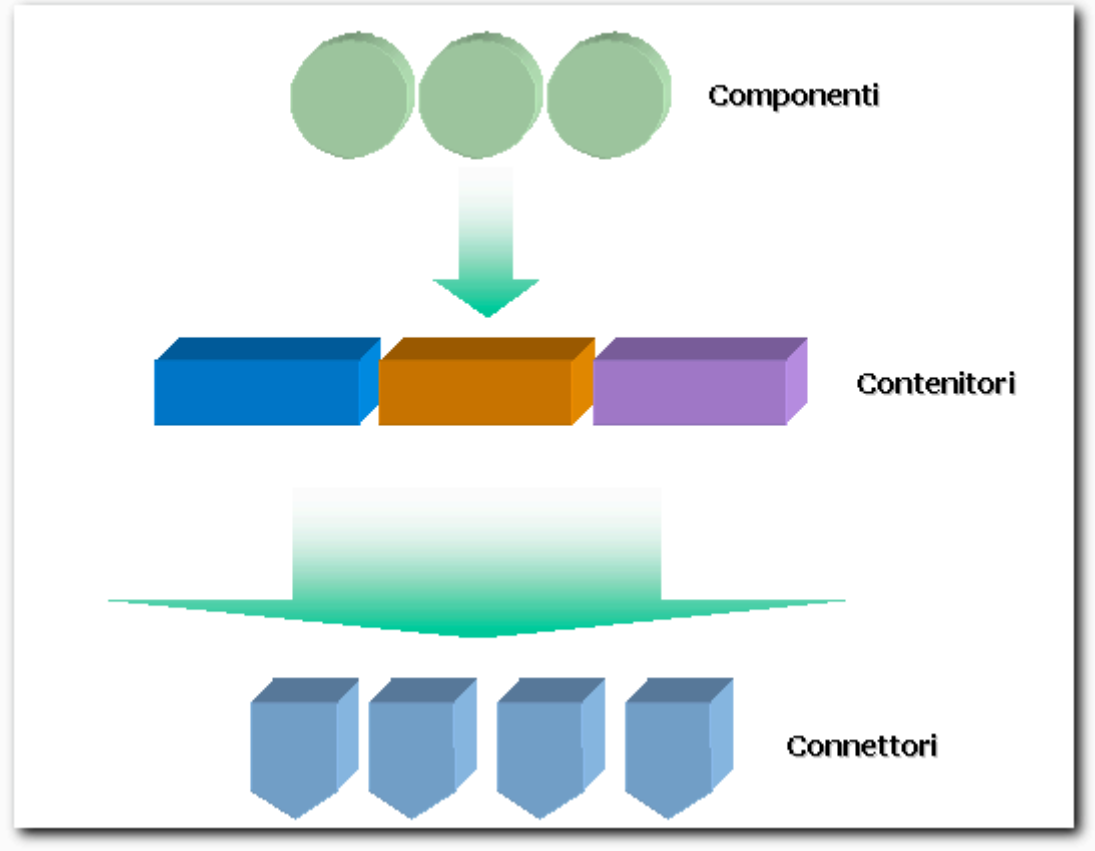


Figura 106: Architettura J2EE

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.

13.3 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 piattaformache provvede autonomamente alle componenti ed al loro supporto.
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 107*) : Client Tier, Web Tier, Business-Tier, EIS-Tier.

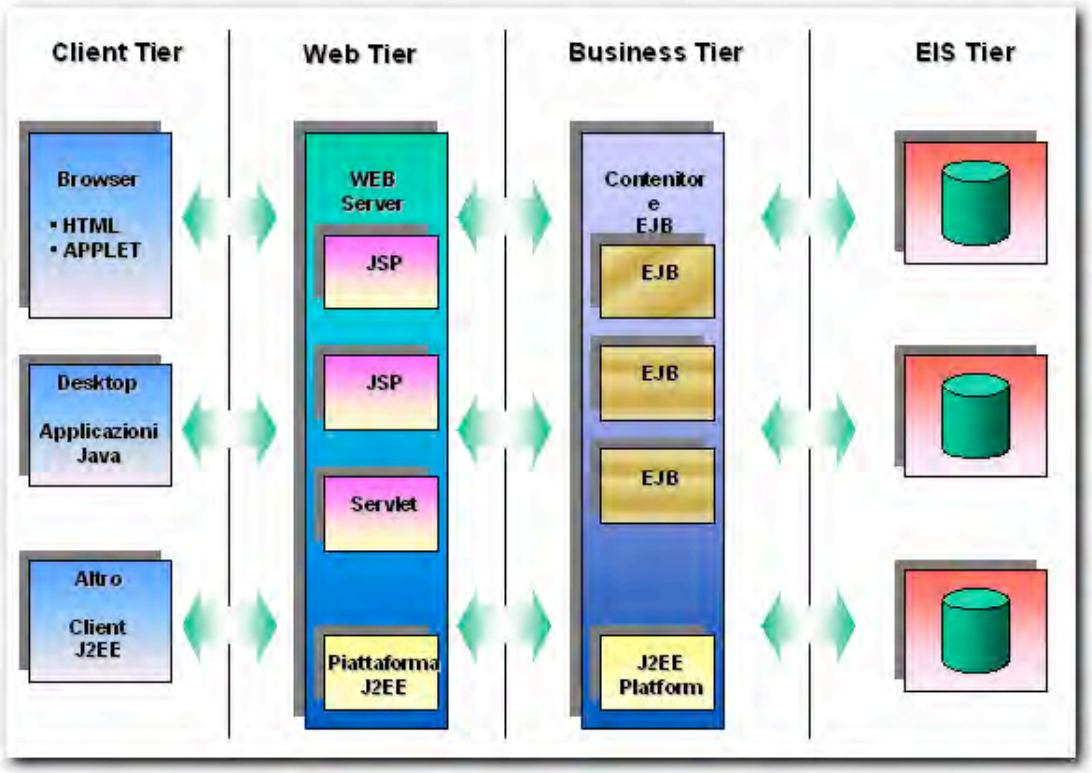


Figura 107: J2EE Application Programmino Model

13.4 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.
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 108* vengono illustrate schematicamente le applicazioni appartenenti a questo strato e le modalità di interconnessione verso gli strati componenti la architettura del sistema.

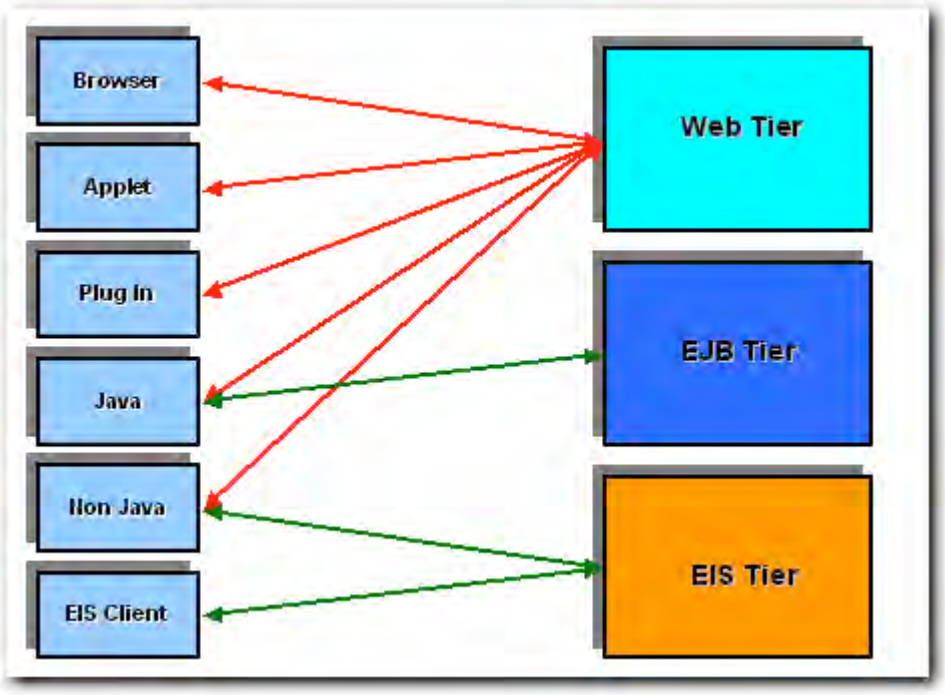


Figura 108: Client Tier

13.5 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 109*).
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.

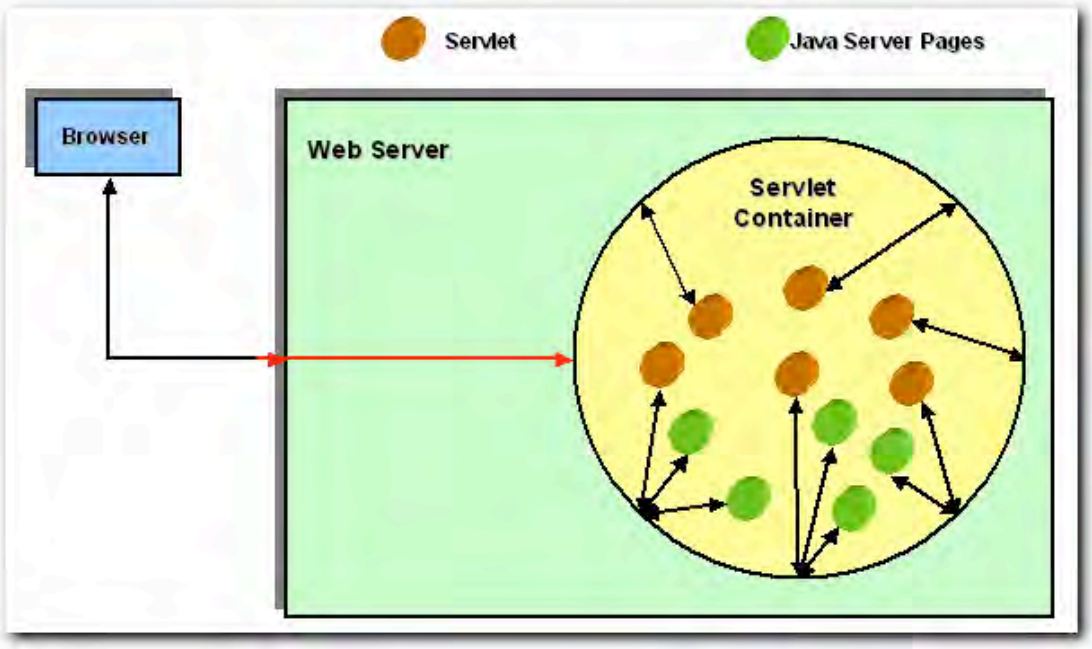


Figura 109: Web Tier

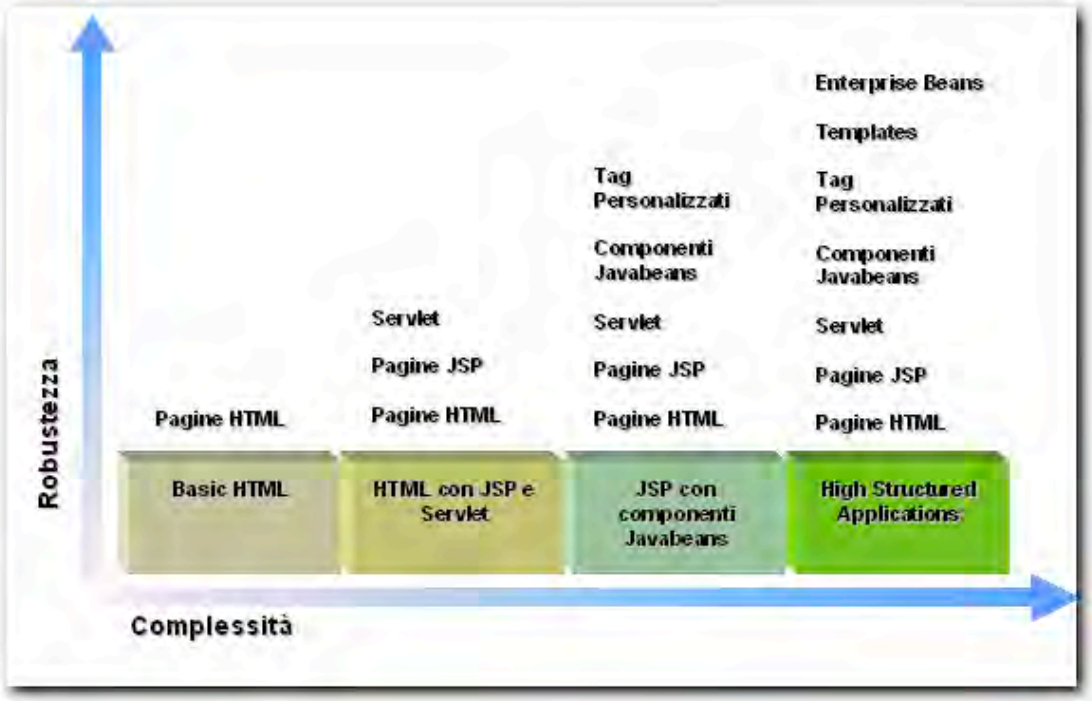


Figura 110: Application Design

Nella *figura 110* viene presentata in modo schematico tutta la gamma di applicazioni web ed il loro uso in relazione a due fattori principali: complessità e robustezza.

13.6 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 objects*".
La tecnologia fornita da Sun per implementare i "business objects" 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 111*).

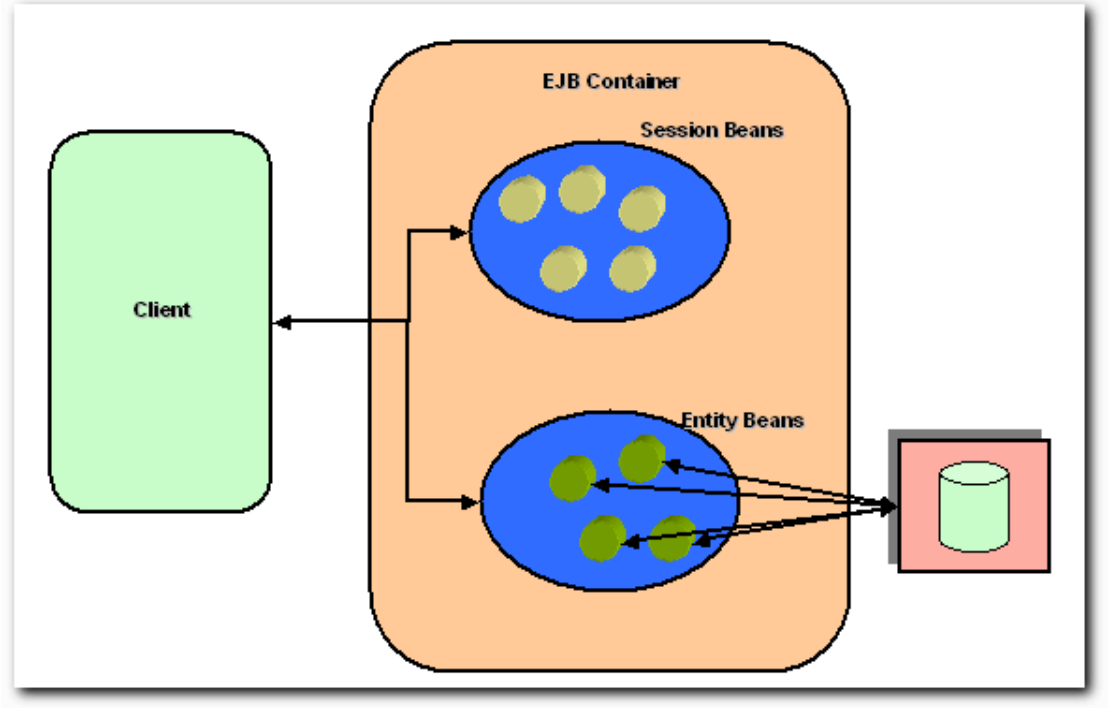


Figura 111: Business Tier

Oltre alle componenti, la architettura EJB definisce altre tre 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 altri 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 nei capitoli successivi).
Quando un client invoca una operazione su un EJB, la chiamata viene intercettata 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.

13.7 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.
Al modello bidimensionale delle vecchie forme di business legato ai sistemi informativi (*figura 112*), 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 113*) .
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.

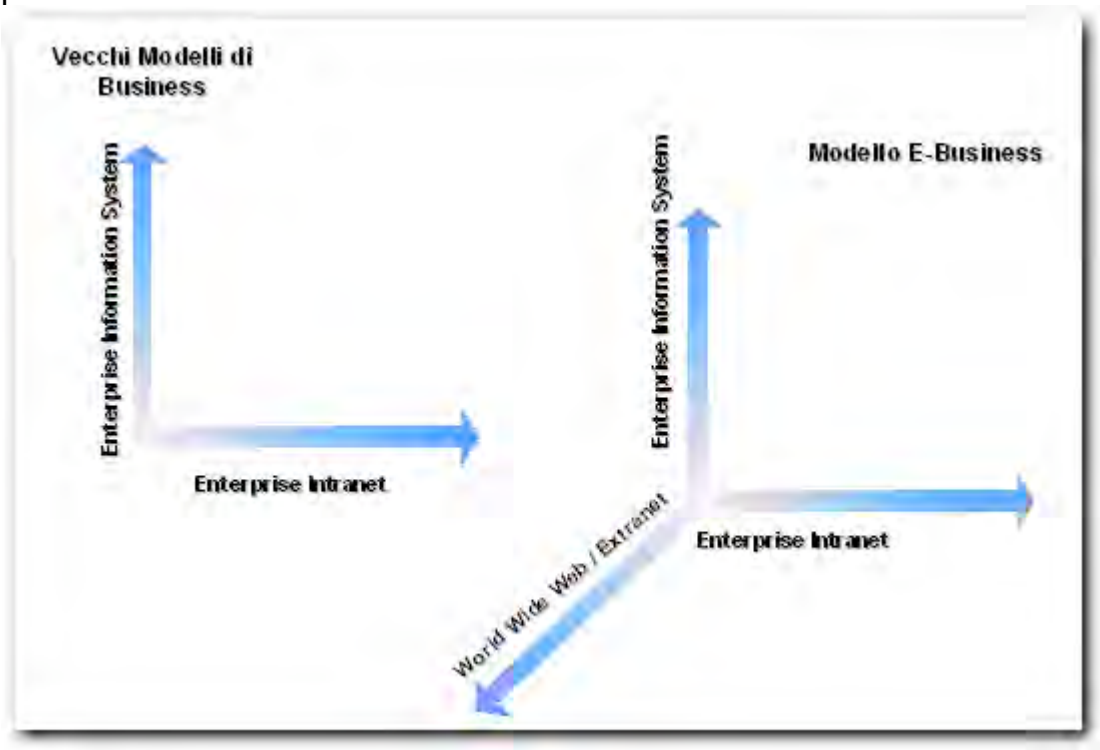


Figura 112: Modelli di business

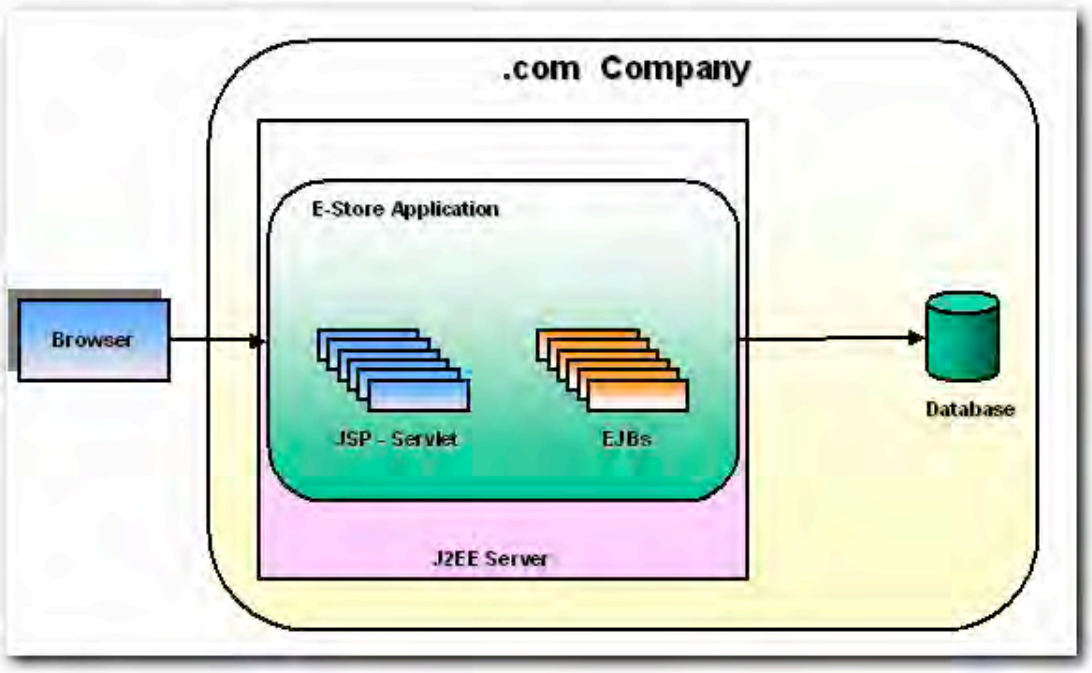


Figura 113: Negozi Virtuali o E-Store

13.8 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.

13.9 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 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.
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.

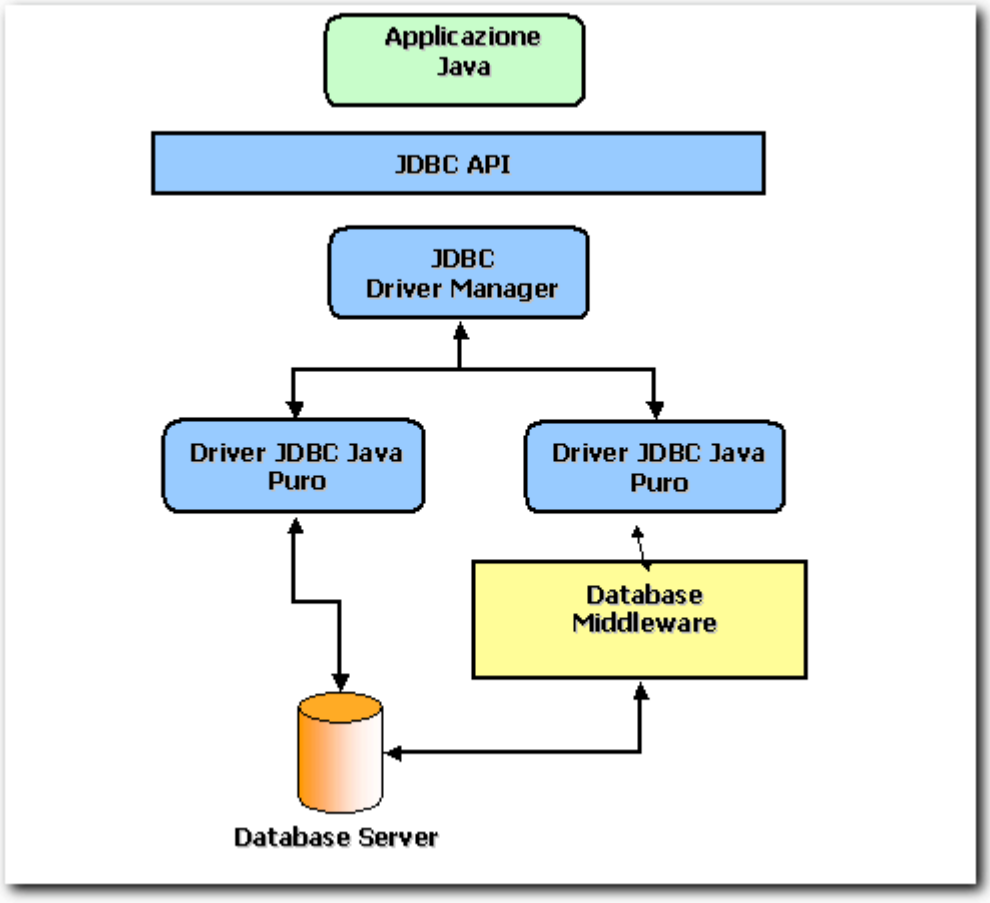


Figura 115: Driver JDBC Tipo 3 e 4

I driver JDBC possono essere suddivisi in quattro tipi fondamentali come da tabella:

Driver JDBC	
Nome	**Descrizione**
Tipo 4	Driver diretto Java nativo
Tipo 3	Driver Java per connessione diretta a middleware.
Tipo 2	JDBC-ODBC Bridge e ODBC driver;
Tipo 1	Driver nativo parzialmente sviluppato con Java.

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 115*) e sono scritti completamente in Java.
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 116*) e rappresentano una buona alternativa in situazioni in cui non siano disponibili driver di tipo 3 o 4.

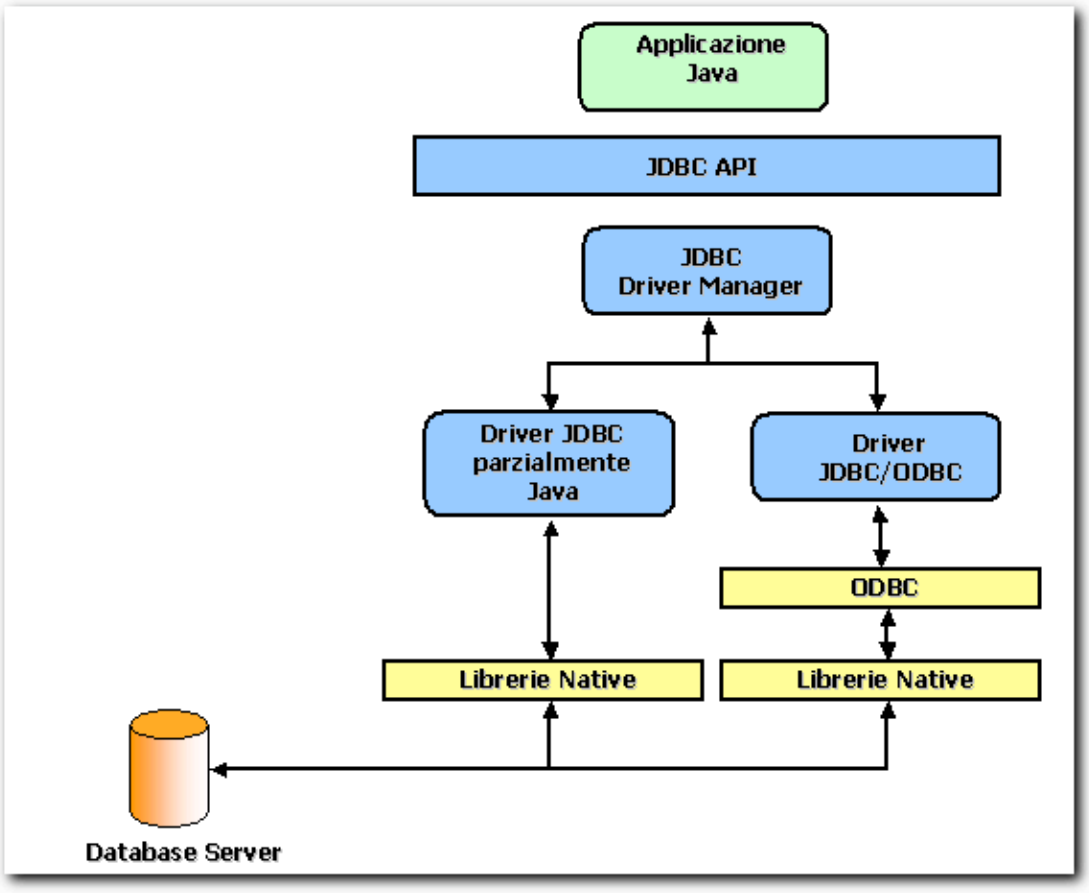


Figura 116: Driver JDBC Tipo 1 e 2

13.10 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.

13.11 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.

13.12 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 117*).
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.

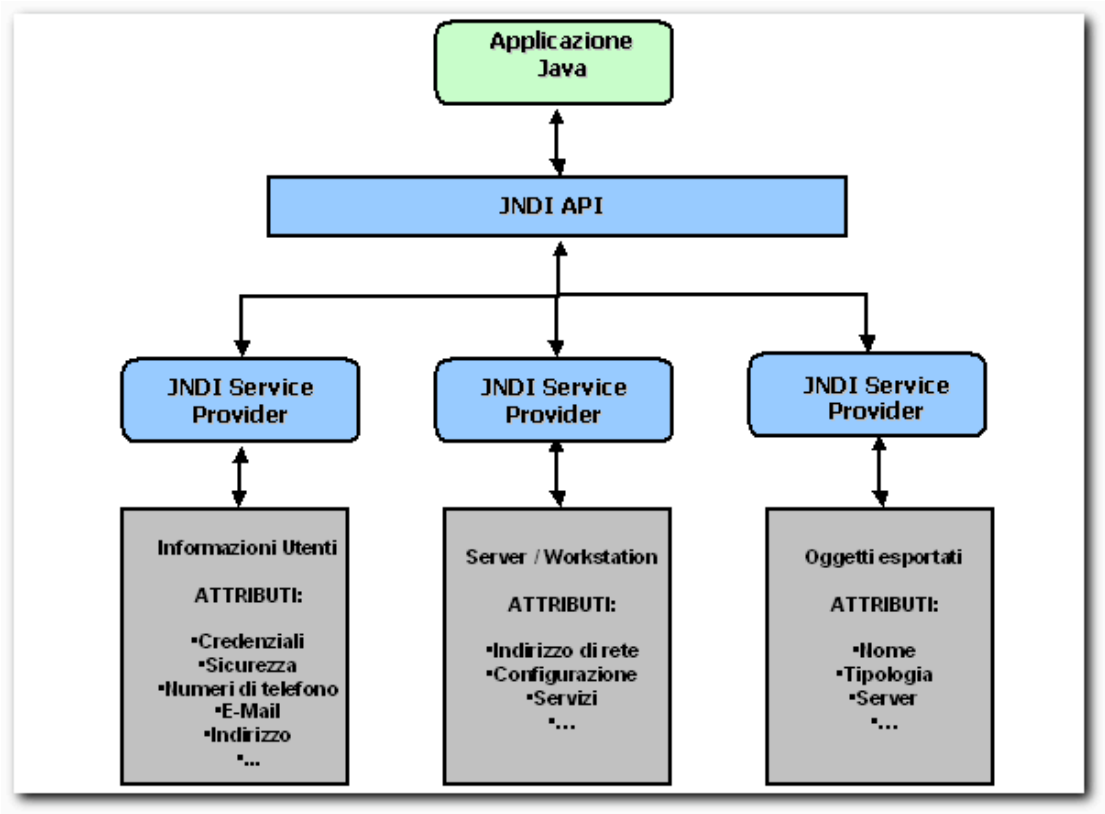


Figura 117: Architettura JNDI

13.13 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. Una applicazione di "Home Banking" ad esempio, deve interfacciarsi con il sistema di messaggistica bancaria ed interbancaria da cui trarre tutte le informazioni relative alla gestione economica dell'utente.
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.

14 ARCHITETTURA DEL WEB TIER

14.1 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, 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 da remoto tramite Web. 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.

14.2 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 questa architettura, costituendone uno strato autonomo. 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 118*).

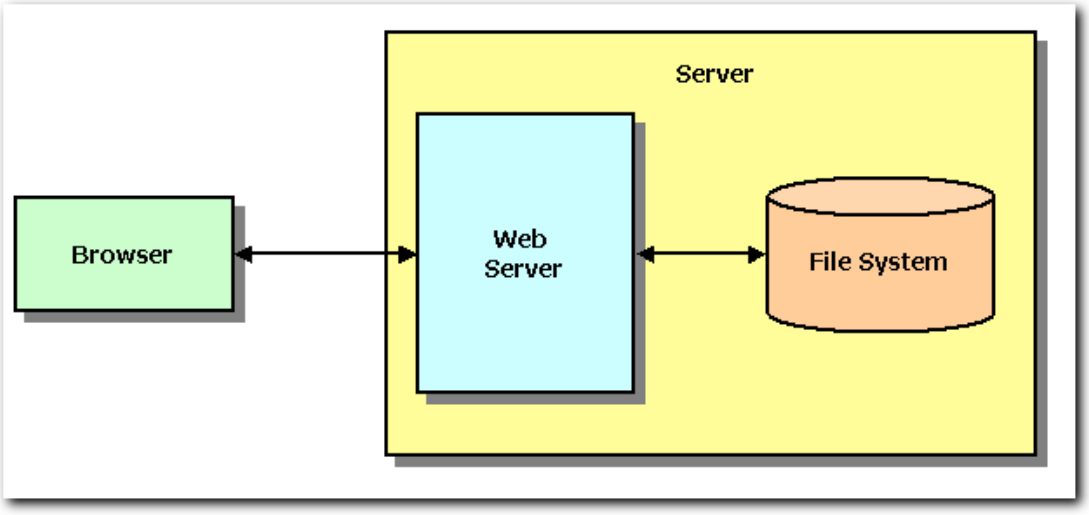


Figura 118: Architettura del web tier

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 figura precedente ora si complica in quanto viene introdotto un nuovo grado di complessità (*Figura 119*) nella architettura del sistema, i quale, 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.

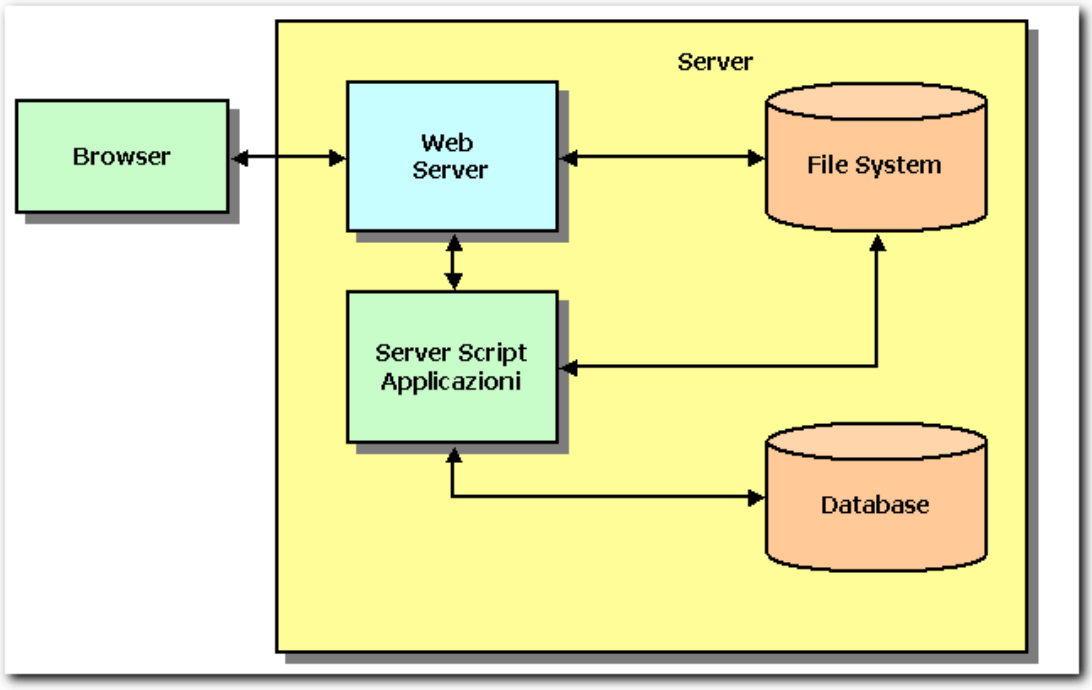


Figura 119: Architetture dinamiche

14.3 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.it/servelt/Hello?nome=Massimo&cognome=Rossi

"http://www.java-net.it" 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, ad esempio, 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 dal momento che i contenuti dei campi di un form vengono inglobati all'interno di un message-header. Rispetto al precedente, il metodo POST consente di inviare una quantità maggiore di dati ed è quindi utilizzato, ad esempio, quando è necessario inviare al server nuovi dati affinché possano essere memorizzati all'interno della Base Dati. Si noti, inoltre, come l'utilizzo del metodo POST nasconda all'utente la struttura applicativa, dal momento che non vengono mostrati i nomi delle variabili utilizzati per inviare le informazioni.

14.4 Sviluppare applicazioni web

Dal punto di vista del flusso informativo, una applicazione web è paragonabile ad una qualsiasi altro tipo di applicazione, con la fondamentale differenza relativa alla presenza del web-server che fornisce l'ambiente di runtime per l'esecuzione. Affinché ciò sia possibile è necessario che esista un modo standard affinché il web-server possa trasmettere i dati inviati dal client all'applicazione, 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.

14.5 Common Gateway Interface

CGI è sicuramente la più comune e datata tra le tecnologie server-side per lo sviluppo di applicazioni web dinamiche. 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 vengono generalmente utilizzati linguaggi quali PERL e PHP, sono semplici da implementare, ma hanno lo svantaggio di essere molto lenti dal momento che la loro esecuzione richiede 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).
Nella *Figura 120* è schematizzato il ciclo di vita di un CGI. Il più grande svantaggio nell'uso dei CGI sta nella scarsa scalabilità della tecnologia: ogni volta che il web-server riceve una richiesta deve, infatti, creare una nuova istanza del CGI. Il dover istanziare un processo per ogni richiesta ha come conseguenza enormi carichi in fatto di risorse macchina e tempi di esecuzione. Tipicamente, per ogni richiesta, all'avvio del CGI devono essere ripetuti i passi seguenti:

1. *Caricamento dell'applicazione e avvio;*
2. *Connessione alla base dati del server;*
3. *Estrazione dei dati dalla richiesta http;*
4. *Elaborazione dei dati;*
5. *Costruzione della risposta in formato HTML;*
6. *Trasmissione della risposta.*

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.

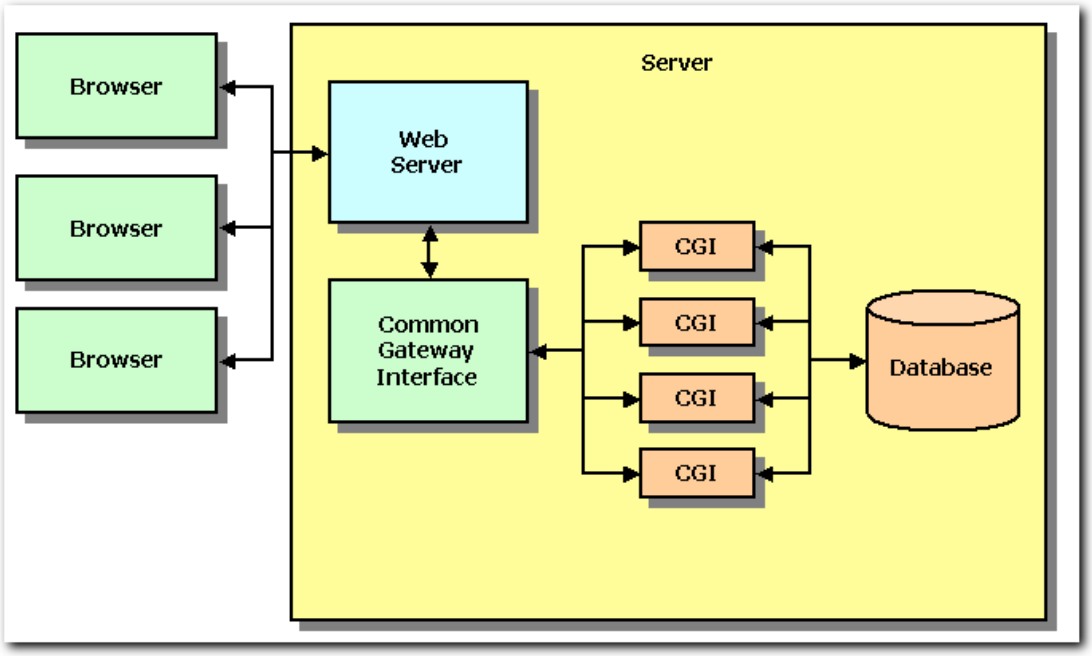


Figura 120: Common Gateway Interface

14.6 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.

14.7 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. 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.

14.8 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 e 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.

15 JAVA SERVLET API

15.1 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 la funzione di 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.

15.2 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 121* 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.

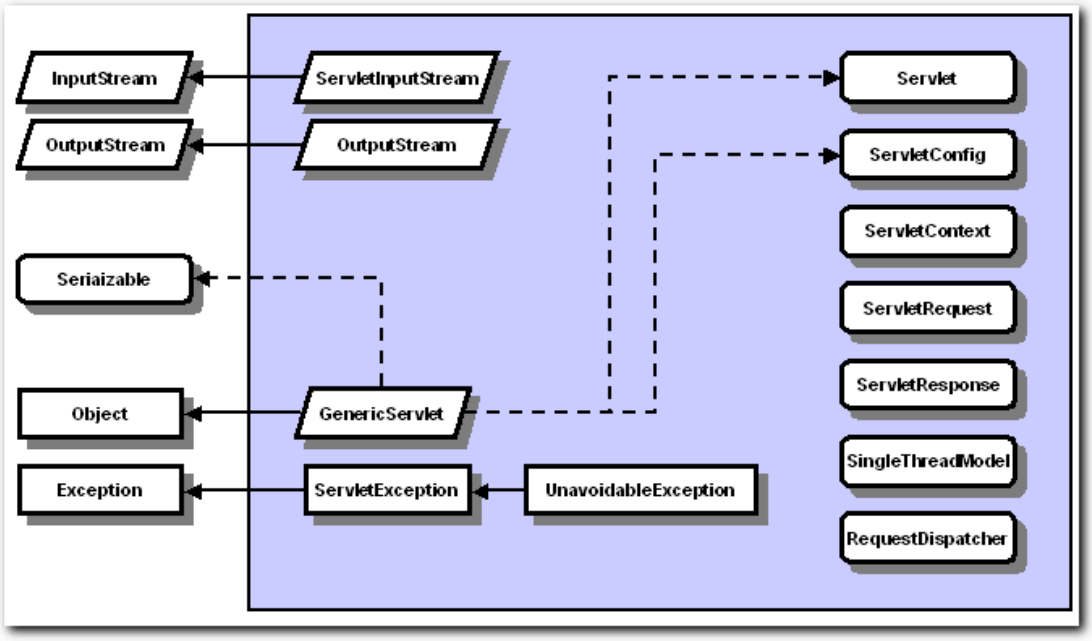


Figura 121: Il package javax.servlet

15.3 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.

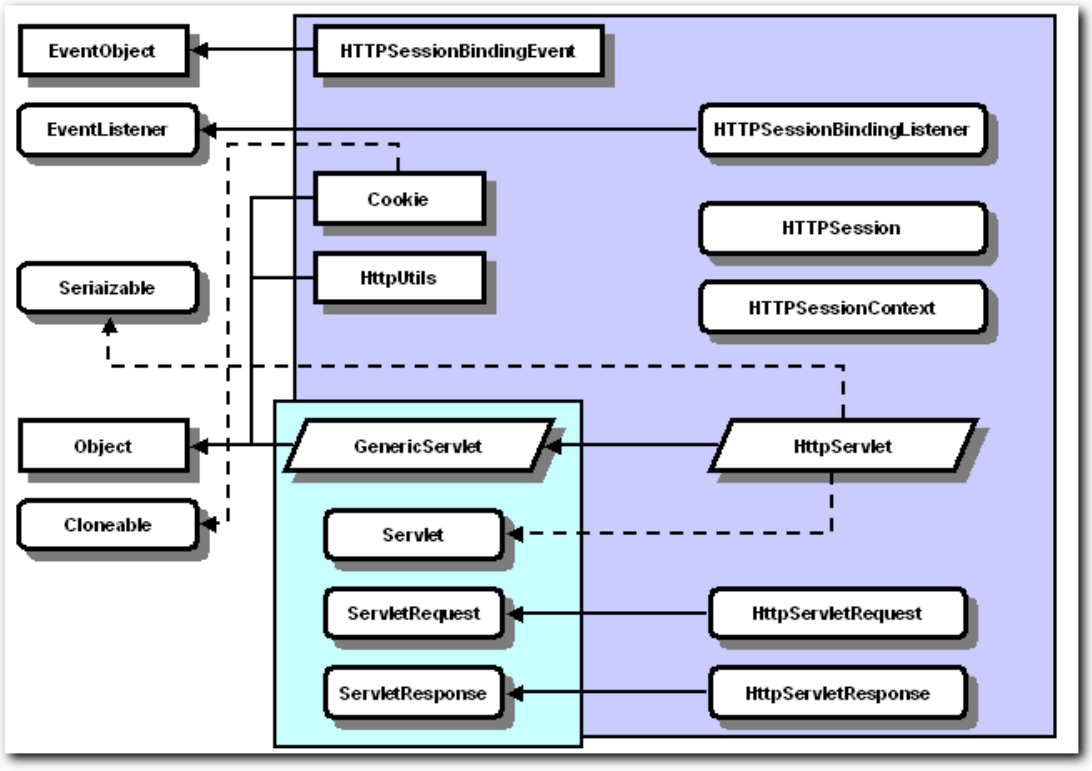


Figura 122: Il package javax.servlet.http

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 122* 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*.

15.4 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 123*).

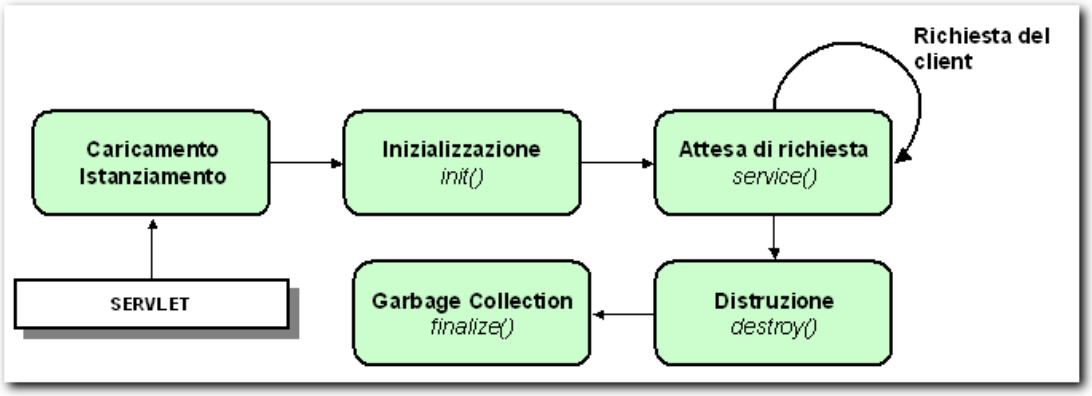


Figura 123: Ciclo di vita di una servlet

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* del package 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 della chiamata proveniente dal 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.

15.5 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 124)*.

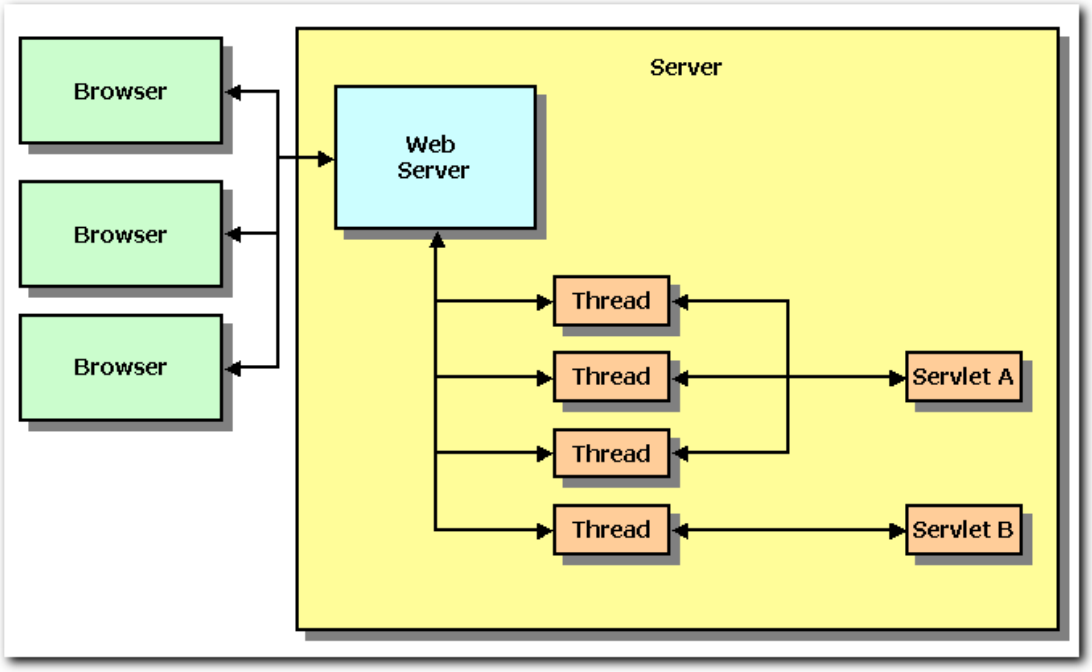


Figura 124: Accesso Concorrente alle Servlet

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 per cui, 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.

15.6 L'interfaccia SingleThreadModel

Quando un thread accede al metodo sincronizzato di una servlet, ottiene il lock (ossia l'esclusiva) 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 125*).

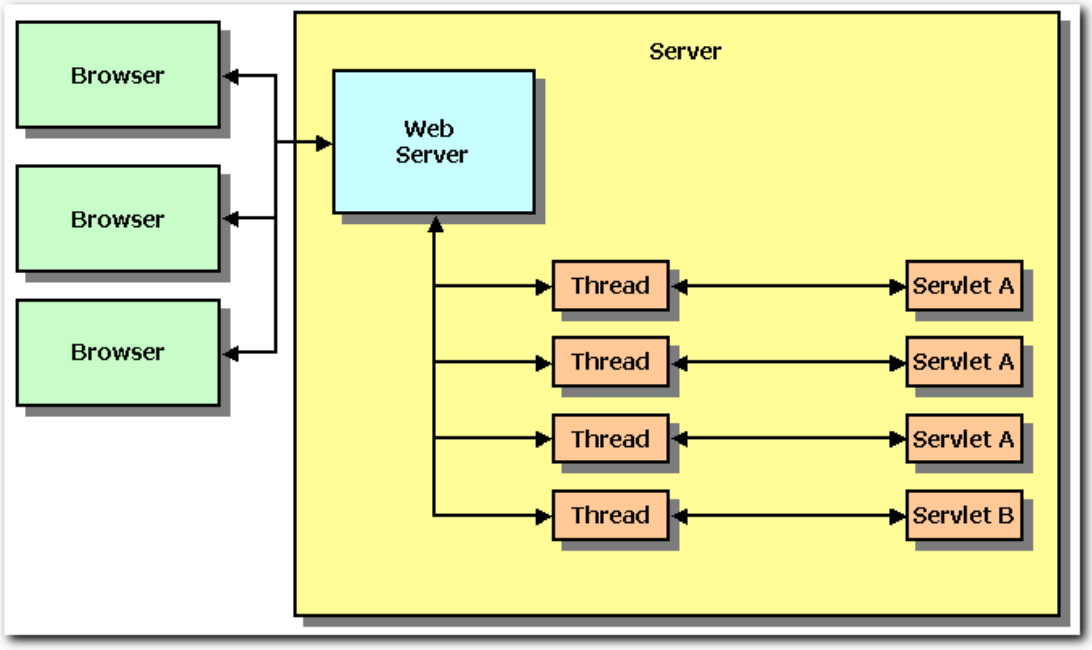


Figura 125: SingleThreadModel

15.7 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.

```
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>");
        }
}
```

15.8 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 126*). 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.

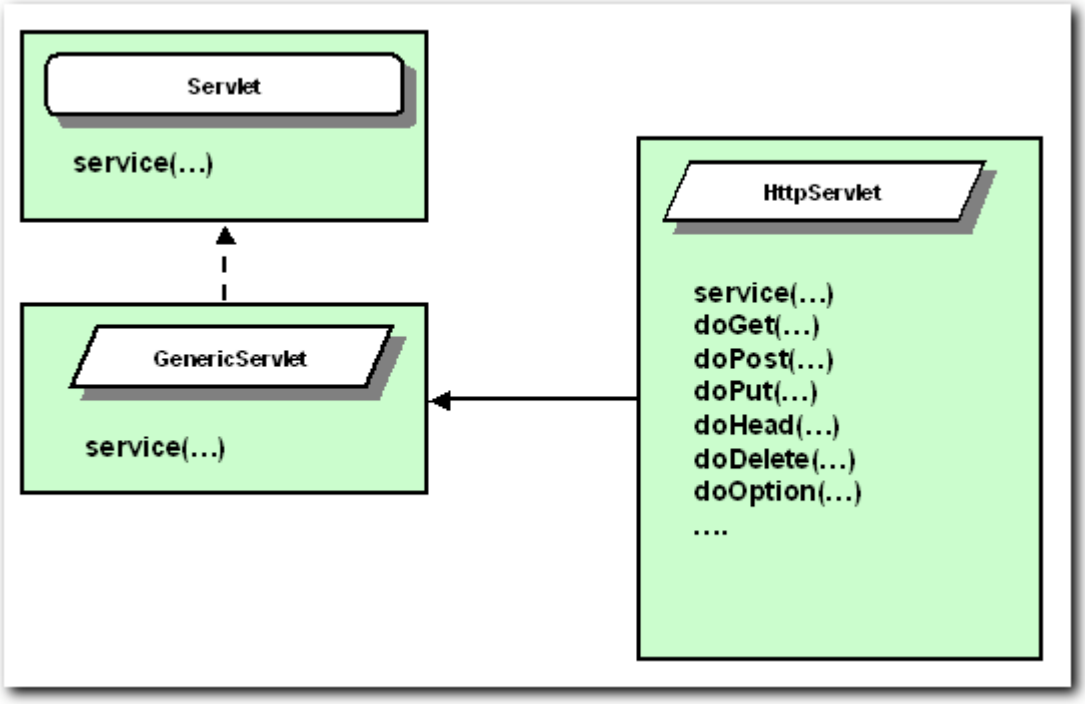


Figura 126: Il metodo service

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 almeno uno di questi metodi all'interno della nostra nuova classe. a seconda del tipo di richieste che dovrà esaudire (*Figura 127*).

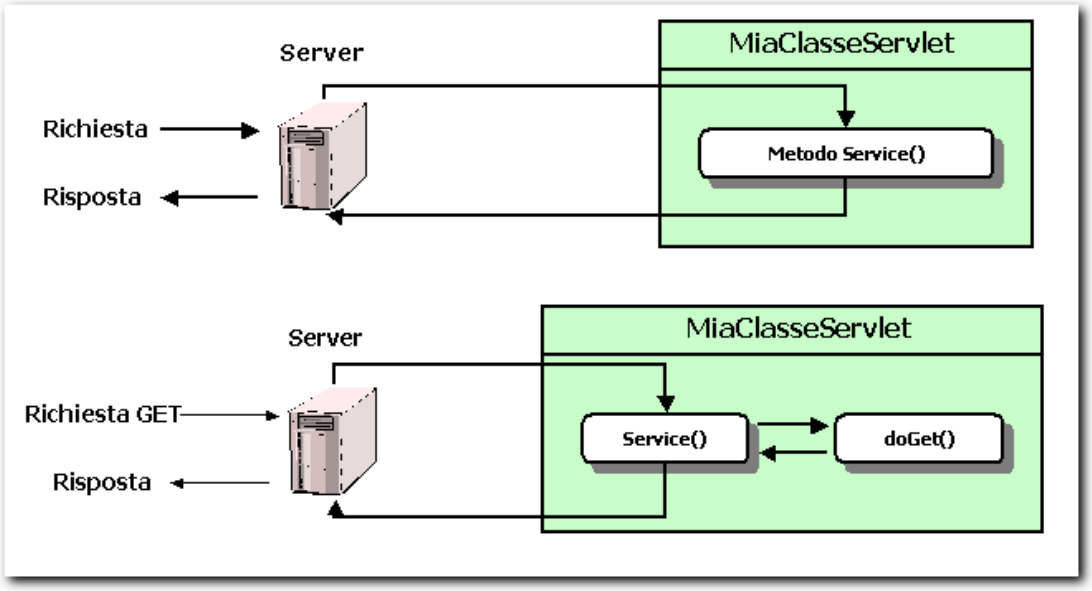


Figura 127: Il metodo Service

16 SERVLET HTTP

16.1 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.

16.2 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, da qui l'uso del protocollo come standard per i web server e le applicazioni Web based.
Il protocollo fu adottato nel 1990 dalla "World Wide Web Global Information Initiative" a partire dalla versione 0.9 come schematizzato nella tabella sottostante:

Il protocollo HTTP	
Versione	**Descrizione**
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: 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 128*. 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.

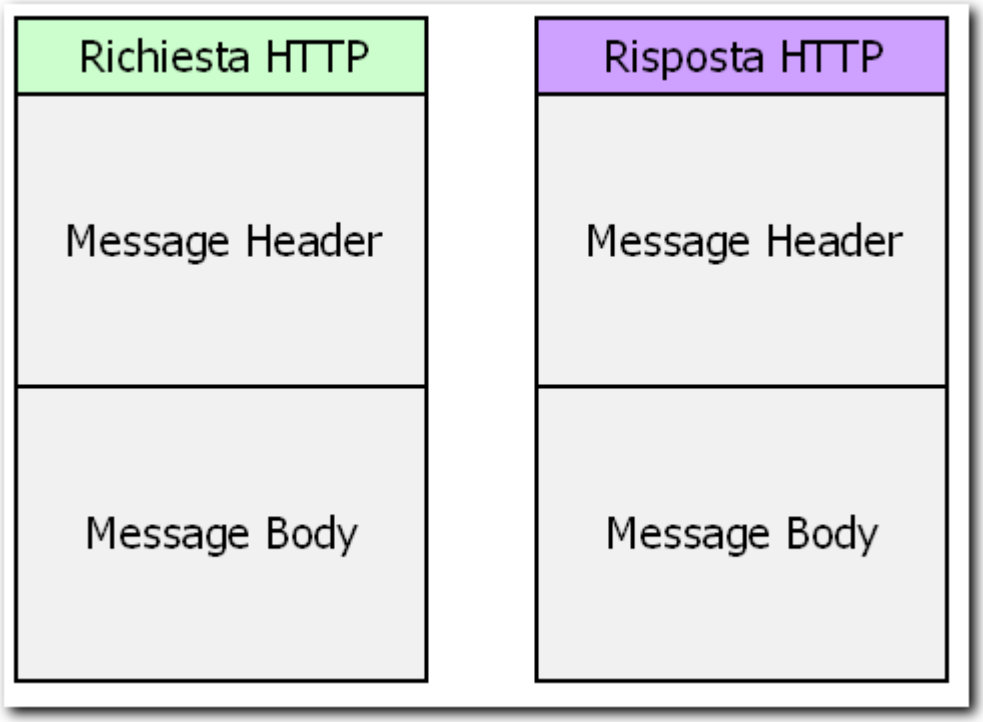


Figura 128: ***Struttura di http Request e Response***

Per dare una definizione formale del protocollo HTTP utilizzeremo la sintassi della forma estesa di Backus Naur che per semplicità indicheremo con l'acronimo BNF. Secondo la 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*

16.3 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 a 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.

16.4 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 elaborazione della richiesta da parte del server, 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

16.5 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 o opzionali. Il campo extension-header consente l'aggiunta di una nuove informazioni non previste, ma necessarie, senza apportare modifiche al protocollo.

16.6 I metodi di request

I metodi definiti dal campo request-method sono necessari al server per poter prendere decisioni sulle modalità di elaborazione 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.

16.7 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 *getInitParameterNames(),* 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 seguente, utilizziamo il metodo *getInitParameter(String)* per ottenere il valore del parametro di input con chiave "CHIAVE_PARAMETRO":

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>");
        }

}
```

16.8 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.

16.9 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.

16.10 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 così come definiti dallo standard HTTP. Il valore di queste costanti può essere utilizzato, mediante 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>");
        }

}
```

16.11 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, compresi 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.

16.12 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.it/servelt/Hello?nome=Massimo&cognome=Rossi

contiene la query string "*?nome=Massimo&cognome=Rossi*". Abbiamo inoltre accennato al fatto che, qualora il metodo utilizzato sia il metodo POST, i dati non vengano trasmessi in forma di query string ma vengono accodati nella sezione dati del protocollo HTTP.
Le regole utilizzare in automatico 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; 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.

16.13 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>.

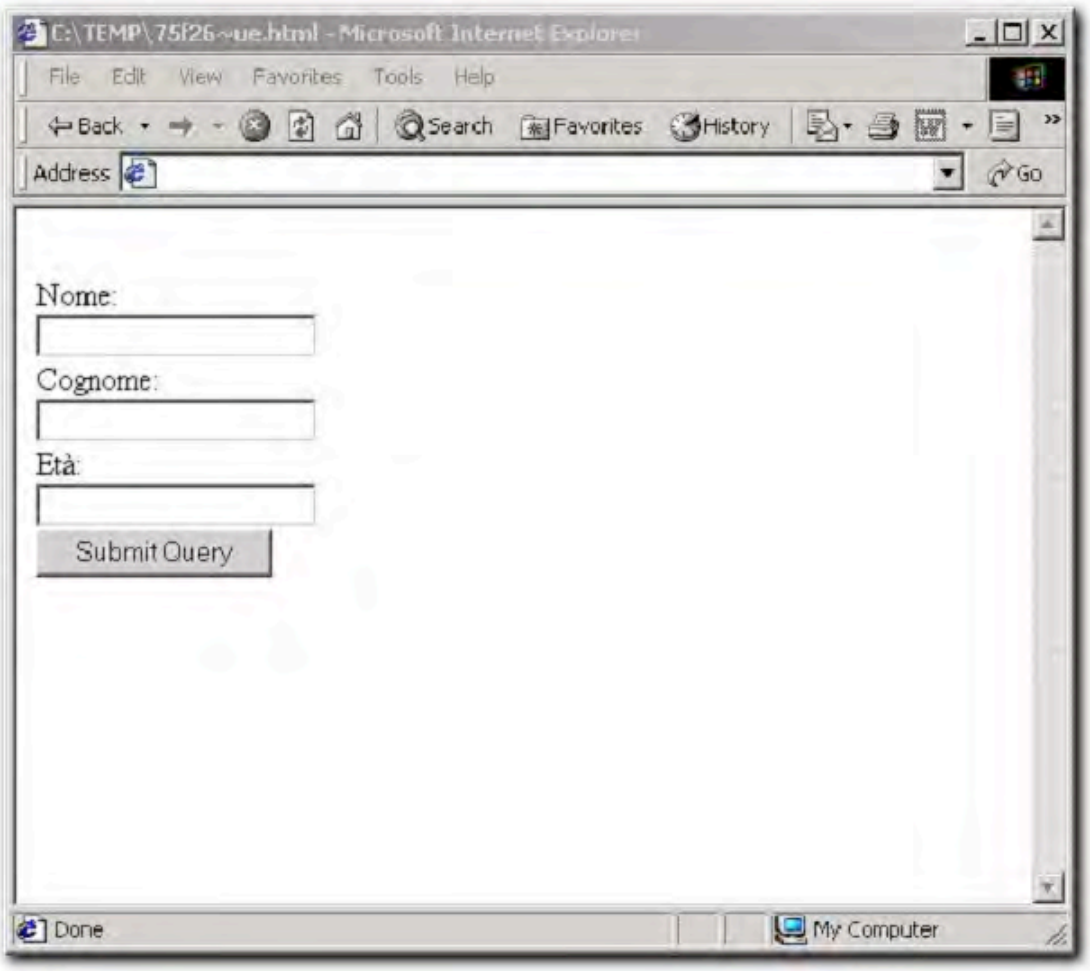


Figura 129: Un form HTML

Vediamo un form di esempio:

```
<html>
<body>
	<FORM method=GET  action=http://www.java-net.it/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 129* viene riportato il form così come il browser lo presenta all'utente.
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.java-net.it/Prova?param=nascosto&nome=nomeutente&cognome=cognomeutente&eta=etautente

16.14 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 al loro 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-net.it indicherà al browser che il cookie sarà valido solo per la macchina www all'interno del dominio java-net.it. 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.it/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.

16.15 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.

```
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);
}
```

16.16 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();
        }

}
```

16.17 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.

16.18 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 130*).

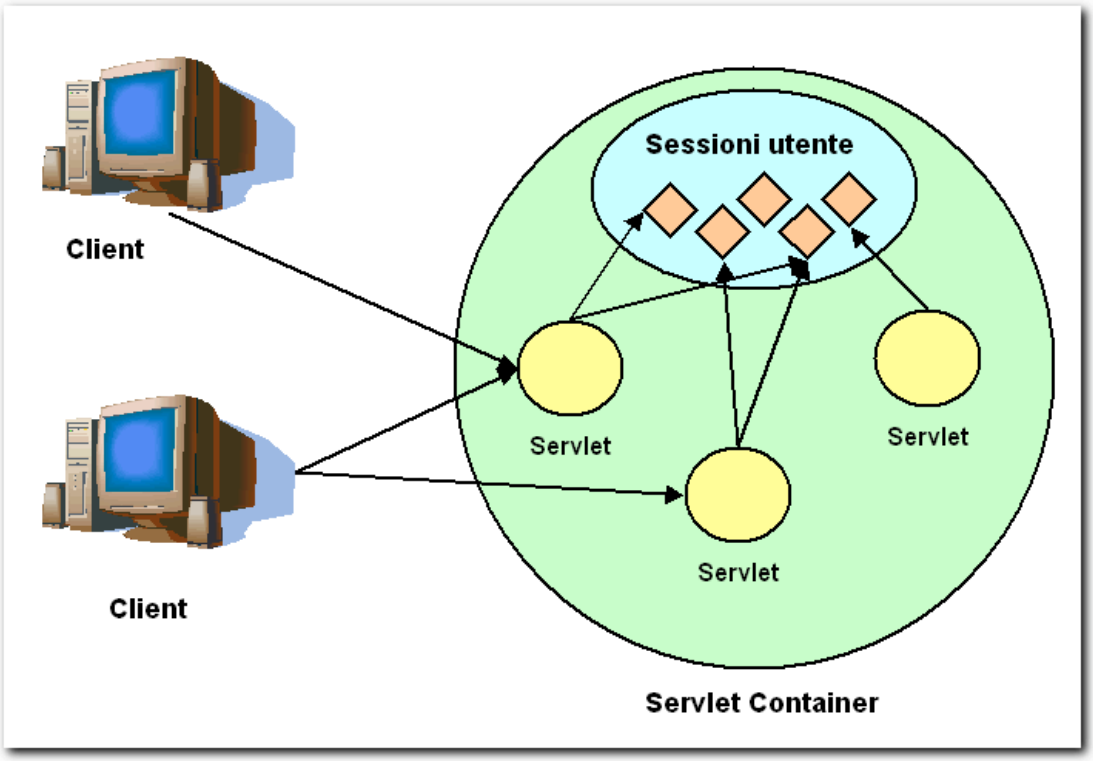


Figura 130: Le sessioni utente sono accessibili da tutte le servlet

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.

16.19 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* .

```
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

String 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 data ed ora. 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.

16.20 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();
        }

}
```

16.21 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.

16.22 URL rewriting

Il browser ha la facoltà di accettare o meno 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.

17 JAVASERVER PAGES

17.1 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.

17.2 JavaServer Pages

Una pagina JSP è un semplice file di testo che fonde codice HTML e 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, i quali saranno inseriti dal programmatore che non dovrà preoccuparsi dell'aspetto puramente grafico.
Da sole, JavaServer Pages consentono di realizzare applicazioni Web dinamiche (*Figura 131*) 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.

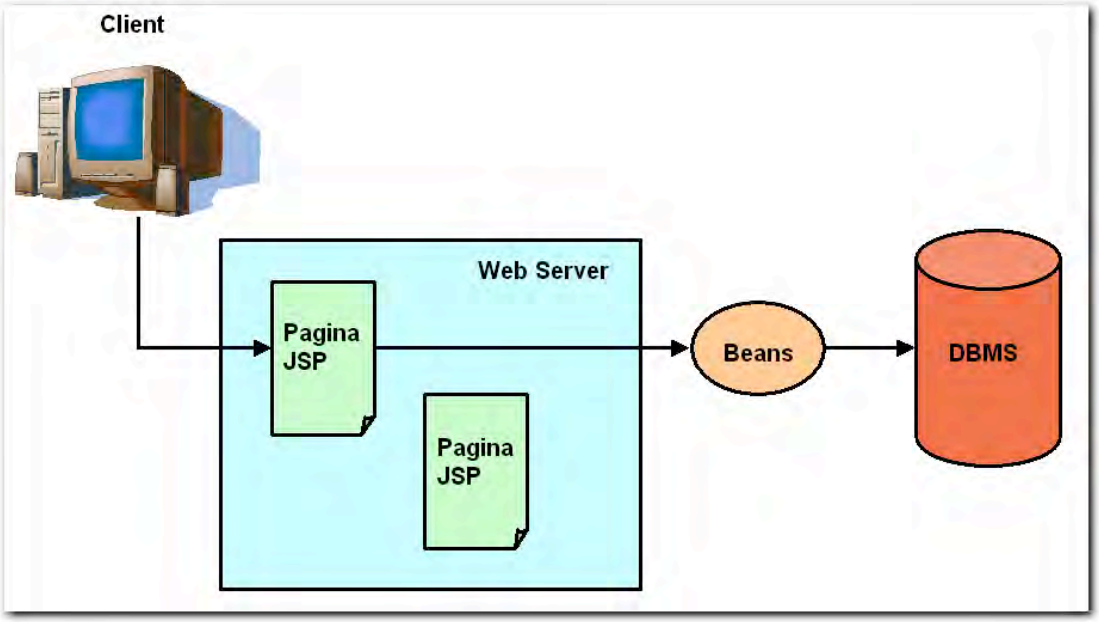


Figura 131: Primo modello di sistema

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 132*).

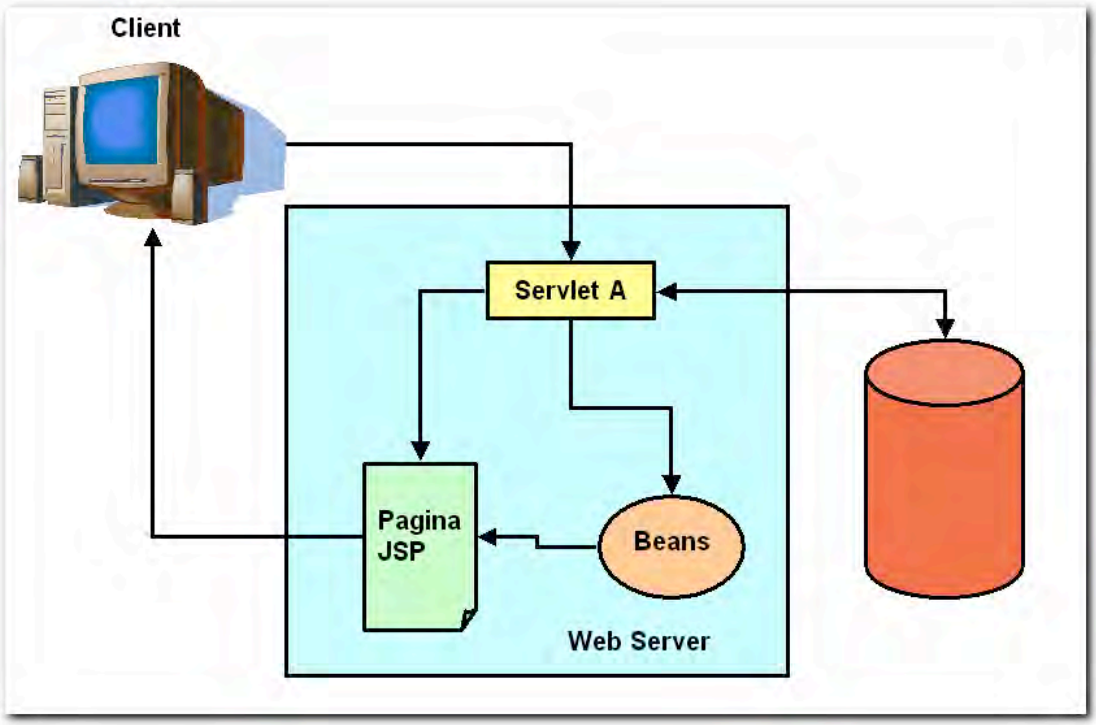


Figura 132: Secondo modello di sistema

In questo nuovo modello, il browser invia la sua richiesta ad una servlet che si preoccuperà di processare i dati di input, utilizzando eventualmente JDBC o interrogando speciali classi delegate al collegamento con 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.

17.3 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.

17.4 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:

```
<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".

```
<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 di spezzare il sorgente Java contenuto delle scriptlets per dar modo al programmatore di non dover fondere tag HTML all'interno del codice sorgente Java.

17.5 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:

1. *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;*
2. *JSP ereditano quindi tutti i meccanismi che il container mette a disposizione delle servlet, compreso 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*.

```
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());
                }
        }

}
```

18 JAVASERVER PAGES: NOZIONI AVANZATE

18.1 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:

TAG JSP		
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.

18.2 Direttive

Le direttive forniscono informazioni aggiuntive sull'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:

Direttive JSP	
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 stream 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 *SingleThreadModel* 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" %>
```

18.3 Dichiarazioni

Una dichiarazione rappresenta, dal punto di vista del container 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;
        }
%>
```

18.4 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 sette e sono i seguenti:

1. ***request***: *rappresenta la richiesta del client*
2. ***response***: *rappresenta la risposta del client*
3. ***out***: *rappresenta lo stream di output html inviato come risposta al client*
4. ***session***: *rappresenta la sessione utente*
5. ***page***: *rappresenta la pagina JSP attuale*
6. ***config***: *rappresenta i dettagli della configurazione del server*
7. ***pageContext***: *rappresenta un container per i metodi relativi alle servlet*

18.5 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 a 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:

```
<%@ 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>
```

18.6 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.

18.7 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.

19 JDBC

19.1 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.

19.2 Architettura di 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:

1. ***JDBC-ODBC bidge****: bridge jdbc/odbc che utilizzano l'interfaccia ODBC del client per connettersi alla base dati;*
2. ***JDBC Native Bridge****:ha sempre funzioni di bridge (ponte), ma traduce le chiamate JDBC in chiamate di driver nativi già esistenti sulla macchina;*
3. ***JDBC Net Bridge****: ha una architettura multi-tier ossia si connette ad un RDBMS tramite un middleware connesso fisicamente al database e con funzioni di proxy;*
4. ***Driver JDBC Java****: 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.

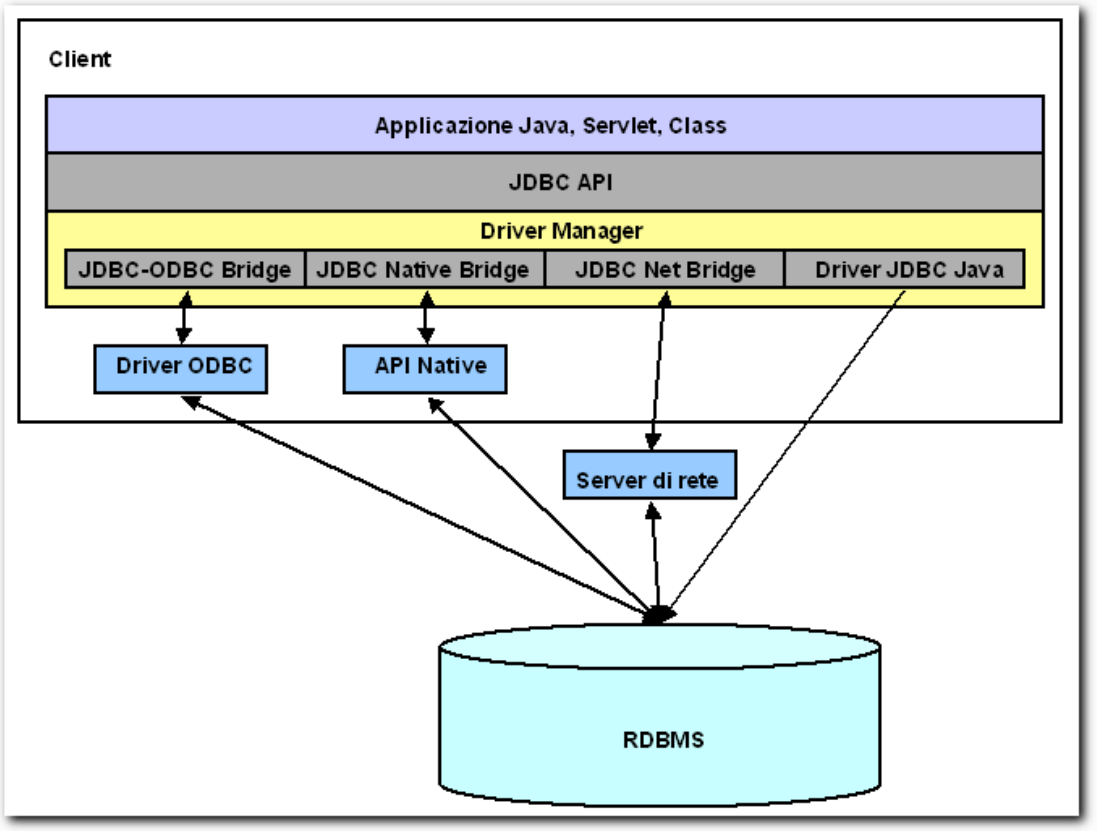


Figura 133: Architettura JDBC

19.3 Driver di tipo 1

Rappresentati dal bridge jdbc/odbc di SUN, utilizzano l'interfaccia ODBC del client per connettersi alla base dati (*Figura 134*).
Questo tipo di driver sono difficili da amministrare e dipendono dall'implemetazione ODBC e dalle piattaforme Microsoft. Il fatto di utilizzare meccanismi intermedi per connettersi alla base dati (lo strato ODBC) fa si che non rappresentino la soluzione ottima per tutte le piattaforme in cui le prestazioni del sistema rappresentano l'aspetto critico (il mapping JDBC – ODBC è complesso ed oneroso in termini di prestazioni).

19.4 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 135*). 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.

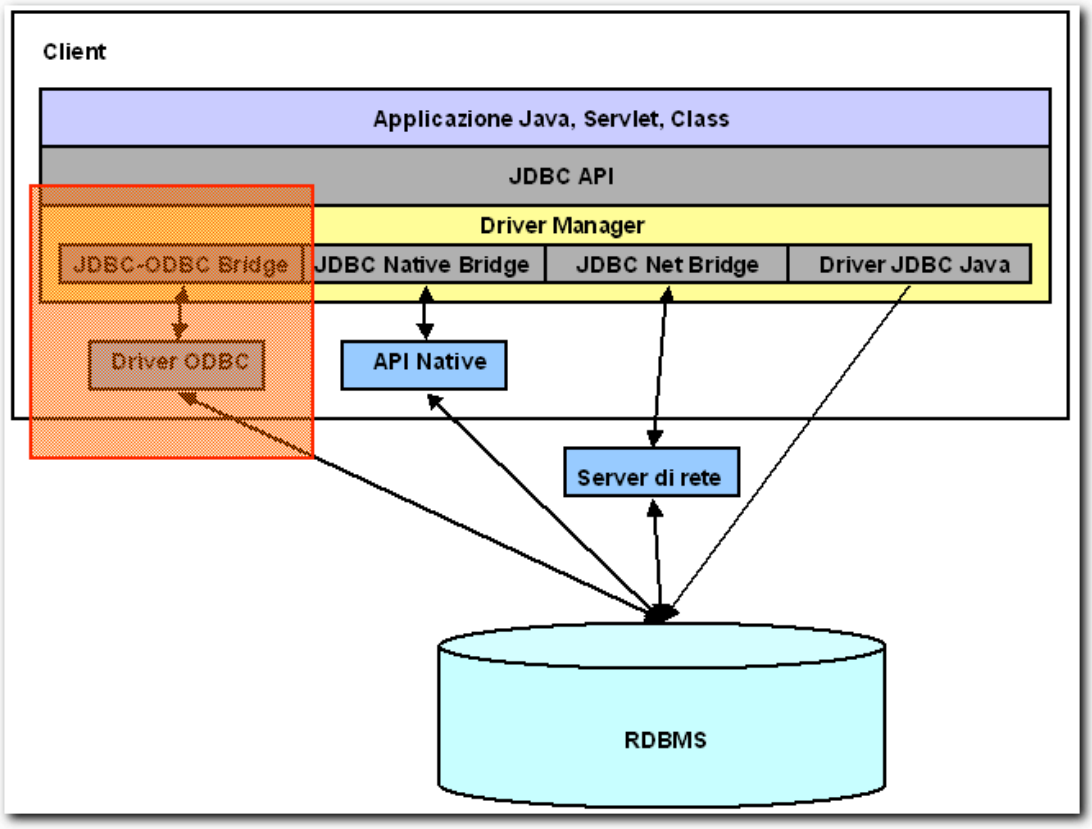


Figura 134: Driver JDBC Tipo 1

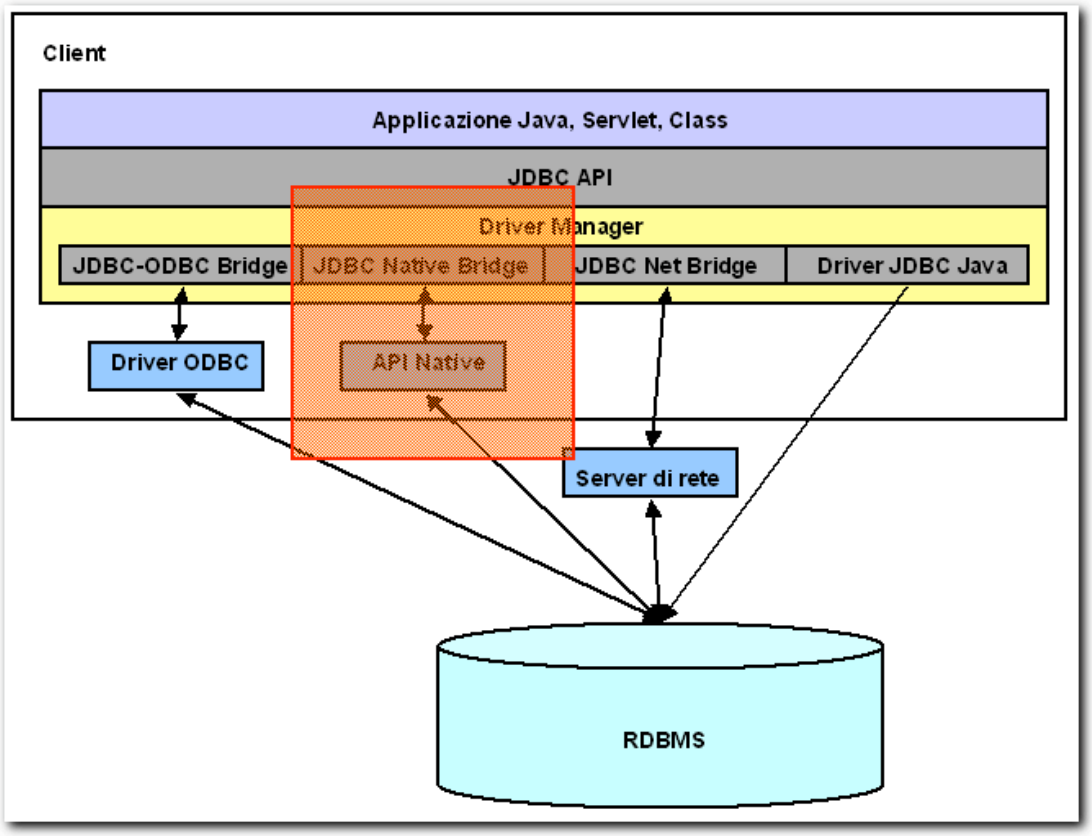


Figura 135: Driver JDBC tipo 2

19.5 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 (*Figura 136*).
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, e 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.

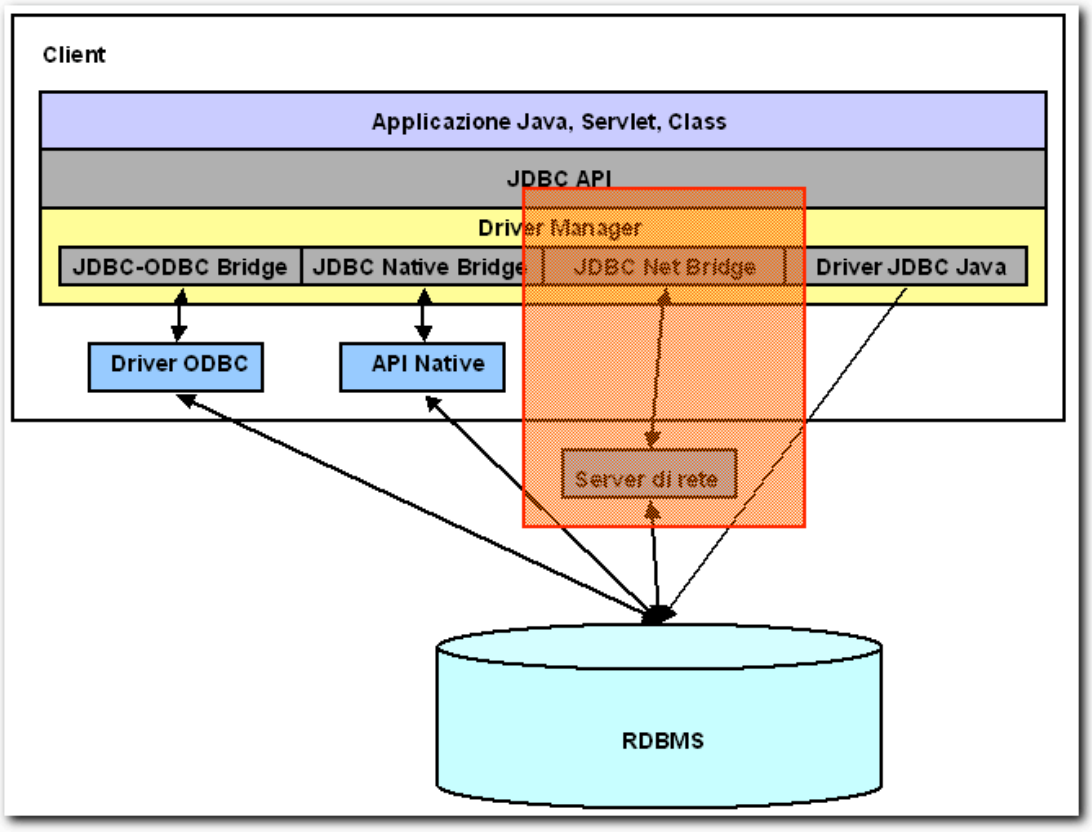


Figura 136: Driver JDBC Tipo 3

19.6 Driver di tipo 4

Un driver di tipo quattro fornisce accesso diretto ad un database ed è un oggetto Java completamente serializzabile (*Figura 137*).
Questo tipo di driver possono funzionare su tutte le piattaforme e dal momento che sono serializzabili possono essere scaricati dal server.

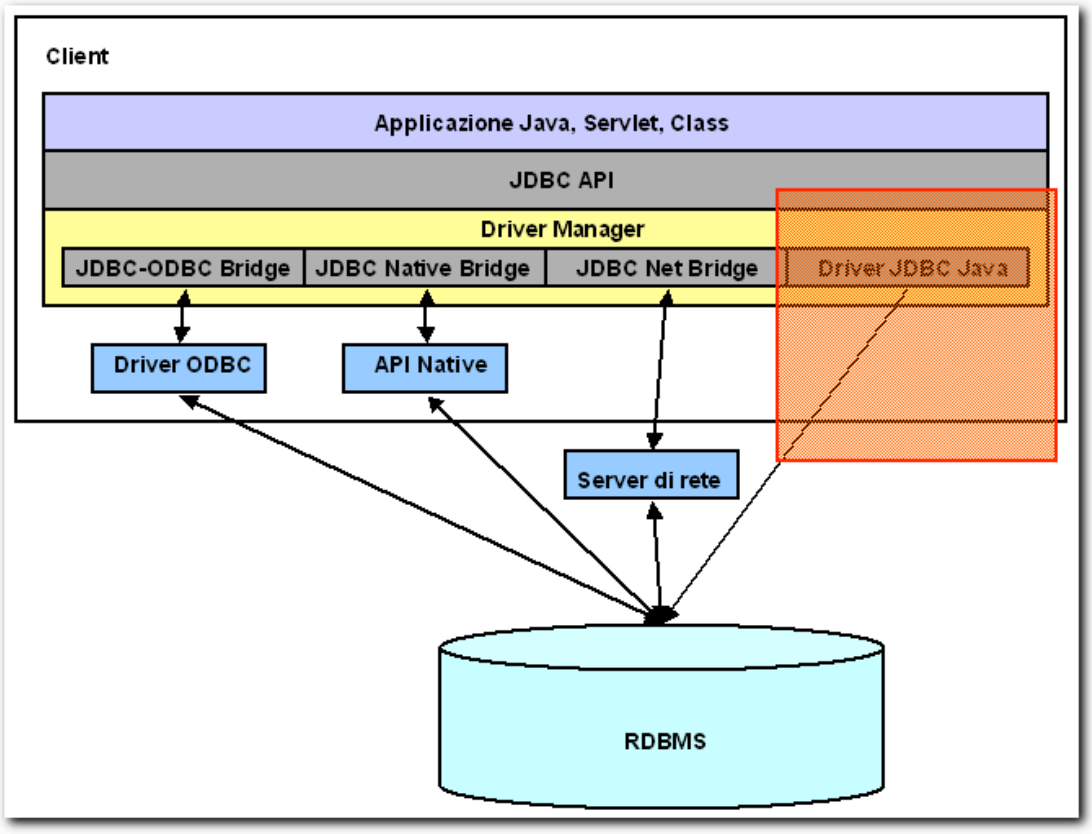


Figura 137: Driver JDBC Tipo 4

19.7 Una prima applicazione di esempio

Prima di scendere nei dettagli della tecnologia JDBC, ci soffermeremo su una prima applicazione di esempio. L'applicazione usa driver JDBC di tipo 1 per connettersi ad un database access contenente una sola tabella chiamata impiegati, come mostrata nella *Figura 138*.

Figura 138: Tabella Access

```
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 se){
			System.out.println(se.getMessage());
			se.printStackTrace(System.out);
			System.exit(1);
		}
	}
}
```

Dopo aver caricato il driver JDBC di tipo 1 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 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*.

19.8 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* 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.

19.9 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.

19.10 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 IMPIEGATI" produrrà un ResultSet contenente due colonne identificate, rispettivamente, dalla stringa "NOME" e dalla stringa "COGNOME" (*Figura 139*). 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 IMPIEGATI" le colonne del ResultSet saranno identificato de un numero intero a partire da 1 da sinistra verso destra (*Figura 140*).

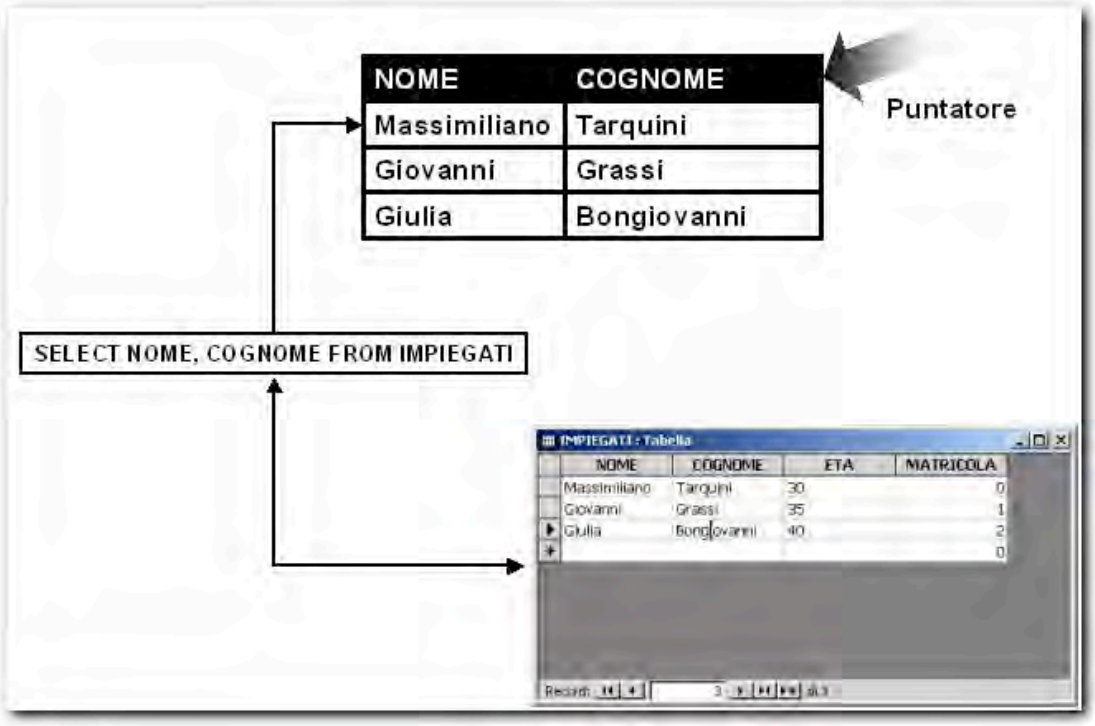


Figura 139: Esempio di SELECT con attributi definiti

```
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);
                }
        }
}
```

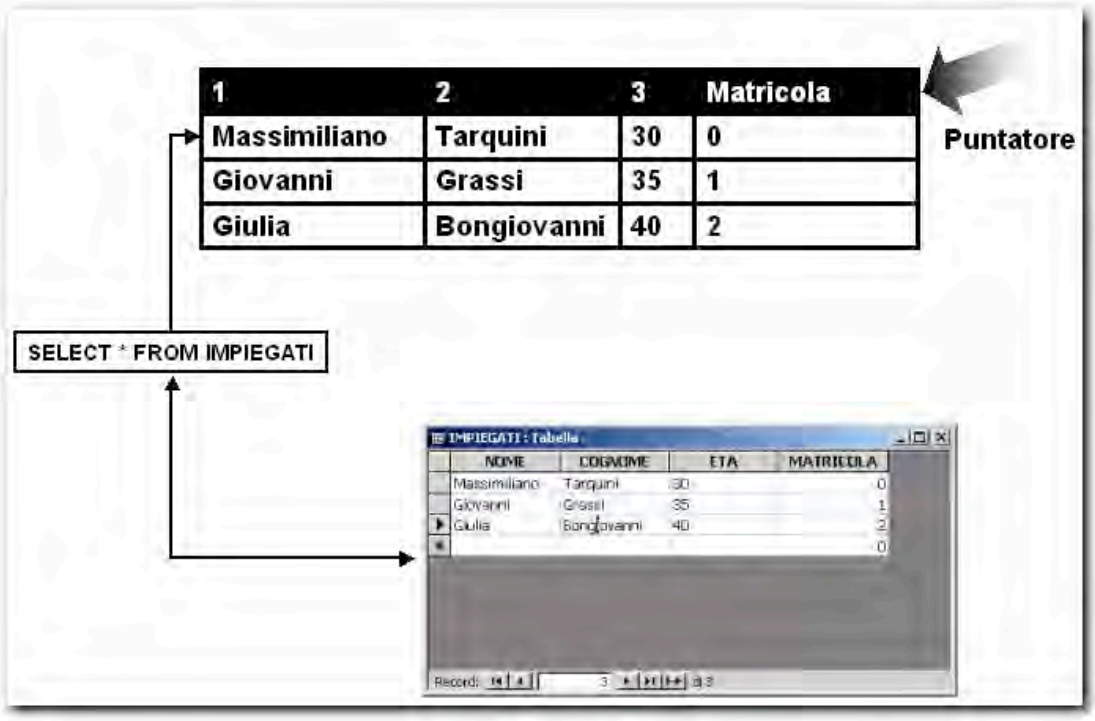


Figura 140: Esempio di SELECT *.

```
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);
                }
        }
}
```

20 REMOTE METHOD INVOCATION

20.1 Introduzione

La Remote Method Invocation (RMI) è il primo strumento nativo Java per la creazione di software distribuito.
La sua introduzione, risalente alla versione 1.1 del linguaggio, è stata effettuata per risolvere il problema di far comunicare tra loro oggetti non residenti sulla stessa macchina. Prima dell'introduzione di RMI, il linguaggio Java metteva a disposizione, come unico strumento, la classica comunicazione basata sui socket. Questo meccanismo, derivante dalle architetture client-server e pertanto funzionale e notevolmente conosciuto, mal si sposa con la filosofia Object Oriented, dal momento che, nel caso si voglia invocare un metodo di un oggetto remoto, l'unico sistema è quello di istaurare tra i due sistemi un protocollo applicativo di comunicazione basato su contenuti. Questo fatto accoppia notevolmente le implementazioni del client e del server, oltre al fatto che per gestire correttamente i socket è necessaria una certa cautela.
Il principale vantaggio di RMI risiede nella caratteristica di essere una implementazione 100% Java, e quindi utilizzabile da qualsiasi oggetto o componente scritto in Java, del più noto e vasto capitolo delle architetture distribuite.
La tecnologie introdotta in RMI, inoltre, rappresenta la base del paradigma Java per oggetti distribuiti, e pertanto è utilizzata anche in altri elementi dell'architettura J2EE, in particolare negli Enterprise JavaBeans.
Il rovescio della medaglia è rappresentato dal fatto che questo vantaggio si configura come un limite nei casi in cui sia necessario integrare oggetti distribuiti scritti in linguaggi arbitrari.
Nella versione 1.3 del linguaggio Java RMI è stato modificato al fine di integrare, come strumento di trasporto, il protocollo IIOP (Inter Internet ORB Protocol) nativo dell'architettura CORBA e maggiormente aderente agli standard internazionali. Per questo motivo dalla versione 1.3 in poi RMI diventa RMI-IIOP.
In questo capitolo verrà analizzato RMI come strumento per la realizzazione di architetture distribuite e sarà effettuata una rapida panoramica illustrativa sui principali aspetti.

20.2 Architetture distribuite

Il concetto di architettura distribuita è basato sulla possibilità di invocare metodi di oggetti dislocati su una macchina qualsiasi presente in rete, come se tali oggetti fossero "locali" al sistema chiamante; ciò è possibile grazie alla separazione delle interfacce client e server, secondo lo schema di cui in figura seguente:

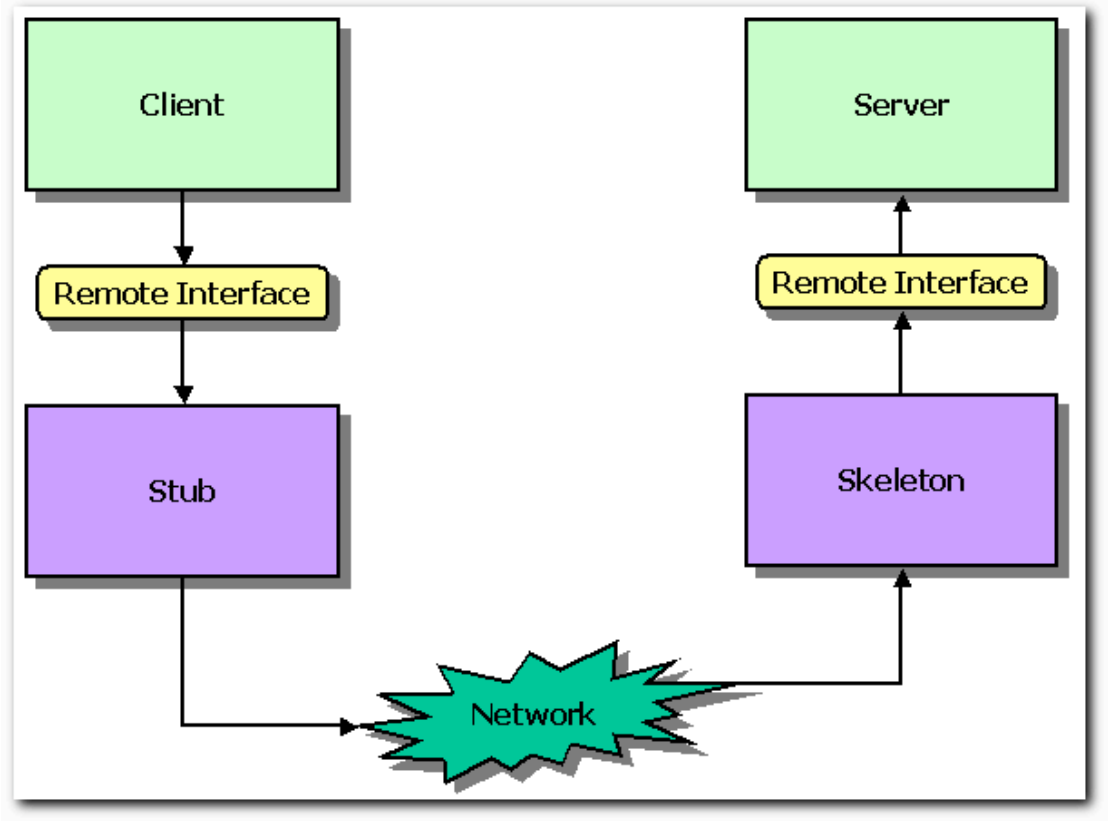


Figura 141: Schema base di architettura distribuita

Il paradigma si caratterizza per la distinzione tra oggetti (indicati, in figura, con client e server), contenenti la logica applicativa, e le interfacce che essi espongono verso i sistemi remoti. L'obiettivo è quello di implementare la *location trasparency*, per cui il client non ha necessità di conoscere i dettagli implementativi dell'oggetto server, né la macchina ove risiede, né la piattaforma e neanche come raggiungerlo. Al client è necessario conoscere solo il nome dell'oggetto da chiamare e la sua interfaccia, mentre i servizi di ricerca dell'oggetto fisico e di verifica della congruenza del protocollo sono delegati ad un apposito strato software, denominato *middleware*.
In realtà, per far sì che tutto funzioni, le chiamate remote debbono essere, in qualche modo, "strutturate" secondo un modello comune. L'interfaccia remota del server è inclusa nello *stub* che rappresenta, per il client, una finta istanza locale dell'oggetto remoto. Il client, grazie alla presenza dello stub, vede l'oggetto server come locale, e pertanto ne invoca i metodi pubblici definiti nella sua interfaccia. Lo stub si preoccupa di passare la chiamata sulla rete, dove viene intercettata dallo *skeleton* che si preoccupa di ricostruire la semantica della chiamata e passarla, per l'esecuzione, all'oggetto server vero e proprio. Sono, quindi, lo stub e lo skeleton ad effettuare il vero lavoro di disaccoppiamento e distribuzione dei sistemi, ma, per fortuna, tali elementi sono invisibili al programmatore, che non si deve preoccupare del loro

sviluppo; è il middleware a crearli sulla base della definizione di interfaccia remota.
Questo schema è comune a tutti gli standard di architetture distribuite, compreso, ad esempio, CORBA. Come detto, la specifica J2EE per java 1.3 e successive versioni, utilizza l'implementazione RMI-IIOP che utilizza come protocollo di trasporto lo standard CORBA.

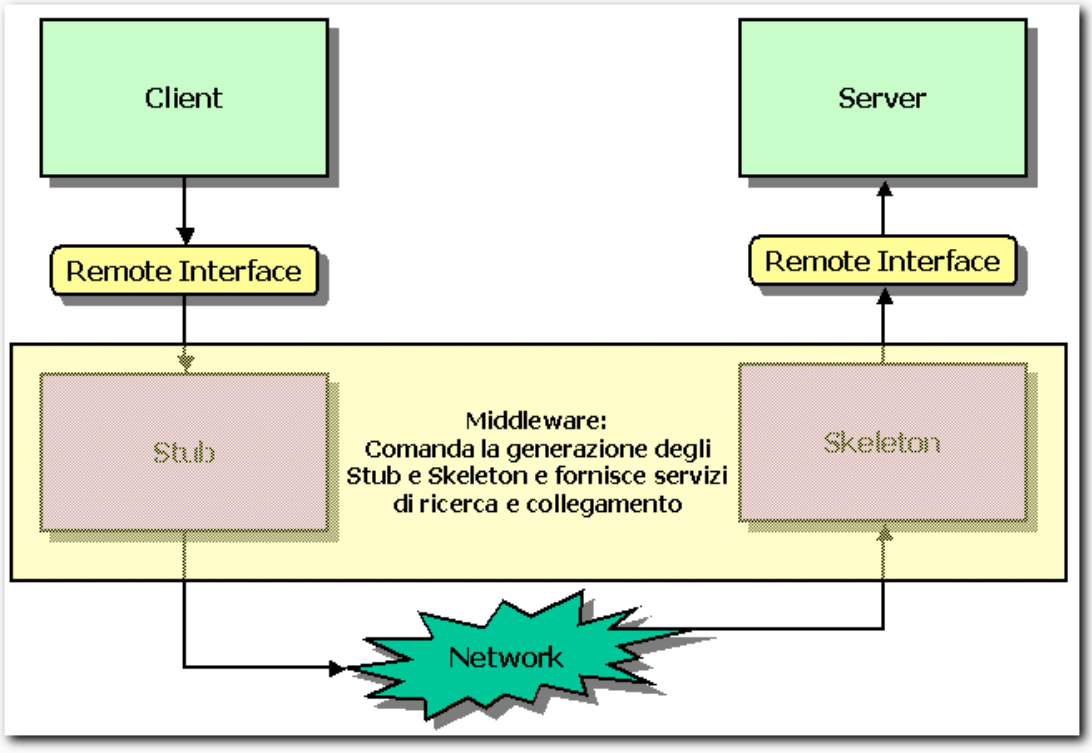


Figura 142: Ruolo del middleware in un'architettura distribuita

20.3 RMI in dettaglio

Facendo riferimento allo schema di architettura distribuita definito precedentemente, RMI non è che una delle possibili implementazioni, e sfrutta una serie di servizi (ed API) appositamente introdotte.
Lo schema generale è presentato nella prossima figura.
Rispetto alle figure precedenti, nella *figura 143* è evidenziato il ruolo svolto da RMI come middleware per i servizi di naming e come elemento applicativo caratterizzante lo strato di trasporto (a livello di protocollo di rete, quello utilizzato è sempre il TCP/IP).

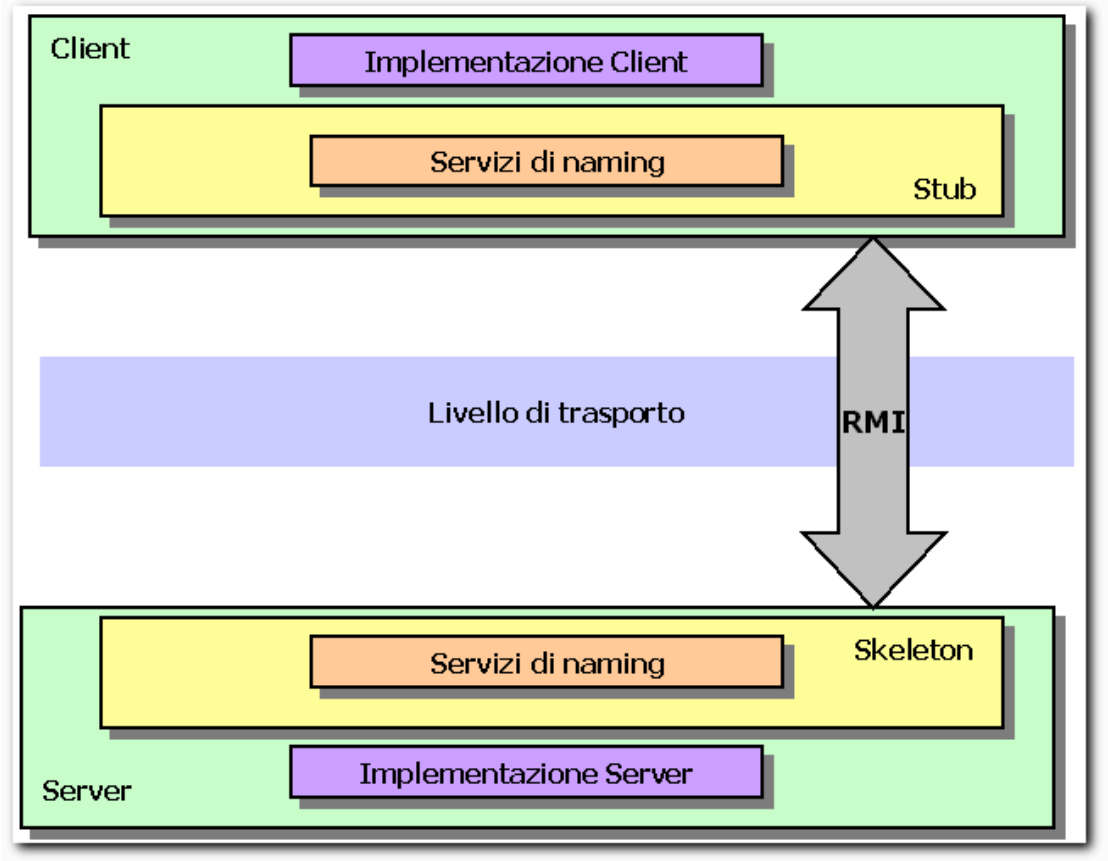


Figura 143: Architettura di RMI

Per poter riconoscere un oggetto presente in un punto qualsiasi della rete, è necessario che vi sia un servizio dedicato, contenente dei meccanismi per l'individuazione univoca dell'oggetto stesso.
Questa esigenza è nota in tutti i protocolli, e ciascun protocollo ha sviluppato i propri meccanismi per:

1. *individuare univocamente i computer, attraverso formalismi standard riconosciuti dai dispositivi delegati a gestire l'instradamento delle comunicazioni e a gestire le liste dei nomi*
2. *fornire informazioni per distinguere i vari servizi*

Ad esempio, per individuare un computer su una rete il meccanismo più semplice utilizzato nel protocollo TCP/IP è quello basato sull'indirizzo IP. Questo indirizzo individua i computer su scala gerarchica attraverso un indirizzo composto da quattro byte; per indicare comunemente questo indirizzo ogni byte viene separato dal successivo da un punto (ad esempio, 127.0.0.1). A questo primo livello di indirizzamento si è aggiunto il servizio (offerto dai DNS - Domain Name Service) che individua il computer con un nome logico con il seguente modello:

nomemacchina.nomesottodominio. … .nomedominio.suffissoIP

Ad esempio, la macchina di nome "antares" sul dominio miodominio.it sarà individuata dal nome antares.miodominio.it
A questo nome è possibile, secondo le specifiche TCP/IP, associare:

1. *un suffisso indicante il "servizio", cui corrisponde la porta sulla quale risponde l'applicativo; ad esempio, riferito al caso precedente, se abbiamo un servizio in ascolto sulla porta 5600 l'indirizzo per interrogare quel servizio è antares.miodominio.it:5600*
2. *un'identificativo del tipo di protocollo utilizzato a livello applicativo: ad esempio, nel caso del protocollo HTTP l'identificativo maggiormente usato è WWW ed un valido indirizzo è http://www.miodominio.it:8080; nel caso, invece, di protocollo FTP un indirizzo valido è ftp://ftp.miodominio.it (in questo caso, non indicando la porta, si utilizza quella di default).*

Questo modo di gestire i nomi consente di individuare in modo univoco qualsiasi host di rete, il suo servizio ed il protocollo applicativo, consentendo di implementare dei servizi di registrazione dei nomi e dei servizi, sia a livello di singolo computer sia di rete locale o internet.
RMI definisce, in modo del tutto analogo agli esempi precedenti, il protocollo associato all'indirizzo del computer da ricercare con la forma

protocol://nomecomputer:porta

e fornisce una utilità per registrare la lista degli oggetti che richiedono i servizi RMI attraverso il tool *rmiregistry* presente nel SDK.

20.4 Capire ed usare RMI

Al fine di comprendere meglio l'utilizzo di RMI, è opportuno avvalerci di un esempio.
Per utilizzare in modo corretto RMI, la prima cosa da fare è definire qual è l'interfaccia esportata dall'oggetto che deve essere chiamato da remoto.
Supponiamo, come esempio, di voler implementare la comunicazione fra due oggetti, di cui quello server supponiamo sia una libreria, che esporta un metodo che ritorna, a partire dal titolo di un libro, la lista degli autori.
Il codice dell'interfaccia, che chiameremo "Libreria", è presentato nel listato seguente

```
import java.rmi.Remote;
```

```
public interface Libreria extends Remote {
        public String nomiDegliAutori(String libro) throws java.rmi.RemoteException;
}
```

Come si nota, l'interfaccia estende una classe specifica di RMI, l'interfaccia *java.rmi.Remote*. Tale interfaccia è necessaria per definire le interfacce che possono essere invocate da una Java Virtual Machine non locale all'implementazione dell'oggetto.
Il metodo invocabile da remoto, inoltre, deve necessariamente emettere una eccezione, la *java.rmi.RemoteException*, che è la superclasse delle eccezioni del package *java.rmi* e comprende tutte le eccezioni che possono intervenire durante la comunicazione remota.
La classe di gestione della libreria (che supponiamo contenere le istanze ai libri) implementa la suddetta interfaccia, ed il codice è descritto di seguito.

```
import java.util.*;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class LibreriaServer extends UnicastRemoteObject implements Libreria {
        Hashtable libri;

        // Implementazione del metodo invocable da remoto
        public String nomiDegliAutori(String titolo) throws RemoteException{
                String autori;
                autori = (String)libri.get(titolo);
                if (autori== null) return ("Libro non trovato");
                else return autori;
        }

        // Costruttore della classe
        public  LibreriaServer() throws RemoteException{
                libri = new Hashtable();
                libri.put("La Divina Commedia","Dante Alighieri");
                libri.put("Il Decameron","Giovanni Boccaccio");
                libri.put("I Promessi Sposi","Alessandro Manzoni");
        }

        public static void main(String[] args) {
                try {
                        LibreriaServer rmiServer = new LibreriaServer();

                        // Con questo metodo vediamo qual è il registro
                        // degli oggetti della macchina locale
                        Registry registroDeGliOggetti = LocateRegistry.getRegistry();

                        // Registro l'oggetto sul registro degli oggetti locale
                        registroDeGliOggetti.rebind("LaMiaLibreria", rmiServer);
                        System.out.println("Classe LibreriaServer registrata e pronta a ricevere
chiamate.");
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }
```

```
}
```

La classe LibreriaServer, per poter esser un oggetto invocabile da remoto via RMI, deve estendere la superclasse *java.rmi.server.UnicastRemoteObject*. Questa classe definisce il comportamento delle classi che la estendono in termini di referenza univoca (la referenza all'oggetto è unica e valida solo quando l'oggetto è attivo) e possibilità di essere invocate secondo il modello point-to-point.
Il metodo nomiDegliAutori è corretto dal momento che ritorna un oggetto, *String*, che implementa in modo nativo l'interfaccia *java.io.Serializable*, e questo garantisce la correttezza della trasmissione dei dati sullo stream TCP (di questo parleremo più avanti).
La classe LibreriaServer contiene anche un metodo main() utilizzato per registrare l'oggetto sul registry dei nomi.
Per la comunicazione con il registry, viene utilizzata la classe *java.rmi.Registry*, ed i metodi per la registrazione dell'oggetto sono *bind* (con dichiarazione univoca della classe) e *rebind* (qualora si voglia dinamicamente modificare il registry). Entrambi i metodi hanno come parametri una stringa, che rappresenta il nome logico dell'oggetto da invocare (in questo caso, *LaMiaLibreria*), e la referenza all'oggetto stesso. Il nome logico può essere scelto arbitrariamente dall'utente e rappresenta il nome logico del servizio cui cercherà di connettersi il client.
Qualche ulteriore osservazione sul metodo *main*:

1. *il meccanismo di registrazione in esso dichiarato non deve risiedere necessariamente all'interno nella classe server, anzi, qualora si abbiano più classi da registrare, è possibile dichiararlo in un'altra classe che si occupi solo della registrazione degli oggetti sul server;*
2. *il secondo obiettivo perseguito dal meccanismo di registrazione) ed in particolare dai metodi bind) è quello di mantenere vivo e disponibile il servizio per le chiamate: questo implica che la chiamata al bind mette il programma in uno stato di ascolto sulla porta definita dal registry (generalmente la 1099).*

Il registry dei nomi fornito dal SDK è la già citata utilità rmiregistry, presente nella directory bin di installazione dell'SDK, che andrà lanciata prima di eseguire la classe LibreriaServer.
Una volta scritta e compilata la classe, è necessario generare i componenti Stub e Skeleton. Per questo compito è presente, fra gli eseguibili del SDK, l'utility *rmic*, la cui invocazione, per la classe LibreriaServer, genera i file LibreriaServer_Stub.class e LibreriaServer_Skel.class.
A questo punto è possibile scrivere il codice di un possibile programma client che, prelevando il nome del libro di cui cercare gli autori dalla riga di

comando, invoca in modo remoto il metodo nomiDegliAutori e stampa l'autore.

```
import java.rmi.Naming;
import Libreria;

public class Client {
        public static void main(String[] args) {
        try {
                // Ricerca dell'oggetto LibreriaServer
                Libreria miaCopiaVirtualeOggettoServer=
                        (Libreria)Naming.lookup("rmi://nomeDelServer/LaMiaLibreria");

                String autori = miaCopiaVirtualeOggettoServer.nomiDegliAutori(args[0]);

                // Stampa del risultato.
                System.out.println("Sulla libreria server gli autori del libro " + args[0] + " sono: "
                                + autori);
        }
        catch (Exception e) {
                System.out.println("Error while looking up account:");
                e.printStackTrace();
        }
 }
}
```

Si noti, innanzitutto, come il client per referenziare ed utilizzare l'oggetto remoto non abbia bisogno del codice dello stesso, ma solo della sua interfaccia ("import Libreria").
La parte più importante del codice è quella della riga:

```
Libreria miaCopiaVirtualeOggettoServer =
                (Libreria) Naming.lookup("rmi://nomeDelServer/LaMiaLibreria");
```

La classe Client "vede" LibreriaServer mediante la sua interfaccia, e ne istanzia una copia virtuale (miaCopiaVirtualeOggettoServer) attraverso i servizi della classe *java.rmi.Naming*.
Il metodo *lookup(String)* ritorna lo Stub dell'oggetto remoto, sulla base del nome dichiarato nella forma:

rmi://nomecomputer:porta/nomeOggettoRemoto

L'oggetto ritornato dal metodo *lookup* è un generico *java.rmi.Remote* e quindi, per poterlo utilizzare, è necessario effettuarne il cast al tipo richiesto (nel caso in oggetto, Libreria).
Dopo il casting, l'oggetto ottenuto *miaCopiaVirtualeOggettoServer* è un riferimento locale all'oggetto server la cui istanza risiede sulla macchina remota; di questo oggetto possono essere invocati i metodi esposti nella sua interfaccia (nel nostro caso il metodo *nomiDegliAutori*).

20.5 L'evoluzione in RMI-IIOP

L'esempio precedentemente esposto è riferito alla specifica 1.1 di Java (quindi RMI); per poterlo far evolvere verso la 1.2 e successive (RMI-IIOP) è necessario effettuare alcune modifiche.
Per quanto riguarda la definizione dell'interfaccia non ci sono modifiche.
Di seguito è presentato il listato di LibreriaServer.java, ove in grassetto sono evidenziate le modifiche effettuate:

```
import java.util.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.*;
import java.rmi.RemoteException;
import java.rmi.registry.*;

public class LibreriaServer extends PortableRemoteObject implements Libreria {
        Hashtable libri;

        public String nomiDegliAutori(String titolo) throws RemoteException{
                String autori;
                autori = (String)libri.get(titolo);
                if (autori== null) return ("Libro non trovato");
                else return autori;
        }

        public  LibreriaServer() throws RemoteException{
                libri = new Hashtable();
                libri.put("La Divina Commedia","Dante Alighieri");
                libri.put("Il Decameron","Giovanni Boccaccio");
                libri.put("I Promessi Sposi","Alessandro Manzoni");
        }

        public static void main(String[] args) {
                try {

                        LibreriaServer rmiServer = new LibreriaServer();

                        // Otteniamo la referenza ad una istanza IIOP via JNDI
                        Hashtable props = new Hashtable();
                        //definiamo le proprietà per la creazione del Context
                        props.put(Context.INITIAL_CONTEXT_FACTORY,
                                        "com.sun.jndi.cosnaming.CNCtxFactory");
                        props.put(Context.PROVIDER_URL, "iiop://localhost:900");
                        Context ctx = new InitialContext(props);
```

```
                    ctx.rebind("LaMiaLibreria", rmiServer );

                    System.out.println("Classe LibreriaServer registrata e pronta a ricevere
                                        chiamate.");
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
        }
}
```

Innanzitutto la superclasse da estendere, per poter generare un oggetto invocabile da remoto, non è più UnicastRemoteObject ma la *PortableRemoteObject* del package *javax.rmi*.
La modifica maggiore, in questo caso, risiede nella modalità di registrazione del servizio al registry; viene creata una Hashtable, contentente una lista di proprietà, da passare al costruttore della classe *Context*. I servizi JNDI (Java Naming & Directory Interface) associano nomi ad oggetti, e la classe *Context* definisce il contenitore delle associazioni utilizzato.
Fra le proprietà da specificare per inizializzare correttamente il *Context*, è necessario definire nella proprietà *Context.INITIAL_CONTEXT_FACTORY* qual è l'implemetazione (JNDI definisce solo interfacce) della classe che si occupa di offrire i servizi di naming. In questo caso la classe specificata è *com.sun.jndi.cosnaming.CNCtxFactory*, che è la classe definita nell'SDK J2EE; nel caso si voglia utilizzare un altro strumento di registry collegato ad un ORB proprietario, è necessario specificare qui la classe opportuna.
L'altra proprietà, *Context.PROVIDER_URL*, definisce qual è l'indirizzo del computer ove risiede il gestore dei servizi di registrazione. Si noti come, in questo caso, il protocollo di trasporto non sia più RMI ma IIOP, che implica una modifica anche del gestore dei servizi di registry che deve esser conforme allo standard CORBA. Nel SDK l'utility di gestione, che sostituisce la *rmiregistry* della versione 1.1, è *tnameserv*, la cui porta di default è la 900.
L'esecuzione della utility *tnameserv* produce l'output in figura seguente:

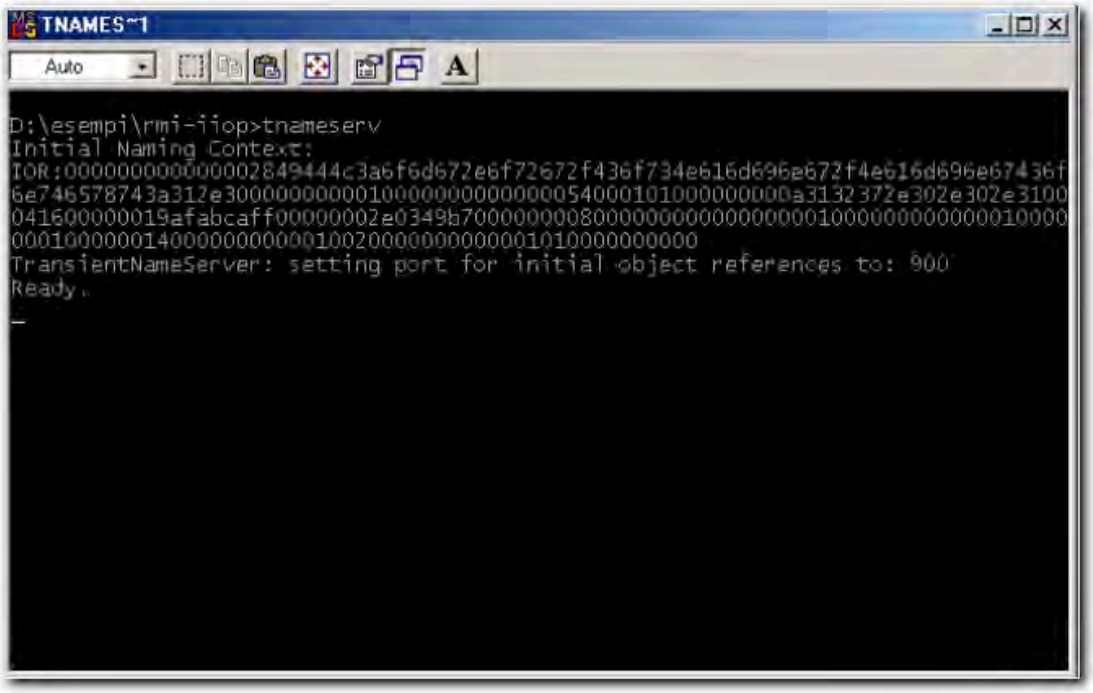


***Figura 144: L'esecuzione della utility* tnameserv**

L'esecuzione della classe LibreriaServer, che registra la classe sul registry server e la mette in attesa di chiamate, produce il seguente output:

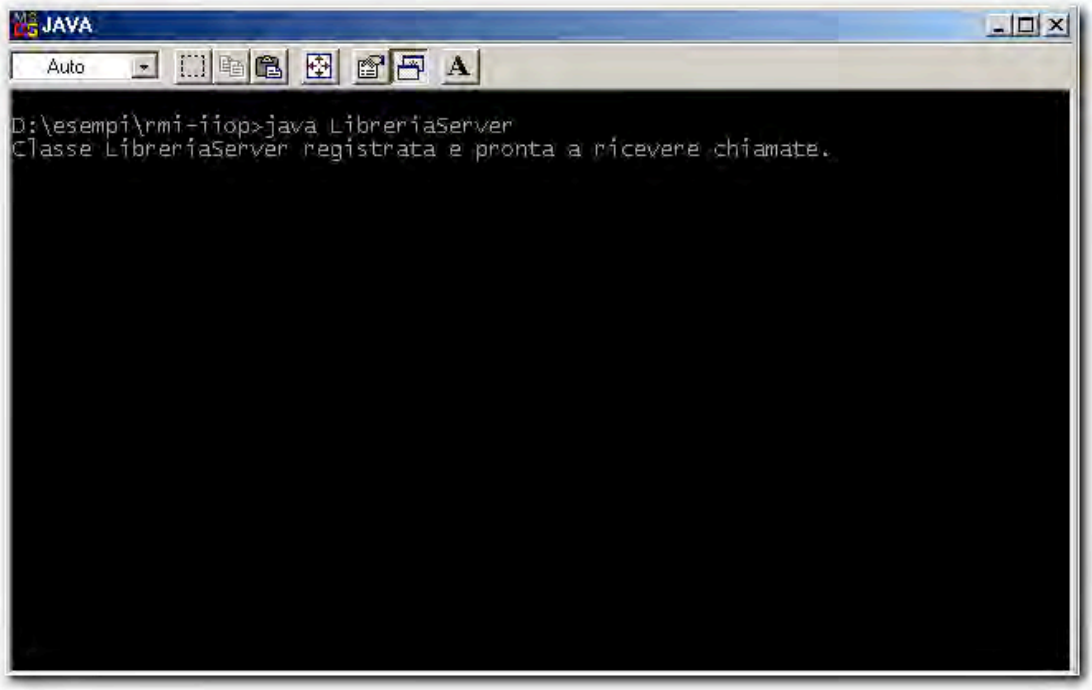


Figura 145: Esecuzione della classe LibreriaServer

Vediamo il codice del client:

```
import javax.naming.*;
import java.util.*;
import javax.rmi.PortableRemoteObject;

public class Client {
        public static void main(String[] args) {
        try {
                // inizializzazione del contesto
                Hashtable props = new Hashtable();

                props.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.cosnaming.
                CNCtxFactory");
                props.put(Context.PROVIDER_URL, "iiop://localhost:900");
                Context ctx = new InitialContext(props);

                // Otteniamo un riferimento al contesto in cui è presente l'oggetto server
                Object riferimento = ctx.lookup("LaMiaLibreria");

                // ricerca e allocazione del finto oggetto server
                Libreria mioLinkOggettoServer =
                        (Libreria)PortableRemoteObject.narrow(riferimento,Libreria.class);

                String autori = mioLinkOggettoServer.nomiDegliAutori(args[0]);

                // Stampa del risultato.
                System.out.println("Sulla libreria server l'autore del libro " + args[0] + " e':\n\t" +
autori);
        }
        catch (Exception e) {
                e.printStackTrace();
        }
 }
}
```

Come risulta dall'analisi delle righe in grassetto, la creazione del *Context* è analoga a quanto visto per il metodo *main* della classe LibreriaServer.
Il primo elemento di novità è rappresentato dalla creazione di un oggetto che fa da riferimento generico all'oggetto remoto, attraverso il metodo *ctx.lookup(String)*. In questa chiamata viene effettuata la ricerca dell'oggetto remoto ma, diversamente da quando visto nel caso Java 1.1, viene restituito un generico oggetto che fa riferimento al contesto in cui è contenuta l'istanza remota.
Il vero riferimento all'oggetto remoto è definito nell'istruzione successiva, in cui viene utilizzato (in modo analogo a quanto visto con l'oggetto Naming nel caso Java 1.1) il metodo *narrow* dell'oggetto *PortableRemoteObject*.
L'output prodotto dall'esecuzione di Client.java è presentato nella figura seguente:

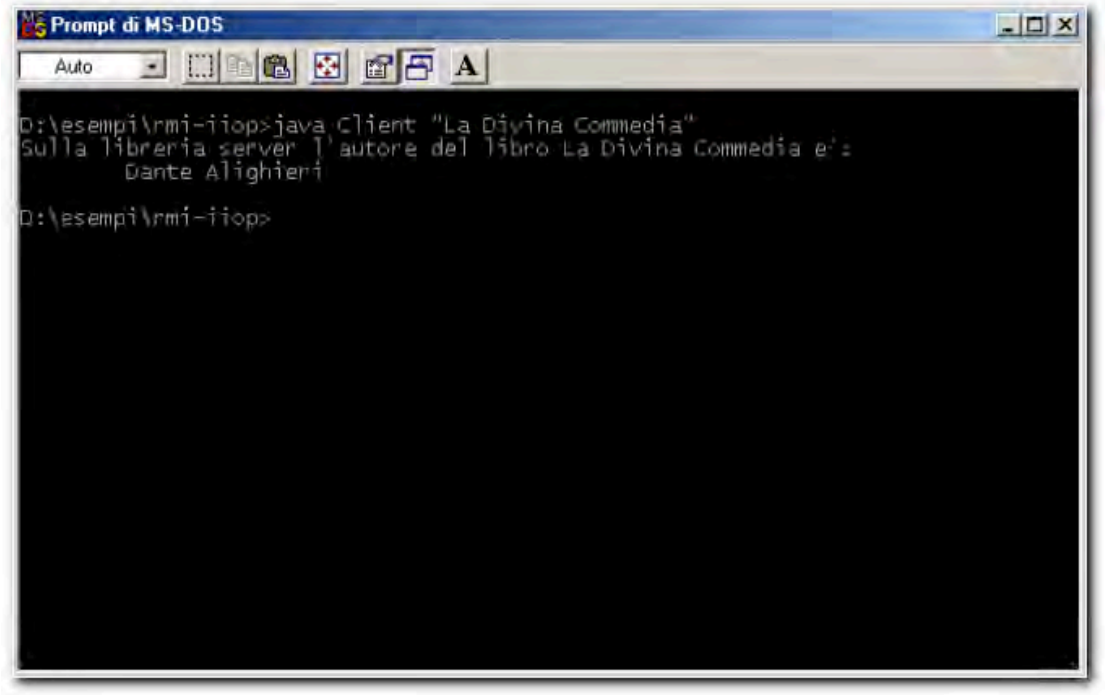


Figura 146: esecuzione di Client.java

20.6 Serializzazione

Uno degli aspetti cruciali quando si lavora con oggetti distribuiti è la garanzia che le comunicazioni fra i vari oggetti avvengano in modo corretto, ossia senza che la distribuzione su computer differenti ne infici la funzionalità. I metodi delle classi, infatti, hanno spesso come argomenti, o come valori di ritorno, dati che, in linguaggio Java, sono a loro volta classi; ad esempio, per comunicare una stringa di caratteri ad un metodo il modo più intuitivo è quello di mettere tra i parametri di input del metodo un dato di tipo *java.lang.String*. Cosa succede all'oggetto, passato come valore al metodo o ritornato dallo stesso, qualora l'oggetto chiamante sia una classe remota? Il protocollo di comunicazione (TCP/IP o UDP) non conosce la struttura delle classi Java: il suo compito si esaurisce nel trasportare blocchi di byte sulla rete secondo il proprio standard. Ogni oggetto, pertanto, quando viene trasmesso sulla rete viaggia secondo il formato binario così come qualsiasi altro tipo di dato primitivo. Per far in modo che la struttura dell'oggetto rimanga valida (ed utilizzabile) durante una trasmissione, è necessario che l'oggetto venga convertito in modo opportuno in uno stream di byte, ed analogamente possa essere ricostruito dal ricevente. Per gestire semplicemente questo processo ci viene in aiuto l'interfaccia **java.io.Serializable**. Questa interfaccia è vuota, quindi le classi che la implementano non debbono implementare alcun metodo in più, ma ha il compito di comunicare al compilatore java la direttiva su come costruirne una

rappresentazione binaria adatta al trasporto su uno stream (in pratica, che l'oggetto in questione deve essere trattato in modo da poter viaggiare su stream in modo "sicuro").
E' utile aggiungere alcune importanti considerazioni:

1. *qualsiasi oggetto da far passare su uno stream deve essere serializzabile. Se una classe non implementa l'interfaccia Serializable (o non estende una classe a sua volta serializzabile) evitate di effettuarne la trasmissione di tale classe: non riuscirete a ricostruire l'oggetto;*
2. *non esiste una eccezione specifica per segnalare un errore sulla serializzazione: la trasmissione di un oggetto non serializzato comporta errori a runtime nel momento in cui si tenta di utilizzare l'oggetto stesso;*
3. *la trasmissione di cui si parla è relativa alla scrittura su qualsiasi tipo di stream (non è solo quella su socket): anche se volete salvare una classe su un file, l'oggetto, per poter essere ricostruito, deve essere serializzabile.*

20.7 Marshalling di oggetti

Il termine marshalling viene normalmente utilizzato per definire il processo di conversione di dati tra piattaforme differenti. Quando, infatti, lo scambio di dati avviene tra computer aventi differenti piattaforme software (sistemi operativi, ma anche bus CORBA, ad esempio), può accadere che i sistemi abbiano diversi modi per trattare determinate categorie di dati. Questo problema viene parzialmente risolto da Java considerando che i tipi di dati primitivi hanno lunghezza standard per tutte le piattaforme. Questo, però, non risolve tutti i problemi dal momento che:

1. *anche se gli oggetti sono identici, la rappresentazione binaria per la trasmissione, su piattaforme differenti, può essere diversa (es. big-endian e little-endian)*
2. *la rappresentazione in binario di oggetti generici può essere fatta in modo differente da piattaforme diverse;*
3. *a volte vengono effettuate delle operazioni di casting forzato, che su un programma stand-alone possono risultare indolori, ma su programmi distribuiti generano errori. Se, ad esempio, su una interfaccia remota ho un certo tipo come definito per il passaggio di parametri, se tale tipo viene forzato il programma potrebbe non funzionare bene.*

I processi di marshalling e unmarshalling vertono sulla conversione degli oggetti nella loro rappresentazione binaria (insieme alla serializzazione), e

vengono effettuati dallo Stub e dallo Skeleton, sulla base della definizione della interfaccia remota. In pratica, stub e skeleton si occupano di preparare gli oggetti al loro invio su rete (ovvero alla loro ricostruzione dalla rappresentazione binaria) basandosi sulla definizione dei tipi presente sull'interfaccia.
Se il processo di marshalling (o quello di unmarshalling) fallisce, la JVM emette a run-time una speciale eccezione, di tipo *MarshalException*.

20.8 Attivazione dinamica di oggetti

Nell'esempio precedente abbiamo visto come l'oggetto remoto si registri al registry server e rimanga disponibile per rispondere alle chiamate dei client remoti.
Questa procedura, tuttavia, può risultare onerosa dal punto di vista dell'uso di risorse qualora diventino numerose le istanze di oggetti da mantenere disponibili in memoria.
Per risolvere questo problema, con la versione 1.2 di RMI è stata introdotta anche la possibilità di istanziare dinamicamente gli oggetti in base alle richieste dei client, delegando le logiche di esecuzione dei processi ad un agente esterno. Questo processo è detto **attivazione** e l'agente è un daemon che ha il compito di intercettare le chiamate dei client ed istanziare gli oggetti; il daemon fornito nell'SDK è l'utility *rmid*.
Senza entrare nei dettagli implementativi, per poter essere attivati su richiesta, gli oggetti server, però, debbono avere speciali caratteristiche. In particolare, essi debbono estendere la superclasse *java.rmi.activation.Activable*, ed implementare uno speciale costruttore, che viene invocato dal daemon, per l'allocazione dinamica dell'oggetto.

21 ENTERPRISE JAVA BEANS

21.1 Introduzione

Come anticipato nei precedenti capitoli, gli Enterprise JavaBeans (EJB) rappresentano uno degli elementi principali dell'architettura J2EE. Si potrebbe, anzi, dire che siano, in realtà, il vero elemento di novità dell'architettura software, e ne costituiscono la sintesi suprema.
Gli EJB non sono una nuova tecnica di programmazione o un set di API da utilizzare, sono uno strumento potentissimo il cui corretto uso prevede un impatto forte sull'intero ciclo di vita del software, con conseguente ripensamento delle "normali" tecniche di analisi, progettazione, sviluppo, messa in esercizio e manutenzione evolutiva.
Queste caratteristiche di sintesi fanno sì che questi elementi siano, in verità, molto complessi da comprendere; per questo motivo i paragrafi che seguono cercheranno di tradurre in un linguaggio per quanto possibile semplice i principali aspetti legati a questa tecnologia.
Non è nostra intenzione esplorare interamente la complessità dell'argomento in questo libro, per chi fosse interessato ad approfondire le proprie conoscenze in materia consigliamo di seguire i riferimenti presenti nella bibliografia.

21.2 Gli EJB: concetti base

Il contesto di riferimento per l'utilizzo degli EJB sono gli ambienti complessi con architetture distribuite.
In passato gli ambienti software complessi, tipici delle grandi organizzazioni, venivano sviluppati in modo disorganico, frutto della divisione funzionale dei task e dei progetti, e sviluppati in maniera eterogenea, utilizzando le tecniche che venivano ritenute, di volta in volta, più opportune per la singola funzionalità. Il risultato di questo processo era, nella gran parte dei casi, una diabolica alchimia di sistemi, ciascuno dei quali sviluppato con logiche, metodologie e tecnologie ad hoc che, preso singolarmente, assolveva ai propri compiti, ma senza una visione d'insieme.
In tale contesto, nei casi in cui emergeva la necessità di integrare sistemi funzionalmente separati, il compito del system integrator risultava il più delle volte assai complesso, dovendo far "parlare" tra loro sistemi con architetture diverse, spesso scritti in linguaggi diversi e magari sviluppati su piattaforme differenti! La necessità di raggiungere lo scopo finiva, spesso, per generare un nuovo sistema, diverso da tutti quelli esistenti e da quelli da integrare, in architettura proprietarie e da mantenere e controllare separatamente, con tutti i problemi di esercizio connessi.

Questa situazione è vera ancora oggi in moltissime grandi organizzazioni (e, per inciso, non potrebbe essere altrimenti, dal momento che lo sviluppo dei sistemi informativi va avanti oramai da molti anni), ma sta emergendo la necessità di standard architetturali che semplifichino il processo di integrazione dei sistemi in ambienti eterogenei.
L'architettura J2EE (e, con gli opportuni distinguo che vedremo in seguito, i Web Services) vuole rappresentare, in questo senso, uno standard per l'integrazione dei sistemi, con gli EJB come passepartout per aprire ogni sistema da integrare.
Il primo paradigma su cui si basa, utilizzato anche in RMI e CORBA, è quello delle **architetture distribuite**, di cui è stata data una breve spiegazione nel capitolo dedicato ad RMI.
Il secondo, fondamentale, concetto su cui si basano gli EJB è quello di **componente**: nell'ottica di decomporre un problema secondo la logica del *divide et impera* (non secondo un'ottica funzionale, ma Object Oriented), un componente è un elemento software (un frammento di codice) che implementa la logica applicativa definita da un insieme di interfacce. Quello di componente è un concetto astratto, non facile da definire, e per questo verrà approfondito nel paragrafo successivo.
Altri importanti elementi su cui si basa l'architettura EJB sono relativi ai concetti di controllo transazionale, load balancing, ciclo di vita degli oggetti, etc; non è obiettivo di questo capitolo approfondire tutti questi aspetti.
Ma cosa sono, in definitiva, gli EJB?
E cosa li distingue da altri sistemi simili, come DCOM?
Essi sono componenti software ciascuno autonomamente responsabile di una parte di logica applicativa delegata al lato server del sistema. Ogni EJB non è un programma autoconsistente, esso incapsula solo una parte di logica applicativa ed il suo ruolo si esplica all'interno del contesto in cui è inserito.
In particolare, il paradigma EJB è composto da:

- *la specifica SUN riguardante il modo in cui i componenti interagiscono con il loro ambiente di riferimento (ossia l'application server)*
- *un insieme di interfacce che il componente deve integrare*

Le caratteristiche distintive degli EJB sono:

- *linguaggio Java: è possibile scrivere componenti in qualsiasi linguaggio, ma gli EJB possono essere scritti solo in Java; questo vincolo non rappresenta,, in pratica, un grande vincolo, considerando che grazie alla caratteristica di Java di essere cross-platform, ogni EJB può essere installato su qualsiasi macchina con qualunque sistema operativo;*
- *la presenza di un ambiente (l'application server) che contiene l'EJB e che si occupa autonomamente del suo ciclo di vita, non delegandolo ad altre applicazioni;*

- *una netta distinzione tra interfacce e implementazione, dal momento che l'interfaccia di un EJB, ossia il meccanismo che consente ad un'entità di colloquiare con l'EJB, è fisicamente separata dal componente stesso contenente l'implementazione;*
- *il fatto che siano componenti server-side, e non oggetti (come i beans o gli ActiveX) o librerie da includere per lo sviluppo; gli EJB sono microsistemi autonomi pronti al deploy, che espongono una sola interfaccia i cui metodi possono essere chiamati da qualsiasi client, sia esso un client GUI, un servlet o un altro EJB.*

Il concetto di *deploy* (parola difficilmente traducibile in una parola italiana altrettanto efficace) è riassumibile come il processo relativo alla configurazione del contesto di riferimento dell'EJB, alla definizione delle sue interazioni con il suo ambiente circostante (l'application server) ed alla messa in esercizio dell'EJB stesso.

21.3 Ragionare per componenti

Progettare e sviluppare EJB vuol dire imparare a ragionare per componenti.
Come detto, quello di componente è un concetto astratto, che sfugge ad ogni tipo di definizione sintetica ed esaustiva; è, però, possibile tracciarne qualche caratteristica utile per definirne i contorni.
Un componente non è un applicazione completa: esso è in grado di eseguire una serie di task, ma non in modo autonomo come un'applicazione.
Una caratteristica fondamentale del concetto di componente è l'aderenza ad uno standard di riferimento, in modo da consentire ai software integrator di costruire applicazioni utilizzando componenti sviluppati da qualsiasi software houses: in questo senso EJB, così come DCOM, sono basati sui rispettivi standard SUN e Microsoft.
In gergo tecnico, si dice che il componente software, per essere tale, deve aderire ad un modello a componenti; un modello a componenti definisce in modo formale le regole di interazione fra i vari elementi componenti.
Rispetto ad un oggetto, un componente presenta il concetto di interfaccia che disaccoppia la natura implementativa dalla sua specifica; questo consente (in teoria) di sostituire componenti aventi la stessa specifica senza doversi curare dell'implementazione sottostante.
Un componente dovrebbe essere relativo ad un argomento "chiuso", ad una problematica definita, ed in questo senso condivide ed estende il concetto di classe. E', infatti, possibile scrivere componenti composti da più di una classe; in questo modo, senza distorcere il paradigma Object Oriented, è possibile "riunire" dentro lo stesso componente un insieme di classi che collaborano tra loro per eseguire un task. L'importante è che il componente esponga i propri metodi attraverso una sola interfaccia verso il mondo esterno.

Ad esempio, immaginiamo di dover realizzare un modulo per l'autenticazione e l'accesso degli utenti ad un sistema. Se supponiamo di autenticare l'utente attraverso login e password, ecco che la problematica dell'autenticazione diventa un argomento limitato, e possiamo pensare di utilizzare un unico componente cui delegare l'implementazione.
Un componente deve esser pensato con l'ottica della riusabilità: mentre lo creiamo, cerchiamo di pensare che la sua funzione non dovrebbe essere solo quella di assolvere ai compiti di cui abbiamo bisogno al momento, ma (per astrazione) anche alle problematiche generali inerenti l'argomento, e alla possibilità di utilizzarlo in un contesto diverso. Se la necessità di astrazione è vera nel disegno di oggetti, lo è ancor di più in quello di un componente.
Tornando all'esempio precedente, potremmo avere che, per la nostra applicazione, la lista degli utenti registrati è contenuta in una tabella di un RDBMS. Ma, nello sviluppo del componente, sarebbe un errore pensare di cablare la logica di autenticazione sull'integrazione con un database (e se, infatti, un domani dovessimo interfacciare un server LDAP?). Ecco che, ad esempio, conviene pensare da subito al componente come non legato al sistema che contiene gli utenti; dovremmo cercare di pensare da subito ad una specifica generale, che incapsuli l'implementazione delle varie tecnologie utilizzate per realizzare la funzionalità.
Diverse, inoltre, sono *le forme in cui si può presentare un componente*. Quando vogliamo utilizzare un componente, sia esso proveniente da una entità esterna oppure scritto da noi, la prima operazione da compiere è quella di inserirlo nell'ambiente operativo in cui dovrà fornire i suoi servizi: abbiamo, cioè, la necessità di installarlo (o, in caso, ad esempio, di EJB, di effettuarne il deploy). In tale caso in componente assume la forma di un componente installato, differente da quella di componente da installare in quanto l'operazione di deploy comporta anche altre operazioni (ad esempio, il componente deve essere registrato nell'elenco dei componenti utilizzabili dall'applicazione).
A run time, ossia durante l'esecuzione del programma, il componente installato non viene utilizzato direttamente, ma ne viene creata una copia in memoria (istanza) contenente i dati e i risultati relativi all'elaborazione in corso. Per analogia con i concetti di classe ed oggetto, l'istanza del componente viene detta componente oggetto. Si noti che, indipendentemente dai dati di ingresso e dal momento in cui viene fotografata l'elaborazione, ogni componente oggetto, benché creato come copia dello stesso componente installato, ha almeno una caratteristica unica e distintiva, l'Object Identificator (OID).
Riassumendo i concetti esposti, nella tabella successiva vengono schematizzati i concetti espressi:

Riepilogo

Vocabolo	Descrizione
Componente	Parte di software che aderisce ad un modello a componenti
Modello a componenti	Standard di riferimento per la costruzione di componenti
Specifica del componente	Specifica della funzionalità del componente stesso (cosa deve fare)
Interfaccia del componente	Insieme dei metodi che il componente espone per poter essere attivato (cosa fa)
Implementazione del componente	Realizzazione della specifica del componente stesso (come lo fa)
Componente installato	Copia del componente inserita nel suo contesto applicativo
Componente oggetto	Istanza del componente installato

Dall'analisi delle caratteristiche dei componenti è evidente che JAVA definisce due tipologie di standard di componenti al suo interno. Il primo, più datato, è quello dei JavaBeans, l'altro, introdotto in J2EE, degli EJB. Tra i due tipi di componenti c'è una profonda differenza: i Javabeans sono componenti di sviluppo, essi possono essere presi e utilizzati a piacimento per realizzare software, ed il loro ambiente di run-time è costituito dall'applicazione che li utilizza. Gli EJB, invece, sono componenti più complessi, che necessitano di un ambiente specifico per essere istanziati ed utilizzati.

21.4 L'architettura a componenti

Il modello cui si fa riferimento negli EJB è quello di un'architettura a componenti. Abbiamo visto cos'è un componente; un'architettura a componenti è, pertanto, un'architettura software di cui una parte (o tutta) è progettata e realizzata attraverso la collaborazione di più componenti.
La realizzazione di un'architettura a componenti prefigura uno scenario di sviluppo software in cui gli attori possono essere numerosi e con differenti ruoli.
Nello schema seguente è riassunto l'insieme dei ruoli nel ciclo di sviluppo a componenti:

Ruoli nel ciclo di sviluppo a componenti	
Ruolo	**Descrizione**
Fornitore dei componenti	È colui che fornisce i componenti. Si noti che non c'è differenza tra componenti sviluppati ad hoc o acquistati sul mercato; questo implica che, dal punto di vista della progettazione, va valutato il costo opportunità dello sviluppo per singolo componente.

L'assemblatore dei componenti	Nell'attività di realizzazione dell'architettura a componenti, è colui che, sulla base della conoscenza del business e delle specifiche di progetto, sceglie e compone tra loro i componenti, come mattoni per la costruzione di un edificio. Si occupa anche dello sviluppo delle parti di applicazione non realizzabili per componenti, come le interfacce grafiche o gli elementi di integrazione.
L'EJB deployer	Si occupa del deploy e della messa in esercizio del sistema a componenti
L'amministratore del sistema	Ha il compito di mantenere in buona efficienza il sistema in esercizio, monitorando il funzionamento ed effettuando interventi di tuning.
Il fornitore dell'application server	E' l'azienda che fornisce l'ambiente che contiene i componenti e le utilità di gestione del ciclo di vita degli stessi

21.5 Il ruolo dell'application server e struttura degli EJB

All'interno dell'architettura generale di figura seguente

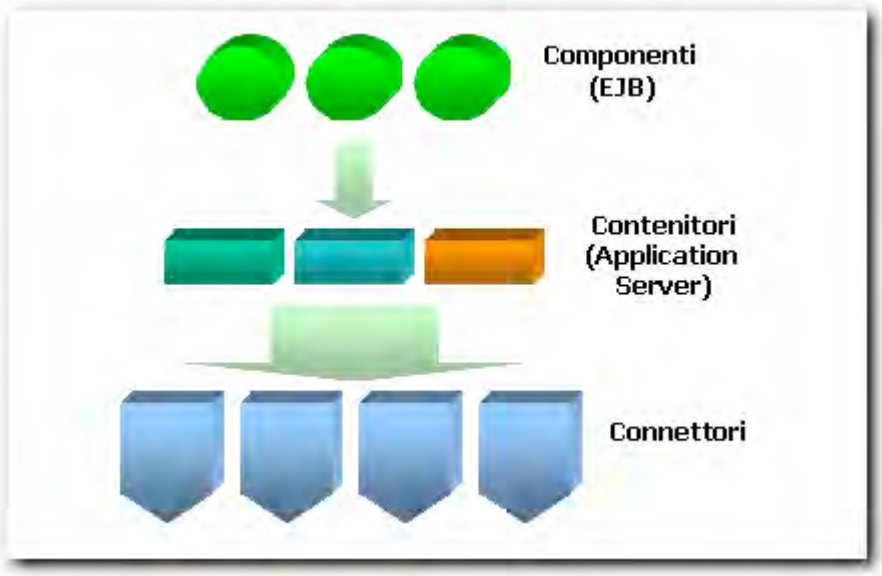


Figura 147: architettura java2

gli EJB rappresentano i componenti, il cui ciclo di vita è realizzato all'interno dei Contenitori. L'Application server (che chiameremo AS nel prosieguo del capitolo) svolge, tipicamente, il ruolo di contenitore degli EJB, ma sarebbe riduttivo pensare solo a questa funzione. Gli AS, infatti, forniscono una ulteriore serie di elementi e servizi di supporto, quali, ad esempio:

- disponibilità di un Web server per la pubblicazione dei servizi su protocollo HTTP
- disponibilità di un container servlet/jsp associato al Web server
- console di amministrazione per la gestione del deploy a caldo, sia di applicazioni Web che di applicazioni enterprise
- console di amministrazione per la gestione e configurazione dinamica dell'ambiente
- supporto e strumenti di gestione per il clustering
- disponibilità di servizi di naming (JNDI)
- disponibilità di un bus CORBA
- disponibilità di utilità per la gestione dei web services
- disponibilità dell'infrastruttura per i servizi di messaging (JMS)

Questo complesso insieme di servizi consente di manutenere facilmente le applicazioni sia in ambiente di sviluppo che di esercizio; l'integrazione dei servizi in un unico ambiente di monitoraggio facilita, inoltre, il lavoro di amministrazione.
Visto che si tratta di "suite" di prodotti e servizi, l'adozione di un AS, inoltre, non rappresenta solo l'infrastruttura tecnologica di base per le architetture a componenti, ma, data la presenza di CORBA, JMS e Web services, anche per la system integration in generale.
Gli AS, pertanto, rappresentano il cuore dell'infrastruttura tecnologica delle applicazioni J2EE; la scelta dell'AS da utilizzare è, quindi, cruciale in fase di progettazione del sistema.
Senza addentrarci troppo nelle differenze tra i vari prodotti esistenti in commercio, ecco alcuni consigli da seguire nella scelta e nell'uso di un AS, sia che siate dei progettisti che degli sviluppatori in architettura J2EE:

- *tutti gli AS java in commercio sono conformi allo standard SUN J2EE; tuttavia le loro caratteristiche progettuali possono essere anche molto differenti. Alcuni AS, infatti, nascono come continuazione ed estensione di esperienze specifiche dei produttori in prodotti "storici" (ad esempio, alcuni nascono da produttori di ORB CORBA, altri da produttori di sistemi di message queing, altri ancora da esperienze nei database relazionali); questo fatto ci consente di capire quale "taglio" progettuale (e quali siano gli elementi di forza di base) dei vari prodotti. Questa considerazione, se non ci dà alcuna informazione sulla qualità del prodotto, può essere utile per inquadrare la scelta dell'AS nel contesto dei prodotti presenti nell'ambiente enterprise di riferimento.*

- *Anche le caratteristiche operative degli AS sono, tra i vari prodotti, diverse. Ogni produttore, vuoi per necessità progettuali, vuoi per integrare al meglio i servizi offerti, ha aggiunto allo standard SUN degli*

elementi proprietari. Queste aggiunte non comportano, generalmente problemi in fase di sviluppo dei componenti (si tratta, di solito, di fare attenzione alle librerie da includere), ma possono comportare differenze anche sostanziali nella fase di deploy. Passare da un AS ad un altro vuol dire, nella gran parte dei casi, modificare il processo di deploy. Dal momento che i principali strumenti di sviluppo presentano anche utility per il deploy, la creazione automatica dei descrittori e la configurazione dell'ambiente, alla scelta di un AS deve corrispondere una scelta sull'ambiente di sviluppo più consono.

- *L'inserimento di un AS in una architettura software complessa comporta l'integrazione con prodotti sviluppati da differenti produttori software. Anche quando scritti in Java e conformi alle direttive SUN, non tutti i prodotti sono compatibili con tutti gli AS! A volte si tratta solo di versioni da aggiornare, altre volte di incompatibilità strutturali; in ogni caso, una buona ricerca sulle FAQ dei produttori degli AS e/o dei prodotti da integrare potrà anticipare brutte sorprese.*

Dal punto di vista degli EJB, l'AS rappresenta l'ecosistema in cui vive una istanza EJB. L'AS, nel suo ruolo di contenitore degli EJB:

- *gestisce la Java Virtual Machine necessaria ad istanziare il componente, con i parametri di configurazione e avvio prescelti;*
- *intercetta le chiamate verso il componente e si preoccupa di capire quale oggetto è coinvolto, di verificare se creare o meno istanze dell'oggetto per gestire la richiesta e di gestire l'univocità degli oggetti identificandoli con l'OID;*
- *gestisce le chiamate agli oggetti integrandosi con i servizi JNDI;*
- *gestisce i servizi di transazionalità;*
- *gestisce la persistenza degli oggetti;*
- *gestisce il connection pooling (insiemi di connessioni a database);*
- *gestisce il garbage collection degli oggetti.*

Tutti queste funzionalità vengono svolte dall'AS in modo trasparente al programmatore, semplificandone notevolmente il lavoro.
L'EJB, all'interno dell'AS, è quindi invisibile ed inaccessibile direttamente; per rendere le sue funzionalità disponibili all'esterno, l'AS crea una copia del bean detta EJB Object.
La funzionalità di container degli EJB può essere compresa con la figura seguente:

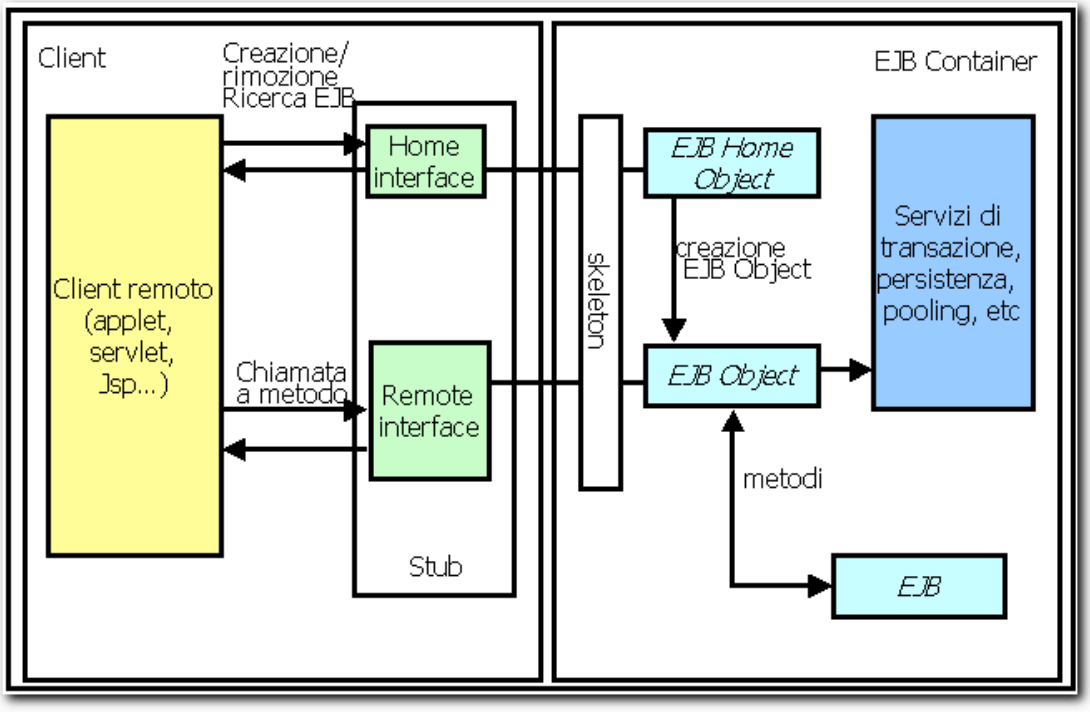


Figura 148: ruolo dei vari elementi componenti un EJB

Il client remoto accede all'EJB attraverso le interfacce *home* e *remote*, incapsulate nei componenti stub-skeleton; lo strato di servizio dell'application server espone i servizi EJB mediante due copie virtuali dell'EJB: l'*EJBHome object* e l'*EJB Object*.
L'EjbObject rappresenta una vera copia dell'Ejb (è il componente oggetto) ed è utile nelle chiamate ai metodi dell'EJB. L'EjbHome object rappresenta, invece, la rappresentazione esterna dell'istanza del componente; in pratica rappresenta l'elemento di gestione del ciclo di vita del componente, essendo responsabile della creazione, della rimozione e della ricerca, nell'insieme dei componenti in memoria, del componente oggetto ricercato dal client.
Le interfacce remote e home sono, per il client, l'interfaccia del componente, gli elementi che consentono la location trasparency e che ci dicono "chi è" e "cosa fa" il componente.
Nella specifica EJB 2.0, con significato analogo al caso precedente, vengono definite anche le componenti *local;* l'EJB, con l'interfaccia local, consente ai client locali (ossia che sono presenti nella stessa virtual machine dell'AS, come ad esempio altri EJB) di evitare lo strato di comunicazione stub-skeleton nelle chiamate al bean, migliorando sensibilmente le prestazioni. In tale caso vengono definiti i due elementi di interfaccia, l'EjbLocal (funzionalmente analoga alla remote) e l'EjbLocalHome (analoga all'EJBHome).

Ad ognuno di questi elementi è associato, nel codice, una chiamata ad un elemento della libreria javax.ejb.

Corrispondenza tra componenti	
Elemento Logico	**Corrispondenza nel codice**
EJB	implements javax.ejb. … (qui si distinguono i tipi di bean)
Home interface	Extends javax.ejb.EjbHome
Remote interface	Extends javax.ejb.EjbObject
LocalHome interface	Extends javax.ejb.EjbLocalHome
Local interface	Extends javax.ejb.EjbLocalObject

1.1 Il concetto di pooling

Uno degli aspetti più importanti da comprendere sugli EJB, e sul rapporto che essi hanno con l'AS, è il concetto di *pooling*. Si tratta della capacità dell'AS di gestire un insieme di risorse (pool) in modo centralizzato, intendendo con la parola gestione tutto il processo legato all'insieme delle risorse coinvolte, che pertanto risulta ottimizzato e controllato.
Due sono, in particolare, i servizi di pooling offerti dall'AS: il bean pooling ed il connection pooling.
Per bean pooling si intende la capacità dell'AS di gestire gruppi di istanze di EJB.
Quando, infatti, un client richiede i servizi di un EJB (ad esempio, ne richiede la creazione o ne invoca un metodo), la sua chiamata all'EJBHome Object (o all'EJBObject) viene intercettata dal container che è in grado di decidere autonomamente se:

- *in caso di creazione, creare effettivamente una nuova istanza dell'EJBObject;*
- *in caso di creazione, utilizzare una istanza "nuova" tra quelle da lui preparate e mantenute in memoria.*

L'AS, infatti, provvede a tenere sempre in memoria un certo numero di bean precaricati che non sono pronti all'uso, dal momento che non sono stati inizializzati. Nel caso l'AS decida di usare uno di questi, provvede a inizializzarne lo stato, creandogli il contesto di riferimento, e poi a renderlo disponibile. Questo processo consente all'AS di risparmiare tempo in fase di caricamento del bean, anche se a scapito di un po' di memoria. Questo aspetto deve essere, però, tenuto sotto controllo, altrimenti le risorse di sistema andrebbero esaurendosi rapidamente. Nel caso in cui esistano dei bean sviluppati in modo da restare in memoria per lungo tempo, così da poter servire il client in momenti differenti, l'AS deve utilizzare la strategia opposta, ossia cercare di risparmiare risorse. Per fare questo utilizza i concetti di passivazione e attivazione (di cui parleremo in seguito), con l'obiettivo finale

di mantenere sempre il numero di bean in memoria entro un range controllato. In sostanza, il concetto di bean pooling prevede, a seconda del tipo di bean, delle politiche che cerchino ottimizzare le prestazioni del sistema a parità di risorse a disposizione dell'AS.
Diverso è il concetto di connection pooling. Questo ha a che fare con un noto problema inerente l'utilizzo di applicazioni Web con database relazionali (in particolare questo problema era frequente nei casi di uso di programmi CGI). Quando un client Web effettua una transazione verso un database, l'applicazione lato server deve, in corrispondenza di ogni richiesta, aprire una connessione verso il database, tenerla aperta per il tempo di lavoro e poi chiuderla. Questo continuo aprire e chiudere le connessioni al database ha un effetto devastante sulle prestazioni dell'applicazione; le operazioni di connect e disconnect su RDBMS sono, infatti, estremamente costose in termini di risorse macchina del sistema che ospita il database, ed anche per il complesso handshaking di protocollo tra client e RDBMS. L'indeterminatezza sul numero dei client, e quindi delle connessioni, unito alla impossibilità di limitare la concorrenza rendeva, inoltre, difficile effettuare un tuning sui sistemi in queste condizioni.
L'idea, quindi, è quella di aprire un certo numero predefinito di connessioni (pool di connessioni) e tenerle sempre aperte per le varie istanze dei processi che debbono utilizzarle. E' il container ad assegnare, in modo trasparente all'applicazione stessa, la risorsa connessione, scegliendone una fra quelle disponibili al momento, ed occupandosi di gestire eventuali accodamenti. In questo modo il carico sul database è costante, con valori di picco predefiniti in fase di configurazione e non aleatori.

21.6 I tre tipi di EJB

I tipi di Enterprise JavaBeans definiti da Sun nella specifica 1.1 sono due: SessionBeans e EntityBeans. Nella più recente specifica 2.0 è stato aggiunto anche un terzo tipo, i MessageDrivenBeans, otre ad essere cambiate alcuni elementi dell'architettura degli altri due tipi. La definizione delle differenze tra le varie specifiche, così come i dettagli implementativi, non saranno oggetto del libro; si è pertanto, deciso di esporre una trattazione ristretta di casistiche e di concetti, al fine di consentire al lettore una panoramica agile anche se, necessariamente, non completa.

21.7 Session Beans

I SessionBeans sono i componenti che, nella architettura EJB, devono occuparsi di incapsulare e definire la business logic delle applicazioni lato server. Il loro nome deriva dalla considerazione che la logica applicativa è un elemento che necessita di una memorizzazione persistente dei dati in essa residenti (le variabili locali necessarie all'elaborazione) limitata alla sessione di

lavoro. Nel caso in cui i dati debbano essere resi persistenti per un tempo arbitrariamente lungo, allora è necessario utilizzare un supporto logico dedicato, come un database, con una rappresentazione adeguata (che, nel modello EJB, è data dagli EntityBean).

21.8 Tipi di Session Beans

Al fine di esplicitare ulteriormente questo concetto, il paradigma EJB distingue i SessionBeans in:

- *stateless: bean che limita la propria esistenza ad una chiamata da parte di un client. L'oggetto viene creato, esegue il metodo invocato dal client (che deve passargli tutti i parametri necessari all'elaborazione) e quindi viene rimosso dall'AS (con politiche solo parzialmente definibili dall'utente). Si noti che il bean può avere parametri locali predefiniti e persistenti, ma tali parametri sono definiti in fase di configurazione (deploy) e non dipendono dal client. Un esempio di bean di questo genere può essere quello che contiene funzioni matematiche di calcolo.*

- *Stateful: bean che vive per un tempo superiore a quello di esecuzione di un metodo, consentendo una conversazione con il client. La rimozione dell'oggetto può essere richiesta dal programmatore nel codice, oppure gestita, tramite timeout, dall'AS. Il bean, durante tutto il colloquio con il client, mantiene lo stato dell'elaborazione e le informazioni sullo specifico client chiamante, consentendo una conversazione asincrona ed esclusiva. Esempi di bean stateful sono tutti quelli in cui è necessario un colloquio continuo tra client e server, come la gestione di un carrello in una applicazione di commercio elettronico.*

Una volta deciso il tipo di SessionBean da sviluppare, vediamo la struttura del codice dei vari elementi da sviluppare. Supponiamo, ad esempio, di voler sviluppare il classico metodo che ritorna una stringa come metodo di uno stateless SessionBean, con il solo ausilio della interfaccia remote.
Come evidenziato anche in RMI, nei paradigmi ad oggetti distribuiti la prima cosa da fare è definire l'interfaccia *remote*, nella quale dichiareremo il nostro metodo di business *ciao()*:

```
import javax.ejb.EJBObject;

public interface SampleSession extends EJBObject {
   public String ciao() throws java.rmi.RemoteException;
}
```

L'interfaccia *remote* estende la classe *javax.ejb.EJBObject*; in questo modo viene detto al container quali sono i metodi delegati all'istanza EJB Object. Dal momento che può essere invocato da remoto, il metodo *ciao()* (la cui logica deve essere dichiarata nel codice del bean) emette una *RemoteException* i modo da poter segnalare eventuali problemi nella chiamata RMI (o, nella specifica 2.0, RMI-IIOP).
Il secondo file da sviluppare (ma, di solito, il suo sviluppo è automaticamente generato dai tools di sviluppo) è quello inerente l'interfaccia Ejb Home.

21.9 File SampleSessionHome.java

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import java.rmi.RemoteException;

public interface SampleSessionHome extends EJBHome {
        SampleSession create() throws CreateException, RemoteException;
}
```

L'interfaccia SampleSessionHome estende la classe javax.ejb.EJBHome, segnalando al container che questa classe è quella delegata alle funzioni di Home object. Il metodo necessariamente presente deve essere il create(), necessario per la creazione dell'oggetto; tale metodo ritorna una istanza del bean SampleSession. Si noti come, oltre alla RemoteException, il metodo ritorni anche l'eccezione CreateException, nel caso in cui la richiesta di creazione dell'oggetto dovesse fallire.

Il file contenente il "vero" session bean è presentato nel listato seguente.

21.10 File SampleSessionBean.java

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.CreateException;

public class SampleSessionBean implements SessionBean {
   private SessionContext ctx;

   public void setSessionContext(SessionContext context) throws RemoteException {
      ctx = context;
   }

   public void ejbActivate() {
   }
```

```
    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    public void ejbCreate() throws CreateException {
          // Qui va scritto eventuale codice da inserire nella creazione dell'oggetto
       }
    public String ciao() {
       return "ciao da SessionSample";
    }
}
```

Come si vede dall'esempio, la classe SampleSessionBean dichiara il proprio status di EJB implementando l'interfaccia javax.ejb.SessionBean; questo comporta la definizione di ben cinque metodi, oltre quelli definiti dall'utente.

Metodi da definire	
Nome Metodo	**Descrizione**
public void ejbCreate()	Definisce la logica da implementare in fase di creazione del bean
public void setSessionContext(SessionContext context)	Definisce la creazione del contesto di riferimento del bean
public void ejbActivate()	Definisce le politiche da attuare nell'attivazione dell'oggetto
public void ejbPassivate()	Definisce le politiche da attuare nella passivazione dell'oggetto
public void ejbRemove()	Definisce le politiche da attuare nella rimozione dell'oggetto

I metodi *ejbActivate()* e *ejbPassivate()* richiamano al concetto di bean pooling ed introducono il concetto della persistenza del bean. Al fine di risparmiare risorse, l'AS provvede a eliminare dalla memoria gli oggetti che, pur dovendo rimanere persistenti, sono meno usati. Quando un Ejb non viene utilizzato per un certo tempo (configurabile), l'AS provvede *passivarlo*, ossia a salvarlo su disco, mantenedo in memoria solamente un riferimento all'handle dell'oggetto in modo da poterlo recuperare. Nel momento in cui un client tenta di invocare di nuovo l'EJB passivato, l'AS provvede a *attivarlo*, ossia a ricaricare in memoria l'oggetto. In verità, per poter ricostruire l'oggetto a partire dal suo salvataggio su disco, l'AS potrebbe non avere la necessità di salvare tutto l'oggetto, ma solo il suo stato ed il suo contesto di riferimento. In tale caso il processo di attivazione associa lo stato ed il contesto salvati ad una istanza del bean (che, pertanto, potrebbe non essere quella originale).

A queste due operazioni, eseguite per default dall'AS, possono essere associate delle azioni da intraprendere, e la specifica delle azioni va inserita nei due metodi succitati.
Si noti che:

- *il processo di passivazione, nel caso dei SessionBeans, coinvolge solo gli stateful beans, non gli stateless. Per questi ultimi, i metodi ejbPassivate() e ejbActivate() sono inutili;*
- *nella operazione di passivazione l'oggetto viene scritto su uno stream, e questa operazione è sicura solo se tutti gli elementi in esso contenuti (che concorrono a determinarne lo stato) implementano l'interfaccia java.io.Serializable;*
- *il processo di passivazione interrompe gli stream eventualmente aperti dall'oggetto; questi vanno necessariamente chiusi nella ejbPassivate() ed eventualmente riaperti nella ejbActivate().*

Ad esempio, nel caso l'Ejb abbia aperto un socket o sia in lettura su un file, è necessario definire nella *ejbPassivate()* il codice necessario per chiudere il socket (o il file) ed il codice per riaprirlo nella *ejbActivate()*.
Il metodo *ejbCreate()* è utile per inizializzare l'oggetto nel momento in cui viene istanziato, così come il metodo *ejbRemove()* ci consente una deallocazione pulita dell'oggetto, evitando carichi al garbage collector.
Il metodo *setSessionContext()* è necessario al container per comunicare all'istanza del bean il suo contesto di riferimento, intendendo con contesto tutte le informazioni d'ambiente necessarie al bean per svolgere il proprio lavoro (ad esempio, variabili d'ambiente o informazioni utente sul client connesso). Il metodo *setSessionContext()* viene invocato dal container durante la fase di creazione dell'oggetto ed il context deve essere salvato in una variabile locale per poterne interrogare le proprietà durante il ciclo di vita dell'oggetto.

21.11 Ciclo di vita di un SessionBean

I vari metodi definiti nel bean non possono essere chiamati in qualsiasi momento, ma la loro invocazione, da parte del client o del container, dipende dallo stato del bean. Nella figura seguente è mostrato il ciclo di vita di uno stateful session bean:

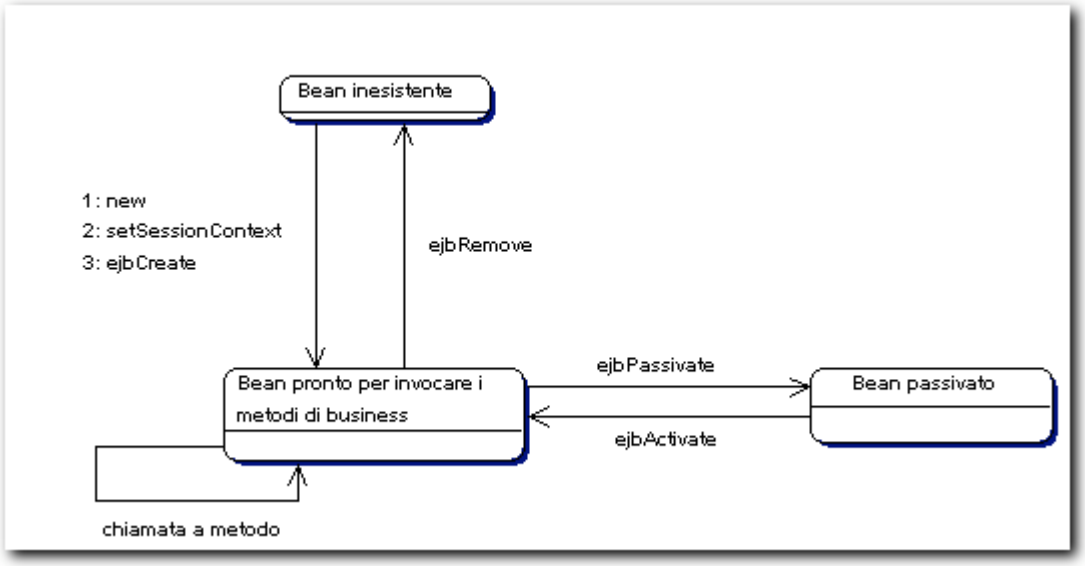


Figura 149: statechart diagram relativo ciclo di vita di uno Stateful SessionBean

Uno stateful SessionBean possiede due stati possibili: attivo e passivato.
Per poter essere definito come bean attivo, deve essere caricato dal container con le chiamate dei metodi proposti; in questo stato può ricevere le chiamate verso i metodi di business provenienti dal client.
Uno Stateless SessionBean, una volta creato, possiede solo lo stato attivo, come definito nella figura seguente:

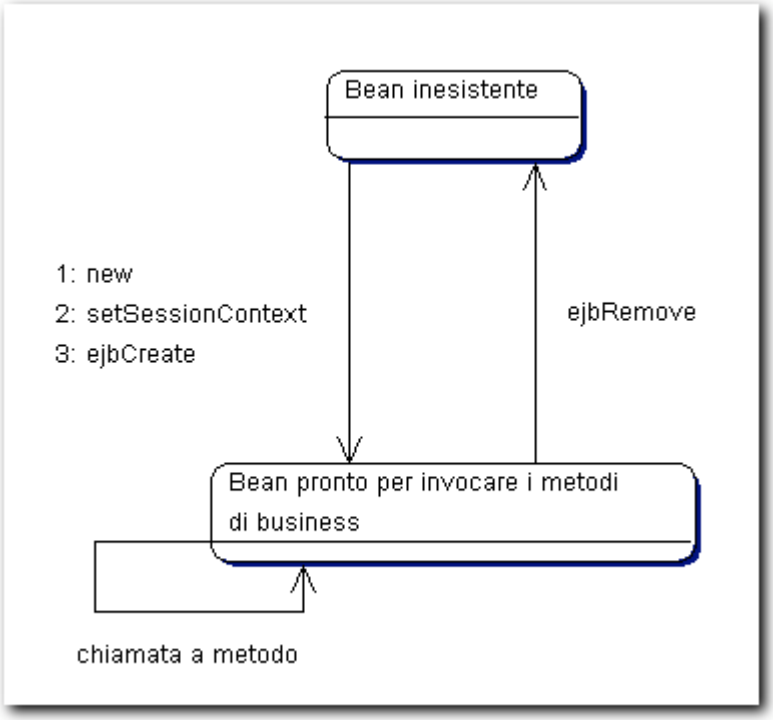


Figura 150: statechart diagram relativo ciclo di vita di uno Stateless SessionBean

In entrambi i casi la fase critica è quella relativa alla creazione del bean e di tutta la catena degli oggetti associati.
Per meglio comprendere la sequenza delle chiamate dei vari metodi definiti ed il ruolo del container, è utile analizzare il seguente sequence diagram, nel quale è esemplificata la sequenza delle chiamate dei metodi nella fase di creazione del SessionBean:

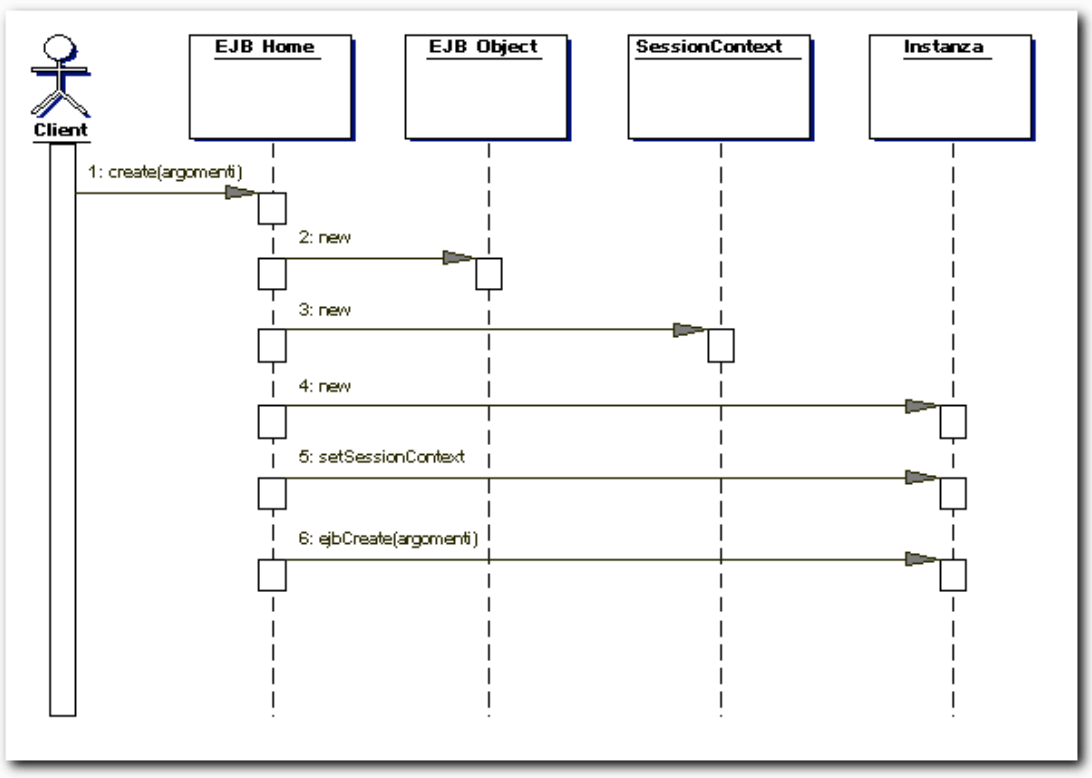


Figura 151: sequence diagram relativo alla fase di creazione del bean

Alla invocazione della *create()* da parte di un client, il container provvede alla ceazione dell'EJBObject, di un contesto e di una istanza del bean. Tutti questi elementi concorrono alla inizializzazione dell'istanza del bean, sulla quale il container invoca il metodo *setSessionContext()* per inizializzarne lo stato e quindo ne invoca la *ejbCreate()* ove vengono definite le logiche definite dal programmatore per la creazione del bean.

21.12 Entity Bean: Definizioni e tipi

Uno dei componenti essenziali di qualsiasi architettura software, ed in particolare di quelle enterprise, è rappresentato dalla necessità di trattare entità che rimangano persistenti per un periodo arbitrario di tempo.

Generalmente lo strumento più utilizzato per gestire la persistenza di dati è rappresentato dai database, con particolare riguardo per quelli relazionali, che rappresentano la gran parte del mercato di riferiemento.
Gli EntityBeans rappresentano il meccanismo J2EE per gestire la persistenza di oggetti; in particolare, secondo la specifica SUN, gli EntityBeans sono la rappresentazione Object Oriented di entità persistenti. Dal momento che il più comune strumento di gestione di entità organizzate e persistenti è rappresentato dai database relazionali, possiamo affermare che gli EntityBeans rappresentano il meccanismo J2EE per virtualizzare l'accesso ai database relazionali. In estrema sintesi, un EntityBean è (nei casi più semplici, ma anche più comuni) la rappresentazione ad oggetti di una tabella di un database relazionale (RDBMS), e l'istanza del bean è la rappresentazione ad oggetti di una riga di una tabella di un RDBMS[12].
La domanda può sorgere spontanea: che senso ha aggiungere un nuovo componente architetturale per l'accesso ai database quando è già presente il paradigma JDBC?
Innanzitutto JDBC rappresenta uno strato che disaccoppia il codice Java dalle caratteristiche specifiche del database, fornendo una interfaccia generica per la comunicazione con qualsiasi RDBMS standard. Tale interfaccia, però, non riduce nel codice la conoscenza dello schema fisico del database e soprattutto non elimina il codice SQL, costringendo il programmatore ad occuparsi della mappatura dei tipi dati del database e di gestire il binding.
Il vantaggio di avere EntityBean è quello di poter mappare nel modello ad oggetti il modello relazionale che, nativo del database, è invece alieno alla programmazione Java. Questa mappatura, oltre a consentire una programmazione più naturale ed omogenea, elimina parecchio codice SQL. E', inoltre, possibile utilizzare i servizi offerti dal container per gestire la comunicazione con il database, delegando dal codice alla configurazione la gestione delle transazioni e la gestione delle connessioni contemporanee (*connection pooling*).
Un'altra problematica in cui ci è utile il container è quella della gestione della sincronizzazione tra i valori presenti sul database e la loro rappresentazione nel bean: ogni nuova istanza di un bean deve generare un nuovo record nella tabella, così come ad ogni modifica dei valori delle variabili del bean deve corrispondere una update dei dati sul database. Questo processo di allineamento e di sincronizzazione della rappresentazione dei dati è detto persistenza. Nella specifica 2 di Ejb vengono definiti due tipi di EntityBean, e la distinzione verte proprio sulla gestione della persistenza:

[12] In realtà la definizione non è propriamente corretta, dal momento che quello di associare un bean ad una tabella è solo uno dei possibili modi di disegno, ma fornisce una idea chiara e semplice del concetto associato ad un Entity Bean

Persistenza di un EJB	
Tipo di persistenza	**Descrizione**
CMP (Container Managed Persistence)	Nei bean di tipo CMP è il container a gestire automaticamente la persistenza. Il programmatore deve occuparsi solo di gestire la logica applicativa, senza doversi occupare della persistenza. Per poter utilizzare questo tipo di bean è, però, necessario comunicare al container (in fase di configurazione) la lista di corrispondenza di quali campi del bean rappresentino attributi di tabelle. Questa configurazione può essere anche complessa, ma consente una astrazione maggiore non legando lo sviluppo alla struttura del database.
BMP (Bean Managed Persistence)	Nei bean di tipo BMP il carico della gestione della persistenza è a carico del programmatore. In pratica è il programmatore che deve occuparsi, ogni volta che deve inserire o modificare dati sul database, di farlo esplicitamente attraverso codice di tipo SQL.

21.13 Struttura di un EntityBean

Oltre alla gestione della persistenza secondo gli schemi suddetti (CMP e BMP), il modello EntityBean ha, rispetto ai Session Beans un altro elemento distintivo, ossia la presenza di uno speciale identificativo dell'oggetto. Dal momento che la corrispondenza logica dell'oggetto è una tabella di un database, è lecito pensare che il bean debba (così com'è in un database con la chiave primaria di una tabella) avere un identificativo speciale.
Tale identificativo non è un attributo, ma viene codificato con una classe da definire in un nuovo file, ed aggiunto alla struttura standard dei files degli EJB. Questo file è, di solito, indicato con la desinenza PK (che sta per Primary Key – chiave primaria) aggiunta al nome del file di interfaccia remote del bean.
Quindi un entity bean (ad esempio, per un generico bean *EntityDiEsempio*) è definito da:

- I file standard EJB: *EntityDiEsempio.java* (interfaccia) *EntityDiEsempioBean.java* (Codice del bean vero e proprio), *EntityDiEsempioHome.java* (Interfaccia Home)
- Il file di chiave primaria *EntityDiEsempioPK.java*

E' importante non confondere la chiave primaria con l'*handle* del bean. Ogni componente oggetto (istanza home e componente oggetto di Session, Entity o MessageDriven) è identificata dal container mediante un identificativo univoco generato dal container stesso, detto handle. Tale identificativo è utile, ad esempio dal lato client, per poter riferire una stessa istanza durante più chiamate succesive (per ottenere l'handle di un oggetto, è definito il metodo *getHandle()*). La chiave primaria, invece, è un identificativo logico, inerente il contenuto informativo del bean ed indipendente dall'istanza.

Vediamo, ora, la struttura di un EntityBean analizzando i metodi specifici definiti per i vari elementi costitutivi del componente.
Per l'interfaccia Home i metodi obbligatori sono:

Metodi di un EntityBean	
Nome Metodo	**Descrizione**
public void create()	Costruttore del bean
public … findByPrimaryKey (…)	Metodo per la ricerca del bean a partire dalla chiave primaria
public … ejbFind … (…)	Metodi per la ricerca del bean

Il metodo *create* ha una funzione speciale negli EntityBeans. Dal momento che gli EntityBean rappresentano i dati presenti nelle tabelle, il metodo *create* è quello che consente di creare nuove istanze del bean, cui deve corrispondere una rappresentazione persistente, ossia nuovi record nella tabella. Di fatto, nel caso in cui un bean rappresenti una tabella, invocare la *create* equivale a creare un nuovo record su tabella.
I *finder methods*, ossia i metodi di ricerca, sono metodi specifici degli EntityBean e la loro definizione e comprensione rappresenta un elemento fondamentale in fase di disegno del bean. Continuando ad associare un EntityBean ad una tabella di un database, l'utilizzo di un finder method di un EntityBean equivale ad effettuare una query di selezione sul database. Così come una SELECT sul database ritorna una serie di record, così un finder method può ritornare una collezione di oggetti che rappresentano il record. Nel caso in cui il metodo possa ritornare solo una istanza (ad esempio, nel caso della *findByPrimaryKey*), il tipo ritornato dal metodo è un'istanza dell'oggetto bean stesso, in modo da consentire al chiamante di manipolare, attraverso i metodi esposti nelle interfacce local e/o remote, i valori del bean, e quindi del record della tabella corrispondente alla chiave primaria.
Per l'interfaccia EntityBean (che definisce il bean vero e proprio) i metodi da definire sono:

Interfaccia EntityBean	
Nome Metodo	**Descrizione**

public void ejbCreate()	Definisce la logica da implementare in fase di creazione del bean
public void ejbPostCreate()	Definisce le azioni da compiere dopo la create()
public void setEntityContext(EntityContext context)	Definisce la creazione del contesto di riferimento del bean
public void unsetEntityContext(EntityContext context)	Invocato dal container, elimina l'associazione del bean con il suo contesto, preparando il bean alla sua distruzione.
public void ejbLoad()	Serve per caricare i dati del database dentro al bean
public void ejbStore()	Serve per rendere persistenti sul database i dati contenuti nel bean
public void ejbActivate()	Definisce le politiche da attuare nell'attivazione dell'oggetto
public void ejbPassivate()	Definisce le politiche da attuare nella passivazione dell'oggetto
public void ejbRemove()	Definisce le politiche da attuare nella rimozione dell'oggetto

Rispetto ai SessionBeans, troviamo il metodo *setEntityContext* che ha significato analogo al metodo *setSessionContext*, e sono presenti quattro nuovi metodi tipici degli EntityBean: *unsetEntityContext* ,*ejbPostCreate*, *ejbLoad* ed *ejbStore*.
Il metodo ejbPostCreate viene eseguito, invocato dal container, immediatamente dopo l'associato metodo ejbCreate. Dal momento che l'ejbCreate è il metodo che il container invoca alla chiamata del metodo create() della interfaccia Home, esso definisce la creazione del bean (e, quindi, della creazione di un record sulla sottostante tabella), non lasciando spazio alla creazione di altra logica oltre quella di creazione. Il metodo ejbPostCreate serve proprio per definire logiche locali e ad esempio, inizializzare variabili utili all'implementazione della logica di business. Si noti che possono essere definiti più metodi *ejbCreate* e per ciascuno di essi va definito un corrispondente *ejbPostCreate*.
Il metodo *unsetEntityContext* interviene in fase di distruzione dell'istanza dell'oggetto e prepara il bean al ciclo di garbage collection liberando le risorse allocate con il metodo *setEntityContext*.
Il metodo ejbLoad, invece, interviene qualora si voglia caricare dati dalla base dati, mentre il metodo ejbStore viene invocato quando è necessario effettuare update sulla tabella sottostante, rendendo persistenti sul database le modifiche effettuate sui campi del bean.

Per capire meglio come funzioni un entity bean, e quali siano i vantaggi nell'uso di questa tecnologia, scriviamo un semplice esempio di bean per la tabella impiegati utilizzata nel capitolo JDBC.

21.14 Un esempio di EntityBean CMP

L'esempio di bean CMP ci consente di evidenziare maggiormente le differenze rispetto alla classica programmazione con JDBC.
Al solito, iniziamo la scrittura del codice dall'interfaccia. In questo caso utilizzeremo, come nel caso del Session Bean del capitolo precedente, l'interfaccia *remote*, ma anche una interfaccia *local*. E' bene sapere, però, che il modo più corretto per accedere ad un EntityBean è quello di utilizzare SessionBeans per interfacciare gli EntityBeans, utilizzando solo le interfacce local per gli Entity e delegando l'accesso da remoto per i Session.
Nel caso di utilizzo di interfacce local, tipicamente la struttura dei nomi dei file (per il bean *Nome*) segue il seguente schema:

Struttura dei nomi dei file	
Nome File	**Tipo di file**
Nome.java	Interfaccia Local
NomeHome.java	Interfaccia localHome
NomeBean.java	Sorgente del Bean vero e proprio
NomeRemote.java	Interfaccia Remote
NomeRemoteHome.java	Interfaccia remoteHome
Nome.java	Interfaccia Local

Il bean che creeremo è una rappresentazione della tabella IMPIEGATI così definita:

Struttura della tabella IMPIEGATI		
Nome dell'attributo	**Tipo**	**Vincoli**
IDImpiegato	Intero	Chiave primaria
Nome	Stringa di caratteri	Obbligatorio
Cognome	Stringa di caratteri	Obbligatorio
TelefonoUfficio	Stringa di caratteri	

Iniziamo, quindi, con la scrittura delle interfacce local e remote

```
package sampleentity;

import javax.ejb.*;
```

```
import java.util.*;

public interface ImpiegatiCMP extends javax.ejb.EJBLocalObject {
	public java.lang.Integer getIDImpiegato();
	public void setNome(java.lang.String nome);
	public java.lang.String getNome();
	public void setCognome(java.lang.String cognome);
	public java.lang.String getCognome();
	public void setTelefonoUfficio(java.lang.String telefonoUfficio);
	public java.lang.String getTelefonoUfficio();
}
```

```
package sampleentity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface ImpiegatiCMPRemote extends javax.ejb.EJBObject {
	public java.lang.Integer getIDImpiegato() throws RemoteException;
	public void setNome(java.lang.String nome) throws RemoteException;
	public java.lang.String getNome() throws RemoteException;
	public void setCognome(java.lang.String cognome) throws RemoteException;
	public java.lang.String getCognome() throws RemoteException;
	public void setTelefonoUfficio(java.lang.String telefonoUfficio) throws RemoteException;
	public java.lang.String getTelefonoUfficio() throws RemoteException;
}
```

Come si vede dalla struttura dei files, i campi della tabella sono quattro: IDImpiegato (la chiave primaria), nome, cognome, TelefonoUfficio (campi informativi). A ciascuno dei campi informativi della tabella corrispondono due metodi set e get per la scrittura e la lettura del campo stesso, disponibili sia per l'interfaccia locale che per quella remota. Per la chiave primaria è presente solo il metodo di lettura del campo.
Si noti, inoltre, come nell'interfaccia *local* i metodi non emettano alcuna eccezione; questo avviene poiché non vengono chiamati da remoto, e pertanto non è possibile che si verifichino problemi di RMI.

A questo punto è necessario definire le classi Home:

```
package sampleentity;

import javax.ejb.*;

public interface ImpiegatiCMPHome extends javax.ejb.EJBLocalHome {
	public ImpiegatiCMP create(java.lang.Integer iDImpiegato)
		throws CreateException;
	public ImpiegatiCMP findByPrimaryKey(java.lang.Integer iDImpiegato)
		throws FinderException;
}
```

```
package sampleentity;

import javax.ejb.*;
```

```
import java.util.*;
import java.rmi.*;

public interface ImpiegatiCMPRemoteHome extends javax.ejb.EJBHome {
        public ImpiegatiCMPRemote create(java.lang.Integer iDImpiegato)
                throws CreateException, RemoteException;
        public ImpiegatiCMPRemote findByPrimaryKey(java.lang.Integer iDImpiegato)
                throws FinderException, RemoteException;
}
```

In questo caso l'unico metodo di ricerca implementato nelle interfacce è il findByPrimaryKey(); qualora sia necessario, è possibile definire tutti i metodi di ricerca richiesti nelle interfacce Home.
Si noti il ruolo delle eccezioni nelle definizioni dei metodi: l'uso di eccezioni specifiche per la creazione dell'oggetto (*CreateException*) e per la ricerca (*FinderException*) consente di valutare immediatamente le conseguenze di errori applicativi nell'inserimento e nella ricerca di righe nella tabella sottostante alla definizione del bean.
Veniamo ora alla definizione del codice del bean.

```
package sampleentity;

import javax.ejb.*;

public abstract class ImpiegatiCMPBean implements EntityBean {

        EntityContext entityContext;

        public java.lang.Integer ejbCreate(java.lang.Integer iDImpiegato) throws CreateException {
                setIDImpiegato(iDImpiegato);
                return null;
        }

        public void ejbPostCreate(java.lang.Integer iDImpiegato) throws CreateException {
        }

        public void ejbRemove() throws RemoveException {
        }

        public abstract void setIDImpiegato(java.lang.Integer iDImpiegato);
        public abstract void setNome(java.lang.String nome);
        public abstract void setCognome(java.lang.String cognome);
        public abstract void setTelefonoUfficio(java.lang.String telefonoUfficio);
        public abstract java.lang.String getTelefonoUfficio();
        public abstract java.lang.String getCognome();
        public abstract java.lang.String getNome();
        public abstract java.lang.Integer getIDImpiegato();

        public void ejbLoad() {
        }

        public void ejbStore() {
        }

        public void ejbActivate() {
```

```
	}

	public void ejbPassivate() {
	}

	public void unsetEntityContext() {
		this.entityContext = null;
	}
	public void setEntityContext(EntityContext entityContext) {
		this.entityContext = entityContext;
	}
}
```

La prima cosa che si nota è il fatto che la classe ed i metodi set e get relativi agli attributi informativi della tabella, siano tutti **abstract**.
Questo è derivato dalla scelta di definire il bean come CMP. La gestione degli accessi alla base dati e della persistenza è, infatti, gestita interamente dal container che si occupa di generare automaticamente i metodi più opportuni per interfacciarsi con la base dati.
Questo è vero anche nel caso in cui si vogliano creare, ad esempio, altri metodi di ricerca: le regole per effettuare query complesse non vanno comunque descritte nel codice del bean ma nel file descrittore del deploy. Questo disaccoppia la scrittura del codice dal processo di deploy, rendendo il codice più snello e manutenibile. Per la scrittura delle eventuali query all'interno dei file di deploy viene utilizzato un linguaggio SQL-like, chiamato EJB-QL, di cui parleremo in seguito.

21.15 Un esempio di EntityBean BMP

Vediamo ora un esempio di EntityBean BMP; come detto, nel caso di uso di bean BMP, la scrittura delle query è delegata al programmatore, che deve inserire il codice SQL all'interno dei metodi che ne hanno necessità (tipicamente almeno dentro la *ejbCreate*).
Al solito iniziamo dalla scrittura delle interfacce:

```
package sampleentity;

import javax.ejb.*;
import java.util.*;

public interface ImpiegatiBMP extends javax.ejb.EJBLocalObject {
	public java.lang.Integer getIDImpiegato();
	public void setNome(java.lang.String nome);
	public java.lang.String getNome();
	public void setCognome(java.lang.String cognome);
	public java.lang.String getCognome();
	public void setTelefonoUfficio(java.lang.String telefonoUfficio);
	public java.lang.String getTelefonoUfficio();
}
```

```
package sampleentity;
```

```
import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface ImpiegatiBMPRemote extends javax.ejb.EJBObject {
	public java.lang.Integer getIDImpiegato() throws RemoteException;
	public void setNome(java.lang.String nome) throws RemoteException;
	public java.lang.String getNome() throws RemoteException;
	public void setCognome(java.lang.String cognome) throws RemoteException;
	public java.lang.String getCognome() throws RemoteException;
	public void setTelefonoUfficio(java.lang.String telefonoUfficio) throws
RemoteException;
	public java.lang.String getTelefonoUfficio() throws RemoteException;
}
```

```
package sampleentity;

import javax.ejb.*;

public interface ImpiegatiBMPHome extends javax.ejb.EJBLocalHome {
	public ImpiegatiBMP create(java.lang.Integer iDImpiegato)
throws CreateException;
	public ImpiegatiBMP findByPrimaryKey(java.lang.Integer iDImpiegato)
throws FinderException;
}
```

```
package sampleentity;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface ImpiegatiBMPRemoteHome extends javax.ejb.EJBHome {
	public ImpiegatiBMPRemote create(java.lang.Integer iDImpiegato)
		throws CreateException, RemoteException;
	public ImpiegatiBMPRemote findByPrimaryKey(java.lang.Integer iDImpiegato)
		throws FinderException, RemoteException;
}
```

Le classi *Home* e *Remote*, rappresentando solo le interfacce ai metodi di business, non cambiano passando da CMP a BMP. Quello che cambia significativamente è il codice del bean, ma prima di analizzarlo, è necessario introdurre il file ImpiegatiPK contenente il codice relativo alla gestione della chiave primaria:

```
package sampleentity;

import java.io.Serializable;

public class ImpiegatiPK implements java.io.Serializable {
	java.lang.Integer iDImpiegato;

	public ImpiegatiPK (java.lang.Integer iDImpiegato) {
		this.iDImpiegato = iDImpiegato;
	}
```

```
	public String toString(){
		return iDImpiegato.toString();
	}

	public boolean equals(Object o){
		if (o instanceof ImpiegatiPK) {
			ImpiegatiPK otherKey = (ImpiegatiPK)o;
			return (iDImpiegato == otherKey.iDImpiegato);
		}
		else return false;
	}

	public int hashCode(){
		return String.valueOf(iDImpiegato).hashCode();
	}
}
```

Il file descrittivo della chiave primaria è molto semplice e di fatto, definisce il campo *iDImpiegato* come un oggetto serializzabile. E' richieta l'implementazione di tre metodi necessari al container:

File descrittivo della chiave primaria	
Nome Metodo	**Descriozione**
toString()	Ritorna una descrizione testuale della chiave primaria; nel caso la chiave primaria sia complessa, ossia formata da più attributi, è necessario costruirla.
equals(Object)	Metodo invocato dal container per comparare chiavi primarie.
hashCode()	Metodo necessario al container per poter utilizzare una Hashtable come meccanismo di memorizzazione delle chiavi primarie.

Vediamo ora il codice del bean:

```
package sampleentity;

import javax.ejb.*;
import java.sql.*;
import java.naming.*;

public class ImpiegatiBMPBean implements EntityBean {

	EntityContext entityContext;
	Integer iDImpiegato;
	String cognome;
	String nome;
	String telefonoUfficio;
```

```
public java.lang.Integer ejbCreate(java.lang.Integer iDImpiegato,
                                   String nome,
                                   String cognome) throws CreateException
{
        PreparedStatement ps =null;
        Connection conn =null;
        try {
                this.iDImpiegato = iDImpiegato ;
                this.nome = nome;
                this.cognome =cognome;

                //Connessione al database
                conn =getConnection();
                //Inserimento del record nel database
                ps = conn.prepareStatement("INSERT INTO IMPIEGATI (IDIMPIEGATO,
                                                NOME, COGNOME) VALUES (?,?,?)");
                ps.setInt (1,iDImpiegato.intValue());
                ps.setString(2,nome);
                ps.setString(3,cognome);
                ps.executeUpdate();

                // Generazione della chiave primaria
                return new ImpiegatiPK(iDImpiegato);
        }
        catch (Exception e){
                throw new CreateException(e.toString());
        }
        finally {
                // Chiudiamo la connessione al database
                try {
                        if (ps !=null) ps.close();
                        if (conn !=null)conn.close();
                }
                catch (Exception e){}
        }
}

public void ejbPostCreate(java.lang.Integer iDImpiegato) throws CreateException {
}

public void ejbRemove() throws RemoveException {
        ImpiegatiPK pk =(ImpiegatiPK)ctx.getPrimaryKey();
        Integer id =pk.iDImpiegato;
        PreparedStatement pstmt =null;
        Connection conn =null;
        try {
                conn =getConnection();
                // Rimozione del record dal database
                ps =conn.prepareStatement("DELETE FROM IMPIEGATI WHERE
                                                IDIMPIEGATO =?");
                ps.setInt (1,id.intValue());
                if (ps.execut eUpdate()==0){
                        throw new RemoveException("Errore in fase di delete");
                }
        }
        catch (Exception ex){
        }
        finally {
                // Chiudiamo la connessione al database
```

```
                    try {
                        if (ps !=null) ps.close();
                        if (conn !=null)conn.close();
                    }
                    catch (Exception e){}
                }
        }

        public void setIDImpiegato(java.lang.Integer iDImpiegato) {
                this.iDImpiegato = iDImpiegato;
        }

        public void setNome(java.lang.String nome) {
                this.nome = nome;
        }
        public void setCognome(java.lang.String cognome) {
                this.cognome = cognome;
        }
        public void setTelefonoUfficio(java.lang.String telefonoUfficio) {
                this.telefonoUfficio = telefonoUfficio;
        }
        public java.lang.String getTelefonoUfficio() {
                return this.telefonoUfficio;
        }
        public java.lang.String getCognome(){
                return this.cognome;
        }
        public java.lang.String getNome(){
                return this.nome;
        }
        public java.lang.Integer getIDImpiegato(){
                return this.iDImpiegato;
        }

        public void ejbLoad() {
                ImpiegatiPK pk =(ImpiegatiPK)ctx.getPrimaryKey();
                Integer id =pk.iDImpiegato;
                PreparedStatement ps =null;
                Connection conn =null;
                try {
                        conn =getConnection();
                        // Otteniamo i dati dal database mediante query SQL
                         ps =conn.prepareStatement("SELECT NOME, COGNOME,
                        TELEFONOUFFICIO FROM IMPIEGATI WHERE IDIMPIEGATO=?");
                        ps.setInt (1,id.intValue());
                        ResultSet rs =ps.executeQuery();
                        rs.next();
                        this.nome =rs.getString("NOME");
                        this.cognome =rs.getString("COGNOME");
                        this.telefonoUfficio =rs.getString("TELEFONOUFFICIO");
                }
                catch (Exception ex){
                        throw new EJBException("Fallito caricamento da database",ex);
                }
                finally {
                        // Chiudiamo la connessione al database
                        try {
                                if (ps !=null) ps.close();
                                if (conn !=null)conn.close();
```

```
		}
		catch (Exception e){}
	}

	public void ejbStore() {
		PreparedStatement ps =null;
		Connection conn =null;
		try {
			// Connession al database
			conn =getConnection();

			// Memorizzazione nel DB
			ps =conn.prepareStatement(	"UPDATE IMPIEGATI SET
			NOME=?,COGNOME=?,TELEFONOUFFICIO=?",
			+"WHERE IDIMPIEGATO =?");
			ps.setString(1,this.nome);
			ps.setString(2,this.cognome);
			ps.setString(3,this.telefonoUfficio);
			ps.setInt (4,this.iDImpiegato.intValue());
			ps.executeUpdate();
		}
		catch (Exception e){
		}
		finally {
			//Rilascio la connessione con il database
			try {
				if (ps !=null) ps.close();
			}
			catch (Exception e){}
			try {
				if (conn !=null) conn.close();
			}
			catch (Exception e){}
		}
	}

	public void ejbActivate() {
	}

	public void ejbPassivate() {
	}

	public void unsetEntityContext() {
		this.entityContext = null;
	}
	public void setEntityContext(EntityContext entityContext) {
		this.entityContext = entityContext;
	}

	public ImpiegatiPK findByPrimaryKey(ImpiegatiPK pk) {
		PreparedStatement ps =null;
		Connection conn =null;
		try {
			conn =getConnection();
			// Otteniamo i dati dal database mediante query SQL
			ps =conn.prepareStatement("SELECT IDIMPIEGATO FROM IMPIEGATI
				WHERE IDIMPIEGATO=?");
			ps.setString(1,pk.toString());
			ResultSet rs = ps.executeQuery();
```

```
                        rs.next();
                        // Se non ci sono stati errori ...
                        return pk;
                }
                catch (Exception ex){
                        throw new EJBException("Fallito caricamento da database",ex);
                }
                finally {
                        // Chiudiamo la connessione al database
                        try {
                                if (ps !=null) ps.close();
                                if (conn !=null)conn.close();
                        }
                        catch (Exception e){}
                }
        }


        /**
        * Recupera il riferimento al bean all'interno del connection pool.
        */
        public Connection getConnection() throws Exception {
                try {
                        Context ctx = new InitialContext();
                        javax.sql.DataSource ds =(javax.sql.DataSource)
                        ctx.lookup("java:comp/env/jdbc/ejbPool");
                        return ds.getConnection();
                }
                catch (Exception e){
                        e.printStackTrace();
                        throw e;
                }
        }
}
```

In estrema sintesi, si può vedere che ad ogni operazione effettuata sul sottostante database (creazione, ricerca, update, delete), corrisponde una equivalente definizione esplicita del corrispondente statement SQL, con notevole appesantimento del codice.
Si noti il metodo *getConnection():* con tale metodo (non obbligatorio, ma utile per incapsulare la logica di connessione) viene effettuato il lookup sul pool delle connessioni, non direttamente sul database; in questo modo il numero di connessioni rimane sotto controllo.

21.16 MessageDriven Beans

I MessageDrive Beans sono strati introdotti nella specifica 2.0 per completare l'architettura J2EE con i paradigmi delle infrastrutture message oriented sfruttando lo strato JMS.
Gli schemi concettuali utilizzati sono quelli tipici delle architetture a messaggi; in prima istanza il modello che prevede che un messaggio sia prodotto da un

ente denominato *producer* ed utilizzato da uno o più enti denominati *consumer*.

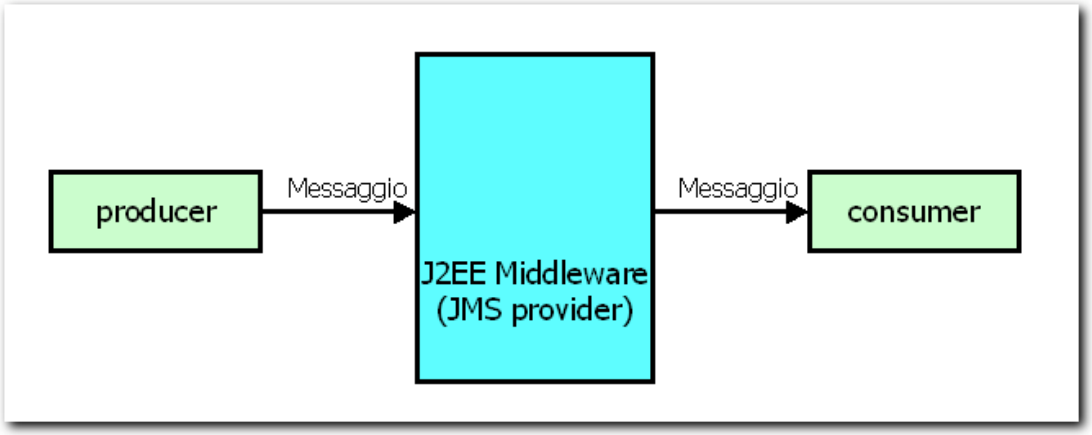


Figura 152: Modello Producer-Consumer

Il provider JMS si pone tra producer e consumer come strato di servizio di gestione dei meccanismi di memorizzazione del messaggio, sincronismo e certificazione della consegna.
I modelli utilizzati per l'implementazione dello schema logico precedente sono due: *punto-punto* e *publisher-subscriber*.
Il modello punto-punto prevede un meccanismo di sincronizzazione dei messaggi basato sul concetto di coda. I messaggi entrano nella coda e ne escono, nell'ordine in cui sono entrati, solo quando il consumer, che deve conoscere i riferimenti esatti della coda per potersi agganciare ad essa, li preleva.
Il modello publisher-subscriber prevede, invece, il concetto di *topic*, ossia di argomento: il producer produce dei messaggi a tema, con uno specifico topic, ed il consumer si registra per le notifiche solo sui messaggi aventi quel tema.
In entrambi i casi il provider JMS si occupa di:

- *avvertire il consumer quando un nuovo messaggio è arrivato, evitando che questi rimanga impegnato in inutili cicli di lettura sulla coda;*
- *mantenere il messaggio nella lista di consegna fino a che tutti i consumer che debbono ricevere il messaggio non lo hanno effettivamente ricevuto; questo implica l'implementazione di logiche store and forward.*
- *certificare, con meccanismi di acknowledge, che il consumer abbia realmente ricevuto il messaggio in forma corretta.*

Questo tipo di architettura presenta dei notevoli vantaggi rispetto alla "normale" chiamata remota di un metodo (ad esempio, utilizzando RMI-IIOP). Le chiamate RMI-IIOP sono essenzialmente delle chiamate in cui un client

invoca un metodo di un server ed attende, in modo sincrono, che questi gli risponda (e nel caso di malfunzionamento del server, il client deve attendere fino al timeout, senza peraltro, ottenere la risposta attesa).
Nella architetture a messaggi (Message Oriented) il consumer ed il producer sono enti disaccoppiati, ciascuno effettua il proprio lavoro in modo autonomo e l'incontro avviene solo per il tramite del middleware. Il consumer non deve, come il generico client RMI-IIOP, attendere in modo sincrono la risposta dal server, ma gestisce in modo autonomo i momenti di lettura dei messaggi; allo stesso modo il server non è obbligato a produrre i risultati delle proprie elaborazioni solo su richiesta, ma può farlo ottimizzando i propri task. E', inoltre, importante sottolineare la sicurezza e la semplicità di implementazione che vengono offerti dallo strato middleware: il programmatore non deve occuparsi di costruire un protocollo di handshake per garantirsi che la comunicazione client-server sia andata a buon fine, ma questo viene garantito dai servizi del middleware; inoltre il middleware fornisce anche gli strumenti di gestione delle eccezioni (nel caso, ad esempio, la comunicazione dovesse interrompersi o il server dovesse andare in blocco). Infine, ma non meno importante, è la considerazione che un singolo task produttore di messaggi può servire un numero qualsiasi di consumer di messaggi, eliminando le problematiche inerenti la gestione multitask o della programmazione concorrente.
Il modello MessageDrivenBeans è in grado di sfruttare tutte questi vantaggi in modo completo, dal momento che il deploy del bean avviene nell'application server, che generalmente contiene un middleware JMS.
Dal punto di vista dello sviluppo, la capacità offerta dal provider JMS di "avvertire" automaticamente il consumer dell'arrivo di un messaggio è la principale caratteristica sfruttata nell'ambito dei MessageDrivenBeans.
Un MessageDrivenBean è un componente che non contiene elementi di logica applicativa, ma sfrutta la intrinseca caratteristica di essere agganciato, in modo nativo, ad un provider JMS. Il suo scopo è quello di essere un concentratore di messaggi, dal momento che il suo ruolo principale è quello del consumer di messaggi.
Per l'interfaccia MessageDrivenBean (che definisce il bean vero e proprio) i metodi da definire sono:

Interfaccia MessageDrivenBean	
Nome metodo	**Descrizione**
public void ejbCreate()	Definisce la logica da implementare in fase di creazione del bean
public void setMessageDrivenContext(MessageDrivenContext context)	Definisce la creazione del contesto di riferimento del bean

public void onMessage(Message messaggio)	E' il metodo che, chiamato dall'AS, avverte il bean che un nuovo messaggio sta arrivando
public void ejbRemove()	Definisce le politiche da attuare nella rimozione dell'oggetto

Il metodo che caratterizza i MessageDriven Beans è l'*onMessage(…)*. Con tale metodo, il container avverte il bean dell'arrivo di un nuovo messaggio, e lo passa come parametro del metodo. Questo metodo è l'unico in cui va scritta un pò di logica applicativa, in particolare va scritto il codice di gestione del messaggio. Qualora la logica applicativa sia onerosa e complessa, è conveniente utilizzare un SessionBean dedicato, e liberare il MessageDriven da ogni onere.
Il MessageDriven Bean, infatti, non fa (e non deve fare) altro che recuperare i messaggi dalla coda o dalla topic cui è agganciato, e in questo suo attendere il messaggio, esso lavora in modo sequenziale. Ogni istanza del bean può lavorare uno ed un solo messaggio alla volta; la logica di gestione di più messaggi concorrenti deve essere delegata al container, che ha la responsabilità di istanziare un numero sufficiente di bean.

21.17 Cenni sul deploy

Come detto in precedenza, il processo di deploy verte sulla installazione degli EJB all'interno del loro contesto di riferimento, ossia l'AS.
Principalmente il processo consiste nella creazione di un archivio (generalmente un file .jar) contenente tutto l'albero delle classi degli EJB compilate, completo di tutti i componenti generati durante la fase di compilazione.
Insieme alle classi, è necessario fornire all'AS delle ulteriori informazioni inerenti i bean e le direttive su come allocarli in memoria.
Ad esempio, nell'esempio di session bean precedentemente descritto, non è specificato se tale bean sia stateless o stateful. Né è specificato quale sia il JNDI name del bean così da poterlo invocare da remoto.
Queste informazioni, insieme, ad altre più specifiche, vanno dichiarate in opportuni file detti descrittori del deploy. Tali file altro non sono che semplici file XML da includere all'interno del jar ove sono archiviate le classi.
Il principale file dedicato al deploy di EJB è il file *ejb-jar.xml*, ove vengono dichiarati tutti gli EJB inclusi nel jar e vengono inserite le informazioni necessarie all'AS per il deploy.
Di seguito è presentato un esempio di tale file ove sono presenti le direttive di deploy per un session bean ed un entità bean:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd'>
<ejb-jar>
   <description></description>
   <enterprise-beans>
      <session>
         <display-name></display-name>
         <ejb-name>SampleSessionBean</ejb-name>
         <home>SampleSessionHome</home>
         <remote>SampleSession</remote>
         <ejb-class>SampleSessionBean</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
      </session>
      <entity>
         <display-name></display-name>
         <ejb-name>SampleEntityBean</ejb-name>
         <home>SampleEntityHome</home>
         <remote>SampleEntity</remote>
         <ejb-class>SampleEntityBean</ejb-class>
         <persistence-type>Bean</persistence-type>
         <prim-key-class>SampleEntityPK</prim-key-class>
         <reentrant>False</reentrant>
      </entity>
   </enterprise-beans>
   <relationships>
      <ejb-relation>
         <ejb-relation-name></ejb-relation-name>
         <ejb-relationship-role>
            <multiplicity>1</multiplicity>
            <relationship-role-source>
               <description></description>
               <ejb-name></ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <multiplicity>1,1</multiplicity>
            <relationship-role-source>
               <description></description>
               <ejb-name>SampleEntity</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>
   </relationships>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>SampleSessionBean</ejb-name>
            <method-name>*</method-name>
         </method>
```

```
        <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
      <container-transaction>
        <method>
          <ejb-name>SampleEntityBean</ejb-name>
          <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

Nella prima parte del file è presente il tag <enterprise-beans>, al cui interno sono definiti i nomi del bean, delle sue interfacce home e remote (in questo caso non si è fatto uso di local). Per il primo bean, il session SampleSessionBean, si noti il tag <Session-type> il cui valore indica se il bean è stateful o stateless. Per il secondo bean, l'entity SampleEntityBean, si noti, invece, la presenza del tag <persistence-type> che indica se si tratta di BMP o CMP.
La generazione dei descrittori XML per il deploy è, generalmente, a carico degli strumenti di sviluppo, i quali sono in grado di generare automaticamente sia l'*ejb-jar.xml*, sia gli eventuali altri file necessari per il deploy su AS specifici. Generalmente ogni AS, infatti, necessita di informazioni specifiche, che vengono inserite in altri file secondo le direttive fornite dal vendor dell'AS.
Un'ultima osservazione è relativa alle direttive JNDI. Normalmente queste direttive sono presenti in un altro file XML, denominato *jndi-definitions.xml*, già presente nell'AS e che va integrato con le informazioni relative ai beans dei quali si sta effettuando il deploy.

21.18 Il linguaggio EJB-QL

Nella versione 2.0 della specifica EJB la SUN ha introdotto un linguaggio simile all'SQL definire le query che riguardano gli EJB, da utilizzare sia all'interno del codice (nel caso di bean BMP) sia nei file di deploy. Questo linguaggio è una sorta di SQL ad oggetti, con sintassi simile allo standard SQL, e che viene tradotto in SQL standard dal container. Senza approfondire tutti i dettagli del linguaggio, vediamo un semplice esempio.
Supponiamo di voler effettuare, in EJB-QL, una semplice query sulla già citata tabella IMPIEGATI, per trovare tutti gli impiegati con un certo nome; la query EJB-QL è:

```
SELECT OBJECT(a) FROM IMPIEGATI AS a WHERE NOME = ?1
```

Come si nota immediatamente, la differenza fondamentale rispetto all'SQL standard, è che il ritorno della query non è un'elenco di attributi ma un oggetto (o una collezione di oggetti), che rappresenta il corrispondente recod.

21.19 L'importanza degli strumenti di sviluppo

Quando Java fece la sua comparsa nel 1995, non esistevano ambienti e strumenti di sviluppo specifici; di solito era sufficiente utilizzare un qualsiasi editor di testo e il Java Development Kit, per scrivere applet o piccole application era più che sufficiente. Con il crescere delle aree di interesse coperte da linguaggio, le applicazioni sviluppate sono divenute sempre più complesse ed articolate, rendendo utile l'utilizzo di strumenti di sviluppo dedicati al linguaggio. Se questo è normalmente vero nella realizzazione di interfacce grafiche, dove la presenza di strumenti GUI migliora notevolmente la produttività, questo forse è ancora più vero nello sviluppo di componenti EJB. La complessità di tali componenti, in termini di numero di classi generate, generazione dei descrittori XML e creazione di ambienti di test, rende praticamente necessario l'utilizzo di un buono strumento di sviluppo Java.
Si consideri che uno strumento di sviluppo Java:

- crea automaticamente le strutture dei files degli EJB;
- spesso presenta utilità grafiche per la creazione rapida di metodi e proprietà, generando automaticamente le corrispondenti interfacce home, local e remote;
- presenta utility per automatizzare la connessione a database e la creazione di query;
- presenta utility per la creazione automatica di bean CMP a partire dalla struttura della tabella;
- presenta utility per effettuare la comunicazione con provider JMS e/o CORBA;
- è in grado di effettuare una compilazione completa ed incrementale;
- è in grado di generare i descrittori XML standard per il deploy;
- fornisce utilità per la creazione automatica dell'archivio di cui effettuare il deploy;
- quando è integrato con un certo AS, consente di generare anche i descrittori custom per il deploy sullo specifico AS;
- quando è integrato con un certo AS, consente di effettuare il deploy in automatico sull'AS;
- fornisce utility per la creazione automatica di test client, per verificare rapidamente le funzionalità dei componenti.

22 INTRODUZIONE AI WEBSERVICES

22.1 Introduzione

La piattaforma J2EE è la soluzione Sun per la costruzione di infrastrutture informatiche aziendali, grazie alle caratteristiche di integrazione con i sistemi legacy, cross platform, scalabilità ed i meccanismi di recovery e sicurezza. E' pertanto, un'infrastruttura completa per la costruzione di quello che viene definito, con una frase che va molto di moda, il sistema nervoso digitale dell'azienda.
Ma cosa succede quando è necessario far comunicare aziende differenti, appartenenti a mercati diversi ed aventi infrastrutture tecnologiche proprietarie (interoperabilità fra sistemi)? Le architetture Java Enterprise stanno rapidamente prendendo piede, ma rappresentano ancora un segmento di mercato specifico e non molto diffuso. Molte aziende utilizzano altre piattaforme tecnologiche, ed anche quando riconoscono il valore dell'architettura Java cercano (dal loro punto di vista, giustamente) di preservare gli investimenti rinviando la costruzione di nuove architetture sino al momento in cui diventa effettivamente necessario.
Questo problema è di non facile soluzione, e, fino ad ora, era stato oggetto di analisi e sviluppi specifici, non essendo presente alcuna piattaforma tecnologica comune.
La grande svolta nel processo di standardizzazione delle comunicazioni, avvenuta con l'avvento e la diffusione su larga scala dei protocolli TCP/IP prima e HTTP poi, ha semplificato, ma non ha risolto, i problemi di incomunicabilità dei sistemi.
Per fare un paragone, il fatto che tutti gli uomini usino le onde sonore come strumento per trasmettere, abbiano le orecchie per decodificare le onde sonore e abbiano la stessa struttura celebrale per interpretare i segnali giunti dall'orecchio non garantisce che due persone possano riuscire a parlare fra loro.
Immaginate di voler avere informazioni su un prodotto venduto in Giappone ma non ancora in Italia (e, supponiamo che ovviamente, ci siano negozi e/o fornitori pronti ad esaudire le vostre richieste). Il problema può esser banalmente decomposto nei seguenti tre passi: primo, cercare quali posti possano avere l'informazione da voi cercata; secondo, una volta scelta una fonte, verificare se in effetti quella fonte è attendibile e contiene la risposta cercata; terzo, avviare la comunicazione, effettuare la richiesta ed ottenere l'informazione.
Ma la cosa più importante … trovare un linguaggio di comunicazione comprensibile, insomma capire e farsi capire! Un italiano ed un giapponese non riusciranno mai a comunicare, a meno che uno dei due non conosca la lingua dell'altro. Oppure entrambi devono conoscere, riconoscere, ed essere

disposi ad usare nella comunicazione un codice comune, ad esempio una terza lingua come l'inglese.
L'idea alla base dei Web Services è proprio questa, ossia cercare di definire un codice comune per le applicazioni informatiche, rispondendo alle domande, che dal lato cliente sono:

1. *dove trovare ciò che si cerca?*
2. *come fare a certificare che un certo ente offre i servizi richiesti?*
3. *come inviare una richiesta comprensibile ed assicurarsi di saper interpretare la risposta?*
4. *con che lingua comunicare?*

e rispettivamente dal lato fornitore dell'informazione sono:

1. *come fare a farsi trovare dai clienti?*
2. *come far capire ai potenziali utenti le possibilità offerte dai servizi offerti?*
3. *come consentire ai clienti di effettuare, in modo congruo e sicuro, le richieste, inviando risposte a loro comprensibili?*
4. *con che lingua comunicare?*

Come vedremo, i Web Services riescono in questo compito, utilizzando strumenti standard e definendo dei protocolli di comunicazione perfettamente rispondenti allo scopo ed implementabili in qualunque linguaggio (Java, naturalmente, compreso).
L'obiettivo di questo capitolo è quello di fornire una panoramica sulle potenzialità del linguaggio Java nella implementazione dei Web Services, indicando gli strumenti e le tecnologie necessarie, ma rimanendo necessariamente ad un livello alto, dal momento che numerosi sono gli elementi che interagiscono. Ipotizziamo, inoltre, che chi arrivi a leggere questo capitolo del libro sia ormai giunto ad un livello di conoscenza del linguaggio sufficiente per poter iniziare ad approfondire l'argomento anche senza l'ausilio dettagliato di una guida di base.

22.2 Definizione di Web Service

Un Web Service è un servizio disponibile in un qualsiasi punto della rete ed erogato sui protocolli HTTP e/o SMTP secondo una architettura Service Oriented (SOA – Service Oriented Architecture).
L'architettura service oriented prevede l'interazione di tre enti distinti:

- un fornitore di servizi, il quale è proprietario e responsabile del software di erogazione del servizio
- un cliente dei servizi erogati dal fornitore

- un ente, detto broker, che offre capacità di catalogazione e ricerca dei servizi offerti dai fornitori.

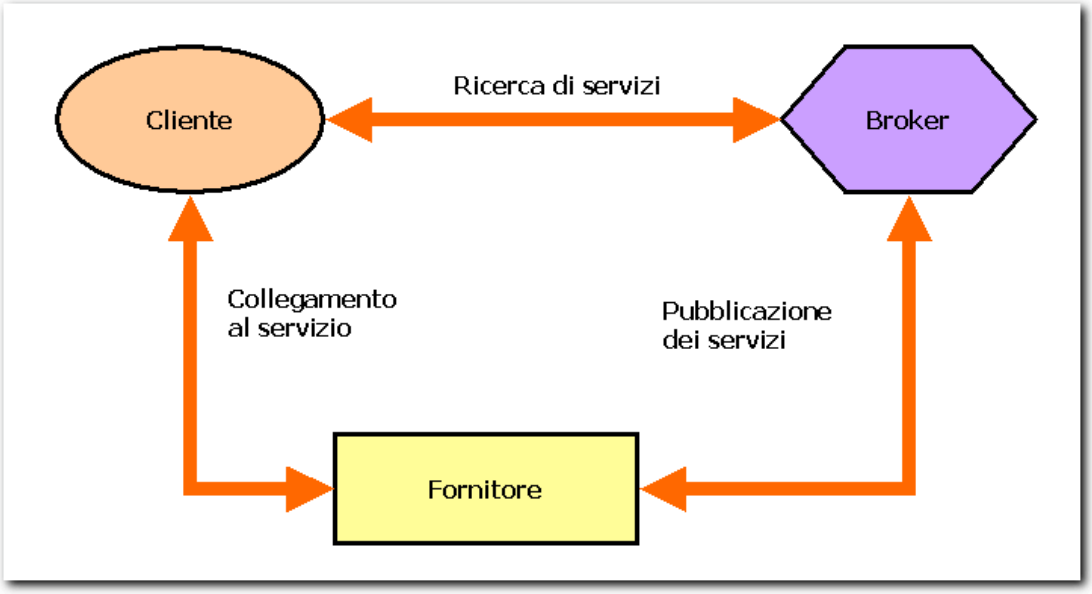


Figura 153: Architettura service oriented

Un Web Service è caratterizzato da due elementi:

- il servizio, inteso come implementazione della logica applicativa
- la descrizione del servizio, intesa come l'interfaccia del servizio stesso.

La descrizione del servizio è codificata in linguaggio XML e segue un insieme di standard che ne definiscono la formalizzazione, il riconoscimento e le regole di interoperabilità.
Come si evince dalla figura 1, le operazioni definite tra gli enti in gioco nell'architettura a servizi sono tre; la specifica dei Web Services individua meccanismi standard per codificare ciascuno di questi tre flussi.
La **pubblicazione dei servizi** coinvolge il broker ed il fornitore. In questo flusso il broker mette a disposizione i suoi servizi specifici e si dichiara pronto ad accettare un nuovo servizio, secondo le specifiche che il fornitore deve dettare. La specifica che sovrintende questa comunicazione è denominata UDDI (Universal Description, Discovery and Integration). La specifica UDDI consente al fornitore di definire le regole per la pubblicazione del servizio. Anche la **ricerca dei servizi** è gestita dalle regole descritte nella specifica

UDDI. In questo caso, UDDI ci consente di vedere il broker come una sorta di motore di ricerca cui chiedere una lista di fornitori che rispondono ai requisiti da noi cercati.
Per il **processo di collegamento** al servizio, i Web services standard ci forniscono due strumenti: il linguaggio WSDL (Web Services Description Language – Linguaggio per la descrizione dei Web Services) ed il protocollo applicativo SOAP (Simple Object Access Protocol – Semplice protocollo di accesso ad oggetti).
Il linguaggio WSDL fornisce una specifica formale di descrizione, in formato XML, di un Web Services in termini comprensibili al client, e che consentono a quest'ultimo di avere tutte le informazioni necessarie per collegarsi al fornitore, capire cosa fa il servizio, costruire le richieste (con descrizione dei tipi dei parametri), interpretare le risposte (con descrizione dei parametri ritornati e degli eventuali codici di errore).
Il protocollo SOAP è l'anima dell'architettura Web Services, in quanto rappresenta il vero protocollo contenente le regole per la comunicazione fra cliente e fornitore.

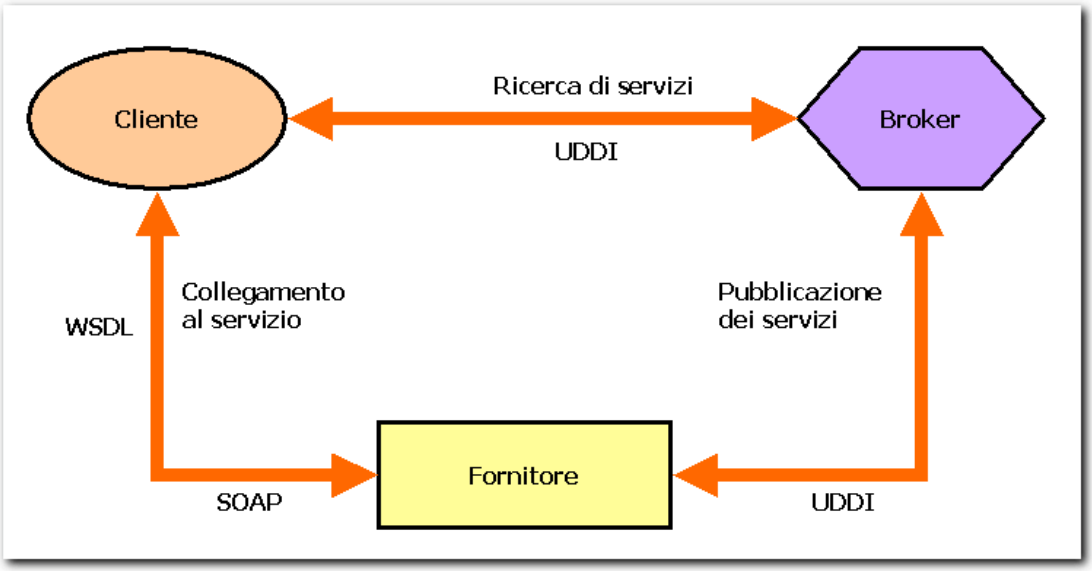


Figura 154: Protocolli di collegamento

22.3 Il protocollo SOAP

SOAP è il protocollo di riferimento per lo scambio di informazioni tra fornitore di servizi e cliente. E' basato sul linguaggio XML, di cui rappresenta una estensione funzionale alla descrizione dei messaggi, ed è caratterizzato dal fatto di appoggiarsi sul protocollo HTTP (o SMTP), dalla intrinseca capacità di far passare le richieste attraverso i firewall, dalla capacità di essere autodescrittivo e completo, in termini di capacità di trasporto, meccanismi di sicurezza e capacità di espressione di meccanismi differenti. E', infatti

possibile utilizzare SOAP per erogare semplici servizi Web, per effettuare chiamate di procedure remote (RPC), per scambiare documenti, per istaurare complessi colloqui tra sistemi interconnessi solo da una connessione TCP/IP.
Come dice la definizione stessa, SOAP è, inoltre, un protocollo basato sulla comunicazione tra oggetti, dal momento che è in grado di descrivere completamente e correttamente l'invocazione di un metodo su un oggetto remoto.
Il cuore della tecnologia SOAP risiede nella struttura dei messaggi.

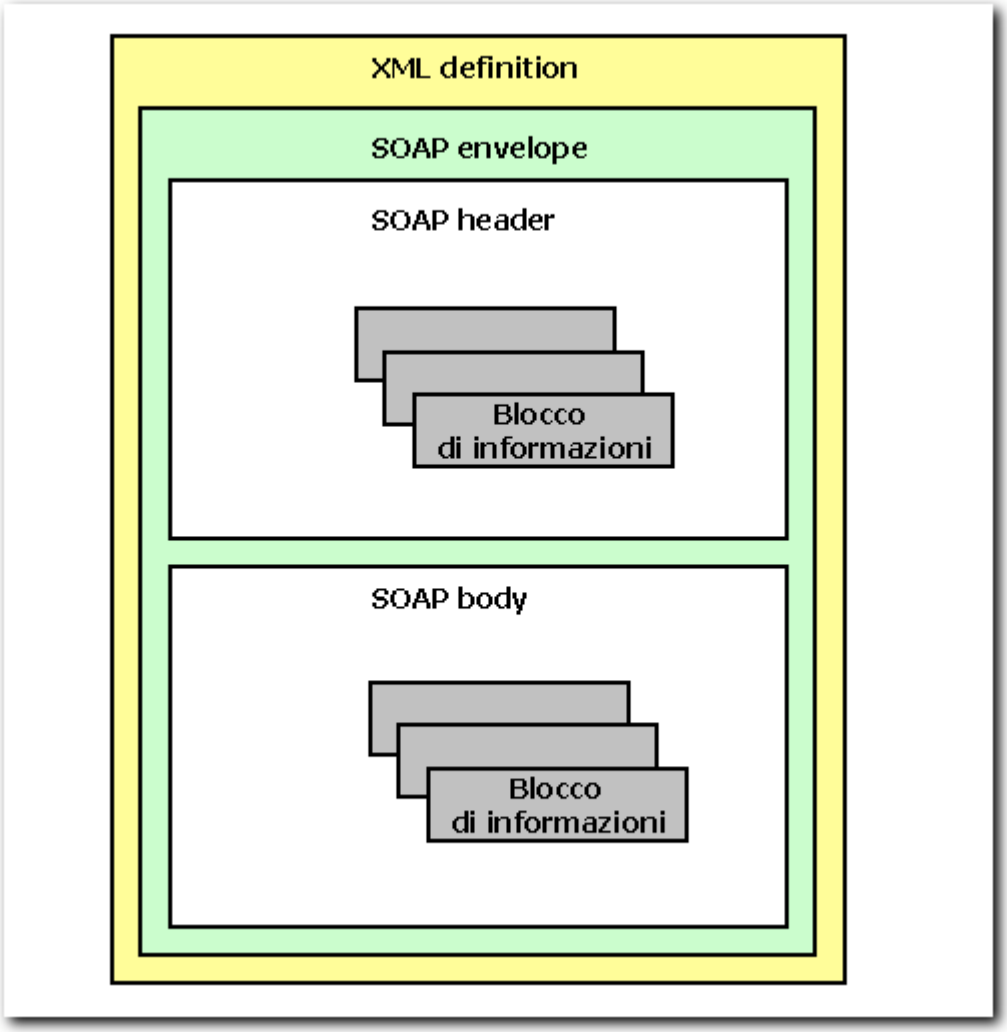


Figura 155: Struttura dei messaggi SOAP

Un messaggio SOAP è:

- un messaggio XML
- strutturato in un envelope, a sua volta composto da un body e da un header
- all'interno dell'header possono essere definite le direttive da inviare al client per il processamento del messaggio

- all'interno del body vengono scritte le informazioni relative alla comunicazione vera e propria, in blocchi XML logici

22.4 Gli strumenti Java

Java contiene numerosi packages di classi realizzate con lo scopo di agevolare la realizzazione di Web Services. Questi strumenti, introdotti successivamente al rilascio della versione 1.3 di Java ed integrati nella versione 1.4, rappresentano il completamento dell'architettura J2EE, che pertanto diventa completa anche dal punto di vista dello sviluppo di piattaforme service oriented. Tutto questo mantenendo inalterate le caratteristiche di scalabilità e portabilità della piattaforma ed aumentandone la flessibilità di disegno architetturale e di integrazione con sistemi eterogenei. Di seguito verrà svolta una breve panoramica di sintesi su questi strumenti.

22.5 JAXP

JAXP (Java Api for Xml Processing) rappresenta lo strumento per il trattamento (processing) di file, messaggi o comunque dati scritti in XML. JAXP consente di effettuare il parsing di un messaggio XML e fornirne una rappresentazione comprensibile in linguaggio Java; i modelli utilizzati sono due:

- SAX (Simple API for XML Parsing) è il modello che prevede il parsing di uno stream XML interpretando i dati mano a mano che arrivano;
- DOM (Document Object Model) è il modello che consente di creare una rappresentazione ad oggetti dell'intera struttura dell'XML.

La scelta del modello da utilizzare è delegata al programmatore ed è, dal punto di vista della capacità di interpretazione dei dati, del tutto equivalente. SAX è un modello più semplice da utilizzare, ed ha il vantaggio di consentire la creazione di una arbitraria rappresentazione ad oggetti dell'XML durante la sua lettura. In questo modo il programmatore può utilizzare le informazioni contenute nell'XML per popolare una rappresentazione ad oggetti da lui definita. Nel caso del DOM, invece, la rappresentazione ad oggetti è già data (anche se, essendo generale, potrebbe non essere conforme alle richieste), ma ha il vantaggio di essere già pronta anche ad eseguire operazioni complesse, quali la ricerca di un elemento o di istanze. Si noti, inoltre, come tale rappresentazione non sia specifica per Java, ma standard per ogni linguaggio.
Il package JAXP fornisce anche strumenti per la trasformazione o la creazione di XML integrati e trasformati con la tecnologia XSLT (XML Stylesheet Language for Trasformations).

La struttura dei package JAXP è la seguente:

Struttura dei package JAXP	
Package	**Descrizione**
javax.xml.parsers	E' il package che contiene le interfacce generiche per il trattamento dei modelli SAX e DOM.
org.w3c.dom	Package che contiene le classi per la definizione dello standard DOM
org.xml.sax	Package che contiene API standard SAX
javax.xml.transform	Contiene API utili per le trasformazioni XSLT

Le API per effettuare il parsing di documenti XML sono contenute nel package java.xml.parsers, e definiscono degli oggetti generici per l'utilizzo dei modelli DOM e SAX; queste classi definiscono interfacce standard che debbono essere implementate da coloro che realizzano parsers[13].

22.6 JAXR

JAXR (Java Api for XML Registries) sono le API java per l'integrazione e/o la creazione di servizi di registry per la ricerca e pubblicazione di Web Services.
Le API JAXR sono costituite da:

- un JAXR client, ossia un set di API che consentono ad un client di accedere ai servizi di registry di un provider remoto
- un JAXR provider, ossia una serie di interfacce per la creazione di un broker

Le API client di JAXR consentono:

- di effettuare il lookup su un broker di servizi, ottenendo una connessione;
- di effettuare ricerche sui servizi presenti nella struttura del provider, partendo dalla ricerca dell'organizzazione che fornisce il servizio (filtrando le ricerche per tassonomie predefinite quali nome, anche non completo, dell'organizzazione e per categoria merceologica dell'azienda o del servizio) e ottenendo, infine, la lista dei servizi offerti dall'azienda cercata;
- di pubblicare (qualora il provider lo consenta) servizi, creando e gestendo i dati relativi all'organizzazione e al tipo di classificazione (tipo

13 Per utilizzare JAXP è, pertanto, necessario avere una implementazione di tali parser; i più famosi sono quelli forniti dall'Apache Group, in particolare Xerces.

di servizio e tipo di azienda, ma anche di definire classificazioni arbitrarie)

22.7 JAXM

JAXM (Java API for XML Messaging) è un insieme di packages contenenti API per la creazione di messaggi scritti in XML.
La creazione e l'invio di messaggi XML secondo JAXM sono conformi al protocollo SOAP, e la struttura dei package prevede il package javax.xml.soap, che include le API per creare messaggi SOAP di richiesta e risposta, e il package javax.xml.messaging necessario per utilizzare un messaging provider (ossia un sistema che, in pieno stile MOM, provvede a recapitare messaggi a vari destinatari) e che pertanto fornisce un sistema unidirezionale di invio messaggi.

22.8 JAX-RPC

JAX-RPC fornisce una serie di API per effettuare chiamate ed eseguire procedure (RPC) su client remoti.
JAX-RPC rappresenta un protocollo basato sulla specifica SOAP: ancora basato su XML, definisce le strutture di envelope e body secondo SOAP e le regole per la definizione delle chiamate remote, per il passaggio dei parametri di input, per riconoscere i risultati e gli eventuali messaggi d'errore. Il vantaggio di questo sistema consiste nella estrema semplicità di sviluppo, sia dal lato client che da quello server, e nella potenza del paradigma RPC, mentre lo svantaggio risiede nel dover ricreare una struttura di architettura distribuita (compreso il concetto di stub), e che pertanto richiede la presenza di un sistema che funziona da middleware (detto JAXP-RPC runtime).

Parte Terza

Design Pattern

23 DESIGN PATTERN

23.1 Introduzione

Pattern Mining significa raccogliere l'esperienza degli analisti e dei programmatori per creare librerie di oggetti riutilizzabili. L'idea alla base di questa disciplina è semplice: collezioniamo e cataloghiamo gli oggetti e le loro interazioni, se rappresentano soluzioni a problemi di interesse comune.
Gli oggetti insieme alle loro interazioni sono detti Design Pattern.
La storia dei Design Pattern inizia all'inizio del 1980 quando Smaltalk era il linguaggio ad oggetti più utilizzato. Come abbiamo visto all'inizio del libro, in quel periodo il paradigma ad oggetti non era ancora diffuso ma, la crescente necessità di dotare sistemi di interfacce utente, spingeva sempre più in questa direzione. Fu nel 1988 che due analisti, *Krasner* e *Pope*, crearono il primo pattern per Smaltalk chiamato *Model-View-Controller* che, divideva il problema dell'interfaccia utente nelle tre parti descritte nella prossima figura.

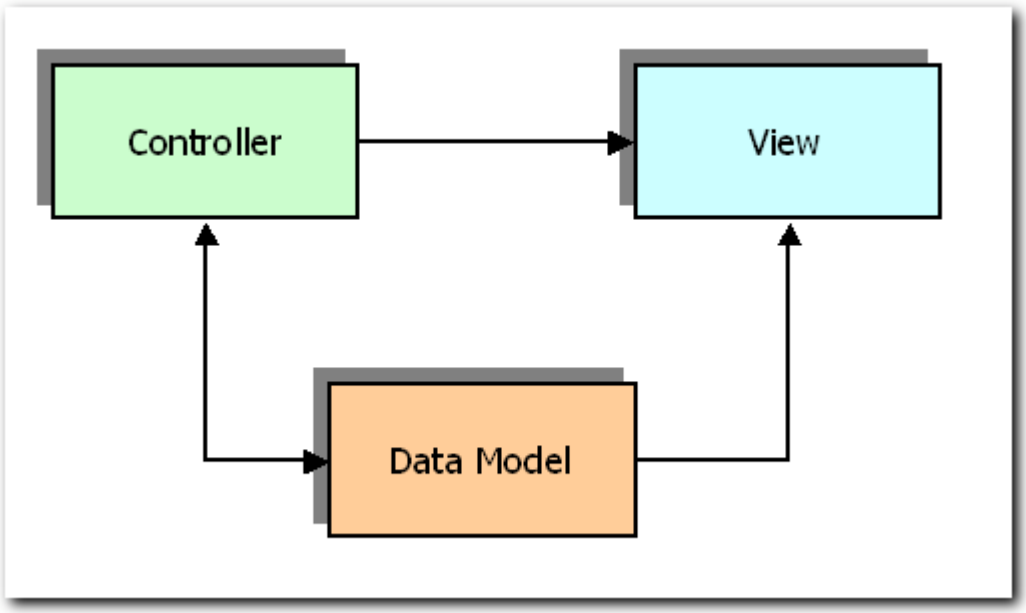


Figura 156: Model-View-Controller

1. *Data Model: contiene le logiche di business di un programma;*
2. *View: rappresenta l'interfaccia utente;*
3. *Controller: rappresenta lo strato di interazione tra interfaccia e logiche di business.*

Ogni strato del modello è un oggetto a se stante con le sue regole per l'elaborazione dei dati, ogni sistema ha caratteristiche proprietarie che definiscono tali regole, ma, il modello è comune a tutti i sistemi che necessitano di interfacce utente.

23.2 Cosa è un design pattern

In molto hanno provato a definire con esattezza cosa sia un pattern e la letteratura contiene numerose definizioni:

- *"Design patterns constitute a set of rules describing how to accomplishcertain tasks in the realm of software development."*

  (I Design Pattern sono collezioni di regole che descrivono come eseguire alcuni compiti di programmazione)

  *(**Pree, 1994**)*

- *"Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation."*

  (I framework focalizzano l'attenzione sul disegno e l'implementazione di una architettura, I Design Pattern sul riutilizzo di disegni architetturali ricorrenti.)

  *(**Coplien & Schmidt, 1995**).*

- *"A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it"*

  (Un pattern riconduce un problema di disegno ricorrente ad una specifica situazione e ne fornisce la soluzione.)

  *(**Buschmann, et. al. 1996**)*

- *"Patterns identify and specify abstractions that are above the level of single classes and instances, or of components."*

  (I pattern identificano e definiscono le astrazioni di classi, oggetti e componenti.)

  *(**Gamma, et al., 1993**)*

Tutte le definizioni concordano con l'identificare i pattern con soluzioni comuni ad un problema di disegno ricorrente.

Questa definizione richiede però di essere approfondita; in particolare ci dobbiamo chiedere cosa sia esattamente un problema.
In generale, un problema è un quesito da chiarire, qualcosa che deve essere identificato, capito e risolto. In particolare, un problema è un insieme di cause ed effetti.
Un problema a sua volta è dipendente da un contesto ovvero, dalle condizioni al contorno, che ne determina le cause.
Risolvere un problema significa quindi identificare le risposte inquadrandole all'interno del contesto di riferimento.
I pattern rappresentano quindi la soluzione migliore a problemi architetturali ricorrenti all'interno di determinati contesti applicativi e forniscono la migliore soluzione ai problemi che i programmatori incontrano tutti i giorni in fase di disegno di una applicazione:

1. *Identificazione dei concetti e la determinazione degli oggetti appropriati;*
2. *La specifica dell'interfaccia degli oggetti;*
3. *La specifica dell'implementazione degli oggetti;*
4. *La definizione delle interazioni tra gli oggetti;*
5. *Riutilizzabilità del codice, manutenibilità, scalabilità e prestazioni.*

23.3 La banda dei quattro

Basta effettuare una semplice ricerca in internet per accorgerci che esistono migliaia di pattern a tutti i possibili livelli di astrazione. Dieci anni dopo la nascita del primo pattern, alla fine del 1990, si iniziò formalmente a catalogare i Design Pattern e alla fine del 1995 fu pubblicato il libro "*Elements of reusable software*" scritto da quattro autori (*Gamma, Helm, Jhonson e Vlissides*) noti come "*Gang of Four*" (banda dei quattro). I ventitrè pattern raccolti all'interno del libro, oggi noti come *GoF* sono considerati i fondamentali e sono suddivisi nei tre gruppi schematizzati nella prossima tabella:

Pattern GoF		
Creational	**Structural**	**Behavioral**

Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

Nei prossimi capitoli introdurremo i ventitré pattern *GoF*. Per una trattazione dettagliata del problema è in ogni modo consigliabile fare riferimento alla letteratura come riccamente suggerito all'interno della bibliografia.

24 CREATIONAL PATTERN

24.1 Introduzione

Il linguaggio Java consente di creare oggetti semplicemente utilizzando l'operatore **new**. In molti casi la natura di un oggetto può variare e risulta comodo poter astrarre il processo di allocazione di una classe, utilizzando classi particolari dette "creatrici".
I pattern creational descrivono i modi migliori per creare oggetti all'interno di un'applicazione. L'importanza di questi pattern sta nel fatto che, un'applicazione deve essere indipendente dal modo in cui gli oggetti sono creati, costruiti e rappresentati.
I pattern creational sono cinque:

Creational Pattern	
Nome	**Descrizione**
Abstract Factory	Fornisce un'interfaccia per creare e tornare oggetti appartenenti ad una stessa famiglia, selezionata tra un gruppo di famiglie.
Factory Method	Fornisce una classe che, in base ad alcuni attributi di input, ritorna un oggetto scelto tra un insieme di oggetti derivati da una classe astratta.
Builder	Separa la costruzione di un oggetto complesso dalla sua rappresentazione. In questo modo è possibile creare differenti rappresentazioni dipendenti dalle caratteristiche dei dati da rappresentare.
Prototype	Crea un clone di un oggetto invece di allocare una nuova classe dello stesso tipo.
Singleton	Fornisce un modello per le classi delle quali deve esserci sempre una ed un solo oggetto in memoria. Fornisce inoltre un singolo punto di accesso all'oggetto allocato.

24.2 Factory Method

Il Factory Method, è il pattern che in base ad alcuni attributi di input, ritorna un oggetto scelto tra un insieme di oggetti.
Tipicamente, gli oggetti sono selezionati tra un insieme di classi derivate da una classe astratta e, di conseguenza, aventi un metodo in comune.

Il problema:

- *Una classe non può decidere in anticipo quale tipo di oggetto creare;*

- *Una classe deve utilizzare le sue sottoclassi per determinare che tipo di oggetto creare;*
- *Abbiamo necessità di incapsulare le logiche necessarie alla creazione di nuovi oggetti.*

Il disegno:

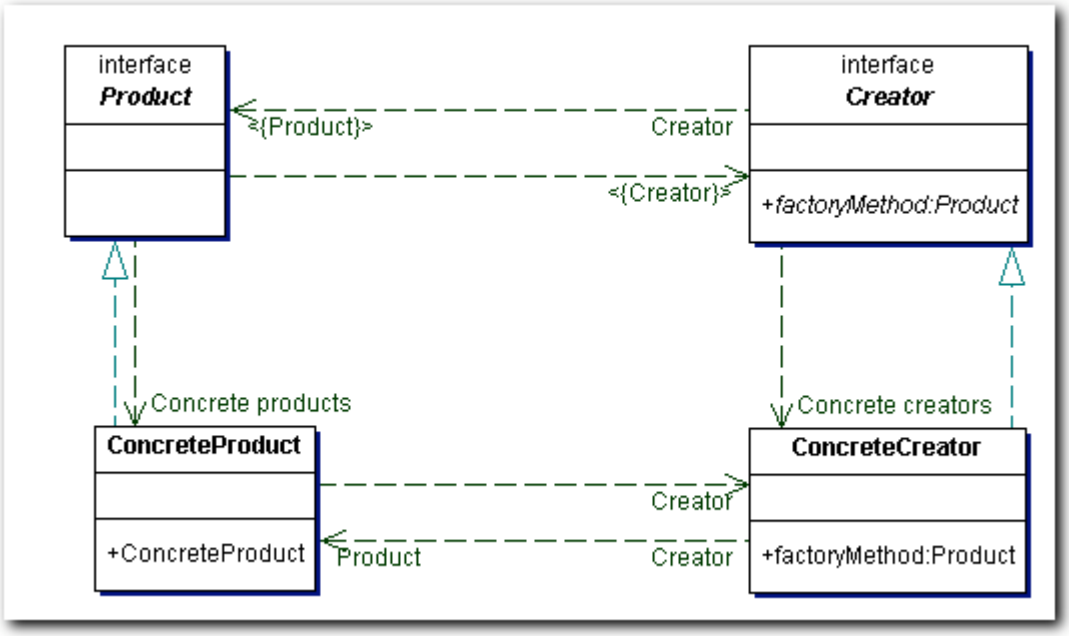


Figura 157: Factory Method

Come lavora:

Mediante l'interfaccia *Creator*, eseguendo il metodo *factoryMethod* deleghiamo alla classe *ConcreateCreator* la creazione di un oggetto di tipo *Product* come schematizzato nel prossimo sequence diagram.

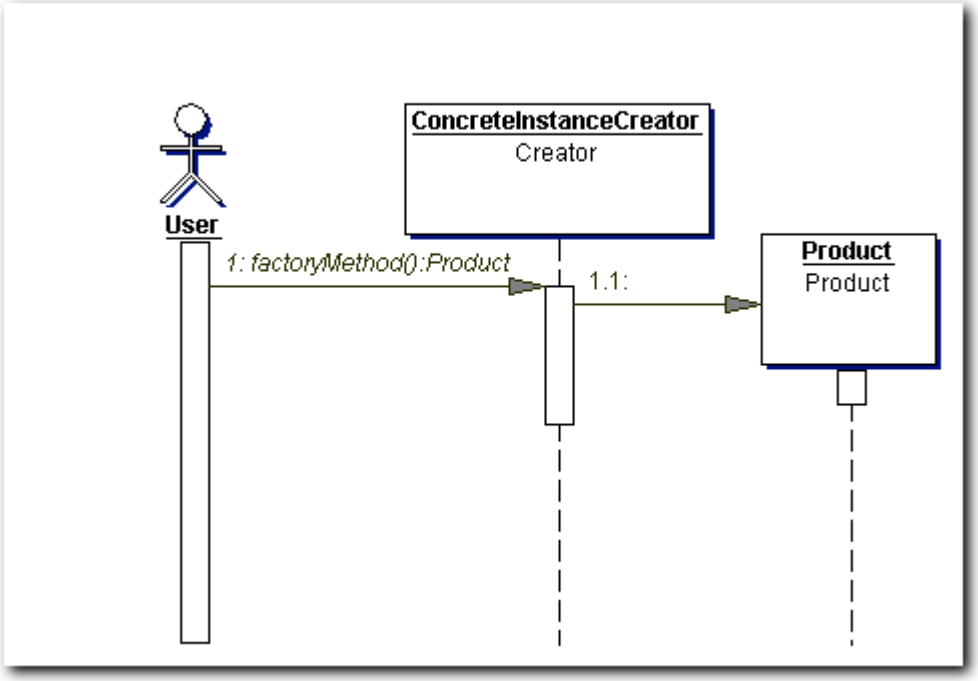


Figura 158: Factory Method Sequence Diagram

Poiché la classe *Product* è una classe astratta, è possibile crearne tante specializzazioni secondo le necessità imposte dal contesto applicativo.
Il metodo *factoryMethod* di *ConcreteCreator* potrà essere specializzato affinché, secondo gli attributi, possa decidere quale classe di tipo *Product* allocare e ritornare all'utente.
Mediante questo pattern, possiamo facilmente ottenere istanze diverse di una classe appartenente alla famiglia definita da *Product*, incapsulando all'interno della classe *ConcreateCreator* le logiche necessarie a creare l'oggetto adeguato a soddisfare le richieste dell'utente.

Il codice:

Nel prossimo esempio, l'interfaccia *TokenCreator* fornisce il metodo *getTokenizer(String)* che rappresenta il nostro metodo factory. Il metodo accetta una stringa come argomento e a seconda che la stringa contenga un punto od una virgola all'interno, ritorna una istanza rispettivamente di *DotTokenizer* o *CommaTokenizer* di tipo *Tokenizer*.
Le classi *DotTokenizer* e *CommaTokenizer* costruiscono un oggetto di tipo *java.util.StringTokenizer* necessario a suddividere la stringa in token separati rispettivamente da un punto o da una virgola. Mediante il metodo *getTokens()*, entrambe restituiscono una enumerazione che rappresenta l'elenco dei token della stringa.

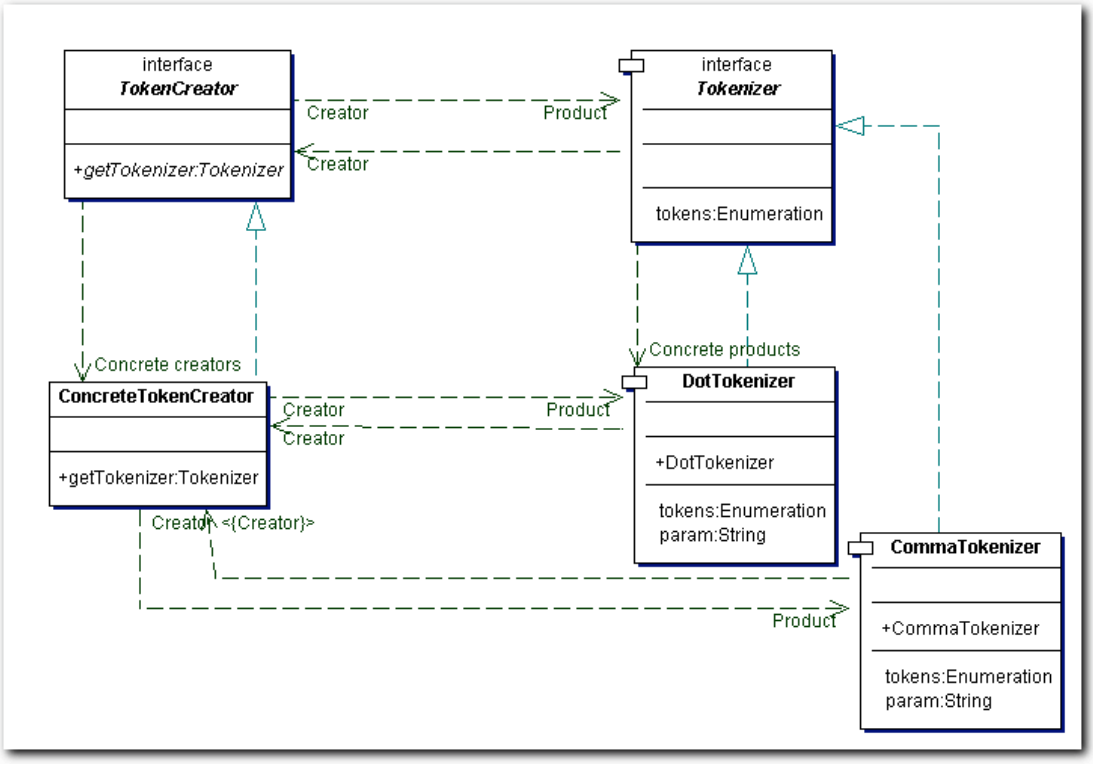


Figura 159: ***Esempio di factory method***

```
package src.pattern.factory;
public interface TokenCreator {
	Tokenizer getTokenizer(String tokenlist);
}
```

```
package src.pattern.factory;
public class ConcreteTokenCreator implements TokenCreator {
	public Tokenizer getTokenizer(String tokenlist){
		if(tokenlist.indexOf(",")!=-1)
			return new CommaTokenizer(tokenlist);

		if(tokenlist.indexOf(".")!=-1)
			return new DotTokenizer(tokenlist);
		return null;
	}
}
```

```
package src.pattern.factory;
import java.util.Enumeration;
public interface Tokenizer {
	Enumeration getTokens();

}
```

```
package src.pattern.factory;
import java.util.Enumeration;
import java.util.StringTokenizer;
public class DotTokenizer implements Tokenizer {

	public DotTokenizer(String param) {
		setParam(param);
	}

	public Enumeration getTokens(){
		StringTokenizer strtok = new StringTokenizer(getParam(),".");
		return strtok;
	}

	public String getParam(){ return param; }

	public void setParam(String param){ this.param = param; }
	private String param;

}
```

```
package src.pattern.factory;
import java.util.Enumeration;
import java.util.StringTokenizer;
public class CommaTokenizer implements Tokenizer {

	public CommaTokenizer (String param) {
		setParam(param);
	}

	public Enumeration getTokens(){
		StringTokenizer strtok = new StringTokenizer(getParam(),",");
		return strtok;
	}

	public String getParam(){ return param; }

	public void setParam(String param){ this.param = param; }
	private String param;

}
```

Infine, il metodo *main* della applicazione:

```
package src.pattern.factory;
import java.util.Enumeration;

/**
 * stampa a video la lista di token di due stringhe.
 * I token sono separati rispettivamente
 * da un punto ed una virgola.
 */
public class TokenPrinter {
	public static void main(String[] argv) {
		TokenCreator token = new ConcreteTokenCreator();
		String dotSeparatedToken = "a,b,c,d,e,f,g";
		String commaSeparatedToken = "h.i.l.m.n.o.p";
```

```
                Enumeration dotTokenList =token.getTokenizer(dotSeparatedToken).getTokens();
                Enumeration commaTokenList =
                        token.getTokenizer(commaSeparatedToken).getTokens();

                //stampo i teken della prima stringa stringa
                while(dotTokenList.hasMoreElements())
                        System.out.print(dotTokenList.nextElement()+" ");

                System.out.println();

                //stampo i teken della seconda stringa stringa
                while(commaTokenList.hasMoreElements())
                        System.out.print(commaTokenList.nextElement()+" ");

                System.out.println();
        }
}
```

24.3 Abstract Factory

Abstract Factory Pattern rappresenta un'astrazione del pattern precedente e fornisce un'interfaccia con un metodo factory che, a sua volta, ritorna una famiglia di classi logicamente correlate tra loro. Un esempio tipico di questo pattern è rappresentato in Java dal meccanismo di selezione delle sembianze di un'interfaccia utente. In particolare, le librerie swing di Java consentono di selezionare l'aspetto delle componenti di interfaccia fornendo diverse possibilità: Motif, Machintosh o Windows9x. Per realizzare questo meccanismo, viene utilizzata una classe factory cui segnaliamo, tramite un apposito metodo, quale sarà l'aspetto desiderato per l'interfaccia utente. La classe factory a sua volta ci ritornerà una famiglia di oggetti, appartenenti all'insieme delle componenti, aventi in comune l'aspetto grafico.

Il problema:

- *Una classe non può decidere in anticipo quale tipo di oggetto creare;*
- *Gli oggetti creati a loro volta appartengono ad un insieme di oggetti logicamente correlati tra loro.*
- *Abbiamo necessità di incapsulare le logiche necessarie alla creazione di nuovi oggetti.*

Il disegno:

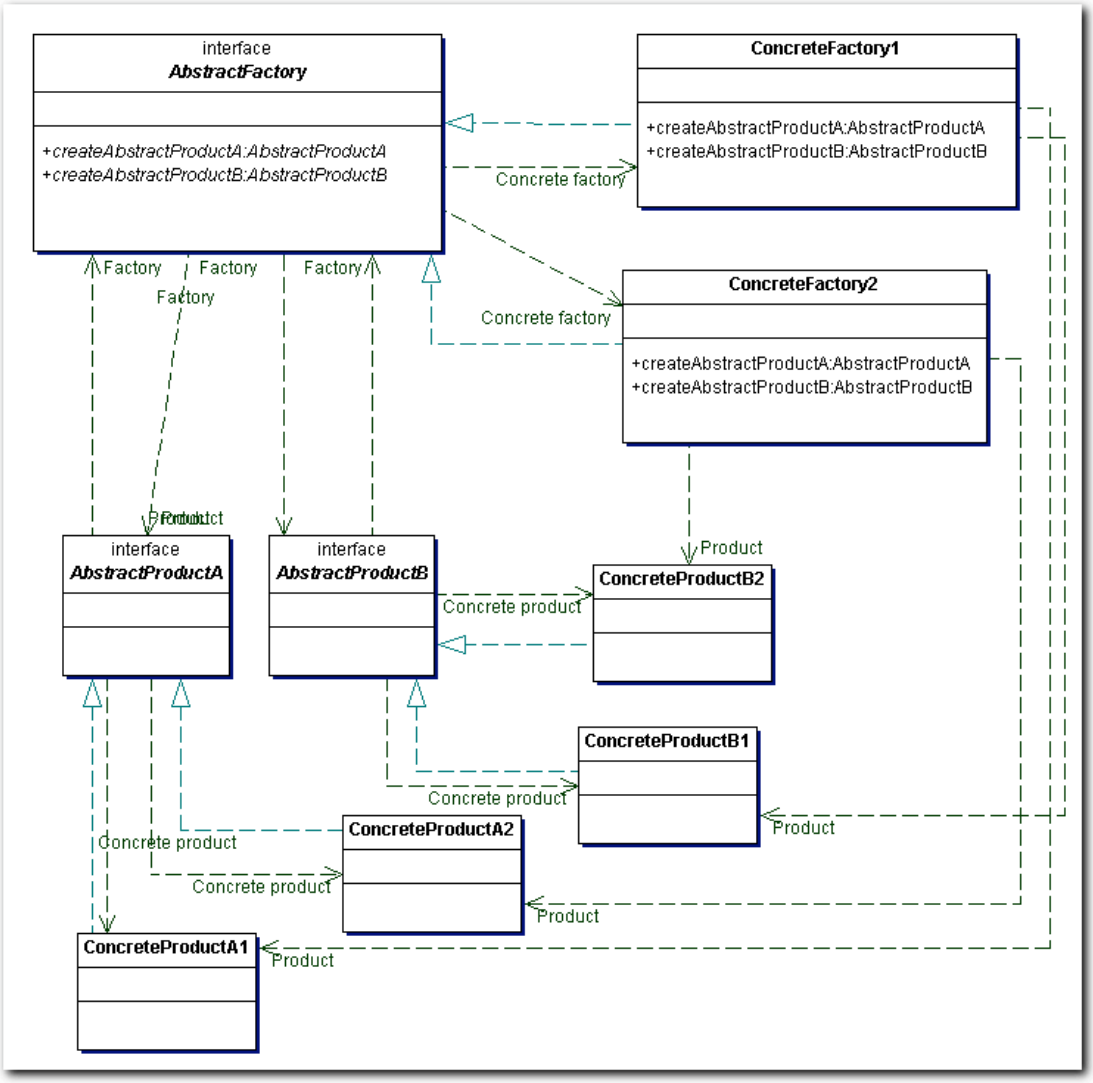


Figura 160: Abstract Factory

Come lavora:

Questo pattern fornisce l'interfaccia *AbstractFactory* contenente i metodi per la creazione delle famiglie di oggetti da restituire. L'implementazione di questi metodi è fornita dalle sottoclassi *ConcreateFactory*. Le interfacce *AbstractProduct*, rappresentano le famiglie di oggetti all'interno delle quali selezionare gli oggetti da ritornare rappresentati dalle classi *ConcreteProduct*.

Il codice:

Nel nostro esempio abbiamo due tipi di prodotti (*Figura precedente*):

AbstractProductA: ConcretePruductA1, ConcretePruductA2
AbstractProductB: ConcretePruductB1, ConcretePruductB2

che implementano le relative interfacce *AbstractProduct*. Gli oggetti *ConcreteFactory* sono a loro volta due e ritornano, rispettivamente, oggetti di tipo 1 (A1, B1) ed oggetti di tipo 2 (A2, B2) dalle famiglie di oggetti A e B

```
package src.pattern.abstractfactory.example;
public interface AbstractFactory {
	AbstractProductA createAbstractProductA();
	AbstractProductB createAbstractProductB();
}
```

```
package src.pattern.abstractfactory;
public class ConcreteFactory1 implements AbstractFactory {
	public AbstractProductA createAbstractProductA(){
		return new ConcreteProductA1();
	}

	public AbstractProductB createAbstractProductB(){
		return new ConcreteProductB1();
	}
}
```

```
package src.pattern.abstractfactory;
public class ConcreteFactory2 implements AbstractFactory {
	public AbstractProductA createAbstractProductA(){
		return new ConcreteProductA2();
	}

	public AbstractProductB createAbstractProductB(){
		return new ConcreteProductB2();
	}
}
```

```
package src.pattern.abstractfactory;
public interface AbstractProductA {
}
```

```
package src.pattern.abstractfactory;
public class ConcreteProductA1 implements AbstractProductA {
	public ConcreteProductA1() {
		System.out.println("ConcreteProductA1");
	}
}
```

```
package src.pattern.abstractfactory;
public class ConcreteProductA2 implements AbstractProductA {
	public ConcreteProductA2() {
		System.out.println("ConcreteProductA2");
	}
}
```

```
package src.pattern.abstractfactory;
public interface AbstractProductB {
}
```

```
package src.pattern.abstractfactory;
public class ConcreteProductB1 implements AbstractProductB {
	public ConcreteProductB1() {
		System.out.println("ConcreteProductB1");
	}
}
```

```
package src.pattern.abstractfactory;
public class ConcreteProductB1 implements AbstractProductB {
	public ConcreteProductB2() {
		System.out.println("ConcreteProductB2");
	}
}
```

Per comodità, definiamo la classe *FactoryHandler* contenente un metodo statico che, a seconda del parametro di input ritorna gli oggetti per creare famiglie di nuovi oggetti di tipo uno o due.

```
package src.pattern.abstractfactory;
public class FactoryHandler {

	private static AbstractFactory af = null;
	static AbstractFactory getFactory(int param)
	{
		if(param==1)
			af = new ConcreteFactory1();
		if(param==2)
			af = new ConcreteFactory2();
		return af;
	}
}
```

Infine il metodo main della applicazione:

```
package src.pattern.abstractfactory;

public class TestClient {
	public static void main(String[] argv) {
		AbstractFactory family1 = FactoryHandler.getFactory(1);
		AbstractProductA a1 = family1.createAbstractProductA();
		AbstractProductB b1 = family1.createAbstractProductB();

		AbstractFactory family2 = FactoryHandler.getFactory(2);
		AbstractProductA a2 = family2.createAbstractProductA();
		AbstractProductB b2 = family2.createAbstractProductB();
	}
}
```

Nella prossima figura è riportato il diagramme delle classi utilizzate.

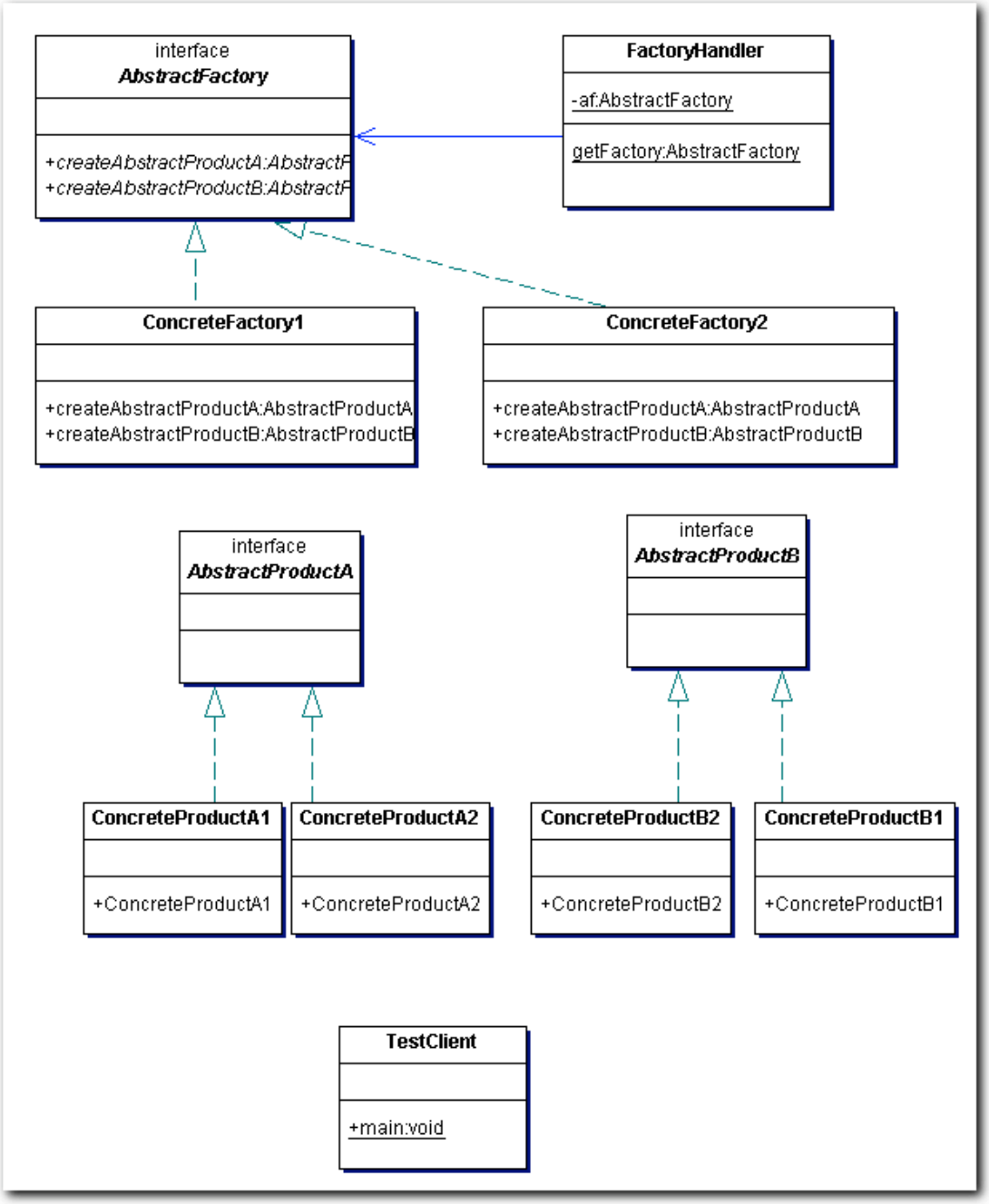


Figura 161: Class-Diagram della applicazione di esempio

24.4 Singleton

Una classe singleton è utile in tutti i contesti applicativi in cui è necessario garantire che esista uno ed un solo oggetto di un determinato tipo in memoria. L'unico modo per risolvere il problema, è quello di rendere la classe responsabile della allocazione in memoria di se stessa, impedendo ad altri oggetti di accedere al costruttore di classe.

Il problema:

- *Deve esistere esattamente una istanza di una classe, e deve essere facilmente accessibile dal client;*
- *L'unica istanza deve poter essere estesa mediante il meccanismo di ereditarietà e il client, deve poter utilizzare l'unica istanza della sottoclasse senza doverne modificare il codice.*

Il disegno:

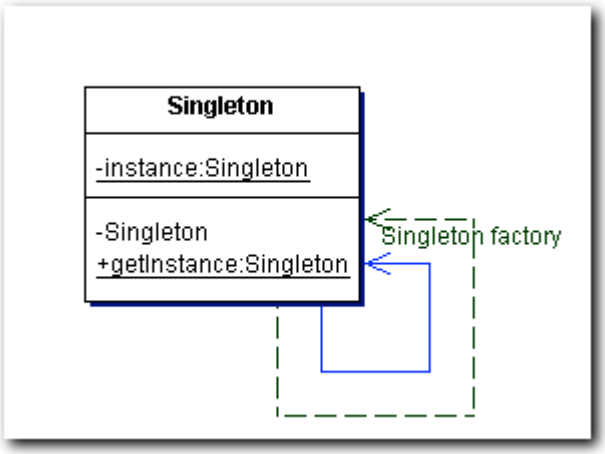


Figura 162: Singleton

Come lavora:

Per comprendere il meccanismo di funzionamento di un Singleton, è necessario fare un passo indietro ricordando alcuni fondamentali del linguaggio Java.

1. *Un oggetto può essere creato a partire dalla sua definizione di classe mediante l'operatore **new**.*
2. *L'operatore **new** utilizza un metodo costruttore della classe. Nel caso in cui non venga specificato nessun costruttore, Java ne garantisce l'esistenza aggiungendone automaticamente uno al momento della compilazione.*
3. *I metodi statici sono gli unici metodi che possono essere eseguiti senza che la classe sia stata allocata in memoria.*
4. *I metodi privati di una classe sono visibili soltanto ai membri della classe stessa.*

Analizziamo ora il seguente codice Java:

```
package src.pattern.singleton;
```

```
public class Singleton {
        private Singleton(){}
        public static Singleton getInstance(){
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }
        private static Singleton instance = null;
}
```

Notiamo subito che la classe contiene la definizione di un unico costruttore privato. Rendere privato il costruttore significa:

1. *Nasconderlo all'operatore **new**;*
2. *Impedire che il compilatore crei automaticamente il costruttore nullo.*

L'effetto di questa dichiarazione è di rendere inutilizzabile l'operatore **new** ovvero, non potremmo utilizzare l'istruzione:

*Singleton s = **new** Singleton()*

per allocare l'oggetto in memoria, pena un errore in fase di compilazione. Non essendoci altri costruttori, la classe non può essere allocata da nessun oggetto esterno che ne debba fare uso.
Il metodo *getInstance()*, al momento della sua esecuzione, controlla se il dato membro statico *instance* di tipo *Singleton* sia uguale a **null**. Nel caso in cui l'esito dell'operazione sia positivo, il metodo crea un oggetto *Singleton*, e lo memorizza nella variabile reference *instance*. Questa operazione è ora possibile poiché il metodo *getInstance()*, in quanto membro della classe, può accedere al costruttore privato.
Infine, questo metodo garantisce che nessun'altro oggetto di tipo *Singleton* possa essere creato, finchè la variabile reference *instance* contiene un riferimento ad un oggetto attivo.
Per garantire ad altri oggetti l'accesso all'unica istanza della classe, il metodo *getInstance()* è dichiarato statico.

Il codice:

```
package src.pattern.singleton;
public class Singleton {
        private Singleton() {
                System.out.println("L'oggetto è stato creato in questo momento!");
        }

        public void Hello() {
                System.out.println("Ciao dall'oggetto singleton");
```

```
        }

        public static Singleton getInstance() {
                if (instance == null) {
                        System.out.println("Creo l'oggetto");
                        instance = new Singleton();
                }
                else{
                        System.out.println("Esiste già un oggetto di questo tipo in memoria");
                }
                return instance;
        }
        private static Singleton instance = null;
}
```

Il metodo main della applicazione, tenta di creare più oggetti di tipo *Singleton*. I messaggi prodotti della classe mostrano chiaramento come l'oggetto possa essere creato una sola volta. Infine, il codice mostra come accedere all'unica istanza dell'oggetto creato.

```
package src.pattern.singleton;

public class TestClient {
        public static void main(String[] argv) {
                Singleton obj1 = Singleton.getInstance();
                Singleton obj2 = Singleton.getInstance();
                Singleton obj3 = Singleton.getInstance();
                Singleton.getInstance().Hello();
        }
}
```

```
Creo l'oggetto
L'oggetto è stato creato in questo momento!
Esiste già un oggetto di questo tipo in memoria
Esiste già un oggetto di questo tipo in memoria
Esiste già un oggetto di questo tipo in memoria
Ciao dall'oggetto singleton
```

24.5 Builder

Un pattern Builder è molto simile al pattern Abstract Factory in quanto entrambi creano oggetti. A differenza del secondo che produce famiglie di oggetti logicamente correlati tra loro, Builder costruisce oggetti complessi dividendone la costruzione in passi elementari che, possono essere riutilizzati per creare oggetti complessi dipendenti solo dai dati che debbono essere rappresentati.

Il problema:

- *Dobbiamo creare diverse rappresentazioni di dati complessi;*
- *La rappresentazione deve dipendere solo dai dati.*

Il disegno:

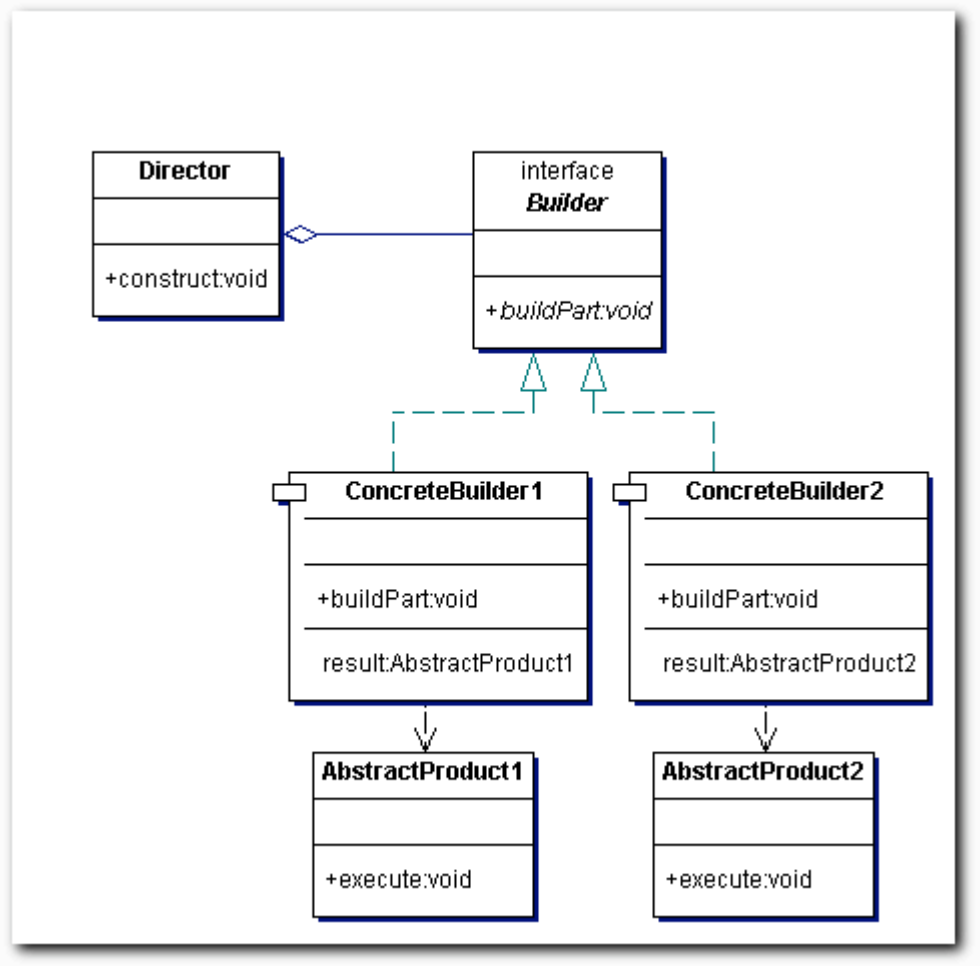


Figura 163: Builder Pattern

Come lavora:

Come mostrato nel diagramma riportato nella figura precedente, gli oggetti che collaborano alla realizzazione del pattern sono: un oggetto Director, alcuni oggetti Builder ed altrettanti oggetti Product. Tutti gli oggetti *Builder*, implementano un'interfaccia contenente i prototipi dei metodi necessari all'oggetto *Director* per costruire un determinato oggetto *Product*. Il processo di costruzione avviene per parti tramite operazioni semplici. Il client passa l'oggetto di tipo *Builde adatto* a *Director* che lo utilizzerà per creare l'oggetto desiderato.

Il codice:

Nel prossimo esempio, cercheremo di mostrare come utilizzare il pattern Builder per creare un documento in formato testo ASCII, da un uno in formato RTF. Le operazioni da compiere in questo caso sono due:

1. *Analizzare il documento RTF (Costruzione);*
2. *Creare il documento di testo (Rappresentazione).*

Il processo di analisi del documento può essere effettuato utilizzando metodi indipendenti dal formato dei dati da comporre. Ad esempio, potremmo utilizzare due metodi che convertono i caratteri e i paragrafi contenuti nel documento rtf, per creare rappresentazioni di documenti di tipo testo o di tipo pdf.

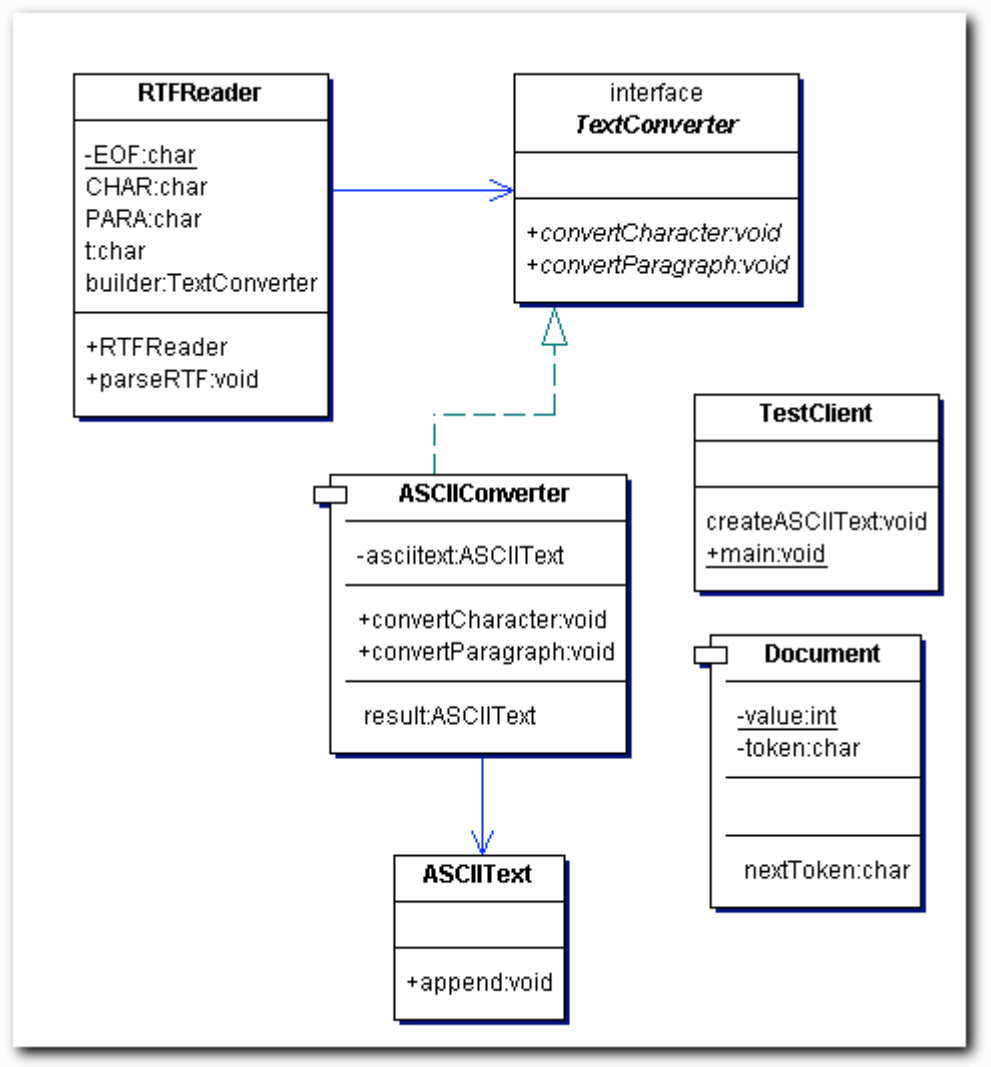


Figura 164: Esempio di pattern Builder

Le classi protagoniste della applicazione di esempio sono schematizzate nella *figura 164*:

- La classe *RTFReader* che rappresenta il *Director*;
- L'interfaccia *TextConverter* che rappresenta l'oggetto stratto *Builder*;
- La classe *ASCIIConverter* di tipo *TextConverter* che contiene l'implementazione del codice necessario ad analizzare il documento rtf;
- La classe *ASCIIText* che rappresenta il prodotto (rappresentazione).

Le classi *TestClient* e *Document* sono rispettivamente il client e la rappresentazione di un documento generico.
Data la complessità del problema che andiamo a rappresentare, il codice si limiterà allo scheletro della applicazione.

```
package src.pattern.builder;
/**
 * CLASSE DIRECTOR
 */
public class RTFReader {
        public RTFReader(TextConverter obj) {
                builder = obj;
        }

        public void parseRTF(Document doc) {
                while ((t = doc.getNextToken()) != EOF) {
                        switch(t){
                                case CHAR: builder.convertCharacter(t);
                                case PARA: builder.convertParagraph();
                        }
                }
        }
        private static final char EOF = '0';
        final char CHAR = 'c';
        final char PARA = 'p';
        char t;
        TextConverter builder;
}
```

```
package src.pattern.builder;
/**
 * CLASSE BUILDER
 */
public interface TextConverter {
        void convertCharacter(char param);
        void convertParagraph();
}
```

```
package src.pattern.builder;
/**
 * IMPLEMENTAZIONE DELLA CLASSE BUILDER
 */
public class ASCIIConverter implements TextConverter {
        private ASCIIText asciitext = new ASCIIText();
```

```
	public void convertCharacter(char param) {
		char asciichar = new Character(param).charValue();
		asciitext.append(asciichar);
	}

	public void convertParagraph() { }

	public ASCIIText getResult() {
		return asciitext;
	}
}
```

```
package src.pattern.builder;
/**
 * PRODUCT
 */
public class ASCIIText {
	public void append(char c) {
	}
}
```

```
package src.pattern.builder;
public class Document {
	private static int value;
	private char token;
	public char getNextToken(){
		return token;
	}
}
```

```
package src.pattern.builder;
/**
 * CLIENT
 */
public class TestClient {
	void createASCIIText(Document doc){
		ASCIIConverter asciibuilder = new ASCIIConverter();
		RTFReader rtfreader = new RTFReader(asciibuilder);
		rtfreader.parseRTF(doc);
		ASCIIText asciitext = asciibuilder.getResult();
	}

	public static void main(String[] argv) {
		TestClient c = new TestClient();
		Document doc = new Document();
		c.createASCIIText(doc);
	}
}
```

24.6 Prototype

Questo pattern può essere utilizzato quando, creare una copia di un oggetto esistente è meno oneroso che crearne uno nuovo. Di fatto, Prototype crea un clone di un oggetto piuttosto che allocare una nuova classe dello stesso tipo.

Il problema:

- *Dobbiamo definire molte sottoclassi che differiscono solamente per il tipo di oggetto creato.*
- *Dobbiamo creare molte copie di uno stesso oggetto il cui processo di costruzione è molto complicato.*

Il disegno:

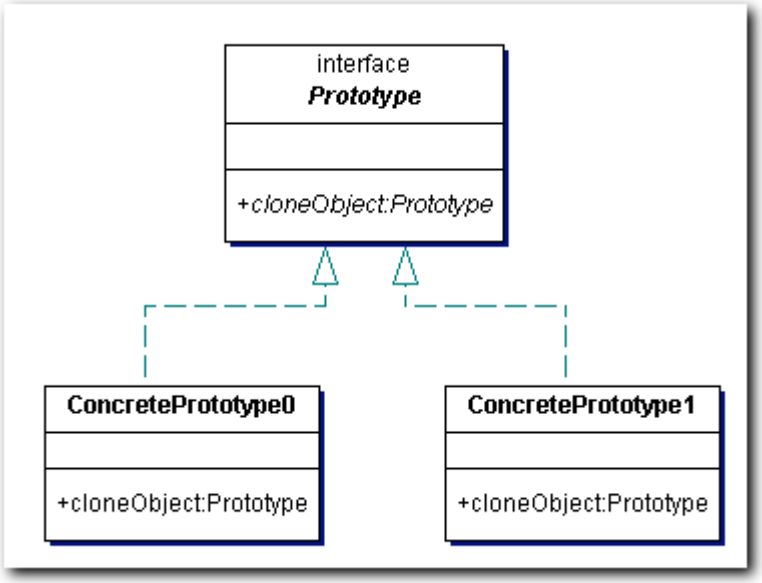


Figura 165: Pattern Prototype

Come lavora:

Molto spesso ci troviamo a dover creare copie di uno stesso oggetto, il cui processo di costruzione è molto complicato, basti pensare ad un oggetto il cui stato dipende da valori estratti tramite JDBC da una database le cui tabelle hanno una struttura macchinosa.
L'idea di questo pattern è quella di utilizzare un prototipo da clonare, per creare copie specializzate di uno stesso oggetto:

1. *Per ridurre drasticamente il numero delle sottoclassi da disegnare;*
2. *Per contenere le prestazioni di una applicazione.*

Per realizzare un prototipo, utilizziamo un'interfaccia che definisce le operazioni di base e il metodo per clonare l'oggetto. Implementando l'interfaccia prototipo, possiamo creare le rappresentazioni reali del prototipo definito.

Il codice:

Supponiamo di avere a disposizione l'intero archivio contenente tutti i nomi ed i numeri di telefono degli abbonati italiani e supponiamo di dover costruire un oggetto che rappresenta l'elenco degli abbonati che vivono a Roma.
Costruire l'elenco abbonati di Roma significa estrarre da una base dati contenente decine milioni di abbonati, solo quelli che vivono a Roma. E' facile immaginare quanto questa operazione sia costosa in termini di prestazioni.
Se dovessimo costruire un'applicazione che distribuisce l'oggetto ad ogni romano nell'elenco telefonico, le prestazioni del sistema sabbero seriamente compromesse.
Nel prossimo esempio mostriamo come utilizzare il pattern *Prototype* per risolvere il problema proposto. Di fatto, sarebbe estremamente più performante creare copie dell'oggetto esistente piuttosto che crearne uno nuovo per ogni abbonato. Le classi coinvolte sono schematizzate nella prossima figura.

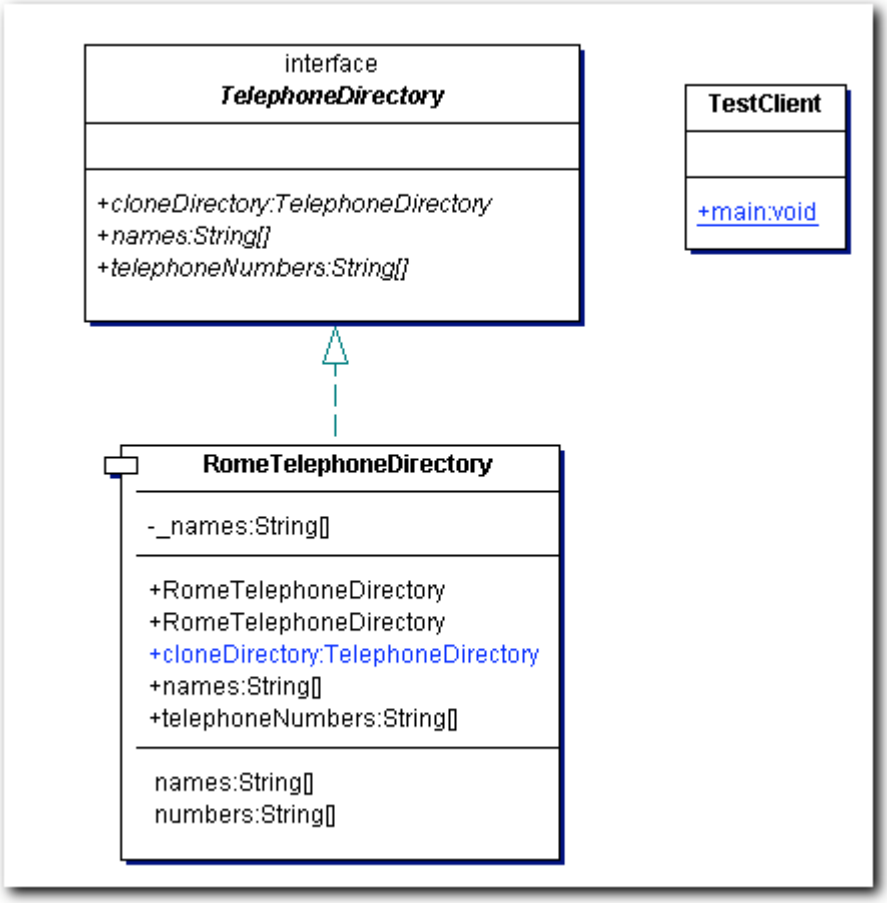


Figura 166: Esempio di prototipe pattern

```
package src.pattern.prototype;
```

```
/**
 * PROTOTYPE
 */
public interface TelephoneDirectory {
	TelephoneDirectory cloneDirectory();
	String[] names();
	String[] telephoneNumbers();
}
```

```
package src.pattern.prototype;
/**
* CONCRETE PROTOTYPE
*/
public class RomeTelephoneDirectory implements TelephoneDirectory {
	public RomeTelephoneDirectory() {
		//Utilizza JDBC per caricare l'elenco telefonico di
		//roma dall'archivio generale
		// degli abbonati italiani
	}

	public RomeTelephoneDirectory(String[] names, String[] numbers) {
		setNames(names);
		setNumbers(numbers);
	}

	public TelephoneDirectory cloneDirectory() {
		return new RomeTelephoneDirectory(names(), telephoneNumbers());
	}

	public String[] names() {
		return _names;
	}

	public String[] telephoneNumbers() {
		return numbers;
	}

	private void setNames(String[] names) {
		this._names = names;
	}

	private void setNumbers(String[] numbers) {
		this.numbers = numbers;
	}

	private String[] _names;
	private String[] numbers;
}
```

25 STRUCTURAL PATTERNS

25.1 Introduzione

I pattern strutturali descrivono come le classi possono combinarsi tra loro a formare strutture complesse.
Nella prossima tabella sono elencati brevemente i sette pattern appartenenti a questo gruppo.

Structural Pattern	
Nome	**Descrizione**
Adapter	Utilizzato per trasformare l'interfaccia di una classe in quella di un'altra classe.
Bridge	Disaccoppia astrazione da implementazione creando un legame tra loro.
Composite	Crea collezioni di oggetti ognuno dei quali può a sua volta essere una collezione di oggetti o un oggetto singolo.
Decorator	Modifica dinamicamente l'interfaccia di un oggetto modificandone soecializzandone le funzionalità senza realmente estendere l'oggetto in questione.
Façade	Raggruppa una gerarchia complessa di oggetti e fornisce un'interfaccia semplificata verso essi.
Flyweigth	Fornisce un modo per contenere la granularità di una applicazione limitando la proliferazione di piccoli oggetti aventi funzionalità simili.
Proxy	Controlla l'accesso ad altri oggetti virtuali o remoti.

25.2 Adapter

Questo pattern ci suggerisce un metodo per trasformare una classe in un'altra quando il contesto suggerisce di utilizzare classi già esistenti, ma con una diversa interfaccia verso il client. Adapter possiede due diverse varianti: i *class pattern* e gli *object pattern*. I primi, descrivono come utilizzare l'ereditarietà per costruire applicazioni con interfacce utente flessibili ed usabili. I secondi, descrivono come comporre oggetti tra loro includendo oggetti in altri oggetti fino a creare strutture arbitrariamente complesse.

Il problema:

- *Vogliamo utilizzare una classe esistente ma, la sua interfaccia non corrisponde con quella desiderata;*
- *Vogliamo definire una classe affinché possa essere utilizzata anche da classi che hanno interfacce incompatibili;*

- *Vogliamo utilizzare alcune sottoclassi, ma è impossibile utilizzare il meccanismo della ereditarietà per creare ulteriori sottoclassi (solo per la variante object). Un oggetto adapter deve quindi poter modificare l'interfaccia della superclasse.*

Il disegno:

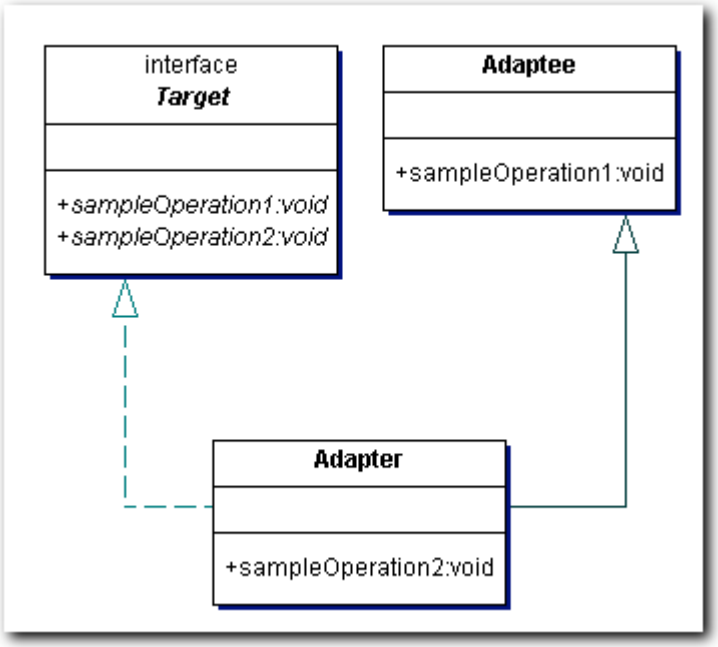


Figura 167: Variante class di Adapter

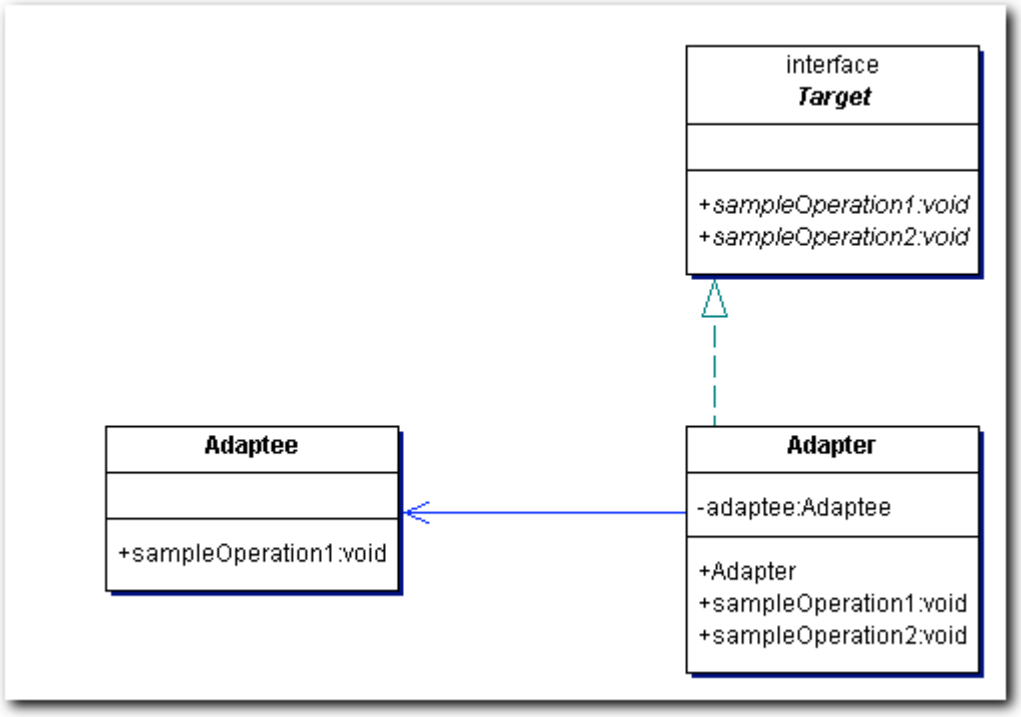


Figura 168: Variante Object di Adapter

Come lavora :

In entrambe le varianti di questo pattern, le classi partecipanti sono: l'interfaccia *Target* che definisce il prototipo dell'interfaccia specifica del client, la classe *Adaptee* che rappresenta l'oggetto la cui interfaccia deve essere modificata, ed infine la class *Adapter* che rappresenta la classe la cui interfaccia deve essere modificata.
La differenza tra le due varianti sta nel fatto che, la variante *class* realizza le classi *Adapter* estendendo ed implementando entrambi le altre classi partecipanti, la variante object *delega* le operazioni di interfaccia dall'oggetto *Adaptee* all'oggetto *Adapter*. Nel secondo caso, *Adaptee* diventa un attributo di *Adapter*.
Di fatto, la prima variante risolve il problema a livello di disegno delle classi, la seconda a livello di oggetti attivi all'interno di una applicazione favorendone la composizione di oggetti a discapito della ereditarietà.

Un esempio (variante class):

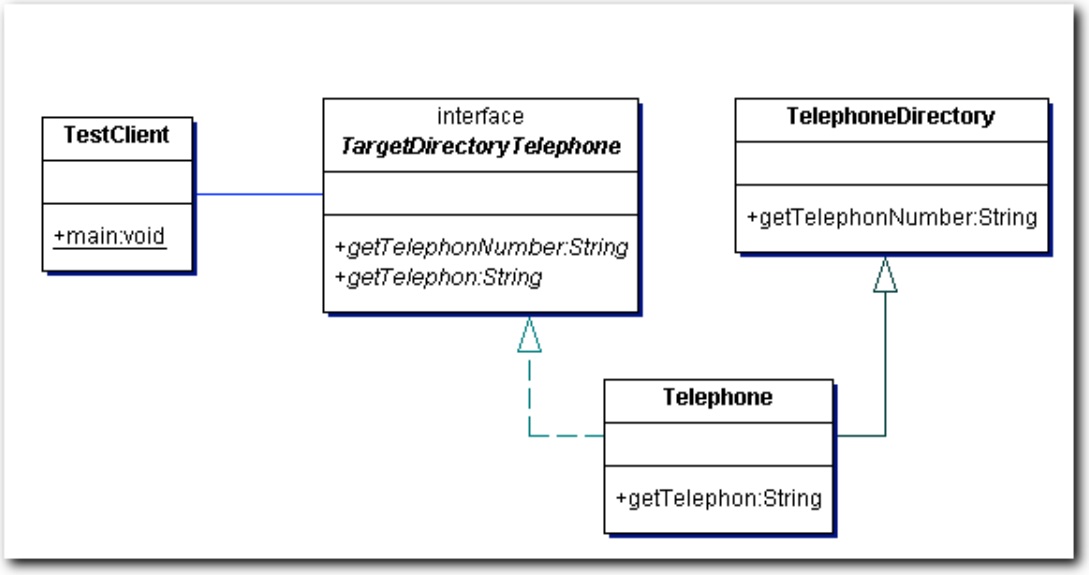


Figura 169: Esempio di variante class di adapter

Nell'esempio utilizzeremo la variante class del pattern per modificare l'interfaccia della classe *TelephoneDirectory* affinché si possa utilizzare il metodo *getTelephone(String)* in vece del metodo nativo *getTelephoneNumber(String)*.

```
/**
 * ADAPTEE
 */
public class TelephoneDirectory {
	public String getTelephonNumber(String nome){
		return "Il numero di telefono di "+nome+" e' 098765433";
	}
}
```

```
/**
 * TARGET
 */
public interface TargetDirectoryTelephone {
	String getTelephonNumber(String nome);
	String getTelephon(String nome);
}
```

```
/**
 * ADAPTER
 */
public class Telephone extends TelephoneDirectory implements TargetDirectoryTelephone {
	public String getTelephon(String nome){
		return getTelephonNumber(nome);
	}
}
```

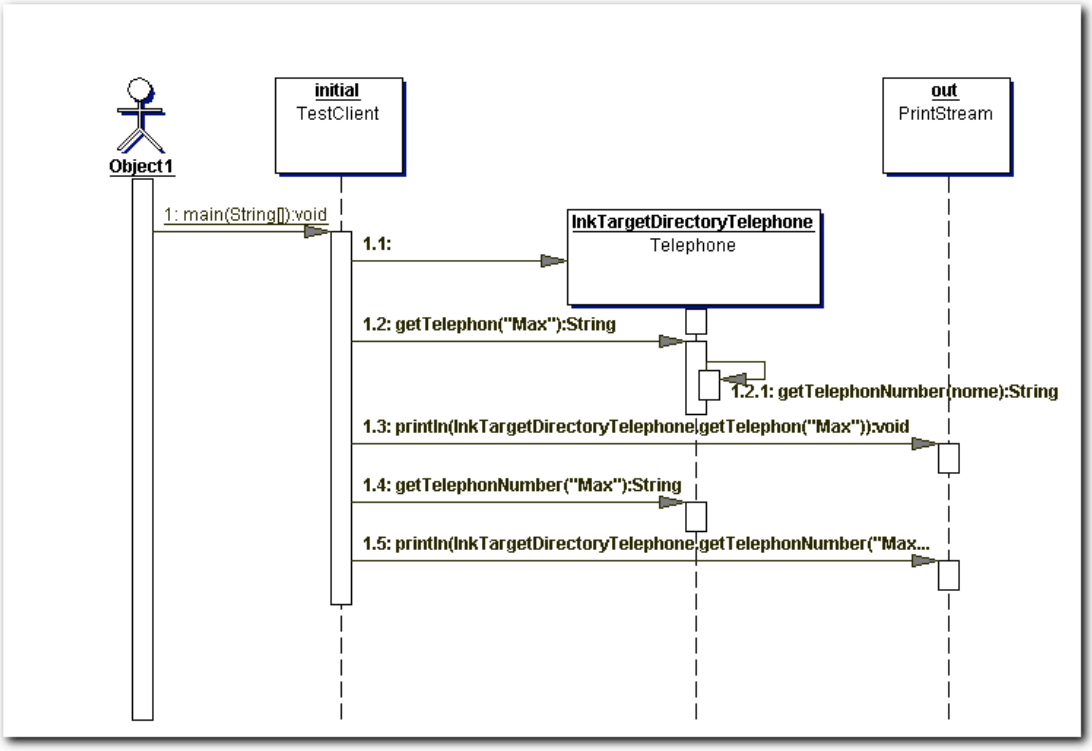


Figura 170: sequence diagram della applicazione client

Infine il client di prova della applicazione che mostra come sia assolutamente indifferente utilizzare un metodo piuttosto che l'altro. Il funzionamento del pattern è schematizzato nel sequence-diagram in *figura 170*.

```
/**
 * CLIENT
 */
public class TestClient {
	public static void main(String[] argv) {
		lnkTargetDirectoryTelephone = new Telephone();
		System.out.println(lnkTargetDirectoryTelephone.getTelephon("Max"));
		System.out.println(lnkTargetDirectoryTelephone.getTelephonNumber("Max"));
	}
	public static TargetDirectoryTelephone lnkTargetDirectoryTelephone;
}
```

Un esempio (variante object):

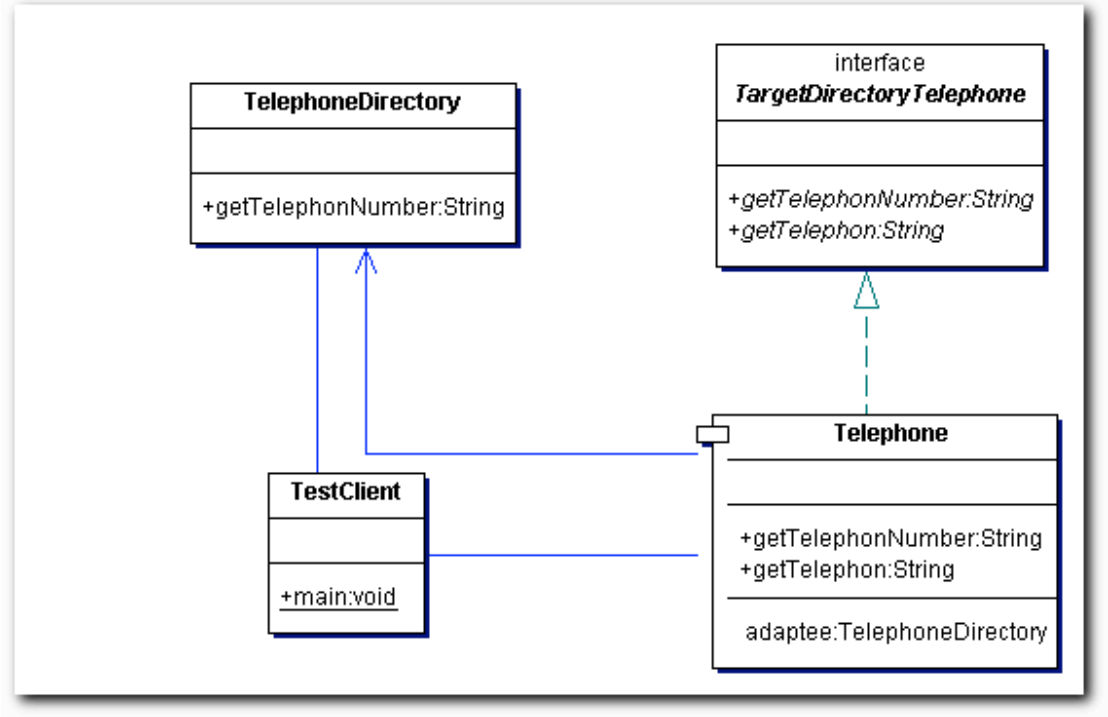


Figura 171: Esempio di variante object di adapter

Nell'esempio utilizzeremo la variante object del pattern modificando il codice sorgente presentato nell'esercizio precedente. Nella *figura 171* è mostrato il class-diagram della nuova applicazione da cui si nota immediatamente che, la classe adapter (*Telephone*) contiene un attributo *adaptee* con relativo metodo setter, mediante il quale utilizza la composizione di oggetti per fornire un interfaccia utilizzabile dalla applicazione client.

```
/**
 * TARGET
 */
public interface TargetDirectoryTelephone {
	String getTelephonNumber(String nome);
	String getTelephon(String nome);
}
```

```
/**
 * ADAPTER
 */
public class Telephone implements TargetDirectoryTelephone {
	public void setAdaptee(TelephoneDirectory adaptee){
		this.adaptee = adaptee;
	}

	public String getTelephonNumber(String nome){
		return adaptee.getTelephonNumber(nome);
	}

	public String getTelephon(String nome){
```

```
                return adaptee.getTelephonNumber(nome);
        }
        private TelephoneDirectory adaptee;
}
```

```
/**
 * ADAPTEE
 */
public class TelephoneDirectory {
        public String getTelephonNumber(String nome){
                return "Il numero di telefono di "+nome+" e' 098765433";
        }
}
```

```
/**
 * CLIENT
 */
public class TestClient {
        public static void main(String[] argv) {
                lnkTelephone = new Telephone();
                lnkTelephoneDirectory = new TelephoneDirectory();
                lnkTelephone.setAdaptee(lnkTelephoneDirectory);
                System.out.println(lnkTelephone.getTelephon("Max"));
                System.out.println(lnkTelephone.getTelephonNumber("Max"));
        }
        public static Telephone lnkTelephone;
        public static TelephoneDirectory lnkTelephoneDirectory;
}
```

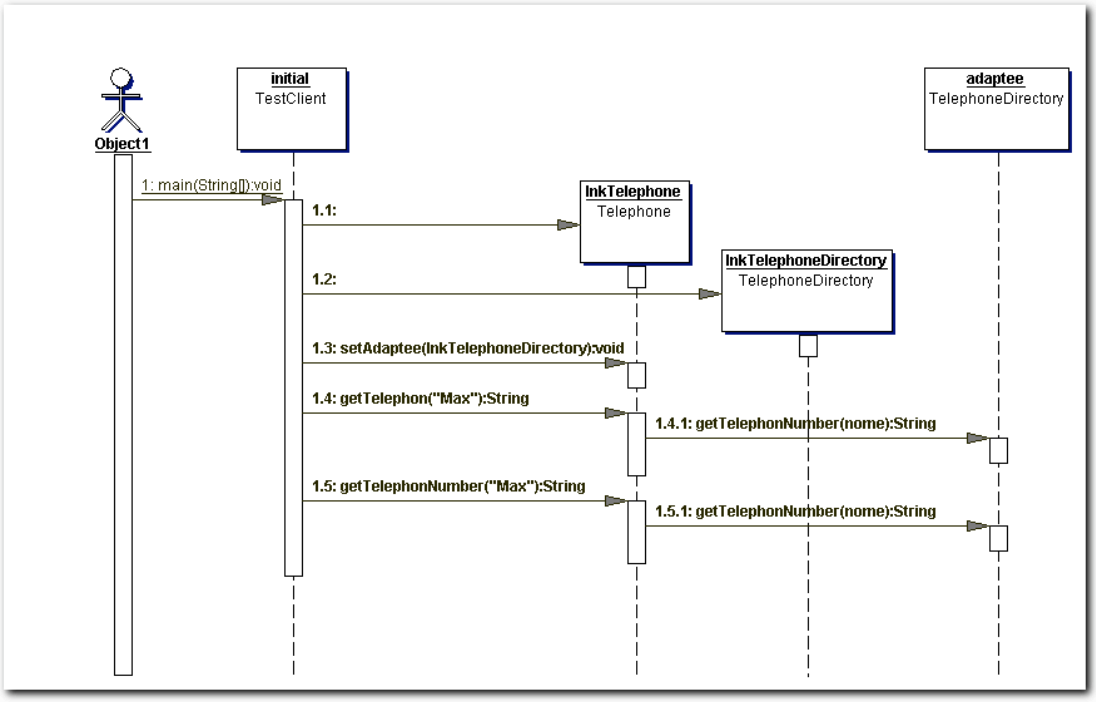


Figura 172: sequence diagram della applicazione client

25.3 Bridge

Il pattern Bridge può essere utilizzato per separare l'interfaccia di una classe dalla sua implementazione affinchè entrambe possano essere modificate indipendentemente l'una dall'altra.
A prima vista, questo pattern potrebbe sembrare simile a quello precedente in cui la classe *Adapter* viene utilizzata per modificare l'interfaccia di un oggetto. Tuttavia, l'intento del pattern Adapter è quello di modificare l'aspetto di un oggetto affinché possa essere adattato al contesto, mentre il pattern Bridge tenta semplicemente di separare interfaccia da implementazione, indipendentemente dal client che dovrà utilizzare l'oggetto.

Il problema:

- *Dobbiamo creare una o più classi mantenendone l'interfaccia verso il client costante;*
- *Non possiamo utilizzare l'ereditarietà per separare l'implementazione di una classe dalla sua interfaccia;*
- *Le eventuali modifiche all'implementazione della classe debbono essere nascoste al client.*

Il disegno:

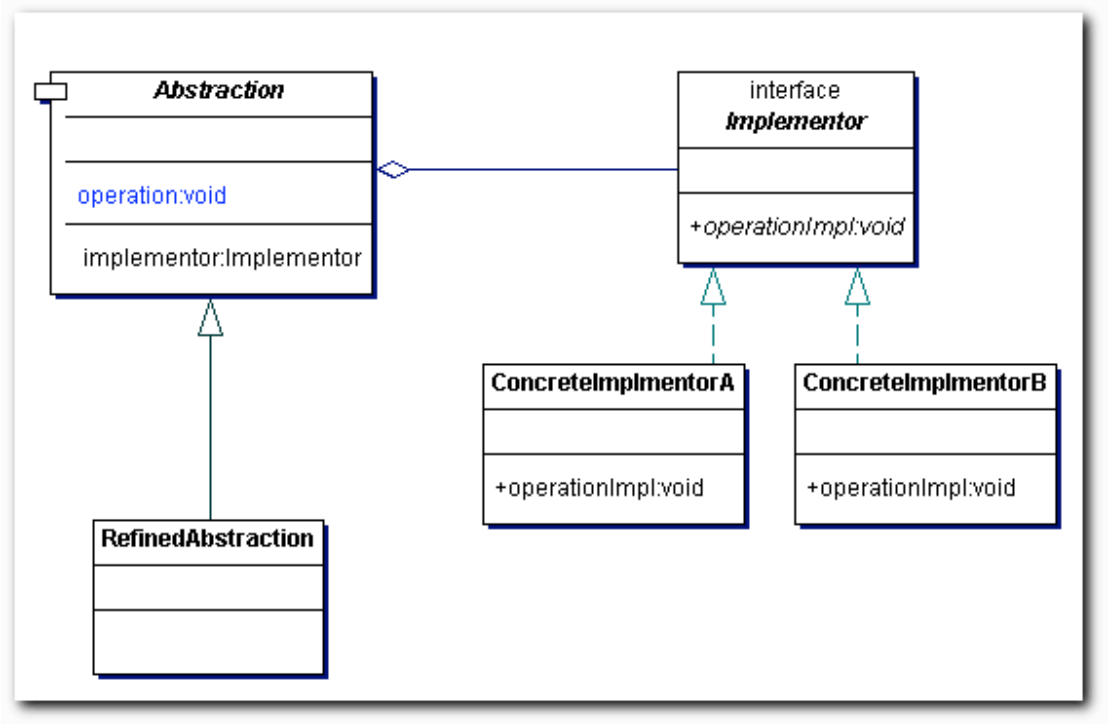


Figura 173: Bridge Pattern

Come funziona:

La classe astratta *Abstraction* rappresenta il ponte (Bridge) verso gli oggetti da mascherare e ne rappresenta quindi l'interfaccia verso il client. La classe *RefinedAbstraction* estende *Abstraction* e oltre a consentire di allocare un oggetto di tipo *Abstraction*, consente di modificare o perfezionare le operazioni della classe padre.
Gli oggetti da mascherare sono rappresentati dalle classi *ConcreteImplementor* che, contengono l'implementazione delle operazioni i cui prototipi sono definiti nell'interfaccia *Implementor*.
La classe *Abstraction* contiene una aggregazione di oggetti di tipo *Implementor* (composizione) e mediante il metodo *getImplementor()* consente di selezionare, se necessario, l'oggetto *ConcreteImplementor* da utilizzare per eseguire *operation()* in base agli attributi in input al metodo. La sequenza delle chiamate è schematizzata nel sequence-diagram in *Figura 174*.

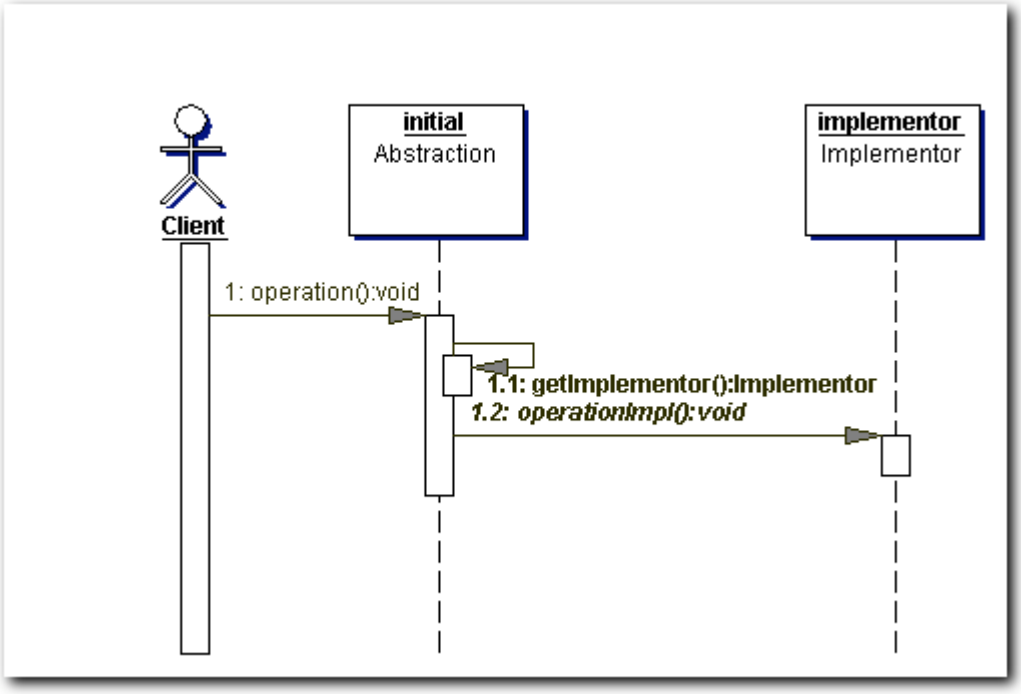


Figura 174: Bridge Pattern: chiamare metodi di interfaccia

Il codice:

Nel prossimo esempio, utilizzeremo la classe *java.util.Vector* per creare un oggetto *Collection* che rappresenta collezione di oggetti. Utilizzeremo Bridge per mascherare al client i dettagli dell'implementazione dell'oggetto che conterrà gli oggetti da collezionare.

Di seguito, modificheremo l'oggetto *Collection* affinché utilizzi un array per contenere i dati da collezionare, e verificheremo che, modificando l'implementazione, l'interfaccia non cambia.
Le classi protagoniste dell'esempio e le loro relazioni, sono schematizzate nel diagramma in *Figura 175*.

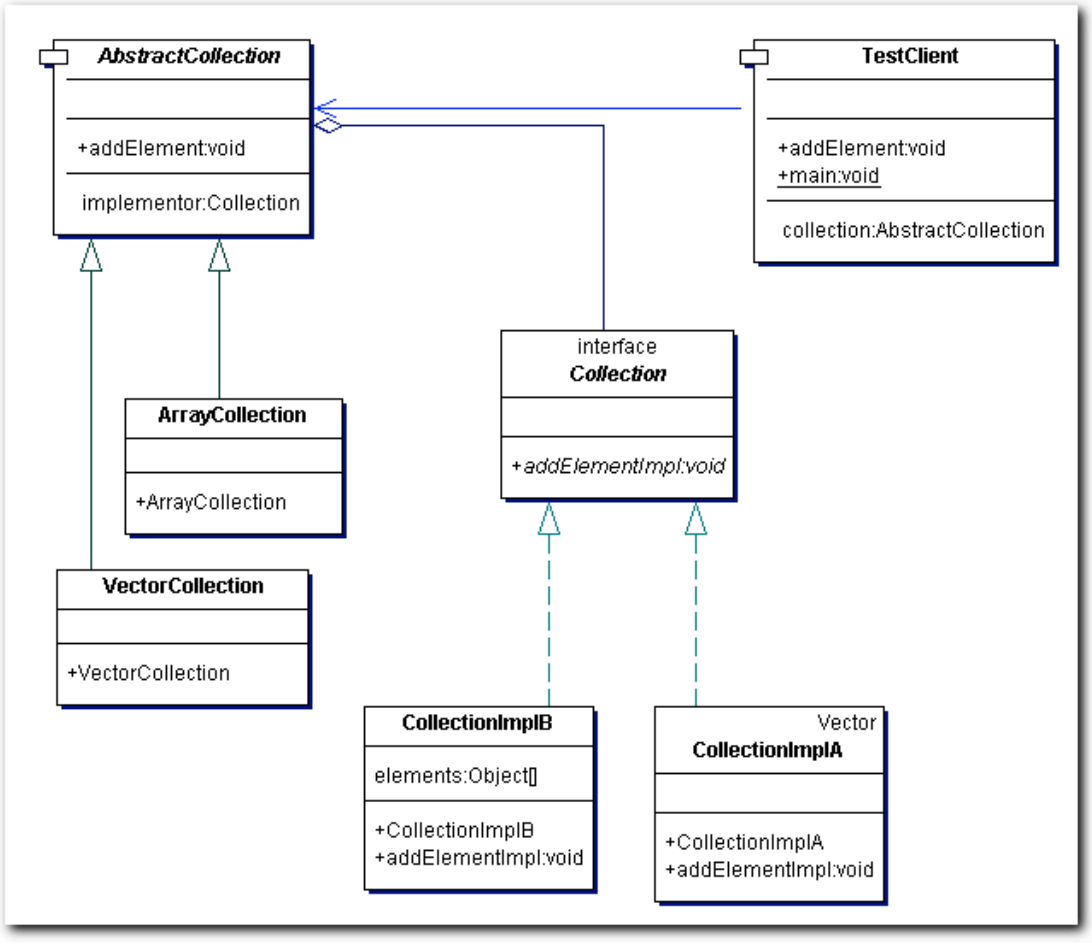


Figura 175: Bridge pattern: l'esercizio

```
package src.pattern.bridge.example;
/**
 * ABSTRACTION
 */
public abstract class AbstractCollection {
	public Collection getImplementor()  {
		return lnkImplementor;
	}

	public void addElement(Object element){
		getImplementor().addElementImpl(element);
	}

	public Collection lnkImplementor;
}
```

```
package src.pattern.bridge.example;
import src.pattern.bridge.Abstraction;

/**
 * REFINED ABSTRACTION
 * Versione specializzata per l'uso della collezione
 * realizzata mediante array
 */
public class ArrayCollection extends AbstractCollection {
	public ArrayCollection() {
		super();
		lnkImplementor = new CollectionImplA();
	}
}
```

```
package src.pattern.bridge.example;
import src.pattern.bridge.Abstraction;

/**
 * REFINED ABSTRACTION
 * Versione specializzata per l'uso della collezione
 * realizzata ereditarietà da Vector
 */
public class VectorCollection extends AbstractCollection {
	public VectorCollection() {
		super();
		lnkImplementor = new CollectionImplB();
	}
}
```

```
package src.pattern.bridge.example;
/**
 * IMPLEMENTOR
 */
public interface Collection {
	public void addElementImpl(Object element);
}
```

```
package src.pattern.bridge.example;
import src.pattern.bridge.Implementor;
/**
 * CONCRETE IMPLEMENTOR
 * Versione di collection realizzata mediante un array
 * di tipi  Object
 */
public class CollectionImplB implements Collection {
	public CollectionImplB() {
		elements = new Object[100];
	}

	public void addElementImpl(Object element){
		elements[lastindex]= element;
		lastindex++;
	}
	Object[] elements = null;
	int lastindex=0;
}
```

```
package src.pattern.bridge.example;
import src.pattern.bridge.Implementor;
import java.util.Vector;
/**
 * CONCRETE IMPLEMENTOR
 * Versione di collection realizzata per ereditarietà
 * da Vector
 */
public class CollectionImplA extends Vector implements Collection {
	public CollectionImplA() {
		super();
	}

	public void addElementImpl(Object element){
		insertElementAt(element,this.size());

	}
}
```

```
package src.pattern.bridge.example;
public class TestClient {
	public src.pattern.bridge.example.AbstractCollection getCollection(){
		return collection;
	}
	public void setCollection(src.pattern.bridge.example.AbstractCollection collection){
		this.collection = collection;
	}

	public void addElement(Object element) {
		getCollection().addElement(element);
	}

	public static void main(String[] argv) {
		TestClient tc = new TestClient();
		src.pattern.bridge.example.AbstractCollection arraydefined = new ArrayCollection();
		src.pattern.bridge.example.AbstractCollection vectordefined
			= new VectorCollection();

		//Utilizziamo l'oggetto collection definito per ereditarietà da Vector
		tc.setCollection(vectordefined);
		tc.addElement(new String("Elemento della collezione"));

		//Allo stesso modo, utilizziamo l'oggetto
		//collection definito utilizzando un array di Object
		tc.setCollection(arraydefined);
		tc.addElement(new String("Elemento della collezione"));
	}

	private AbstractCollection collection;
}
```

Infine, la classe client della nostra applicazione

25.4 Decorator

Applicare il pattern Decorator ad un'oggetto, è come modificarne l'aspetto in maniera dinamica aggiungendo operazioni al run-time senza utilizzare il meccanismo della ereditarietà. Di fatto, lo scopo di Decorator è quello di aggiungere funzionalità ad un oggetto e non ad una classe.

Il problema:

- *Dobbiamo aggiungere funzionalità ad un oggetto in maniera dinamica e trasparente ad altri oggetti;*
- *Dobbiamo aggiungere funzionalità che possono essere distaccate dall'oggetto;*
- *Dobbiamo aggiungere funzionalità ad una classe ma, non è possibile utilizzare il meccanismo dell'ereditarietà oppure, vogliamo limitare la granularità di una applicazione limitando in profondità le gerarchie di ereditarietà.*

Il disegno:

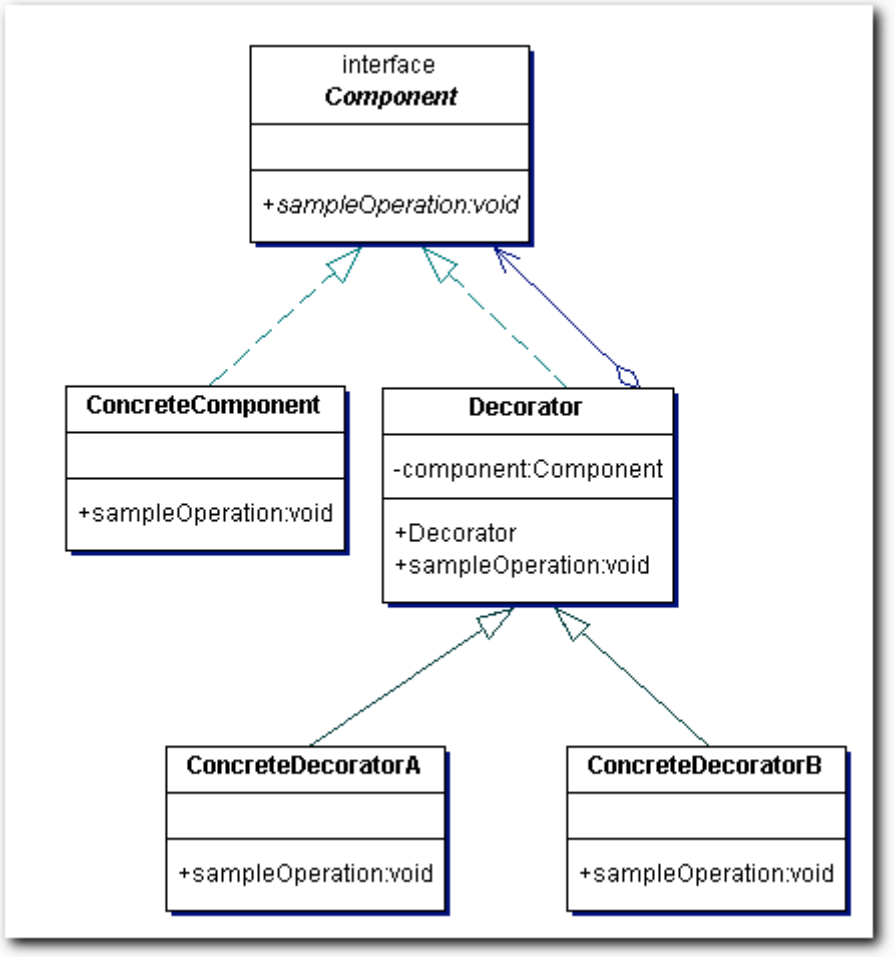


Figura 176: Pattern Decorator

Come funziona:

Un *Decorator* è un oggetto che ha un interfaccia identica all'oggetto da modificare: *Component*. *Component* a sua volta è anche un dato membro per la classe *Decorator*. Ogni volta che si accede ad una funzionalità di *Decorator*, la chiamata viene riggirata alla relativa funzionalità di *Component* . L'oggetto *ConcreteComponent* rappresenta l'implementazione dell'interfaccia *Component*. Le funzionalità aggiuntive possono essere aggiunte nella definizione del metodo di *Decorator* prima o dopo la chiamata al metodo di *Component* come mostrato nel prossimo sequence-diagram.

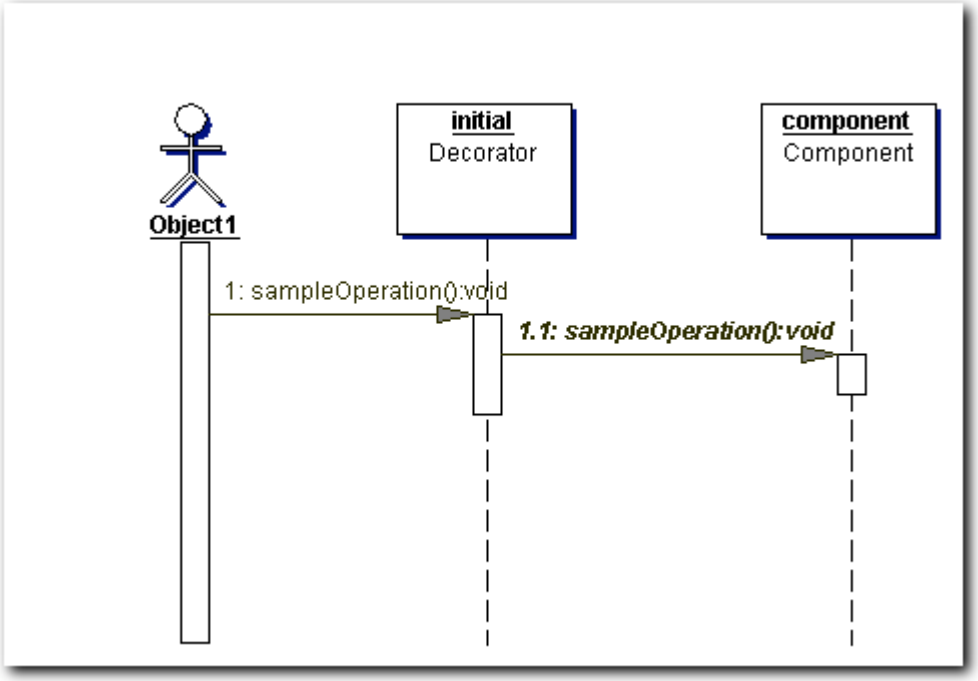


Figura 177: Sequence-Diagram di Decorator

Questo pattern rappresenta uno strumento di programmazione estremamente potente e flessibile in quanto, sottoclassando la classe *Decorator*, è possibile creare infinite variazioni di questo oggetto (*ConcreteDecorator*).

Il codice:

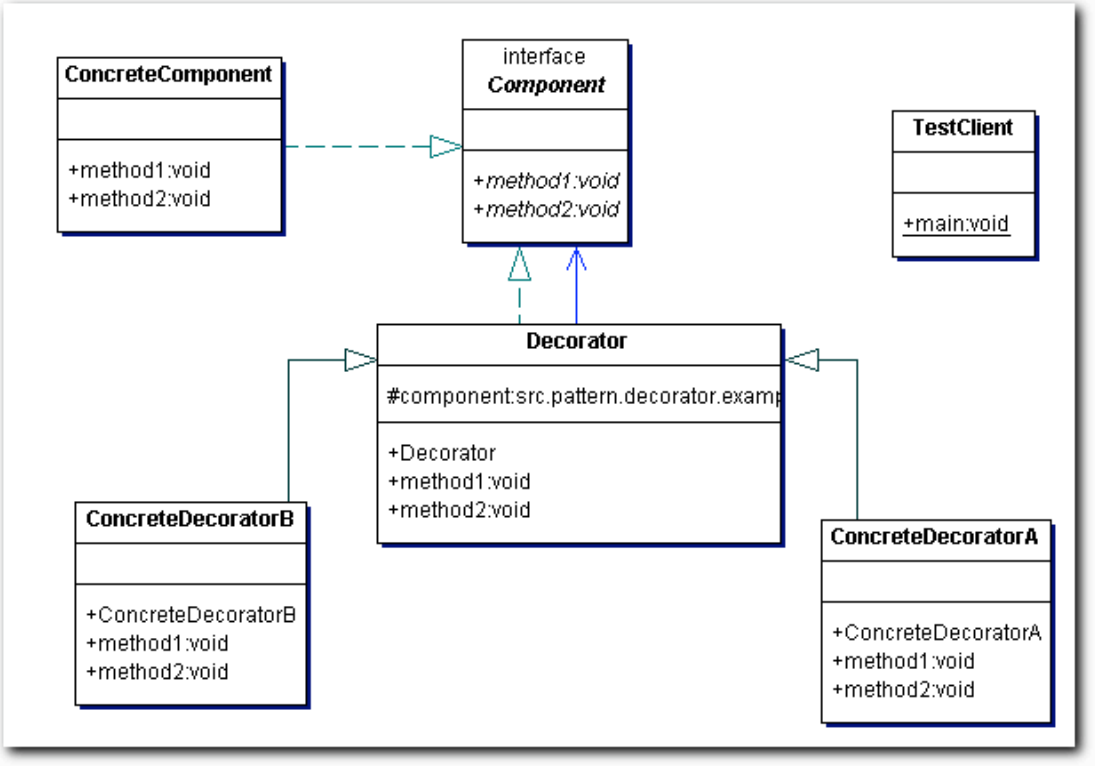


Figura 178: Esempio per decorator

Nel prossimo esempio verrà dimostrato quanto sia semplice aggiungere funzionalità ad un oggetto decorandolo al run-time creando infinite possibili decorazioni.

```
package src.pattern.decorator.example;
/**
 * COMPONENT
 */
public interface Component {
	public void method1();
	public void method2();
}
```

```
package src.pattern.decorator.example;
/**
 * CONCRETE COMPONENT
 */
public class ConcreteComponent implements Component{

	public void method1(){
		System.out.println("Metodo 1 di Concrete Component");
	}
	public void method2(){
		System.out.println("Metodo 2 di Concrete Component");
	}

}
```

```
package src.pattern.decorator.example;
/**
 * DECORATOR
 */
public class Decorator implements Component {
	public Decorator(src.pattern.decorator.example.Component aComp) {
		component = aComp;
	}
	public void method1(){
		System.out.println("--> Funzionalità aggiunta da Decorator");
		component.method1();
	}
	public void method2(){
		System.out.println("--> Funzionalità aggiunta da Decorator");
		component.method2();
	}
	protected src.pattern.decorator.example.Component component;
}
```

```
package src.pattern.decorator.example;

/**
 * CONCRETE DECORATOR
 */
public class ConcreteDecoratorB extends Decorator {
	public ConcreteDecoratorB(src.pattern.decorator.example.Component aComp) {
		super(aComp);
	}

	public void method1() {
		System.out.println("--> Funzionalità aggiunta da ConcreteDecoratorB");
		component.method1();
	}

	public void method2() {
		System.out.println("--> Funzionalità aggiunta da ConcreteDecoratorB");
		component.method2();
	}

}
```

```
package src.pattern.decorator.example;
/**
 * CONCRETE DECORATOR
 */
public class ConcreteDecoratorA extends Decorator {
	public ConcreteDecoratorA(src.pattern.decorator.example.Component aComp) {
		super(aComp);
	}

	public void method1() {
		System.out.println("--> Funzionalità aggiunta da ConcreteDecoratorA");
		component.method1();
	}

	public void method2() {
		System.out.println("--> Funzionalità aggiunta da ConcreteDecoratorA");
```

```
                component.method2();
        }
}
```

```
package src.pattern.decorator.example;
/**
 * CLIENT DI TEST
 */
public class TestClient {
        public static void main(String[] args){
                System.out.println("Creo un componente  non decorato:");
                Component comp = new ConcreteComponent();
                comp.method1();
                comp.method2();
                System.out.println("---------------------------------------------------------------");
                System.out.println("Creo un componente decorando ConcreteComponent con
                        Decorator:");
                Component decorator = new Decorator(comp);
                decorator.method1();
                decorator.method2();
                System.out.println("---------------------------------------------------------------");
                System.out.println("Creo un componente decorando decorator con
                        ConcreteDecoratorA:");
                Component decorator2 = new ConcreteDecoratorA(decorator);
                decorator2.method1();
                decorator2.method2();
                System.out.println("---------------------------------------------------------------");
                System.out.println("Creo un componente decorando decorator2 con
                        ConcreteDecoratorB:");
                Component decorator3 = new ConcreteDecoratorB(decorator2);
                decorator3.method1();
                decorator3.method2();
 }
}
```

25.5 Composite

Capita spesso di dover creare componenti in forma di semplici oggetti o oggetti composti. Composite copre entrambi i casi e consente di comporre oggetti utilizzando strutture ad albero per rappresentare gerarchie "tutto-o-parti". Di fatto, un oggetto Composite è una collezione di oggetti organizzati ad albero: le foglie dell'albero rappresentano oggetti semplici, i nodi collezioni di oggetti (*Figura 170*). In entrambe i casi, il client continuerà ad utilizzare oggetti semplici o composti utilizzando la stessa interfaccia, senza doversi preoccupare della natura dell'oggetto che sta utilizzando.

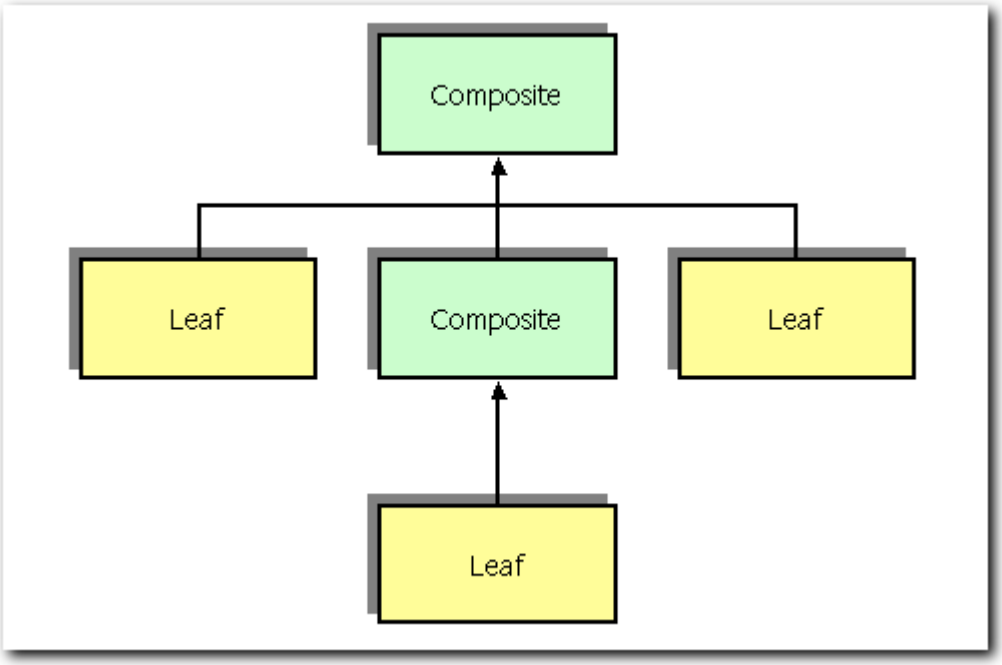


Figura 179: Struttura ad albero di composite

Le applicazioni grafiche sono l'esempio classico, utilizzato per descrivere questo pattern. Questo tipo di applicazioni raggruppano ricorsivamente componenti semplici (cerchi, rettangoli, linee, quadrati ecc.) o composte, per creare strutture complesse (*Figura 180*). Ogni componente, semplice o composta che sia, ha in comune con le altre alcuni metodi primitivi.
Ad esempio, una componente *immagine* è una collezione di componenti semplici che rappresentano le figure geometriche. Condividendo con queste un metodo *draw*, l'immagine potrà essere creata semplicemente invocando il metodo suddetto e propagandone l'esecuzione a tutti gli oggetti della collezione fino ad arrivare alle componenti semplici.

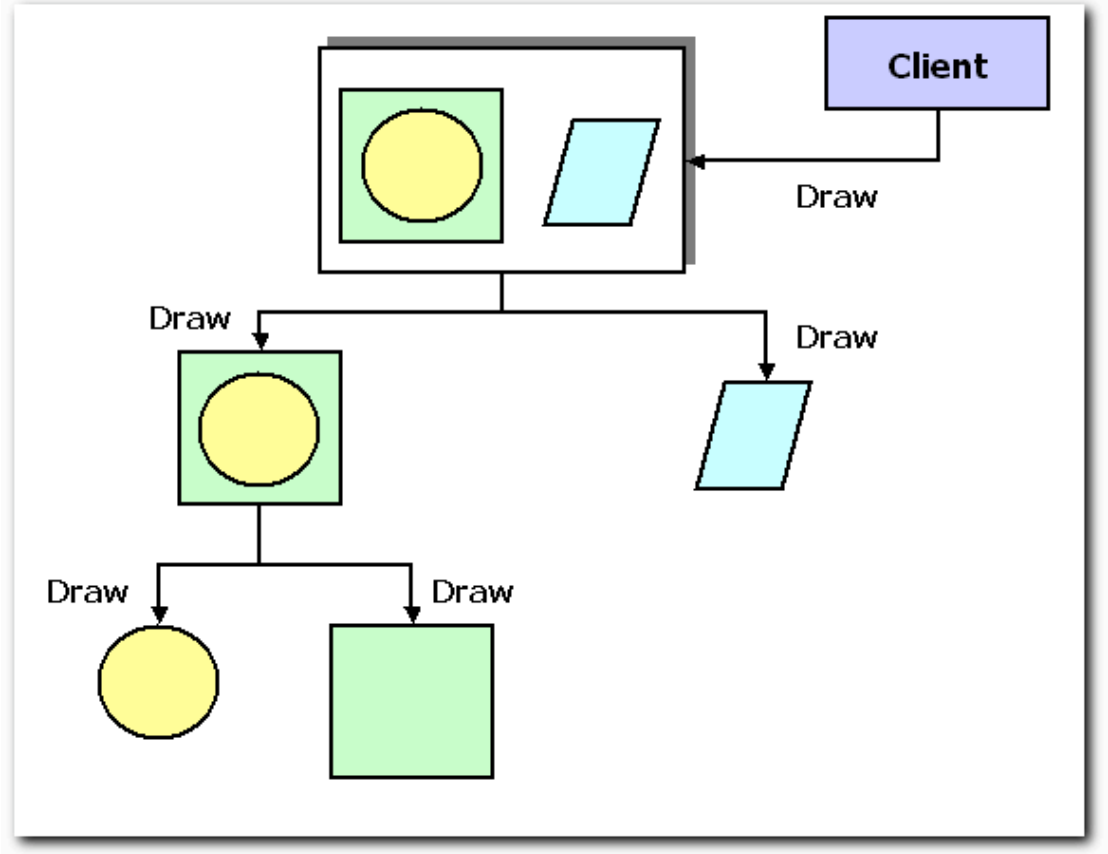


Figura 180: Struttura ricorsiva di una immagine

Il problema:

- *Vogliamo rappresentare gerarchie tutto o parti;*
- *Vogliamo che il client ignori le differenze tra oggetti semplici e oggetti composti, utilizzando sempre la stessa interfaccia.*

Il disegno:

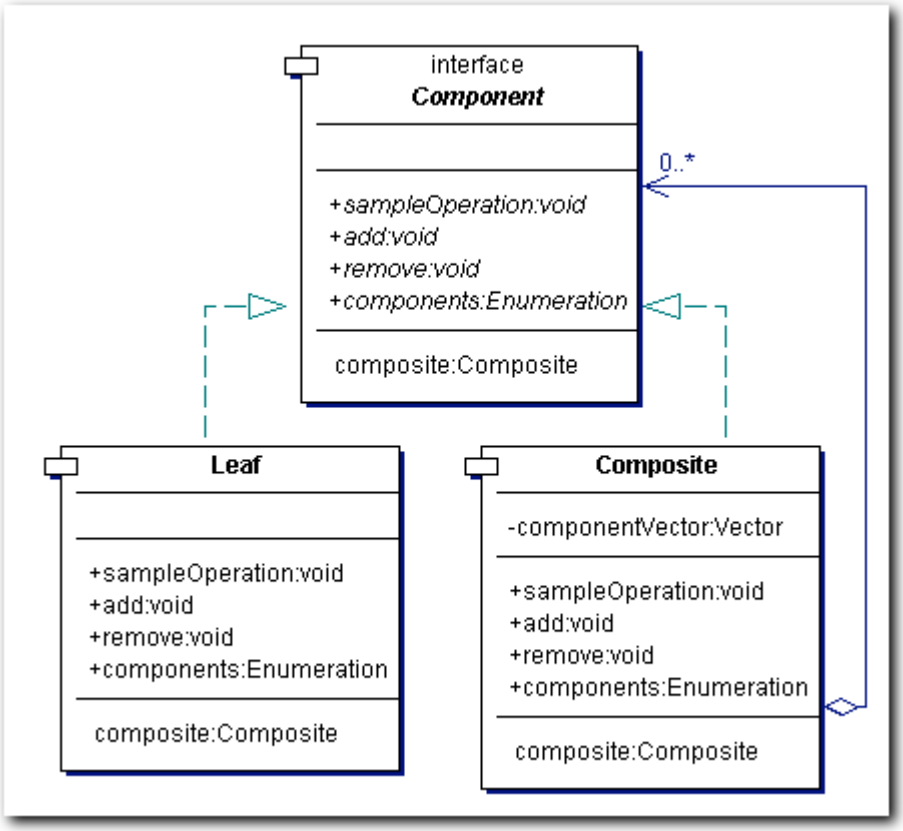


Figura 181: Composite Pattern

Come lavora:

I partecipanti di Composite sono: l'interfaccia *Component*, la classe *Composite* e la classe *Leaf*. *Component* contiene i prototipi dei metodi degli oggetti *Leaf* o *Composite* e rappresenta ciò che il client percepisce della struttura sottostante. La classe *Leaf* rappresenta le foglie dell'albero e contiene la definizione degli oggetti primitivi che compongono la struttura. La classe *Composite* rappresenta i nodi dell'albero. Come tale, *Composite* è una collezione di oggetti dello stesso tipo od oggetti di tipo *Leaf*. Compito della classe è inoltre quello di propagare alle classi figlie le chiamate ai metodi comuni definiti all'interno di *Component*.

Il codice:

Nel prossimo esempio simuleremo la costruzione del quadro astratto mostrato in *figura 180*. Le classi utilizzate per realizzare l'esercizio sono schematizzate nella prossima figura. In particolare, la classe *Form* rappresenta la classe *Leaf* del pattern e la classe Picture C la classe *Composite*.

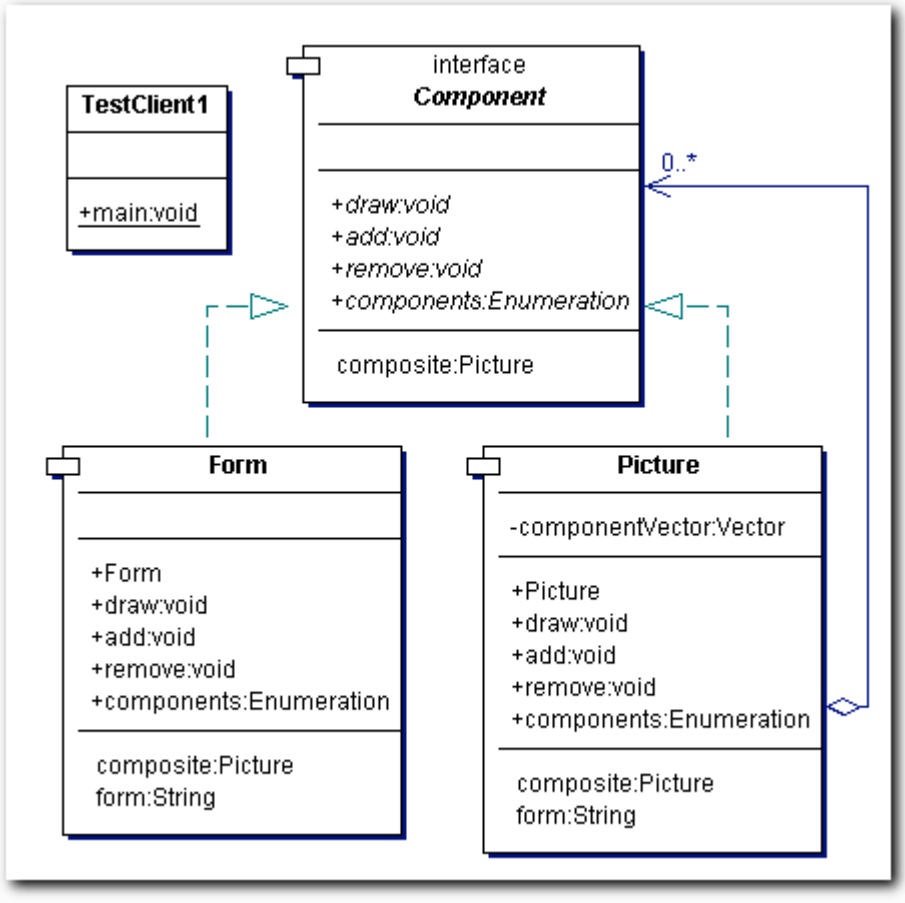


Figura 182: Composite Pattern: l'esempio

```
package src.pattern.composite;
import java.util.Enumeration;
/**
 * COMPONENT
 */
public interface Component {
	Picture getComposite();
	void draw();
	void add(Component component);
	void remove(Component component);

	Enumeration components();
}
```

```
package src.pattern.composite;
import java.util.Vector;
import java.util.Enumeration;

/**
 * COMPOSITE
 */
public class Picture implements Component {
```

```
	public Picture(String text) {
		setForm(text);
	}


	public Picture getComposite(){
		return this;
	}

	public void draw(){
		System.out.println("Creazione dell'immagine: "+getForm());
		java.util.Enumeration components = components();
		while (components.hasMoreElements()) {
			((Component)components.nextElement()).draw();
		}
	}

	public void add(Component component){
		componentVector.addElement(component);
	}

	public void remove(Component component){
		componentVector.removeElement(component);
	}

	public Enumeration components(){
		return componentVector.elements();
	}

	public String getForm(){ return form; }

	public void setForm(String form){ this.form = form; }


	private Vector componentVector = new Vector();
	private String form;
}
```

```
package src.pattern.composite;
import java.util.Enumeration;
/**
 * LEAF
 */
public class Form implements Component {
	public Form(String text) {
		setForm(text);
	}

	public Picture getComposite(){
		// Write your code here
		return null;
	}

	public void draw(){
		System.out.println("Aggiungo all'immagine: "+ getForm());
	}

	public void add(Component component){
```

```
                // Write your code here
        }

        public void remove(Component component){
                // Write your code here
        }

        public Enumeration components(){
                // Write your code here
                return null;
        }

        public String getForm(){ return form; }

        public void setForm(String form){ this.form = form; }

        private String form;
}
```

Il client di test:

```
package src.pattern.composite;
public class TestClient1 {
        public static void main(String[] argv) {
                Component quadroastratto =
                        new Picture("Quadrato con cerchio dentro e parallelepipedo");

                //Creazione delle primitive
                Component cerchio = new Form("cerchio");
                Component quadrato = new Form("quadrato");
                Component parallelepipedo = new Form("parallelepipedo");

                //Creo l'immagine del quadrato con dentro il cerchio
                Component quadrato_e_cerchio = new Picture("Quadrato con cerchio dentro");
                quadrato_e_cerchio.add(quadrato);
                quadrato_e_cerchio.add(cerchio);

                //Aggiungo l'immagine al quadro astratto
                quadroastratto.add(quadrato_e_cerchio);

                //Aggiungo il parallelepipedo al quadro astratto
                quadroastratto.add(parallelepipedo);

                //Disegno il quadro
                quadroastratto.draw();

        }

}
```

25.6 Façade

La naturale evoluzione di una applicazione comporta spesso un aumento della complessità e spesso, il numero degli oggetti è talmente alto che risulta difficile capire come gli oggetti interagiscono tra loro. Inoltre, molti di questi

oggetti potrebbero essere combinati a formare sottosistemi, ognuno dei quali con una interfaccia molto complessa.
Il pattern Façade definisce una interfaccia semplificata verso questi sottosistemi, nascondendone la complessità interna ed adattandone l'uso al client che ne dovrà fare uso. Nelle prossime figure è rappresentato un sistema sviluppato rispettivamente senza e con Façade.

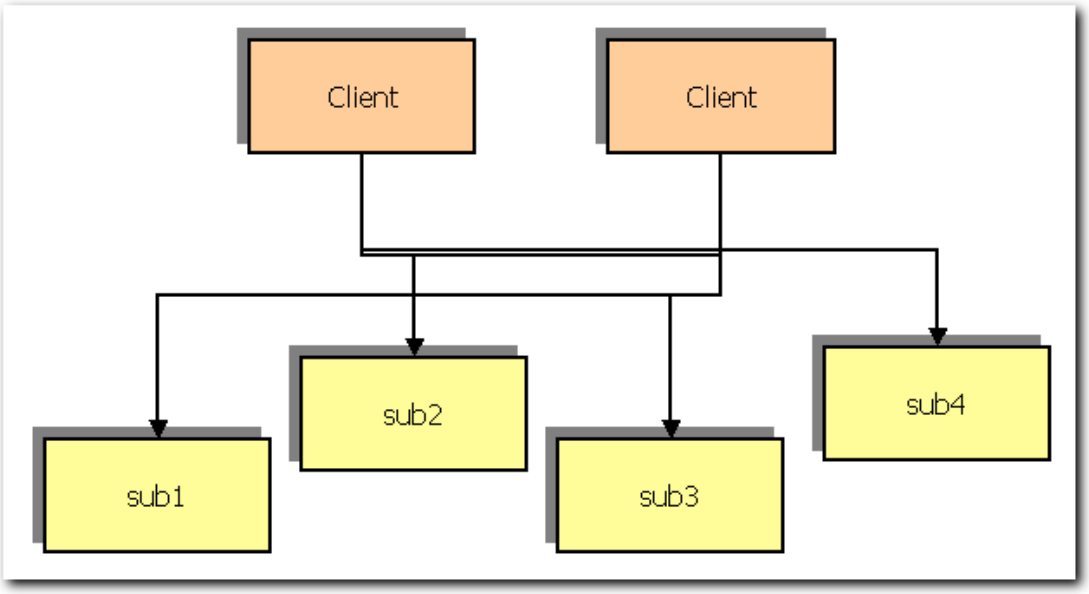


Figura 183: Schema generico di una applicazione

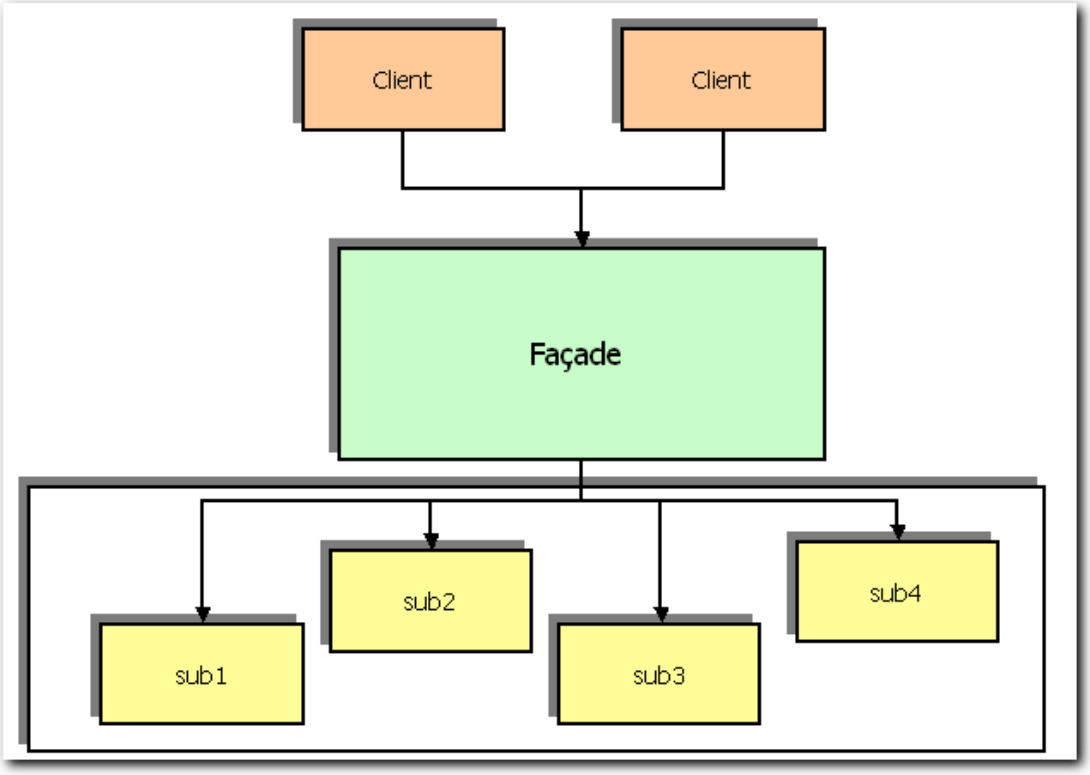


Figura 184: Schema di una applicazione realizzata mediante Facade

L'architettura JDBC rappresenta un esempio classico di architetture che utilizzano questo pattern. Di fatto, mediante JDBC potremo connetterci ad ogni database, senza doverci preoccupare delle specifiche di ogni sistema, utilizzando un interfaccia comune e semplificata.

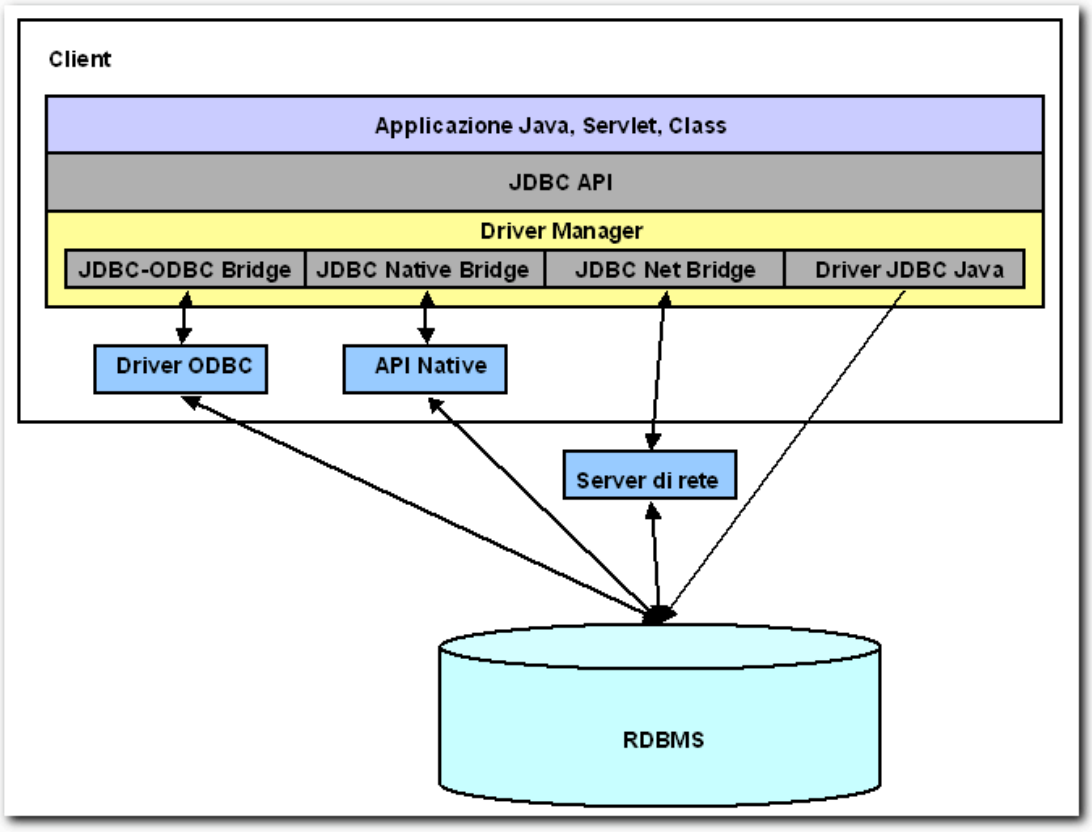


Figura 185: L'architettura JDBC è un Facade

Il problema:

- *Vogliamo semplificare l'accesso ad un sistema molto complesso;*
- *Vogliamo ridurre il numero delle dipendenze tra i client e le classi che compongono il sistema;*
- *Vogliamo suddividere un sistema complesso in sottosistemi semplici.*

Il disegno:

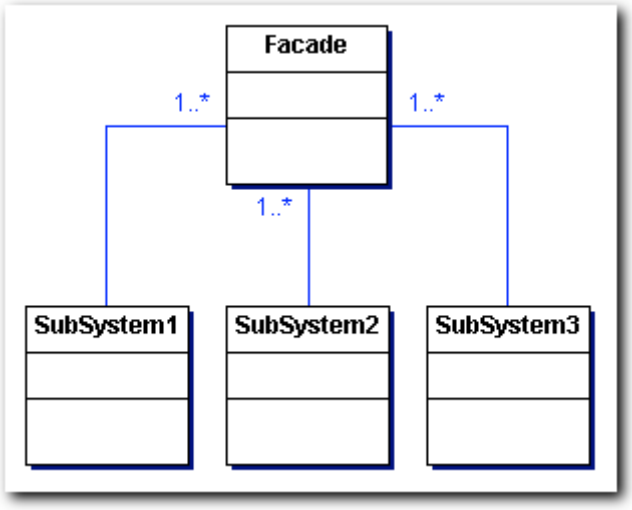


Figura 186: Il pattern facade

Come funziona:

Differentemente dagli altri pattern i cui partecipanti hanno strutture e funzioni precise con interazioni ben definite, Façade fornisce solo un modello generico per l'organizzazione delle componenti di un sistema in sottosistemi.
Per far questo, il pattern prevede l'uso di una classe *Facade* che rappresenta il sottosistema dal punto di vista del client e contiene le logiche applicative necessarie a definire quale sottosistema utilizzare per servire una richiesta del client. Nessun riferimento può essere fatto ai sottosistemi la cui natura dipende solo dalle necessità del programmatore.

Il codice:

```
/**
 * FACADE
 */
public class Facade
{
	private Subsystem1 subsystem1 = new Subsystem1();
	private Subsystem2 subsystem 2 = new Subsystem2();
	private Subsystem3 subsystem 3 = new Subsystem3();

	public void executeRequest()
	{
		subsystem 1.doRequest();
		subsystem 2. doRequest ();
		subsystem 3. doRequest ();
	}
}
```

```
/**
 * SUBSYSTEM 1
 */
```

```
public class Subsystem1
{
	public void doRequest ()
	{
		System.out.pritnln("Sottosistema 1");
	}
}
```

```
/**
 * SUBSYSTEM 2
 */
public class Subsystem2
{
	public void doRequest ()
	{
		System.out.pritnln("Sottosistema 2");
	}
}
```

```
/**
 * SUBSYSTEM 3
 */
public class Subsystem1
{
	public void doRequest ()
	{
		System.out.pritnln("Sottosistema 3");
	}
}
```

25.7 Flyweight

Flyweight può essere utilizzato in tutti quei casi in cui molti oggetti hanno la necessità di condividere molte informazioni oppure, nel caso in cui molti oggetti privati possono essere rimpiazzati da alcuni oggetti condivisi riducendo drasticamente il consumo della memoria da parte della applicazione. Di fatto, le componenti di una applicazione possono utilizzare due tipi di informazioni: le informazioni intrinseche e quelle estrinseche. Come schematizzato nella *figura 187*, le informazioni estrinseche sono quelle che dipendono dall'oggetto che le gestisce e possono variare da componente a componente. Differentemente, quelle intrinseche sono informazioni immutabili e costanti.

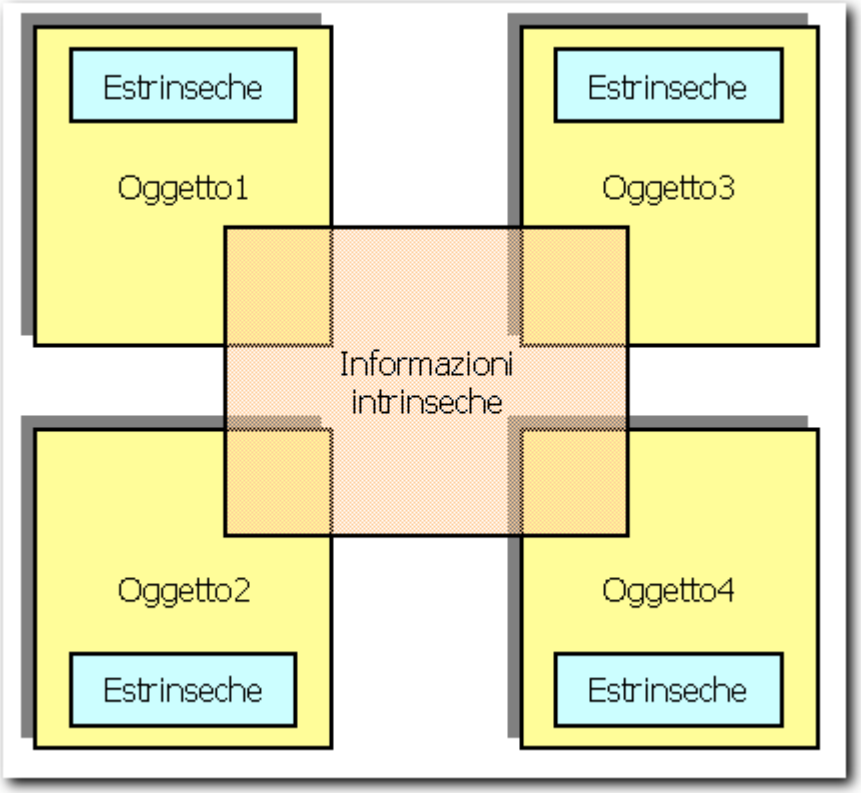


Figura 187: Informazioni intrinseche ed estrinseche

Riuscendo a raggruppare a fattor comune le informazioni intrinseche, è possibile fare in modo che esse vengano raggruppate e condivise tra tutti gli oggetti che ne debbano fare uso, senza memorizzarle tante volte quanti sono gli oggetti che ne necessitano. Diremo quindi che un Flyweight è un oggetto condivisibile che rappresenta informazioni intrinseche, disaccoppiato dalle informazioni estrinseche rappresentate per mezzo di altri oggetti.

Il problema:

- *Una applicazione utilizza un grande numero di oggetti;*
- *La memorizzazione degli oggetti è molto onerosa a causa della quantità delle informazioni;*
- *Le informazioni estrinseche possono essere disaccoppiate dagli oggetti che, di conseguenza, possono essere rimpiazzati da un numero minore di componenti;*
- *L'applicazione non dipende dall'interfaccia degli oggetti.*

Il disegno:

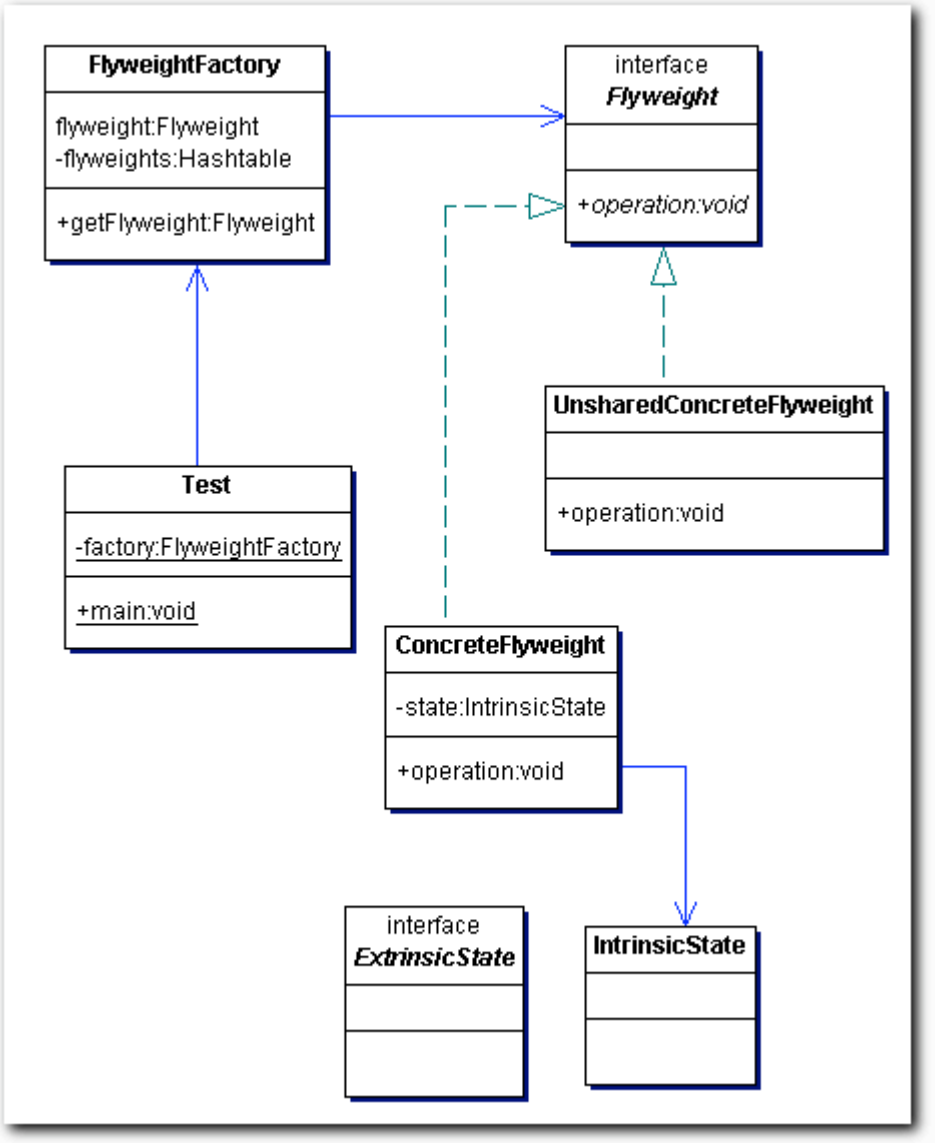


Figura 188: Il pattern Flyweight

Come funziona:

Gli oggetti *Flyweight* implementano l'interfaccia omonima che, rappresenta il punto di vista del client ripetto alle informazioni condivise. *Flyweight* contiene inoltre le definizioni dei metodi necessari a passare alle componenti i dati estrinsechi. In generale, non tutte le sottoclassi di Flyweight debbono essere necessariamente condivisibili (*UnsharedConcreteFlyweigth*). Gli oggetti *ConcreteFlyweigth* sono oggetti condivisibili che implementano i prototipi definiti in *Flyweight*, sono il contenitore per le informazioni intrinseche (*IntrinsicState*) e se necessario, accettano informazioni estrinseche per manipolare i dati (*ExtrinsicState*).

L'oggetto *FlyweigthFactory* crea e colleziona gli oggetti *Flyweight*. Quando un client richiede un *Flyweight*, questa componente restituisce un oggetto esistente o, se non ne esiste nessuno, ne crea una nuovo.

Il codice:

```
package src.pattern.flyweight;
/**
 * FLYWEIGHT
 */
public interface Flyweight
{
	void execute( ExtrinsicState state );
}
```

```
package src.pattern.flyweight;
import java.util.Hashtable;
/**
 * FLYWEIGHT FACTORY
 */
public class FlyweightFactory
{
	Flyweight flyweight =null;
	private Hashtable flyweights = new Hashtable();

	public Flyweight getFlyweight( Object key )
	{
		flyweight = (Flyweight) flyweights.get(key);
		if( flyweight == null )
		{
			flyweight = new ConcreteFlyweight();
			flyweights.put( key, flyweight );
		}
		return flyweight;
	}
}
```

```
package src.pattern.flyweight;
/**
* UNSHARED CONCRETE LYWEIGHT
*/
public class UnsharedConcreteFlyweight implements Flyweight {
	public void execute(ExtrinsicState state) {
	}
}
```

```
package src.pattern.flyweight;
/**
*  CONCRETE FLYWEIGHT
*/
public class ConcreteFlyweight implements Flyweight {
	private IntrinsicState state;

	public void execute(ExtrinsicState state) {
	}
}
```

```
package src.pattern.flyweight;
/**
 * INFORMAZIONI ESTRINSECHE
 */
public interface ExtrinsicState{
}
```

```
package src.pattern.flyweight;
/**
 * INFORMAZIONI INTRINSECHE
 */
public class IntrinsicState{
}
```

Il client di prova:

```
package src.pattern.flyweight;
/**
 * CLIENT
 */
public class Test
{
	public static void main( String[] arg )
	{
		try
		{
			factory = new FlyweightFactory();
			Flyweight fly1 = factory.getFlyweight( "Max" );
			Flyweight fly2 = factory.getFlyweight( "Alex" );
		}
		catch( Exception e )
		{
			e.printStackTrace();
		}
	}

	private static FlyweightFactory factory;
}
```

25.8 Proxy

Proxy è utilizzato quando abbiamo bisogno di rappresentare un oggetto complesso mediante uno più semplice. Se creare un oggetto è costoso in fatto di risorse del computer, Proxy consente di posporne la creazione fino alla prima richiesta del client. Una componente Proxy, tipicamente, ha le stesse operazioni dell'oggetto che rappresenta e dopo averlo caricato, rigira le richieste del client, come mostrato nel prossimo diagramma.

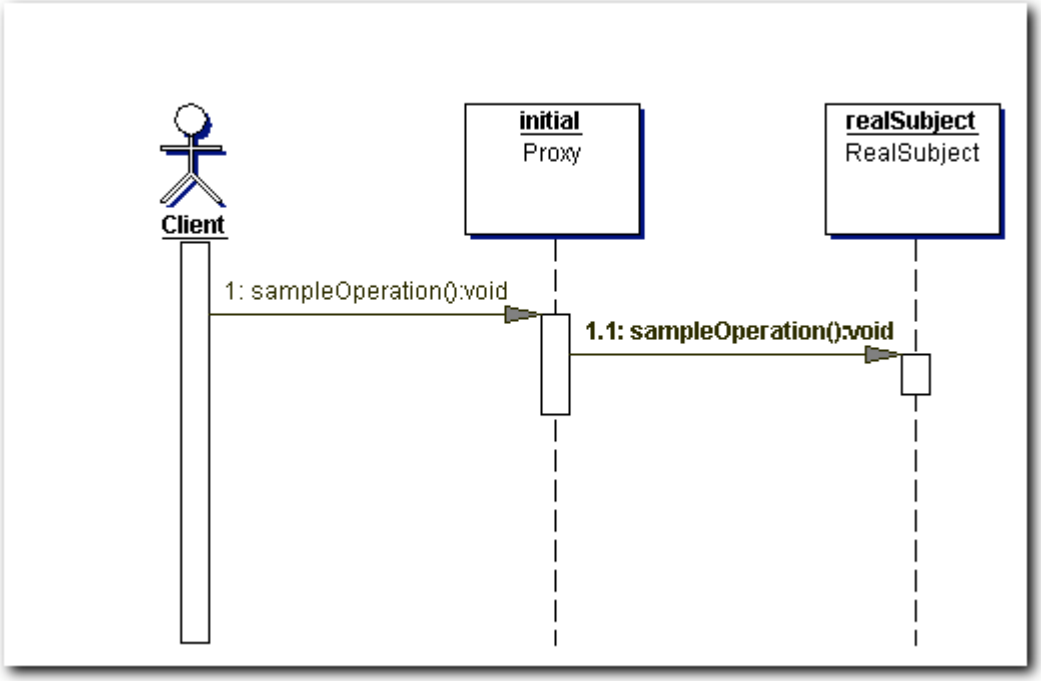


Figura 189: Esecuzione del metodo sampleOperation

Il problema:

- *Un oggetto richiede molto tempo per essere caricato;*
- *L'oggetto da utilizzare è su una macchina remote e richiederebbe troppo tempo caricarlo;*

Il disegno:

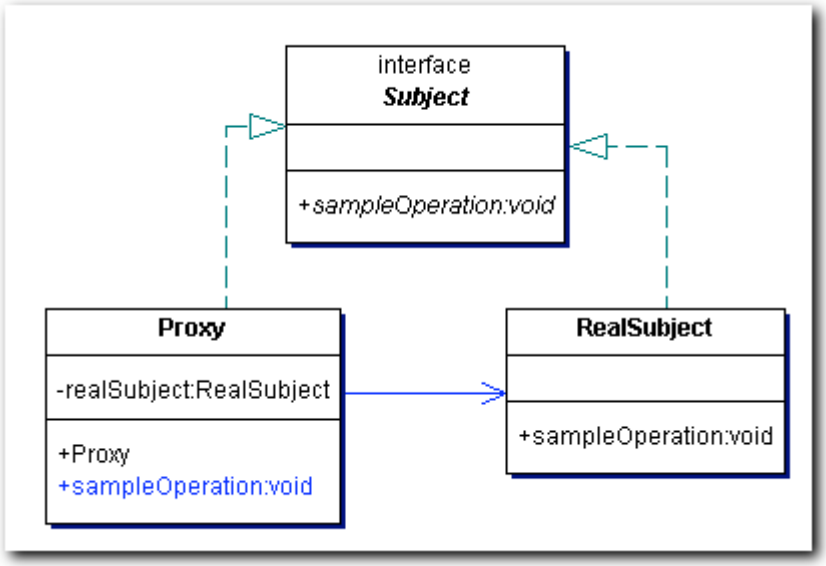


Figura 190: Il pattern Proxy

Come funziona:

L'oggetto *RealSubject*, rappresenta l'oggetto rappresentato da *Proxy* che, mantiene un riferimento a tale componente per potervi accedere su richiesta. Inoltre, *Proxy* è responsabile del ciclo di vita di *RealSubject*: creazione e distruzione. Entrambi *Proxy* e *RealSubject*, devono implementare l'interfaccia *Subject*.
Di fatto, un Proxy può assumere diverse forme e a seconda della sua natura, le sue responsabilità possono variare:

1. *I Proxy remoti, hanno la responsabilità di codificare una richiesta e gli argomenti affinché possano essere trasmessi all'oggetto in esecuzione su una macchina differente;*
2. *I Proxy virtuali, possono memorizzare temporaneamente le richieste consentendo di posticipare l'accesso al client;*
3. *I Proxy di protezione verificano che il client abbia i permessi sufficienti ad accedere all'oggetti rappresentato.*

Un esempio di *Proxy* remoto è rappresentato dall'architettura di RMI (Remote Method Invocation). La componente *Remote Interface* rappresenta l'interfaccia *Subject*, *Stub* e *Skeleton* le componenti *Proxy*.

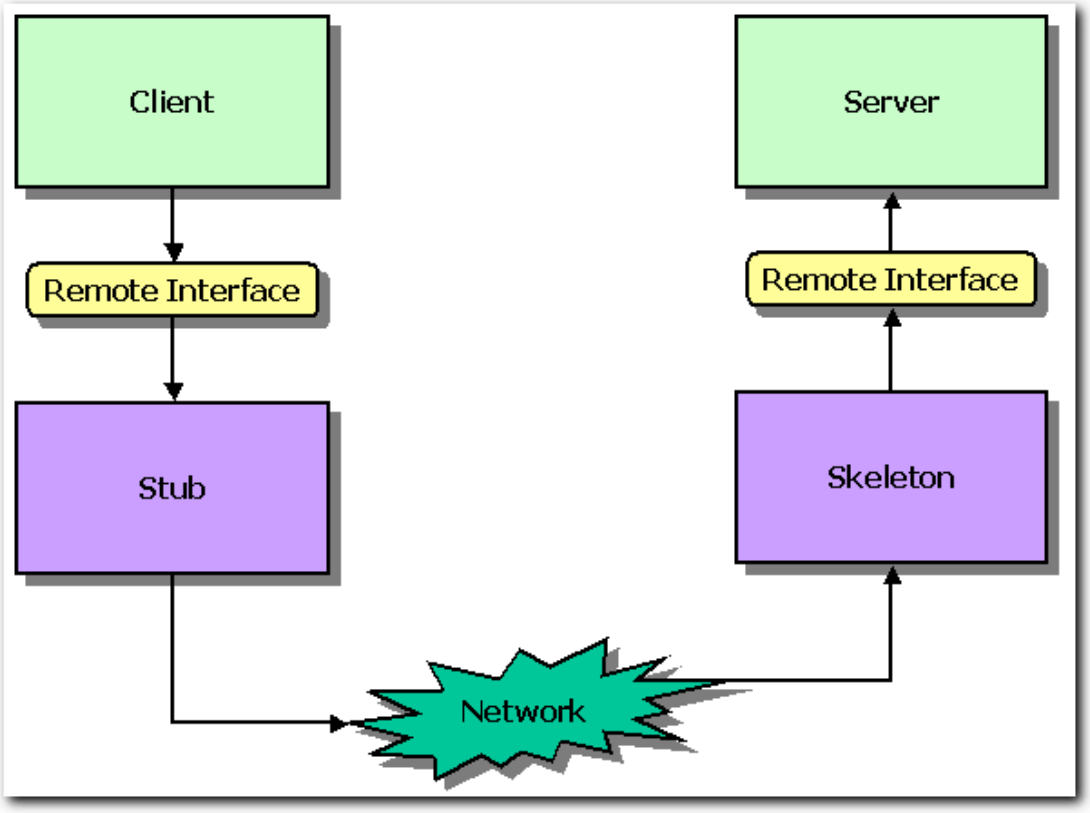


Figura 191: Stub e Skeleton in RMI rappresentano un Proxy Remoto

Il codice:

```
package src.pattern.proxy;
/**
 * SUBJECT
 */
public interface Subject {
	void sampleOperation(String client);
}
```

```
package src.pattern.proxy;
/**
 * PROXY
 */
public class Proxy implements Subject {
	public Proxy(){
	}

	public void sampleOperation(String client){
		if(realSubject==null)
			realSubject = new RealSubject();
		realSubject.sampleOperation(client);
	}

	private RealSubject realSubject = null;
```

```
}
```

```
package src.pattern.proxy;
/**
 * REAL SUBJECT
 */
public class RealSubject implements Subject {
	public void sampleOperation(String client){
		System.out.println("Sono stato eseguito da:"+client);
	}
}
```

Il client di test:

```
package src.pattern.proxy;
/**
 * CLIENT DI PROVA
 */
public class Test {
	public static void main(String[] argv) {
		lnkSubject = new Proxy();
		lnkSubject.sampleOperation("CLIENT DI PROVA");
	}

	private static Subject lnkSubject;
}
```

26 BEHAVIORAL PATTERN

26.1 Introduzione

I behavioral pattern affrontano il problema della comunicazione tra oggetti.

Behavioral Pattern	
Nome	**Descrizione**
Chain of responsibility	Fornisce un modello che consente di creare una catena di oggetti indipendenti lungo la quale propagare le richieste del client.
Command	Separa l'esecuzione di un commando dal contesto che lo ha prodotto.
Interpreter	Definisce come includere elementi del linguaggio all'interno di un programma.
Iterator	Formalizza come spostarci attraverso una lista di dati all'interno di una classe.
Mediator	Indica come semplificare la comunicazione tra classi utilizzando una classe esterna che nasconda ad ognuna i dettaglia dell'altra.
Memento	Cattura lo stato di un oggetto affinchè possa essere ripristinato in seguito.
Observer	Definisce il modello mediante il quale un numero arbitrario di classi può ricevere notifiche relative ad un cambiamento.
State	Fornisce un contenitore per le istanze dei dati membro di una classe.
Strategy	Incapsula un algoritmo all'interno di una classe.
Template Method	Fornisce la definizione astratta di un algoritmo.
Visitor	Aggiunge funzionalità ad una classe.

26.2 Chain of responsibility

Questo pattern consente ad un numero arbitrario di classi di tentare di gestire una richiesta dal client, facendo si che nessuna di queste conosca le responsabilità delle altre. Il meccanismo si ferma non appena una delle classi della catena è in grado di gestire la richiesta.
In altre parole, la richiesta di un client si propaga all'interno di una catena di oggetti fino a che uno di loro decide di poter assolvere alle necessità del client (*Figura 192*).

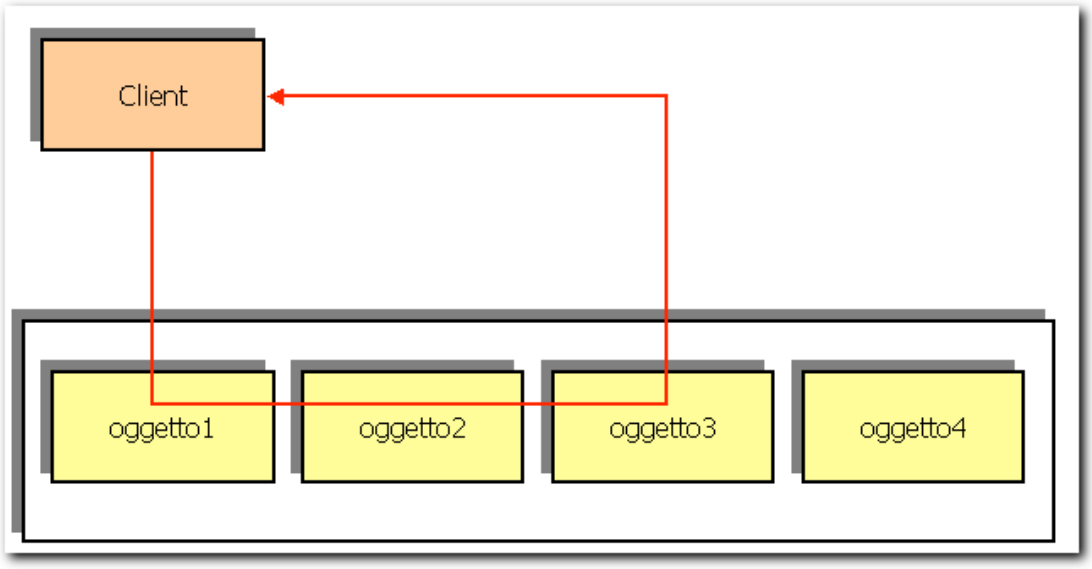


Figura 192: Propagazione della richiesta in una catena di oggetti

Il problema:

- *Più di un oggetto potrebbe assolvere ad una richiesta, ma il gestore non è conosciuto a priori. Un gestore può decidere autonomamente di rispondere al client.*
- *Vogliamo distribuire una richiesta ad uno o più oggetti, senza esplicitare quale dovrà farsene carico;*
- *L'insieme di oggetti che può farsi carico di una richiesta deve essere aggiornato dinamicamente.*

Il disegno:

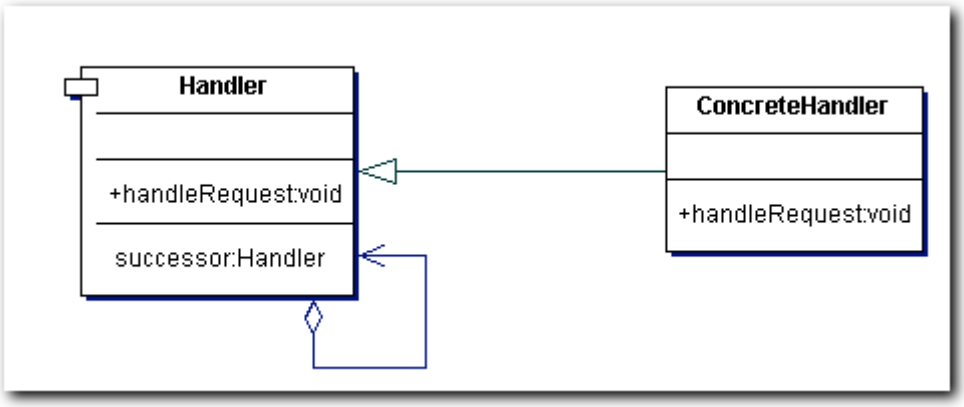


Figura 193: Pattern Chain of Responsibility

Come funziona:

I partecipanti a questo pattern sono solamente due: un oggetto *Handler* ed un tipo *ConcreteHandler*. L'oggetto *Handler*, implementa la gestione della catena di oggetti attraverso la quale muoverci e rappresenta il punto di partenza per la propagazione della richiesta.
Gli oggetti di tipo *ConcreateHandler* sono invece quelli che gestiscono, se possibile, la richiesta del client. Nel caso in cui la richiesta non possa essere presa in carico, continuano la propagazione al proprio successore all'interno della catena.

Il codice:

Nalla *figura 194* è riportato il diagramma delle classi partecipanti al codice di esempio.

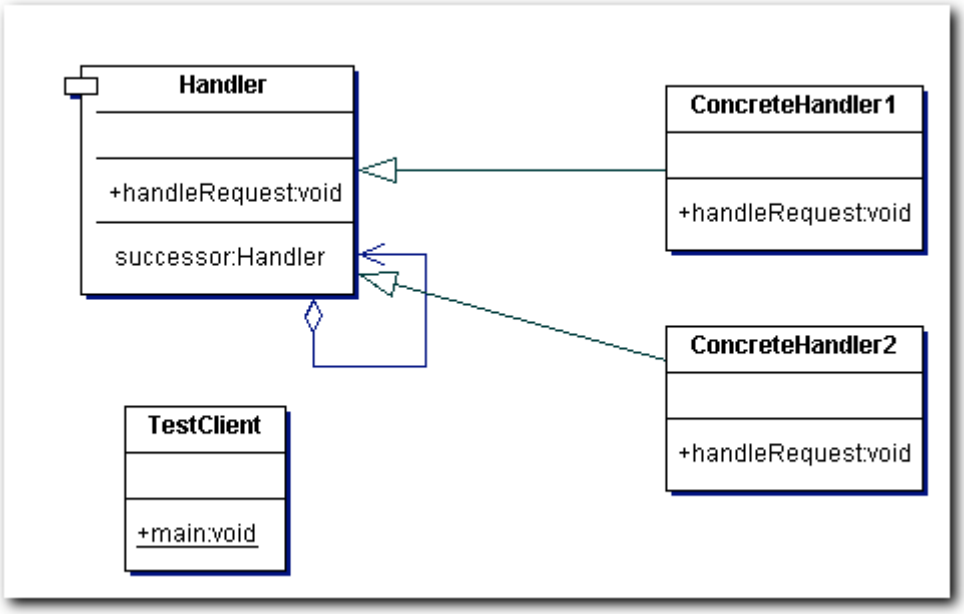


Figura 194: Chain of Responsibility: un esempio

```
package src.pattern.chain;
/**
 * HANDLER
 */
public class Handler {
	public void handleRequest(){
		if (successor != null) {
			successor.handleRequest();
		}
	}

	public void setSuccessor(Handler successor){
		this.successor = successor;
	}

	public Handler getSuccessor(){
```

```
                return successor;
        }

        private Handler successor;
}
```

```
package src.pattern.chain;
/**
 * CONCRETE HANDLER
 */
public class ConcreteHandler1 extends Handler {
        public void handleRequest()
        {
                if( /* Possiamo gestire la richiesta == */ true ){
                        System.out.println("ConcreteHandler1 puo gestire la riuchiesta");
                }
                else{
                        if(getSuccessor()!=null)
                                getSuccessor().handleRequest();
                }
        }
}
```

```
package src.pattern.chain;
/**
 * CONCRETE HANDLER
 */
public class ConcreteHandler2 extends Handler {
        public void handleRequest()
        {
                if( /* Possiamo gestire la richiesta == */ true ){
                        System.out.println("ConcreteHandler2 puo gestire la riuchiesta");
                }
                else{
                        if(getSuccessor()!=null)
                                getSuccessor().handleRequest();
                }
        }
}
```

Il client di test:

```
package src.pattern.chain;
/**
 * CLIENT DI TEST
 */
public class TestClient {
        public static void main( String[] arg ){
                try{
                        Handler handler1 = new ConcreteHandler1();
                        Handler handler2 = new ConcreteHandler1();
                        Handler handler3 = new ConcreteHandler2();
                        Handler handler4 = new ConcreteHandler2();

                        handler1.setSuccessor( handler2 );
                        handler2.setSuccessor( handler3 );
                        handler3.setSuccessor( handler4 );
```

```
                        handler1.handleRequest();
                }
                catch( Exception e ){
                        e.printStackTrace();
                }
        }
}
```

26.3 Command

Il pattern precedente propaga la richiesta di un client all'interno di una catena di oggetti. Il command pattern, rigira la richiesta ad un solo oggetto fornendo un interfaccia semplice e conosciuta al client in questione. Di fatto, se la nostra applicazione utilizza un comando la cui interfaccia non è adatta al client, ed il comando non può essere modicato, mediante questo pattern possiamo riadattarla tante volte quanti sono i client utilizzati dall'applicazione.

Il problema:

- *Dobbiamo aggiungere nuove interfacce ad un comando;*
- *Non vogliamo o, non possiamo, modificarne il codice.*

Il disegno:

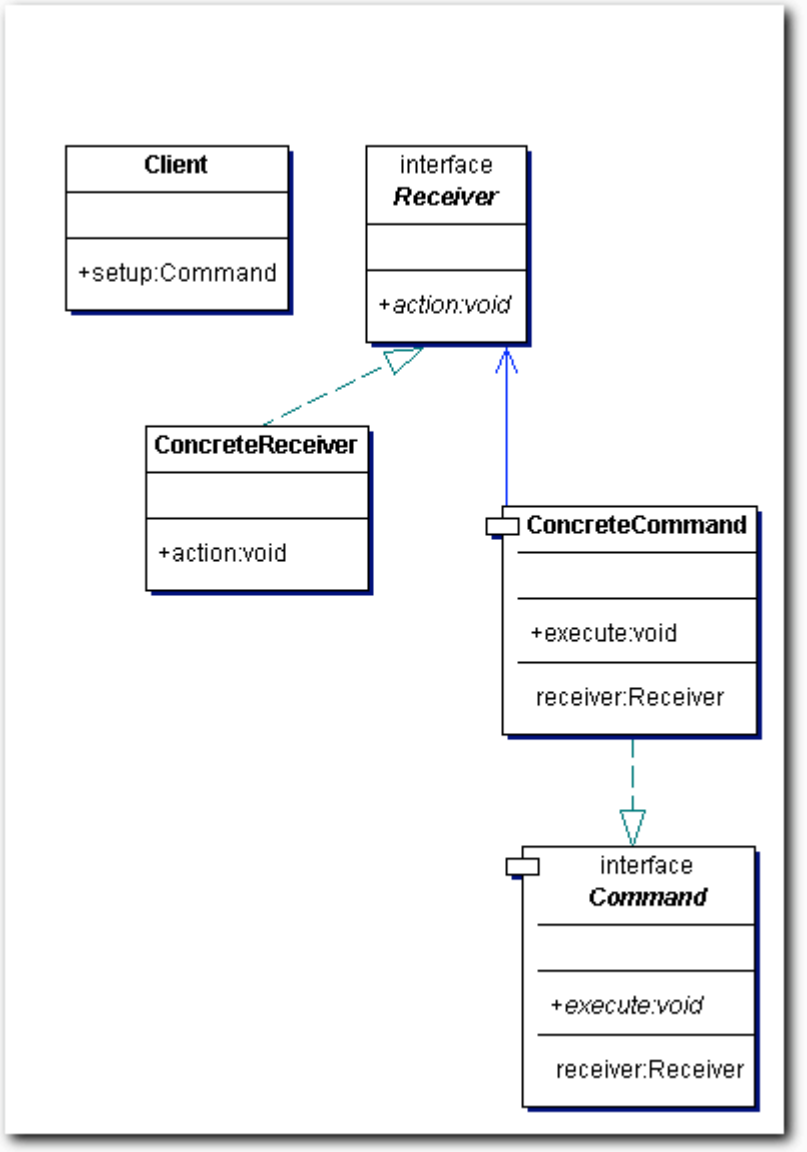


Figura 195: Command Pattern

Come funziona:

Per realizzare il suo scopo, il pattern disaccoppia la classe *Client* dalla classe *Receiver*: *Client*, rappresenta l'oggetto che invoca un comando, *Receiver* quello che cattura la richiesta ed esegue il comando richiesto da *Client*.

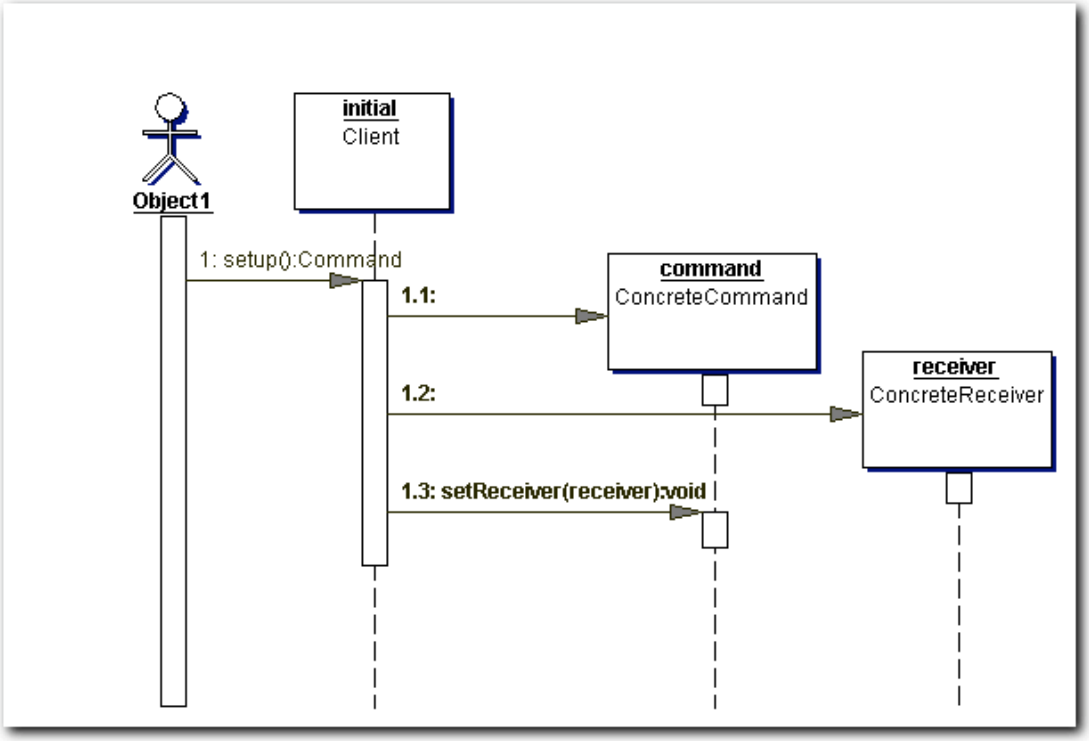


Figura 196: Associazione Command-Receiver

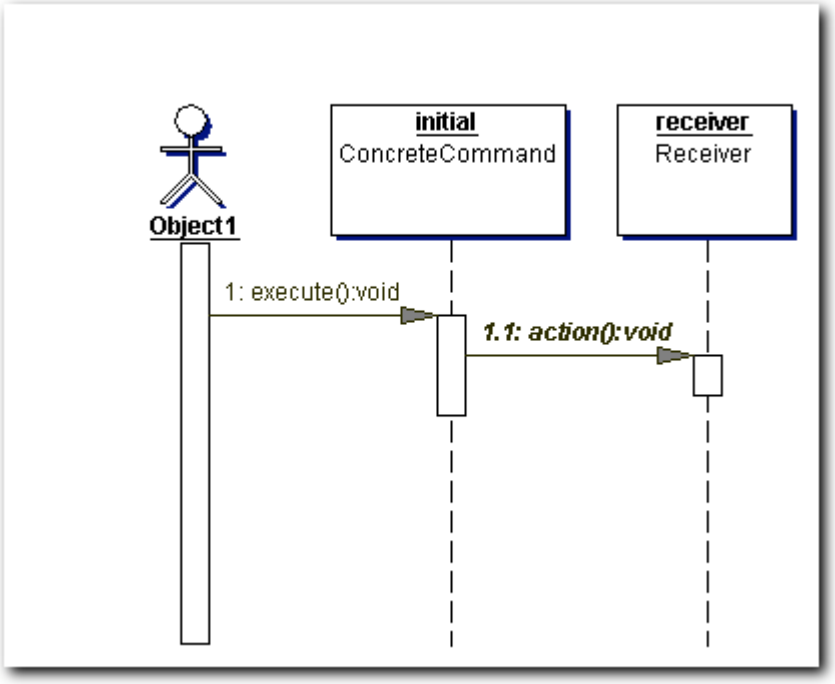


Figura 197: Esecuzione del comando

L'interfaccia *Command* rappresenta il comando da eseguire dal punto di vista del Client. Questa interfaccia contiene i metodi necessari ad associare ad un oggetto di tipo *Command* un oggetto di tipo *Receiver* come mostrato nel

prossimo sequenze diagram, nonché i metodi necessari ad eseguire il comando.
Quando un client esegue il metodo *execute()* di Command, questo utilizza Receiver per eseguire il comando ed eventualmente, ritornare (*Figura 197*).

Il codice:

```
package src.pattern.command;
/**
 * CLIENT
 * Crea un oggetto ConcreteCommand e gli associa un receiver.
 */
public class Client
{
	public Command setup()
	{
		Command command = new ConcreteCommand();
		Receiver receiver = new ConcreteReceiver();
		command.setReceiver( receiver );
		return command;
	}
}
```

```
package src.pattern.command;
/**
 * COMMAND INTERFACE
 */
public interface Command
{
	void setReceiver( Receiver receiver );
	Receiver getReceiver();
	void execute();
}
```

```
package src.pattern.command;
/**
 * RECEIVER
 */
public interface Receiver
{
	void action();
}
```

```
package src.pattern.command;
/**
 * CONCRETE COMMAND
 */
public class ConcreteCommand implements Command
{
	private Receiver receiver;

	public void setReceiver( Receiver receiver )
	{
		this.receiver = receiver;
	}
```

```
	public Receiver getReceiver()
	{
		return receiver;
	}

	public void execute()
	{
		receiver.action();
	}
}
```

```
package src.pattern.command;
/**
 * CONCRETE RECEIVER
 */
public class ConcreteReceiver implements Receiver
{
	public void action()
	{
	}
}
```

Infine, il client di test:

```
package src.pattern.command;
/**
 * TEST CLIENT
 */
public class Test
{
	public static void main( String[] arg )
	{
		try
		{
			Client client = new Client();
			Command command = client.setup();
			command.execute();
		}
		catch( Exception e )
		{
			e.printStackTrace();
		}
	}
}
```

26.4 Interpreter

Scrivendo una applicazione, capita spesso cche gli oggetti debbano fare uso di grammatiche semplici per realizzare determinati compiti, come ad esempio il riconoscimento di determinati pattern all'interno di una stringa. In questo caso, invece di realizzare algoritmi specializzati che ricercano sottostringhe all'interno di una stringa, l'algoritmo di ricerca potrebbe interpretare una

espressione regolare che risolva il problema. Semplicemente, questo pattern descrive come definire una grammatica semplice e come interpretarne le regole sintattiche e semantiche.

Il problema:

- *Dobbiamo interpretare linguaggi semplici rappresentati mediante espressioni regolari.*

Il disegno:

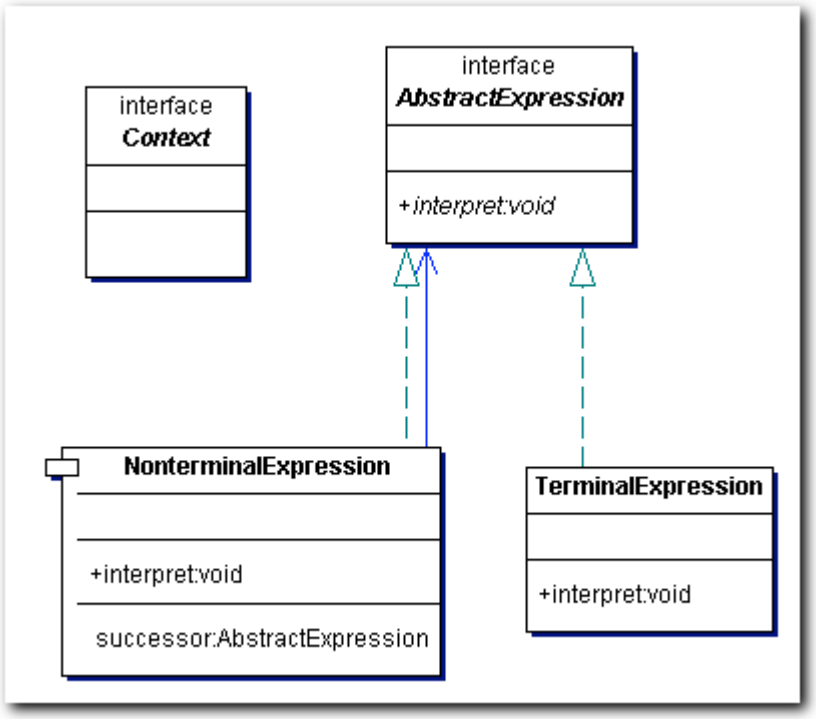


Figura 198: Pattern Interpreter

Come funziona:

Per risolvere il problema, questo pattern utilizza una classe per rappresentare ogni regola grammaticale. Queste classi sono definite a partire dalla classe astratta *AbstractExpression* e rappresentano:

1. *espressioni terminali (TerminalExpression): le espressioni associate con i simboli terminali della grammatica;*
2. *espressioni non (NonTerminalExpression): regole della grammatica.*

La classe *NonTerminalExpression*, a sua volta può rappresentare:

1. *Espressioni di controllo di flusso (AlternationExpression);*
2. *Espressioni di ripetizione (RepetitionExpressio);*
3. *Espressioni semplici di flusso (SequenceExpression).*

Le espressioni regolari definite da questa grammatica sono rappresentate mediante un albero sintattico (*Figura 199*) fatto di oggetti *TerminalExpression* e *NonTerminalExpression*, il cui metodo *interpret*, può essere utilizzato per interpretare l'espressione realizzata da ogni oggetto.
Un interprete prende come attributo un oggetto di tipo *Context*, contenente i parametri di input necessari ad eseguire l'espressione regolare.
Infine, gli oggetti di tipo *NonTerminalExpression* definiscono il metodo *successor* che consente di impostare la prossima espressione della grammatica regolare all'interno dell'albero sintattico.

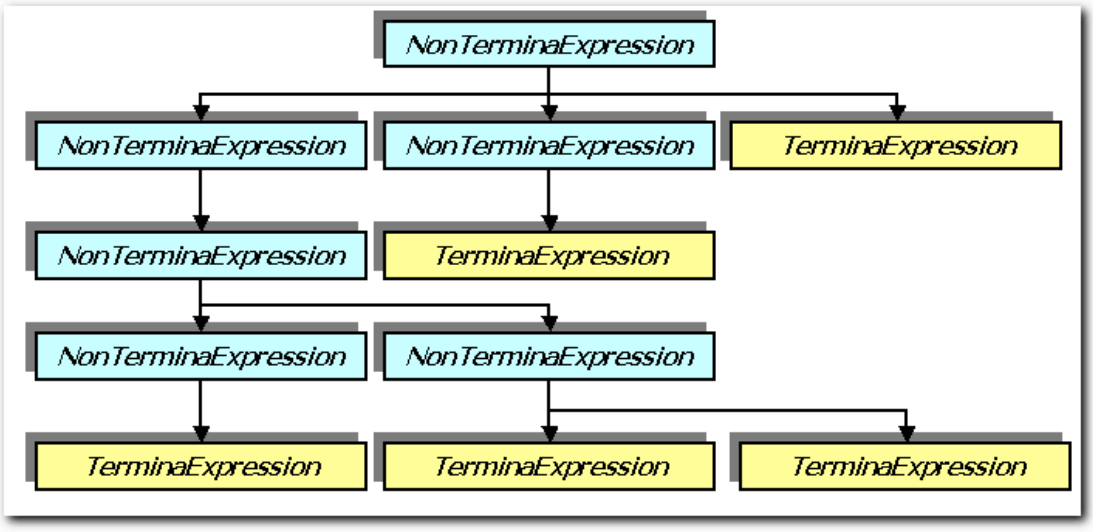


Figura 199: Albero sintattico di una espressione regolare

Il codice:

```
package src.pattern.interpreter;
/**
 * ABSTRACT EXPRESSION
*/
public interface AbstractExpression
{
        void interpret( Context context );
}
```

```
package src.pattern.interpreter;
/**
 * CONTEXT
 */
public interface Context
{
```

```
}
```

```
package src.pattern.interpreter;
/**
 * TERMINAL EXPRESSION
 */
public class TerminalExpression implements AbstractExpression
{
	public void interpret( Context context )
	{
	}
}
```

```
package src.pattern.interpreter;
/**
 * NON TERMINAL EXPRESSION
 */
public class NonterminalExpression implements AbstractExpression
{
	private AbstractExpression successor;

	public void setSuccessor( AbstractExpression successor )
	{
		this.successor = successor;
	}

	public AbstractExpression getSuccessor()
	{
		return successor;
	}

	public void interpret( Context context )
	{
	}
}
```

Infine, il client di test costruisce l'albero sintattico schematizzato in *figura 189*, ed esegue l'interprete.

```
package src.pattern.interpreter;
/**
* TEST CLIENT
*/
public class Test
{
	public static void main( String[] arg )
	{
		NonterminalExpression rule1 = new NonterminalExpression();
		NonterminalExpression rule2 = new NonterminalExpression();
		NonterminalExpression rule3 = new NonterminalExpression();
		TerminalExpression rule4 = new TerminalExpression();

		rule1.setSuccessor( rule2 );
		rule2.setSuccessor( rule3 );
		rule3.setSuccessor( rule4 );

		rule1.interpret( new Context() {} );
```

```
        }
}
```

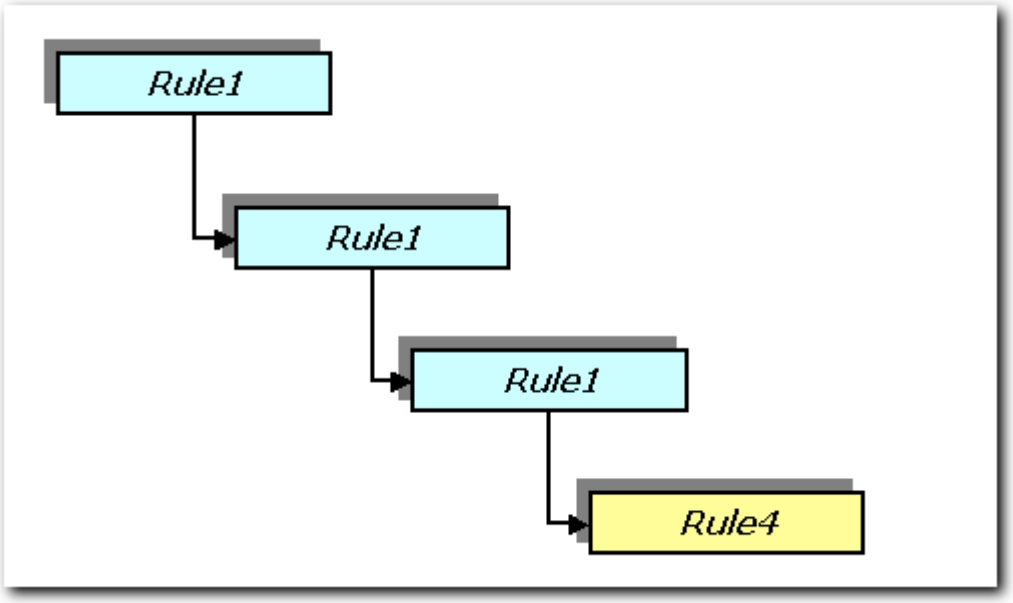


Figura 200: Albero sintattico costruito dall'applicazione Test

26.5 Iterator

Questo pattern è uno dei più semplici e dei più usati e descrive come accedere agli elementi di un oggetto composto, sequenzialmente, senza che l'oggetto mostri la sua struttura interna. Iterator consente, inoltre, di navigare all'interno della lista di oggetti (*figura 201*) in modi differenti, a seconda delle necessità del programmatore.

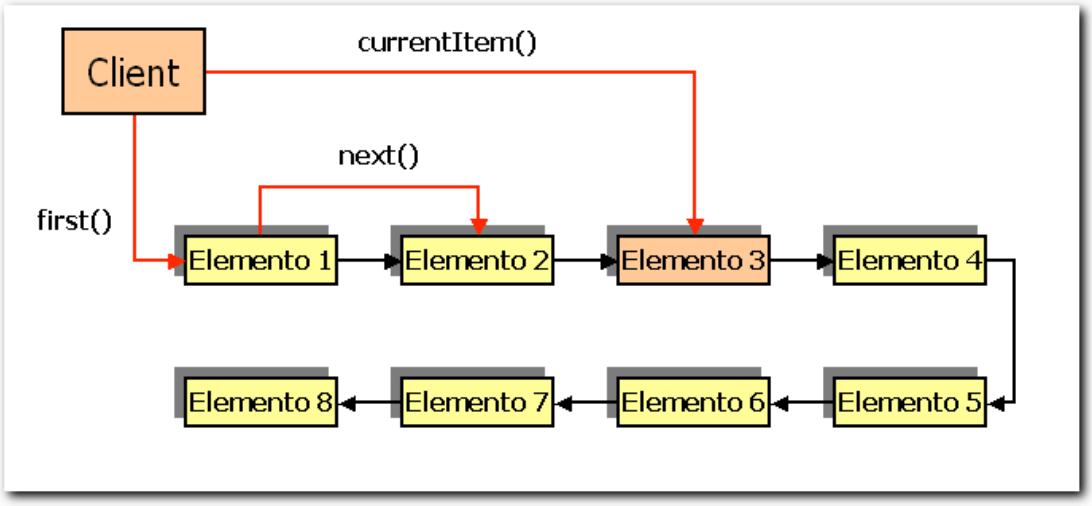


Figura 201: Liste di oggetti

Il problema:

- *Vogliamo creare liste di oggetti, senza che il client ne conosca i dettagli dell'implementazione.*

Il disegno:

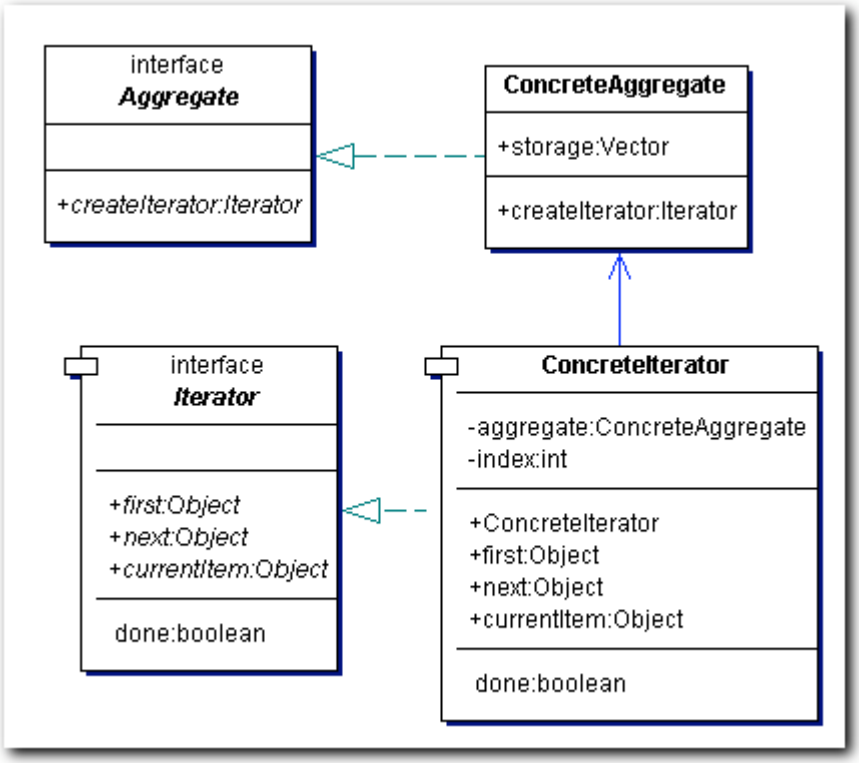


Figura 202: Pattern Iterator

Come funziona:

L'idea del pattern, è quella di decentrare la responsabilità dell'attraversamento della lista ad un oggetto di tipo *Iterator*, definito a partire dall'interfaccia omonima. L'interfaccia *Iterator* rappresenta l'aggregato di oggetti dal punto di vista del client, e contiene il prototipo dei metodi necessari a navigare all'interno della lista:

- *Il metodo first(), ritorna il primo oggetto della lista;*
- *Il metodo next(), ritorna il prossimo oggetto della lista;*
- *Il metodo currentItem(), ritorna l'oggetto corrente.*

L'oggetto *ConcreteIterator* rappresenta una possibile implementazione di *Iterator*. Specializzando questa classe, possiamo creare tanti oggetti *Iterator*, quanti sono i modi con cui vogliamo accedere alla lista.

Gli altri partecipanti al pattern sono l'interfaccia *Aggregate*,e la sua implementazione *ConcreteAggregate*. *Aggregate* definisce il modello utilizzato da *ConcreteAggregate* per realizzare l'aggregato di oggetto associato all'oggetto *ConcreteIterator*.

Il codice:

```
package src.pattern.iterator;
/**
 * ITERATOR
 */
public interface Iterator
{
	Object first() throws IndexOutOfBoundsException;
	Object next() throws IndexOutOfBoundsException;
	boolean isDone();
	Object currentItem() throws IndexOutOfBoundsException;
}
```

```
package src.pattern.iterator;
/**
 * CONCRETE ITERATOR
 */
public class ConcreteIterator implements Iterator
{
	private ConcreteAggregate aggregate;
	private int index = 0;

	public ConcreteIterator( ConcreteAggregate aggregate )
	{
		this.aggregate = aggregate;
	}

	public Object first()
			throws IndexOutOfBoundsException
	{
		Object object = null;
		if( !aggregate.storage.isEmpty() )
		{
			index = 0;
			object = aggregate.storage.firstElement();
		}
		else
		{
			throw new IndexOutOfBoundsException();
		}
		return object;
	}

	public Object next()
			throws IndexOutOfBoundsException
	{
		Object object = null;
		if( index + 1 < aggregate.storage.size() )
		{
			index += 1;
```

```
                object = aggregate.storage.elementAt(index);
            }
            else
            {
                throw new IndexOutOfBoundsException();
            }
            return object;
        }

        public boolean isDone()
        {
            boolean result = false;
            if( aggregate.storage.isEmpty() ||
                index == aggregate.storage.size() - 1 )
            {
                result = true;
            }
            return result;
        }

        public Object currentItem()
                throws IndexOutOfBoundsException
        {
            Object object = null;
            if( index < aggregate.storage.size() )
            {
                object = aggregate.storage.elementAt( index );
            }
            else
            {
                throw new IndexOutOfBoundsException();
            }
            return object;
        }
}
```

```
package src.pattern.iterator;
/**
 * AGGREGATE
 */
public interface Aggregate
{
        Iterator createIterator();
}
```

```
package src.pattern.iterator;
import java.util.Vector;
/**
 * CONCRETE AGRREGATE
 */
public class ConcreteAggregate implements Aggregate
{
        public Vector storage = new Vector();

        public Iterator createIterator()
        {
            return new ConcreteIterator( this );
        }
```

```
}
```

Infine, il client di test:

```
package src.pattern.iterator;
/**
 * TEST CLIENT
 */
public class Test
{
	public static void main( String arg[] )
	{
		try
		{
			Aggregate aggregate = new ConcreteAggregate();
			Iterator iterator = aggregate.createIterator();
		}
		catch( Exception e )
		{
			e.printStackTrace();
		}
	}
}
```

26.6 Mediator

La programmazione orientata ad oggetti, incoraggia la distribuzione dei compiti tra oggetti. Questa caratteristica da spesso forma a strutture complesse con molte connessioni tra oggetti. Nel caso peggiore (*Figura 203*), ogni oggetto è connesso con tutti gli altri.

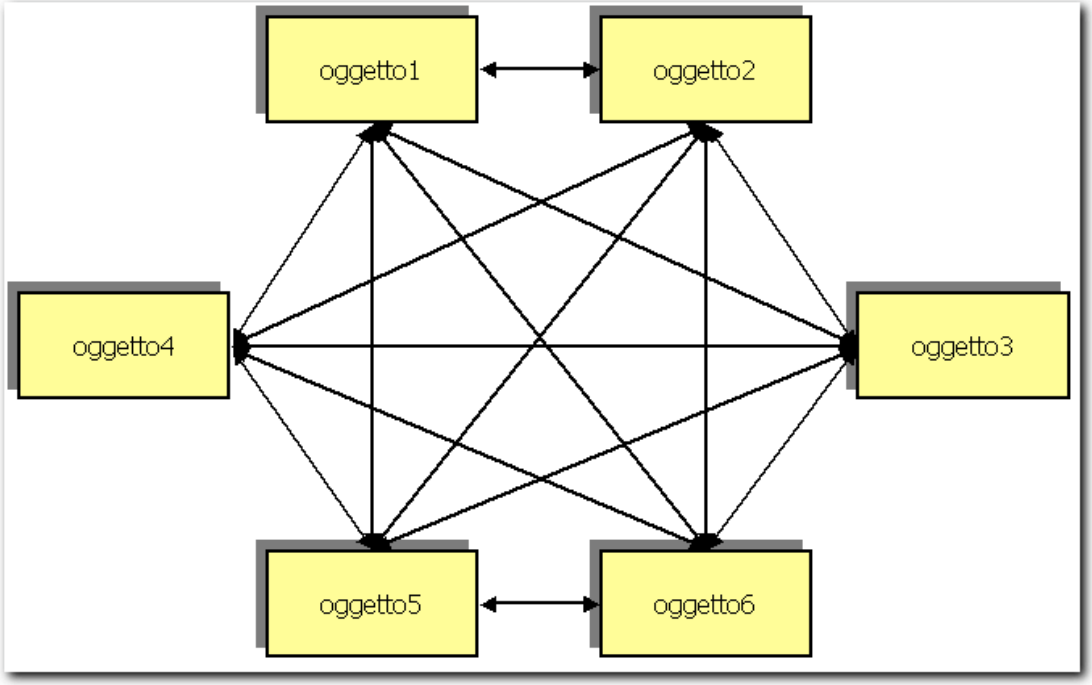


Figura 203: Strutture complesse con molte connessioni tra oggetti

Di fatto, se la scomposizione di un sistema in oggetti aumenta la possibilità di riutilizzare il codice, d'altra parte, l'uso di troppe connessioni tende a ridurla. Utilizzare molte connessioni significa infatti creare dipendenze tali tra oggetti che, l'applicazione si comporterà come un sistema monolitico.
Scopo del pattern Mediator, è quello di disaccoppiare le logiche di interazione dagli oggetti, evitando che debbano fare riferimento esplicito l'uno all'altro. Un Mediator è di fatto un oggetto che definisce come gruppi di oggetti debbano interagire tra loro.

Il problema:

- *Dobbiamo disaccoppiare le interazioni tra oggetti, dagli oggetti stessi per favorire la riutilizzabilità del codice;*
- *Vogliamo far interagire alcuni oggetti in maniera dinamica, definendo volta per volta le relazioni tra loro;*
- *Vogliamo che alcuni oggetti rimangano indipendenti l'uno dall'altro.*

Il disegno:

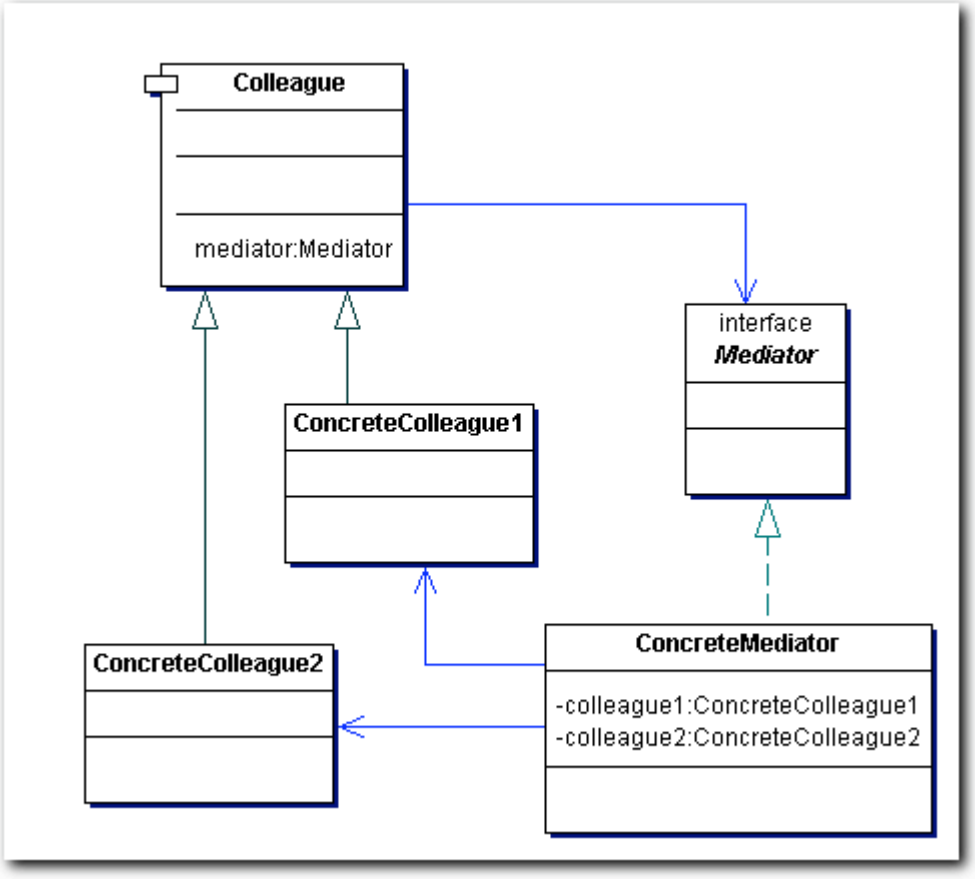


Figura 204: Pattern Mediator

Come funziona:

Un oggetto di tipo *Mediator*, è responsabile del controllo e del coordinamento delle interazioni tra un gruppo di oggetti. Il gruppo di oggetti è rappresentato da *Colleague*. Ogni classe di tipo *Colleague* conosce il suo *Mediator* e comunica con esso ogni volta che oggetti della collezione debbono interagire con oggetti di altre collezioni.
Le classi *ConcreteMediator* e *ConcreteColleague*, sono rispettivamente una possibile implementazione ed una specializzazione di *Colleague* e *Mediator*.

Il codice:

```
package src.pattern.mediator;
/**
 * MEDIATOR
 */
public interface Mediator
{
	//definisce i prototipi dei metodi per
	//interagire con gli oggetti di una collezione
}
```

```
package src.pattern.mediator;
/**
 * CONCRETE MEDIATOR
 */
public class ConcreteMediator implements Mediator
{
	private ConcreteColleague1 colleague1 = new ConcreteColleague1();
	private ConcreteColleague2 colleague2 = new ConcreteColleague2();
}
```

```
package src.pattern.mediator;
/**
 * COLLEAGUE
 */
public class Colleague
{
	private Mediator mediator;

	public Mediator getMediator()
	{
		return mediator;
	}

	public void setMediator( Mediator mediator )
	{
		this.mediator = mediator;
	}
}
```

```
package src.pattern.mediator;
/**
 * CONCRETE COLLEAGUE
*/
public class ConcreteColleague1 extends Colleague
{
}
```

```
package src.pattern.mediator;
/**
 * CONCRETE COLLEAGUE
*/
public class ConcreteColleague2 extends Colleague
{
}
```

26.7 Strategy

Il pattern strategy definisce una famiglia di algoritmi correlati, incapsulati all'interno di una classe driver chiamata *Context*. Scopo di strategy, è quello di rendere intercambiabili gruppi di algoritmi, indipendentemente dal client che li utilizza. Di fatto, possiamo utilizzare questo pattern se la nostra applicazione necessita di fare la stessa cosa in modi differenti.

Il problema:

- *Dobbiamo sviluppare una applicazione che deve realizzare la stessa funzionalità in modo differenti;*
- *Il client non deve conoscere i dettagli dell'implementazione e le diverse versioni di uno stesso algoritmo;*
- *Dobbiamo poter modificare dinamicamente gli algoritmi;*
- *Non tutte le classi hanno necessità di supportare tutte le versioni di un algoritmo.*

Il disegno:

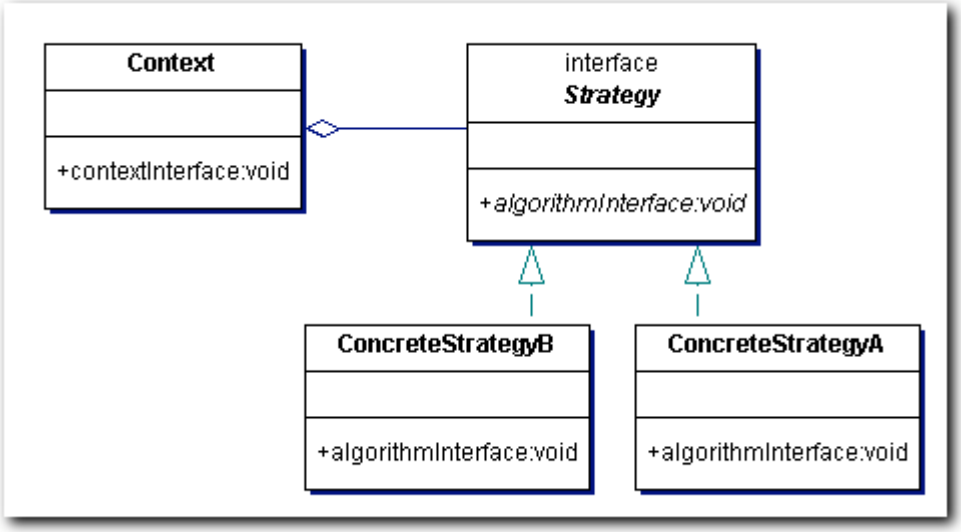


Figura 205: Il pattern strategy: prima versione

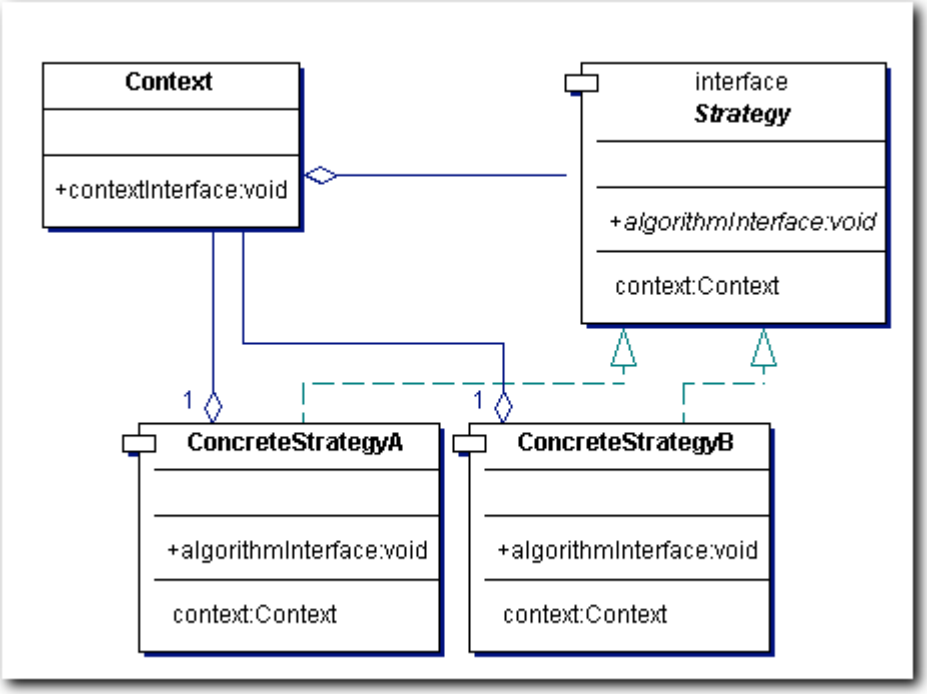


Figura 206: Pattern Strategy: seconda versione

Come funziona:

I partecipanti al pattern sono tre: la classe *Context*, l'interfaccia *Strategy*, gli oggetti *ConcreteStrategy*.
L'interfaccia *Strategy*, definisce i prototipi dei metodi comuni a tutti gli oggetti *ConcreteStrategy*. Ogni oggetto di tipo *ConcreteStrategy*, ottenuto per ereditarietà da *Strategy*, rappresenta un algoritmo appartenente alla famiglia.
L'oggetto *Context* incapsula gli algoritmi e rappresenta l'interfaccia verso il Client. Il comportamento di Context può variare a seconda del problema che dobbiamo affrontare:

1. *Il client tramite Context richiede che venga utilizzato uno specifico algoritmo;*
2. *Context decide quale algoritmo utilizzare in base al valore degli attributi ricevuti dal client.*

Il codice:

Come schematizzato nelle due figure 204 e 206, esistono due differenti modi per implementare il pattern Strategy.
Nella prima versione del pattern, la classe *Context* rigira gli attributi ricevuti dal client all'oggetto *Strategy* che dovrà processarli, come mostrato nel prossimo esempio:

```
package src.pattern.strategy;
import java.util.Vector;
/**
 * CONTEXT
 */
public class Context {
	public Context() {
		Strategy strategyA = new ConcreteStrategyA();
		Strategy strategyB = new ConcreteStrategyB();
		lnkStrategy = new Vector();
		lnkStrategy.add(strategyA);
		lnkStrategy.add(strategyB);
	}
	public void contextInterface() {
		Strategy strategy = (Strategy)lnkStrategy.elementAt(0);
		strategy.algorithmInterface();
	}

	private Vector lnkStrategy;
}
```

```
package src.pattern.strategy;
/**
 * STRATEGY
 */
public interface Strategy {
	void algorithmInterface();
}
```

```
package src.pattern.strategy;
/**
 * CONCRETE STRATEGY
 */
public class ConcreteStrategyA implements Strategy {
	public void algorithmInterface(){ }
}
```

```
package src.pattern.strategy;
/**
 * CONCRETE STRATEGY
 */
public class ConcreteStrategyB implements Strategy {
	public void algorithmInterface(){ }
}
```

Se gli algoritmo appartenenti ad una stessa famiglia, necessitano di attributi differenti, utilizzando la prima versione del pattern dovremmo:

1. *Aggiungere all'interfaccia Strategy tanti metodi quante sono le varianti per gli attributi da passare;*
2. *Definire un unico metodo sopportanto il fatto di dover trasmettere ad un algoritmo attributi inutili.*

Per risolvere la questione possiamo utilizzare la seconda versione del pattern secondo la quale *Context* riceve i dati dal client, li memorizza, e li mette a disposizione all'oggetto *Strategy* passando se stesso come attributo.

```
package src.pattern.strategy.secversion;
import java.util.Vector;
/**
 * CONTEXT
 */
public class Context {
	public Context() {
		Strategy strategyA = new ConcreteStrategyA();
		Strategy strategyB = new ConcreteStrategyB();
		lnkStrategy = new Vector();
		lnkStrategy.add(strategyA);
		lnkStrategy.add(strategyB);
	}

	public void contextInterface() {
		Strategy strategy = (Strategy)lnkStrategy.elementAt(0);
		//Context passa se stesso a Strategy
		strategy.setContext(this);
		strategy.algorithmInterface();
	}

   private Vector lnkStrategy;
}
```

```
package src.pattern.strategy;
/**
 * STRATEGY
 */
public interface Strategy {
	void algorithmInterface();
	void setContext(Context ctx);
}
```

```
package src.pattern.strategy;
/**
 * CONCRETE STRATEGY
 */
public class ConcreteStrategyA implements Strategy {
	public void algorithmInterface(){ }
	public void setContext(Context context){
		lnkContext = context;
	}

	private Context lnkContext;
}
```

```
package src.pattern.strategy;
/**
 * CONCRETE STRATEGY
 */
public class ConcreteStrategyB implements Strategy {
	public void algorithmInterface(){ }
```

```
        public void setContext(Context context){
                lnkContext = context;
        }

        private Context lnkContext;
}
```

26.8 State

Questo pattern consente ad una classe di modificare il proprio comportamento, non l'interfaccia, al cambiare del suo stato interno. Immaginiamo ad esempio, di dover sviluppare un oggetto che rappresenta una connessione via TCP/IP. Tale oggetto, durante il suo ciclo di vita, potrebbe trovarsi in uno dei seguenti stati (*Figura 207*):

- *Attesa di connessione;*
- *Connessione aperta;*
- *Ascolto;*
- *Ricezione;*
- *Connessione chiusa;*
- *Trasmissione.*

A seconda dello stato dell'oggetto, la nostra classe si comporterà in maniera differente. Ad esempio, potremmo aprire una connessione solo se l'oggetto si trova nello stato *Attesa di connessione* oppure, potremmo ricevere dati solo se lo stato dell'oggetto è *Connessione aperta* e non *Trasmissione*.

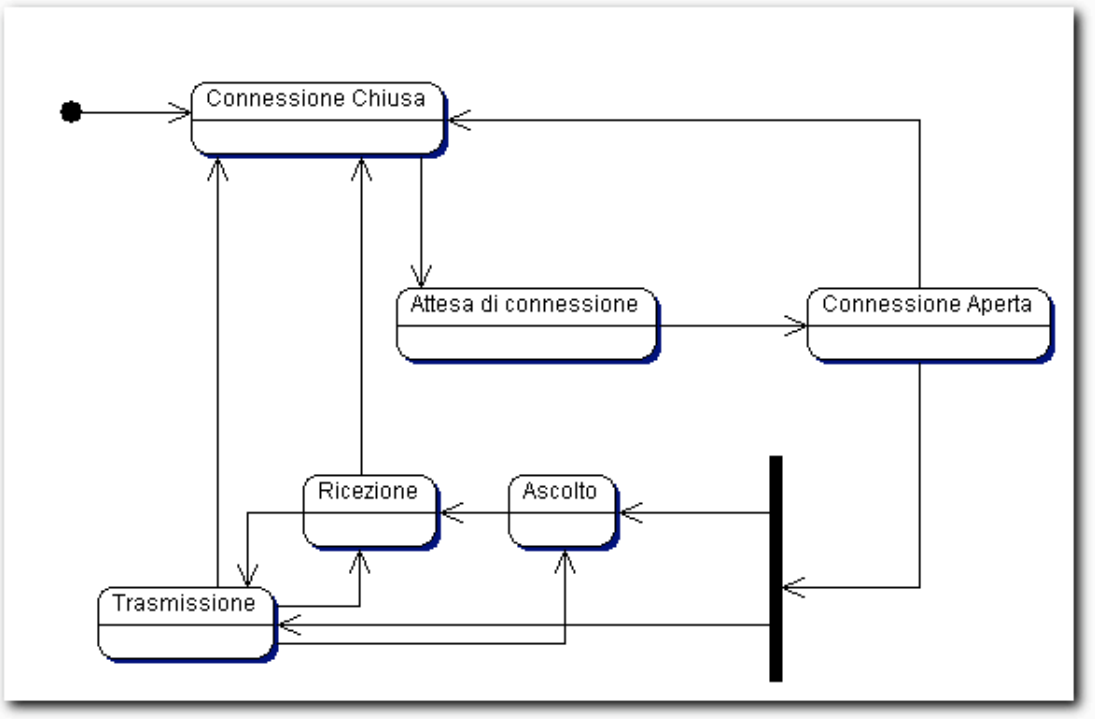


Figura 207: Stati di una connessione TCP/IP

Il problema:

- *Necessitiamo di un oggetto che modifichi il proprio comportamento a seconda del proprio stato interno.*

Il disegno:

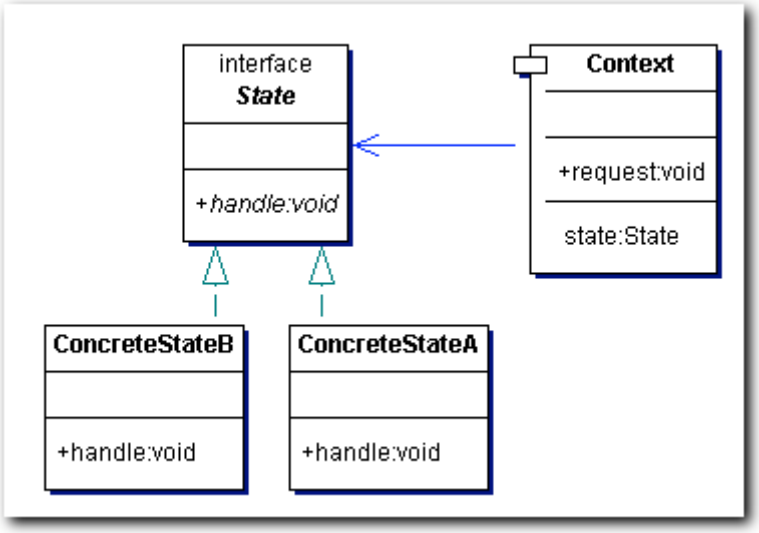


Figura 208: State Pattern

Come funziona:

I partecipanti al pattern sono le classi *State*, *ConcreteState* e *Context*. L'idea del pattern è quello di utilizzare una classe di tipo *State* (*ConcreteState*), per modificare lo stato interno dell'oggetto rappresentato da *Context*.

Il codice:

```
package src.pattern.state;
/**
 * CONTEXT
 */
public class Context{
	private State state;
	public void setState( State state ){
		this.state = state;
	}
	public State getState(){
		return state;
	}
	public void request(){
		state.handle();
	}
```

```
}
```

```
package src.pattern.state;
/**
 * STATE
 */
public interface State{
	void handle();
}
```

```
package src.pattern.state;
/**
 * CONCRETE STATE
 */
public class ConcreteStateA implements State{
	public void handle()	{
	}
}
```

```
package src.pattern.state;
/**
 * CONCRETE STATE
 */
public class ConcreteStateB implements State{
	public void handle()	{
	}
}
```

Infine, il client di test:

```
package src.pattern.state;
/**
 * CLIENT DI TEST
 */
public class Test{
	public static void main( String[] arg ){
		try{
			State state = new ConcreteStateA();
			Context context = new Context();
			context.setState( state );
			context.request();
		}
		catch( Exception e ){
			e.printStackTrace();
		}
	}
}
```

26.9 Template

Template realizza un algoritmo lasciando che i dettagli vengano definiti all'interno di sottoclassi. Questo pattern è considerato uno dei fondamentali, ed ogni programmatore Java lo ha utilizzato inconsapevolmente decine di volte. L'idea alla base di Template è che, le parti di un algoritmo ben definite

possono essere implementate all'interno di una classe base astratta, quelle soggette a varianti all'interno di classi derivate. Al contrario, le parti di un algoritmo condivise da più classi possono essere raggruppate a fattor comune ed incapsulate all'interno di una classe base evitando che vengano ripetute all'interno di tutte le sottoclassi.

Il problema:

- *Applicabile a i contesti in cui alcune classi variano facilmente il proprio comportamento interno;*
- *Dobbiamo realizzare algoritmi che necessitano di molte variazioni;*
- *Alcune classi possono essere implementate a partire da una classe base, da cui ereditare tutte le operazioni in comune.*

Il disegno:

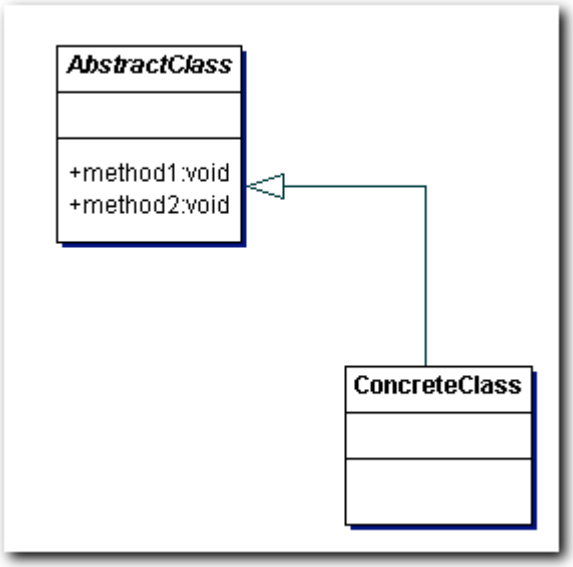


Figura 209: Pattern Template

Come funziona:

Fondamentalmente questo patterm è semplice da rappresentare. Di fatto, esistono solo due partecipanti: la classe base *AbstractClass* e la sottoclasse *ConcreteClass*. La prima rappresenta ciò che di un algoritmo può essere raggruppato a fattor comune, la seconda ne contiene le varie specializzazioni. Ciò che è interessante notare è che un *Template* contiene quattro tipi di metodi differenti, definiti in base all'uso che ne viene fatto dalla sottoclasse:

1. *Concrete Methods (Figure 210,211): tutti I metodi che contengono alcune funzioni di base utilizzabili da tutte le sottoclassi;*

2. *Abstract Methods (Figura 212): tutti i metodi incompleti che vanno quindi implementati all'interno delle sottoclassi;*
3. *Hook Methods (Figure 213): tutti I metodi che contengono alcune funzioni di base ma possono essere ridefiniti all'interno della sottoclasse;*
4. *Template Method (Figure 214,215): tutti i metodi che contengono chiamate a metodi Concrete, Abstract o Hook. Tali metodi non possono essere modificati nella sottoclasse e descrivono un algoritmo senza necessariamente implementarne i dettagli.*

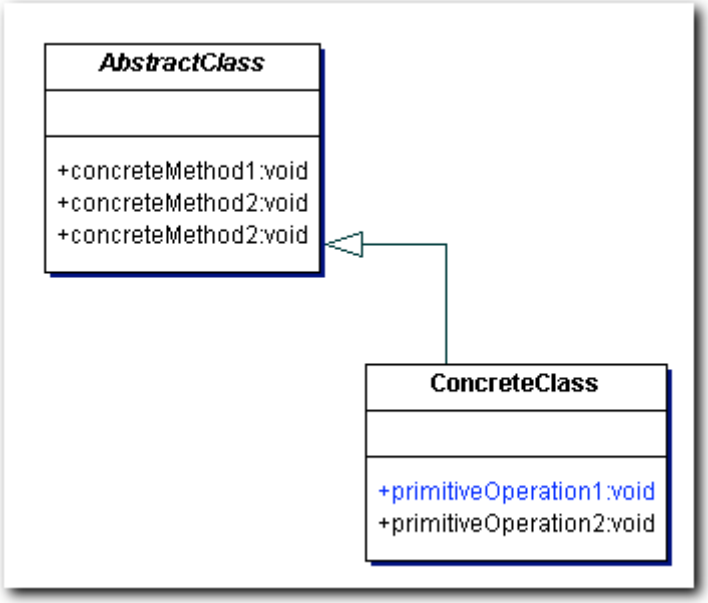


Figura 210: Concrete Methods

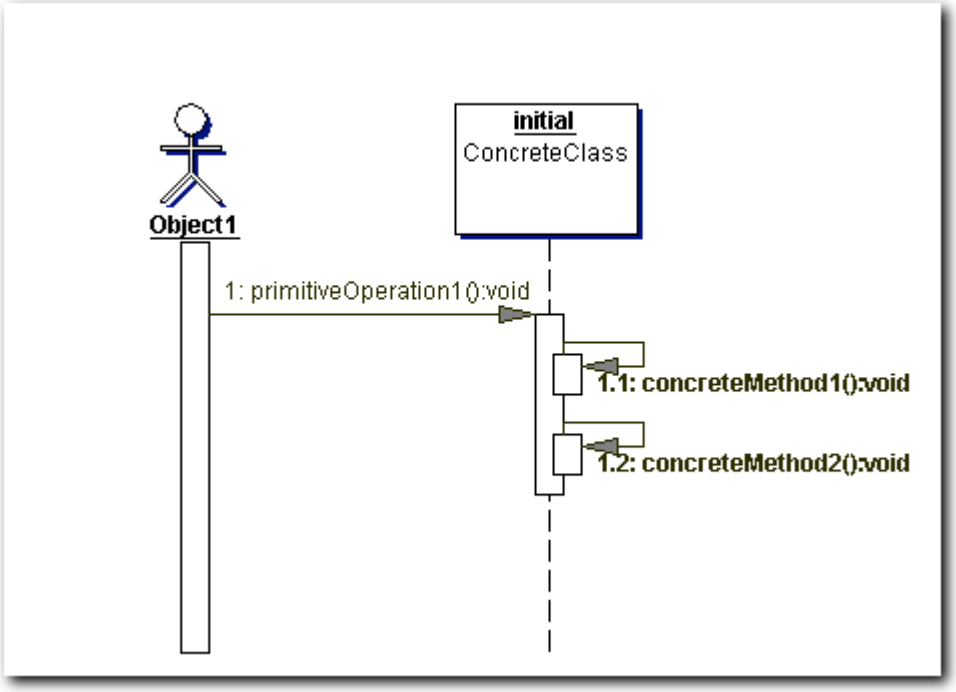


Figura 211: Funzionamento interno di Concrete Method

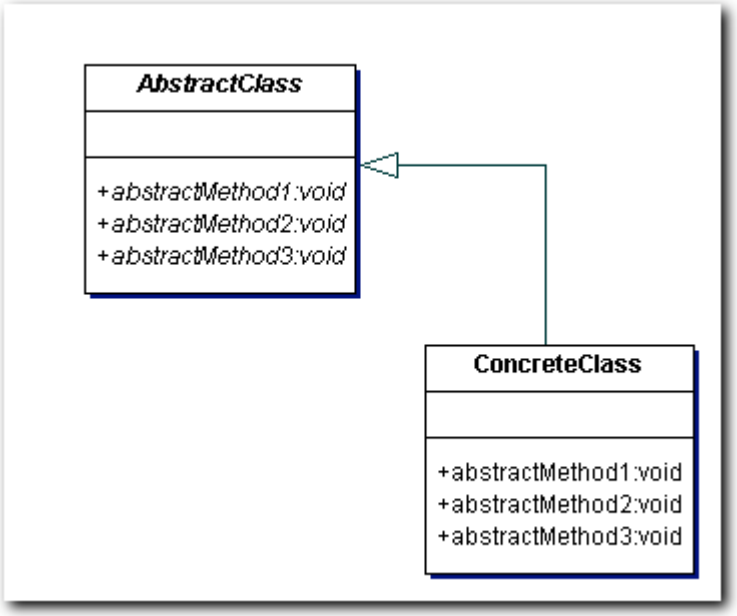


Figura 212: Abstract Method

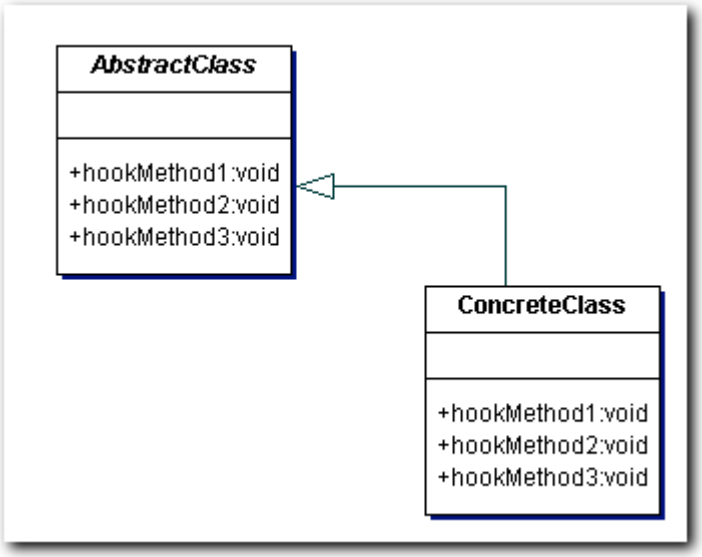


Figura 213: Hook Method

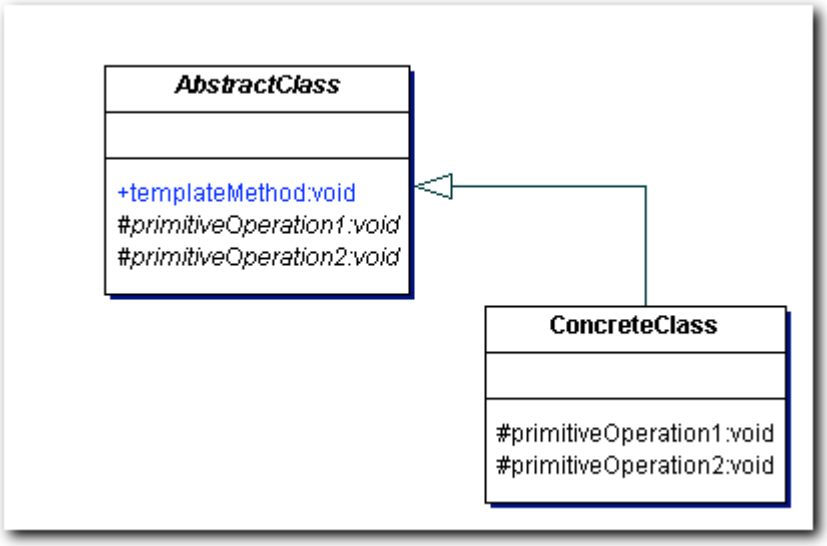


Figura 214: Template Method

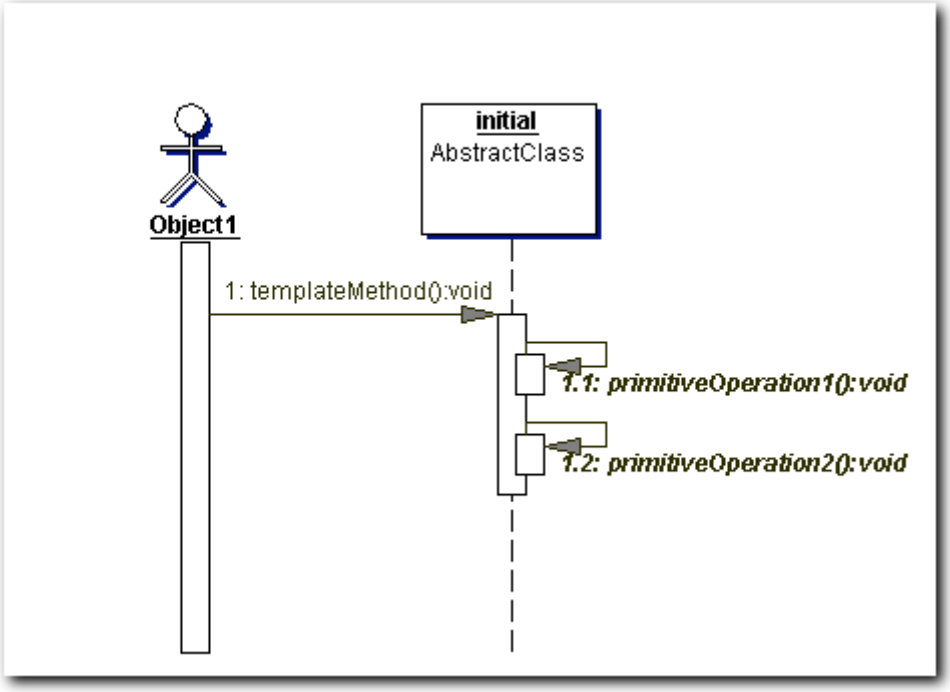


Figura 215: Funzionamento interno di Template Method

Il codice di Concrete Method:

```
package src.pattern.template.concretemethod;
/**
 * ABSTRACT CLASS
 */
public abstract class AbstractClass {
	public void concreteMethod1() {
	}

	public void concreteMethod2() {
	}
}
```

```
package src.pattern.template.concretemethod;
/**
 * CONCRETE CLASS
 */
public class ConcreteClass extends AbstractClass {
	public void primitiveOperation1() {
		concreteMethod1();
		concreteMethod2();
	}

	public void primitiveOperation2() {
	}
}
```

Il codice di Abstract Method:

```
package src.pattern.template.abstractmethod;
/**
 * ABSTRACT CLASS
 */
public abstract class AbstractClass {
	public abstract void abstractMethod1();

	public abstract void abstractMethod2();

	public abstract void abstractMethod3();
}
```

```
package src.pattern.template.abstractmethod;
/**
 * CONCRETE CLASS
 */
public class ConcreteClass extends AbstractClass {
	public void abstractMethod1(){ }

	public void abstractMethod2(){ }

	public void abstractMethod3(){ }
}
```

Il codice di Hook Method:

```
package src.pattern.template.hookmethod;
/**
 * ABSTRACT CLASS
 */
public abstract class AbstractClass {
	public void hookMethod1() {
	}

	public void hookMethod2() {
	}

	public void hookMethod3() {
	}
}
```

```
package src.pattern.template.hookmethod;
/**
 * CONCRETE CLASS
 */
public class ConcreteClass extends AbstractClass {
	public void hookMethod1() {
	}

	public void hookMethod2() {
	}

	public void hookMethod3() {
	}
}
```

Il codice di Template Method:

```
package src.pattern.template.templatemethod;
/**
 * ABSTRACT CLASS
 */
public abstract class AbstractClass
{
	public void templateMethod(){
		primitiveOperation1();
		primitiveOperation2();
	}

	protected abstract void primitiveOperation1();
	protected abstract void primitiveOperation2();
}
```

```
package src.pattern.template.templatemethod;
/**
 * CONCRETE CLASS
 */
public class ConcreteClass extends AbstractClass {
	protected void primitiveOperation1() {
	}

	protected void primitiveOperation2() {
	}
}
```

26.10 Visitor

Il pattern Visitor rappresenta un operazione che deve essere eseguita sugli elementi di una struttura di oggetti, consentendo di definire nuove operazioni senza modificare le classi che definiscono gli elementi su cui operare. Visitor consente inoltre di raggruppare tutte le operazioni correlate definendole all'interno di una unica classe. Quando la struttura dati è condivisa da molte applicazioni, questo meccanismo consente di dotare ogni applicazione, solo delle funzionalità di cui necessità.
Nel caso in cui la struttura di oggetti è molto complessa e dobbiamo aggiungere spesso nuove funzionalità alla nostra applicazione, possiamo utilizzare questo pattern per incapsulare le nuove funzionalità all'interno di classi Visitor. In questo modo, possiamo evitare di dover modificare le definizioni degli oggetti nella struttura ogni volta che dobbiamo aggiungere una nuova funzionalità.

Il problema:

- *Una struttura di oggetti contiene molte classi di oggetti con interfacce differenti tra loro e abbiamo la necessità di operare su tali oggetti che dipendono sulle loro classi concrete;*

- *Dobbiamo eseguire molte operazioni, tra loro distinte e non correlate tra loro, su oggetti appartenenti ad una struttura arbitrariamente complessa senza alterare le loro definizioni di classe.*
- *Dobbiamo definire spesso nuove operazioni da applicare agli oggetti appartenenti ad una struttura arbitrariamente complessa.*

Il disegno:

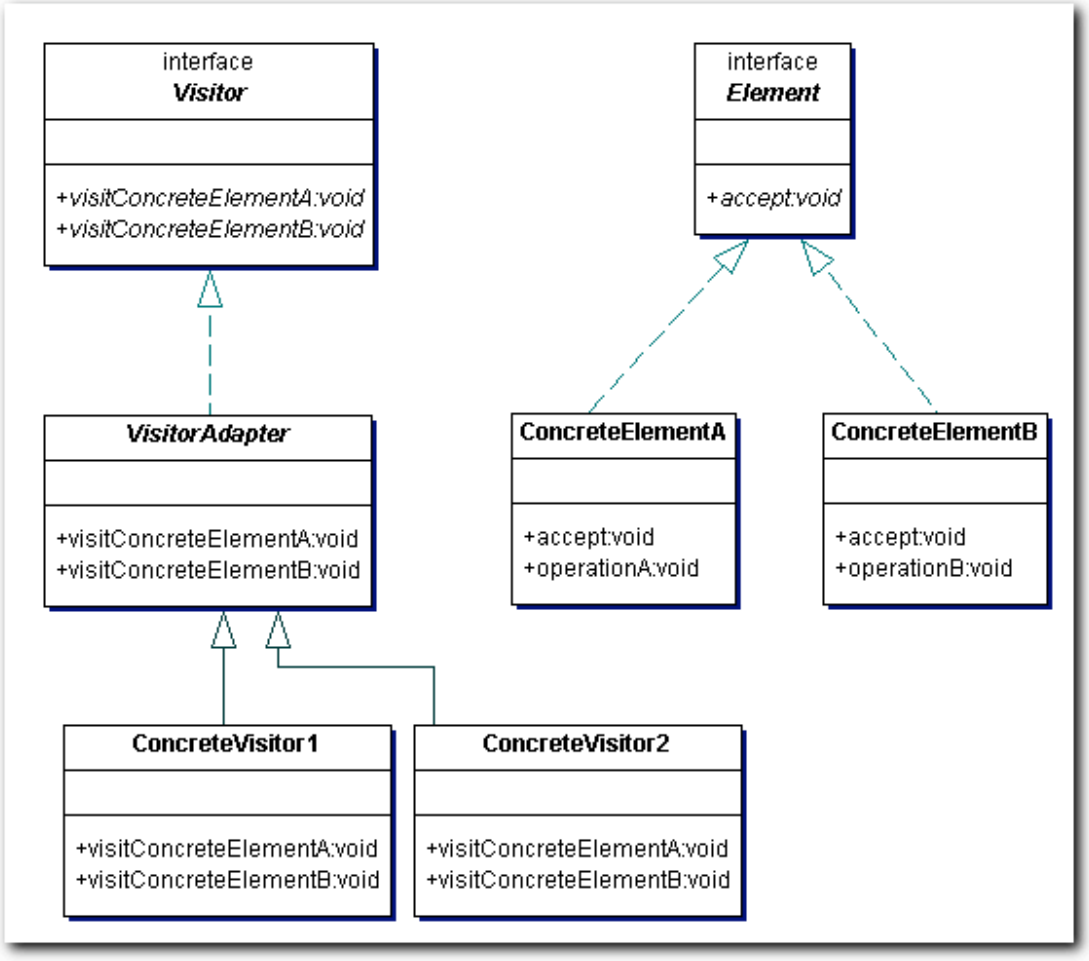


Figura 216: Il pattern Visitor

Come funziona:

L'interfaccia *Visitor* definisce una operazione di tipo "*Visit*" per ogni classe di tipo *Element* appartenente alla struttura di oggetti. Il nome di ogni funzione dichiarata e la lista degli attributi, consente di identificare l'elemento a cui il metodo accede.
Ad esempio, l'interfaccia *Visitor* definita nel class-diagram in *figura 216*, definisce i due metodi:

```
void visitConcreteElementA(Element element);
void visitConcreteElementB(Element element);
```

che consentono di accedere rispettivamente alle classi *ConcreteElementA* e *ConcreteElementB* di tipo *Element*.
Tramite questa interfaccia, è quindi possibile accedere ad ogni elemento appartenente alla struttura di oggetti.
La classe *VisitorAdapter* rappresenta l'implementazione astratta dell'interfaccia *Visitor* e contiene le definizioni dei metodi comuni a tutti gli oggetti *ConcreteVisitor* contenenti le definizioni di tutti i metodi appartenenti alla classe base.
Ogni oggetto *ConcreteVisitor* provvede al contesto relativo all'algoritmo da implementare e memorizza è responsabile di tener traccia del proprio stato interno.
Infine, le classi di tipo *Element*, rappresentano gli oggetti della struttura e definiscono una operazione *accept* che accetta un oggetto visitor come argomento. Il funzionamento del metodo *accept* è schematizzato nel prossimo sequence-diagram.

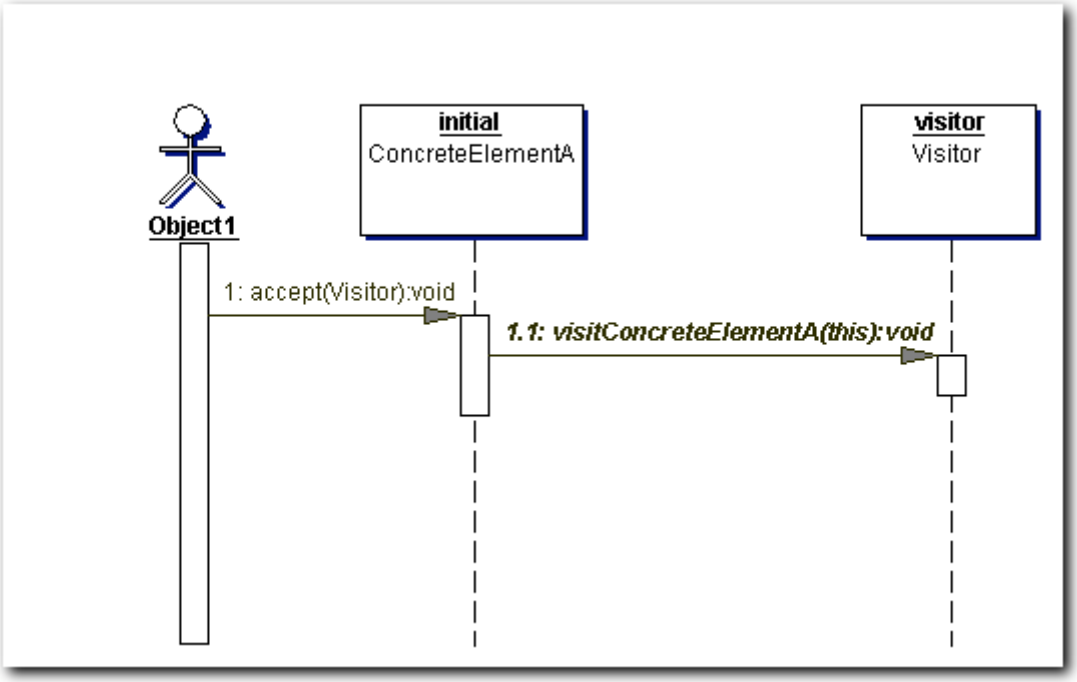


Figura 217: Il metodo accept

Il codice:

```
package src.pattern.visitor;

/**
 * VISITOR
 */
```

```
public interface Visitor {
	void visitConcreteElementA(Element element);

	void visitConcreteElementB(Element element);
}
```

```
package src.pattern.visitor;

/**
 * VISITOR ADAPTER
 */
abstract public class VisitorAdapter implements Visitor {
	public void visitConcreteElementA(Element element){
		//Codice
	}

	public void visitConcreteElementB(Element element){
		//Codice
	}
}
```

```
package src.pattern.visitor;

/**
 * CONCRETE VISITOR 1
 */
public class ConcreteVisitor1 extends VisitorAdapter {
	public void visitConcreteElementA(Element element) {
	}

	public void visitConcreteElementB(Element element) {
	}
}
```

```
package src.pattern.visitor;

/**
 * CONCRETE VISITOR 2
 */
public class ConcreteVisitor2 extends VisitorAdapter {
	public void visitConcreteElementA(Element element) {
	}

	public void visitConcreteElementB(Element element) {
	}
}
```

```
package src.pattern.visitor;

/**
 * ELEMENT
 */
public interface Element {
	void accept(Visitor visitor);
}
```

```
package src.pattern.visitor;
```

```
/**
 * CONCRETE ELEMENT
 */
public class ConcreteElementA implements Element {
	public void accept(Visitor visitor) {
		visitor.visitConcreteElementA(this);
	}

	public void operationA() {
	}
}
```

```
package src.pattern.visitor;

/**
 * CONCRETE ELEMENT
 */
public class ConcreteElementB implements Element {
	public void accept(Visitor visitor) {
		visitor.visitConcreteElementB(this);
	}

	public void operationB() {
	}
}
```

26.11 Observer

Uno degli effetti secondari del dividere una applicazione in oggetti che cooperano tra loro, è rappresentato dalla necessità di mantenere consistente lo stato di tutti gli oggetti cooperanti, senza rinunciare a caratteristiche quali l'incapsulamento per mantenere la riutilizzabilità del codice.
Un esempio tipico è quello delle librerie disegnate per creare interfacce grafiche. Le java swing ad esempio, separano le logiche di presentazione dell'interfaccia dalle logiche di business dell'applicazione (*Figura 218*), consentendo di utilizzare gli oggetti appartenenti al primo strato indipendentemente dalla applicazione che stiamo sviluppando.

Figura 218: Separazione fisica tra presentazione e logiche di business in una interfaccia grafica

Il problema in questo schema è quello di consentire agli oggetti appartenenti al secondo strato, di ascoltare gli oggetti appartenenti al primo strato per rendersi conto se, l'utente ne ha modificato lo stato e decidere di conseguenza come aggiornare il proprio. Ad esempio, alcune classi appartenente alle logiche di business potrebbero voler sapere se un utente ha modificato il valore all'interno di un campo di testo oppure se ha selezionato un nuovo valore all'interno di una lista di elementi.
Il pattern Observer definisce una relazione "uno a molti" tra oggetti; in questo modo se un oggetto modfica il proprio stato, tutti gli oggetti correlati ne vengono notificati affinché possano aggiornare il proprio stato automaticamente (*Figura 219*).

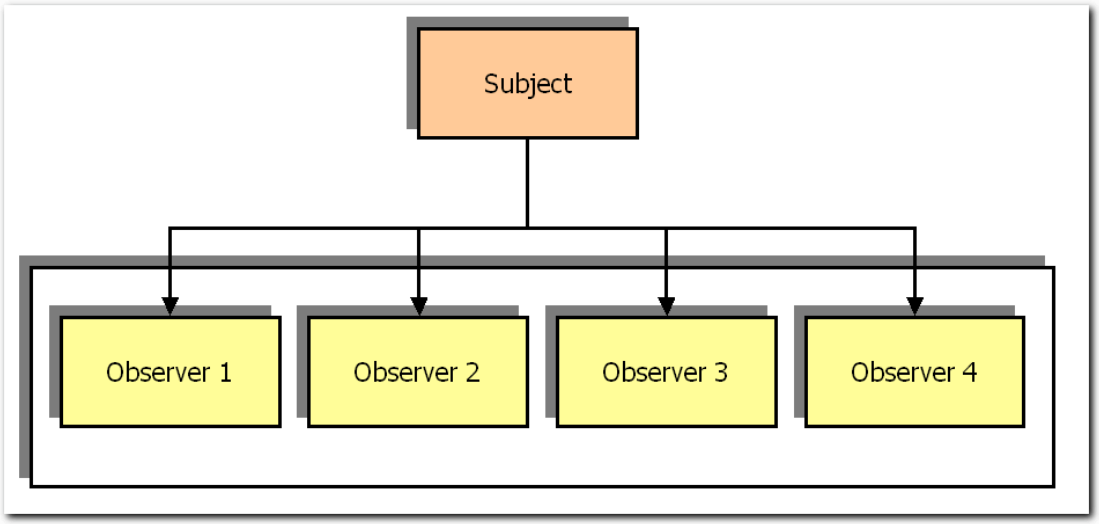


Figura 219: Notifica del cambiamento di stato di Subject ad Observer

Il problema:

- *Dobbiamo creare una struttura di oggetti tale che, il cambiamento di un oggetto deve essere notificato ad un numero imprecisato di altri oggetti;*
- *Un oggetto deve poter notificare un cambiamento di stato ad altri oggetti indipendentemente dalla loro natura.*

Il disegno:

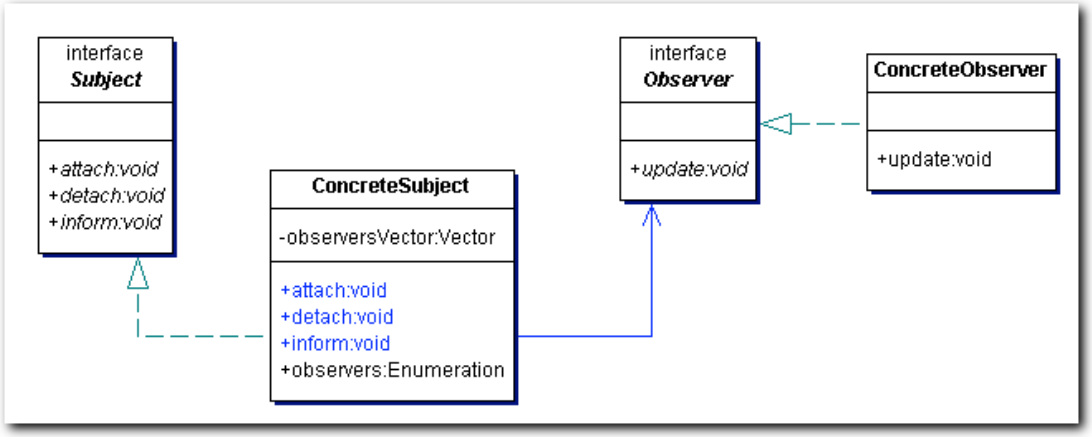


Figura 220: Il pattern Observer

Come funziona:

Nel disegno, qualsiasi numero di oggetti di tipo *Observer* può osservare il cambiamento di stato di un oggetto di tipo *Subject*.
L'interfaccia *Subject* definisce la struttura dell'oggetto che dovrà notificare il proprio cambiamento di stato fornendo i proptotipi dei metodi necessari ad aggiungere, rimuovere o gestire gli oggetti di tipo *Observer*.
La responsabilità di implementare i metodi definiti nell'interfaccia *Subject* è lasciata gli oggetti *ConcreteSubject* che, memorizzano gli stati di interesse per gli oggetti *Observer* e inviano la notifica del cambiamento dello stato ai propri osservatori mediante il metodo *inform* il cui funzionamento è schematizzato nella prossima figura:

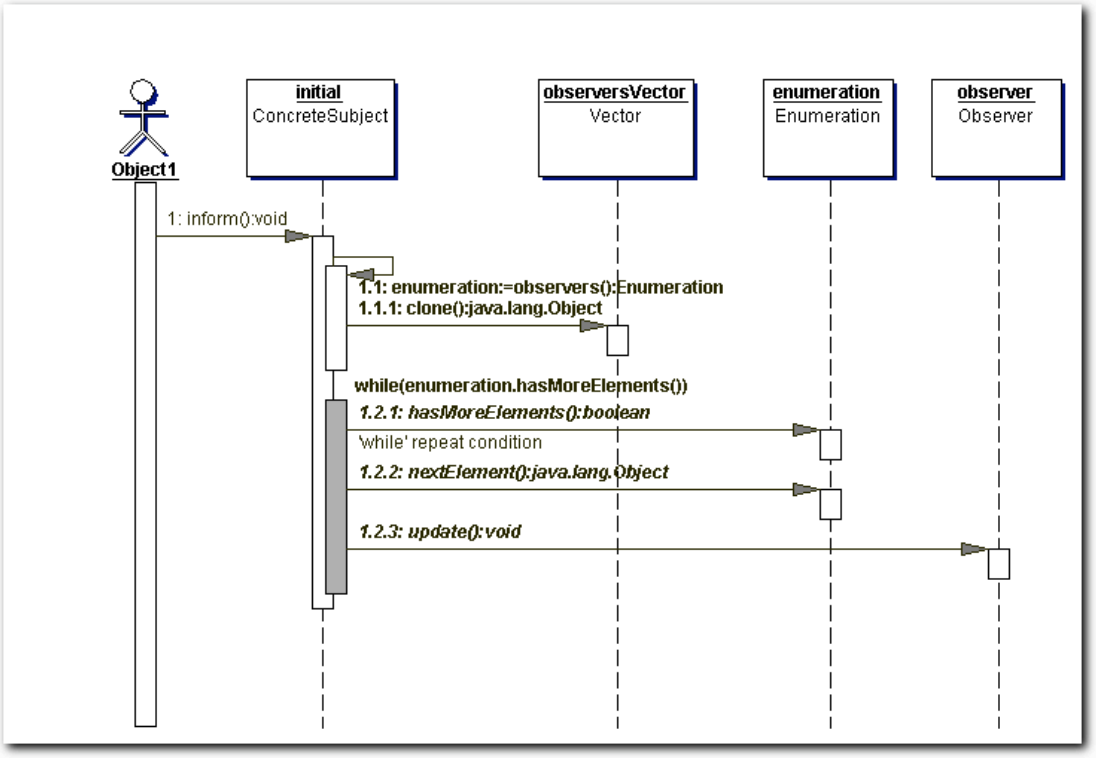


Figura 221: Il metodo inform di subject

L'interfaccia *Observer* definisce il prototipo per tutti gli oggetti che necessitano la notifica da un oggetto *ConcreteSubject*. L'interfaccia mette a disposizione il metodo *update()* utilizzato dal metodo inform per notificare il cambiamento di stato (*Figura 221*).
Infine, gli oggetti *ConcreteObserver* ,implementano l'interfaccia *Observer*, mantengono un riferimento ad un oggetto *ConcreteSubject* e mamorizzano lo stato che deve rimenere consistente con tale oggetto.

Il codice:

```
package src.pattern.observer;

/**
 * SUBJECT
 */
public interface Subject {
	void attach(Observer observer);

	void detach(Observer observer);

	void inform();
}
```

```
package src.pattern.observer;

import java.util.Vector;
```

```
import java.util.Enumeration;

/**
 * CONCRETE SUBJECT
 */
public class ConcreteSubject implements Subject {
        public void attach(Observer observer){
                observersVector.addElement(observer);
        }

        public void detach(Observer observer){
                observersVector.removeElement(observer);
        }

        public void inform(){
                java.util.Enumeration enumeration = observers();
                while (enumeration.hasMoreElements()) {
                        ((Observer)enumeration.nextElement()).update();
                }
    }

        public Enumeration observers(){
                return ((java.util.Vector) observersVector.clone()).elements();
        }

        private Vector observersVector = new java.util.Vector();
}
```

```
package src.pattern.observer;

/**
 * OBSERVER
 */
public interface Observer {
        void update();
}
```

```
package src.pattern.observer;

/**
 * CONCRETE OBSERVER
 */
public class ConcreteObserver implements Observer {
        public void update(){
                // Codice per gestire lo stato dell'oggetto
                //dopo la notifica dell'oggetto Subject
        }
}
```

26.12 Memento

Talvolta è necessario salvare lo stato interno di un oggetto per poi ripristinarlo successivamente. Basti pensare al meccanismo di "undo" che consente all'utente, in caso di errore, di percorrere a ritroso le operazioni effettuate per ripristinare l'ultimo stato consistente salvato.

Parlando di oggetti, sorge però spontaneo notare che gli oggetti incapsulano il loro stato rendendolo inaccessibile e violando l'incapsulamento si rischia di compromettere l'affidabilità e la manutenibilità della applicazione.
Scopo di Memento, è quello di consentire il salvataggio dello stato di un oggetto e renderlo accessibile se necessario senza violare il principio dell'incapsulamento.

Il problema:

- *Dobbiamo realizzare una applicazione che necessità di salvare lo stato di un oggetto per poi recuperarlo in seguito;*
- *Dobbiamo salvare temporaneamente lo stato di un oggetto;*
- *Dobbiamo catturare lo stato di un oggetto senza violare l'incapsulamento.*

Il disegno:

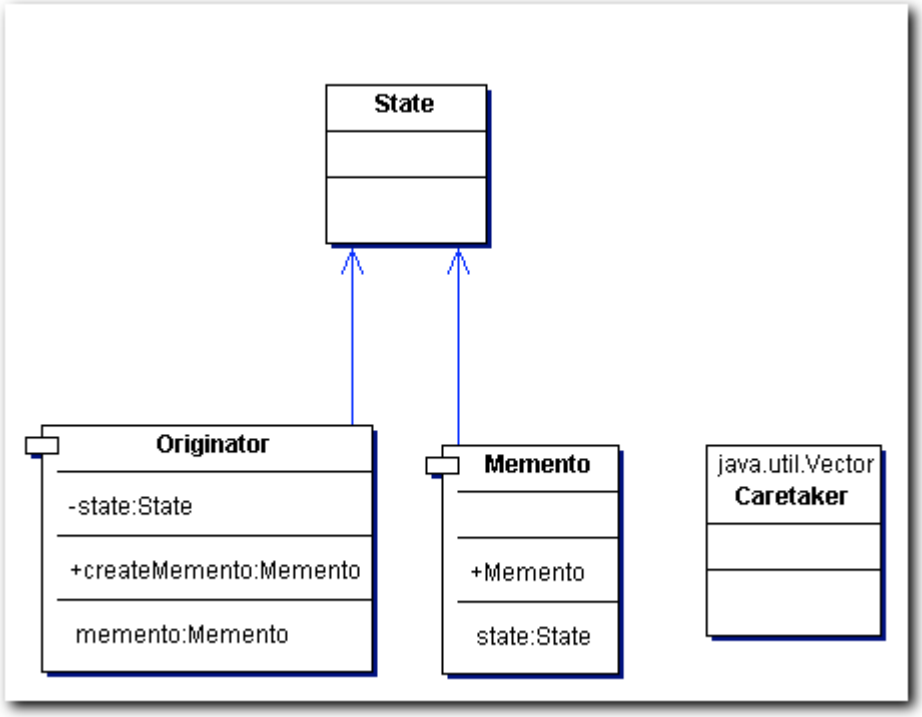


Figura 222: Il pattern Memento

Come funziona:

Originator rappresenta l'oggetto che ha necessità di salvare il proprio stato, rappresentato dalla classe *State*. Per far questo crea un nuovo oggetto di tipo *Memento* che, utilizzerà in seguito per ripristinare lo stato desiderato.

Un oggetto *Memento*, memorizza lo stato di un oggetto *Originator*. Un oggetto di questo tipo può salvare tutto o parte dello stato di *Originator*, a discrezione del suo creatore.
Infine, la classe *Caretaker* (Custode), ha la responsabilità di tener traccia degli oggetti *Memento* creati da *Originator*, e consente di navigare all'interno della lista, esaminando gli stati salvati, su richiesta dell'utente.

Il codice:

```
package src.pattern.memento;

/**
 * STATE
 */
public class State{
}
```

```
package src.pattern.memento;

/**
 * ORIGINATOR
 */
public class Originator{
	private State state;

	public void setMemento( Memento memento ){
		state = memento.getState();
	}

	public Memento createMemento(){
		return new Memento( state );
	}
}
```

```
package src.pattern.memento;

/**
 * MEMENTO
 */
public class Memento{
	private State state = null;

	public Memento( State state ){
		this.state = state;
	}

	public State getState()	{
		return state;
	}

	public void setState( State state ) {
		this.state = state;
	}
}
```

```
package src.pattern.memento;

/**
 * CARETAKER
 */
public class Caretaker extends java.util.Vector{
}
```

Infine, il client di test che mostra il funzionamento del pattern:

```
package src.pattern.memento;

/**
 * TEST CLIENT
 */
public class Test{
	public static void main( String arg[] ){
		try{
			Caretaker caretaker = new Caretaker();
			Originator originator = new Originator();
			Memento memento = originator.createMemento();
			caretaker.addElement( memento );
		}
		catch( Exception e ){
			e.printStackTrace();
		}
	}
}
```

27 GLOSSARIO DEI TERMINI

ABSTRACT
Modificatore Java utilizzato per dichiarare classi e metodi astratti. Un metodo astratto definisce soltanto il prototipo di se stesso dichiarando nome parametri di input e di output. L'mplementazione di un metodo astratto deve essere contenuta all'interno di una sottoclasse definita a partire dalla classe astratta.

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.

API (Application Programming Interface)
Una API consiste di funzioni e variabili che i programmatori possono utilizzare all'interno della loro applicazione.

Applet
Componenti Java caricate da un browser come parte di una pagina HTML.

Applet Viewer
Applicazione che fornisce l'ambiente necessario alla esecuzione di una applet Java. Internet Explorer, Hotjava, Netscape Navigator o AppletViewer sono applicazioni di questo tipo.

Applicazione
Programma Java di tipo stand-alone.

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.

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.

---

boolean
Tipo primitivo Java che contiene solo valori di verità o falsità *true* o *false*.

---

byte
Tipo primitivo Java che rappresenta un numero intero con segno ad 8 bit.

---

cast
Promozione esplicita di un tipo java in un altro.

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.

---

catch
L'istruzione catch compare sempre alla fine di un blocco di controllo delle eccezioni a seguito di una istruzione *try*. L'istruzione catch contiene un parametro che definisce il tipo di eccezione che deve essere gestita tramite il suo blocco di istruzioni.

---

char
Tipo primitivo Java a 16 bit contenente un carattere rappresentato secondo lo standard UNICODE.

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).

class
Contenitore per dati e metodi Java. La parola chiave omonima è utilizzata in Java per dichiarare una classe.

class path
Indirizzo completo della cartella sul sistema locale contenente i file java compilati.

Client
Applicazione che stabilisce una connessione con un server allo scopo di trasmettere dati.

Compilation Unit
Definizione di una classe Java. Una unità di compilazione generalmente contiene la definizione di un'unica classe.

Compiler
Programma che traduce il codice sorgente in codice eseguibile.

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 Managment Group per la comunicazione tra oggetti distribuiti.

Costruttore
Metodo invocato automaticamente quando viene creato un nuovo oggetto. Responsabile dell'impostazione dello stato iniziale di un oggetto, ha lo stesso nome della classe di appartenenza.

Datagramma

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.

double
Tipo primitivo Java. Rappresenta un numero a virgola mobile a 64 bit.

E-Commerce
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.

Eccezione
Segnala, mediate propagazione di oggetti, il verificarsi di una condizione indesiderata. Le eccezioni Java sono ooggetti che derivano dalla superclasse *Throwable* e sono propagate mediante l'istruzione *throw*.

EJB
Enterprise Java Beans rappresentano nella architettura J2EE lo standard per lo sviluppo di Oggetti di Business distribuiti.

Email
La 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.

extends
Permette di definire una classe a partire da una classe base o superclasse,

sfruttando il meccanismo della ereditarietà tra oggetti. In Java, ogni classe che non sia sottoclasse di una classe base, deriva automaticamente da *java.lang.Object*.

Extranet
Rete basata su tecnologia internet, estende una Intranet al di fuori della azienda proprietaria.

FAQ
"Frequently Asked Questions" ovvero " domande più frequenti".

final
Modificatore che consente di creare variabili che si comportano come una costante. Le variabili di questo tipo vengono inizializzate solo una volta al momento della dichiarazione e qualsiasi altro tentativo di assegnamento genera un errore di compilazione.

finalize
Meto ereditato dalla classe base *java.lang.Object* ed invocato automaticamente dal garbage collector prima di rimuove un oggetto e rilasciare la memoria allocata. Può essere utilizzato dal programmatore mediante il meccanismo di overriding, per rilasciare le risorse che altrimenti non potrebbero essere rilasciate dal garbage collector.

finally
Insieme alla parole chiave *try/catch*, rappresenta una istruzione guardiana per la gestione delle eccezioni Java. A differenza di *catch*, che segue obbligatoriamente un blocco *try*, la parola chiave *finally* è opzionale e consente di dichiarare blocchi di gestione eseguiti sempre a prescindere dall'eccezione generata dal codice nel blocco *try*.

FTP
"File Transfer Potocol" protocollo di comunicazione precedente all' HTTP, permette il trasferimento di file tra due computer.

float
Tipo primitivo Java. Rappresenta un numero a virgola mobile a 32 bit.

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.

Garbage Collector
Processo responsabile della gestione della memoria utilizzata dalla Java Virtual Machine per creare oggetti. Quando un oggetto non deve essere più utilizzato dal altri oggetti, il garbage collector lo elimina rilasciando le risorse allocate e rendendole disponibili per la creazione di nuovi oggetti.

GC
Forma abbreviata per Garbage Collector.

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.

---

hostname
Nome menmonico assegnato ad un computer collegato ad internet.

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
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.

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.

implements
Utilizzato per definire una classe sfruttando il meccanismo di ereditarietà, a partire da una interfaccia. A differenza di extends che limita l'ereditarietà ad una sola super classe, una classe Java può essere definita a partire da molte interfacce. Se utilizzato per definire una classe non stratta, tutti i metodi definiti nelle interfacce specificate debbono essere implementati all'interno del corpo della classe stessa.

import

Incapsulamento
Parlando di programmazione Object-Oriente, è la tecnica che consente di nascondere i dati di un oggetto (vd. Private, protected), consentendone l'accesso solo mediante l'invocazione di metodi pubblici.

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
E' 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.

J2EE
Acronimo di 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.

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).

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.

---

28 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;

- Stroustrup,Bearne
  "***An Overview of the C++ Programming Language***"
  http://www.research.att.com/~bs/crc.ps - 1991;

- Winder, Russell and Roberts, Graham
  "***A (Very!) Short History of Java***"
  1997

- Abadi, Martin and Cardelli, Luca
  A Theory of Objects
  Springer-Verlag – 1996

- Dahl, Ole-Johan and Hoare, C.A.R.
  "***Hierarchical Program Structures***"
  Structured Programming Ed: Ole-Johan Dahl – 1972

- Bryan Youmans
  Java: Cornerstone of the Global Network Enterprise?
  Spring 1997

- Donald G.Drake
  A quick tutorial on how implementing threads in Java
  Pubblicato su www.javaworld.com

- Alexander, Christopher, Ishikawa, Sara, *et. al.*,
  A Pattern Language
  xford University Press, New York, 1977.

- Alpert, S., Brown, K. and Woolf, B.,
  The Design Patterns Smalltalk Companion
  Addison-Wesley, 1998.

- Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *A*
  System of Patterns
  John Wiley and Sons, New York, 1996.

- Coplien, James O.
  Advanced C++ Programming Styles and Idioms
  Addison-Wesley, Reading, MA., 1992.

- Coplien, James O. and Schmidt, Douglas C.
  Pattern Languages of Program Design
  Addison-Wesley, 1995.

- Gamma, E., Helm, T., Johnson, R. and Vlissides, J.,
  Design Patterns: Abstraction and Reuse of Object Oriented Design.
  Proceedings of ECOOP '93, 405-431.

- Gamma, Eric; Helm, Richard; Johnson, Ralph and Vlissides, John
  Design Patterns. Elements of Reusable Software.
  Addison-Wesley, Reading, MA, 1995

- Krasner, G.E. and Pope, S.T.,
  A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80.
  Journal of Object-Oriented Programmng I(3)., 1988

- Pree, Wolfgang,
  Design Patterns for Object Oriented Software Development
  Addison-Wesley,1994.

- Riel, Arthur J.
  Object-Oriented Design Heuristics
  Addison-Wesley, Reading, MA, 1996