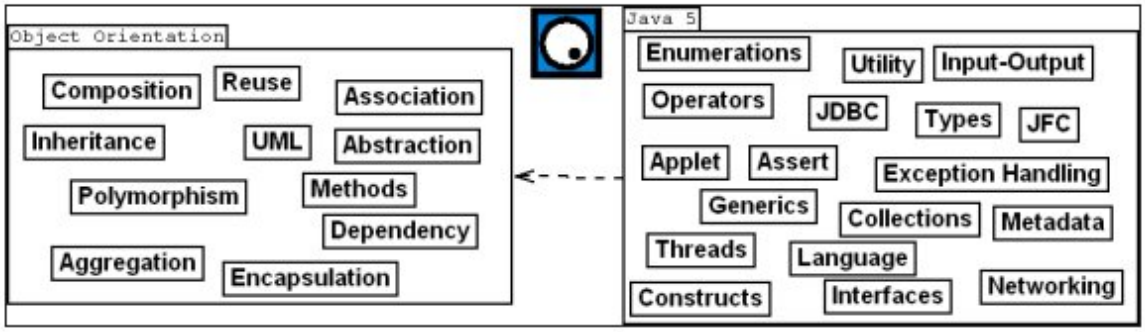
Object Oriented && Java 5



Claudio De Sio Cesari

SECONDA EDIZIONE


CLAUDIO DE SIO
OO & JAVA TRAINING,
MENTORING & CONSULTING

Indice degli argomenti

Premessa

"Object Oriented && Java 5" è un manuale sul linguaggio di programmazione Java aggiornato alla versione 5, che pone particolare enfasi sul supporto che il linguaggio offre all'object orientation. Solitamente, infatti, la difficoltà maggiore che incontra un neo-programmatore Java è nel riuscire a sfruttare in pratica proprio i paradigmi della programmazione ad oggetti. Questo testo si sforza quindi di fornire tutte le informazioni necessarie al lettore, per intraprendere la strada della programmazione Java, nel modo più corretto possibile, ovvero in maniera "Object Oriented".

Caratteristiche del testo

Una delle caratteristiche fondamentali di tale testo è che è strutturato in modo tale da facilitare l'apprendimento del linguaggio anche a chi non ha mai programmato, o chi ha programmato con linguaggi procedurali. Infatti, la forma espositiva e i contenuti sono stati accuratamente scelti basandosi sull'esperienza che ho accumulato come formatore in diversi linguaggi di programmazione, mentoring e consulenze su numerosi progetti. In particolare, per Sun Educational Services Italia, ho avuto la possibilità di erogare corsi per migliaia di discenti su tecnologia Java, analisi e progettazione object oriented ed UML.
Questo testo inoltre, è stato creato sulle ceneri di altri due manuali su Java di natura free: "Il linguaggio Object Oriented Java" e "Object Oriented && Java 5" (prima edizione). Entrambi questi manuali sono stati scaricati decine di migliaia di volte da http://www.claudiodesio.com e da molti importanti mirror specializzati nel settore, riscuotendo tantissimi consensi da parte dei lettori. In particolare la prima edizione di "Object Oriented && Java 5" è stato consigliato come libro di testo in diversi corsi di programmazione in varie importanti atenei italiani, ed utilizzato in tante aziende IT come riferimento per la formazione dei propri dipendenti.
Questa seconda edizione è quindi il risultato di un lavoro di anni, basato non sulla sola esperienza didattica, ma anche su centinaia di feedback ricevuti dai lettori.

Java 6

In libreria è anche possibile acquistare sotto il nome di "Manuale di Java 6", una versione aggiornata alla versione 6 del linguaggio, di questo testo (editore Hoepli). E' possibile acquistare "Manuale di Java 6" direttamente on line tramite questo link: http://www.hoepli.it/libro.asp?ib=8820336588. Rispetto al testo che state leggendo,

"Manuale di Java 6" oltre che essere aggiornato all'ultimissima versione di Java, è formattato e corretto dall'editore Hoepli, con un numero di pagine, un formato, un peso e un prezzo decisamente contenuti.
Il lettore deciso a stampare questo testo quindi, è liberissimo di farlo. È probabile che la spesa della stampa e della rilegatura però, sia paragonabile a quella dell'eventuale acquisto del libro che trovate in libreria. Inoltre il risultato della stampa non sarà certamente paragonabile a quella di una stampa Hoepli, e non conterrà argomenti riguardanti Java 6 come per esempio JDBC 4.0. Nel caso vogliate stampare questo libro quindi, valutate l'acquisto di "Manuale di Java 6".

A chi si rivolge

A partire dalla versione 5 (o 1.5) il linguaggio Java si è molto evoluto, diventando estremamente più complesso e potente, e obbligando gli sviluppatori ad aggiornare le proprie conoscenze non senza sforzi. Tale svolta è dovuta essenzialmente alla concorrenza spietata dei nuovi linguaggi che sono nati in questi anni, e che cercano di insediare la leadership che Java si è saputa creare in diverse nicchie tecnologiche. Il cambiamento di rotta che il linguaggio ha subito va verso una complessità maggiore, con l'introduzione di nuove potenti caratteristiche, come le enumerazioni, i generics o le annotazioni.
La versione 5 di Java, per esigenze di marketing è nota anche sotto il nome di "Tiger" (in italiano "tigre").
Questo testo affronta in modo approfondito le nuove caratteristiche introdotte da Tiger ma senza tralasciare tutte le altre evoluzioni del linguaggio dalla versione 1.0 in poi.
Lo studio di questo testo quindi, oltre che al neofita che si avvicina alla programmazione orientata agli oggetti, è consigliato anche al programmatore Java già affermato che vuole aggiornarsi alle principali caratteristiche della versione 5 di Java.
Come già detto, è un testo molto adatto alla didattica, e quindi dovrebbe risultare un ottimo strumento per diversi corsi di programmazione sia universitari sia aziendali.
Infine, il libro copre tutti gli argomenti che fanno parte della certificazione Sun Certified Java Programmer (codice CX-310-055), e quindi può risultare molto utile anche per prepararsi al test finale.

Struttura del testo

Questo testo è il frutto di anni di evoluzione, e si estende per circa 700 pagine. Esso è suddiviso in cinque parti principali:

1. Le basi del linguaggio
2. Object Orientation
3. Caratteristiche avanzate del linguaggio
4. Le librerie fondamentali
5. Java 5 Tiger

La parte 1 intitolata "**Le basi del linguaggio**", è pensata per permettere un approccio a Java "non traumatico" a chiunque. Lo studio di questa sezione infatti, dovrebbe risultare utile sia al neofita che vuole iniziare a programmare da zero, sia al programmatore esperto che ha bisogno solo di rapide consultazioni. Infatti è presente un'esauriente copertura dei concetti fondamentali del linguaggio quali classi, oggetti, metodi, operatori, costruttori e costrutti di controllo. Inoltre vengono riportati in dettaglio tutti i passi che servono per creare applicazioni Java, senza dare per scontato niente. Vengono quindi introdotti la storia, le caratteristiche, l'ambiente di sviluppo e viene spiegato come consultare la documentazione ufficiale.
Al termine dello studio di questa parte, il lettore dovrebbe possedere tutte le carte in regola per iniziare a programmare in Java.

La parte 2 intitolata "**Object Orientation**", è interamente dedicata al supporto che Java offre ai paradigmi della programmazione ad oggetti. Questa è forse la parte più impegnativa ed originale di questo testo. I paradigmi dell'object orientation vengono presentati in maniera tale che il lettore impari ad apprezzarne l'utilità in pratica. In più si è cercato di fare un'operazione di schematizzazione degli argomenti abbastanza impegnativa, in particolare del polimorfismo. In questo modo speriamo di facilitare l'apprendimento al lettore. Consigliamo lo studio di questa sezione a chiunque, anche a chi programma in Java da tempo.
Il titolo stesso di questo testo, porta con sé un messaggio intrinseco e molto significativo: se object oriented non è true, allora è superfluo sapere se Java 5 è true o false (cfr.Unità Didattica 4.1 paragrafo relativo agli operatori booleani).
Al termine di questa parte, il lettore dovrebbe aver appreso almeno a livello teorico, le nozioni fondamentali dell'Object Orientation applicabili a Java.

Nella parte 3 intitolata "**Caratteristiche avanzate del linguaggio**", vengono trattati tutti gli argomenti che completano la conoscenza del linguaggio. Vengono quindi spiegate diverse parole chiave, ed in particolare, tutti i modificatori del linguaggio. Inoltre vengono presentate anche le definizioni di classe astratta e di interfaccia, argomenti complementari all'Object Orientation. Per completezza vengono esposti anche alcuni argomenti avanzati quali le classi innestate, le classi anonime, gli inizializzatori di

istanze e statici. Un modulo a parte è dedicato alla gestione delle eccezioni, degli errori e delle asserzioni. Lo studio di questa sezione, è in particolar modo consigliata a chi vuole conseguire la certificazione Sun Java 2 Programmer. Più in generale, è consigliata a chi vuole conoscere ogni minima particolarità del linguaggio. Anche se vi considerate già programmatori Java, è altamente improbabile che abbiate già usato con profitto un inizializzatore d'istanza…
Al termine di questa parte, il lettore che ha studiato attentamente, dovrebbe aver appreso tutti i "segreti di Java".

La parte 4 "**Le librerie fondamentali**" è dedicata all'introduzione delle principali librerie. Restiamo fermamente convinti che la piena autonomia su certi argomenti possa derivare solo dallo studio della documentazione ufficiale. Questa sezione quindi non potrà assolutamente sostituire la documentazione fornita dalla Sun. Tuttavia, si propone di semplificare l'approccio alle librerie fondamentali, senza nessuna pretesa di essere esauriente. In particolare vengono introdotte le classi più importanti, con le quali prima o poi bisognerà fare i conti. Dopo una intensa immersione nel mondo dei thread, saranno introdotti i package java.lang e java.util. Tra le classi presentate per esempio, la classe System, StringTokenizer, le classi Wrapper, le classi per internazionalizzare le nostre applicazioni e quelle del framework "Collections". Introdurremo anche le principali caratteristiche dell'input-output e del networking in Java. Inoltre, esploreremo il supporto che Java offre a due altri linguaggi cardine dell'informatica dei nostri giorni, per la gestione dei dati: l'SQL e l'XML. Inoltre introdurremo le applet, impareremo a creare interfacce grafiche con le librerie AWT e Swing, e gestire gli eventi su di esse. Argomenti come i Thread, le Collections o i package java.lang e java.util fanno parte anche del programma di certificazione.
Il lettore che avrà studiato attentamente anche questa quarta parte, dovrebbe oramai essere in grado di superare l'esame SCJP, e di avere delle solide basi teoriche su cui basare i propri sforzi di programmazione.

Nella parte 5 intitolata "**Java 5 Tiger**", come è facile intuire, è interamente dedicata all'approfondimento alle nuove e rivoluzionarie caratteristiche della release 1.5 del linguaggio. Dai cicli for migliorati ai generics, dalle enumerazioni alle annotazioni, ci sono ancora tanti argomenti da dover studiare. È stata fatta la scelta di separare questa sezione dalle altre, per rendere più graduale l'apprendimento per chi inizia, e più semplice la consultazione a chi è già pratico.

Ogni parte è suddivisa in moduli. Ogni modulo è suddiviso in unità didattiche che spesso sono a loro volta suddivise in vari paragrafi.

Ogni modulo è contrassegnato da un livello di complessità (Alto, Medio o Basso), in modo tale da permettere al lettore di organizzare al meglio il tempo e la concentrazione da dedicare al suo studio.
In ogni modulo sono anche prefissati degli obiettivi da raggiungere. Questo dovrebbe permettere di individuare da subito i punti cardine degli argomenti del modulo, senza perdersi nelle varie osservazioni e precisazioni esposte.
Inoltre ogni modulo termina con un paragrafo dal titolo fisso: "Riepilogo". Questo paragrafo ha lo scopo di riassumere e stimolare la riflessione sugli argomenti appena trattati, nonché di rafforzare nel lettore la terminologia corretta da utilizzare.
Alla fine di ogni modulo vengono anche presentati diversi esercizi (con relative soluzioni), al fine di verificare il grado di comprensione degli argomenti.
Questi sono tutti originali ed ideati dall'autore, e progettati al fine di non essere noiosi, e soprattutto di essere utili. Gli esercizi sono parte integrante del manuale e sono da considerarsi complementari alla teoria. Spesso hanno la forma di domande del tipo "vero o falso", e sono progettati accuratamente al fine di verificare l'attenzione dedicata dal lettore allo studio dei concetti, e l'effettivo apprendimento degli approfondimenti proposti. Lo stile delle domande, è ispirato a quello utilizzato dai test di certificazione Sun Java 2 programmer.
Si ricorda al lettore che la natura gratuita del documento implica probabili imperfezioni, di cui l'autore rifiuta ogni responsabilità diretta o indiretta. Qualsiasi segnalazione d'errore (tecnico, grammaticale, tipografico etc…) è gradita, e può essere inoltrata all'autore all'indirizzo claudio@claudiodesio.com.

EJE

Sul mio sito http://www.claudiodesio.com o da http://sourceforge.net/projects/eje potete scaricare gratuitamente anche una copia di EJE, un editor Java open source creato da me, creato proprio per supportare questo testo. È semplice da utilizzare, intuitivo, e non richiede prerequisiti particolari. Se state iniziando a programmare, vi consiglio vivamente di effettuarne il download.

Licenza

Questo manuale ha natura assolutamente gratuita. È possibile utilizzarlo in qualsiasi ambito (privato, aziendale, accademico etc…) senza che sia corrisposta nessuna cifra né all'autore, né a nessun altra persona che lo pretenda. Non è possibile vendere copie di questo testo, né in formato elettronico né cartaceo. È possibile stampare questo testo per scopo personale, ma resta valido quanto detto nel paragrafo intitolato "Java 6".

claudio@claudiodesio.com

L'indirizzo e-mail claudio@claudiodesio.com, si è saturato presto grazie ai centinaia di feedback ricevuti dai lettori della prima versione di questo testo. La mia unica richiesta ai lettori di "Object Oriented && Java 5 – seconda edizione", rimane la stessa. Al termine della lettura, sarebbe gradita una vostra valutazione, positiva o negativa che sia. Sono gradite anche le segnalazioni di errori, nonché qualsiasi suggerimento.
Ovviamente, nessun obbligo…
Sperando che apprezziate il mio sforzo, vi ringrazio e vi auguro un buon lavoro…

Claudio De Sio Cesari

Ringraziamenti

I miei personali ringraziamenti vanno a tutte le persone a cui sono legato affettivamente.

In particolare alle mie ragioni di vita: Andrea, Simone e la loro dolce mammina Rosalia…

Ai miei genitori e fratelli che mi hanno sempre supportato…

A tutte le persone di Didactica che hanno creduto in me sin dall'inizio.

Inoltre, vorrei ringraziare tutti i miei colleghi di lavoro in Sun, con cui ho il privilegio di lavorare…

Un ringraziamento particolare va a tutte le persone che mi hanno apprezzato e voluto bene in SSC di Napoli, con cui ho condiviso oltre un anno di lavoro e piacere. Grazie ragazzi, non vi dimenticherò …

Ringrazio con il cuore anche tutti i miei attuali colleghi di lavoro (nonché amici) in EDS Pozzuoli, persone eccezionali sia dal punto di vista lavorativo che umano. Quando finirà la mia esperienza con voi spero di lasciarvi un ricordo positivo, come quello che io avrò di voi…

Colgo l'occasione per ringraziare anche tutte le persone che hanno scritto all'autore per commenti, suggerimenti, consigli e complimenti. Grazie a loro soprattutto, sono riuscito a trovare la voglia e il tempo per dare alla luce questo testo. Purtroppo non posso citare in questa sede tutti i nomi che vorrei (sono diverse centinaia!), né vorrei fare torto a qualcuno. Molte mail, riportavano solo complimenti e ringraziamenti. Alcune di queste mi hanno fatto sorridere, altre mi hanno messo di buon umore, altre mi hanno fatto sentire utile, qualcuno mi ha fatto addirittura commuovere.

Ma non posso che ringraziare in maniera particolare soprattutto che si è offerto spontaneamente di segnalarmi eventuali errori, omissioni e altro come Antonio Flaccomio, della Nortel Networks di Palermo, e Luca Caserta di Didactica di Napoli.

Per quest'ultima versione del testo, un contributo notevole è stato apportato dal lavoro incredibile di Giulio Zoccoramazzo. Una persona che ha dimostrato non solo una notevole competenza, ma una gentilezza, un altruismo, una classe ed una sensibilità fuori dal comune. Giulio, ha dedicato molte ore in una lettura attenta, critica e costruttiva verso questo testo, individuando decine e decine di errori, e regalandomi preziosi suggerimenti e addirittura alcuni esempi di codice. Un contribuito tanto grande poteva giungere solamente da una persona molto speciale, il mio GRAZIE e la mia sincera ammirazione per te sono veramente immensi! Grazie ancora!

Dulcis in fundo, non posso dimenticare il "maestro consulente" Giuseppe Coppola, che come un fratello maggiore, mi ha spianato la strada in Sun con grande altruismo e lealtà. Grazie ancora Consulente!

A chi è rivolto l'invito a leggere questo manuale:

- A chi non ha mai programmato
- A chi non ha mai programmato in Java
- A chi ha provato a leggere altri testi ma ha ancora le idee confuse
- A chi vuole solo chiarirsi qualche idea sull'Object Orientation
- A chi è stato costretto dalla sua azienda a scrivere Servlet e JSP, senza aver mai imparato Java
- A chi vuole migrare dal C/C++, Delphi o SmallTalk a Java in maniera veloce
- A chi vuole migrare da Visual Basic (o linguaggi simili) a Java, ma senza fretta
- A chi non conosce l'inglese e vuole imparare il linguaggio Java
- A chi prima di spendere soldi in libreria per un manuale tradotto (male) dall'inglese, vuole assicurarsi che Java sia il linguaggio giusto
- A chiunque sia curioso di leggere questo testo
- A chi vuole capire l'essenza della libreria standard di Java
- A chi vuole conoscere le nuove caratteristiche di Java 5

A chi NON è rivolto l'invito a leggere questo manuale:

- A chi interessano le tecnologie Java ma non il linguaggio
- A chi è troppo convinto che la programmazione sia solo "smanettare", e che tutti i linguaggi sono uguali
- A chi non ha pazienza
- A chi pensa che imparare Java sia facile come imparare Visual Basic

Preparazione e convenzioni di scrittura:

`Courier New`	E' lo stile utilizzato per scrivere parti di codice di Java.
`[ ]`	Nelle definizioni formali, le parentesi quadre saranno utilizzate per rinchiudere parti di codice che sono considerate opzionali.
Questo è un "nota bene"	Questo stile viene utilizzato per enfatizzare alcune affermazioni. E' equivalente ad un "nota bene".
javac Pippo.java	Questo stile è utilizzato per le frasi che appaiono nella console.

- Non sono richiesti requisiti particolari al lettore, ma quantomeno una conoscenza seppur parziale della composizione e del funzionamento di un personal computer. E'ovviamente avvantaggiato il discente che ha già avuto esperienza di programmazione con altri linguaggi.

- Si darà per scontato che il lettore utilizzi e sappia utilizzare un sistema operativo Windows-Dos, almeno per quanto riguarda le funzionalità di base (cfr. App. A). Se il lettore utilizza un altro sistema operativo come Linux, probabilmente non avrà bisogno di consigli… deve solo stare attento a trovare delle corrispondenze tra ciò che è riportato in questo testo, e ciò che fa parte del proprio contesto operativo (per esempio: "prompt Ms-Dos" è equivalente a "shell Unix", il separatore "\" è equivalente a "/", il separatore "**;**" è equivalente "**:**", etc…).

- Per semplicità consigliamo di raccogliere in un'unica cartella di lavoro, tutti gli esercizi (almeno inizialmente). Il consiglio dell'autore è quello di crearsi una cartella nella directory "C:\", e chiamarla per esempio "CorsoJava".

E' probabile che il lettore voglia copiare ed incollare parti di codice dal manuale, direttamente sull'editor di programmazione utilizzato. Questa operazione potrebbe importare caratteri non interpretabili da alcuni editor (per esempio dal "blocco note"). Solitamente il problema è causato dai simboli di virgolette (" e "), che bisogna poi ri-scrivere. In

questa revisione, abbiamo cercato di utilizzare nel codice solo caratteri "copiabili", ma è possibile che ci sia scappato qualcosa…

Parte I

“Le basi del linguaggio”

La parte I è pensata per permettere un approccio a Java “non traumatico” a chiunque. Lo studio di questa sezione infatti, dovrebbe risultare utile sia al neofita che vuole iniziare a programmare da zero, sia al programmatore esperto che ha bisogno solo di rapide consultazioni. Infatti è presente un’esauriente copertura dei concetti fondamentali del linguaggio quali classi, oggetti, metodi, operatori, costruttori e costrutti di controllo. Inoltre vengono riportati in dettaglio tutti i passi che servono per creare applicazioni Java, senza dare per scontato niente. Vengono quindi introdotti la storia, le caratteristiche, l’ambiente di sviluppo e viene spiegato come consultare la documentazione ufficiale.
Al termine dello studio di questa parte, il lettore dovrebbe possedere tutte le carte in regola per iniziare a programmare in Java.

1

Complessità: bassa

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper definire il linguaggio di programmazione Java e le sue caratteristiche (unità 1.1).
2. Interagire con l'ambiente di sviluppo: il Java Development Kit (unità 1.2, 1.5, 1.6).
3. Saper digitare, compilare e mandare in esecuzione una semplice applicazione. (unità 1.3, 1.4, 1.5, 1.6).

1 Introduzione

L'obbiettivo primario di questo modulo è far sì che anche il lettore che si avvicina per la prima volta alla programmazione possa in qualche modo ottenere dei primi, concreti risultati. Non daremo nulla per scontato: dalla definizione di Java alla descrizione dell'ambiente di sviluppo, dal processo che occorre seguire per eseguire la nostra prima applicazione alla descrizione (e risoluzione) di eventuali messaggi di errore.

1.1 Introduzione a Java (le 5 w)

In questo primo paragrafo introdurremo Java, seguendo la famosa regola descrittiva delle "5 W" del giornalismo moderno.

1.1.1 Cosa è Java (what)

Secondo alcune statistiche, attualmente Java è il linguaggio di programmazione con il più alto numero di sviluppatori nel mondo. Il successo crescente ed inarrestabile presso la comunità dei programmatori ha fatto sì che Sun Microsystems, società che ha sviluppato il linguaggio, negli anni investisse molto su esso. In poco tempo infatti, si sono evolute ed affermate tecnologie (JSP, EJB, Applet, Midlet etc.), basate sul linguaggio, che si sono diffuse in molti ambiti del mondo della programmazione. Oggigiorno esistono miliardi di congegni elettronici che utilizzano tecnologia Java: SCADA, telefoni cellulari, decoder, smart card, robot che passeggiano su Marte etc…

Con il termine "Java", quindi, ci si riferisce sia al linguaggio sia alla tecnologia che include tutte le tecnologie di cui sopra.
In questo testo ci dedicheremo essenzialmente a parlare del linguaggio, ed introdurremo solamente le tecnologie Java più importanti.
Java è un linguaggio orientato agli oggetti (object oriented), ovvero supporta i paradigmi dell'incapsulamento, dell'ereditarietà e del polimorfismo. Ciò ne rende l'apprendimento ostico per chi ha esperienze radicate di programmazione procedurale. Inoltre si tratta di un linguaggio in continua evoluzione e gli sviluppatori devono continuamente aggiornarsi per restare al passo. Dalla versione 1.0 alla versione 1.5 (detta anche versione 5), i cambiamenti sono stati veramente tanti e importanti.

1.1.2 Breve storia di Java (who, where & when)

Java ha visto la luce a partire da ricerche effettuate alla Stanford University all'inizio degli anni Novanta.

Già in questo Java si differenzia da molti altri importanti linguaggi, nati per volere di una multinazionale allo scopo di conquistare nuove fette di mercato. Nello sviluppo di Java ci si impose invece di creare un "super-linguaggio" che superasse i limiti e i difetti di altri linguaggi. Più semplicemente ci si impose di creare il "linguaggio ideale".

Nel 1992 nasce il linguaggio Oak (in italiano "quercia"), prodotto da Sun Microsystems e realizzato da un gruppo di esperti sviluppatori, che formavano il cosiddetto "Green Team". Erano capitanati da James Gosling, oggigiorno uno tra i più famosi e stimati "guru" informatici del pianeta, nonché vicepresidente di Sun. Sembra che il nome Oak derivi dal fatto che Gosling e i suoi colleghi, nel periodo in cui svilupparono questo nuovo linguaggio, avessero avuto come unica compagnia quella di una quercia che si trovava proprio fuori la finestra dell'ufficio in cui lavoravano.
In un primo momento Sun decise di destinare questo nuovo prodotto alla creazione di applicazioni complesse per piccoli dispositivi elettronici. In particolare, sembrava che i campi strategici da conquistare in quegli anni fossero la domotica e la TV via cavo. In realtà i tempi non erano ancora maturi per argomenti quali il "video on demand" e gli "elettrodomestici intelligenti". Solo dopo qualche anno, infatti, si è iniziato a richiedere video tramite Internet o TV via cavo. E ancora oggi l'idea di avere un "robot" che svolga le faccende domestiche rimane solo un sogno per le casalinghe.
Nel 1993, con l'esplosione di Internet (negli Stati Uniti), nacque l'idea di trasmettere codice eseguibile attraverso pagine HTML. La nascita delle applicazioni che utilizzano la tecnologia CGI (Common Gateway Interface) rivoluzionò il World Wide Web.

Il mercato che sembrò più appetibile allora, divenne ben presto proprio Internet. Nel 1994 viene quindi prontamente realizzato un browser che fu chiamato per breve tempo "Web Runner" (dal film "Blade Runner", il preferito di Gosling) e poi, in via definitiva, "HotJava". Per la verità non si trattava di uno strumento molto performante, ma la sua nascita dimostrò al mondo che con Java si poteva iniziare a programmare per Internet.
Il 23 maggio del 1995 Oak, dopo una importante rivisitazione, viene ribattezzato ufficialmente col nome "Java".

Il nome questa volta sembra derivi da una tipologia indonesiana di caffè molto famosa negli Stati Uniti, che pare fosse la preferita di Gosling. Per quanto Gosling sia un grande genio, non si può dire che abbia molta fantasia nello scegliere nomi!

Contemporaneamente Netscape Corporation annuncia la scelta di dotare il suo allora omonimo e celeberrimo browser della Java Virtual Machine (JVM). Si tratta del software che permette di eseguire programmi scritti in Java su qualsiasi piattaforma. Questo significava una nuova rivoluzione nel mondo Internet: grazie alla tecnologia Applet, le pagine Web diventavano interattive a livello client (ovvero le applicazioni vengono eseguite direttamente sulla macchina dell'utente di Internet, e non su un server remoto). Gli utenti potevano per esempio utilizzare giochi direttamente sulle pagine Web ed usufruire di chat dinamiche e interattive. In breve tempo, inoltre, Sun Microsystems mette a disposizione gratuitamente il kit di sviluppo JDK (Java Development Kit) scaricabile dal sito http://java.sun.com. Nel giro di pochi mesi i download del JDK 1.02a diventarono migliaia e Java iniziò ad essere sulla bocca di tutti. La maggior parte della pubblicità di cui usufruì Java nei primi tempi era direttamente dipendente dalla possibilità di scrivere piccole applicazioni in rete, sicure, interattive ed indipendenti dalla piattaforma, chiamate proprio "applet" (in italiano si potrebbe tradurre "applicazioncina"). Nei primi tempi, quindi, sembrava che Java fosse il linguaggio giusto per creare siti Web spettacolari. In realtà, Java era molto più che un semplice strumento per rendere piacevole alla vista la navigazione. Inoltre ben presto furono realizzati strumenti per ottenere certi risultati con minor sforzo (vedi Flash di Macromedia). Tuttavia, la pubblicità che a Java hanno fornito Netscape (nel 1995 un colosso che combatteva alla pari con Microsoft la "guerra dei browser"…) e successivamente altre grandi società quali IBM, Oracle etc. ha dato i suoi frutti.
Negli anni Java è diventato sempre più la soluzione ideale a problemi (come la sicurezza) che accomunano aziende operanti in settori diversi come banche, software house e compagnie d'assicurazioni.
Inoltre, nel nuovo millennio la tecnologia Java ha conquistato nuove nicchie di mercato

come le smart card, i telefoni cellulari e il digitale terrestre. Queste conquiste sono sicuramente state un successo. Java ha infatti dato un grosso impulso alla diffusione di questi beni di consumo negli ultimi anni.
Infine, grazie alla sua filosofia "write once, run everywhere" ("scritto una volta gira dappertutto"), la tecnologia Java è potenzialmente eseguibile anche su congegni che non hanno ancora visto la luce.
Oggi Java è quindi un potente e affermatissimo linguaggio di programmazione. Conta il più alto numero di sviluppatori attivi nel mondo. La tecnologia Java ha invaso la nostra vita quotidiana essendo presente, per esempio, nei nostri telefoni cellulari.
Probabilmente si tratta del primo linguaggio di programmazione il cui nome è entrato nel vocabolario di chi di programmazione non ne sa nulla. Infatti, spesso si sente parlare di Java Chat, giochi Java, Java card, etc…

1.1.3 Perché Java (why)

Nei mesi che precedettero la nascita di Oak, nel mondo informatico circolavano statistiche alquanto inquietanti. Una di esse calcolava che, in una prima release di un'applicazione scritta in C++, fosse presente in media un bug ogni cinquanta righe di codice! Considerando anche che il C++ era all'epoca il linguaggio più utilizzato nella programmazione, la statistica risultava ancora più allarmante.
In un'altra si affermava che, circa per un ottanta per cento, la presenza dei bug fosse dovuta fondamentalmente a tre ragioni:

1. scorretto uso dell'aritmetica dei puntatori;
2. abuso delle variabili globali ;
3. conseguente utilizzo incontrollato del comando `goto`.

> **Il lettore che non ha mai sentito parlare di questi argomenti, non si allarmi! Ciò non influirà sulla comprensione degli argomenti descritti in questo testo. Di seguito tenteremo di dare delle definizioni semplificate.**

L'aritmetica dei puntatori permette ad uno sviluppatore di deallocare (e quindi riutilizzare) aree di memoria al runtime. Ovviamente questo dovrebbe favorire il risparmio di risorse e la velocità dell'applicazione. Ma, tramite l'aritmetica dei puntatori, il programmatore ha a disposizione uno strumento tanto potente quanto pericoloso e difficile da utilizzare.
Per variabili globali intendiamo le variabili che sono dichiarate all'interno del programma chiamante, accessibili da qualsiasi funzione dell'applicazione in qualsiasi momento. Esse rappresentano per la maggior parte dei programmatori una scorciatoia

molto invitante per apportare modifiche al codice che hanno da manutenere. Giacché il flusso di lavoro di un'applicazione non è sempre facilmente prevedibile, per cercare di evitare che le variabili globali assumano valori indesiderati, il programmatore avrà a disposizione il "famigerato" comando `goto`. Questo comando permette di saltare all'interno del flusso di esecuzione dell'applicazione, da una parte di codice ad un'altra, senza tener conto più di tanto della logica di progettazione. Ovviamente l'utilizzo reiterato di variabili globali e di `goto` aumenta la cosiddetta "entropia del software", garantendo alle applicazioni vita breve.
A partire da queste ed altre statistiche, Java è stato creato proprio per superare i limiti esistenti negli altri linguaggi. Alcuni dei punti fermi su cui si basò lo sviluppo del linguaggio furono l'eliminazione dell'aritmetica dei puntatori e del comando `goto`. In generale si andò nella direzione di un linguaggio potente, moderno, chiaro, ma soprattutto robusto e "funzionante". In molti punti chiave del linguaggio è favorita la robustezza piuttosto che la potenza; basta pensare all'assenza dell'aritmetica dei puntatori.

In questo testo sottolineeremo spesso come il linguaggio supporti tali caratteristiche.

Gli sviluppatori originali di Java hanno cercato di realizzare il linguaggio preferito dai programmatori, arricchendolo delle caratteristiche migliori degli altri linguaggi e privandolo delle peggiori e delle più pericolose.

1.1.4 Caratteristiche di Java

Java ha alcune importanti caratteristiche che permetteranno a chiunque di apprezzarne i vantaggi.
Sintassi: è simile a quella del C e del C++, e questo non può far altro che facilitare la migrazione dei programmatori da due tra i più importanti ed utilizzati linguaggi esistenti. Chi non ha familiarità con questo tipo di sintassi può inizialmente sentirsi disorientato e confuso, ma ne apprezzerà presto l'eleganza e la praticità.
Gratuito: per scrivere applicazioni commerciali non bisogna pagare licenze a nessuno. Infatti il codice Java si può scrivere anche utilizzando editor di testo come il Blocco Note, e non per forza un complicato IDE con una costosa licenza. Ovviamente esistono anche tanti strumenti di sviluppo a pagamento, che possono accelerare lo sviluppo delle applicazioni Java. Ma esistono anche eccellenti prodotti gratuiti ed open source come Eclipse (http://www.eclipse.org), che non hanno nulla da invidiare a nessun altro tool.
Robustezza: questa è soprattutto una conseguenza di una gestione delle eccezioni chiara e funzionale, e di un meccanismo automatico della gestione della memoria (Garbage

Collection) che esonera il programmatore dall'obbligo di dover deallocare memoria quando ce n'è bisogno, punto tra i più delicati nella programmazione. Inoltre il compilatore Java è molto "severo". Il programmatore è infatti costretto a risolvere tutte le situazioni "poco chiare", garantendo al programma maggiori chance di corretto funzionamento. È molto meglio ottenere un errore in compilazione che in esecuzione…
Libreria e standardizzazione: Java possiede un'enorme libreria di classi standard, ottimamente documentate. Ciò rende Java un linguaggio di alto livello e permette anche ai neofiti di creare applicazioni complesse in breve tempo. Per esempio, è piuttosto semplice gestire finestre di sistema (interfacce grafiche utente), collegamenti a database e connessioni di rete. E questo indipendentemente dalla piattaforma su cui si sviluppa. Inoltre, grazie alle specifiche di Sun, non esisteranno per lo sviluppatore problemi di standardizzazione, come compilatori che compilano in modo differente.
Indipendenza dall'architettura: grazie al concetto di macchina virtuale ogni applicazione, una volta compilata, potrà essere eseguita su una qualsiasi piattaforma (per esempio un PC con sistema operativo Windows o una workstation Unix).
Questa è sicuramente la caratteristica più importante di Java. Infatti, nel caso in cui si debba implementare un programma destinato a diverse piattaforme, non ci sarà la necessità di doverlo convertire radicalmente da questa a quella. E' evidente quindi che la diffusione di Internet ha favorito e favorirà sempre di più la diffusione di Java.
Java Virtual Machine: ciò che rende di fatto possibile l'indipendenza dalla piattaforma è la Java Virtual Machine (da ora in poi JVM), un software che svolge un ruolo da interprete (ma non solo) per le applicazioni Java. Più precisamente: dopo aver scritto il nostro programma Java, bisogna prima compilarlo (per i dettagli riguardante l'ambiente e il processo di sviluppo, vi rimandiamo al paragrafo successivo). Otterremo così non un file eseguibile (ovvero la traduzione in linguaggio macchina del file sorgente che abbiamo scritto in Java), ma un file che contiene la traduzione del nostro listato in un linguaggio molto vicino al linguaggio macchina, detto "bytecode". Una volta ottenuto questo file dobbiamo interpretarlo. A questo punto la JVM interpreterà il bytecode ed il nostro programma andrà finalmente in esecuzione. Quindi, se una piattaforma qualsiasi possiede una Java Virtual Machine, ciò sarà sufficiente per renderla potenziale esecutrice di bytecode. Infatti, da quando ha visto la luce Java, i browser più diffusi implementano al loro interno una versione della JVM, capace di mandare in esecuzione le applet Java. Ecco quindi svelato il segreto dell'indipendenza della piattaforma: se una macchina possiede una JVM, può eseguire codice Java.

Un browser mette a disposizione una JVM solamente per le applet, non per le applicazioni standard. Spesso però le applicazioni scritte in Java vengono distribuite con una installazione di JVM incorporata.

Si parla di "macchina virtuale" perché in pratica questo software è stato implementato per simulare un hardware. Si potrebbe affermare che “il linguaggio macchina sta ad un computer come il bytecode sta ad una Java Virtual Machine”.
Oltre che permettere l’indipendenza dalla piattaforma, la JVM permette a Java di essere un linguaggio multithreaded (caratteristica di solito tipica dei sistemi operativi), ovvero capace di mandare in esecuzione più processi in maniera parallela. Inoltre offre meccanismi di sicurezza molto potenti, la "supervisione" del codice da parte del Garbage Collector, validi aiuti per gestire codice al runtime e tanto altro…
Orientato agli oggetti: Java fornisce strumenti che praticamente ci "obbligano" a programmare ad oggetti. I paradigmi fondamentali della programmazione ad oggetti (ereditarietà, incapsulamento, polimorfismo) sono più facilmente apprezzabili e comprensibili. Java è più chiaro e schematico che qualsiasi altro linguaggio orientato agli oggetti. Sicuramente chi impara Java potrà in un secondo momento accedere in modo più naturale alla conoscenza di altri linguaggi orientati agli oggetti.
Semplice: qui bisogna fare una precisazione. Java è un linguaggio molto complesso, considerandone la potenza e tenendo presente che ci obbliga ad imparare la programmazione ad oggetti. In compenso si possono ottenere risultati insperati in un tempo relativamente breve. Durante la lettura di questo manuale apprezzeremo sicuramente le semplificazioni che ci offre Java. Abbiamo per esempio già accennato al fatto che non esiste l’aritmetica dei puntatori, grazie all’implementazione della Garbage Collection. Provocatoriamente Bill Joy, vicepresidente di Sun Microsystems negli anni in cui nacque il linguaggio, nonché creatore di software storici come il sistema operativo Solaris e l’editor di testo Vi, propose come nome alternativo a Java "C++--". Questo per sottolineare con ironia che il nuovo linguaggio voleva essere un nuovo C++, ma senza le sue caratteristiche peggiori (o, se vogliamo, senza le caratteristiche più difficili da utilizzare e quindi più pericolose).
Sun ha inoltre pubblicizzato Java dalla versione 5 in poi, sostituendo il termine “Simplicity” con “ease of development” (in italiano “facilità di sviluppo”). Infatti, con l’introduzione nella versione 5 di caratteristiche rivoluzionarie come i Generics o le Annotazioni, sembra aver cambiato strada. Il linguaggio è diventato più difficile da imparare, ma i programmi dovrebbero però essere più semplici da scrivere. Insomma, imparare Java, non sarà una passeggiata.
Sicurezza: ovviamente, avendo la possibilità di scrivere applicazioni interattive in rete, Java possiede anche caratteristiche di sicurezza molto efficienti. Come c’insegna però la vita quotidiana, nulla è certo al 100%. Intanto, oggi come oggi, Java è semplicemente il linguaggio “più sicuro” in circolazione. Per esempio, il solo lanciare un’applicazione Java implica isolare il codice nella JVM, senza un diretto accesso alla memoria.

Risorse di sistema richieste: questo è il punto debole più evidente di Java. Infatti, non esistendo l'aritmetica dei puntatori, la gestione della memoria è delegata alla Garbage Collection della JVM. Questa garantisce il corretto funzionamento dell'applicazione (ovvero la memoria ancora utilizzabile non viene deallocata), nonché una notevole agevolazione per la scrittura del codice, ma non favorisce certo l'efficienza. Inoltre i tipi primitivi di Java non si possono definire "leggeri". Per esempio i caratteri sono a 16 bit, le stringhe immutabili e non esistono tipi numerici senza segno (gli "unsigned").
Chi scrive ha iniziato il proprio percorso di programmatore Basic con un Commodore 64 che metteva a disposizione 38911 byte. Attualmente invece (anno 2006) stiamo utilizzando un Pentium 4 con 1024 MB, e siamo consapevoli che tra qualche mese quest'ultima affermazione farà sorridere! Le prestazioni di un'applicazione Java quindi migliorano di mese in mese grazie ai continui miglioramenti degli hardware dove l'applicazione viene eseguita, e la stessa viene eseguita in maniera corretta!

La questione è anche dipendente dalle nicchie di mercato a cui Java è destinato. Infatti, per scrivere un driver di periferica, Java non è assolutamente raccomandabile. Molto meglio C, C++ o Assembler. Ma se parliamo di applicazioni distribuite o di rete (client-server, peer-to-peer, Web Services etc...), Java diventa l'ambiente di sviluppo ideale, anche se parliamo di prestazioni.

1.2 Ambiente di sviluppo

Per scrivere un programma Java, abbiamo bisogno di:

1. **Un programma che permetta di scrivere il codice**. Per iniziare può andar bene un semplice Text Editor, come ad esempio Notepad (Blocco Note) di Windows, Edit del Dos o Vi di Unix (sconsigliamo WordPad…).
   È anche possibile eseguire il download gratuito dell'editor Java open source EJE, agli indirizzi http://www.claudiodesio.com/eje_it.htm e http://sourceforge.net/projects/eje/. Si tratta di un editor Java di semplice utilizzo, che offre alcune comodità rispetto ad un editor di testo generico. Tra le sue caratteristiche ricordiamo la colorazione della sintassi di Java, e il completamento del testo di alcune espressioni. Soprattutto è possibile compilare e mandare in esecuzione file, tramite la pressione di semplici pulsanti.
   L'autore ringrazia anticipatamente il lettore che utilizzerà EJE e restituirà feedback all'indirizzo eje@claudiodesio.com. I feedback saranno presi in considerazione per lo sviluppo delle future versioni dell'editor. Per maggiori informazioni rimandiamo le persone interessate alle pagine del sito riguardanti

EJE o all'appendice C di questo testo.
Tuttavia, suggeriamo al lettore di utilizzare anche Notepad, almeno per questo primo modulo. Così acquisirà maggiore consapevolezza degli argomenti di basso livello.
È opportuno provare gli esercizi prima con Notepad, per poi ripeterli con EJE.

2. **Il Java Development Kit versione Standard Edition** (da ora in poi **JDK**). E' scaricabile gratuitamente sul sito di riferimento dello sviluppatore Java: http://java.sun.com, con le relative note d'installazione e documentazione. Nell'appendice B è descritto il processo da seguire per procurarsi ed installare correttamente l'ambiente di sviluppo su un sistema operativo Windows 95/98/2000/ME/XP.
   Abbiamo già accennato al fatto che Java è un linguaggio che si può considerare in qualche modo sia compilato che interpretato. Per iniziare abbiamo quindi bisogno di un compilatore e di una JVM. Il JDK offre tutto l'occorrente per lavorare in modo completo. Infatti implementa una suite di applicazioni, come un compilatore, una JVM, un formattatore di documentazione, un'altra JVM per interpretare applet, un generatore di file JAR (Java ARchive) e così via.

Si possono scaricare diverse versioni di questo software. Nel periodo in cui è stato realizzato questo testo, la versione più recente è la 5.0, che consigliamo vivamente di scaricare. Tutti gli esercizi e gli esempi contenuti in questo testo sono stati testati con la versione 5.0 di questo software.

1.2.1 Ambienti di sviluppo più complessi

Esistono anche ambienti di sviluppo visuali più complessi che integrano editor, compilatore ed interprete, come JBuilder di Borland, JDeveloper di Oracle, Sun One Studio di Sun stessa. Esistono anche IDE open source molto potenti e gratuiti come NetBeans (http://www.netbeans.org) ed Eclipse (http://www.eclipse.org). In particolare quest'ultimo è consigliamo caldamente come strumento di sviluppo professionale. Ognuno di questi strumenti favorisce di sicuro una velocità di sviluppo maggiore, ma per quanto riguarda il periodo d'apprendimento iniziale è preferibile di certo scrivere tutto il codice senza aiuti. Il rischio è non raggiungere una conoscenza "seria" di Java. Ci è capitato spesso di conoscere persone che programmavano con questi strumenti da anni, senza avere chiari alcuni concetti fondamentali.
Il lettore tenga conto che se inizia a lavorare con uno dei tool di cui sopra, dovrà studiare

non solo Java, ma anche il manuale del tool. Inoltre, stiamo parlando di strumenti che richiedono requisiti minimi di sistema molto alti, e per gli esercizi che verranno proposti in questo manuale tale scelta sembra inopportuna.
EJE è stato creato proprio con l'intento di evitare di utilizzare tali strumenti. Infatti possiede solo alcune delle comodità dei tool di cui sopra e non bisogna "studiarlo".

EJE ha bisogno almeno della versione 1.4 del JDK.

Quindi, iniziamo con il Blocco Note e abituiamoci da subito ad avere a che fare con più finestre aperte contemporaneamente: quelle dell'editor, e quella del prompt dei comandi (i dettagli saranno descritti nei prossimi paragrafi). Subito dopo aver provato il Blocco Note, sarà possibile utilizzare EJE per valutarne l'utilità e la semplicità d'utilizzo.

1.3 Struttura del JDK

Il JDK (o J2SE) è formato da diverse cartelle:

- **bin**: contiene tutti i file eseguibili del JDK, ovvero "javac", "java", "jar", "appletviewer" etc…
- **demo**: contiene varie dimostrazioni di cosa è possibile fare con Java.
- **include** e **lib**: contengono librerie scritte in C e in Java che sono utilizzate dal JDK.
- **jre**: sta per Java Runtime Environment (JRE). Affinché un'applicazione Java risulti eseguibile su una macchina, basta installare solo il JRE. Si tratta della JVM con il supporto per le librerie supportate nella versione corrente di Java. Il JRE viene installato in questa cartella automaticamente, quando viene installato il JDK.

  È necessario l'intero JDK però, se vogliamo sviluppare applicazioni Java, altrimenti per esempio non potremo compilare i nostri file sorgente.

- **docs**: questa cartella deve essere scaricata ed installata a parte (cfr. Appendice B) e contiene la documentazione della libreria standard di Java, più vari tutorial. Come già asserito, risulterà indispensabile.

Inoltre. nella cartella principale, oltre a vari file di servizio (licenza, copyright etc…) ci sarà anche un file di nome "src.zip". Una volta scompattato "src.zip", sarà possibile dare uno sguardo ai file sorgenti (i ".java") della libreria.

La libreria si evolve nel tempo dando vita alle varie versioni di Java. La versione 1.0 supportava circa 700 classi (bastava scaricare meno di 3 MB), la versione 1.4 oltre 4.700 (e bisogna scaricare circa 40 MB, più altri 35 di documentazione)! Quindi oltre a migliorare le tecnologie e le prestazioni, Sun aggiorna le librerie (dichiarando "deprecato" tutto ciò che è stato migliorato da altro). Lo sviluppatore deve sempre aggiornarsi, se non per passione, perché le novità potrebbero semplificargli il lavoro.

1.3.1 Passi dello sviluppatore

Dopo aver installato correttamente il JDK e impostato opportunamente le eventuali variabili di ambiente (cfr. Appendice B), saremo pronti per scrivere la nostra prima applicazione (prossimo paragrafo).
In generale dovremo eseguire i seguenti passi:

1. **Scrittura del codice**: scriveremo il codice sorgente della nostra applicazione utilizzando un editor. Come già detto precedentemente, Notepad (Blocco Note) di Windows va benissimo (non esageriamo… va benino…), per adesso.
2. **Salvataggio**: salveremo il nostro file con suffisso `.java` .

   Se il lettore utilizza Notepad bisogna salvare il file chiamandolo `nomeFile.java` ed includendo il nome tra virgolette, come in `"nomeFile.java"`. L'utilizzo delle virgolette non è invece necessario per salvare file con EJE. Non è necessario neanche specificare il suffisso `.java` di default.

3. **Compilazione**: una volta ottenuto il nostro file Java, dobbiamo aprire un Prompt di Dos (prompt dei comandi). Da questo prompt dobbiamo spostarci (consultare l'appendice A se non si è in grado) nella cartella in cui è stato salvato il nostro file sorgente e compilarlo tramite il comando:

   ```
   javac nomeFile.java
   ```

   Se la compilazione ha esito positivo, verrà creato un file chiamato "nomeFile.class". In questo file, come abbiamo già detto, ci sarà la traduzione in bytecode del file sorgente.

4. **Esecuzione**: a questo punto potremo mandare in esecuzione il programma invocando l'interpretazione della Java Virtual Machine. Basta scrivere dal prompt Dos il seguente comando:

   java nomeFile

   (notare che "nomeFile" è senza suffissi).
   L'applicazione, a meno di errori di codice, verrà eseguita dalla JVM.

1.4 Primo approccio al codice

Diamo subito uno sguardo alla classica applicazione "Hello World". Trattasi del tipico primo programma che rappresenta il punto di partenza dell'apprendimento di ogni linguaggio di programmazione.
In questo modo inizieremo a familiarizzare con la sintassi e con qualche concetto fondamentale come quello di classe e di metodo.
Avvertiamo il lettore che inevitabilmente qualche punto rimarrà oscuro, e che bisognerà dare per scontate alcuni parti di codice.
Vedremo anche come compilare e come mandare in esecuzione il nostro "miniprogramma". Il fine è stampare a video il messaggio "Hello World!".
Segue il listato:

```
1 public class HelloWorld
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

Questo programma deve essere salvato esattamente col nome della classe, prestando attenzione anche alle lettere maiuscole o minuscole. Tale condizione è necessaria per mandare in esecuzione l'applicazione. Il file che conterrà il listato appena presentato dovrà quindi chiamarsi "HelloWorld.java".
Attenzione: i numeri non fanno parte del codice ma ci saranno utili per la sua analisi (non bisogna scriverli).

Sconsigliamo il "copia– incolla" del codice. Almeno per i primi tempi, si cerchi di scrivere tutto il codice possibile. Consigliamo al lettore di scrivere riga dopo riga dopo averne letto la seguente analisi.

1.5 Analisi del programma "HelloWorld"

Analizziamo quindi il listato precedente riga per riga:

Riga 1:

```
public class HelloWorld
```

Dichiarazione della classe `HelloWorld`. Come vedremo, ogni applicazione Java è costituita da classi. Questo concetto fondamentale sarà trattato in dettaglio nel prossimo modulo.

> **E' da sottolineare subito che tutto il codice scritto in applicazioni Java, a parte poche eccezioni (le importazioni di librerie e le dichiarazioni d'appartenenza ad un package), è sempre incluso all'interno della definizione di qualche classe (o strutture equivalenti come le interfacce). Abituiamoci anche ad anteporre la parola chiave `public` alla dichiarazione delle nostre classi; nel modulo 9 capiremo il perché.**

Riga 2:

```
{
```

Questa parentesi graffa aperta indica l'inizio della classe `HelloWorld`, che si chiuderà alla riga 7 con una corrispondenteparentesi graffa chiusa. Il blocco di codice compreso da queste due parentesi definisce la classe `HelloWorld`.

> **Sulle tastiere italiane non esiste un tasto per stampare le parentesi graffe. Possiamo però ottenerne la scrittura in diversi modi, di seguito i più utilizzati:**
> **- tenendo premuto il tasto ALT e scrivendo 123 con i tasti numerici che si trovano sulla destra della vostra tastiera, per poi rilasciare l'ALT;**
> **- tenendo premuti i tasti CONTROL - SHIFT - ALT, e poi il tasto con il simbolo della parentesi quadra aperta "[".**

Riga 3:

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

Questa riga è bene memorizzarla da subito, poiché deve essere definita in ogni applicazione Java. Trattasi della dichiarazione del metodo `main()`. In Java, il termine "metodo" è sinonimo di "azione" (i metodi saranno trattati in dettaglio nel prossimo

modulo). In particolare il metodo `main()` definisce il punto di partenza dell'esecuzione di ogni programma. La prima istruzione che verrà quindi eseguita in fase di esecuzione, sarà quella che la JVM troverà subito dopo l'apertura del blocco di codice che definisce questo metodo.
Oltre alla parola "`main`", la riga 3 contiene altre parole di cui studieremo in dettaglio il significato nei prossimi capitoli. Purtroppo, come già anticipato, quando si inizia a studiare un linguaggio ad oggetti come Java, è impossibile toccare un argomento senza toccarne tanti altri. Per adesso il lettore si dovrà accontentare della seguente tabella:

`public`	Modificatore del metodo. I modificatori sono utilizzati in Java come nel linguaggio umano sono utilizzati gli aggettivi. Se si antepone un modificatore alla dichiarazione di un elemento Java (un metodo, una variabile, una classe etc...), questo cambierà in qualche modo (a seconda del significato del modificatore) le sue proprietà. In questo caso, trattasi di uno specificatore d'accesso che rende di fatto il metodo accessibile anche al di fuori della classe in cui è stato definito.
`static`	Altro modificatore del metodo. La definizione di `static` è abbastanza complessa. Per ora il lettore si accontenti di sapere che è essenziale per la definizione del metodo `main()`.
`void`	E' il tipo di ritorno del metodo. Significa "vuoto" e quindi questo metodo non restituisce nessun tipo di valore. Il metodo `main()` non deve mai avere un tipo di ritorno diverso da `void`.
`main`	Trattasi ovviamente del nome del metodo (detto anche identificatore del metodo).
`(String args[])`	Alla destra dell'identificatore di un metodo, si definisce sempre una coppia di parentesi tonde che racchiude opzionalmente una lista di parametri (detti

	anche argomenti del metodo). Il metodo `main()`, in ogni caso, vuole sempre come parametro un array di stringhe (agli array troveremo dedicata nel terzo modulo un'intera unità didattica). Notare che `args` è l'identificatore (nome) dell'array, ed è l'unica parola che può variare nella definizione del metodo `main()`, anche se per convenzione si utilizza sempre `args`.

Riga 4:

```
{
```

Questa parentesi graffa indica l'inizio del metodo `main()`, che si chiuderà alla riga 6 con una parentesi graffa chiusa. Il blocco di codice compreso tra queste due parentesi definisce il metodo.

Riga 5:

```
System.out.println("Hello World!");
```

Questo comando stamperà a video la stringa "Hello World!". Anche in questo caso, giacché dovremmo introdurre argomenti per i quali il lettore non è ancora maturo, preferiamo rimandare la spiegazione dettagliata di questo comando ai prossimi capitoli. Per ora ci basterà sapere che stiamo invocando un metodo appartenente alla libreria standard di Java che si chiama `println()`, passandogli come parametro la stringa che dovrà essere stampata.

Riga 6:

```
}
```

Questa parentesi graffa chiusa bilancia l'ultima che è stata aperta, ovvero chiude il blocco di codice che definisce il metodo `main()`.

Sulle tastiere italiane non esiste un tasto per stampare le parentesi graffe. Possiamo però ottenerne la scrittura in diversi modi, di seguito i più utilizzati:
- tenendo premuto il tasto ALT e scrivendo 125 con i tasti numerici che si trovano sulla destra della vostra tastiera, per poi rilasciare l'ALT;
- tenendo premuti i tasti Control - Shift - Alt, e poi il tasto con il simbolo della parentesi quadra chiusa "]".

Riga 7:

```
}
```

Questa parentesi graffa invece chiude il blocco di codice che definisce la classe `HelloWorld`.

1.6 Compilazione ed esecuzione del programma HelloWorld

Una volta riscritto il listato sfruttando un text editor (supponiamo il Notepad di Windows), dobbiamo salvare il nostro file in una cartella di lavoro, chiamata ad esempio "`CorsoJava`".

> **Se il lettore ha deciso di utilizzare il Notepad, presti attenzione nel momento del salvataggioad includere il nome del file (che ricordiamo deve essere obbligatoriamente `HelloWorld.java`) tra virgolette: `"HelloWorld.java"`. Tutto ciò si rende necessario perché Notepad è un editor di testo generico, non un editor per Java, e quindi tende a salvare i file con il suffisso".txt".**
> **L'utilizzo delle virgolette non è necessario per salvare file con EJE, che è invece un editor per Java. Non è necessario neanche specificare il suffisso `.java` di default.**

A questo punto possiamo iniziare ad aprire un prompt di Dos e spostarci all'interno della nostra cartella di lavoro. Dopo essersi accertati che il file `HelloWorld.java` esiste, possiamo passare alla fase di compilazione. Se lanciamo il comando:

```
javac HelloWorld.java
```

mandiamo il nostro file sorgente in input al compilatore che ci mette a disposizione il JDK. Se al termine della compilazione non ci viene fornito nessun messaggio d'errore, vuol dire che la compilazione ha avuto successo.
A questo punto possiamo notare che nella nostra cartella di lavoro è stato creato un file di nome `HelloWorld.class`. Questo è appunto il file sorgente tradotto in bytecode, pronto per essere interpretato dalla JVM.
Se quindi lanciamo il comando:

```
java HelloWorld
```

il nostro programma, se non sono lanciate eccezioni dalla JVM, sarà mandato in esecuzione, stampando il tanto sospirato messaggio.

Si consiglia di ripetere questi passi con le semplificazioni che ci offre EJE.

1.7 Possibili problemi in fase di compilazione ed esecuzione

Di solito, nei primi tempi, gli aspiranti programmatori Java ricevono spesso messaggi apparentemente misteriosi da parte del compilatore e dell'interprete Java. Non bisogna scoraggiarsi! Bisogna avere pazienza e imparare a leggere i messaggi che vengono restituiti. Inizialmente può sembrare difficile, ma in breve tempo ci si accorge che gli errori che si commettono sono spesso gli stessi…

1.7.1 Possibili messaggi di errore in fase di compilazione

1.

```
javac: Command not found
```

In questo caso non è il compilatore che ci sta segnalando un problema, bensì è lo stesso sistema operativo che non riconosce il comando "javac", il quale dovrebbe chiamare il compilatore del JDK. Probabilmente quest'ultimo non è stato installato correttamente. Un tipico problema è di non aver impostato la variabile d'ambiente PATH (cfr. Appendice B).

```
HelloWorld. java: 6: Method printl( java.lang.String) not found in class
java.io.PrintStream.
System.out.printl (" Hello World!");
                  ^
```

(JDK fino alla versione 1.2)

Oppure:

```
HelloWorld.java:6: cannot resolve symbol
symbol : method printl (java.lang.String)
location: class java.io.PrintStream
System.out.printl(" Hello World!");
                  ^
```

(JDK versione 1.3 o successive)
Se riceviamo questo messaggio, abbiamo semplicemente scritto "`printl`" in luogo di "`println`". Il compilatore non può da solo accorgersi che è stato

semplicemente un errore di battitura, ed allora ci ha segnalato che il metodo `printl()` non è stato trovato. In fase di debug, è sempre bene prendere coscienza dei messaggi di errore che ci fornisce il compilatore, tenendo ben presente però che ci sono limiti a questi messaggi ed alla comprensione degli errori da parte del compilatore stesso.

2.
```
HelloWorld.java:1: 'class' or 'interface' expected
Class HelloWorld
^
```
(qualsiasi versione del JDK)
In questo caso invece avremo scritto class con lettera maiuscola e quindi la JVM ha richiesto esplicitamente una dichiarazione di classe (o di interfaccia, concetto che chiariremo più avanti). "Class", lo ripetiamo, non è la stessa cosa di "class" in Java.

3.
```
HelloWorld.java:6: ';' expected
System.out.println(" Hello World!")                    ^
```
(qualsiasi versione del JDK)
In questo caso il compilatore ha capito subito che abbiamo dimenticato un punto e virgola, che serve a concludere ogni statement (istruzione). Purtroppo il nostro compilatore non sarà sempre così preciso. In alcuni casi, se dimentichiamo un punto e virgola, o peggio dimentichiamo di chiudere un blocco di codice con una parentesi graffa, il compilatore ci potrebbe segnalare l'esistenza di errori inesistenti in successive righe di codice.

1.7.2 Possibili messaggi relativi alla fase di interpretazione

Solitamente in questa fase vengono lanciate dalla JVM quelle che vengono definite "eccezioni". Le eccezioni saranno trattate nel Modulo 10.

1.
```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Se questo è il messaggio di risposta ad un tentativo di mandare in esecuzione un programma, forse abbiamo definito male il metodo `main()`. Probabilmente

abbiamo dimenticato di dichiararlo `static` o `public`, oppure abbiamo scritto male la lista degli argomenti (che deve essere `String args[]`), o magari non abbiamo chiamato "`main`" il metodo.
Un altro classico motivo che potrebbe provocare questo messaggio è il dichiarare il nome del file con un nome diverso da quello della classe.

2.

```
Exception in thread "main" java.lang.NoClassDefFoundError: Helloworld (wrong
name
: HelloWorld)
at java.lang.ClassLoader.defineClass0(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java:486)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:11
1)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
at java.lang.ClassLoader.loadClass(ClassLoader.java:297)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
at java.lang.ClassLoader.loadClass(ClassLoader.java:253)
at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:313)
```

Non c'è da spaventarsi! Questo è semplicemente il caso in cui la JVM non trova la classe Java contenente il bytecode da interpretare. Infatti, come si può notare, è stato lanciato il comando "`java Helloworld`" invece di "`java HelloWorld`". Non sempre quindi messaggi lunghi richiedono radicali correzioni!

E' possibile che abbiate installato sulla vostra macchina, programmi che impostano la variabile del sistema operativo CLASSPATH (per esempio Oracle, o XMLSpy). In tal caso, non è più possibile eseguire i nostri file con il comando:

java HelloWorld

ma bisogna aggiungere la seguente opzione:

java –classpath . HelloWorld

Con EJE invece è possibile compilare e lanciare l'applicativo, senza nessun lavoro extra.

Riepilogo

In questo modulo abbiamo definito Java come linguaggio e come tecnologia, ed abbiamo descritto il linguaggio mediante le sue caratteristiche. Abbiamo inoltre descritto l'ambiente di sviluppo ed abbiamo imparato a compilare e mandare in esecuzione una semplice applicazione Java. Abbiamo approcciato un piccolo programma, per avere un'idea di alcuni concetti fondamentali. Infine abbiamo anche posto l'accento sull'importanza di leggere i messaggi di errore che il compilatore o la JVM mostrano allo sviluppatore.

Esercizi modulo 1

Esercizio 1.a)

Digitare, salvare, compilare ed eseguire il programma `HelloWorld`. Consigliamo al lettore di eseguire questo esercizio due volte: la prima volta utilizzando il Notepad e il prompt Dos, e la seconda utilizzando EJE.

> **EJE permette di inserire parti di codice preformattate tramite il menu "Inserisci". Per eventuali problemi con EJE è possibile consultare l'Appendice C.**

Esercizio 1.b)
Caratteristiche di Java, Vero o Falso:

1. Java è il nome di una tecnologia e contemporaneamente il nome di un linguaggio di programmazione.
2. Java è un linguaggio interpretato ma non compilato.
3. Java è un linguaggio veloce ma non robusto.
4. Java è un linguaggio difficile da imparare perché in ogni caso obbliga ad imparare l'object orientation.
5. La Java Virtual Machine è un software che supervisiona il software scritto in Java.
6. La JVM gestisce la memoria automaticamente mediante la Garbage Collection.
7. L'indipendenza dalla piattaforma è una caratteristica poco importante.
8. Java non è adatto per scrivere un sistema sicuro.
9. La Garbage Collection garantisce l'indipendenza dalla piattaforma.
10. Java è un linguaggio gratuito che raccoglie le caratteristiche migliori di altri linguaggi, e ne esclude quelle ritenute peggiori e più pericolose.

Esercizio 1.c)
Codice Java, Vero o Falso:

1. La seguente dichiarazione del metodo `main()` è corretta:
   ```
   public static main(String argomenti[]) {...}
   ```

2. La seguente dichiarazione del metodo `main()` è corretta:
   `public static void Main(String args[]){...}`
3. La seguente dichiarazione del metodo main() è corretta:
   `public static void main(String argomenti[]) {...}`
4. La seguente dichiarazione del metodo main() è corretta:
   `public static void main(String Argomenti[]) {...}`
5. La seguente dichiarazione di classe è corretta:
   `public class {...}`
6. La seguente dichiarazione di classe è corretta:
   `public Class Auto {...}`
7. La seguente dichiarazione di classe è corretta:
   `public class Auto {...}`
8. E' possibile dichiarare un metodo al di fuori del blocco di codice che definisce una classe.
9. Il blocco di codice che definisce un metodo è delimitato da due parentesi tonde.
10. Il blocco di codice che definisce un metodo è delimitato da due parentesi quadre.

Esercizio 1.d)
Ambiente e processo di sviluppo, Vero o Falso:

1. La JVM è un software che simula un hardware.
2. Il bytecode è contenuto in un file con suffisso ".class".
3. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, mandarlo in esecuzione ed infine compilarlo.
4. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, compilarlo ed infine mandarlo in esecuzione.
5. Il nome del file che contiene una classe Java deve coincidere con il nome della classe, anche se non si tiene conto delle lettere maiuscole o minuscole.
6. Una volta compilato un programma scritto in Java è possibile eseguirlo su un qualsiasi sistema operativo che abbia una JVM.
7. Per eseguire una qualsiasi applicazione Java basta avere un browser.
8. Il compilatore del JDK viene invocato tramite il comando "javac" e la JVM viene invocata tramite il comando "java".
9. Per mandare in esecuzione un file che si chiama "pippo.class", dobbiamo lanciare il seguente comando dal prompt: "java pippo.java".
10. Per mandare in esecuzione un file che si chiama "pippo.class", dobbiamo lanciare il seguente comando dal prompt: "java pippo.class".

Soluzioni esercizi modulo 1

Esercizio 1.a)

Non è fornita una soluzione per questo esercizio.

Esercizio 1.b)
Caratteristiche di Java, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Falso**
8. **Falso**
9. **Falso**
10. **Vero**

Per avere spiegazioni sulle soluzioni dell'esercizio 1.b sopra riportate, il lettore può rileggere il paragrafo 1.1.

Esercizio 1.c)
Codice Java, Vero o Falso:

1. **Falso** manca il tipo di ritorno (`void`).
2. **Falso** l'identificatore dovrebbe iniziare con lettera minuscola (`main`).
3. **Vero**
4. **Vero**
5. **Falso** manca l'identificatore.
6. **Falso** la parola chiave si scrive con lettera iniziale minuscola (`Class`).
7. **Vero**
8. **Falso**
9. **Falso** le parentesi sono graffe.

10. **Falso** le parentesi sono graffe.

Esercizio 1.d)
Ambiente e processo di sviluppo, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso** bisogna prima compilarlo per poi mandarlo in esecuzione.
4. **Vero**
5. **Falso** bisogna anche tenere conto delle lettere maiuscole o minuscole.
6. **Vero**
7. **Falso** un browser basta solo per eseguire applet.
8. **Vero**
9. **Falso** il comando giusto è java pippo.
10. **Falso** il comando giusto è java pippo.

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper definire il linguaggio di programmazione Java e le sue caratteristiche. (unità 1.1)	☐	
Interagire con l'ambiente di sviluppo: il Java Development Kit. (unità 1.2, 1.5, 1.6)	☐	
Saper digitare, compilare e mandare in esecuzione una semplice applicazione. (unità 1.3, 1.4, 1.5, 1.6)	☐	

Note:

2

Complessità: media

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.1, 2.2, 2.3, 2.4, 2.5).
2. Saper dichiarare una classe (unità 2.1).
3. Istanziare oggetti da una classe (unità 2.2).
4. Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4).
5. Dichiarare ed invocare un metodo (unità 2.3).
6. Saper dichiarare ed inizializzare una variabile (unità 2.4).
7. Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4).
8. Dichiarare ed invocare un metodo costruttore (unità 2.5).
9. Saper accennare alla definizione di costruttore di default (unità 2.5).
10. Saper accennare alla definizione di package (unità 2.6).

2 Componenti fondamentali di un programma Java

Da questo modulo parte lo studio del linguaggio vero e proprio. Dapprima familiarizzeremo con i concetti fondamentali che sono alla base di un programma Java, sia in maniera teorica che pratica. Anche in questo caso, non daremo nulla per scontato e quindi anche il lettore che non ha mai programmato dovrebbe poter finalmente iniziare. I lettori che invece hanno precedenti esperienze con la programmazione procedurale, troveranno profonde differenze che è bene fissare da subito. Il consiglio per questi ultimi è di non ancorarsi troppo a quello che già si conosce. È inutile per esempio forzare una classe ad avere il ruolo che poteva avere una funzione nella programmazione procedurale. Meglio far finta di partire da zero…

2.1 Componenti fondamentali di un programma Java

Ecco una lista di concetti che sono alla base della conoscenza di Java:

1) Classe

2) Oggetto
3) Membro:
    a. Attributo (variabile membro)
    b. Metodo (metodo membro)
4) Costruttore
5) Package

Di seguito vengono fornite al lettore le nozioni fondamentali per approcciare nel modo migliore le caratteristiche del linguaggio che saranno presentate a partire dal prossimo modulo. Prima, però, introduciamo una convenzione ausiliaria per chi non ha confidenza con la programmazione ad oggetti.

2.1.1 Convenzione per la programmazione Java

L'apprendimento di un linguaggio orientato agli oggetti può spesso essere molto travagliato, specialmente se si possiedono solide "radici procedurali". Abbiamo già accennato al fatto che Java, a differenza del C++, in pratica "ci obbliga" a programmare ad oggetti. Non ha senso imparare il linguaggio senza sfruttare il supporto che esso offre alla programmazione ad oggetti. Chi impara il C++, ha un vantaggio illusorio. Può infatti permettersi di continuare ad utilizzare il paradigma procedurale, e quindi imparare il linguaggio insieme ai concetti object oriented, ma mettendo inizialmente questi ultimi da parte. Infatti, in C++, è comunque possibile creare funzioni e programmi chiamanti. In realtà si tratta di un vantaggio a breve termine, che si traduce inevitabilmente in uno svantaggio a lungo termine.
Con Java lo scenario si presenta più complesso. Inizieremo direttamente a creare applicazioni che saranno costituite da un certo numero di classi. Non basterà dare per scontato che "Java funziona così". Bisognerà invece cercare di chiarire ogni singolo punto poco chiaro, in modo tale da non creare troppa confusione, che, a lungo andare, potrebbe scoraggiare. Per fortuna Java è un linguaggio con caratteristiche molto chiare e coerenti, e questo ci sarà di grande aiuto. Alla fine l'aver appreso in modo profondo determinati concetti dovrebbe regalare al lettore maggiore soddisfazione.
Proviamo da subito ad approcciare il codice in maniera schematica, introducendo una convenzione.
In questo testo distingueremo due ambiti quando sviluppiamo un'applicazione Java:

1. **Ambito del compilatore** (o **compilativo**, o **delle classi**)
2. **Ambito della Java Virtual Machine** (o **esecutivo**, o **degli oggetti**)

All'interno dell'ambito del compilatore codificheremo la parte strutturale della nostra applicazione.
L'ambito esecutivo invece ci servirà per definire il flusso di lavoro che dovrà essere eseguito.
Distinguere questi due ambiti sarà utile per comprendere i ruoli che dovranno avere all'interno delle nostre applicazioni i concetti introdotti in questo modulo, anche se questa distinzione può essere considerata improduttiva dal lettore con precedenti esperienze di programmazione object oriented.

2.2 Le basi della programmazione object oriented: classi ed oggetti

I concetti di classe ed oggetto sono strettamente legati. Esistono infattidefinizioni "ufficiali" derivanti dalla teoria della programmazione orientata agli oggetti, che di seguito presentiamo:

- **Definizione 1**:
  una **classe** è un'astrazione indicante un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità.
- **Definizione 2**:
  un **oggetto** è un'istanza (ovvero, una creazione fisica) di una classe.

A questo punto, il lettore più "matematico" sicuramente avrà le idee un po' confuse. Effettivamente definire il concetto di classe tramite la definizione di oggetto, e il concetto di oggetto tramite la definizione di classe, non è certo il massimo della chiarezza! La situazione però è molto meno complicata di quella che può sembrare. Ogni concetto che fa parte della teoria dell'Object Orientation, infatti, esiste anche nel mondo reale. Questa teoria è nata, proprio per soddisfare l'esigenza umana di interagire con il software in maniera "naturale" (cfr. Paragrafo 5.1). Si tratterà quindi solo di associare la giusta idea al nuovo termine.
Passiamo subito ad un esempio:

```
public class Punto
{
    public int x;
    public int y;
}
```

Con Java è possibile creare codice per astrarre un qualsiasi concetto del mondo reale. Abbiamo appena definito una classe `Punto`. Evidentemente lo scopo di questa classe è definire il concetto di punto (a due dimensioni), tramite la definizione delle sue coordinate su di un piano cartesiano. Le coordinate sono state astratte mediante due attributi (variabili) dichiarati di tipo intero (`int`) e pubblici (`public`), chiamati `x` ed `y`.
Questa classe potrebbe costituire una parte di un'applicazione Java, magari un'applicazione di disegno…
Possiamo salvare questo listato in un file chiamato "Punto.java", ed anche compilarlo tramite il comando:

```
javac Punto.java
```

otterremo così il file "Punto.class".
Non possiamo però mandarlo in esecuzione tramite il comando:

```
java Punto
```

perchè otterremmo un messaggio di errore. Infatti, in questa classe, è assente il metodo `main()` (cfr. modulo precedente), che abbiamo definito come punto di partenza dell'esecuzione di ogni applicazione Java. Se lanciassimo il comando:

```
java Punto
```

la JVM cercherebbe questo metodo per eseguirlo, e non trovandolo terminerebbe istantaneamente l'esecuzione.
Ciò è assolutamente in linea con la definizione di classe (def. 1). Infatti, se volessimo trovare nel mondo reale un sinonimo di classe, dovremmo pensare a termini come “idea”, “astrazione”, “concetto”, “modello” “definizione”. Quindi, definendo una classe `Punto` con codice Java, abbiamo solo definito all'interno del nostro futuro programma il concetto di punto, secondo la nostra interpretazione e nel contesto in cui ci vogliamo calare (per esempio un programma di disegno).
Con il codice scritto finora, in realtà, non abbiamo ancora definito nessun punto vero e proprio (per esempio il punto di coordinate x=5 e y=6). Quindi, come nel mondo reale, se non esistono punti concreti (ma solo la definizione di punto) non può accadere nulla che abbia a che fare con un punto e così, nel codice Java, la classe `Punto` non è "eseguibile" da sola!
Occorre quindi definire gli "oggetti punto", ovvero le creazioni fisiche realizzate a partire dalla definizione data dalla classe.

Nel contesto della programmazione ad oggetti, una classe dovrebbe quindi limitarsi a definire che struttura avranno gli oggetti che da essa saranno istanziati. "Istanziare", come abbiamo già affermato, è il termine object oriented che si usa in luogo di "creare fisicamente", ed il risultato di un'istanziazione viene detto "oggetto".
Per esempio, istanzieremo allora oggetti dalla classe `Punto`, creando un'altra classe che contiene un metodo `main()`*:*

```
1    public class Principale
2    {
3        public static void main(String args[])
4        {
5            Punto punto1;
6            punto1 = new Punto();
7            punto1.x = 2;
8            punto1.y = 6;
9            Punto punto2 = new Punto();
10           punto2.x = 0;
11           punto2.y = 1;
12           System.out.println(punto1.x);
13           System.out.println(punto1.y);
14           System.out.println(punto2.x);
15           System.out.println(punto2.y);
16       }
17    }
```

Analizziamo la classe `Principale`.
Alla riga 5 è dichiarato un oggetto di tipo `Punto` che viene chiamato `punto1`. Ma è solamente alla riga 6 che avviene l'istanziazione (creazione) della classe `Punto`. La parola chiave `new`, di fatto, istanzia la classe `Punto`, a cui viene assegnato il nome `punto1`. Dalla riga 6 in poi è possibile utilizzare l'oggetto `punto1`. Precisamente, alle righe 7 e 8, assegniamo alle coordinate `x` e `y` del `punto1`, rispettivamente i valori interi `2` e `6`. In pratica, sfruttando la definizione che ci ha fornito la classe `Punto`, abbiamo creato un oggetto di tipo `Punto`, che è individuato dal nome `punto1`. Notiamo anche, l'utilizzo dell'operatore "dot" (che in inglese significa "punto", ma nel senso del simbolo di punteggiatura) per accedere alle variabili `x` e `y`.
Ricapitolando, abbiamo prima dichiarato l'oggetto alla riga 5, l'abbiamo istanziato alla riga 6, e l'abbiamo utilizzato (impostando le sue variabili) alle righe 7 e 8.

Alla riga 9 poi, abbiamo dichiarato ed istanziato con un'unica riga di codice un altro oggetto dalla classe `Punto`, chiamandolo `punto2`. Abbiamo poi impostato le coordinate di quest'ultimo a `0` e `1`. Abbiamo infine stampato le coordinate di entrambi i punti.
Le definizioni di classi ed oggetto dovrebbero essere un po' più chiare: la classe è servita per definire come saranno fatti gli oggetti. L'oggetto rappresenta una realizzazione fisica della classe.
In questo esempio, abbiamo istanziato due oggetti diversi da una stessa classe. Entrambi questi oggetti sono punti, ma evidentemente sono punti diversi (si trovano in diverse posizioni sul riferimento cartesiano).

Questo ragionamento, in fondo, è già familiare a chiunque, giacché derivante dal mondo reale che ci circonda. L'essere umano, per superare la complessità della realtà, raggruppa gli oggetti in classi. Per esempio, nella nostra mente esiste il modello definito dalla classe `Persona`. Ma nella realtà esistono miliardi di oggetti di tipo `Persona`, ognuno dei quali ha caratteristiche uniche.

Comprendere le definizioni 1 e 2, ora, non dovrebbe rappresentare più un problema.

2.2.1 Osservazione importante sulla classe Punto

Nell'esempio precedente abbiamo potuto commentare la definizione di due classi. Per la prima (la classe `Punto`), abbiamo sottolineato la caratteristica di rappresentare "un dato". E' da considerarsi una parte strutturale dell'applicazione, e quindi svolge un ruolo essenziale nell'ambito compilativo. Nell'ambito esecutivo, la classe `Punto` non ha un ruolo attivo. Infatti sono gli oggetti istanziati da essa che influenzano il flusso di lavoro del programma.
Questo può definirsi come il caso standard. In un'applicazione object oriented, una classe dovrebbe limitarsi a definire la struttura comune di un insieme di oggetti, e non dovrebbe mai "possedere" né variabili né metodi. Infatti, la classe `Punto` non possiede le variabili `x` e `y`, bensì, dichiarando le due variabili, definisce gli oggetti che da essa saranno istanziati, come possessori di quelle variabili. Si noti che mai all'interno del codice dell'ambito esecutivo è presente un'istruzione come:

```
    Punto.x
```

ovvero

```
NomeClasse.nomeVariabile
```

(che tra l'altro produrrebbe un errore in compilazione) bensì:

```
punto1.x
```

ovvero

```
nomeOggetto.nomeVariabile
```

L'operatore "." è sinonimo di "appartenenza". Sono quindi gli oggetti a possedere le variabili dichiarate nella classe (che tra l'altro verranno dette "variabili d'istanza", ovvero dell'oggetto).
Infatti, i due oggetti istanziati avevano valori diversi per `x` e `y`, il che significa che l'oggetto `punto1` ha la sua variabile `x` e la sua variabile `y`, mentre l'oggetto `punto2` ha la sua variabile `x` e la sua variabile `y`.
Le variabili di `punto1` sono assolutamente indipendenti dalle variabili di `punto2`.
Giacché le classi non hanno membri (variabili e metodi), non eseguono codice e non hanno un ruolo nell'ambito esecutivo, e per quanto visto sono gli oggetti protagonisti assoluti di quest'ambito.

2.2.2 Osservazione importante sulla classe Principale

Come spesso accade quando si approccia Java, appena definita una nuova regola, subito si presenta l'eccezione (che se vogliamo dovrebbe confermarla!). Puntiamo infatti la nostra attenzione sulla classe `Principale`. E' una classe che esegue codice contenuto all'interno dell'unico metodo `main()`, che per quanto detto è assunto per default quale punto di partenza di un'applicazione Java.
In qualche modo, infatti, i creatori di Java dovevano stabilire un modo per far partire l'esecuzione di un programma. La scelta è stata compiuta in base ad una questione pratica e storica: un'applicazione scritta in C o C++ ha come punto di partenza per default un programma chiamante che si chiama proprio `main()`. In Java i programmi chiamanti non esistono, ma esistono i metodi, ed in particolare i metodi statici, ovvero dichiarati con un modificatore `static`.
Questo modificatore sarà trattato in dettaglio nel modulo 9. Per ora limitiamoci a sapere che un membro dichiarato statico appartiene alla classe e che tutti gli oggetti istanziati da essa condivideranno i membri statici.

Concludendo, giacché la classe `Principale` contiene il metodo `main()`, può eseguire codice.

In ogni applicazione Java deve esserci una classe che contiene il metodo `main()`. Questa classe dovrebbe avere il nome dell'applicazione stessa, astraendo il flusso di lavoro che deve essere eseguito. In linea teorica, quindi, la classe contenente il metodo `main()` non dovrebbe contenere altri membri.
E' anche bene che il lettore eviti di utilizzare il modificatore `static` a breve termine nelle proprie applicazioni ed aspetti che "i tempi maturino". Meglio non buttare via tempo prezioso. L'accenno fatto ad un argomento complesso come `static` ha l'unico scopo di fare un po' di luce su un punto oscuro del discorso.

2.2.3 Un'altra osservazione importante

Abbiamo presentato l'esempio precedente per la sua semplicità. È infatti "matematico" pensare ad un punto sul piano cartesiano come formato da due coordinate chiamate `x` e `y`. Ci sono concetti del mondo reale la cui astrazione non è così standard.
Per esempio, volendo definire l'idea (astrazione) di un'auto, dovremmo parlare delle caratteristiche e delle funzionalità comuni ad ogni auto. Ma, se confrontiamo l'idea (ovvero la classe) di auto che definirebbe un intenditore di automobili (supponiamo si chiami Pippo) con quella di una persona che non ha neanche la patente (supponiamo si chiami Pluto), ci sarebbero significative differenze. Pluto potrebbe definire un'auto come "un mezzo di trasporto (quindi che si può muovere), con quattro ruote ed un motore", ma Pippo potrebbe dare una definizione alternativa molto più dettagliata. Pippo potrebbe introdurre caratteristiche come assicurazione, telaio, modello, pneumatici etc…
Entrambe queste definizioni potrebbero essere portate nella programmazione sotto forma di classi.
Per convenienza "ragioneremo come Pluto", creando una classe dichiarante come attributo un intero chiamato `numeroRuote` inizializzato a `4`. Inoltre per semplicità definiamo un'altra variabile intera `cilindrata` (in luogo del motore) ed un metodo che potremmo chiamare `muoviti()`.

Il lettore è rimandato ai prossimi paragrafi per la comprensione e l'utilizzo del concetto di metodo. Per adesso si limiti a sapere che il

metodo servirà a definire una funzionalità che deve avere il concetto che si sta astraendo con la classe.

```
public class Auto
{
    public int numeroRuote = 4;
    public int cilindrata; // quanto vale?
    public void muoviti()
    {
        // implementazione del metodo...
    }
}
```

Ogni auto ha 4 ruote (di solito), si muove ed ha una cilindrata (il cui valore non è definibile a priori). Una Ferrari Testarossa ed una Fiat Cinquecento, hanno entrambe 4 ruote, una cilindrata e si muovono, anche se in modo diverso. La Testarossa e la Cinquecento sono da considerarsi oggetti della classe `Auto`, e nella realtà esisterebbero come oggetti concreti.
Per esercizio lasciamo al lettore l'analisi della seguente classe:

```
1    public class Principale2
2    {
3        public static void main(String args[])
4        {
5            Auto fiat500;
6            fiat500 = new Auto();
7            fiat500.cilindrata = 500;
8            fiat500.muoviti();
9            Auto ferrariTestarossa = new Auto();
10           ferrariTestarossa.cilindrata = 3000;
11           ferrariTestarossa.muoviti();
12       }
13    }
```

2.3 I metodi in Java

Nella definizione di classe, quando si parla di caratteristiche, in pratica ci si riferisce ai dati (variabili e costanti), mentre col termine funzionalità ci si riferisce ai **metodi**.

Abbiamo già accennato al fatto che il termine "metodo" è sinonimo di "azione". Quindi, affinché un programma esegua qualche istruzione, deve contenere metodi. Per esempio è il metodo `main()`, che per default, è il punto di partenza di ogni applicazione Java. Una classe senza metodo `main()`, come la classe `Punto`, non può essere mandata in esecuzione, ma solo istanziata all'interno di un metodo di un'altra classe (nell'esempio precedente, nel metodo `main()` della classe `Principale`).
Il concetto di metodo è quindi anch'esso alla base della programmazione ad oggetti. Senza metodi, gli oggetti non potrebbero comunicare tra loro. Essi possono essere considerati messaggi che gli oggetti si scambiano. Inoltre, i metodi rendono i programmi più leggibili e di più facile manutenzione, lo sviluppo più veloce e stabile, evitano le duplicazioni e favoriscono il riuso del software.

Il programmatore che si avvicina a Java dopo esperienze nel campo della programmazione strutturata, spesso tende a confrontare il concetto di funzione con il concetto di metodo. Sebbene simili nella forma e nella sostanza, bisognerà tener presente, che un metodo ha un ruolo differente rispetto ad una funzione. Nella programmazione strutturata, infatti, il concetto di funzione era alla base. Tutti i programmi erano formati da un cosiddetto programma chiamante e da un certo numero di funzioni. Queste avevano di fatto il compito di risolvere determinati "sottoproblemi" generati da un'analisi di tipo top-down, allo scopo di risolvere il problema generale. Come vedremo più avanti, nella programmazione orientata agli oggetti, i "sottoproblemi" saranno invece risolti tramite i concetti di classe ed oggetto, che a loro volta faranno uso di metodi.

E' bene che il lettore cominci a distinguere nettamente due fasi per quanto riguarda i metodi: dichiarazione e chiamata (ovvero la definizione e l'utilizzo).

2.3.1 Dichiarazione di un metodo

La dichiarazione definisce un metodo. Ecco la sintassi:

```
[modificatori] tipo_di_ritorno nome_del_metodo
([parametri]) {corpo_del_metodo}
```

dove:

- **modificatori**: parole chiave di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche di un metodo. Tutti i modificatori

sono trattati in maniera approfondita nel Modulo 9, ma per ora ci accontenteremo di conoscerne superficialmente solo alcuni. Esempi di modificatori sono `public` e `static`.

- **tipo di ritorno:** il tipo di dato che un metodo potrà restituire dopo essere stato chiamato. Questo potrebbe coincidere sia con un tipo di dato primitivo come un `int`, sia con un tipo complesso (un oggetto) come una stringa (classe `String`). È anche possibile specificare che un metodo non restituisca nulla (`void`).
- **nome del metodo**: identificatore del metodo.**parametri**: dichiarazioni di variabili che potranno essere passate al metodo, e di conseguenza essere sfruttate nel corpo del metodo, al momento della chiamata. Il numero di parametri può essere zero, ma anche maggiore di uno. Se si dichiarano più parametri, le loro dichiarazioni saranno separate da virgole.
- **corpo del metodo**: insieme di istruzioni (statement) che verranno eseguite quando il metodo sarà invocato.

La coppia costituita dal nome del metodo e l'eventuale lista dei parametri viene detta "firma" (in inglese "signature") del metodo.

Per esempio, viene di seguito presentata una classe chiamata `Aritmetica` che dichiara un banale metodo di somma di due numeri :

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return (a + b);
    }
}
```

Si noti che il metodo presenta come modificatore la parola chiave `public`. Si tratta di uno specificatore d'accesso che rende il metodo `somma()` accessibile da altre classi. Precisamente, i metodi di altre classi potranno, dopo aver istanziato un oggetto della classe `Aritmetica`, invocare il metodo `somma()`. Anche per gli specificatori d'accesso approfondiremo il discorso più avanti.
Il tipo di ritorno è un `int`, ovvero, un intero. Ciò significa che questo metodo avrà come ultima istruzione un comando (`return`) che restituirà un numero intero.
Inoltre la dichiarazione di questo metodo evidenzia come lista di parametri due interi

(chiamati `a` e `b`). Questi saranno sommati all'interno del blocco di codice e la loro somma verrà restituita, tramite il comando `return`, come risultato finale. La somma tra due numeri interi non può essere che un numero intero.
Concludendo, la dichiarazione di un metodo definisce quali azioni deve compiere il metodo stesso, quando sarà chiamato.

Si noti che nell'esempio sono anche state utilizzate le parentesi tonde per circondare l'operazione di somma. Anche se in questo caso non erano necessarie per la corretta esecuzione del metodo, abbiamo preferito utilizzarle per rendere più chiara l'istruzione.

2.3.2 Chiamata (o invocazione) di un metodo

Presentiamo un'altra classe "eseguibile" (ovvero contenente il metodo `main()`), che istanzia un oggetto dalla classe `Aritmetica` e chiama il metodo `somma()`:

```
1    public class Uno
2    {
3        public static void main(String args[])
4        {
5            Aritmetica oggetto1 = new Aritmetica();
6            int risultato = oggetto1.somma(5, 6);
7        }
8    }
```

In questo caso notiamo subito che l'accesso al metodo `somma()` è avvenuto sempre tramite l'operatore "dot" come nel caso dell'accesso alle variabili. Quindi, tutti i membri (attributi e metodi) pubblici definiti all'interno di una classe saranno accessibili tramite un'istanza della classe stessa che sfrutta l'operatore dot. La sintassi da utilizzare è la seguente:

```
    nomeOggetto.nomeMetodo();
```

e

```
    nomeOggetto.nomeAttributo;
```

L'accesso al metodo di un oggetto provoca l'esecuzione del relativo blocco di codice. In quest'esempio, quindi, abbiamo definito una variabile intera `risultato` che ha immagazzinato il risultato della somma. Se non avessimo fatto ciò, non avrebbe avuto senso definire un metodo che restituisse un valore, dal momento che non l'avremmo utilizzato in qualche modo! Da notare che avremmo anche potuto aggiungere alla riga 6:

```
  int risultato = oggetto1.somma(5, 6);
```

la riga seguente:

```
   System.out.println(risultato);
```

o equivalentemente sostituire la riga 6 con:

```
   System.out.println(oggetto1.somma(5,6));
```

In questo modo, stampando a video il risultato, avremmo potuto verificare in fase di runtime del programma la reale esecuzione della somma.
Esistono anche metodi che non hanno parametri in input ma, per esempio, un metodo che somma sempre gli stessi numeri e quindi restituisce sempre lo stesso valore:

```
public class AritmeticaFissa
{
    public int somma()
    {
        return (5 + 6);
    }
}
```

oppure metodi che non hanno tipo di ritorno e dichiarano come tipo di ritorno `void` (vuoto) come il metodo `main()`. Infatti il metodo `main`() è il punto di partenza del runtime di un'applicazione Java, quindi non deve restituire un risultato, non venendo chiamato esplicitamente da un altro metodo.
Esistono ovviamente anche metodi che non hanno né parametri in input, né in output (tipo di ritorno `void`), come per esempio un semplice metodo che visualizza a video sempre lo stesso messaggio:

```
public class Saluti
{
    public void stampaSaluto()
    {
        System.out.println("Ciao");
    }
}
```

Segue una banale classe che istanzia un oggetto dalla classe `Saluti` e chiama il metodo `stampaSaluto()`. Come si può notare, se il metodo ha come tipo di ritorno `void`, non bisogna "catturarne" il risultato.

```
1    public class Due
2    {
3        public static void main(String args[])
4        {
5            Saluti oggetto1 = new Saluti();
6            oggetto1.stampaSaluto();
7        }
8    }
```

2.3.3 Varargs

Dalla versione 5 di Java è possibile anche utilizzare metodi che dichiarano come lista di parametri i cosiddetti **variable arguments**, più brevemente noti come **varargs**. In pratica è possibile creare metodi che dichiarano un numero non definito di parametri di un certo tipo. La sintassi è un po' particolare, e fa uso di tre puntini (come i puntini sospensivi), dopo la dichiarazione del tipo. Per esempio, è possibile invocare il seguente metodo:

```
public class AritmeticaVariabile {
    public void somma(int… interi) {
      . . .
    }
}
```

in una qualsiasi dei seguenti modi:

```
AritmeticaVariabile ogg = new
AritmeticaVariabile();ogg.somma();
ogg.somma(1,2);
ogg.somma(1,4,40,27,48,27,36,23,45,67,9,54,66,43);
ogg.somma(1);
```

L'argomento varargs sarà approfondito nel Modulo 6, relativo al polimorfismo, e in dettaglio nel Modulo 18.

2.4 Le variabili in Java

Nella programmazione tradizionale, una **variabile** è una porzione di memoria in cui è immagazzinato un certo tipo di dato. Per esempio, un intero in Java è immagazzinato in 32 bit. I tipi di Java saranno argomento del prossimo capitolo.
Anche per l'utilizzo delle variabili possiamo distinguere due fasi: dichiarazione ed assegnazione. L'assegnazione di un valore ad una variabile è un'operazione che si può ripetere molte volte nell'ambito esecutivo, anche contemporaneamente alla dichiarazione stessa.

2.4.1 Dichiarazione di una variabile:

La sintassi per la dichiarazione di una variabile è la seguente:

```
[modificatori] tipo_di_dato nome_della_variabile [ =
inizializzazione];
```

dove:

- **modificatori**: parole chiavi di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche della variabile.
- **tipo di dato**: il tipo di dato della variabile.
- **nome della variabile**: identificatore della variabile.
- **inizializzazione**: valore di default con cui è possibile valorizzare una variabile.

Segue una classe che definisce in maniera superficiale un quadrato:

```
public class Quadrato
{
```

```
    public int altezza;
    public int larghezza;
    public final int NUMERO_LATI = 4;
}
```

In questo caso abbiamo definito due variabili intere chiamate `altezza` e `larghezza`, ed una costante, `NUMERO_LATI`. Il modificatore `final` infatti (che sarà trattato sarà trattato in dettaglio nel modulo 9), rende una variabile costante nel suo valore.
E' anche possibile definire due variabili dello stesso tipo con un'unica istruzione come nel seguente esempio:

```
public class Quadrato
{
    public int altezza, larghezza;
    public final int NUMERO_LATI = 4;
}
```

In questo caso possiamo notare una maggiore compattezza del codice ottenuta evitando le duplicazioni, ma una minore leggibilità.
La dichiarazione di una variabile è molto importante. La posizione della dichiarazione definisce lo "scope", ovvero la visibilità, ed il ciclo di vita della variabile stessa. In pratica potremmo definire tre diverse tipologie di variabili, basandoci sul posizionamento della dichiarazione:

2.4.2 Variabili d'istanza

Una **variabile** è detta **d'istanza** se è dichiarata in una classe, ma al di fuori di un metodo. Le variabili definite nella classe `Quadrato` sono tutte di istanza. Esse condividono il proprio ciclo di vita con l'oggetto (istanza) cui appartengono. Quando un oggetto della classe `Quadrato` è istanziato, viene allocato spazio per tutte le sue variabili d'istanza che vengono inizializzate ai relativi valori nulli (una tabella esplicativa è presentata nel modulo successivo). Nel nostro caso, le variabili altezza e larghezza saranno inizializzate a zero. Ovviamente `NUMERO_LATI` è una costante esplicitamente inizializzata a `4`, ed il suo valore non potrà cambiare. Una variabile d'istanza smetterà di esistere quando smetterà di esistere l'oggetto a cui appartiene.

2.4.3 Variabili locali

Una **variabile** è detta **locale** (o di **stack**, o **automatica**, o anche **temporanea**) se è dichiarata all'interno di un metodo. Essa smetterà di esistere quando terminerà il metodo. Una variabile di questo tipo, a differenza di una variabile di istanza, non sarà inizializzata ad un valore nullo al momento dell'istanza dell'oggetto a cui appartiene. È buona prassi inizializzare comunque una variabile locale ad un valore di default, nel momento in cui la si dichiara. Infatti il compilatore potrebbe restituire un messaggio di errore laddove ci sia possibilità che la variabile non venga inizializzata al runtime. Nel seguente esempio la variabile `z` è una variabile locale:

```
public int somma(int x, int y)
{
    int z = x + y;
    return z;
}
```

2.4.4 Parametri formali

Le variabili dichiarate all'interno delle parentesi tonde che si trovano alla destra dell'identificatore nella dichiarazione di un metodo, sono dette **parametri** o **argomenti** del metodo.
Per esempio, nella seguente dichiarazione del metodo `somma()` vengono dichiarati due parametri interi `x` e `y`:

```
public int somma(int x, int y)
{
    return (x + y);
}
```

I parametri di un metodo saranno inizializzati, come abbiamo già potuto notare, al momento della chiamata del metodo. Infatti, per chiamare il metodo `somma()` dovremo passare i valori ai parametri, per esempio nel seguente modo:

```
  int risultato = oggetto1.somma(5, 6);
```

In particolare, all'interno del metodo `somma()`, la variabile `x` varrà `5`, mentre `y` varrà `6`. Dal momento che il passaggio di parametri avviene sempre per valore, potremmo anche scrivere:

```
int a = 5, b = 6;
int risultato = oggetto1.somma(a, b);
```

ed ottenere lo stesso risultato.
E' importante sottolineare che un parametro si può considerare anche una variabile locale del metodo, avendo stessa visibilità e ciclo di vita. Le differenze sussistono solo nella posizione della dichiarazione, non nel processo di immagazzinamento in memoria. Il concetto e la modalità di allocazione di memoria di una variabile d'istanza si differenziano a tal punto dai concetti e dalle modalità di allocazione di memoria di una variabile locale (o di un parametro), che è possibile assegnare in una stessa classe ad una variabile locale (o un parametro) e ad una variabile d'istanza lo stesso identificatore. Quest'argomento sarà trattato in dettaglio nel modulo 5.

Condividendo il ciclo di vita con il metodo in cui sono dichiarate, non ha senso anteporre alle variabili locali un modificatore d'accesso come `public`.

2.5 I metodi costruttori

Esistono in Java, metodi speciali che hanno "proprietà". Tra questi c'è sicuramente il metodo costruttore. Questo ha le seguenti caratteristiche:

1. Ha lo stesso nome della classe.
2. Non ha tipo di ritorno.
3. È chiamato automaticamente (e solamente) ogni volta che è istanziato un oggetto, relativamente a quell'oggetto.
4. È presente in ogni classe.

2.5.1 Caratteristiche di un costruttore

Un costruttore ha sempre e comunque lo stesso nome della classe in cui è dichiarato. È importante anche fare attenzione a lettere maiuscole e minuscole.
Il fatto che non abbia tipo di ritorno non significa che il tipo di ritorno è `void`, ma che non dichiara alcun tipo di ritorno!
Per esempio, viene presentata una classe con un costruttore dichiarato esplicitamente:

```
public class Punto
{
```

```
    public Punto() //metodo costruttore
    {
        System.out.println("costruito un Punto");
    }
    int x;
    int y;
}
```

Si noti che verrà eseguito il blocco di codice del costruttore, ogni volta che sarà istanziato un oggetto. Analizziamo meglio la sintassi che permette di istanziare oggetti. Per esempio:

```
Punto punto1; //dichiarazione
punto1 = new Punto();  // istanza
```

che è equivalente a:

```
Punto punto1 = new Punto(); //dichiarazione ed istanza
```

Come accennato in precedenza, è la parola chiave `new` che istanzia formalmente l'oggetto. Perciò basterebbe la sintassi:

```
new Punto();
```

per istanziare l'oggetto. In questo modo, però, l'oggetto appena creato non avrebbe un "nome" (si dirà "reference") e quindi non sarebbe utilizzabile.
Solitamente quindi, quando si istanzia un oggetto, gli si assegna un reference, dichiarato precedentemente.
Appare quindi evidente che l'ultima parte dell'istruzione da analizzare (`Punto()`), non va interpretata come "nomeDellaClasse con parentesi tonde", bensì come "chiamata al metodo costruttore". In corrispondenza dell'istruzione suddetta, un programma produrrebbe il seguente output:

```
costruito un Punto
```

Questo, è l'unico modo per chiamare un costruttore, che mancando di tipo di ritorno non può considerarsi un metodo ordinario.

L'utilità del costruttore non è esplicita nell'esempio appena proposto. Essendo un metodo (anche se speciale), può avere una lista di parametri. Di solito un costruttore è utilizzato per inizializzare le variabili d'istanza di un oggetto. È quindi possibile codificare il seguente costruttore all'interno della classe `Punto`:

```
public class Punto
{
    public Punto(int a, int b)
    {
        x = a;
        y = b;
    }
    public int x;
    public int y;
}
```

Con questa classe non sarà più possibile istanziare gli oggetti con la solita sintassi:

```
Punto punto1 = new Punto();
```

otterremmo un errore di compilazione, dal momento che staremmo cercando di chiamare un costruttore che non esiste (quello senza parametri)!

Questo problema è dovuto alla mancanza del costruttore di default, che sarà trattato nelle prossime pagine.

La sintassi da utilizzare però, potrebbe essere la seguente.

```
Punto punto1 = new Punto(5,6);
```

Questa ci permetterebbe anche d'inizializzare l'oggetto direttamente senza essere costretti ad utilizzare l'operatore dot. Infatti, la precedente riga di codice è equivalente alle seguenti:

```
Punto punto1 = new Punto();
punto1.x = 5;
punto1.y = 6;
```

2.5.2 Costruttore di default

Quando creiamo un oggetto, dopo l'istanza che avviene grazie alla parola chiave `new`, c'è sempre una chiamata ad un costruttore. Il lettore potrà obiettare che nelle classi utilizzate fino a questo punto non abbiamo mai fornito costruttori. Eppure, come appena detto, abbiamo chiamato costruttori ogni volta che abbiamo istanziato oggetti! Java, linguaggio fatto da programmatori per programmatori, ha una caratteristica molto importante che molti ignorano. Spessovengono inseriti automaticamente e trasparentemente dal compilatore Java comandi non inseriti dal programmatore,. Se infatti proviamo a compilare una classe sprovvista di costruttore, il compilatore ne fornisce uno implicitamente. Il costruttore inserito non contiene comandi che provocano qualche conseguenza visibile al programmatore. Esso è detto "**costruttore di default**" e non ha parametri. Ci giustifica il fatto che fino ad ora non abbiamo mai istanziato oggetti passando parametri al costruttore.
Se per esempio codificassimo la classe `Punto` nel modo seguente:

```
public class Punto
{
    public int x;
    public int y;
}
```

Al momento della compilazione, il compilatore aggiungerebbe ad essa il costruttore di default:

```
public class Punto
{
    public Punto()
    {
    //nel costruttore di default
    //sembra non ci sia scritto niente . . .
    }
    public int x;
    public int y;
}
```

Ecco perché fino ad ora abbiamo istanziato gli oggetti di tipo `Punto` con la sintassi:

```
Punto p = new Punto();
```

Se non esistesse il costruttore di default, avremmo dovuto imparare prima i costruttori e poi gli oggetti…e la strada sarebbe stata davvero in salita…

Questa è una delle caratteristiche che fa sì che Java possa essere definito linguaggio semplice! Infatti, il fatto che venga implicitamente inserito dal compilatore Java un costruttore all'interno delle classi, ci ha permesso di parlare di istanze di oggetti senza per forza dover prima spiegare un concetto tanto singolare come il costruttore.

Sottolineiamo una volta di più che il costruttore di default viene inserito in una classe dal compilatore se e solo se il programmatore non ne ha fornito uno esplicitamente. Nel momento in cui il programmatore fornisce ad una classe un costruttore, sia esso con o senza parametri, il compilatore non inserirà quello di default. L'argomento sarà ulteriormente approfondito nel Modulo 8.
L'ordine dei membri all'interno della classe non è importante. Possiamo scrivere prima i metodi, poi i costruttori e poi le variabili d'istanza, o alternare un costruttore una variabile e un metodo; non ha importanza per il compilatore. La creazione di un oggetto di una classe in memoria, infatti, non provoca l'esecuzione in sequenza del codice della classe come se fosse un programma procedurale. Solitamente però la maggior parte del codice che abbiamo visto tende a definire prima le variabili d'istanza, poi i costruttori ed infine i metodi.

Un costruttore senza parametri inserito dal programmatore non si chiama costruttore di default.

2.5.3 Package

Un **package** in Java permette di raggruppare in un'unica entità complessa classi Java logicamente correlate. Fisicamente il package non è altro che una cartella del nostro sistema operativo, ma non tutte le cartelle sono package. Per eleggere una cartella a package, una classe Java deve dichiarare nel suo codice la sua appartenenza a quel determinato package, ed inoltre ovviamente, deve risiedere fisicamente all'interno di essa. Giacché trattasi di un concetto non essenziale per approcciare in maniera corretta al linguaggio, è stato scelto di rimandare al modulo 9 la trattazione dettagliata di questo argomento.

Riepilogo

In questo modulo abbiamo introdotto i principali componenti di un'applicazione Java. Ogni file che fa parte di un'applicazione Java conterrà il listato di una ed una sola classe (tranne eccezioni di cui il lettore per adesso farà a meno). Inoltre, ogni classe è opzionalmente dichiarata appartenente ad un package.
Ogni classe solitamente contiene definizioni di variabili, metodi e costruttori.
Le variabili si dividono in tre tipologie, definite dal posizionamento della dichiarazione: d'istanza, locali e parametri (che si possono in sostanza considerare locali). Le variabili hanno un tipo (nel prossimo modulo tratteremo in dettaglio l'argomento) e un valore. In particolare le variabili d'istanza rappresentano gli attributi (le caratteristiche) di un oggetto, ed è possibile accedervi tramite l'operatore dot applicato ad un oggetto.
I metodi contengono codice operativo e definiscono le funzionalità degli oggetti. Hanno bisogno di essere dichiarati all'interno di una classe e di essere utilizzati tramite l'operatore dot applicato ad un oggetto. I metodi possono o meno restituire un risultato.
I costruttori sono metodi speciali che si trovano all'interno di ogni classe. Infatti, se il programmatore non ne fornisce uno esplicitamente, il compilatore Java aggiungerà il costruttore di default.
Per ora, progettare un'applicazione anche semplice sembra ancora un'impresa ardua…

Esercizi modulo 2

Esercizio 2.a)

Viene fornita (copiare, salvare e compilare) la seguente classe:

```
public class NumeroIntero
{
    public int numeroIntero;
    public void stampaNumero()
    {
        System.out.println(numeroIntero);
    }
}
```

Questa classe definisce il concetto di numero intero come oggetto. In essa vengono dichiarata una variabile (ovviamente) intera ed un metodo che stamperà la variabile stessa.

1. Scrivere, compilare ed eseguire una classe che:istanzierà almeno due oggetti dalla classe `NumeroIntero` (contenente ovviamente un metodo `main()`), cambierà il valore delle relative variabili e testerà la veridicità delle avvenute assegnazioni, sfruttando il metodo `stampaNumero()`.
2. Aggiungerà un costruttore alla classe `NumeroIntero`, che inizializzi la variabile d'istanza.

Due domande ancora:

1. A che tipologia di variabili appartiene la variabile `numeroIntero` definita nella classe `NumeroIntero`?
2. Se istanziamo un oggetto della classe `NumeroIntero`, senza assegnare un nuovo valore alla variabile `numeroIntero`, quanto varrà quest'ultima?

Esercizio 2.b)
Concetti sui componenti fondamentali, Vero o Falso:

1. Una variabile d'istanza deve essere per forza inizializzata dal programmatore.
2. Una variabile locale condivide il ciclo di vita con l'oggetto in cui è definita.
3. Un parametro ha un ciclo di vita coincidente con il metodo in cui è dichiarato: nasce quando il metodo viene invocato, muore quando termina il metodo.

4. Una variabile d'istanza appartiene alla classe in cui è dichiarata.
5. Un metodo è sinonimo di azione, operazione.
6. Sia le variabili sia i metodi sono utilizzabili di solito mediante l'operatore dot, applicato ad un'istanza della classe dove sono stati dichiarati.
7. Un costruttore è un metodo che non restituisce mai nientee infatti ha come tipo di ritorno `void`.
8. Un costruttore viene detto di default, se non ha parametri.
9. Un costruttore è un metodo e quindi può essere utilizzato mediante l'operatore dot, applicato ad un'istanza della classe dove è stato dichiarato
10. Un package è fisicamente una cartella che contiene classi, le quali hanno dichiarato esplicitamente di far parte del package stesso nei rispettivi file sorgente.

Esercizio 2.c)
Sintassi dei componenti fondamentali. Vero o Falso:

1. Nella dichiarazione di un metodo, il nome è sempre seguito dalle parentesi che circondano i parametri opzionali, ed è sempre preceduto da un tipo di ritorno.
2. Il seguente metodo è dichiarato in maniera corretta:
```
public void metodo ()
{
  return 5;
}
```
3. Il seguente metodo è dichiarato in maniera corretta:
```
public int metodo ()
{
  System.out.println("Ciao");
}
```
4. La seguente variabile è dichiarata in maniera corretta:
```
public int a = 0;
```
5. La seguente variabile *x* è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
Punto.x = 10;
```
6. La seguente variabile *x* è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
Punto.p1.x = 10;
```

7. La seguente variabile *x* è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
x = 10;
```
8. Il seguente costruttore è utilizzato in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
p1.Punto();
```
9. Il seguente costruttore è dichiarato in maniera corretta:
```
public class Computer {
  public void Computer()
  {

  }
}
```
10. Il seguente costruttore è dichiarato in maniera corretta:
```
public class Computer {
  public computer(int a)
  {

  }
}
```

Soluzioni esercizi modulo 2

Esercizio 2.a)

Di seguito viene listata una classe che aderisce ai requisiti richiesti:

```
public class ClasseRichiesta
{
  public static void main (String args [])
  {
    NumeroIntero uno = new NumeroIntero();
    NumeroIntero due = new NumeroIntero();
    uno.numeroIntero = 1;
    due.numeroIntero = 2;
    uno.stampaNumero();
    due.stampaNumero();
  }
}
```

Inoltre un costruttore per la classe `NumeroIntero` potrebbe impostare l'unica variabile d'istanza `numeroIntero`:

```
public class NumeroIntero
{
    public int numeroIntero;
    public NumeroIntero(int n)
    {
        numeroIntero = n;
    }
    public void stampaNumero()
    {
        System.out.println(numeroIntero);
    }
}
```

In tal caso, però, per istanziare oggetti dalla classe `NumeroIntero`, non sarà più possibile utilizzare il costruttore di default (che non sarà più inserito dal compilatore). Quindi la seguente istruzione produrrebbe un errore in compilazione:

```
NumeroIntero uno = new NumeroIntero();
```

Bisogna invece creare oggetti passando al costruttore direttamente il valore della variabile da impostare, per esempio:

```
NumeroIntero uno = new NumeroIntero(1);
```

Risposte alle due domande:

1. Trattasi di una variabile d'istanza, perché dichiarata all'interno di una classe, al di fuori di metodi.
2. Il valore sarà zero, ovvero il valore nullo per una variabile intera. Infatti, quando si istanzia un oggetto, le variabili d'istanza vengono inizializzate ai valori nulli, se non esplicitamente inizializzate ad altri valori.

Esercizio 2.b)
Concetti sui componenti fondamentali. Vero o Falso:

1. **Falso** una variabile locale deve essere per forza inizializzata dal programmatore.
2. **Falso** una variabile d'istanza condivide il ciclo di vita con l'oggetto in cui è definita.
3. **Vero.**
4. **Falso** una variabile d'istanza appartiene ad un oggetto istanziato dalla classe in cui è dichiarata.
5. **Vero.**
6. **Vero** .
7. **Falso** un costruttore è un metodo che non restituisce mai niente, infatti non ha tipo di ritorno.
8. **Falso** un costruttore viene detto di default, se viene inserito dal compilatore. Inoltre non ha parametri.
9. **Falso** un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
10. **Vero.**

Esercizio 2.c)
Sintassi dei componenti fondamentali. Vero o Falso:

1. **Vero.**
2. **Falso** tenta di restituire un valore intero avendo tipo di ritorno `void`.
3. **Falso** il metodo dovrebbe restituire un valore intero.

4. **Vero.**
5. **Falso** l'operatore dot deve essere applicato all'oggetto e non alla classe:
```
Punto p1 = new Punto();
p1.x = 10;
```
6. **Falso** L'operatore dot deve essere applicato all'oggetto e non alla classe, ed inoltre la classe non "contiene" l'oggetto.
7. **Falso** L'operatore dot deve essere applicato all'oggetto. Il compilatore non troverebbe infatti la dichiarazione della variabile x.
8. **Falso** un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
9. **Falso** il costruttore non dichiara tipo di ritorno e deve avere nome coincidente con la classe.
10. **Falso** il costruttore deve avere nome coincidente con la classe.

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.1, 2.2, 2.3, 2.4, 2.5)	☐	
Saper dichiarare una classe (unità 2.1)	☐	
Istanziare oggetti da una classe (unità 2.2)	☐	
Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4)	☐	
Dichiarare ed invocare un metodo (unità 2.3)	☐	
Saper dichiarare ed inizializzare una variabile (unità 2.4)	☐	
Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4)	☐	
Dichiarare ed invocare un metodo costruttore (unità 2.5)	☐	
Comprendere il costruttore di default (unità 2.5)	☐	

Note:

3

Complessità: bassa

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper utilizzare le convenzioni per il codice Java (unità 3.1) .
2. Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.2).
3. Saper gestire casting e promotion (unità 3.2).
4. Saper utilizzare i reference, e capirne la filosofia (unità 3.4).
5. Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.4).
6. Saper utilizzare la classe String (unità 3.4).
7. Saper utilizzare gli array (unità 3.5).
8. Saper commentare il proprio codice ed essere in grado di utilizzare lo strumento javadoc per produrre documentazione tecnica esterna (unità 3.1, 3.4).

3 Identificatori, tipi di dati ed array

In questo modulo saranno dapprima definite alcune regole fondamentali che la programmazione Java richiede. Poi passeremo ad introdurre i tipi di dati definiti dal linguaggio e tutte le relative problematiche. Per completare il discorso introdurremo gli array, che in Java sono oggetti abbastanza "particolari" rispetto ad altri linguaggi.

3.1 Stile di codifica

Il linguaggio Java:

1. E' a schema libero.
2. E' case sensitive.
3. Supporta (ovviamente) i commenti.
4. Ha parole chiave.
5. Ha regole ferree per i tipi di dati, ed alcune semplici convenzioni per i nomi.

3.1.1 Schema Libero

Potremmo scrivere un intero programma in Java tutto su di un'unica riga, oppure andando a capo dopo ogni parola scritta: il compilatore compilerà ugualmente il nostro

codice se esso è corretto. Il problema semmai è dello sviluppatore, che avrà difficoltà a capire il significato del codice!
Esistono quindi metodi standard d'indentazione del codice, che facilitano la lettura di un programma Java. Di seguito è presentata una classe che utilizza uno dei due più usati metodi di formattazione:

```
public class Classe
{
    public int intero;
    public void metodo()
    {
        intero = 10;
        int unAltroIntero = 11;
    }
}
```

Con questo stile (che è utilizzato anche dai programmatori C), il lettore può capire subito dove la classe ha inizio e dove ha fine, dato che le parentesi graffe che delimitano un blocco di codice si trovano incolonnate. Stesso discorso per il metodo: risultano evidenti l'inizio, la fine e la funzionalità del metodo.
Un altro stile molto utilizzato (ma solo dai "javisti") è il seguente:

```
public class Classe {
    public int intero;
    public void metodo() {
        intero = 10;
        int unAltroIntero = 11;
    }
}
```

per il quale valgono circa le stesse osservazioni fatte per il primo metodo.
Si raccomanda, per una buona riuscita del lavoro che sarà svolto in seguito dal lettore, una rigorosa applicazione di uno dei due stili appena presentati. In questo manuale utilizzeremo entrambi gli stili.

Se utilizzate EJE come editor, potete formattare il vostro codice con entrambi gli stili, mediante la pressione del menu apposito, o tramite il relativo bottone sulla barra degli strumenti, o attraverso la scorciatoia

di tastiera CTRL-SHIFT-F. Per configurare lo stile da utilizzare, scegliere il menu File-Options (o premere F12) e impostare il "Braces style" con lo stile desiderato.

Alcuni dei tipici errori che il programmatore alle prime armi commette sono semplici dimenticanze. E' frequente dimenticare di chiudere una parentesi graffa di una classe, o dimenticare il "`;`" dopo un'istruzione, o le parentesi tonde che seguono l'invocazione di un metodo. Per questo è consigliato al lettore di abituarsi a scrivere ogni istruzione in maniera completa, per poi pensare a formattare in maniera corretta. Per esempio, se dichiariamo una classe è buona prassi scrivere entrambe le parentesi graffe, prima di scrivere il codice contenuto in esse. I tre passi seguenti dovrebbero chiarire il concetto al lettore:

- passo 1: dichiarazione:

```
public class Classe {}
```

- passo 2: formattazione:

```
public class Classe {

}
```

- passo 3: completamento:

```
public class Classe {
    public int intero;
    ...
}
```

3.1.2 Case sensitive

Java è un linguaggio case sensitive, ovvero fa distinzione tra lettere maiuscole e minuscole. Il programmatore alle prime armi tende a digerire poco questa caratteristica del linguaggio. Bisogna ricordarsi di non scrivere ad esempio `class` con lettera maiuscola, perché per il compilatore non significa niente. L'identificatore `unAltroIntero` è diverso dall'identificatore `unaltroIntero`. Bisogna quindi fare attenzione e, specialmente nei primi tempi, avere un po' di pazienza.

3.1.3 Commenti

Commentare opportunamente il codice è una pratica che dovrebbe essere considerata obbligatoria dal programmatore. Così, infatti, le correzioni da apportare ad un programma risulteranno nettamente semplificate.
Java supporta tre tipi diversi di commenti al codice:

1. Commenti su un'unica riga:

```
// Questo è un commento su una sola riga
```

2. Commento su più righe:

```
/*
   Questo è un commento
   su più righe
*/
```

3. Commento "Javadoc":

```
/**
   Questo commento permette di produrre
   la documentazione del codice
   in formato HTML, nello standard Javadoc
*/
```

Nel primo caso tutto ciò che scriveremo su di una riga dopo aver scritto "//" non sarà preso in considerazione dal compilatore. Questa sintassi permetterà di commentare brevemente alcune parti di codice. Per esempio:

```
// Questo è un metodo
public void metodo() {
  . . .
}

public int a; //Questa è una variabile
```

Il commento su più righe potrebbe essere utile ad esempio per la descrizione delle funzionalità di un programma. In pratica il compilatore non prende in considerazione tutto ciò che scriviamo tra "/*" ed "*/". Per esempio:

```
/*
  Questo metodo è commentato…
  public void metodo() {
    . . .
  }
*/
```

Queste due prime tipologie di commenti sono ereditate dal linguaggio C++.

Ciò che invece rappresenta una novità è il terzo tipo di commento. L'utilizzo in pratica è lo stesso del secondo tipo, e permette di definire commenti su più righe. In più offre la possibilità di produrre in modo standard, in formato HTML, la documentazione tecnica del programma, sfruttando un comando chiamato "javadoc". Per esempi pratici d'utilizzo il lettore è rimandato alla fine di questo modulo.

3.1.4 Regole per gli identificatori

Gli identificatori (nomi) dei metodi, delle classi, degli oggetti, delle variabili, delle costanti (ma anche delle interfacce, le enumerazioni e le annotazioni che studieremo più avanti), hanno due regole da rispettare.

1. Un identificatore non può coincidere con una **parola chiave** (**keyword**) di Java. Una parola chiave ha un certo significato per il linguaggio di programmazione. Tutte le parole chiave di Java sono costituite da lettere minuscole. Nella seguente tabella sono riportate tutte le parole chiave di Java in ordine alfabetico, fatta eccezione per le ultime arrivate: `assert` introdotta nella release 1.4 , `enum` e `@interface` introdotte solo dalla versione 5 di Java:

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
false	final	finally	float	for
goto	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws

transient	true	try	void	volatile
while	assert	enum	@interface	

Possiamo notare alcune parole chiave che abbiamo già incontrato come `int`, `public`, `void`, `return` e `class`, e di cui già conosciamo il significato. Ovviamente non potremo chiamare una variabile `class`, oppure un metodo `void`.

Possiamo notare anche le parole riservate `goto` e `const`, che non hanno nessun significato in Java, ma che non possono essere utilizzate come identificatori. Esse sono dette parole riservate (reserved words). Notare anche che `@interface` (parola utilizzata per dichiarare le cosiddette annotazioni, cfr. Modulo 19), iniziando con il simbolo di "chiocciola" @, non potrebbe comunque essere utilizzata quale identificatore, come spiegato nel prossimo punto. Anche in questo caso quindi non si dovrebbe parlare tecnicamente di parola chiave.

2. In un identificatore:

   a. il primo carattere può essere `A-Z`, `a-z`, `_`, `$`
   b. il secondo ed i successivi possono essere `A-Z`, `a-z`, `_`, `$`, `0-9`

Quindi: "`a2`" è un identificatore valido, mentre "`2a`" non lo è.

3.1.5 Regole facoltative per gli identificatori e convenzioni per i nomi

Se quando utilizziamo gli identificatori rispettiamo le due regole appena descritte, non otterremo errori in compilazione. Ma esistono direttive standard fornite dalla Sun per raggiungere uno standard anche nello stile d'implementazione. E' importantissimo utilizzare queste direttive in un linguaggio tanto standardizzato quanto Java.

1. Gli identificatori devono essere significativi. Infatti, se scriviamo un programma utilizzando la classe `a`, che definisce le variabili `b`, `c`, `d` e il metodo `e`, sicuramente ridurremo la comprensibilità del programma stesso.
2. Di solito l'identificatore di una variabile è composto da uno o più sostantivi, per esempio `numeroLati`, o `larghezza` o anche `numeroPartecipantiAlSimposio`. Gli identificatori dei metodi solitamente conterranno verbi, ad esempio

`stampaNumeroPartecipantiAlSimposio`, o `somma`. Quest'ultima è una direttiva che è dettata, più che da Sun, dal buon senso.

3. Esistono convenzioni per gli identificatori, così come in molti altri linguaggi. In Java sono semplicissime:

- **Convenzione per le classi:**
  Un identificatore di una classe (ma questa regola vale anche per le interfacce, le enumerazioni e le annotazioni che studieremo più avanti) deve sempre iniziare per lettera maiuscola. Se composto da più parole, queste non si possono separare, perché il compilatore non può intuire le nostre intenzioni. Come abbiamo notato in precedenza, bisogna invece unire le parole in modo tale da formare un unico identificatore, e fare iniziare ognuna di esse con lettera maiuscola. Esempi di identificatori per una classe potrebbero essere:

```
Persona
MacchinaDaCorsa
FiguraGeometrica
```

- **Convenzione per le variabili:**
  Un identificatore di una variabile deve sempre iniziare per lettera minuscola. Se l'identificatore di una variabile deve essere composto da più parole, valgono le stesse regole in uso per gli identificatori delle classi (tranne ovviamente il fatto che la prima lettera deve sempre essere minuscola). Quindi, esempi di identificatori per una variabile potrebbero essere:

```
pesoSpecifico
numeroDiMinutiComplessivi
x
```

- **Convenzione per i metodi:**
  Per un identificatore di un metodo valgono le stesse regole in uso per gli identificatori delle variabili. Potremo in ogni caso sempre distinguere un identificatore di una variabile da un identificatore di un metodo, giacché quest'ultimo è sempre seguito da parentesi tonde. Inoltre, per quanto già affermato, il nome di un metodo dovrebbe contenere almeno un verbo. Quindi, esempi di identificatori validi per un metodo potrebbero essere:

```
sommaDueNumeri(int a, int b)
```

```
cercaUnaParola(String parola)
stampa()
```

- **Convenzione per le costanti:**
  Gli identificatori delle costanti, invece, si devono distinguere nettamente dagli altri, e tutte le lettere dovranno essere maiuscole. Se l'identificatore è composto di più parole, queste si separano con un underscore (simbolo di sottolineatura). Per esempio:

```
NUMERO_LATI_DI_UN_QUADRATO
PI_GRECO
```

> **Non esiste un limite al numero dei caratteri di cui può essere composto un identificatore.**

3.2 Tipi di dati primitivi

Java definisce solamente otto tipi di dati primitivi:

- Tipi interi: `byte`, `short`, `int`, `long`.
- Tipi floating point (o a virgola mobile): `float` e `double`.
- Tipo testuale: `char`.
- Tipo logico-booleano: `boolean`.

> **In Java non esistono tipi senza segno (unsigned), e per la rappresentazione dei numeri interi negativi viene utilizzata la regola del "complemento a due".**

3.2.1 Tipi di dati interi, casting e promotion

I tipi di dati interi sono quattro: `byte`, `short`, `int` e `long`. Essi condividono la stessa funzionalità (tutti possono immagazzinare numeri interi positivi o negativi), ma differiscono per quanto riguarda il proprio intervallo di rappresentazione. Infatti un `byte` può immagazzinare un intero utilizzando un byte (otto bit), uno `short` (due

byte), un `int` (quattro byte) ed un `long` (otto byte). Lo schema seguente riassume dettagliatamente i vari intervalli di rappresentazione:

Tipo	**Intervallo di rappresentazione**
`byte`	8 bit (da -128 a +127)
`short`	16 bit (da -32.768 a +32.767)
`int`	32 bit (da -2.147.483.648 a +2.147.483.647)
`long`	64 bit (da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)

Si noti che l'intervallo di rappresentazione dei tipi primitivi interi è sempre compreso tra un minimo di `-2` elevato al numero di bit meno uno, ed un massimo di `2` elevato al numero di bit meno uno, sottraendo alla fine un'unità. Quindi potremmo riscrivere la tabella precedente nel seguente modo:

Tipo	**Intervallo di rappresentazione**
`byte`	8 bit (da -2^{7} a $2^{7}-1$)
`short`	16 bit (da -2^{15} a $2^{15}-1$)
`int`	32 bit (da -2^{31} a $2^{31}-1$)
`long`	64 bit (da -2^{63} a $2^{63}-1$)

Si noti anche che, per ogni tipo numerico, esiste un numero pari di numeri positivi e di numeri negativi. Infatti il numero "`0`" è considerato un numero positivo.

Per immagazzinare un intero si possono utilizzare tre forme: decimale, ottale ed esadecimale. Per la notazione ottale basta anteporre al numero intero uno `0` (zero), mentre per la notazione esadecimale `0` e `x` (indifferentemente maiuscola o minuscola). Ecco qualche esempio d'utilizzo di tipi interi:

```
byte b = 10; //notazione decimale
short s = 022; //notazione ottale
int i = 1000000000; //notazione decimale
long l = 0x12acd; //notazione esadecimale
```

Le notazioni ottale ed esadecimale non hanno un grande utilizzo in Java, ma è possibile che nell'esame di certificazione SCJP si utilizzino interi con notazioni di questi tipi.

Esiste una serie di osservazioni da fare riguardanti i tipi di dati primitivi. Facciamo subito un esempio e consideriamo la seguente assegnazione:

```
byte b = 127;
```

Il compilatore è in grado di capire che `127` è un numero appartenente all'intervallo di rappresentazione dei `byte` e quindi l'espressione precedente è corretta e compilabile senza problemi. Allo stesso modo, le seguenti istruzioni:

```
byte b = 128; //il massimo per short è 127
short s = 32768; //il massimo per short è 32767
int i = 2147483648; //il massimo per int è 2147483647
```

provocano errori in compilazione.
C'è da fare però una precisazione. Consideriamo la seguente istruzione:

```
byte b = 50;
```

Questo statement è corretto. Il numero intero `50` è tranquillamente compreso nell'intervallo di rappresentazione di un `byte`, che va da `-128` a `+127`. Il compilatore determina la grandezza del valore numerico, e controlla se è compatibile con il tipo di dato dichiarato. Allora consideriamo quest'altro statement:

```
b = b * 2;
```

Ciò darà luogo ad un errore in compilazione! Infatti il compilatore non eseguirà l'operazione di moltiplicazione per controllare la compatibilità con il tipo di dato dichiarato. Invece promuoverà automaticamente i due operandi ad `int`. Quindi, se `50*2` è un `int`, non può essere immagazzinato in `b` che è un `byte`.

Il fenomeno appena descritto è conosciuto come "promozione automatica nelle espressioni", più brevemente "promotion". Per gli operatori binari esistono quattro regole, che dipendono dai tipi degli operandi in questione:
- se uno degli operandi è `double`, l'altro operando sarà convertito in `double`;

- se il più "ampio" degli operandi è un `float`, l'altro operando sarà convertito in `float`;
- se il più "ampio" degli operandi è un `long`, l'altro operando sarà converito in `long`;
- in ogni altro caso entrambi gli operandi saranno convertiti in `int`.
La promozione automatica ad intero degli operandi avviene prima che venga eseguita una qualsiasi operazione binaria.

Ma, evidentemente, `50*2` è immagazzinabile in un `byte`. Esiste una tecnica per forzare una certa quantità ad essere immagazzinata in un certo tipo di dato. Questa tecnica è nota come **cast** (o **casting**). La sintassi da utilizzare per risolvere il nostro problema è:

```
b = (byte)(b * 2);
```

In questo modo il compilatore sarà avvertito che un'eventuale perdita di precisione, è calcolata e sotto controllo.
Bisogna essere però molto prudenti nell'utilizzare il casting in modo corretto. Infatti se scrivessimo:

```
b = (byte)128;
```

il compilatore non segnalerebbe nessun tipo d'errore. Siccome il cast agisce troncando i bit in eccedenza (nel nostro caso, dato che un `int` utilizza 32 bit, mentre un `byte` solo 8, saranno troncati i primi 24 bit dell'`int`), la variabile `b` avrà il valore di `-128` e non di `128`!
Un altro tipico problema di cui preoccuparsi è la somma di due interi. Per le stesse ragioni di cui sopra, se la somma di due interi supera il range consentito, è comunque possibile immagazzinarne il risultato in un intero senza avere errori in compilazione, ma il risultato sarà diverso da quello previsto. Per esempio, le seguenti istruzioni saranno compilate senza errori:

```
int a = 2147483647;// Massimo valore per un int
int b = 1;
int risultato = a+b;
```

ma il valore della variabile risultato sarà di `-2147483648`!

Anche la divisione tra due interi rappresenta un punto critico! Infatti il risultato finale, per quanto detto sinora, non potrà che essere immagazzinato in un intero, ignorando così eventuali cifre decimali.
Inoltre, se utilizziamo una variabile `long`, a meno di cast espliciti, essa sarà sempre inizializzata con un intero. Quindi, se scriviamo:

```
long l = 2000;
```

Dobbiamo tener ben presente che `2000` è un `int` per default, ma il compilatore non ci segnalerà errori perché un `int` può essere tranquillamente immagazzinato in un `long`. Per la precisione dovremmo scrivere:

```
long l = 2000L;
```

che esegue con una sintassi più compatta il casting da `int` a `long`. Quindi, un cast a `long` si ottiene con una sintassi diversa dal solito, posponendo una "elle" maiuscola o minuscola al valore intero assegnato.

Si preferisce utilizzare la notazione maiuscola, dato che una "elle" minuscola si può confondere con un numero "uno" in alcuni ambienti di sviluppo. Notare che saremmo obbligati ad un casting a `long` nel caso in cui volessimo assegnare alla variabile `l` un valore fuori dell'intervallo di rappresentazione di un `int`. Per esempio:
`long l = 3000000000;`
produrrebbe un errore in compilazione. Bisogna eseguire il casting nel seguente modo:
`long l = 3000000000L;`

3.2.2 Tipi di dati a virgola mobile, casting e promotion

Java utilizza per i valori floating point lo standard di decodifica IEEE-754. I due tipi che possiamo utilizzare sono:

Tipo	Intervallo di rappresentazione
`float`	32 bit (da +/–1.40239846^{-45} a +/–3.40282347^{+38})
`double`	64 bit (da +/–$4.94065645841246544^{-324}$ a +/–$1.79769313486231570^{+328}$)

E' possibile utilizzare la notazione esponenziale o ingegneristica (la "`e`" può essere sia maiuscola sia minuscola). Per quanto riguarda cast e promotion, la situazione cambia rispetto al caso dei tipi interi. Il default è `double` e non `float` come ci si potrebbe aspettare. Ciò implica che se vogliamo assegnare un valore a virgola mobile ad un `float`, non possiamo fare a meno di un cast. Per esempio, la seguente riga di codice provocherebbe un errore in compilazione:

```
float f = 3.14;
```

Anche in questo caso, il linguaggio ci viene incontro permettendoci il cast con la sintassi breve:

```
float f = 3.14F;
```

La "effe" può essere sia maiuscola sia minuscola.
Esiste, per quanto ridondante, anche la forma contratta per i `double`:

```
double d = 10.12E24;
```

è equivalente a scrivere:

```
double d = 10.12E24D;
```

Alcune operazioni matematiche potrebbero dare risultati che non sono compresi nell'insieme dei numeri reali (per esempio "infinito"). Le classi wrapper `Double` e `Float` (cfr. Modulo 12), forniscono le seguenti costanti statiche:

```
Float.NaN
Float.NEGATIVE_INFINITY
Float.POSITIVE_INFINITY
Double.NaN
Double.NEGATIVE_INFINITY
Double.POSITIVE_INFINITY
```

Dove `NaN` sta per "Not a Number" ("non un numero"). Per esempio:

```
double d = -10.0 / 0.0;
System.out.println(d);
```

produrrà il seguente output:

```
NEGATIVE_INFINITY
```

E' bene aprire una piccola parentesi sui modificatori `final` e `static`. Infatti, una costante statica in Java è una variabile dichiarata `final` e `static`. In particolare, il modificatore `final` applicato ad una variabile farà in modo che ogni tentativo di cambiare il valore di tale variabile (già inizializzata) produrrà un errore in compilazione. Quindi una variabile `final` è una costante.
Per quanto riguarda il modificatore `static`, come abbiamo già avuto modo di asserire precedentemente, il discorso è più complicato. Quando dichiariamo una variabile statica, allora tale variabile sarà condivisa da tutte le istanze di quella classe. Questo significa che, se abbiamo la seguente classe che dichiara una variabile statica (anche detta "variabile di classe"):

```
public class MiaClasse {
  public static int variabileStatica = 0;
}
```

se istanziamo oggetti da questa classe, essi condivideranno il valore di `variabileStatica`. Per esempio, il seguente codice:

```
MiaClasse ogg1 = new MiaClasse();
MiaClasse ogg2 = new MiaClasse();
System.out.println(ogg1.variabileStatica + "-" +
  ogg2.variabileStatica);
ogg1.variabileStatica = 1;
System.out.println(ogg1.variabileStatica + "-" +
  ogg2.variabileStatica);
ogg2.variabileStatica = 2;
System.out.println(ogg1.variabileStatica + "-" +
  ogg2.variabileStatica);
```

produrrà il seguente output:

```
0-0
1-1
2-2
```

I modificatori `static` e `final`, saranno trattati in dettaglio nel Modulo 9.

Sembra evidente che Java non sia il linguaggio ideale per eseguire calcoli! Consiglio saggio: in caso di espressioni aritmetiche complesse, utilizzare solamente `double`. In altri tempi questo consiglio sarebbe stato considerato folle. Ma purtroppo, persino utilizzando solo variabili `double`, non saremo sicuri di ottenere risultati precisissimi in caso di espressioni molto complesse. Infatti, avendo i `double` una rappresentazione numerica comunque limitata, a volte devono essere arrotondati.
In alcuni casi, per ottenere risultati precisi è sufficiente utilizzare il modificatore `strictfp` (cfr. Modulo 9). Altre volte è necessario utilizzare una classe della libreria Java: `BigDecimal` (package `java.math`; cfr. Documentazione ufficiale) in luogo del tipo `double`.

3.2.3 Tipo di dato logico - booleano

Il tipo di dato `boolean` utilizza solo un bit per memorizzare un valore e gli unici valori che può immagazzinare sono `true` e `false`. Per esempio:

```
boolean b = true;
```

Vengono definiti **Literals** i valori (contenenti lettere o simboli) che vengono specificati nel codice sorgente invece che al runtime. Possono rappresentare variabili primitive o stringhe e possono apparire solo sul lato destro di un'assegnazione o come argomenti quando si invocano metodi. Non è possibile assegnare valori a literals. Chiaramente i literals devono contenere caratteri o simboli (come per esempio il punto).
Esempi di literals sono ovviamente `true` e `false`, ma anche , e i valori assegnati alle stringhe ed ai caratteri. Segue qualche esempio di literals:

```
boolean isBig = true;
char c = 'w';
String a = "\n";
int i = 0x1c;
float f = 3.14f;
double d = 3.14;
```

3.2.4 Tipo di dato primitivo letterale

Il tipo `char` permette di immagazzinare caratteri (uno per volta). Utilizza l'insieme Unicode per la decodifica. Unicode è uno standard a 16 bit, che contiene tutti i caratteri

della maggior parte degli alfabeti del mondo. Fortunatamente l'insieme ASCII, che probabilmente è più familiare al lettore, è un sottoinsieme dello Unicode. I primi 256 caratteri dei due insiemi coincidono. Per maggiori informazioni, http://www.unicode.org.

Lo standard Unicode si è ultimamente evoluto alla versione 4.0, che contiene altri caratteri supplementari (la maggior parte dei quali simboli ideografici Han…), fino a giungere a 21 bit! Java, dalla versione 5, si è evoluto a sua volta per supportare tutti i nuovi caratteri, con uno stratagemma basato sul tipo `int` (per informazioni consultare la documentazione ufficiale).

Possiamo assegnare ad un `char` un qualsiasi carattere che si trova sulla nostra tastiera (ma anche il prompt Dos deve supportare il carattere richiesto). Oppure possiamo assegnare ad un `char` direttamente un valore Unicode in esadecimale, che identifica univocamente un determinato carattere, anteponendo ad essa il prefisso `\u`. In qualsiasi caso, dobbiamo comprendere tra apici singoli, il valore da assegnare. Ecco qualche esempio:

```
char primoCarattere = 'a';
char car ='@';
char letteraGrecaPhi ='\u03A6';  (lettera "Φ")
char carattereUnicodeNonIdentificato ='\uABC8';
```

Esiste anche la possibilità di immagazzinare caratteri di escape come:

- `\n` che equivale ad andare a capo (tasto return)
- `\\` che equivale ad un solo \ (tasto backslash)
- `\t` che equivale ad una tabulazione (tasto TAB)
- `\'` che equivale ad un apice singolo
- `\"` che equivale ad un doppio apice (virgolette)

Si noti che è possibile utilizzare caratteri in espressioni aritmetiche. Per esempio, possiamo sommare un `char` con un `int`. Infatti, ad ogni carattere, corrisponde un numero intero. Per esempio:

```
int i = 1;
char a = 'A';
char b = (char)(a+i); // b ha valore 'B'!
```

3.3 Tipi di dati non primitivi: reference

Abbiamo già visto come istanziare oggetti da una certa classe. Dobbiamo prima dichiarare un oggetto di tale classe con una sintassi di questo tipo:

```
NomeClasse nomeOggetto;
```

per poi istanziarlo utilizzando la parola chiave `new`.
Dichiarare un oggetto quindi è del tutto simile a dichiarare un tipo di dato primitivo. Il "nome" che diamo ad un oggetto è detto "reference". Infatti non si sta parlando di una variabile tradizionale, bensì di una variabile che alcuni definiscono “puntatore”.
Possiamo definire un puntatore come una variabile che contiene un indirizzo in memoria.
C’è una sottile e potente differenza tra la dichiarazione di un tipo di dato primitivo ed uno non primitivo. Consideriamo un esempio, partendo dalla definizione di una classe che astrae in maniera banale il concetto di data:

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

Per il nostro esempio, `Data` sarà quindi un tipo di dato non primitivo (astratto). Come tipo di dato primitivo consideriamo un `double`. Consideriamo le seguenti righe di codice, supponendo che si trovino all’interno di un metodo `main()` di un’altra classe:

```
double unNumero = 5.0;
Data unGiorno = new Data();
```

Potrebbe aiutarci una schematizzazione di come saranno rappresentati i dati in memoria quando l’applicazione è in esecuzione.
In realtà, riportare fedelmente queste informazioni è un’impresa ardua, ma se vogliamo anche inutile. Infatti non avremo mai a che fare direttamente con la gestione della memoria se programmiamo in Java.
Può ad ogni modo essere utile immaginare la situazione in memoria, con il tipo di schematizzazione, frutto di una convenzione, illustrato in figura 3.1:

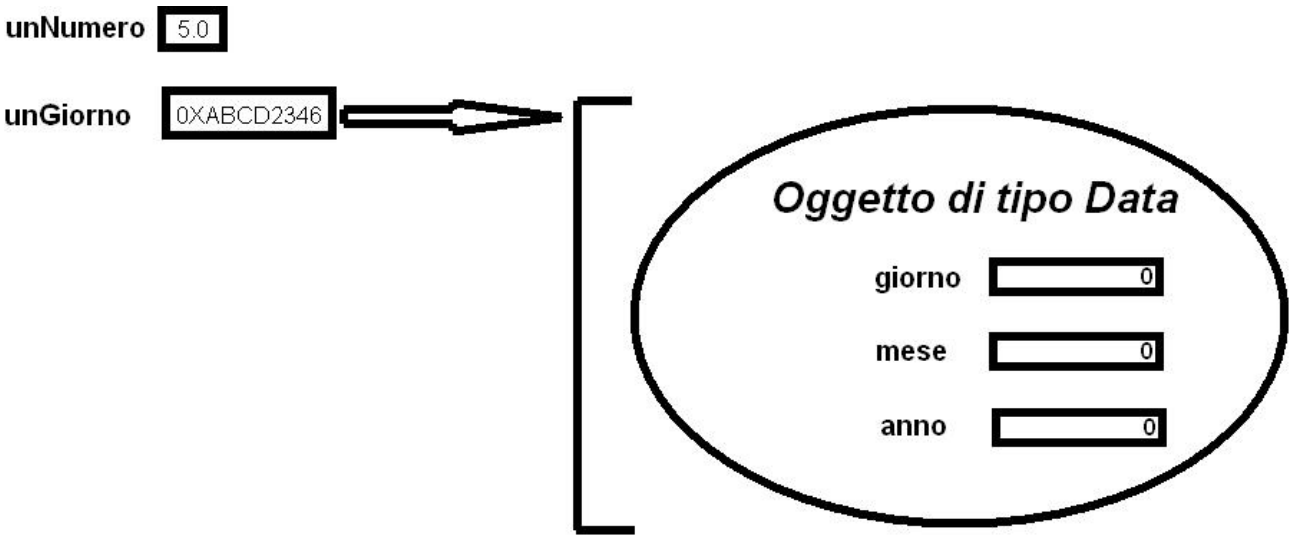


Figura 3.1 – Convenzione di schematizzazione della memoria

La differenza pratica tra un reference ed una variabile è evidente nelle assegnazioni. Consideriamo il seguente frammento di codice:

```
double unNumero = 5.0;
double unAltroNumero = unNumero;
Data unGiorno = new Data();
Data unAltroGiorno = unGiorno;
```

Sia per il tipo di dato primitivo, sia per quello complesso, abbiamo creato un comportamento equivalente: dichiarazione ed assegnazione di un valore, e rassegnazione di un altro valore.
La variabile `unAltroNumero` assumerà lo stesso valore della variabile `unNumero`, ma le due variabili rimarranno indipendenti l'una dall'altra. Il valore della variabile `unNumero` verrà infatti copiato nella variabile `unAltroNumero`. Se il valore di una delle due variabili sarà modificato in seguito, l'altra variabile non apporterà modifiche al proprio valore.
Il reference `unAltroGiorno`, invece, assumerà semplicemente il valore (cioè l'indirizzo) del reference `unGiorno`. Ciò significa che `unAltroGiorno` punterà allo stesso oggetto cui punta `unGiorno`.
La figura 3.2 mostra la situazione rappresentata graficamente:

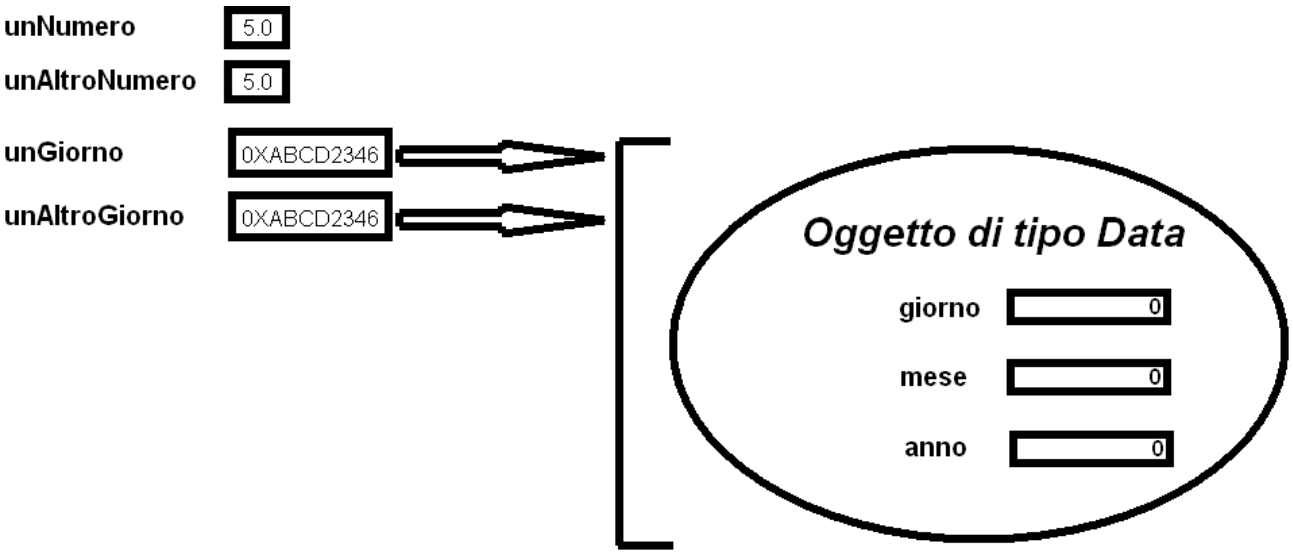


Figura 3.2 – Convenzione di schematizzazione della memoria

Quindi, se in seguito sarà apportata una qualche modifica all'oggetto comune tramite uno dei due reference, ovviamente questa sarà verificabile anche tramite l'altro reference. Per intenderci:

```
unGiorno.anno
```

è sicuramente equivalente a:

```
unAltroGiorno.anno
```

3.3.1 Passaggio di parametri per valore

"Il passaggio di parametri in Java avviene sempre per valore".

Quest'affermazione viene contraddetta in alcuni testi, ma basterà leggere l'intero paragrafo per fugare ogni dubbio.

Quando si invoca un metodo che come parametro prende in input una variabile, al metodo stesso viene passato solo il valore (una copia) della variabile, che quindi rimane immutata anche dopo l'esecuzione del metodo. Per esempio consideriamo la classe:

```
public class CiProvo
{
    public void cambiaValore(int valore)
    {
        valore = 1000;
    }
}
```

il seguente frammento di codice:

```
CiProvo ogg = new CiProvo();
int numero = 10;
ogg.cambiaValore(numero);
System.out.println("il valore del numero è " + numero);
```

produrrà il seguente output:

```
il valore del numero è 10
```

Infatti il parametro `valore` del metodo `cambiaValore()`, nel momento in cui è stato eseguito il metodo, non coincideva con la variabile `numero`, bensì immagazzinava solo la copia del suo valore (`10`). Quindi ovviamente la variabile `numero` non è stata modificata.
Stesso discorso vale per i tipi reference: viene sempre passato il valore del reference, ovvero l'indirizzo in memoria. Consideriamo la seguente classe:

```
public class CiProvoConIReference
{
    public void cambiaReference(Data data)
    {
        data = new Data();
        // Un oggetto appena istanziato
        // ha le variabili con valori nulli
    }
}
```

il seguente frammento di codice:

```
CiProvoConIReference ogg = new CiProvoConIReference();
Data dataDiNascita = new Data();
dataDiNascita.giorno = 26;
dataDiNascita.mese = 1;
dataDiNascita.anno = 1974;
ogg.cambiaReference(dataDiNascita);
System.out.println("Data di nascita = "
  + dataDiNascita.giorno + "-"+ dataDiNascita.mese
  + "-" +dataDiNascita.anno );
```

produrrà il seguente output:

```
Data di nascita = 26-1-1974
```

Valgono quindi le stesse regole anche per i reference.
Se il metodo `cambiaReference()` avesse cambiato i valori delle variabili d'istanza dell'oggetto avremmo avuto un output differente. Riscriviamo il metodo in questione:

```
public void cambiaReference(Data data)
{
    data.giorno=29; // data punta allo stesso indirizzo
    data.mese=7     // della variabile dataDiNascita
}
```

Il fatto che il passaggio avvenga sempre per valore garantisce che un oggetto possa essere modificato e, contemporaneamente, si è certi che dopo la chiamata del metodo il reference punti sempre allo stesso oggetto.
In altri linguaggi, come il C, è permesso anche il passaggio di parametri "per riferimento". In quel caso al metodo viene passato l'intero riferimento, non solo il suo indirizzo, con la conseguente possibilità di poterne mutare l'indirizzamento. Questa caratteristica non è stata importata in Java, perché considerata (a ragione) una minaccia alla sicurezza. Molti virus, worm etc… sfruttano infatti la tecnica del passaggio per riferimento.

Per documentarsi sul passaggio per riferimento in C (e in generale dei puntatori), consigliamo la semplice quanto efficace spiegazione data da Bruce Eckel nel modulo 3 del suo "Thinking in C++ - 2nd Edition", gratuitamente scaricabile da http://www.mindview.net.

Java ha scelto ancora una volta la strada della robustezza e della semplicità, favorendola rispetto alla mera potenza del linguaggio.

Alcuni autori di altri testi affermano che il passaggio di parametri in Java avviene per valore per i tipi di dato primitivi, e per riferimento per i tipi di dato complesso.
Chi scrive ribadisce che si tratta di un problema di terminologia. Probabilmente, se ignorassimo il linguaggio C, anche noi daremmo un significato diverso al "passaggio per riferimento". L'importante è capire il concetto senza fare confusione.
Confusione che addizionalmente è data anche dalla teoria degli Enterprise JavaBeans (EJB), dove effettivamente si parla di passaggio per valore e per riferimento. Ma in quel caso ci si trova in ambiente distribuito, ed il significato cambia ancora…

3.3.2 Inizializzazione delle variabili d'istanza

Abbiamo già asserito che, nel momento in cui è istanziato un oggetto, tutte le sue variabili d'istanza (che con esso condividono il ciclo di vita) vengono inizializzate ai rispettivi valori nulli. Di seguito è presentata una tabella che associa ad ogni tipo di dato il valore con cui viene inizializzata una variabile di istanza al momento della sua creazione:

Variabile	**Valore**
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' (NULL)
boolean	false
Ogni tipo reference	null

3.4 Introduzione alla libreria standard

Come già accennato più volte, Java possiede un'enorme e lussuosa libreria di classi standard, che costituisce uno dei punti di forza del linguaggio. Essa è organizzata in vari package (letteralmente pacchetti, fisicamente cartelle) che raccolgono le classi secondo un'organizzazione basata sul campo d'utilizzo. I principali package sono:

- **`java.io`** contiene classi per realizzare l'input – output in Java (trattato nel modulo 13)
- **`java.awt`** contiene classi per realizzare interfacce grafiche, come `Button` (trattato nel modulo 15)
- **`java.net`** contiene classi per realizzare connessioni, come `Socket` (trattato nel modulo 13)
- **`java.applet`** contiene un'unica classe: `Applet`. Questa permette di realizzare applet (argomento trattato nel modulo 15)
- **`java.util`** raccoglie classi d'utilità, come `Date` (trattato nel modulo 12)
- **`java.lang`** è il package che contiene le classi nucleo del linguaggio, come `System` e `String`. (trattato nel modulo 12)

Abbiamo parlato di "package principali", perché Sun li ha dichiarati tali per anni. Probabilmente oggigiorno bisognerebbe aggiungere altri package a questa lista…

3.4.1 Il comando `import`

Per utilizzare una classe della libreria all'interno di una classe che abbiamo intenzione di scrivere, bisogna prima importarla. Supponiamo di voler utilizzare la classe `Date` del package `java.util`. Prima di dichiarare la classe in cui abbiamo intenzione di utilizzare `Date` dobbiamo scrivere:

```
import java.util.Date;
```

oppure, per importare tutte le classi del package `java.util`:

```
import java.util.*;
```

Di default in ogni file Java è importato automaticamente tutto il package `java.lang`, senza il quale non potremmo utilizzare classi fondamentali quali `System` e `String`. Notiamo che questa è una delle caratteristiche che rende Java definibile come "semplice". Quindi, nel momento in cui compiliamo una classe Java, il compilatore anteporrà il comando:

```
import java.lang.*;
```

alla dichiarazione della nostra classe.

L'asterisco non implica l'importazione delle classi appartenenti ai "sottopackage" (per esempio `import java.awt.*` né di `java.awt.event.*`). Quindi l'istruzione `import java.*` non importa tutti i package fondamentali.

Per dare un'idea della potenza e della semplicità di Java, viene presentata di seguito una semplicissima classe. Istanziando qualche oggetto da alcune classi del package `java.awt` (una libreria grafica trattata nel modulo 15) ed assemblandoli con un certo criterio otterremo, con poche righe, una finestra con un bottone. La finestra, sfruttando la libreria `java.awt`, erediterà lo stile grafico del sistema operativo su cui gira. Quindi sarà visualizzato lo stile di Windows XP su Windows XP, lo stile Motif su sistema operativo Solaris e così via. Il lettore può farsi un'idea di come è possibile utilizzare la libreria standard, e della sua potenza, analizzando il seguente codice:

```
import java.awt.*;

public class FinestraConBottone {
  public static void main(String args[]) {
    Frame finestra = new Frame("Titolo");
    Button bottone = new Button("Cliccami");
    finestra.add(bottone);
    finestra.setSize(200,100);
    finestra.setVisible(true);
  }
}
```

Basta conoscere un po' d'inglese per interpretare il significato di queste righe.

La classe `FinestraConBottone` è stata riportata a puro scopo didattico. La pressione del bottone non provocherà nessuna azione, come nemmeno il tentativo di chiudere la finestra. Solo il ridimensionamento della finestra è permesso perché rientra nelle caratteristiche della classe `Frame`. Quindi, per chiudere l'applicazione, bisogna spostarsi sul prompt Dos da dove la si è lanciata e terminare il processo in esecuzione mediante il comando CTRL-C (premere contemporaneamente i tasti "ctrl" e "c"). Se utilizzate EJE, premere il pulsante "interrompi processo".
Anche se l'argomento ha incuriosito, non consigliamo di perdere tempo nel creare interfacce grafiche inutili ed inutilizzabili: bisogna prima imparare Java! Alle interfacce grafiche e alla libreria AWT, è comunque dedicato l'intero modulo 15.

3.4.2 La classe String

In Java le stringhe, a differenza della maggior parte dei linguaggi di programmazione, non sono array di caratteri (`char`), bensì oggetti. Le stringhe, in quanto oggetti, dovrebbero essere istanziate con la solita sintassi tramite la parola chiave `new`. Per esempio:

```
String nome = new String("Mario Rossi");
```

Abbiamo anche sfruttato il concetto di costruttore introdotto nel precedente modulo. Java però, come spesso accade, semplifica la vita del programmatore permettendogli di utilizzare le stringhe, come se si trattasse di un tipo di dato primitivo. Per esempio, possiamo istanziare una stringa nel seguente modo:

```
String nome = "Mario Rossi";
```

che è equivalente a scrivere:

```
String nome = new String("Mario Rossi");
```

Per assegnare un valore ad una stringa bisogna che esso sia compreso tra virgolette, a differenza dei caratteri per cui vengono utilizzati gli apici singoli.

Anche in questo caso possiamo sottolineare la semplicità di Java. Il fatto che sia permesso utilizzare una classe così importante come `String`, come se fosse un tipo di dato primitivo, ci ha permesso d'approcciare i primi esempi di codice senza un ulteriore "trauma", che avrebbe richiesto inoltre l'introduzione del concetto di costruttore.
Il fatto che `String` sia una classe ci garantisce una serie di metodi di utilità, semplici da utilizzare e sempre disponibili, per compiere operazioni con le stringhe. Qualche esempio è rappresentato dai metodi `toUpperCase()`, che restituisce la stringa su cui è chiamato il metodo con ogni carattere maiuscolo (ovviamente esiste anche il metodo `toLowerCase()`), `trim()`, che restituisce la stringa su cui è chiamato il metodo ma senza gli spazi che precedono la prima lettera e quelli che seguono l'ultima, e `equals(String)` che permette di confrontare due stringhe.

La classe `String` è chiaramente una classe particolare. Un'altra caratteristica da sottolineare è che un oggetto `String` è immutabile. I metodi di cui sopra, infatti, non vanno a modificare l'oggetto su cui sono chiamati ma, semmai, ne restituiscono un altro. Per esempio le seguenti righe di codice:

```
String a = "claudio";
String b = a.toUpperCase();
System.out.println(a); // a rimane immutato
System.out.println(b); // b è la stringa maiuscola
```

produrrebbero il seguente output:

```
claudio
CLAUDIO
```

3.4.3 La documentazione della libreria standard di Java

Per conoscere la classe `String` e tutte le altre classi, basta andare a consultare la documentazione. Aprire il file "index.html" che si trova nella cartella "API" della cartella "Docs" del JDK. Se non trovate questa cartella, effettuate una ricerca sul disco rigido. Potreste infatti averla installata in un'altra directory. Se la ricerca fallisce, probabilmente non avete ancora scaricato la documentazione e bisogna scaricarla (http://java.sun.com), altrimenti potete iniziare a studiare anche un altro linguaggio di programmazione! È assolutamente fondamentale che il lettore inizi da subito la sua esplorazione e conoscenza della documentazione. Il vero programmatore Java ha grande familiarità con essa e sa sfruttarne la facilità di consultazione nel modo migliore. In

questo testo, a differenza di altri, saranno affrontati argomenti concernenti le classi della libreria standard, ma in maniera essenziale. Quindi non perderemo tempo nel descrivere tutti i metodi di una classe (tranne in alcuni casi richiesti dall'esame di certificazione SCJP). Piuttosto ci impegneremo a capire quali sono le filosofie alla base dell'utilizzo dei vari package. Questo essenzialmente perché:

1. Riteniamo la documentazione ufficiale insostituibile.
2. Il sito della Sun ed Internet sono fonti inesauribili di informazioni ed esempi.
3. Le librerie sono in continua evoluzione.

Se utilizzate EJE, è possibile consultare la documentazione direttamente da EJE. Questo a patto di installare la cartella "docs" all'interno della cartella del jdk (parallelamente a "bin", "lib", "jre" etc…). Altrimenti è possibile scegliere la posizione della documentazione in un secondo momento.
Molti tool di sviluppo (compreso EJE) permettono la lettura dei metodi di una classe, proponendo una una lista popup ogniqualvolta lo sviluppatore utilizza l'operatore dot. È possibile quindi scrivere il metodo da utilizzare in maniera rapida, semplicemente selezionandolo da questa lista. Ovviamente questo è uno dei bonus che offrono i tool, a cui è difficile rinunciare. Bisogna però sempre ricordarsi di utilizzare metodi solo dopo averne letto la documentazione. "Andare a tentativi", magari fidandosi del nome del metodo, è una pratica assolutamente sconsigliabile. La pigrizia potrebbe costare ore di debug.

La documentazione delle Application Programming Interface (API) di Java è in formato HTML e permette una rapida consultazione. E' completa e spesso esaustiva. Non è raro trovare rimandi a link online, libri o tutorial interni alla documentazione stessa. Nella figura 3.3, viene riportato uno snapshot riguardante la classe `java.awt.Button`.

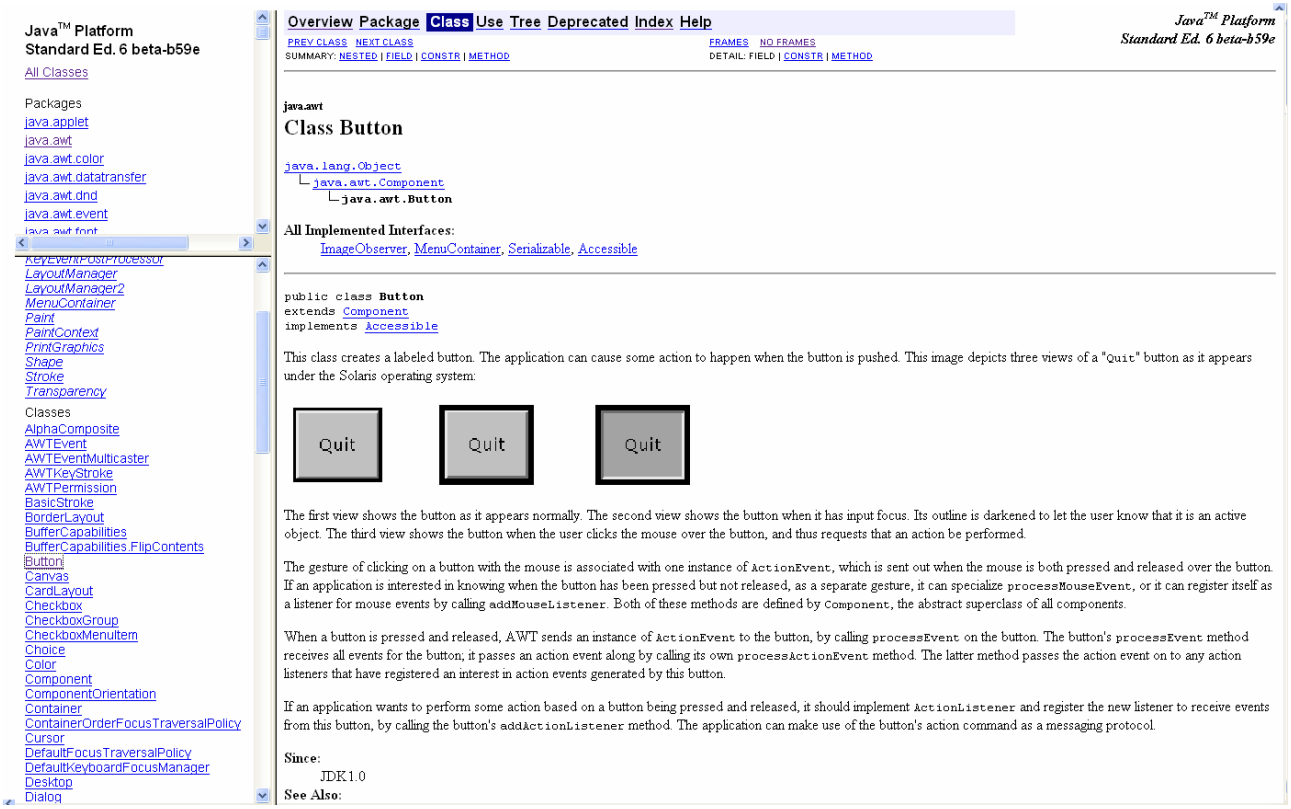


Figura 3.3 – Documentazione ufficiale della classe `java.awt.Button`

Di default, la documentazione appare divisa in tre frame: in alto a sinistra vengono riportati i nomi di tutti i package, in basso a sinistra i nomi di tutte le classi e, nel frame centrale, la descrizione di ciò che è stato richiesto nel frame in basso a sinistra. Questo ipertesto rappresenta uno strumento insostituibile ed è invidiato dai programmatori di altri linguaggi.

3.4.4 Lo strumento javadoc

Abbiamo prima accennato alla possibilità di generare documentazione in formato HTML delle nostre classi, sul modello della documentazione delle classi standard di Java. Infatti, nonostante la mole di informazioni riportate all'interno della documentazione standard, esiste un "trucco" per la generazione automatica: lo strumento "javadoc". Esso permette di generare ipertesti come quello della libreria standard con le informazioni sulle classi che scriveremo. Dobbiamo solamente:

- Scrivere all'interno dell'ipertesto il codice accompagnato dai commenti che devono essere formattati. Bisogna utilizzare commenti del terzo tipo, quelli compresi tra `/**` e `*/`).
- Utilizzare il tool javadoc. E' molto semplice. Dal prompt, digitare:

  javadoc nomeFile.java

  e saranno generati automaticamente tutti i file HTML che servono… provare per credere.

Potremo generare documentazione solo per classi dichiarate `public`. Ovviamente, possiamo commentare classi, metodi, costruttori, variabili, costanti ed interfacce (se dichiarate `public`). Inoltre il commento deve precedere quello che si vuol commentare. Per esempio:

```
/**
 Questo è un metodo!
*/
public void metodo(){
  . . .
}
```

Se si utilizza EJE è possibile generare la documentazione javadoc semplicemente cliccando sul bottone apposito. La documentazione verrà generata in una cartella "docs" creata al volo nella cartella dove si trova il file sorgente.

3.4.5 Gli array in Java

Un array è una collezione di tipi di dati primitivi, o di reference, o di altri array. Gli array permettono di utilizzare un solo nome per individuare una collezione costituita da vari elementi, che saranno accessibili tramite indici interi. In Java gli array sono, in quanto collezioni, oggetti.
Per utilizzare un array bisogna passare attraverso tre fasi: dichiarazione, creazione ed inizializzazione (come per gli oggetti).

3.4.6 Dichiarazione

Di seguito presentiamo due dichiarazioni di array. Nella prima dichiariamo un array di `char` (tipo primitivo), nella seconda dichiariamo un array di istanze di `Button` (classe appartenente al package `java.awt`):

```
char alfabeto [];        oppure        char [] alfabeto;
Button bottoni [];       oppure        Button [] bottoni;
```

In pratica, per dichiarare un array, basta posporre (oppure anteporre) una coppia di parentesi quadre all'identificatore.

3.4.7 Creazione

Un array è un oggetto speciale in Java e, in quanto tale, va istanziato in modo speciale. La sintassi è la seguente:

```
alfabeto = new char[21];
bottoni = new Button[3];
```

Come si può notare, è obbligatorio specificare al momento dell'istanza dell'array la dimensione dell'array stesso. A questo punto però tutti gli elementi dei due array sono inizializzati automaticamente ai relativi valori nulli. Vediamo allora come inizializzare esplicitamente gli elementi dell'array.

8.1.1. 3.4.8 Inizializzazione

Per inizializzare un array, bisogna inizializzarne ogni elemento singolarmente:

```
alfabeto [0] = 'a';
alfabeto [1] = 'b';
alfabeto [2] = 'c';
alfabeto [3] = 'd';
. . . . . .
alfabeto [20] = 'z';
```

```
bottoni [0] = new Button();
bottoni [1] = new Button();
bottoni [2] = new Button();
```

L'indice di un array inizia sempre da zero. Quindi un array dichiarato di 21 posti, avrà come indice minimo `0` e massimo `20` (un array di dimensione `n` implica il massimo indice a `n-1`)
Il lettore avrà sicuramente notato che può risultare alquanto scomodo inizializzare un array in questo modo, per di più dopo averlo prima dichiarato ed istanziato. Ma Java ci viene incontro dando la possibilità di eseguire tutti e tre i passi principali per creare un array tramite una particolare sintassi che di seguito presentiamo:

```
char alfabeto [] = {'a', 'b', 'c', 'd', 'e',…, 'z'};
Button bottoni [] = { new Button(), new Button(), new
Button()} ;
```

Si noti la differenza tra un array dichiarato di tipo di dato primitivo o complesso. Il primo contiene direttamente i suoi elementi. Il secondo contiene solo reference, non gli elementi stessi.
Esiste anche una variabile chiamata `length` che, applicata ad un array, restituisce la dimensione effettiva dell'array stesso. Quindi:

```
    alfabeto.length
```

varrà 21.

Solitamente uno dei vantaggi che porta l'uso di array è sfruttare l'indice all'interno di un ciclo. I cicli saranno trattati nel prossimo modulo.

3.4.9 Array Multidimensionali

Esistono anche array multidimensionali, che sono array di array. A differenza della maggior parte degli altri linguaggi di programmazione, in Java quindi, un array bidimensionale non deve per forza essere rettangolare. Di seguito è presentato un esempio:

```
int arrayNonRettangolare [][] = new int[4][];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
```

```
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . . .
arrayNonRettangolare [3][7] = 10;
```

oppure, equivalentemente:

```
int arrayNonRettangolare [][] = {
    {1,2},
    {1,0,0,0},
    {0,0,0,0,0,0},
    {0,0,0,0,0,0,0,10}
};
```

3.4.10 Limiti degli array

Essenzialmente gli array sono caratterizzati da due limitazioni:

1. Non sono eterogenei. Per esempio un array di `Button` deve contenere solo reference ad oggetti `Button`.
2. Non sono ridimensionabili. Trattasi di oggetti e quindi il seguente codice:

   ```
   int mioArray [] = {1, 2, 3, 4};
   mioArray = new int[6];
   ```

   non copia il contenuto del primo array nel secondo, ma semplicemente assegna al reference una nuova istanza di array.

Entrambi questi problemi possono essere superati. Il polimorfismo (argomento affrontato nel Modulo 6) permetterà di creare collezioni eterogenee, ovvero array che contengono oggetti di tipo diverso. Inoltre il metodo statico `arraycopy()` della classe `System`, benché scomodo da utilizzare, risolve il secondo problema. Infine, la libreria fornisce una serie di classi (e di interfacce) note sotto il nome di "Collections", che astraggono proprio l'idea di collezioni eterogenee ridimensionabili. Le Collections saranno ampiamente trattate nel Modulo 12.

Riepilogo

In questo modulo abbiamo introdotto alcune caratteristiche fondamentali del linguaggio e imparato ad utilizzare alcune convenzioni (o regole di buona programmazione) per il codice. Abbiamo visto l'importanza dell'indentazione del codice e delle convenzioni per gli identificatori. Abbiamo non solo studiato gli otto tipi di dati primitivi di Java, ma anche alcuni dei problemi relativi ad essi, e i concetti di casting e promotion. Fondamentale è stata la discussione sul concetto di reference, che il lettore deve aver appreso correttamente per non incontrare problemi nel seguito del suo studio. Inoltre sono stati introdotti la libreria standard, la sua documentazione, il comando javadoc, e abbiamo trattato una classe fondamentale come la classe `String`. Infine abbiamo descritto gli array con le loro caratteristiche e i loro limiti.

Esercizi modulo 3

Esercizio 3.a)

Scrivere un semplice programma che svolga le seguenti operazioni aritmetiche correttamente, scegliendo accuratamente i tipi di dati da utilizzare per immagazzinare i risultati di esse.
Una divisione tra due interi `a = 5`, e `b = 3`. Immagazzinare il risultato in una variabile `r1`, scegliendone il tipo di dato appropriatamente.
Una moltiplicazione tra un `char c = 'a'`, ed uno `short s = 5000`. Immagazzinare il risultato in una variabile `r2`, scegliendone il tipo di dato appropriatamente.
Una somma tra un `int i = 6` ed un `float f = 3.14F`. Immagazzinare il risultato in una variabile `r3`, scegliendone il tipo di dato appropriatamente.
Una sottrazione tra `r1`, `r2` e `r3`. Immagazzinare il risultato in una variabile `r4`, scegliendone il tipo di dato appropriatamente.
Verificare la correttezza delle operazioni stampandone i risultati parziali ed il risultato finale. Tenere presente la promozione automatica nelle espressioni, ed utilizzare il casting appropriatamente.

Basta una classe con un `main()` che svolge le operazioni.

Esercizio 3.b)

Scrivere un programma con i seguenti requisiti.
Utilizza una classe `Persona` che dichiara le variabili `nome`, `cognome`, `eta` (età). Si dichiari inoltre un metodo `dettagli()` che restituisce in una stringa le informazioni sulla persona in questione. Ricordarsi di utilizzare le convenzioni e le regole descritte in questo modulo.
Utilizza una classe `Principale` che, nel metodo `main()`, istanzia due oggetti chiamati `persona1` e `persona2` della classe `Persona`, inizializzando per ognuno di essi i relativi campi con sfruttamento dell'operatore dot.
Dichiarare un terzo reference (`persona3`) che punti ad uno degli oggetti già istanziati. Controllare che effettivamente `persona3` punti allo oggetto voluto, stampando i campi di `persona3` sempre mediante l'operatore dot.
Commentare adeguatamente le classi realizzate e sfruttare lo strumento javadoc per produrre la relativa documentazione.

Nella documentazione standard di Java sono usate tutte le regole e le convenzioni descritte in questo capitolo. Basta osservare che `String` inizia con lettera maiuscola, essendo una classe. Si può concludere che anche `System` è una classe.

Esercizio 3.c)
Array, Vero o Falso:

1. Un array è un oggetto e quindi può essere dichiarato, istanziato ed inizializzato.
2. Un array bidimensionale è un array i cui elementi sono altri array.
3. Il metodo `length` restituisce il numero degli elementi di un array.
4. Un array non è ridimensionabile.
5. Un array è eterogeneo di default.
6. Un array di interi può contenere come elementi `byte`, ovvero le seguenti righe di codice non producono errori in compilazione:
```
int arr [] = new int[2];
byte a = 1, b=2;
arr [0] = a;arr [1] = b;
```
7. Un array di interi può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:
```
char a = 'a', b = 'b';
int arr [] = {a,b};
```
8. Un array di stringhe può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:
```
String arr [] = {'a' , 'b'};
```
9. Un array di stringhe è un array bidimensionale, perché le stringhe non sono altro che array di caratteri. Per esempio:
```
String arr [] = {"a" , "b"};
```
è un array bidimensionale.
10. Se abbiamo il seguente array bidimensionale:
```
int arr [][]= {
  {1, 2, 3},
  {1,2},
  {1,2,3,4,5}
};
```
risulterà che:
```
arr.length = 3;
    arr[0].length = 3;
    arr[1].length = 2;
    arr[2].length = 5;
```

```
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 1;
arr[1][1] = 2;
arr[1][2] = 3;
arr[2][0] = 1;
arr[2][1] = 2;
arr[2][2] = 3;
arr[2][3] = 4;
arr[2][4] = 5;
```

Soluzioni esercizi modulo 3

Esercizio 3.a)

```
public class Esercizio3A {
  public static void main (String args[]) {
    int a = 5, b = 3;
    double r1 = (double)a/b;
    System.out.println("r1 = " + r1);
    char c = 'a';
    short s = 5000;
    int r2 = c*s;
    System.out.println("r2 = " + r2);
    int i = 6;
    float f = 3.14F;
    float r3 = i + f;
    System.out.println("r3 = " + r3);
    double r4 = r1 - r2 - r3;
    System.out.println("r4 = " + r4);
  }
}
```

Esercizio 3.b)

```
  public class Persona {
    public String nome;
    public String cognome;
    public int eta;
    public String dettagli() {
      return nome + " " + cognome + " anni " + eta;
    }
  }
  public class Principale {
    public static void main (String args []) {
      Persona persona1 = new Persona();
      Persona persona2 = new Persona();
      persona1.nome = "Mario";
      persona1.cognome = "Rossi";
      persona1.eta = 30;
      System.out.println("persona1 "
```

```
            +persona1.dettagli());
        persona2.nome = "Giuseppe";
        persona2.cognome = "Verdi";
        persona2.eta = 40;
        System.out.println("persona2 "
            +persona2.dettagli());
        Persona persona3 = persona1;
        System.out.println("persona3 "
            +persona3.dettagli());
    }
}
```

Esercizio 3.c)
Array, Vero o Falso:

1. **Vero.**
2. **Vero**.
3. **Falso** la variabile `length` restituisce il numero degli elementi di un array.
4. **Vero.**
5. **Falso.**
6. **Vero** un `byte` (che occupa solo 8 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
7. **Vero** un `char` (che occupa 16 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
8. **Falso** un `char` è un tipo di dato primitivo e `String` è una classe. I due tipi di dati non sono compatibili.
9. **Falso** in Java la stringa è una oggetto istanziato dalla classe `String` e non un array di caratteri.
10. **Falso** tutte le affermazioni sono giuste tranne `arr[1][2] = 3;` perché questo elemento non esiste.

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper utilizzare le convenzioni per il codice Java (unità 3.1).	☐	
Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.2).	☐	
Saper gestire casting e promotion (unità 3.2).	☐	
Saper utilizzare i reference e capirne la filosofia (unità 3.4).	☐	
Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.4).	☐	
Saper utilizzare la classe String (unità 3.4).	☐	
Saper utilizzare gli array (unità 3.5).	☐	

Note:

4

Complessità: bassa

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Conoscere e saper utilizzare i vari operatori (unità 4.1).
2. Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2, 4.3).
3. Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.2, 4.4).

4. Operatori e Gestione del flusso di esecuzione

4.1 Operatori di base

Di seguito è presentata una lista completa degli operatori che Java mette a disposizione.

> **Java eredita in blocco tutti gli operatori del linguaggio C e quindi, per alcuni di essi, l'utilizzo è alquanto raro. Dunque non analizzeremo in dettaglio tutti gli operatori, anche se per ognuno c'è una qualche trattazione su questo testo, dal momento che si tratta sempre di argomenti richiesti per superare l'esame di certificazione SCJP.**

4.1.1 Operatore d'assegnazione

L'operatore = non ha bisogno di essere commentato.

4.1.2 Operatori aritmetici

La seguente tabella riassume gli operatori aritmetici semplici definiti dal linguaggio:

Descrizione	Operatore
Somma	+

Sottrazione	**-**
Moltiplicazione	*****
Divisione	**/**
Modulo	**%**

L'unico operatore che può risultare non familiare al lettore è l'operatore modulo.
Il risultato dell'operazione modulo tra due numeri coincide con il resto della divisione fra essi. Per esempio:

```
5 % 3 = 2
10 % 2 = 0
100 % 50 = 0
```

Java ha ereditato dalla sintassi del linguaggio C anche altri operatori, sia binari (con due operandi) che unari (con un solo operando). Alcuni di essi, oltre a svolgere un'operazione, assegnano anche il valore del risultato ad una variabile utilizzata nell'operazione stessa:

Descrizione	Operatore
Somma ed assegnazione	**+=**
Sottrazione ed assegnazione	**-=**
Moltiplicazione ed assegnazione	***=**
Divisione ed assegnazione	**/=**
Modulo ed assegnazione	**%=**

In pratica se abbiamo:

```
int i = 5;
```

scrivere:

```
i = i + 2;
```

è equivalente a scrivere:

```
i += 2;
```

4.1.3 Operatori (unari) di pre e post-incremento (e decremento)

La seguente tabella descrive questa singolare tipologia di operatori:

Descrizione	Operatore	Esempio
Pre-incremento di un'unità	**++**	**++i**
Pre-decremento di un'unità	**--**	**--i**
Post-incremento di un'unità	**++**	**i++**
Post-decremento di un'unità	**--**	**i--**

Se vogliamo incrementare di una sola unità una variabile numerica, possiamo equivalentemente scrivere:

```
i = i + 1;
```

oppure:

```
i += 1;
```

ma anche:

```
i++;
```

oppure:

```
++i;
```

ottenendo comunque lo stesso risultato. Infatti, in tutti i casi, otterremo che il valore della variabile `i` è stato incrementato di un'unità, ed assegnato nuovamente alla variabile stessa. Quindi anche questi operatori svolgono due compiti (incremento ed assegnazione). Parleremo di operatore di pre-incremento nel caso in cui anteponiamo l'operatore d'incremento ++ alla variabile. Parleremo, invece, di operatore di post-incremento nel caso in cui posponiamo l'operatore di incremento alla variabile. La differenza tra questi due operatori "composti" consiste essenzialmente nelle priorità che essi hanno rispetto all'operatore di assegnazione. L'operatore di pre-incremento ha maggiore priorità dell'operatore di assegnazione =. L'operatore di post-incremento ha minor priorità rispetto all'operatore di assegnazione =. Ovviamente le stesse regole valgono per gli operatori di decremento.

Facciamo un paio di esempi per rendere evidente la differenza tra i due operatori. Il seguente codice utilizza l'operatore di pre-incremento:

```
x = 5;
y = ++x;
```

Dopo l'esecuzione delle precedenti istruzioni avremo che:

```
x=6
y=6
```

Il seguente codice invece utilizza l'operatore di post-incremento:

```
x = 5;
y = x++;
```

In questo caso avremo che:

```
x = 6
y = 5
```

4.1.4 Operatori bitwise

La seguente tabella mostra tutti gli operatori bitwise (ovvero che eseguono operazioni direttamente sui bit) definiti in Java:

Descrizione	Operatore
NOT	**~**
AND	**&**
OR	**|**
XOR	**^**
Shift a sinistra	**<<**
Shift a destra	**>>**
Shift a destra senza segno	**>>>**
AND e assegnazione	**&=**
OR e assegnazione	**|=**
XOR e assegnazione	**^=**
Shift a sinistra e assegnazione	**<<=**
Shift a destra e assegnazione	**>>=**

Shift a destra senza segno e assegnazione	**>>=**

Tutti questi operatori binari sono molto efficienti giacché agiscono direttamente sui bit, ma in Java si utilizzano raramente. Infatti in Java non esiste l'aritmetica dei puntatori e, di conseguenza, lo sviluppatore non è abituato a "pensare in bit".
L'operatore NOT ~ è un operatore unario, dato che si applica ad un solo operando. Per esempio, sapendo che la rappresentazione binaria di 1 è 00000001, avremo che ~1 varrà 11111110 ovvero -2.
Tale operatore, applicato ad un numero intero, capovolgerà la rappresentazione dei suoi bit scambiando tutti gli 0 con 1 e viceversa.
Gli operatori `&`, |, e ^, si applicano a coppie di operandi, e svolgono le relative operazioni logiche di conversioni di bit riassunte nelle seguente tabella della verità:

Operando1	**Operando2**	**Op1** AND **Op2**	**Op1** OR**O p2**	**Op1** XOR **Op2**
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**1**
1	**1**	**1**	**1**	**0**

Gli operatori di shift (operatori di scorrimento) provocano lo scorrimento di un certo numero di bit verso una determinata direzione, quella della rappresentazione binaria del dato in questione. Il numero dei bit da scorrere è rappresentato dall'operando a destra dell'operazione. I bit che dopo lo scorrimento si trovano al di fuori della rappresentazione binaria del numero vengono eliminati. I bit che invece "rimangono vuoti" vengono riempiti con i valori 0 oppure 1 a seconda del caso. In particolare, lo scorrimento a sinistra provoca un riempimento con i valori 0 dei bit lasciati vuoti sulla destra della rappresentazione binaria del numero. Anche lo scorrimento a destra senza segno riempie i bit lasciati vuoti con degli 0. Lo scorrimento a destra con segno, invece, provoca il riempimento di 0 oppure di 1, a seconda che l'ultima cifra a sinistra prima dello scorrimento (bit del segno) sia 0 oppure 1, ovvero che la cifra prima dello scorrimento sia positiva o negativa. Consideriamo i seguenti esempi, se abbiamo:

```
byte a = 35; //rappresentazione binaria 00100011
```

e shiftiamo (scorriamo) a destra di due posizioni:

```
a = a >> 2;
```

avremo che:

```
a = 8 // rappresentazione binaria 00001000
```

Se invece abbiamo:

```
byte b = -8; //rappresentazione binaria 11111000
```

e shiftiamo b di una posizione:

```
b = b >> 1;
```

avremo che:

```
b = -4 //rappresentazione binaria 11111100
```

Facciamo ora un esempio di scorrimento a destra senza segno:

```
int a = -1;
a = a >>> 24;
11111111111111111111111111111111 ovvero  -1
>>> 24
00000000000000000000000011111111 ovvero  255
```

Ricordiamo che la promozione automatica delle espressioni avviene per ogni operatore binario e quindi anche per l'operatore di scorrimento a destra senza segno.

L'operazione di scorrimento a destra equivale a dividere l'operando di sinistra per 2 elevato all'operando situato alla destra nell'espressione. Il risultato viene arrotondato per difetto nelle operazioni con resto. Ovvero:
op1 >> op2 equivale a op1 diviso (2 elevato a op2) ;
Similmente, l'operazione di scorrimento a sinistra equivale a moltiplicare l'operando di sinistra per 2 elevato all'operando situato sulla destra dell'operazione. Ovvero:

`op1 << op2` equivale a `op1` moltiplicato (`2` elevato a `op2`).

4.1.5 Operatori relazionali o di confronto

Il risultato delle operazioni basate su operatori relazionali è sempre un valore `boolean`, ovvero `true` o `false`.

Operatore	*Simbolo*	*Applicabilità*
Uguale a	`==`	**Tutti i tipi**
Diverso da	`!=`	**Tutti i tipi**
Maggiore	`>`	**Solo i tipi numerici**
Minore	`<`	**Solo i tipi numerici**
Maggiore o uguale	`>=`	**Solo i tipi numerici**
Minore o uguale	`<=`	**Solo i tipi numerici**

Un classico errore che l'aspirante programmatore commette spesso è scrivere **`=`** in luogo di **`==`**.

Se confrontiamo due reference con l'operatore ==, il risultato risulterà `true` se e solo se i due reference puntano allo stesso oggetto, altrimenti `false` . Infatti viene confrontato sempre il valore delle variabili in gioco. Il valore di una variabile reference, come sappiamo (cfr. Modulo 3), è l'indirizzo in memoria dell'oggetto a cui punta. Ecco spiegato il perché.

La classe `String`, di cui abbiamo già parlato nel precedente modulo, gode di una singolare particolarità che si può notare con l'utilizzo dell'operatore `==`. Abbiamo visto come sia possibile istanziare la classe `String` come un tipo di dato primitivo, per esempio:

```
String linguaggio = "Java";
```

Le istanze così ottenute, però, sono trattate diversamente da quelle che vengono istanziate con la sintassi tradizionale:

```
String linguaggio = new String("Java");
```

Infatti le istanze che vengono create senza la parola chiave `new`, come se `String` fosse un tipo di dato primitivo, vengono poi poste in un speciale pool di stringhe (un insieme speciale) dalla Java Virtual Machine e riutilizzate, al fine di migliorare le prestazioni. Questo significa che il seguente frammento di codice:

```
String a = "Java";
String b = "Java";
String c = new String("Java");
System.out.println(a==b);
System.out.println(b==c);
```

produrrà il seguente output:

```
true
false
```

Infatti `a` e `b` punteranno esattamente allo stesso oggetto. Mentre nel caso di `c`, avendo utilizzato la parola chiave `new`, verrà creato un oggetto ex novo, anche se con lo stesso contenuto dei precedenti. Ricordiamo che il giusto modo per confrontare due stringhe rimane l'utilizzo del metodo `equals()`. Infatti il seguente frammento di codice:

```
System.out.println(a.equals(b));
System.out.println(b.equals(c));
```

produrrà il seguente output:

```
true
true
```

A tal proposito è bene ricordare che la classe `String` definisce un metodo chiamato `intern()`, che prova a recuperare proprio l'oggetto `String` dal pool di stringhe, utilizzando il confronto che fornisce il metodo `equals()`. Nel caso nel pool non esista la stringa desiderata, questa viene aggiunta, e viene restituito un reference ad essa. In pratica, date due stringhe `t` ed `s`:

```
s.intern() == t.intern()
```

è `true` se se solo se

```
s.equals(t)
```

vale `true`.

4.1.6 Operatori logico – booleani

Quelli seguenti sono operatori che utilizzano solo operandi di tipo booleano. Il risultato di un'operazione basata su tali operatori è di tipo `boolean`:

Descrizione	Operatore
NOT logico	**!**
AND logico	**&**

OR logico	**|**
XOR logico	**^**
Short circuit AND	**&&**
Short circuit OR	**||**
AND e assegnazione	**&=**
OR e assegnazione	**|=**
XOR e assegnazione	**^=**

È facile trovare punti di contatto tra la precedente lista di operatori e la lista degli operatori
bitwise. Gli operandi a cui si applicano gli operatori booleani però possono essere solo di tipo booleano.

E' consuetudine utilizzare le versioni short circuit ("corto circuito") di AND ed OR. Ad esempio, la seguente riga di codice mostra come avvantaggiarsi della valutazione logica di corto circuito:
boolean flag = ((a != 0) && (b/a > 10))
Affinché l'espressione tra parentesi sia vera, bisogna che entrambi gli operandi dell'AND siano veri. Se il primo tra loro è in partenza falso, non ha senso andare a controllare la situazione dell'altro operando. In questo caso sarebbe addirittura dannoso perché porterebbe ad una divisione per zero (errore riscontrabile solo al runtime e non in fase di compilazione). Quest'operatore short circuit, a differenza della sua versione tradizionale (&), fa evitare il secondo controllo in caso di fallimento del primo. Equivalentemente l'operatore short circuit ||, nel caso la prima espressione da testare risultasse verificata, convalida l'intera espressione senza alcuna altra (superflua) verifica.

4.1.7 Concatenazione di stringhe con +

In Java l'operatore +, oltre che essere un operatore aritmetico, è anche un operatore per concatenare stringhe. Per esempio il seguente frammento di codice:

```
String nome = "James ";
String cognome = "Gosling";
String nomeCompleto = "Mr. " + nome + cognome;
```

farà in modo che la stringa `nomeCompleto`, abbia come valore *`Mr. James Gosling`*. Se "sommiamo" un qualsiasi tipo di dato con una stringa, il tipo di dato sarà automaticamente convertito in stringa, e ciò può spesso risultare utile.

Il meccanismo di concatenazione di stringhe viene in realtà gestito dietro le quinte dalla Java Virtual Machine, sfruttando oggetti di tipo `StringBuffer`. Si tratta di una classe che rappresenta una stringa ridimensionabile, mettendo a disposizione un metodo come `append()` (in italiano "aggiungi"). In pratica, per eseguire la concatenazione del precedente esempio, la JVM istanzierà uno `StringBuffer` e sfrutterà il metodo `append()` per realizzare la concatenazione. Infine dovrà convertire lo `StringBuffer` nuovamente in `String`. Per tale motivo quindi, laddove l'applicazione che stiamo scrivendo debba avere particolare riguardo per le prestazioni, è consigliabile utilizzare direttamente la classe `StringBuffer`, per evitare lavoro inutile alla JVM. Tuttavia siamo sicuri che nella maggior parte dei casi la differenza non si riuscirà a notare…
Intanto consigliamo vivamente di consultare la documentazione della classe `StringBuffer` che si trova nel package `java.lang`.
Dalla versione 5 di Java, inoltre, è stata introdotta anche una nuova classe: `StringBuilder`. Questa è del tutto simile a `StringBuffer` e ne consigliamo la consultazione della documentazione solo se il lettore ha già studiato il modulo 11 relativo ai thread…

4.1.8 Priorità degli operatori

Nella seguente tabella sono riportati, in ordine di priorità, tutti gli operatori di Java. Alcuni di essi non sono ancora stati trattati.

Separatori	**. [] () ; ,**
da sx a dx	**++ -- + - ~ ! (tipi di dati)**
Da sx a dx	*** / %**
Da sx a dx	**+ -**
Da sx a dx	**<< >> >>>**

Da sx a dx	**< > <= >= instanceof**
Da sx a dx	**== !=**
Da sx a dx	**&**
Da sx a dx	^
Da sx a dx	|
Da sx a dx	**&&**
Da sx a dx	||
Da dx a sx	**?:**
da dx a sx	**= *= /= %= += -= <<= >>= >>>= &= ^= |=**

Fortunatamente non è necessario conoscere a memoria tutte le priorità per programmare. Nell'incertezza è sempre possibile utilizzare le parentesi tonde così come faremmo nell'aritmetica tradizionale. Ovviamente, non potendo usufruire di parentesi quadre e graffe, poiché in Java queste sono adoperate per altri scopi, sostituiremo il loro utilizzo sempre con parentesi tonde.
Quindi, se abbiamo le seguenti istruzioni:

```
int a = 5 + 6 * 2 - 3;
int b = (5 + 6) * (2 - 3);
int c = 5 + (6 * (2 - 3));
```

le variabili `a`, `b` e `c` varranno rispettivamente `14`, `-11` e `-1`.

4.2 Gestione del flusso di esecuzione

In ogni linguaggio di programmazione esistono costrutti che permettono all'utente di controllare la sequenza delle istruzioni immesse. Essenzialmente possiamo dividere questi costrutti in due categorie principali:

1. **Condizioni (o strutture di controllo decisionali)**: permettono, durante la fase di runtime, una scelta tra l'esecuzione di istruzioni diverse, a seconda che sia verificata una specificata condizione.

2. **Cicli (strutture di controllo iterative)**: permettono, in fase di runtime, di decidere il numero di esecuzioni di determinate istruzioni.

In Java le condizioni che si possono utilizzare sono essenzialmente due: il costrutto `if` ed il costrutto `switch`, cui si aggiunge l'operatore ternario. I costrutti di tipo ciclo invece sono tre: `while`, `for` e `do`. Dalla versione 5 di Java è stato introdotto un quarto tipo di ciclo chiamato "ciclo for migliorato".
Tutti i costrutti possono essere annidati. I costrutti principali (almeno da un punto di vista storico) sono la condizione `if` ed il ciclo `while`. Un programmatore in grado di utilizzare questi due costrutti sarà in grado di codificare un qualsiasi tipo di istruzione. La sintassi di questi due costrutti è alquanto banale e per questo vengono anche detti "costrutti di programmazione semplici".

4.3 Costrutti di programmazione semplici

In questo paragrafo studieremo i costrutti `if` e `while`. Inoltre introdurremo un operatore che non abbiamo ancora studiato, detto operatore ternario.

4.3.1 Il costrutto `if`

Questa condizione permette di prendere semplici decisioni basate su valori immagazzinati. In fase di runtime la Java Virtual Machine testa un'espressione booleana e, a seconda che essa risulti vera o falsa, esegue un certo blocco di istruzioni, oppure no. Un'espressione booleana è un'espressione che come risultato può restituire solo valori di tipo `boolean`, vale a dire `true` o `false`. Essa di solito si avvale di operatori di confronto e, magari, di operatori logici. La sintassi è la seguente:

```
if (espressione-booleana) istruzione;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
```

Nell'esempio, l'istruzione di stampa sarebbe eseguita se e solo se la variabile `numeroLati` avesse valore 3. In quel caso l'espressione booleana `numeroLati == 3` varrebbe `true` e quindi sarebbe eseguita l'istruzione che segue l'espressione. Se invece l'espressione risultasse `false`, sarebbe eseguita direttamente la prima eventuale

istruzione che segue l'istruzione di stampa. Possiamo anche estendere la potenzialità del costrutto `if` mediante la parola chiave `else`:

```
if (espressione-booleana) istruzione1;
else istruzione2;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
else
    System.out.println("Questo non è un triangolo");
```

`if` potremmo tradurlo con "se"; `else` con "altrimenti". In pratica, se l'espressione booleana è vera verrà stampata la stringa "Questo è un triangolo"; se è falsa verrà stampata la stringa "Questo non è un triangolo".
Possiamo anche utilizzare blocchi di codice, con il seguente tipo di sintassi:

```
if (espressione-booleana) {
    istruzione_1;
    ..........;
    istruzione_k;
} else {
    istruzione_k+1;
    ............;
    istruzione_n;
}
```

ed anche comporre più costrutti nel seguente modo

```
if (espressione-booleana) {
    istruzione_1;
    ..........;
    istruzione_k;
} else if (espressione-booleana){
    istruzione_k+1;
    ............;
    istruzione_j;
```

```
} else if (espressione-booleana){
    istruzione_j+1;
    .............;
    istruzione_h;
} else {
    istruzione_h+1;
    .............;
    istruzione_n;
}
```

Possiamo anche annidare questi costrutti. I due seguenti frammenti di codice possono sembrare equivalenti. In realtà il frammento di codice a sinistra mostra un `if` che annida un costrutto `if – else`. Il frammento di codice a destra invece mostra un costrutto `if – else` che annida un costrutto `if`:

```
. . .
if (x != 0)
    if (y < 10)
        z = 5;
    else
        z = 7;
. . .
```

```
. . .
if (x != 0){
    if (y < 10)
        z = 5;
    } else
        z = 7;
. . .
```

Si consiglia sempre di utilizzare un blocco di codice per circondare anche un'unica istruzione. Infatti, questa pratica aggiunge leggibilità. Inoltre capita spesso di aggiungere istruzioni in un secondo momento…

4.3.2 L'operatore ternario

Esiste un'operatore non ancora trattato qui, che qualche volta può sostituire il costrutto `if`. Si tratta del cosiddetto **operatore ternario** (detto anche **operatore condizionale**), che può regolare il flusso di esecuzione come una condizione. Di seguito si può leggerne la sintassi:

```
variabile = (espr-booleana) ? espr1 : espr2;
```

dove se il valore della `espr-booleana` è `true` si assegna a `variabile` il valore di `espr1;` altrimenti si assegna a `variabile` il valore di `espr2`.

Requisito indispensabile è che il tipo della `variabile` e quello restituito da `espr1` ed `espr2` siano compatibili. E' escluso il tipo `void`.
L'operatore ternario non può essere considerato un sostituto del costrutto `if`, ma è molto comodo in alcune situazioni. Il seguente codice:

```
String query = "select * from table " +
    (condition != null ? "where " + condition : "");
```

crea una stringa contenente una query SQL (per il supporto che Java offre a SQL cfr. Modulo 14) e, se la stringa `condition` è diversa da `null`, aggiunge alla query la condizione.

4.3.3 Il costrutto while

Questo ciclo permette di iterare uno statement (o un insieme di statement compresi in un blocco di codice) tante volte fino a quando una certa condizione booleana è verificata.
La sintassi è la seguente:

```
[inizializzazione;]
while (espr. booleana) {
    corpo;
    [aggiornamento iterazione;]
}
```

Come esempio proponiamo una piccola applicazione che stampa i primi dieci numeri:

```
public class WhileDemo {
    public static void main(String args[]) {
        int i = 1;
        while (i <= 10) {
            System.out.println(i);
            i++;
        }
    }
}
```

Analizziamo in sequenza le istruzioni che verrebbero eseguite in fase di runtime. Viene in primo luogo dichiarata ed inizializzata a `1` una variabile intera `i`. Poi inizia il ciclo, in

cui è esaminato il valore booleano dell'espressione in parentesi. Siccome `i` è uguale ad `1`, `i` è anche minore di `10` e la condizione è verificata. Quindi viene eseguito il blocco di codice relativo, nel quale prima sarà stampato il valore della variabile `i` (ovvero `1`) e poi verrà incrementata la variabile stessa di un'unità. Terminato il blocco di codice, verrà nuovamente testato il valore dell'espressione booleana. Durante questo secondo tentativo, la variabile `i` varrà `2`. Quindi, anche in questo caso, sarà eseguito di nuovo il blocco di codice. Verrà allora stampato il valore della variabile `i` (ovvero `2`) ed incrementata nuovamente di una unità, la variabile stessa. Questo ragionamento si ripete fino a quando la variabile `i` non assume il valore `11`. Quando ciò accadrà, il blocco di codice non verrà eseguito, dal momento che l'espressione booleana non sarà verificata. Il programma quindi eseguirà le istruzioni successive al blocco di codice e quindi terminerà.

4.4 Costrutti di programmazione avanzati

In questo paragrafo ci occuperemo di tutti gli altri costrutti di programmazione che regolano il flusso di un'applicazione.

4.4.1 Il costrutto `for`

Ecco la sintassi per il `for`, nel caso d'utilizzo di una o più istruzioni da iterare.
una istruzione:

```
for (inizializzazione; espr. booleana; aggiornamento)
istruzione;
```

più istruzioni:

```
for (inizializzazione; espr. booleana; aggiornamento)
{
    istruzione_1;
    ..........;
    istruzione_n;
}
```

Il consiglio è sempre di utilizzare comunque i blocchi di codice anche nel caso di istruzione singola.

Presentiamo un esempio che stampa i primi 10 numeri partendo da `10` e terminando ad `1`:

```
public class ForDemo
{
    public static void main(String args[])
    {
        for (int n = 10; n > 0; n--)
        {
            System.out.println(n);
        }
    }
}
```

In questo caso notiamo che la sintassi è più compatta rispetto a quella relativa al `while`. Tra le parentesi tonde relative ad un ciclo `for`, dichiariamo addirittura una variabile locale `n` (che smetterà di esistere al termine del ciclo). Potevamo anche dichiararla prima del ciclo, nel caso fosse stata nostra intenzione utilizzarla anche fuori da esso. Per esempio, il seguente codice definisce un ciclo che utilizza una variabile dichiarata esternamente a esso, e utilizzata nel ciclo e fuori:

```
public void forMethod(int j)
{
    int i = 0;
    for (i = 1; i < j; ++i)
    {
        System.out.println(i);
    }
    System.out.println("Numero iterazioni = " + i);
}
```

La sintassi del ciclo `for` è quindi molto flessibile. In pratica, se nel `while` utilizziamo le parentesi tonde solo per l'espressione booleana, nel `for` le utilizziamo per inserirci rispettivamente prima l'inizializzazione di una variabile, poi l'espressione booleana ed infine l'aggiornamento che sarà eseguito ad ogni iterazione. Si noti che queste tre istruzioni possono anche essere completamente indipendenti tra loro.
Potremmo anche dichiarare più variabili all'interno, più aggiornamenti e, sfruttando operatori condizionali, anche più condizioni. Per esempio il seguente codice è valido:

```
public class For {
    public static void main(String args[]) {
        for (int i = 0, j = 10; i < 5 || j > 5; i++, j--)
{
            System.out.println("i="+i);
            System.out.println("j="+j);
        }
    }
}
```

Come è possibile notare, le dichiarazioni vanno separate da virgole, ed hanno il vincolo di dover essere tutte dello stesso tipo (in questo caso `int`). Anche gli aggiornamenti vanno separati con virgole, ma non ci sono vincoli in questo caso. Si noti che in questo "settore" del `for` avremmo anche potuto eseguire altre istruzioni, per esempio invocare metodi:

```
for (int i = 0, j = 10; i < 5 || j > 5; i++, j--,
  System.out.println("aggiornamento"))
{
  ...
}
```

Questo è il ciclo più utilizzato, vista la sua grande semplicità e versatilità. Inoltre, è l'unico ciclo che permette di dichiarare una variabile con visibilità interna al ciclo stesso. Il ciclo `while` è molto utilizzato quando non si sa quanto a lungo verranno eseguite le istruzioni e soprattutto nei cicli infiniti, dove la sintassi è banale:

```
while (true){
    . . .
}
```

Segue un ciclo `for` infinito:

```
for (;true;) {
    . . .
}
```

Che è equivalente a:

```
    for (;;) {
        . . .
    }
```

4.4.2 Il costrutto do

Nel caso in cui si desideri la certezza che le istruzioni in un ciclo vengano eseguite almeno nella prima iterazione, è possibile utilizzare il ciclo `do`.
Di seguito la sintassi:

```
[inizializzazione;]
do {
    corpo;
    [aggiornamento iterazione;]
} while (espr. booleana);
```

in questo caso viene eseguito prima il blocco di codice e poi viene valutata l'espressione booleana (condizione di terminazione) che si trova a destra della parola chiave `while`. Ovviamente se l'espressione booleana è verificata, viene rieseguito il blocco di codice, altrimenti esso termina. Si noti il punto e virgola situato alla fine del costrutto. L'output del seguente mini-programma:

```
public class DoWhile {
  public static void main(String args[]) {
    int i = 10;
    do {
      System.out.println(i);
    } while(i < 10);
  }
}
```

è:

```
10
```

Quindi la prima iterazione, è stata comunque eseguita.

4.4.3 Cicli for migliorato

Se state utilizzando un JDK versione 1.5 o superiore sarà possibile anche utilizzare una quarta tipologia di ciclo: l' "**enhanced for loop**", che in italiano potremmo tradurre con "**ciclo for migliorato**". L'aggettivo che però più si addice a questo ciclo non è "migliorato", bensì "semplificato". L'enhanced for infatti non è più potente di un tradizionale ciclo `for` e può sostituirlo solo in alcuni casi. In altri linguaggi il ciclo for migliorato viene chiamato "**foreach**" (in italiano "per ogni") e, per tale ragione, anche in Java si tende ad utilizzare questo nome, sicuramente più agevole da pronunciare e probabilmente più appropriato. La parola chiave `foreach`, però, non esiste in Java. Viene invece riutilizzata la parola chiave `for`. La sintassi è molto semplice e compatta:

```
for (variabile_temporanea : oggetto_iterabile) {
    corpo;
}
```

dove `oggetto_iterabile` è l'array (o un qualsiasi altro oggetto su cui è possibile iterare, come una `Collection`; cfr. Modulo 12) sui cui elementi si vogliono eseguire le iterazioni. Invece `variabile_temporanea` dichiara una variabile a cui, durante l'esecuzione del ciclo, verrà assegnato il valore dell' i-esimo elemento dell'`oggetto_iterabile` all'i-esima iterazione. In pratica, `variabile_temporanea` rappresenta all'interno del blocco di codice del nuovo ciclo for un elemento dell'`oggetto_iterabile`. Quindi, presumibilmente, il `corpo` del costrutto utilizzerà tale variabile. Notare che non esiste un'espressione booleana per la quale il ciclo deve terminare. Questo è già indicativo del fatto che tale ciclo viene usato soprattutto quando si vuole iterare su tutti gli elementi dell'oggetto iterabile. Facciamo un esempio:

```
int [] arr = {1,2,3,4,5,6,7,8,9};
for (int tmp : arr) {
    System.out.println(tmp);
}
```

Il precedente frammento di codice stampava a video tutti gli elementi dell'array.
Il ciclo foreach ha diversi limiti rispetto al ciclo `for` tradizionale. Per esempio non è possibile eseguire cicli all'indietro, ne è possibile farlo su più oggetti contemporaneamente e non è possibile accedere all'indice dell'array dell'elemento corrente. In realtà è sempre possibile dichiarare un contatore all'esterno del ciclo e

incrementarlo all'interno, ma a quel punto è forse meglio utilizzare un semplice ciclo `while`.
Per ulteriori dettagli sul ciclo foreach, si rimanda il lettore all'Unità Didattica 17.1.

4.4.4 Il costrutto `switch`

Il costrutto `switch`, si presenta come alternativa al costrutto `if`. A differenza di `if`, non è possibile utilizzarlo in ogni situazione in cui c'è bisogno di scegliere tra l'esecuzione di parti di codice diverse. Di seguito presentiamo la sintassi:

```
switch (variabile di test) {
    case valore_1:
        istruzione_1;
    break;
    case valore_2: {
        istruzione_2;
        ..........;
        istruzione_k;
    }
    break;
    case valore_3:
    case valore_4: {  //blocchi di codice opzionale
        istruzione_k+1;
        ..........;
        istruzione_j;
    }
    break;
    [default: {    //clausola default opzionale
        istruzione_j+1;
        ..........;
        istruzione_n;
    }]
}
```

In pratica, a seconda del valore intero che assume la variabile di test, vengono eseguite determinate espressioni. La variabile di test deve essere di un tipo di dato compatibile con un intero, ovvero un `byte`, uno `short`, un `char`, oppure, ovviamente, un `int` (ma non un `long`).

In realtà come variabile di test può essere utilizzata anche un'enumerazione o una classe wrapper come `Integer`, `Short`, `Byte` o `Character`. Siccome questi argomenti sono trattati più avanti nel testo, non aggiungeremo altro. Quest'ultima osservazione è valida solo a partire dalla versione 5 di Java in poi.

Inoltre `valore_1...valore_n` devono essere espressioni costanti e diverse tra loro. Si noti che la parola chiave `break` provoca l'immediata uscita dal costrutto. Se, infatti, dopo aver eseguito tutte le istruzioni che seguono un'istruzione di tipo `case`, non è presente un'istruzione `break`, verranno eseguiti tutti gli statement (istruzioni) che seguono gli altri `case`, sino a quando non si arriverà ad un `break`. Di seguito viene presentato un esempio:

```
public class SwitchStagione {
    public static void main(String args[]) {
    int mese = 4;
    String stagione;
    switch (mese) {
        case 12:
        case 1:
        case 2:
            stagione = "inverno";
        break;
        case 3:
        case 4:
        case 5:
            stagione = "primavera";
        break; //senza questo break si ha estate
        case 6:
        case 7:
        case 8:
            stagione = "estate";
        break;
        case 9:
        case 10:
        case 11:
        stagione = "autunno";
        break;
```

```
        default: //la clausola default è opzionale
            stagione = "non identificabile";
        }
        System.out.println("La stagione e' " + stagione);
    }
}
```

Se state utilizzando come editor EJE, potete sfruttare scorciatoie per creare i cinque principali costrutti di programmazione. Potete infatti sfruttare il menu "Inserisci" (o "Insert" se avete scelto la lingua inglese), o le eventuali scorciatoie con la tastiera (CTRL-2, CTRL-3, CTRL-4, CTRL-5, CTRL-6). In particolare è anche possibile selezionare una parte di codice per poi circondarla con un costrutto.

4.4.5 Due importanti parole chiave: `break` e `continue`

La parola chiave `break` è stata appena presentata come comando capace di fare terminare il costrutto `switch`. Ma `break` è utilizzabile anche per far terminare un qualsiasi ciclo. Il seguente frammento di codice provoca la stampa dei primi dieci numeri interi:

```
int i = 0;
while (true) //ciclo infinito
{
    if (i > 10)
        break;
    System.out.println(i);
    i++;
}
```

Oltre a `break`, esiste la parola chiave `continue`, che fa terminare non l'intero ciclo, ma solo l'iterazione corrente.
Il seguente frammento di codice provoca la stampa dei primi dieci numeri, escluso il cinque:

```
int i = 0;
do
```

```
{
    i++;
    if (i == 5)
        continue;
    System.out.println(i);
}
while(i <= 10);
```

Sia `break` sia `continue` possono utilizzare etichette (label) per specificare, solo nel caso di cicli annidati, su quale ciclo devono essere applicati. Il seguente frammento di codice stampa, una sola volta, i soliti primi dieci numeri interi:

```
int j = 1;
pippo: //possiamo dare un qualsiasi nome ad una label
while (true)
{
    while (true)
    {
        if (j > 10)
            break pippo;
        System.out.println(j);
        j++;
    }
}
```

Una label ha quindi la seguente sintassi:

```
nomeLabel:
```

Una label può essere posizionata solo prima di un ciclo, non dove si vuole. Ricordiamo al lettore che in Java non esiste il comando `goto`, anzi dobbiamo dimenticarlo…

Riepilogo

Questo modulo è stato dedicato alla sintassi Java che abbiamo a disposizione per influenzare il controllo di un programma. Abbiamo descritto (quasi) tutti gli operatori

supportati da Java, anche quelli meno utilizzati, e i particolari inerenti ad essi. Inoltre abbiamo introdotto tutti i costrutti che governano il flusso di esecuzione di un'applicazione dividendoli in costrutti semplici ed avanzati. In particolare, abbiamo sottolineato l'importanza della condizione `if` e del ciclo `for`, sicuramente i più utilizzati tra i costrutti. In particolare il ciclo `for` ha una sintassi compatta ed elegante, e soprattutto permette la dichiarazione di variabili con visibilità limitata al ciclo stesso.

Conclusioni Parte I

In questi primi quattro moduli sono stati affrontati argomenti riguardanti il linguaggio Java. Essi, per quanto non familiari possano risultare al lettore (pensare ai componenti della programmazione come classi e oggetti), rappresentano il nucleo del linguaggio. Il problema è che sino ad ora abbiamo imparato a conoscere questi argomenti, ma non ad utilizzarli nella maniera corretta. Ovvero, per quanto il lettore abbia diligentemente studiato, non è probabilmente ancora in grado di sviluppare un programma in maniera "corretta". Negli esercizi infatti non è mai stata richiesta l'implementazione di un'applicazione seppur semplice "da zero". Per esempio, vi sentite in grado di creare un'applicazione che simuli una rubrica telefonica? Quali classi creereste? Quante classi creereste? Solo iniziare sembra ancora un'impresa, figuriamoci portare a termine l'applicazione.
Dal prossimo capitolo, che inizia una nuova parte del testo, verranno introdotti argomenti riguardanti l'object orientation, molto più teorici. Questi concetti amplieranno notevolmente gli orizzonti del linguaggio, facendoci toccare con mano vantaggi insperati. Siamo sicuri che il lettore apprenderà agevolmente i concetti che in seguito saranno presentati, ma ciò non basta. Per saper creare un'applicazione "da zero" bisognerà acquisire esperienza sul campo e un proprio metodo di approccio al problema. È una sfida che se vinta darà i suoi frutti e che quindi conviene accettare…

Esercizi modulo 4

Esercizio 4.a)

Scrivere un semplice programma, costituito da un'unica classe, che sfruttando esclusivamente un ciclo infinito, l'operatore modulo, due costrutti `if`, un `break` ed un `continue`, stampi solo i primi cinque numeri pari.

Esercizio 4.b)

Scrivere un'applicazione che stampi i 26 caratteri dell'alfabeto (inglese-americano) con un ciclo.

Esercizio 4.c)

Scrivere una semplice classe che stampi a video la tavola pitagorica.

> **Suggerimento 1: non sono necessari array.**
> **Suggerimento 2: il metodo `System.out.println()` stampa l'argomento che gli viene passato e poi sposta il cursore alla riga successiva; infatti `println` sta per "print line". Esiste anche il metodo `System.out.print()`, che invece stampa solamente il parametro passatogli.**
> **Suggerimento 3: sfruttare un doppio ciclo innestato**

Esercizio 4.d)
Operatori e flusso di esecuzione, Vero o Falso:

1. Gli operatori unari di pre-incremento e post-incremento applicati ad una variabile danno lo stesso risultato, ovvero se abbiamo:
   ```
   int i = 5;
   ```
   sia
   ```
   i++;
   ```
   sia
   ```
   ++i;
   ```
   aggiornano il valore di `i` a `6`;
2. `d += 1` è equivalente a `d++` dove `d` è una variabile `double`.

3. Se abbiamo:
```
int i = 5;
int j = ++i;
int k = j++;
int h = k--;
boolean flag = ((i != j) && ( (j <= k ) ||  (i <= h))
);
```
`flag` avrà valore `false`.
4. L'istruzione:
```
System.out.println(1 + 2 + "3");
```
stamperà `33`.
5. Il costrutto `switch` può in ogni caso sostituire il costrutto `if`.
6. L'operatore ternario può in ogni caso sostituire il costrutto `if`.
7. Il costrutto `for` può in ogni caso sostituire il costrutto `while`.
8. Il costrutto `do` può in ogni caso sostituire il costrutto `while`.
9. Il costrutto `switch` può in ogni caso sostituire il costrutto `while`.
10. I comandi `break` e `continue` possono essere utilizzati nei costrutti `switch`, `for`, `while` e `do` ma non nel costrutto `if`.

Soluzioni esercizi modulo 4

Esercizio 4.a)

```
public class TestPari {
  public static void main(String args[]){
    int i = 0;
    while (true)
    {
      i++;
      if (i > 10)
        break;
      if ((i % 2) != 0)
        continue;
      System.out.println(i);
    }
  }
}
```

Esercizio 4.b)

```
public class TestArray {
  public static void main(String args[]){
    for (int i = 0; i < 26; ++i){
      char c = (char)('a' + i);
      System.out.println(c);
    }
  }
}
```

Esercizio 4.c)

```
public class Tabelline {
  public static void main(String args[]) {
    for (int i = 1; i <= 10; ++i){
      for (int j = 1; j <= 10; ++j){
        System.out.print(i*j + "\t");
      }
```

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

Esercizio 4.d)
Operatori e flusso di esecuzione, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso** la variabile booleana `flag` avrà valore `true`. Le espressioni "atomiche" valgono rispettivamente `true-false-true`, sussistendo le seguenti uguaglianze : `i = 6, j = 7, k = 5, h = 6`. Infatti `(i != j)` vale `true` e inoltre `(i <= h)` vale `true`. L' espressione `( (j <= k ) || (i <= h)) )` vale `true`, sussistendo l'operatore OR. Infine l'operatore AND fa sì che la variabile `flag` valga `true`.
4. **Vero.**
5. **Falso** `switch` può testare solo una variabile intera (o compatibile) confrontandone l'uguaglianza con costanti (in realtà dalla versione 5 si possono utilizzare come variabili di test anche le enumerazioni e il tipo `Integer`…). Il costrutto `if` permette di svolgere controlli incrociati sfruttando differenze, operatori booleani etc…
6. **Falso** l'operatore ternario è sempre vincolato ad un'assegnazione del risultato ad una variabile. Questo significa che produce sempre un valore da assegnare e da utilizzare in qualche modo (per esempio passando un argomento invocando un metodo). Per esempio, se `i` e `j` sono due interi, la seguente espressione:
   `i < j ? i : j;`
   provocherebbe un errore in compilazione (oltre a non avere senso).
7. **Vero.**
8. **Falso** il `do` in qualsiasi caso garantisce l'esecuzione della prima iterazione sul codice. Il `while` potrebbe prescindere da questa soluzione.
9. **Falso** lo `switch` è una condizione non un ciclo.
10. **Falso** il `continue` non si può utilizzare nello `switch` ma solo nei cicli.

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Conoscere e saper utilizzare i vari operatori (unità 4.1)	□	
Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2, 4.3)	□	
Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.2, 4.4)	□	

Note:

Parte II

“Object Orientation”

La parte II è interamente dedicata al supporto che Java offre ai paradigmi della programmazione ad oggetti. Questa è forse la parte più impegnativa ed originale di questo testo. I paradigmi dell’object orientation vengono presentati in maniera tale che il lettore imparerà ad apprezzarne l’utilità in pratica. In più si è cercato di compiere un’operazione di schematizzazione degli argomenti abbastanza impegnativa, in particolare del polimorfismo. In questo modo speriamo di facilitare l’apprendimento. Consigliamo lo studio di questa sezione a chiunque, anche a chi programma in Java da tempo.
Al termine di questa parte il lettore dovrebbe aver appreso, almeno a livello teorico, le nozioni fondamentali dell’Object Orientation applicabili a Java.

5

Complessità: media

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere le ragioni della nascita della programmazione ad oggetti (unità 5.1).
2. Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 5.2).
3. Saper definire ed utilizzare il concetto di astrazione (unità 5.2).
4. Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 5.3, 5.4).
5. Comprendere l'utilizzo e l'utilità del reference this (unità 5.4).
6. Comprendere l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione)(unità 5.5, 5.6).
7. Conoscere la filosofia di Java per quanto riguardo la semplicità di apprendimento (unità 5.3, 5.5).
8. Conoscere le conseguenze dell'utilizzo contemporaneo di incapsulamento ed ereditarietà (unità 5.6).

5 Programmazione ad oggetti utilizzando Java: Incapsulamento ed Ereditarietà

Questo è il primo modulo che si occupa di approfondire il supporto offerto da Java all'Object Orientation. In particolare verranno introdotti i primi paradigmi di questa scienza, come l'incapsulamento, l'ereditarietà e l'astrazione. Cercheremo di introdurre il lettore all'argomento, partendo con l'esposizione delle ragioni storiche della nascita dell'Object Orientation.

5.1 Breve storia della programmazione ad oggetti

Scrivere un programma significa in qualche modo simulare su un computer concetti e modelli fisici e matematici. Nei suoi primi tempi, la programmazione era concepita come una serie di passi lineari. Invece di considerare lo scopo del programma nella sua interezza creandone un modello astratto, si cercava di arrivare alla soluzione del

problema superando passaggi intermedi. Questo modello di programmazione orientato ai processi, con il passare del tempo, e con il conseguente aumento delle dimensioni dei programmi, ha apertamente manifestato i suoi difetti. Infatti, aumentando il numero delle variabili e delle interazioni da gestire tra esse, un programmatore in difficoltà aveva a disposizione strumenti come le variabili globali, ed il comando `goto`. In questo modo, agli inizi degli anni Settanta, per la programmazione procedurale fu coniato il termine dispregiativo "spaghetti code", dal momento che i programmi, crescendo in dimensioni, davano sempre più l'idea di assomigliare ad una massa di pasta aggrovigliata.
La programmazione orientata agli oggetti nacque storicamente sin dagli anni Sessanta con il linguaggio Simula-67. In realtà non si trattava di un linguaggio orientato agli oggetti "puro", ma con esso furono introdotti fondamentali concetti della programmazione quali le classi e l'ereditarietà. Fu sviluppato nel 1967 da Kristen Nygaard dell'università di Oslo e Ole Johan Dahl del Centro di Calcolo Norvegese e, a dispetto dell'importanza storica, non si può parlare di un vero e proprio successo presso il grande pubblico.
Nei primi anni '70 nacque il linguaggio SmallTalk, sviluppato inizialmente da Alan Kay all'Università dello Utah e successivamente da Adele Goldberg e Daniel Ingalls dello Xerox Park, centro di ricerca di Palo Alto in California. SmallTalk si può considerare un linguaggio ad oggetti "puro"; introdusse l'incapsulamento e la release SmallTalk-80 ebbe anche un discreto successo negli Stati Uniti. A lungo andare però, sebbene considerato da molti come ideale ambiente di programmazione, rimase confinato (come Simula) nei centri di ricerca universitari di tutto il mondo, considerato come ambiente di studio più che di sviluppo.
L'introduzione nel mondo della programmazione dei concetti di classe e di oggetto, che di fatto rendono i programmi più facilmente gestibili, non provocò quindi immediatamente una rivoluzione nell'informatica. Ciò fu dovuto al fatto che agli inizi degli anni Settanta ottenevano i maggiori successi linguaggi come il C. Un esempio su tutti: il sistema operativo Unix, tutt'oggi ancora utilizzatissimo, nacque proprio in quegli anni, ed il suo kernel (nucleo) era scritto in C.
Negli anni '80 però ci si rese conto della limitatezza della programmazione strutturata, il che fu essenzialmente dovuto ad una progressiva evoluzione dell'ingegneria del software, che iniziava a realizzare i programmi con una filosofia incrementale. Linguaggi come il C, come abbiamo già detto, offrono strumenti per apportare modifiche al software come le variabili globali e il comando `goto`, che si possono considerare ad alto rischio. Ecco che allora fu provvidenzialmente introdotta l'estensione del linguaggio C, realizzata da Bjarne Stroustrup, nota con il nome di C++. Questo nuovo linguaggio ha effettivamente rivoluzionato il mondo della programmazione. Fu scelto come linguaggio standard tra tanti linguaggi object oriented dalle grandi major

(Microsoft, Borland etc…), le quali iniziarono a produrre a loro volta tool di sviluppo che "estendevano" la programmazione C++.
Essendo un'estensione del C, un qualsiasi programma scritto in C deve poter essere compilato da un compilatore C++. Ciò, anche se ha favorito la migrazione in massa dei programmatori C verso il C++, si è rivelato anche il limite essenziale di quest'ultimo. Infatti si possono scrivere programmi che fanno uso sia della filosofia ad oggetti sia di quella procedurale, abbassando così le possibilità di buon funzionamento dei programmi stessi. Da qui l'idea di realizzare un nuovo linguaggio che doveva essere "veramente" orientato agli oggetti. Javapropone uno stile di programmazione che quasi "obbliga" a programmare correttamente ad oggetti. Inoltre, rispetto al C++, sono stati eliminati tutti gli strumenti "ambigui" e "pericolosi", come ad esempio goto, l'aritmetica dei puntatori e, di fatto, per utilizzare un qualcosa che assomigli ad una variabile globale, ci si deve proprio impegnare!
Possiamo concludere che, se il C++ ha il merito di aver fatto conoscere al grande pubblico la programmazione ad oggetti, Java ha il merito di averla fatta capire!
La programmazione orientata agli oggetti è una scienza, o meglio una filosofia adattabile alla programmazione. Essa si basa su concetti esistenti nel mondo reale, con i quali abbiamo a che fare ogni giorno. È già stato fatto notare al lettore che gli esseri umani posseggono da sempre i concetti di classe e di oggetto. L'astrazione degli oggetti reali in classi fa superare la complessità della realtà. In questo modo possiamo osservare oggetti completamente differenti, riconoscendo in loro caratteristiche e funzionalità che li accomunano, e quindi associarli ad una stessa classe. Per esempio, sebbene completamente diversi, un sassofono ed un pianoforte appartengono entrambi alla classe degli strumenti musicali. La programmazione ad oggetti inoltre, utilizzando il concetto di incapsulamento, rende i programmi composti da classi che nascondono i dettagli di implementazione dietro ad interfacce pubbliche, le quali permettono la comunicazione tra gli oggetti stessi che fanno parte del sistema. È favorito il riuso di codice già scritto, anche grazie a concetti quali l'ereditarietà ed il polimorfismo, che saranno presto presentati al lettore.

5.2 I paradigmi della programmazione ad oggetti

Ciò che caratterizza un linguaggio orientato agli oggetti è il supporto che esso offre ai cosiddetti "paradigmi della programmazione ad oggetti":

1. Incapsulamento
2. Ereditarietà

3. Polimorfismo

A differenza di altri linguaggi di programmazione orientati agli oggetti, Java definisce in modo estremamente chiaro i concetti appena accennati. Anche programmatori che si ritengono esperti di altri linguaggi orientati agli oggetti come il C++, studiando Java potrebbero scoprire significati profondi in alcuni concetti che prima si ritenevano chiari. Nel presente e nel prossimo modulo introdurremo i tre paradigmi in questione. Segnaliamo al lettore che non tutti i testi parlano di tre paradigmi. In effetti si dovrebbero considerare paradigmi della programmazione ad oggetti anche l'astrazione ed il riuso (ed altri…). Questi due sono spesso considerati secondari rispetto agli altri, non per minor potenza ed utilità, ma per il supporto alla definizione di linguaggio orientato agli oggetti. Infatti l'astrazione ed il riuso sono concetti che appartengono anche alla programmazione procedurale.

In realtà i paradigmi fondamentali dell'object orientation sono tanti. Ma non sembra questa la sede ideale dove approfondire il discorso…

5.2.1 Astrazione e riuso

L'**astrazione** potrebbe definirsi come "l'arte di sapersi concentrare solo sui dettagli veramente essenziali nella descrizione di un'entità". In pratica l'astrazione è un concetto chiarissimo a tutti noi, dal momento che lo utilizziamo in ogni istante della nostra vita. Per esempio, mentre state leggendo questo manuale, vi state concentrando sull'apprenderne correttamente i contenuti, senza badare troppo alla forma, ai colori, allo stile e tutti particolari fisici e teorici che compongono la pagina che state visualizzando (o almeno lo speriamo!).
Per formalizzare un discorso altrimenti troppo "astratto", potremmo parlare di almeno tre livelli di astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

1. Astrazione funzionale
2. Astrazione dei dati
3. Astrazione del sistema

Adoperiamo l'astrazione funzionale ogni volta che implementiamo un metodo. Infatti, tramite un metodo, riusciamo a portare all'interno di un'applicazione un concetto dinamico, sinonimo di azione, funzione. Per scrivere un metodo ci dovremmo limitare alla sua implementazione più robusta e chiara possibile. In questo modo avremo la

possibilità di invocare quel metodo ottenendo il risultato voluto, senza dover tener presente l'implementazione del metodo stesso.

Lo stesso concetto era valido anche nella programmazione procedurale, grazie alle funzioni.

Adoperiamo l'astrazione dei dati ogni volta che definiamo una classe, raccogliendo in essa solo le caratteristiche e le funzionalità essenziali degli oggetti che essa deve definire nel contesto in cui ci si trova.

Potremmo dire che l'astrazione dei dati "contiene" l'astrazione funzionale.

Adoperiamo l'astrazione del sistema ogni volta che definiamo un'applicazione nei termini delle classi essenziali che devono soddisfare agli scopi dell'applicazione stessa. Questo assomiglia molto da vicino a quello che nella programmazione procedurale era chiamato metodo "Top down".

Potremmo affermare che l'astrazione del sistema "contiene" l'astrazione dei dati, e, per la proprietà transitiva, l'astrazione funzionale.

Il **riuso** è invece da considerarsi una conseguenza dell'astrazione e degli altri paradigmi della programmazione ad oggetti (incapsulamento, ereditarietà e polimorfismo).

Il riuso era un paradigma valido anche per la programmazione procedurale. In quel caso era una conseguenza dell'astrazione funzionale e del metodo "Top down" nella programmazione procedurale.

5.3 *Incapsulamento*

L'incapsulamento è la chiave della programmazione orientata agli oggetti. Tramite esso, una classe riesce ad acquisire caratteristiche di robustezza, indipendenza e riusabilità. Inoltre la sua manutenzione risulterà più semplice al programmatore.
Una qualsiasi classe è essenzialmente costituita da dati e metodi. La filosofia dell'incapsulamento è semplice. Essa si basa sull'accesso controllato ai dati mediante

metodi che possono prevenirne l'usura e la non correttezza. A livello di implementazione, ciò si traduce semplicemente nel dichiarare privati gli attributi di una classe e quindi inaccessibili fuori dalla classe stessa. A tale scopo introdurremo un nuovo modificatore: `private`.
L'accesso ai dati potrà essere fornito da un'interfaccia pubblica costituita da metodi dichiarati public e quindi accessibili da altre classi. In questo modo, tali metodi potrebbero ad esempio permettere di realizzare controlli prima di confermare l'accesso ai dati privati.
Se l'incapsulamento è gestito in maniera intelligente, le nostre classi potranno essere utilizzate nel modo migliore e più a lungo, giacché le modifiche e le revisioni potranno riguardare solamente parti di codice non visibili all'esterno.
Se volessimo fare un esempio basandoci sulla realtà che ci circonda, potremmo prendere in considerazione un telefono. La maggior parte degli utenti, infatti, sa utilizzare il telefono, ma ne ignora il funzionamento interno. Chiunque può alzare la cornetta, comporre un numero telefonico e conversare con un'altra persona, ma pochi conoscono in dettaglio la sequenza dei processi scatenati da queste poche, semplici azioni. Evidentemente, per utilizzare il telefono, non è necessario prendere una laurea in telecomunicazioni: basta conoscere la sua interfaccia pubblica (costituita dalla cornetta e dai tasti), non la sua implementazione interna.
Di seguito è presentato un esempio che dovrebbe chiarire al lettore l'argomento. Supponiamo di voler scrivere un'applicazione che utilizza la seguente classe, la quale astrae in maniera semplice il concetto di data:

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

Come può utilizzare la nostra applicazione tale astrazione? Ogni volta che serve un oggetto `Data`, il codice da scrivere sarà simile al seguente:
(Codice 5.1)

```
...
Data unaData = new Data();
unaData.giorno = 14;
unaData.mese = 4;
unaData.anno = 2004;...
```

Dove sta il problema? Non è raro che i valori delle variabili dell'oggetto debbano essere impostati al runtime in maniera dinamica, probabilmente dall'utente. Supponiamo che la nostra applicazione permetta all'utente di inserire la sua data di nascita, magari mediante un'interfaccia grafica. In tal caso il codice da scrivere sarà simile al seguente:
(Codice 5.2)

```
...
Data unaData = new Data();
unaData.giorno = interfaccia.dammiGiornoInserito();
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();...
```

dove i metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito()` dell'oggetto `interfaccia` restituiscono un intero inserito dall'utente dell'applicazione. Supponiamo che l'utente abbia inserito rispettivamente i valori 32 per il giorno, 13 per il mese e 1800 per l'anno; ecco che i problemi del codice iniziano a risultare evidenti. Come è possibile evitare definitivamente problemi come questo per la classe `Data`? Elenchiamo alcune possibili soluzioni:

1. Si potrebbe limitare la possibilità degli inserimenti sull'interfaccia grafica all'utente. Il problema sarebbe risolto, ma solo nel caso che l'impostazione della data avvenga sempre e comunque tramite l'interfaccia grafica. Inoltre il problema sarà risolto solo in quest'applicazione e quindi non in maniera definitiva. Ma se volessimo riutilizzare (il riuso dovrebbe essere un paradigma fondamentale della programmazione ad oggetti) in un'altra applicazione la classe `Data`, senza riutilizzare la stessa interfaccia grafica, saremmo costretti a scrivere nuovamente del codice che gestisce il problema.
2. Potremmo delegare al codice dei metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito(` dell'oggetto `interfaccia` i controlli necessari alla giusta impostazione della data. Ma anche in questo caso rimarrebbero tutti i problemi esposti per la soluzione 1).
3. Utilizzare l'incapsulamento, modificando la classe `Data` nel modo seguente:

(Codice 5.3)

```
public class Data {
    private int giorno;
```

```
    private int mese;
    private int anno;

    public void setGiorno(int g) {
       if (g > 0 && g <= 31) {
            giorno = g;
        }
        else {
            System.out.println("Giorno non valido");
        }
    }

    public int getGiorno() {
        return giorno;
    }

    public void setMese(int m) {
        if (m > 0 && m <= 12) {
            mese = m;
        }
        else {
            System.out.println("Mese non valido");
        }
    }

    public int getMese() {
        return mese;
    }

    public void setAnno(int a) {
        anno = a;
    }

    public int getAnno() {
        return anno;
    }
}
```

Implementare l'incapsulamento con codice Java consiste il più delle volte nel dichiarare tutti dati privati e fornire alla classe metodi pubblici di tipo "set" e "get" per accedervi rispettivamente in scrittura e lettura.

Questi metodi solitamente (ma non obbligatoriamente) seguono una convenzione che è utilizzata anche nella libreria standard. Se abbiamo una variabile privata dovremmo chiamare questi metodi con la sintassi `setNomeVariabile()` e `getNomeVariabile()`.
Quindi, anche se all'inizio potrà sembrare noioso (la seconda versione della classe `Data` è nettamente più estesa della prima), implementare l'incapsulamento non richiede grossa inventiva da parte dello sviluppatore.

Cerchiamo ora di chiarire quali sono i vantaggi. Nel momento in cui abbiamo dichiarato i dati privati, per la definizione del modificatore `private` essi non saranno più accessibili mediante l'operatore dot, a meno che il codice che vuole accedere al dato privato non si trovi nella classe che lo ha dichiarato. Questo implica che il codice 5.1 e il codice 5.2 produrrebbero un errore in compilazione, perché tenterebbero di assegnare valori a variabili non visibili in quel contesto (classi diverse dalla classe `Data`). I codici 5.1 e 5.2 devono essere rispettivamente sostituiti con i seguenti:

(Codice 5.1.bis)
```
...
Data unaData = new Data();
unaData.setGiorno(14);
unaData.setMese(4);
unaData.setAnno(2004);...
```

(Codice 5.2.bis)
```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Ovvero, implementando l'incapsulamento, per sfruttare i dati dell'oggetto `Data`, saremo costretti ad utilizzare l'interfaccia pubblica dell'oggetto, costituita dai metodi pubblici

"set e get", così come quando vogliamo utilizzare un telefono siamo costretti ad utilizzare l'interfaccia pubblica costituita dai tasti e dalla cornetta. Infatti i metodi "set e get" hanno implementazioni che si trovano internamente alla classe `Data` e quindi possono accedere ai dati privati. Inoltre, nel codice 5.3, si può notare che, per esempio, il metodo `setGiorno()` imposta la variabile `giorno` con il parametro che gli viene passato se risulta compresa tra `1` e `31`, altrimenti stampa un messaggio di errore. Quindi, a priori, ogni oggetto `Data` funziona correttamente! Questo implica maggiori opportunità di riuso e robustezza del codice.
Altro immenso vantaggio: il codice è molto più facile da manutenere e si adatta ai cambiamenti. Per esempio, il lettore avrà sicuramente notato che il codice 5.3 risolve relativamente i problemi della classe `Data`. Infatti permetterebbe l'impostazione del giorno al valore 31, anche se la variabile mese vale 2 (Febbraio che ha 28 giorni, ma 29 negli anni bisestili…). Bene, possiamo far evolvere la classe `Data`, introducendo tutte le migliorie che vogliamo all'interno del codice 5.3, ma non dovremmo cambiare una riga per i codici 5.1 bis e 5.2 bis! Per esempio, se il metodo `setGiorno()`, viene cambiato nel seguente modo:

```
public void setGiorno(int g) {
    if (g > 0 && g <= 31 && mese != 2) {
        giorno = g;
    }
    else {
        System.out.println("Giorno non valido");
    }
}
```

bisognerà ricompilare solo la classe `Data`, ma i codici 5.1.bis e 5.2.bis rimarranno inalterati! Dovrebbe risultare ora chiaro al lettore che la programmazione ad oggetti si adatta meglio alla filosofia di modifiche iterative ed incrementali che è applicata al software moderno.

Ovviamente potremmo continuare a cambiare il codice di questo metodo fino a quando non sarà perfetto. La nostra superficialità è dovuta al fatto che nella libreria standard è già stata implementata una classe `Date` (nel package `java.util`), che mette a disposizione anche ore, minuti, secondi, millisecondi, giorni della settimana, ora legale etc…

Al lettore dovrebbe ora risultare chiara l'utilità dei metodi "set", che da ora in poi chiameremo "mutator methods". Potrebbe però avere ancora qualche riserva sui metodi "get", che da adesso chiameremo "accessor methods".
Con un paio di esempi potremmo fugare ogni dubbio.
Supponiamo di volere verificare dal codice 5.2 bis l'effettivo successo dell'impostazione dei dati dell'oggetto `unaData`, stampandone i dati a video. Dal momento in cui:

```
System.out.println(unaData.giorno);
```

restituirà un errore in compilazione, e:

```
System.out.println(unaData.setGiorno());
```

non ha senso perché il tipo di ritorno del metodo `setGiorno()` è `void`, appare evidente che l'unica soluzione rimane:

```
System.out.println(unaData.getGiorno());
```

Inoltre anche un accessor method potrebbe eseguire controlli come un mutator method. Per esempio, nella seguente classe l'accessor method gestisce l'accesso ad un conto bancario personale, mediante l'inserimento di un codice segreto:

```
public class ContoBancario {
    private String contoBancario = "5000000 di Euro";
    private int codice = 1234;
    private int codiceInserito;

    public void setCodiceInserito(int cod){
        codiceInserito = cod;
    }

    public int getCodiceInserito(){
        return codiceInserito;
    }


    public String getContoBancario() {
```

```
            if (codiceInserito == codice) {
                  return contoBancario;
            }
            else {
                return "codice errato!!!";
            }
        }
        . . .
    }
```

5.3.1 Prima osservazione sull'incapsulamento

Sino ad ora abbiamo visto esempi di incapsulamento abbastanza classici, dove nascondevamo all'interno delle classi gli attributi mediante il modificatore `private`. Nulla ci vieta di utilizzare `private`, anche come modificatore di metodi, ottenendo così un "incapsulamento funzionale". Un metodo privato infatti, potrà essere invocato solo da un metodo definito nella stessa classe, che potrebbe a sua volta essere dichiarato pubblico.
Per esempio la classe `ContoBancario`, definita precedentemente, in un progetto potrebbe evolversi nel seguente modo:

```
public class ContoBancario {
    . . .
    public String getContoBancario(int codiceDaTestare)
    {
        return controllaCodice(codiceDaTestare);
    }

    private String controllaCodice(int codiceDaTestare) {
        if (codiceInserito == codice) {
            return contoBancario;
        }
        else  {
            return "codice errato!!!";
        }
    }
}
```

Ciò favorirebbe il riuso di codice in quanto, introducendo nuovi metodi (come probabilmente accadrà in un progetto che viene manutenuto), questi potrebbero risfruttare il metodo `controllaCodice()`.

5.3.2 Seconda osservazione sull'incapsulamento

Solitamente si pensa che un membro di una classe dichiarato `private` diventi "inaccessibile da altre classi". Questa frase è ragionevole per quanto riguarda l'ambito della compilazione, dove la dichiarazione delle classi è il problema da superare. Ma, se ci spostiamo nell'ambito della Java Virtual Machine dove, come abbiamo detto i protagonisti assoluti non sono le classi ma gli oggetti, dobbiamo rivalutare l'affermazione precedente. L'incapsulamento infatti permetterà a due oggetti istanziati dalla stessa classe di accedere in "modo pubblico" ai rispettivi membri privati. Consideriamo la seguente classe `Dipendente`:

```
public class Dipendente  {
    private String nome;
    private int anni; //intendiamo età in anni
    . . .
    public String getNome() {
        return nome;
    }
    public void setNome(String n) {
        nome = n;
    }
    public int getAnni() {
        return anni;
    }
    public void setAnni(int n) {
        anni = n;
    }
    public int getDifferenzaAnni(Dipendente altro) {
        return (anni – altro.anni);
   }
}
```

Nel metodo `getDifferenzaAnni()` notiamo che è possibile accedere direttamente alla variabile `anni` dell'oggetto `altro`, senza dover utilizzare il metodo `getAnni()`.

Il lettore è invitato a riflettere soprattutto sul fatto che il codice precedente è valido per la compilazione, ma, il seguente metodo:

```
public int getDifferenzaAnni(Dipendente altro){
    return (getAnni() - altro.getAnni());
}
```

favorirebbe sicuramente di più il riuso di codice, e quindi è da considerarsi preferibile. Infatti, il metodo `getAnni()` si potrebbe evolvere introducendo controlli, che conviene richiamare piuttosto che riscrivere.

5.3.3 Il reference `this`

L'esempio precedente potrebbe aver provocato nel lettore qualche dubbio. Sino ad ora avevamo dato per scontato che l'accedere ad una variabile d'istanza all'interno della classe dove è definita fosse un "processo naturale" che non aveva bisogno di reference. Ad esempio, all'interno del metodo `getGiorno()` nella classe `Data` accedevamo alla variabile `giorno`, senza referenziarla. Alla luce dell'ultimo esempio e considerando che potrebbero essere istanziati tanti oggetti dalla classe `Data`, ci potremmo chiedere: se `giorno` è una variabile d'istanza, a quale istanza appartiene? La risposta a questa domanda è: dipende "dall'oggetto corrente", ovvero dall'oggetto su cui è chiamato il metodo `getGiorno()`. Per esempio, in fase d'esecuzione di un certa applicazione potrebbero essere istanziati due particolari oggetti, che supponiamo si chiamino `mioCompleanno` e `tuoCompleanno`. Entrambi questi oggetti hanno una propria variabile `giorno`. Ad un certo punto, all'interno del programma potrebbe presentarsi la seguente istruzione:

```
System.out.println(mioCompleanno.getGiorno());
```

Sarà stampato a video il valore della variabile giorno dell'oggetto `mioCompleanno`, ma dal momento che sappiamo come una variabile anche all'interno di una classe potrebbe (e dovrebbe) essere referenziata, dovremmo sforzarci di capire come fa la Java Virtual Machine a scegliere la variabile giusta senza avere a disposizione reference!
In realtà si tratta di un'altra iniziativa del compilatore Java. Se il programmatore non referenzia una certa variabile d'istanza, al momento della compilazione il codice sarà modificato dal compilatore stesso, che aggiungerà un reference all'oggetto corrente davanti alla variabile. Ma quale reference all'oggetto corrente? La classe non può conoscere a priori i reference degli oggetti che saranno istanziati da essa in fase di runtime!

Java introduce una parola chiave che per definizione coincide ad un reference all'oggetto corrente: `this` (in italiano "questo"). Il reference `this` viene quindi implicitamente aggiunto nel bytecode compilato, per referenziare ogni variabile d'istanza non esplicitamente referenziata.
Ancora una volta Java cerca di facilitare la vita del programmatore. Infatti in un linguaggio orientato agli oggetti puro, non è permesso non referenziare le variabili d'istanza.
In pratica il metodo `getGiorno()` che avrà a disposizione la JVM dopo la compilazione sarà:

```
 public int getGiorno() {
   return this.giorno; //il this lo aggiunge il
compilatore
 }
```

In seguito vedremo altri utilizzi del reference "segreto" `this`.

Anche in questo caso, abbiamo notato un altro di quei comportamenti del linguaggio che definire Java "semplice". Se non ci siamo posti il problema del referenziare i membri di una classe sino a questo punto, vuol dire che anche questa volta "Java ci ha dato una mano".

5.3.4 Due stili di programmazione a confronto

Nel secondo modulo abbiamo distinto le variabili d'istanza dalle variabili locali. La diversità tra i due concetti è tale che il compilatore ci permette di dichiarare una variabile locale (o un parametro di un metodo) ed una variabile d'istanza, aventi lo stesso identificatore, nella stessa classe. Infatti la JVM alloca le variabili locali e le variabili d'istanza in differenti aree di memoria (dette rispettivamente Stack e Heap Memory).
La parola chiave `this` s'inserisce in questo discorso nel seguente modo. Abbiamo più volte avuto a che fare con passaggi di parametri in metodi, al fine di inizializzare variabili d'istanza. Siamo stati costretti, sino ad ora, ad inventare per il parametro passato un identificatore differente da quello della variabile d'istanza da inizializzare.
Consideriamo la seguente classe:

```
public class Cliente
{
    private String nome, indirizzo;
    private int numeroDiTelefono;
```

```
    . . .
    public void setCliente(String n, String ind, int num)
    {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
}
```

Notiamo l'utilizzo dell'identificatore `n` per inizializzare `nome`, `num` per `numeroDiTelefono` e `ind` per `indirizzo`. Non c'è nulla di sbagliato in questo. Conoscendo però l'esistenza di `this`, abbiamo la possibilità di scrivere equivalentemente:

```
public class Cliente
{
  . . .
  public void setCliente(String nome, String indirizzo,
  int numeroDiTelefono)
  {
    this.nome = nome;
    this.indirizzo = indirizzo;
    this.numeroDiTelefono = numeroDiTelefono;
  }
  . . .
}
```

Infatti, tramite la parola chiave `this`, specifichiamo che la variabile referenziata appartiene all'istanza. Di conseguenza la variabile non referenziata sarà il parametro del metodo. Non c'è ambiguità, quindi, nel codice precedente.
Questo stile di programmazione è da alcuni (compreso chi vi scrive) considerato preferibile. In questo modo, infatti, non c'è possibilità di confondere le variabili con nomi simili. Nel nostro esempio potrebbe capitare di assegnare il parametro `n` alla variabile d'istanza `numeroDiTelefono` ed il parametro `num` alla variabile `nome`. Potremmo affermare che l'utilizzo di `this` aggiunge chiarezza al nostro codice.
Il lettore noti che se scrivessimo:

```
public class Cliente
{
  . . .
  public void setCliente(String nome, String indirizzo,
  int numeroDiTelefono)
  {
      nome = nome;
      indirizzo = indirizzo;
      numeroDiTelefono = numeroDiTelefono;
  }
  . . .
}
```

il compilatore, non trovando riferimenti espliciti, considererebbe le variabili sempre locali e quindi non otterremmo il risultato desiderato.

5.4 Quando utilizzare l'incapsulamento

Se volessimo essere brevi, dovremmo dire che non ci sono casi in cui è opportuno o meno utilizzare l'incapsulamento. Una qualsiasi classe di una qualsiasi applicazione dovrebbe essere sviluppata utilizzando l'incapsulamento. Anche se all'inizio di un progetto può sembrarci che su determinate classi usufruire dell'incapsulamento sia superfluo, l'esperienza insegna che è preferibile l'applicazione in ogni situazione. Facciamo un esempio banale. Abbiamo già accennato al fatto, peraltro scontato, che per realizzare un'applicazione a qualsiasi livello (sempre che non sia veramente elementare), bisogna apportare a quest'ultima modifiche incrementali. Un lettore con un minimo d'esperienza di programmazione non potrà che confermare l'ultima affermazione. Supponiamo di voler scrivere una semplice applicazione che, assegnati due punti, disegni il segmento che li unisce. Supponiamo inoltre che si utilizzi la seguente classe non incapsulata `Punto`, già incontrata nel modulo 2:

```
public class Punto {
    public int x, y;
    . . .
}
```

l'applicazione, in un primo momento, istanzierà ed inizializzerà due punti con il seguente frammento di codice:

```
(Codice 5.4). . .
Punto p1 = new Punto();
Punto p2 = new Punto();
p1.x = 5;
p1.y = 6;
p2.x = 10;
p2.y = 20;. . .
```

Supponiamo che l'evolversi della nostra applicazione renda necessario che i due punti non debbano trovarsi fuori da una certa area piana ben delimitata. Ecco risultare evidente che la soluzione migliore sia implementare l'incapsulamento all'interno della classe `Punto` in questo modo:

```
public class Punto {
    private int x, y;
    private final int VALORE_MASSIMO_PER_X=10 ;
    private final int VALORE_MINIMO_PER_X=-10 ;
    private final int VALORE_MASSIMO_PER_Y=10 ;
    private final int VALORE_MINIMO_PER_Y=-10 ;
    public void setX(int a) {
      if (a <= VALORE_MASSIMO_PER_X &&  a >=
          VALORE_MINIMO_PER_X){
         x = a;
         System.out.println("X è OK!");
      }
      else {
          System.out.println("X non valida");
      }
    }
    public void setY(int a){
      if (a <= VALORE_MASSIMO_PER_Y && a >=
          VALORE_MINIMO_PER_Y){
        y = a;
        System.out.println("Y è OK!");
      }
      else {
        System.out.println("Y non valida");
      }
    }
```

```
    . . .
}
```

Purtroppo però, dopo aver apportato queste modifiche alla classe `Punto`, saremo costretti a modificare anche il frammento di codice 5.4 dell'applicazione nel modo seguente:

(Codice 5.5)

```
Punto p1 = new Punto();
Punto p2 = new Punto();
p1.setX(5);
p1.setY(6);
p2.setX(10);
p2.setY(20);
. . .
```

Saremmo partiti meglio con la classe `Punto` forzatamente incapsulata in questo modo:

```
public class Punto {
    private int x, y;
    public void setX(int a) {
        x = a;
    }
    public void setY(int a) {
        y = a;
    }
     . . .
}
```

dal momento che avremmo modificato solo il codice all'interno dei metodi d'accesso e saremmo stati costretti a stare al codice 5.5 all'interno dell'applicazione che utilizza `Punto`.

Il codice 5.5 potrebbe essere stato utilizzato in molte altre parti dell'applicazione…

Arriviamo alla conclusione che l'incapsulamento è "prassi ed obbligo" in Java. Un linguaggio orientato agli oggetti "puro" come lo SmallTalk, infatti, non permetterebbe la

dichiarazione di attributi pubblici. Java però vuole essere un linguaggio semplice da apprendere, ed in questo modo non costringe l'aspirante programmatore ad imparare prematuramente un concetto ad ogni modo complesso quale l'incapsulamento. In particolare, nei primi tempi, non se ne apprezzerebbe completamente l'utilizzo, dovendo comunque approcciare troppi concetti nuovi contemporaneamente. Tuttavia, non bisognerebbe mai permettere di sacrificare l'incapsulamento per risparmiare qualche secondo di programmazione. Le conseguenze sono ormai note al lettore.

EJE, permette di creare automaticamente i metodi mutator (set) ed accessor (get) a partire dalla definizione della variabile da incapsulare. Basta aprire il menu "inserisci" e cliccare su "Proprietà JavaBean" (oppure premere CTRL-9). Per Proprietà JavaBean intendiamo una variabile d'istanza incapsulata. Un semplice wizard chiederà di inserire prima il tipo della variabile e poi il nome. Una volta fornite queste due informazioni EJE adempirà al suo compito.
Il nome "Proprietà JavaBean", deriva dalla teoria dei JavaBeans, una tecnologia che nei primi anni di Java ebbe molto successo, attualmente "passata di moda". Il nome JavaBean però è ancora oggi sulla bocca di tutti, anche grazie al "riciclaggio" di tale termine nella tecnologia JSP. In inglese vuol dire "chicco di Java" (ricordiamo che Java è il nome di una tipologia di caffé.

5.5 Ereditarietà

Sebbene l'ereditarietà sia un argomento semplice da comprendere, non è sempre utilizzata in maniera corretta. E' è la caratteristica della programmazione ad oggetti che mette in relazione di estensibilità più classi che hanno caratteristiche comuni. Come risultato avremo la possibilità di ereditare codice già scritto (e magari già testato), e quindi gestire insiemi di classi collettivamente, giacché accomunate da caratteristiche comuni. Le regole per gestire correttamente l'ereditarietà sono semplici e chiare.

5.5.1 La parola chiave extends

Consideriamo le seguenti classi, che non incapsuliamo per semplicità:

```
public class Libro {
    public int numeroPagine;
    public int prezzo;
```

```
    public String autore;
    public String editore;
    . . .
}
public class LibroSuJava{
    public int numeroPagine;
    public int prezzo;
    public String autore;
    public String editore;
    public final String ARGOMENTO_TRATTATO = "Java";
    . . .
}
```

Notiamo che le classi `Libro` e `LibroSuJava` rappresentano due concetti in relazione tra loro e quindi dichiarano campi in comune. In effetti, l'ereditarietà permetterà di mettere in relazione di estensione le due classi con la seguente sintassi:

```
public class LibroSuJava extends Libro {
    public final String ARGOMENTO_TRATTATO = "Java";
    . . .
}
```

In questo modo la classe `LibroSuJava` erediterà tutti i campi pubblici della classe che estende. Quindi, anche se non sono state codificate esplicitamente, nella classe `LibroSuJava` sono presenti anche le variabili pubbliche `numeroPagine`, `prezzo`, `autore` ed `editore` definite nella classe `Libro`.
In particolare diremo che `LibroSuJava` è **sottoclasse** di `Libro`, e `Libro` è **superclasse** di `LibroSuJava`.

5.5.2 Ereditarietà multipla ed interfacce

In Java non esiste la cosiddetta "ereditarietà multipla" così come esiste in C++. Questa permette ad una classe di estendere più classi contemporaneamente. In pratica non è possibile scrivere:

```
public class Idrovolante extends Nave, Aereo {
    . . .
}
```

anche se a livello concettuale, l'estensione multipla esisterebbe. Per esempio, una certa persona potrebbe estendere programmatore e marito.
Ma l'ereditarietà multipla è spesso causa di problemi implementativi, cui il programmatore deve poi rimediare. In Java è quindi stata fatta ancora una volta una scelta a favore della robustezza, ma a discapito della potenza del linguaggio, ed ogni classe potrà estendere una sola classe alla volta. In compenso però, tramite il concetto di "interfaccia", Java offre supporto ad un meccanismo di simulazione dell'ereditarietà multipla. Questa simulazione non presenta ambiguità e sarà presentata nel modulo 9.

5.5.3 La classe Object

Come abbiamo già osservato più volte, il compilatore Java inserisce spesso nel bytecode compilato alcune istruzioni che il programmatore non ha inserito, sempre al fine di agevolare lo sviluppatore sia nell'apprendimento che nella codifica.
Abbiamo inoltre già asserito che la programmazione ad oggetti si ispira a concetti reali. Tutta la libreria standard di Java è stata pensata ed organizzata in maniera tale da soddisfare la teoria degli oggetti. Siccome la realtà è composta da oggetti (tutto può considerarsi un oggetto, sia elementi concretamente esistenti, sia concetti astratti), nella libreria standard di Java esiste una classe chiamata `Object`, che astrae il concetto di "oggetto generico". Esso appartiene al package `java.lang` ed è di fatto la superclasse di ogni classe. È in cima della gerarchia delle classi e quindi tutte le classi ereditano i membri di `Object` (il lettore può verificare questa affermazione dando uno sguardo alla documentazione). Se definiamo una classe che non estende altre classi, essa automaticamente estenderà `Object`. Ciò significa che se scrivessimo:

```
public class Arte {
    . . .
}
```

il compilatore in realtà "scriverebbe" questa classe nel seguente modo:

```
public class Arte extends Object {
    . . .
}
```

Nel mondo reale tutto è un oggetto, quindi in Java tutte le classi estenderanno `Object`.

5.6 Quando utilizzare l'ereditarietà

Quando si parla di ereditarietà si è spesso convinti che per implementarla basti avere un paio di classi che dichiarino campi in comune. In realtà ciò potrebbe essere interpretato come un primo passo verso un'eventuale implementazione di ereditarietà. Il test decisivo deve però essere effettuato mediante la cosiddetta "**is a**" relationship (la relazione "**è un**").

5.6.1 La relazione "is a"

Per un corretto uso dell'ereditarietà, il programmatore dovrà porsi una fondamentale domanda: un oggetto della candidata sottoclasse "è un" oggetto della candidata superclasse? Se la risposta alla domanda è negativa, l'ereditarietà non si deve utilizzare. Effettivamente, se l'applicazione dell'ereditarietà dipendesse solamente dai campi in comune tra due classi, potremmo trovare relazioni d'estensione tra classi quali `Triangolo` e `Rettangolo`. Per esempio:

```
public class Triangolo {
    public final int NUMERO_LATI = 3;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    . . . . . .
}

public class Rettangolo extends Triangolo {
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
    . . . . . .
}
```

ma ovviamente un rettangolo non è un triangolo e per la relazione "is a" questa estensione non è valida. La nostra esperienza ci dice che se iniziassimo un progetto software incrementale, senza utilizzare questo test, potremmo arrivare ad un punto dove la soluzione migliore per continuare è ricominciare da capo!

5.6.2 Generalizzazione e specializzazione

Sono due termini che definiscono i processi che portano all'implementazione dell'ereditarietà.
Si parla di generalizzazione se, a partire da un certo numero di classi, si definisce una superclasse che ne raccoglie le caratteristiche comuni.
Viceversa si parla di specializzazione quando, partendo da una classe, si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specializzati.
L'utilità della specializzazione è evidente: supponiamo di voler creare una classe `MioBottone`, le cui istanze siano visualizzate come bottoni da utilizzare su un'interfaccia grafica. Partire da zero creando ogni pixel è molto complicato. Se invece estendiamo la classe `Button` del package `java.awt`, dobbiamo solo aggiungere il codice che personalizzerà il `MioBottone`.
Nell'ultimo esempio avevamo a disposizione la classe `Triangolo` e la classe `Rettangolo`. Abbiamo notato come il test "is a", fallendo, ci "sconsigli" l'implementazione dell'ereditarietà. Eppure queste due classi hanno campi in comune e non sembra che sia un evento casuale. In effetti sia il `Triangolo` sia il `Rettangolo` sono ("is a") entrambi poligoni.
La soluzione a questo problema è "naturale". Basta generalizzare le due astrazioni in una classe `Poligono`, che potrebbe essere estesa dalle classi `Triangolo` e `Rettangolo`. Per esempio:

```
public class Poligono {
    public int numeroLati;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    . . . . . .
}

public class Triangolo extends Poligono {
    public final int NUMERO_LATI = 3;
    . . . . . .
}

public class Rettangolo extends Poligono {
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
```

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

Se fossimo partiti dalla classe `Poligono` per poi definire le due sottoclassi, avremmo parlato invece di specializzazione.

E' fondamentale notare che un'astrazione scorretta dei dati potrebbe essere amplificata dall'implementazione dell'ereditarietà (e dall'incapsulamento).

5.6.3 Rapporto ereditarietà-incapsulamento

Dal momento che l'incapsulamento si può considerare obbligatorio e l'ereditarietà un prezioso strumento di sviluppo, bisognerà chiedersi che cosa provocherà l'utilizzo combinato di entrambi i paradigmi. Ovvero: che cosa erediteremo da una classe incapsulata? Abbiamo già affermato che estendere una classe significa ereditarne i membri non privati. E' quindi escluso che, in una nuova classe `Ricorrenza` ottenuta specializzando la classe `Data` definita precedentemente, si possa accedere alle variabili `giorno`, `mese` e `anno`, direttamente, giacché queste non saranno ereditate. Ma essendo tutti i metodi d'accesso alle variabili dichiarati pubblici nella superclasse, verranno ereditati e quindi utilizzabili nella sottoclasse. Concludendo, anche se la classe `Ricorrenza` non possiede esplicitamente le variabili private di `Data`, può comunque usufruirne tramite l'incapsulamento. In pratica le possiede virtualmente.

5.6.4 Modificatore protected

Oltre ai modificatori `private` e `public`, esiste un terzo specificatore d'accesso: `protected`. Un membro dichiarato protetto sarà accessibile solo dalle classi appartenenti allo stesso package in cui è dichiarato e può essere ereditato da sottoclassi appartenenti a package differenti. I package e tutti gli specificatori d'accesso saranno argomento del modulo 9.
Come abbiamo visto, però, non è necessario dichiarare una variabile `protected` per ereditarla nelle sottoclassi. Avere a disposizione i metodi mutator (set) ed accessor (get) nelle sottoclassi è più che sufficiente. Inoltre dichiarare protetta una variabile d'istanza significa renderla pubblica a tutte le classi dello stesso package. Questo significa che la variabile non sarà veramente incapsulata per le classi dello stesso package.

L'utilizzo del modificatore `protected` inoltre, porta altre conseguenze che saranno esplicitate nel modulo 9.

5.6.5 Conclusioni

Le conclusioni che potremmo trarre dall'argomento ereditarietà sono ormai chiare ma non definitive. Il concetto di ereditarietà ci ha aperto una strada con tante diramazioni: la strada della programmazione ad oggetti. Dalla definizione dell'ereditarietà nascono fondamentali concetti come il polimorfismo, nuove potenti parole chiave come `super` e ci saranno nuove situazioni di programmazione da dover gestire correttamente. Dal prossimo modulo in poi, quindi, ci caleremo nel supporto che Java offre alla programmazione ad oggetti avanzata e nelle caratteristiche più complicate del linguaggio.

Riepilogo

In questo modulo è stato introdotto il supporto che offre Java all'object orientation. Dopo una panoramica storica sono stati elencati alcuni fondamentali paradigmi, ovvero astrazione, riuso, incapsulamento, ereditarietà e polimorfismo. Sono stati formalizzati il concetto di astrazione e quello di riuso. Anche essendo paradigmi fondamentali della programmazione orientata agli oggetti, l'astrazione e il riuso sono paradigmi validi anche per la programmazione strutturata. Per questa ragione sono spesso considerati paradigmi "secondari", ma il lettore dovrebbe tenere ben presente che il loro utilizzo è assolutamente cruciale per programmare in Java, come cercheremo di dimostrare in questo manuale.
Sono stati poi esposti al lettore i vantaggi dell'incapsulamento, ovvero maggiore manutenibilità, robustezza e riusabilità. Poi, con l'introduzione dell'ereditarietà, abbiamo aperto una nuova porta per la programmazione, che esploreremo nel prossimo modulo con il polimorfismo.
Scrivere codice per implementare incapsulamento ed ereditarietà è molto semplice. Inoltre è anche semplice capire quando applicare questi concetti: l'incapsulamento sempre, e l'ereditarietà quando vale la relazione "is a".
È stato evidenziato che l'ereditarietà e l'incapsulamento coesistono tranquillamente, anzi si convalidano a vicenda. Inoltre l'utilizzo corretto dell'astrazione è la base per non commettere errori che sarebbero amplificati da incapsulamento ed ereditarietà, e questo sarà "dimostrato" nei prossimi moduli…
Abbiamo anche introdotto il reference `this` anche se per ora non sembra una parola chiave indispensabile per programmare. La sua introduzione però, è stata utile per chiarire alcuni punti oscuri che potevano (anche inconsciamente) rappresentare un ostacolo alla completa comprensione degli argomenti.

Esercizi modulo 5

Esercizio 5.a)
Object Orientation in generale (teoria), Vero o Falso:

1. L'Object Orientation esiste solo da pochi anni.
2. Java è un linguaggio object oriented non puro, SmallTalk è un linguaggio object oriented puro.
3. Tutti i linguaggi orientati agli oggetti supportano allo stesso modo i paradigmi object oriented.
4. Si può dire che un linguaggio è object oriented se supporta incapsulamento, ereditarietà e polimorfismo; infatti altri paradigmi come l'astrazione e il riuso appartengono anche alla filosofia procedurale.
5. Applicare l'astrazione significa concentrarsi solo sulle caratteristiche importanti dell'entità da astrarre.
6. La realtà che ci circonda è fonte d'ispirazione per la filosofia object oriented
7. L'incapsulamento ci aiuta ad interagire con gli oggetti, l'astrazione ci aiuta ad interagire con le classi.
8. Il riuso è favorito dall'implementazione degli altri paradigmi object oriented.
9. L'ereditarietà permette al programmatore di gestire in maniera collettiva più classi.
10. L'incapsulamento divide gli oggetti in due parti separate: l'interfaccia pubblica e l'implementazione interna. Per l'utilizzo dell'oggetto basta conoscere l'implementazione interna e non bisogna conoscere l'interfaccia pubblica.

Esercizio 5.b)
Object Orientation in Java (teoria), Vero o Falso:

1. L'implementazione dell'ereditarietà implica scrivere sempre qualche riga in meno.
2. L'implementazione dell'incapsulamento implica scrivere sempre qualche riga in più.
3. L'ereditarietà è utile solo se si utilizza la specializzazione. Infatti, specializzando ereditiamo nella sottoclasse (o sottoclassi) membri della superclasse che non bisogna riscrivere. Con la generalizzazione invece creiamo una classe in più, e quindi scriviamo più codice.

4. Implementare l'incapsulamento non è tecnicamente obbligatorio in Java, ma indispensabile per programmare correttamente.
5. L'ereditarietà multipla non esiste in Java perché non esiste nella realtà.
6. L'interfaccia pubblica di un oggetto è costituita anche dai metodi accessor e mutator.
7. Una sottoclasse è più "grande" di una superclasse (nel senso che solitamente aggiunge caratteristiche e funzionalità nuove rispetto alla superclasse).
8. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da specializzazione tra le classi `Squadra` e `Giocatore`.
9. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da generalizzazione tra le classi `Squadra` e `Giocatore`.
10. In generale, se avessimo due classi `Padre` e `Figlio`, non esisterebbe ereditarietà tra queste due classi.

Esercizio 5.c)
Object Orientation in Java (pratica), Vero o Falso:

1. L'implementazione dell'ereditarietà implica l'utilizzo della parola chiave `extends`.
2. L'implementazione dell'incapsulamento implica l'utilizzo delle parole chiave `set` e `get`.
3. Per utilizzare le variabili incapsulate di una superclasse in una sottoclasse, bisogna dichiararle almeno `protected`.
4. I metodi dichiarati privati non vengono ereditati nelle sottoclassi.
5. L'ereditarietà multipla in Java non esiste ma si può solo simulare con le interfacce.
6. Una variabile privata risulta direttamente disponibile (tecnicamente come se fosse pubblica) tramite l'operatore dot, a tutte le istanze della classe in cui è dichiarata.
7. La parola chiave `this` permette di referenziare i membri di un oggetto che sarà creato solo al runtime all'interno dell'oggetto stesso
8. Se compiliamo la seguente classe:

```
public class CompilatorePensaciTu {
    private int var;
    public void setVar(int v) {
      var = v;
    }
    public int getVar()
      return var;
```

```
    }
  }
```

il compilatore in realtà la trasformerà in:

```
  import java.lang.*;

  public class CompilatorePensaciTu extends Object {
    private int var;
    public CompilatorePensaciTu() {
    }
    public void setVar(int v) {
      this.var = v;
    }
    public int getVar()
      return this.var;
    }
  }
```

9. Compilando le seguenti classi, non si otterranno errori in compilazione:

```
  public class Persona {
    private String nome;
    public void setNome(String nome) {
      this.nome = nome;
    }
    public String getNome() {
      return this.nome;
    }
  }

  public class Impiegato extends Persona {
    private int matricola;
    public void setMatricola(int matricola) {
      this.matricola = matricola;
    }
    public int getMatricola () {
      return this.matricola;
    }
    public String getDati() {
      return getNome() + "\nnumero" + getMatricola();
    }
  }
```

10. Alla classe `Impiegato` descritta nel punto 9) non è possibile aggiungere il seguente metodo:

```
public void setDati(String nome, int matricola) {
  setNome(nome);
  setMatricola(matricola);
}
```

perché produrrebbe un errore in compilazione.

Esercizio 5.d)
Incapsulare e completare le seguenti classi :

```
public class Pilota {
  public String nome;

  public Pilota(String nome) {
    // settare il nome
  }
}
public class Auto {
  public String scuderia;
  public Pilota pilota;

  public Auto (String scuderia, Pilota pilota) {
    // settare scuderia e pilota
  }
  public String dammiDettagli() {
    // restituire una stringa descrittiva dell'oggetto
  }

}
```

Tenere presente che le classi Auto e Pilota devono poi essere utilizzate dalle seguenti classi:

```
public class TestGara {
  public static void main(String args[]) {
    Gara imola = new Gara("GP di Imola");
    imola.corriGara();
```

```
    String risultato = imola.getRisultato();
    System.out.println(risultato);
  }
}

public class Gara {
  private String nome;
  private String risultato;
  private Auto griglia [];

  public Gara(String nome){
    setNome(nome);
    setRisultato("Corsa non terminata");
    creaGrigliaDiPartenza();
  }

  public void creaGrigliaDiPartenza(){
    Pilota uno = new Pilota("Pippo");
    Pilota due = new Pilota("Pluto");
    Pilota tre = new Pilota("Topolino");
    Pilota quattro = new Pilota("Paperino");
    Auto autoNumeroUno = new Auto("Ferrari", uno);
    Auto autoNumeroDue = new Auto("Renault", due);
    Auto autoNumeroTre = new Auto("BMW", tre);
    Auto autoNumeroQuattro = new Auto("Mercedes",
                                      quattro);
    griglia = new Auto[4];
    griglia[0] = autoNumeroUno;
    griglia[1] =  autoNumeroDue;
    griglia[2] = autoNumeroTre;
    griglia[3] =  autoNumeroQuattro;
  }

  private void corriGara() {
    int numeroVincente = (int)(Math.random()*4);
    Auto vincitore = griglia[numeroVincente];
    String risultato = vincitore.dammiDettagli();
    setRisultato(risultato);
  }
```

```
    public void setRisultato(String vincitore) {
      this.risultato = "Il vincitore di " +
this.getNome()
      + ": " + vincitore;
    }

    public String getRisultato() {
      return risultato;
    }

    public void setNome(String nome) {
      this.nome = nome;
    }

    public String getNome() {
       return nome;
    }
  }
```

Analisi dell'esercizio

La classe `TestGara` contiene il metodo `main()` e quindi determina il flusso di esecuzione dell'applicazione. È molto leggibile: si istanzia un oggetto gara e la si chiama "GP di Imola", si fa correre la corsa, si richiede il risultato e lo si stampa a video.
La classe `Gara` invece contiene pochi e semplici metodi e tre variabili d'istanza: `nome` (il nome della gara), `risultato` (una stringa che contiene il nome del vincitore della gara se è stata corsa) e griglia (un array di oggetti `Auto` che partecipano alla gara).
Il costruttore prende in input una stringa con il nome della gara che viene opportunamente settato. Inoltre il valore della stringa risultato è impostata a "Corsa non terminata". Infine è chiamato il metodo `creaGrigliaDiPartenza()`.
Il metodo `creaGrigliaDiPartenza()` istanzia quattro oggetti `Pilota` assegnando loro dei nomi. Poi, istanzia quattro oggetti `Auto` assegnando loro i nomi delle scuderie ed i relativi piloti. Infine istanzia ed inizializza l'array griglia con le auto appena create.
Una gara dopo essere stata istanziata è pronta per essere corsa.

Il metodo `corriGara()` contiene codice che va analizzato con più attenzione. Nella prima riga, infatti, viene chiamato il metodo `random()` della classe `Math` (appartenente al package `java.lang` che viene importato automaticamente). La classe `Math` astrae il concetto di "matematica" e sarà descritta nel modulo 12. Essa contiene metodi che astraggono classiche funzioni matematiche, come la radice quadrata o il logaritmo. Tra questi metodi utilizziamo il metodo `random()` che restituisce un numero generato in maniera casuale di tipo `double`, compreso tra `0` ed `0,9999999…` (ovvero il numero `double` immediatamente più piccolo di `1`). Nell'esercizio abbiamo moltiplicato per `4` questo numero, ottenendo un numero `double` casuale compreso tra `0` e `3,99999999…` Questo poi viene "castato" ad intero e quindi vengono troncate tutte le cifre decimali. Abbiamo quindi ottenuto che la variabile `numeroVincente` immagazzini al runtime un numero generato casualmente, compreso tra `0` e `3`, ovvero i possibili indici dell'array griglia.

Il metodo `corriGara()` genera quindi un numero casuale tra `0` e `3`. Lo utilizza per individuare l'oggetto `Auto` dell'array griglia che vince la gara. Per poi impostare il risultato tramite il metodo `dammiDettagli()` dell'oggetto `Auto` (che scriverà il lettore).
Tutti gli altri metodi della classe sono di tipo accessor e mutator.

Esercizio 5.e)
Data la seguente classe:

```
public class Persona {
    private String nome;

    public void setNome(String nome) {
      this.nome = nome;
    }

    public String getNome() {
      return nome;
    }

}
```

Commentare la seguente classe Impiegato, evidenziando dove sono utilizzati i paradigmi object oriented, incapsulamento, ereditarietà e riuso.

```
public class Impiegato extends Persona {

    private int matricola;
    public void setDati(String nome, int matricola) {
        setNome(nome);
        setMatricola(matricola);
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public int getMatricola() {
      return matricola;
    }

    public String dammiDettagli() {
      return getNome() + ", matricola: " +
getMatricola();
    }
}
```

Soluzioni esercizi modulo 5

Esercizio 5.a)
Object Orientation in generale (teoria), Vero o Falso:

1. **Falso** esiste dagli anni ’60.
2. **Vero**.
3. **Falso** per esempio nel C++ esiste l’ereditarietà multipla e in Java no.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Falso** bisogna conoscere l’interfaccia pubblica e non l’implementazione interna.

Esercizio 5.b)
Object Orientation in Java (teoria), Vero o Falso:

1. **Falso** il processo di generalizzazione implica scrivere una classe in più e ciò non sempre implica scrivere qualche riga in meno.
2. **Vero**.
3. **Falso** anche se dal punto di vista della programmazione la generalizzazione può non farci sempre risparmiare codice, essa ha comunque il pregio di farci gestire le classi in maniera più naturale, favorendo l’astrazione dei dati. Inoltre apre la strada all’implementazione del polimorfismo.
4. **Vero.**
5. **Falso** l’ereditarietà multipla esiste nella realtà, ma non esiste in Java perché tecnicamente implica dei problemi di difficile risoluzione come il caso dell’ereditarietà “a rombo” in C++.
6. **Vero**.
7. **Vero**.
8. **Falso** una squadra non “è un” giocatore, né un giocatore “è una” squadra. Semmai una squadra “ha un” giocatore ma questa non è la relazione di ereditarietà. Si tratta infatti della relazione di associazione…

9. **Vero** infatti entrambe le classi potrebbero estendere una classe `Partecipante`.
10. **Falso** un `Padre` è sempre un `Figlio`, o entrambe potrebbero estendere la classe `Persona`.

Esercizio 5.c)
Object Orientation in Java (pratica), Vero o Falso:

1. **Vero**.
2. **Falso** non si tratta di parole chiave ma solo di parole utilizzate per convenzione.
3. **Falso** possono essere private ed essere utilizzate tramite i metodi accessor e mutator.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero** anche se come vedremo più avanti, il costruttore di default non è vuoto.
9. **Vero.**
10. **Falso**.

Esercizio 5.d)

```
public class Pilota {
  private String nome;

  public Pilota(String nome) {
    setNome(nome);
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public String getNome() {
    return nome;
  }
}
public class Auto {
  private String scuderia;
  private Pilota pilota;
```

```
    public Auto (String scuderia, Pilota pilota) {
      setScuderia(scuderia);
      setPilota(pilota);
    }
    public void setScuderia(String scuderia) {
      this.scuderia = scuderia;
    }
    public String getScuderia() {
      return scuderia;
    }
    public void setPilota(Pilota pilota) {
      this.pilota = pilota;
    }
    public Pilota getPilota() {
      return pilota;
    }
    public String dammiDettagli() {
      return getPilota().getNome() + " su "+
getScuderia();
    }
  }
```

Esercizio 5.e)

```
  public class Impiegato extends Persona { //Ereditarietà
    private int matricola;

    public void setDati(String nome, int matricola) {
      setNome(nome); //Riuso ed ereditarietà
      setMatricola(matricola); //Riuso
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola; //incapsulamento
    }

    public int getMatricola() {
      return matricola; //incapsulamento
```

```
    }

    public String dammiDettagli() {
         //Riuso, incapsulamento ed ereditarietà
      return getNome() + ", matricola: " +
getMatricola();
    }
  }
```

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere le ragioni della nascita della programmazione ad oggetti (unità 5.1)	☐	
Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 5.2)	☐	
Saper definire ed utilizzare il concetto di astrazione (unità 5.2)	☐	
Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 5.3, 5.4)	☐	
Comprendere l'utilizzo e l'utilità del reference this (unità 5.4)	☐	
Comprendere l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 5.5, 5.6)	☐	
Conoscere la filosofia di Java per quanto riguardo la semplicità di apprendimento (unità 5.3, 5.5)	☐	

Note:

6

Complessità: alta

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere il significato del polimorfismo (unità 6.1).
2. Saper utilizzare l'overload, l'override ed il polimorfismo per dati (unità 6.2 e 6.3).
3. Comprendere e saper utilizzare le collezioni eterogenee, i parametri polimorfi, ed i metodi virtuali (unità 6.3).
4. Sapere utilizzare l'operatore instanceof ed il casting di oggetti (unità 6.3).

6 Programmazione ad oggetti utilizzando Java: polimorfismo

Questo modulo è interamente dedicato al paradigma più complesso dell'Object Orientation: il polimorfismo. Si tratta di un argomento abbastanza vasto ed articolato, che viene sfruttato relativamente poco rispetto alla potenza che mette a disposizione dello sviluppatore. Molti programmatori Java non riescono neanche a definire questo fondamentale strumento di programmazione. Alla fine di questo modulo, invece, il lettore dovrebbe comprenderne pienamente l'importanza. Nel caso non fosse così, potrà sempre tornare a leggere queste pagine in futuro. Una cosa è certa: comprendere il polimorfismo è molto più semplice che implementarlo.

6.1 Polimorfismo

Il **polimorfismo** (dal greco "molte forme") è un altro concetto che dalla realtà, è stato importato nella programmazione ad oggetti. Esso permette di riferirci con un unico termine a "entità" diverse. Ad esempio, sia un telefono fisso sia un portatile permettono di telefonare, dato che entrambi i mezzi sono definibili come telefoni. Telefonare, quindi, può essere considerata un'azione polimorfica (ha diverse implementazioni). Il polimorfismo in Java è argomento complesso, che si dirama in vari sottoargomenti. Utilizzando una convenzione con la quale rappresenteremo mediante rettangoli i concetti, e con ovali i concetti che hanno una reale implementazione in Java, cercheremo di schematizzare il polimorfismo e le sue espressioni.

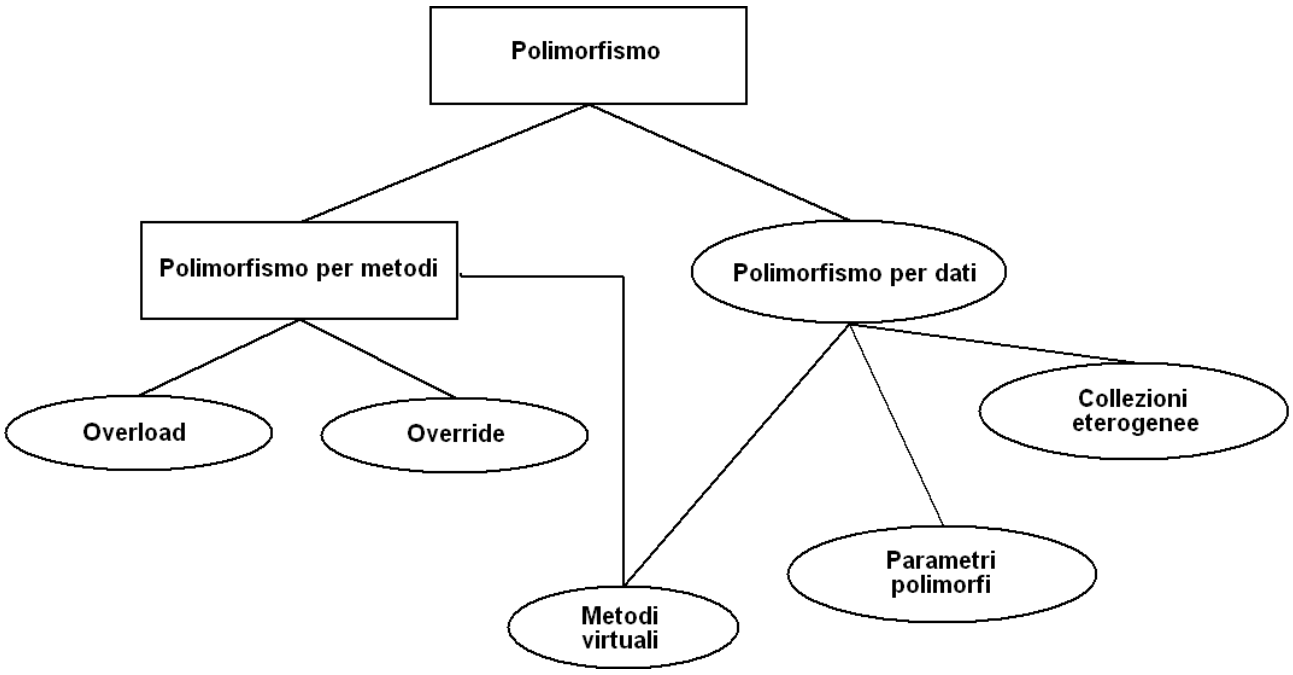


Figura 6.1 - "Il polimorfismo in Java"

6.1.1 Convenzione per i reference

Prima di iniziare a definire i vari aspetti del polimorfismo, presentiamo una convenzione per la definizione di una variabile di tipo reference. Definire precisamente che cosa sia un reference e come sia rappresentato in memoria non è cosa semplice. Di solito s'identifica un reference con un puntatore (anche se in realtà non è corretto). In molti testi relativi ad altri linguaggi di programmazione, un puntatore viene definito come "una variabile che contiene un indirizzo". In realtà la definizione di puntatore cambia da piattaforma a piattaforma!
Quindi ancora una volta utilizziamo una convenzione. Possiamo definire un reference come una variabile che contiene due informazioni rilevanti: l'indirizzo in memoria e l'intervallo di puntamento definito dalla relativa classe.
Consideriamo il seguente rifacimento della classe `Punto`:

```
public class Punto {
    private int x;
    private int y;
    public void setX(int x) {
        this.x = x;
    }
```

```
    public void setY(int y){
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

Per esempio, se scriviamo:

```
Punto ogg = new Punto();
```

possiamo supporre che il reference `ogg` abbia come indirizzo un valore numerico, ad esempio `10023823`, e come intervallo di puntamento `Punto`.
In particolare, ciò che abbiamo definito come intervallo di puntamento farà sì che il reference `ogg` possa accedere all'interfaccia pubblica della classe `Punto`, ovvero a tutti i membri pubblici (`setX(), setY(), getX(), getY()`) dichiarati nella classe `Punto` tramite il reference `ogg`.
L'indirizzo invece farà puntare il reference ad una particolare area di memoria dove risiederà il particolare oggetto istanziato. Di seguito viene riportato uno schema rappresentativo nella Figura 6.2:

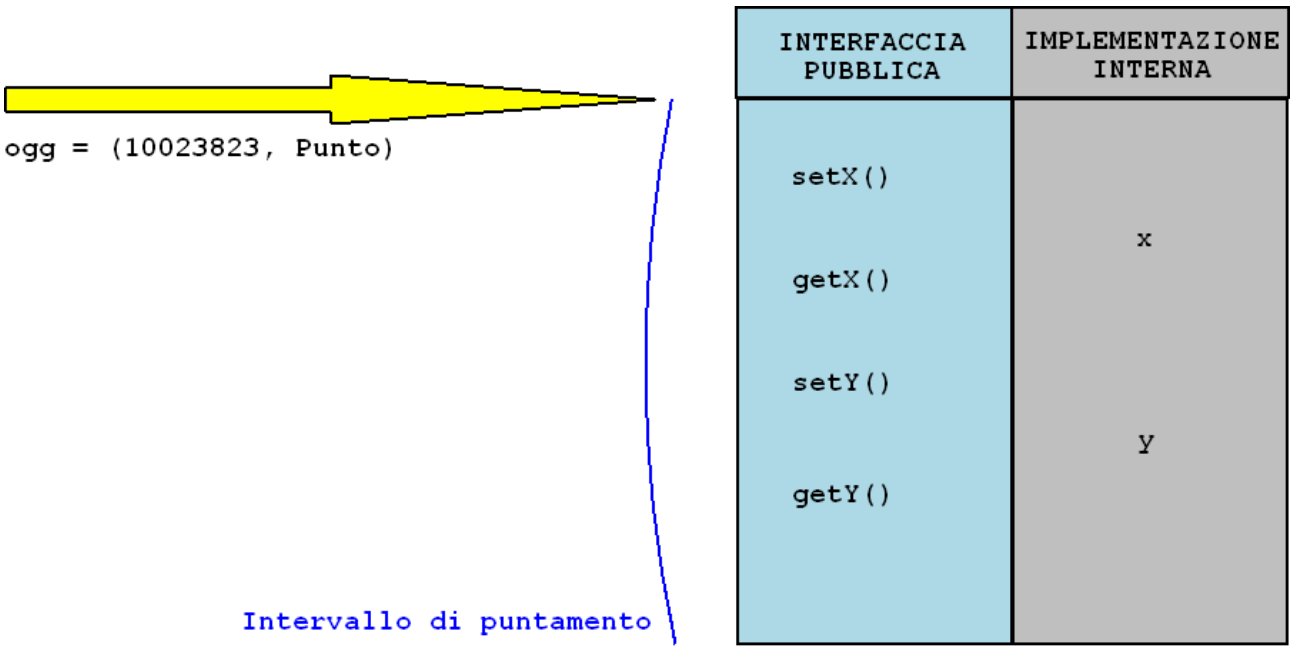


Figura 6.2: "Convenzione per i reference"

6.2 Polimorfismo per metodi

Il polimorfismo per metodi, per quanto asserito sino ad ora, ci permetterà di utilizzare lo stesso nome per metodi differenti. In Java esso trova una sua realizzazione pratica sotto due forme: l'**overload** (che potremmo tradurre con "sovraccarico") e l'**override** (che potremmo tradurre con "riscrittura").

6.2.1 Overload

Definizione 1:
In un metodo, la coppia costituita dall'identificatore e dalla lista dei parametri è detta "**segnatura**" o "**firma**" del metodo.

In Java un metodo è univocamente determinato non solo dal suo identificatore, ma anche dalla sua lista di parametri, cioè dalla sua firma. Quindi, in una classe possono convivere metodi con lo stesso nome, ma con differente firma. Su questo semplice concetto si fonda una delle caratteristiche più utili di Java: l'overload. Tramite esso il programmatore potrà utilizzare lo stesso nome per metodi diversi. Ovviamente tutto ciò deve avere un senso logico. Per esempio potremmo assegnare lo stesso nome a due

metodi che concettualmente hanno la stessa funzionalità, ma soddisfano tale funzionalità in maniera differente. Presentiamo di seguito un banale esempio di overload:

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return a + b;
    }
    public float somma(int a, float b)
    {
        return a + b;
    }
    public float somma(float a, int b)
    {
        return a + b;
    }
    public int somma(int a, int b, int c)
    {
        return a + b + c;
    }
    public double somma(int a, double b, int c)
    {
        return a + b + c;
    }
}
```

In questa classe ci sono ben cinque metodi che hanno lo stesso nome e svolgono somme, ma in modo differente. Se volessimo implementare questi metodi in un altro linguaggio che non supporta l'overload, dovremmo inventare un nome nuovo per ogni metodo. Per esempio il primo di essi si potrebbe chiamare `sommaDueInt()`, il secondo `sommaUnIntEUnFloat()`, il terzo `sommaUnFloatEUnInt()`, il quarto `sommaTreInt()`, il quinto addirittura `sommaUnIntUnDoubleEUnFloat()`! A questo punto pensiamo sia evidente al lettore l'utilità dell'overload. Notiamo che la lista dei parametri ha tre criteri di distinzione:

1. tipale (Es.: `somma(int a, int b)` è diverso da `somma(int a, float b)`)

2. numerico (Es.: `somma(int a, int b)` è diverso da `somma(int a, int b, int c)`)
3. posizionale (Es.: `somma(int a, float b)` è diverso da `somma(float a, int b)`)

Gli identificatori che utilizziamo per i parametri non sono quindi criteri di distinzione per i metodi (Esempio: `somma(int a, int b)` non è diverso da `somma(int c, int d)`).
Il tipo di ritorno non fa parte della firma di un metodo, quindi non ha importanza per l'implementazione dell'overload.

In alcuni testi l'overload non è considerato aspetto polimorfico di un linguaggio. In questi testi il polimorfismo stesso è definito in maniera diversa da com'è stato definito in questo contesto. Come sempre, è tutto relativo all'ambito in cui ci si trova. Se ci fossimo trovati a discutere di analisi e progettazione object oriented anziché di programmazione, neanche noi avremmo inserito l'overload come argomento del polimorfismo. Se prescindiamo dal linguaggio infatti, ma non è questo il nostro caso, l'overload non dovrebbe neanche esistere. Inoltre il polimorfismo è stato definito da molti autori come una conseguenza all'implementazione dell'ereditarietà. L'overload e l'ereditarietà in effetti non hanno nulla a che vedere. Se ci limitiamo però a considerare la definizione che abbiamo dato di polimorfismo (stesso nome a cose diverse), l'overload è sicuramente da considerarsi una delle implementazioni del polimorfismo.

Un altro esempio di overload (che abbiamo già sfruttato inconsapevolmente in questo testo) riguarda il metodo `println()`. Infatti esistono ben dieci metodi `println()` diversi. È possibile passare al metodo non solo stringhe, ma anche interi, boolean, array di caratteri o addirittura `Object`. Il lettore può verificarlo per esercizio consultando la documentazione (suggerimento: `System.out`, è un oggetto della classe `PrintStream` appartenente al package `java.io`).

6.2.2 Varargs

Come già accennato, quando abbiamo presentato il concetto di metodo nel Modulo 2, dalla versione 5 di Java è stata introdotta la possibilità di utilizzare come argomenti dei metodi: i cosiddetti **varargs** (abbreviativo per **variable arguments**). Con i varargs è

possibile fare in modo che un metodo accetti un numero non precisato di argomenti (compresa la possibilità di passare zero parametri), evitando così la creazione di metodi "overloadati". Per esempio, la seguente implementazione della classe `Aritmetica` potrebbe sostituire con un unico metodo l'overload dell'esempio precedente:

```
public class Aritmetica
{
    public double somma(double… doubles)
    {
        double risultato = 0.0D;
        for (double tmp : doubles)
        {
            risultato += tmp;
        }
        return risultato;
    }
}
```

Infatti, tenendo conto che `ogg` è un oggetto di tipo `Aritmetica`, segue codice valido:

```
System.out.println(ogg.somma(1,2,3));
System.out.println(ogg.somma());
System.out.println(ogg.somma(1,2));
System.out.println(ogg.somma(1,2,3,5,6,8,2,43,4));
```

Ovviamente tutti i risultati saranno di tipo `double` (a meno di casting). Segue l'output delle precedenti righe di codice:

```
0.0
6.0
3.0
74.0
```

Il risultato sarà ovviamente di tipo `double` (a meno di casting).
Effettivamente i varargs, all'interno del metodo dove sono dichiarati, sono considerati a tutti gli effetti degli array. Quindi, come per gli array, se ne può ricavare la dimensione, con la variabile `length`, e usarli nei cicli. Nell'esempio abbiamo sfruttato il nuovo costrutto foreach (già introdotto nel Modulo 4 e in dettaglio nell'Unità Didattica 17.1).

Il vantaggio di avere varargs in luogo di un array o di una collection risiede essenzialmente nel fatto che, per chiamare un metodo che dichiara argomenti variabili, non bisogna creare array o `Collection`. Un metodo con varargs viene semplicemente invocato come si fa con un qualsiasi overload. Se infatti avessimo avuto per esempio un array al posto di varargs per il metodo somma:

```
public double somma(double[] doubles)
{
    double risultato = 0.0D;
    for (double tmp : doubles)
    {
        risultato += tmp;
    }
    return risultato;
}
```

per invocare il metodo avremmo dovuto ricorrere alle seguenti righe di codice:

```
double[] doubles = {1.2D, 2, 3.14, 100.0};
System.out.println(ogg.somma(doubles));
```

Insomma, i varargs danno l'illusione dell'utilizzo dell'overload, anche se ovviamente non possono sostituirlo sempre.

E' possibile dichiarare un unico varargs a metodo. Quindi il seguente metodo darà luogo ad un errore in compilazione.
Inoltre è possibile dichiarare anche altri parametri oltre ad un (unico) varargs a metodo, ma il varargs deve occupare l'ultima posizione tra i parametri.

L'argomento varargs viene trattato in dettaglio nell'Unità Didattica 18.1.

6.2.3 Override

L'override, e questa volta non ci sono dubbi, è invece considerato una potentissima caratteristica della programmazione ad oggetti, ed è da qualcuno superficialmente identificato con il polimorfismo stesso. L'override (che potremmo tradurre con "riscrittura") è il termine object oriented che viene utilizzato per descrivere la

caratteristica che hanno le sottoclassi di ridefinire un metodo ereditato da una superclasse. Ovviamente non esisterà override senza ereditarietà. Una sottoclasse non è mai meno specifica di una classe che estende e quindi potrebbe ereditare metodi che hanno bisogno di essere ridefiniti per funzionare correttamente nel nuovo contesto.
Ad esempio, supponiamo che una ridefinizione della classe `Punto` (che per convenzione assumiamo bidimensionale) dichiari un metodo `distanzaDallOrigine()` il quale calcola, con la nota espressione geometrica, la distanza tra un punto di determinate coordinate dall'origine degli assi cartesiani. Ovviamente questo metodo ereditato all'interno di un'eventuale classe `PuntoTridimensionale` ha bisogno di essere ridefinito per calcolare la distanza voluta, tenendo conto anche della terza coordinata. Vediamo quanto appena detto sotto forma di codice:

```
public class Punto
{
    private int x, y;

    public void setX()
    {
        this.x = x;
    }
    public int getX()
    {
        return x;
    }
    public void setY()
    {
        this.y = y;
    }
    public int getY()
    {
        return y;
    }
    public double distanzaDallOrigine()
    {
        int tmp = (x*x) + (y*y);
        return Math.sqrt(tmp);
```

```
    }
}

public class PuntoTridimensionale extends Punto
{
     private int z;

     public void setZ()
     {
         this.z = z;
     }
     public int getZ()
     {
         return z;
     }

     public double distanzaDallOrigine()
     {
         int tmp = (getX()*getX()) + (getY()*getY())
           + (z*z); // N.B. : x ed y non sono ereditate
         return Math.sqrt(tmp);
     }
}
```

Per chi non ricorda come si calcola la distanza geometrica tra due punti, se abbiamo i punti `P1` e `P2` tali che hanno coordinate rispettivamente `(x1,y1)` e `(x2,y2)`, la distanza tra essi sarà data da:

`d(P1,P2)` = $\sqrt{(x1-x2)^2+(y1-y2)^2}$

Ora, se `P2` coincide con l'origine, ha coordinate `(0,0)` e quindi la distanza tra un punto `P1` e l'origine sarà data da:

`d(P1,P2)` = $\sqrt{(x1)^2+(y1)^2}$

Il metodo `sqrt()` della classe `Math` (package `java.lang`) restituisce un valore di tipo double, risultato della radice quadrata del parametro

passato (il lettore è invitato sempre e comunque a consultare la documentazione).

E' stato possibile invocarlo con la sintassi `NomeClasse.nomeMetodo` anziché `nomeOggetto.nomeMetodo` perché trattasi di un metodo statico. Discuteremo in dettaglio i metodi statici nel modulo 9. Il lettore si accontenti per il momento di sapere che un metodo statico "appartiene alla classe".
Il quadrato di `x` e di `y` è stato ottenuto mediante la moltiplicazione del valore per se stesso. Per esercizio il lettore può esplorare la documentazione della classe `Math`, per cercare l'eventuale presenza di un metodo per elevare numeri a potenza (suggerimento: come si dice potenza in inglese?).

Il lettore è invitato a riflettere sulle discutibili scelte di chiamare una classe che astrae un punto bidimensionale `Punto` e inserire il metodo `distanzaDallOrigine()` nella stessa classe. Non stiamo così violando il paradigma dell'astrazione?

Si può osservare come è stato possibile ridefinire il blocco di codice del metodo `DistanzaDallOrigine` per introdurre le terza coordinata di cui tenere conto, affinché il calcolo della distanza sia eseguito correttamente nella classe `PuntoTridimensionale`.
È bene notare che ci sono regole da rispettare per l'override.

1. Se decidiamo di riscrivere un metodo in una sottoclasse, dobbiamo utilizzare la stessa identica firma, altrimenti utilizzeremo un overload in luogo di un override.
2. Il tipo di ritorno del metodo deve coincidere con quello del metodo che si sta riscrivendo.
3. Il metodo ridefinito non deve essere meno accessibile del metodo che ridefinisce. Per esempio, se un metodo ereditato è dichiarato protetto, non si può ridefinire privato, semmai pubblico.

C'è da fare una precisazione riguardo la seconda regola. In realtà, dalla versione 5 di Java in poi, il tipo di ritorno del metodo può coincidere anche con una sottoclasse del tipo di ritorno del metodo originale. In questo caso si parla di "tipo di ritorno covariante". Per esempio, sempre considerando il rapporto di ereditarietà che

sussiste tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto`, fosse presente il seguente metodo:

```
public Punto getAddress() {
    return this;
}
```

allora sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
public PuntoTridimensionale getAddress () {
    return this;
}
```

Esiste anche un'altra regola, che sarà trattata nel modulo 10 relativo alle eccezioni.

6.2.4 Override e classe Object: metodi toString(), clone(), equals() e hashcode()

Abbiamo detto che la classe `Object` è la superclasse di tutte le classi. Ciò significa che, quando codificheremo una classe qualsiasi, erediteremo tutti gli 11 metodi di `Object`. Tra questi c'è il metodo `toString()` che avrebbe il compito di restituire una stringa descrittrice dell'oggetto.
Nella classe `Object`, che astrae il concetto di oggetto generico, tale metodo non poteva adempiere a questo scopo, giacché un'istanza della classe `Object` non ha variabili d'istanza che lo caratterizzano. E' quindi stato deciso di implementare il metodo `toString()` in modo tale che restituisca una stringa contenente informazioni sul reference del tipo:

```
NomeClasse@indirizzoInEsadecimale
```

che per la convenzione definita in precedenza potremmo interpretare come:

```
intervalloDiPuntamento@indirizzoInEsadecimale
```

Per esercizio, il lettore può provare a "stampare un reference" qualsiasi con un'istruzione del tipo `(System.out.println(nomeDiUnReference))`.
Un altro metodo degno di nota è il metodo `equals()`. Esso è destinato a confrontare due reference (sul primo viene chiamato il metodo e il secondo viene passato come parametro) e restituisce un valore booleano `true` se e solo se i due reference puntano ad uno stesso oggetto (stesso indirizzo di puntamento). Ma questo tipo di confronto, come

abbiamo avuto modo di constatare nel modulo 3, è fattibile anche mediante l'operatore ==. In effetti, il metodo `equals()` nella classe `Object` è definito come segue:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

La classe `Object` è troppo generica per poter avere un'implementazione più accurata… Quindi, in molte sottoclassi di `Object`, come `String`, il metodo `equals()` è stato riscritto in modo tale da restituire `true` anche nel caso di confronto tra due reference che puntano ad oggetti diversi, ma con gli stessi contenuti.
Se vogliamo confrontare due oggetti creati da una nostra classe, quindi, dovremmo effettuare l'override del metodo `equals()`, in modo tale che restituisca `true` se le variabili d'istanza dei due oggetti coincidono. Per esempio, un buon override del metodo `equals()` per la classe `Punto` potrebbe essere:

```
public boolean equals(Object obj) {
    if (obj instanceof Punto) return false;
        Punto that = (Punto obj);
        return this.x == that.x && this.y == that.y;
    }
}
```

Nella prima riga viene eseguito un controllo sfruttando l'operatore `instanceof` che verrà presentato tra qualche pagina, anche se ha un nome abbastanza esplicativo…

Infine, è importante sottolineare che se facciamo override del metodo `equals()` nelle nostre classi, dovremmo anche fare override di un altro metodo: `hashcode()`. Quest'ultimo viene utilizzato silenziosamente da alcune classi (molto importanti come `Hashtable` ed `HashMap`) del framework Collections (cfr. Modulo 12) in maniera complementare al metodo `equals()` per confrontare l'uguaglianza di due oggetti presenti nella stessa collezione.
Un **hash code** non è altro che un `int` che rappresenta l'univocità di un oggetto. Nelle API della classe `java.lang.Object`, nella descrizione del metodo `hashCode()`, viene introdotto il cosiddetto "contratto" di `hashcode()`: se due oggetti sono uguali relativamente al loro metodo `equals()`, allora devono avere anche lo stesso hash code.

Il viceversa non fa parte del contratto: due oggetti con lo stesso hash code non devono essere obbligatoriamente uguali relativamente al loro metodo `equals()`.
Per esempio, un buon override del metodo `hashcode()` per la classe `Punto` potrebbe essere:

```
public int hashCode() {
    return x + y;
}
```

La creazione di un buon metodo `hashcode()` potrebbe anche essere molto più complessa. Per esempio, il seguente metodo è sicuramente più efficiente del precedente per quanto riguarda sia la precisione che le prestazioni:

```
public int hashCode() {
    return x ^ y;
}
```

Altri metodi potrebbero risultare più accurati, rendendo minore la probabilità di errore, ma ovviamente bisogna tenere conto delle prestazioni. Infatti, in alcune situazioni una collezione potrebbe invocare in maniera insistente il metodo `hashcode()`. Un buon algoritmo di hash dovrebbe costituire un compromesso accettabile tra precisione e performance. Ovviamente l'implementazione dipende anche dalle esigenze dell'applicazione.
È importante comunque tenere presente queste osservazioni, per non perdere tempo in debug evitabili.
Infine, la classe `Object` definisce un metodo `clone()` (dichiarato `protected`), che restituisce una copia dell'oggetto corrente. In altre parole, la clonazione di un oggetto produce un oggetto della stessa classe dell'originale con le variabili d'istanza contenti gli stessi valori. Essendo dichiarato `protected`, per essere reso disponibile per l'invocazione nelle nostre classi, bisognerà sottoporlo a override cambiandone il modificatore a `public`, come nel seguente frammento di codice:

```
public Object clone() {
    return super.clone();
}
```

In realtà il discorso sul metodo `clone()` si dovrebbe concludere con l'implementazione da parte della classe dell'interfaccia `Clonable` e con l'inserimento della clausola "`throws CloneNotSupportedException`" per il metodo precedente. Le interfacce e le eccezioni sono però argomento che studieremo rispettivamente solo nei moduli 9 e 10.

6.2.5 Annotazione Override

Uno degli errori più subdoli che può commettere un programmatore è sbagliare un override. Ovvero ridefinire in maniera non corretta un metodo che si vuole riscrivere in una sottoclasse, magari digitando una lettera minuscola piuttosto che una maiuscola, o sbagliando il numero di parametri in input. In tali casi in compilatore non segnalerà errori, dal momento che non può intuire il tentativo di override in atto. Quindi, per esempio, il seguente codice:

```
public class MiaClasse {
    public String tostring() { // si scrive toString()
        . . .
    }
}
```

è compilato correttamente, ma non sussisterà nessun override in `MiaClasse`. Il grosso problema è che il problema si presenterà solo in fase di esecuzione dell'applicazione e non è detto che lo si riesca a correggere velocemente.
Dalla versione 5 di Java è stata introdotta una nuova struttura dati vagamente simile ad una classe, detta **Annotazione**. Un'annotazione permette di "annotare" qualsiasi tipo di componente di un programma Java: dalle variabili ai metodi, dalle classi alle annotazioni stesse! Con il termine "annotare" intendiamo qualificare, marcare. Ovvero, se per esempio annotiamo una classe, permetteremo ad un software (per esempio il compilatore Java) di "rendersi conto" che tale classe è stata marcata, così che potrà implementare un certo comportamento di conseguenza. L'argomento è piuttosto complesso e viene trattato in dettaglio nel Modulo 19. L'esempio più intuitivo di annotazione, però, definita dal linguaggio stesso, riguarda proprio il problema di implementazione dell'override che stavamo considerando. Esiste nel package `java.lang` (solo dalla versione 5 in poi) l'annotazione `Override`. Questa la si può (e la si deve) utilizzare per marcare i metodi che vogliono essere override di metodi ereditati. Per esempio:

```
public class MiaClasse {
    @Override
    public String tostring() {
        . . .
    }
}
```

Notiamo come l'utilizzo di un'annotazione richieda l'uso di un carattere del tutto nuovo alla sintassi Java: `@`.
Se marchiamo con `Override` il nostro metodo, nel caso violassimo una qualche regola dell'override, come nel seguente codice:

```
  public class MiaClasse {
    @Override
    public String tostring() {
        . . .
    }
  }
```

allora otterremmo un errore direttamente in compilazione:

```
method does not override a method from its superclass
  @Override public String tostring() {
   ^
 1 error
```

Avere un errore in compilazione, ovviamente, è molto meglio che averlo in fase di esecuzione.

Come già asserito precedentemente, Java, dalla versione 5 in poi, anche se ha abbandonato oramai la strada della semplicità, prosegue sulla strada della robustezza.

6.3 Polimorfismo per dati

Il **polimorfismo per dati** permette essenzialmente di poter assegnare un reference di una superclasse ad un'istanza di una sottoclasse. Per esempio, tenendo conto che `PuntoTridimensionale` è sottoclasse di `Punto`, sarà assolutamente legale scrivere:

```
Punto ogg = new PuntoTridimensionale();
```

Il reference `ogg`, infatti, punterà ad un indirizzo che valida il suo intervallo di puntamento. Praticamente l'interfaccia pubblica dell'oggetto creato (costituita dai metodi `setX()`, `getX()`, `setY()`, `getY()`, `setZ()`, `getZ()` e `distanzaDallOrigine()`) contiene l'interfaccia pubblica della classe `Punto` (costituita dai metodi `setX()`, `getX()`, `setY()`, `getX()` e `distanzaDallOrigine()`) e così il reference `ogg` "penserà" di puntare ad un oggetto `Punto`. Se volessimo rappresentare graficamente questa situazione, potremmo basarci sulla seguente figura 6.3:

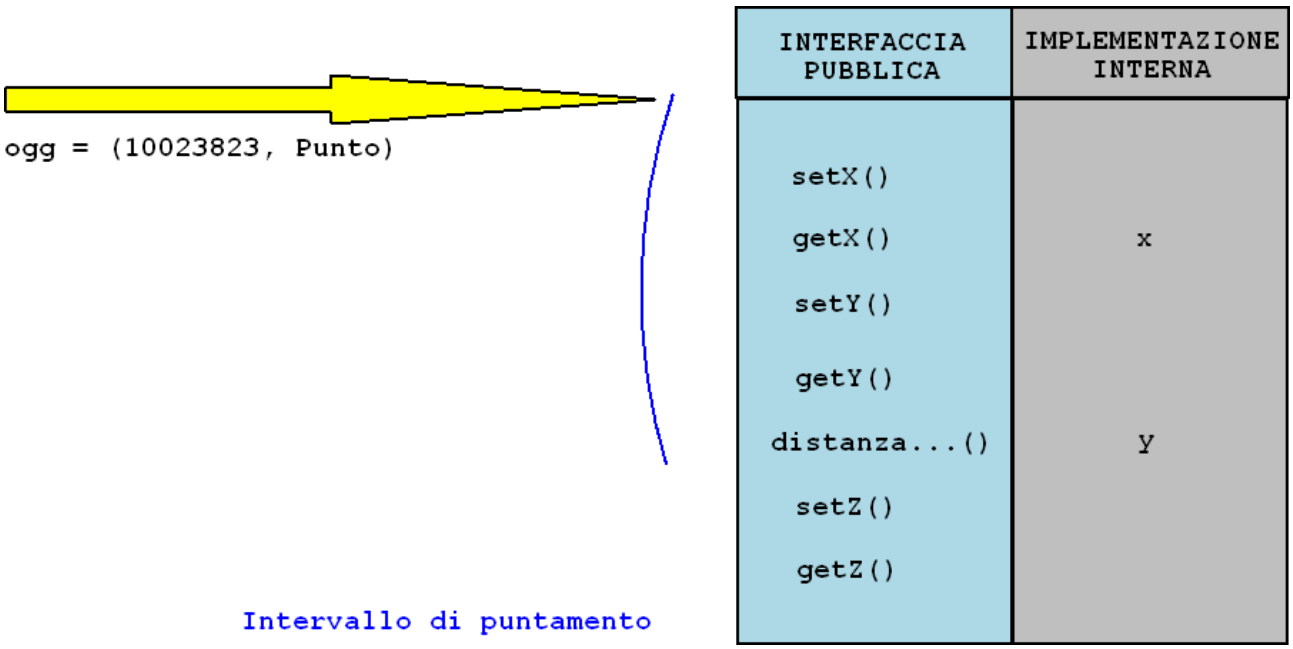


Figura 6.3: "Polimorfismo per dati"

Questo tipo d'approccio ai dati ha però un limite. Un reference di una superclasse, non potrà accedere ai campi dichiarati per la prima volta nella sottoclasse. Nell'esempio otterremmo un errore in compilazione se tentassimo di accedere ai metodi accessor e mutator, alla terza coordinata `z` del `PuntoTridimensionale`, tramite il reference `ogg`. Per esempio, la codifica della seguente riga:

```
    ogg.setZ(5);
```

produrrebbe un errore in fase di compilazione, dal momento che `ogg` è un reference che ha un intervallo di puntamento di tipo `Punto`.
A questo punto solitamente il lettore tende a chiedersi: "OK, ho capito che significa polimorfismo per dati, ma a che serve?"
Terminata la lettura di questo modulo avremo la risposta.

6.3.1 Parametri polimorfi

Sappiamo che i parametri in Java sono sempre passati per valore. Ciò implica che passare un parametro di tipo reference ad un metodo significa passare il valore numerico del reference, in altre parole il suo indirizzo. A quest'indirizzo potrebbe risiedere un oggetto istanziato da una sottoclasse, grazie al polimorfismo per dati.
In un metodo, un parametro di tipo reference si dice **parametro polimorfo** quando, anche essendo di fatto un reference relativo ad una determinata classe, può puntare ad un oggetto istanziato da una sottoclasse. In pratica, sfruttando il polimorfismo per dati, un parametro di un metodo potrebbe in realtà puntare ad oggetti diversi. È il caso del metodo `println()` che prende un parametro di tipo `Object`.

```
Punto p1 = new Punto();
System.out.println(p1);
```

Il lettore ha già appreso precedentemente che tutte le classi (compresa la classe `Punto`) sono sottoclassi di `Object`. Quindi potremmo chiamare il metodo `println()` passandogli come parametro, anziché un'istanza di `Object`, un'istanza di `Punto` come `p1`. Ma a questo tipo di metodo possiamo passare un'istanza di qualsiasi classe, dal momento che vale il polimorfismo per dati ed ogni classe è sottoclasse di `Object`.

Come già abbiamo notato, ogni classe eredita dalla classe `Object` il metodo `toString()`. Molte classi della libreria standard di Java eseguono un override di questo metodo, restituendo stringhe

descrittive dell'oggetto. Se al metodo `println()` passassimo come parametro un oggetto di una classe che non ridefinisce il metodo `toString()`, verrebbe chiamato il metodo `toString()` ereditato dalla classe `Object` (come nel caso della classe `Punto`). Una implementazione del metodo `toString()` potrebbe essere la seguente:

```
public String toString()  {
    return "(" + getX() + "," + getY() + ")";
}
```

6.3.2 Collezioni eterogenee

Una collezione eterogenea, è una collezione composta da oggetti diversi (ad esempio un array di `Object` che in realtà immagazzina oggetti diversi). Anche la possibilità di sfruttare collezioni eterogenee è garantita dal polimorfismo per dati. Infatti un array dichiarato di `Object` potrebbe contenere ogni tipo di oggetto:

```
Object arr[] = new Object[3];
arr[0] = new Punto();    //arr[0], arr[1], arr[2]
arr[1] = "Hello World!"; //sono reference ad Object
arr[2] = new Date();     //che puntano ad oggetti
                         //istanziati da sottoclassi
```

il che equivale a:

```
Object arr[]={new Punto(),"Hello World!",new Date()};
```

Presentiamo di seguito un esempio allo scopo di intuire la potenza e l'utilità di questi concetti.
Per semplicità (e soprattutto per pigrizia) non incapsuleremo le nostre classi.
Immaginiamo di voler realizzare un sistema che stabilisca le paghe dei dipendenti di un'azienda, considerando le seguenti classi:

```
public class Dipendente {
    public String nome;
    public int stipendio;
    public int matricola;
    public String dataDiNascita;
```

```
    public String dataDiAssunzione;
}

public class Programmatore extends Dipendente {
    public String linguaggiConosciuti;
    public int anniDiEsperienza;
}

public class Dirigente extends Dipendente {
    public String orarioDiLavoro;
}

public class AgenteDiVendita extends Dipendente {
    public String [] portafoglioClienti;
    public int provvigioni;
}
. . .
```

Il nostro scopo è realizzare un metodo che stabilisca le paghe dei dipendenti. Potremmo ora utilizzare una collezione eterogenea di dipendenti ed un parametro polimorfo per risolvere il problema in modo semplice, veloce e abbastanza elegante. Infatti, potremmo dichiarare una collezione eterogenea di dipendenti:

```
Dipendente [] arr = new Dipendente [180];
arr[0] = new Dirigente();
arr[1] = new Programmatore();
arr[2] = new AgenteDiVendita();
. . .
```

Esiste tra gli operatori di Java un operatore binario costituito da lettere: `instanceof`. Tramite esso si può testare a che tipo di oggetto punta in realtà un reference:

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Programmatore) {
        dip.stipendio = 1200;
    }
    else if (dip instanceof Dirigente){
        dip.stipendio = 3000;
    }
```

```
    else if (dip instanceof AgenteDiVendita) {
        dip.stipendio = 1000;
    }
    . . .
}
```

Ora, possiamo chiamare questo metodo all'interno di un ciclo foreach (di 180 iterazioni), passandogli tutti gli elementi della collezione eterogenea, e raggiungere così il nostro scopo:

```
. . .
for (Dipendente dipendente : arr) {
    pagaDipendente(dipendente);
    . . .
}
```

L'operatore `instanceof` restituisce `true` se il primo operando è un reference che punta ad un oggetto istanziato dal secondo operando o ad un oggetto istanziato da una sottoclasse del secondo operando.
Ciò implica che se il metodo `pagaDipendente()` fosse scritto nel seguente modo:

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Dipendente) {
        dip.stipendio = 1000;
    }
    else if (dip instanceof Programmatore){
        . . .
```

tutti i dipendenti sarebbero pagati allo stesso modo.

Nella libreria standard esiste un nutrito gruppo di classi fondamentali per lo sviluppo, note sotto il nome di Collections. Si tratta di collezioni tutte eterogenee e ridimensionabili. Le Collections rappresentano l'argomento principale del Modulo 12 relativo ai package `java.lang` e `java.util`.

6.3.3 Casting di oggetti

Nell'esempio precedente abbiamo osservato che l'operatore `instanceof` ci permette di testare a quale tipo di istanza punta un reference. Ma precedentemente abbiamo anche notato che il polimorfismo per dati, quando implementato, fa sì che il reference che punta ad un oggetto istanziato da una sottoclasse non possa accedere ai membri dichiarati nelle sottoclassi stesse. Esiste però la possibilità di ristabilire la piena accessibilità all'oggetto tramite il meccanismo del casting di oggetti.
Rifacciamoci all'esempio appena presentato. Supponiamo che lo stipendio di un programmatore dipenda dal numero di anni di esperienza. In questa situazione, dopo aver testato che il reference `dip` punta ad un'istanza di `Programmatore`, avremo bisogno di accedere alla variabile `anniDiEsperienza`. Se tentassimo di accedervi mediante la sintassi `dip.anniDiEsperienza`, otterremmo sicuramente un errore in compilazione. Ma se utilizziamo il meccanismo del casting di oggetti supereremo anche quest'ultimo ostacolo.
In pratica dichiareremo un reference a `Programmatore`, e lo faremo puntare all'indirizzo di memoria dove punta il reference `dip`, utilizzando il casting per confermare l'intervallo di puntamento. Il nuovo reference, essendo "giusto", ci permetterà di accedere a qualsiasi membro dell'istanza di `Programmatore`.
Il casting di oggetti sfrutta una sintassi del tutto simile al casting tra dati primitivi:

```
if (dip instanceof Programmatore) {
    Programmatore pro = (Programmatore) dip;
    . . .
```

Siamo ora in grado di accedere alla variabile `anniDiEsperienza` tramite la sintassi:

```
. . .
if (pro.anniDiEsperienza > 2)
. . .
```

Come al solito, cerchiano di chiarirci le idee cercando di schematizzare la situazione con una rappresentazione grafica:.

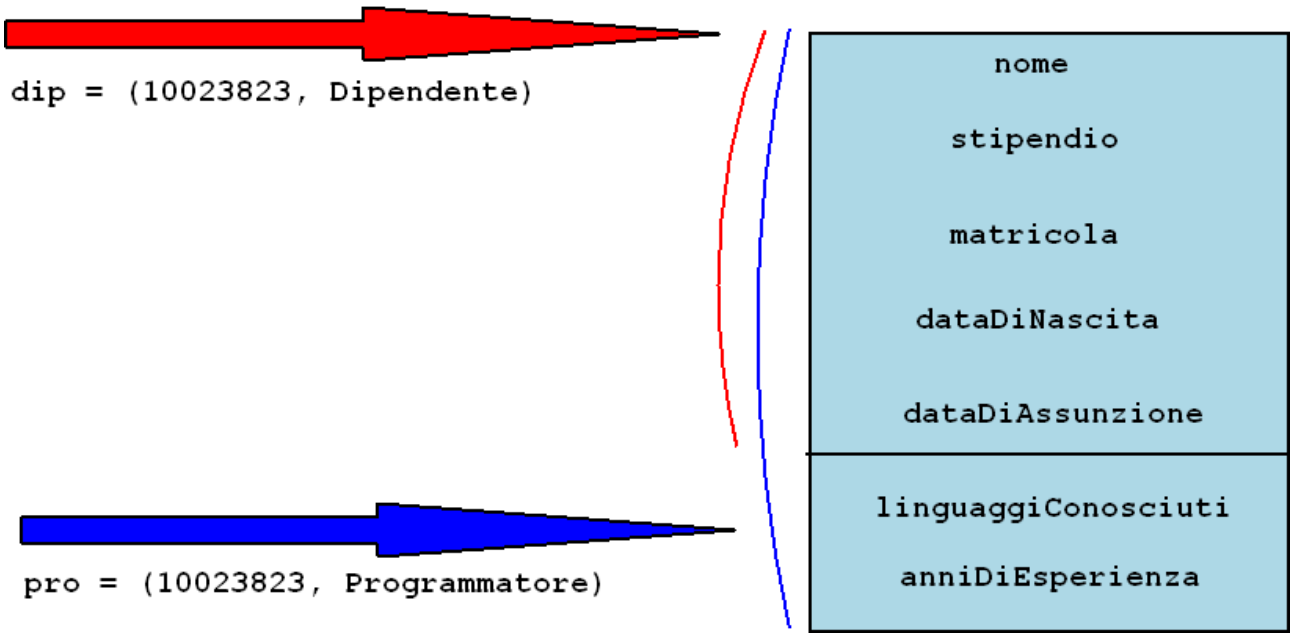


Figura 6.4: "Due diversi tipi di accesso per lo stesso oggetto"

Notare come i due reference abbiano lo stesso valore numerico (indirizzo), ma differente intervallo di puntamento. Da questo dipende l'accessibilità all'oggetto.
Nel caso tentassimo di assegnare al reference `pro` l'indirizzo di `dip` senza l'utilizzo del casting, otterremmo un errore in compilazione ed un relativo messaggio che ci richiede un casting esplicito. Ancora una volta il comportamento del compilatore conferma la robustezza del linguaggio. Il compilatore non può stabilire se ad un certo indirizzo risiede un determinato oggetto piuttosto che un altro. È solo in esecuzione che Java Virtual Machine può sfruttare l'operatore `instanceof` per risolvere il dubbio.

Il casting di oggetto non si deve considerare come strumento standard di programmazione, ma piuttosto come un utile mezzo per risolvere problemi progettuali. Una progettazione ideale farebbe a meno del casting di oggetti. Per quanto ci riguarda, all'interno di un progetto, la necessità del casting ci porta a pensare ad una "forzatura" e quindi ad un eventuale aggiornamento della progettazione.

Ancora una volta osserviamo un altro aspetto che fa definire Java linguaggio semplice. Il casting è un argomento che esiste anche in altri linguaggi e riguarda i tipi di dati primitivi numerici. Esso viene realizzato troncando i bit in eccedenza di un tipo di dato il

cui valore vuole essere forzato ad entrare in un altro tipo di dato "più piccolo". Notiamo che, nel caso di casting di oggetti, non viene assolutamente troncato nessun bit, quindi si tratta di un processo completamente diverso! Se però ci astraiamo dai tipi di dati in questione la differenza non sembra sussistere e Java permette di utilizzare la stessa sintassi, facilitando l'apprendimento e l'utilizzo al programmatore.

6.3.4 Invocazione virtuale dei metodi

Un'invocazione ad un metodo `m` può definirsi **virtuale** quando `m` è definito in una classe `A`, ridefinito in una sottoclasse `B` (override) e invocato su un'istanza di `B`, tramite un reference di `A` (polimorfismo per dati). Quando s'invoca in maniera virtuale il metodo `m`, il compilatore "pensa" di invocare il metodo `m` della classe `A` (virtualmente). In realtà viene invocato il metodo ridefinito nella classe `B`. Un esempio classico è quello del metodo `toString()` della classe `Object`. Abbiamo già accennato al fatto che esso viene sottoposto a override in molte classi della libreria standard. Consideriamo la classe `Date` del package `java.util`. In essa il metodo `toString()` è riscritto in modo tale da restituire informazioni sull'oggetto `Date` (giorno, mese, anno, ora, minuti, secondi, giorno della settimana, ora legale…). Consideriamo il seguente frammento di codice:

```
. . .
Object obj = new Date();
String s1 = obj.toString();
. . .
```

Il reference `s1` conterrà la stringa che contiene informazioni riguardo l'oggetto `Date`, unico oggetto istanziato. Schematizziamo:

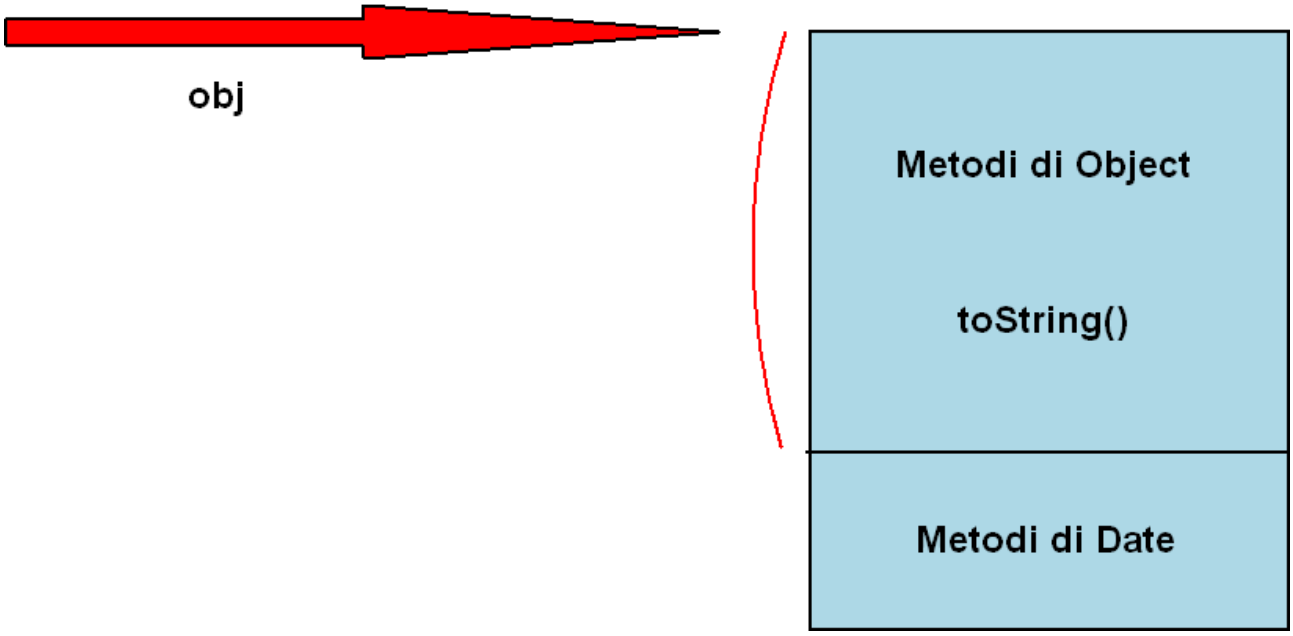


Figura 6.5: “Invocazione virtuale dei metodi dietro le quinte”

Il reference `obj` può accedere solamente all’interfaccia pubblica della classe `Object` e quindi anche al metodo `toString()`. Il reference punta però a un’area di memoria dove risiede un oggetto della classe `Date`, nella quale il metodo `toString()` ha una diversa implementazione.

Il metodo `toString()` era già stato chiamato in maniera virtuale nell’esempio del paragrafo sui parametri polimorfi.

6.3.5 Esempio d’utilizzo del polimorfismo

Supponiamo di avere a disposizione le seguenti classi:

```
public class Veicolo {
    public void accelera() {
        . . .
    }
    public void decelera() {
        . . .
```

```
    }
}

public class Aereo extends Veicolo {
    public void decolla() {
        . . .
    }
    public void atterra() {
        . . .
    }
    public void accelera() {
        // override del metodo ereditato
        . . .
    }
    public void decelera() {
        // override del metodo ereditato
        . . .
    }
    . . .
}

public class Automobile extends Veicolo {
    public void accelera() {
        // override del metodo ereditato
        . . .
    }
    public void decelera() {
        // override del metodo ereditato
        . . .
    }
    public void innestaRetromarcia() {
        . . .
    }
    . . .
}

public class Nave extends Veicolo {
    public void accelera() {
```

```
        // override del metodo ereditato
        . . .
    }
    public void decelera() {
        // override del metodo ereditato
        . . .
    }
    public void gettaAncora() {
        . . .
    }
    . . .
}
```

La superclasse `Veicolo` definisce i metodi `accelera` e `decelera`, che vengono poi ridefiniti in sottoclassi più specifiche quali `Aereo`, `Automobile` e `Nave`. Consideriamo la seguente classe, che fa uso dell'overload:

```
public class Viaggiatore {
    public void viaggia(Automobile a) {
        a.accelera();
        . . .
    }
    public void viaggia(Aereo a) {
        a.accelera();
        . . .
    }
    public void viaggia(Nave n) {
        n.accelera();
        . . .
    }
    . . .
}
```

Nonostante l'overload rappresenti una soluzione notevole per la codifica della classe `Viaggiatore`, notiamo una certa ripetizione nei blocchi di codice dei tre metodi. Sfruttando infatti un parametro polimorfo ed un metodo virtuale, la classe `Viaggiatore` potrebbe essere codificata in modo più compatto e funzionale:

```
public class Viaggiatore {
    public void viaggia(Veicolo v) { //param. Polimorfo
        v.accelera(); //metodo virtuale
        . . .
    }
    . . .
}
```

Il seguente frammento di codice infatti, utilizza le precedenti classi:

```
    Viaggiatore claudio = new Viaggiatore();
    Automobile fiat500 = new Automobile();
    // avremmo potuto istanziare anche una Nave o un
Aereo
    claudio.viaggia(fiat500);
```

Notiamo la chiarezza e la versatilità del codice. Inoltre ci siamo calati in un contesto altamente estensibile: se volessimo introdurre un nuovo `Veicolo` (supponiamo la classe `Bicicletta`), ci basterebbe codificarla senza toccare quanto scritto finora! Anche il seguente codice costituisce un valido esempio:

```
    Viaggiatore claudio = new Viaggiatore();
    Aereo piper = new Aereo();
    claudio.viaggia(piper);
```

6.3.6 Conclusioni

In questo modulo abbiamo potuto apprezzare il polimorfismo ed i suoi molteplici aspetti. Speriamo che il lettore abbia appreso almeno le definizioni presentate in questo modulo e ne abbia intuito l'utilità. Non sarà sicuramente immediato imparare ad utilizzare correttamente i potenti strumenti della programmazione ad oggetti. Certamente può aiutare molto conoscere una metodologia object oriented, o almeno UML. L'esperienza rimane come sempre la migliore "palestra".
Nel prossimo modulo sarà presentato un esercizio guidato, che vuole essere un primo esempio di approccio alla programmazione ad oggetti.

Riepilogo

In questo modulo abbiamo introdotto alcuni metodi di `Object` come `toString()`, l'operatore `instanceof` e il casting di oggetti. Abbiamo soprattutto esplorato il supporto di Java al polimorfismo, dividendolo in vari sottoargomenti. Il polimorfismo si divide in polimorfismo per metodi e per dati. Il polimorfismo per metodi è solo un concetto che trova una sua implementazione in Java tramite overload ed override. Il polimorfismo per dati ha di per sé una implementazione in Java e si esprime anche tramite parametri polimorfi e collezioni eterogenee. Inoltre l'utilizzo contemporaneo del polimorfismo per dati e dell'override dà luogo alla possibilità di invocare metodi in maniera virtuale. Tutti questi strumenti di programmazione sono molto utili ed un loro corretto utilizzo deve diventare per il lettore uno degli obiettivi fondamentali. Chiunque riesce a "smanettare" con Java, ma ci sono altri linguaggi più adatti per "smanettare" . Lo sviluppo Java va accompagnato dalla ricerca di soluzioni di progettazione per agevolare la programmazione. Se non si desidera fare questo, forse è meglio cambiare linguaggio. Una volta consigliavamo Visual Basic, ma ora anche lui è orientato agli oggetti…

Esercizi modulo 6

Esercizio 6.a)
Polimorfismo per metodi, Vero o Falso:

1. L'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diverso tipo di ritorno.
2. L'overload di un metodo implica scrivere un altro metodo con nome differente e stessa lista di parametri.
3. La segnatura (o firma) di un metodo è costituita dalla coppia identificatore – lista di parametri.
4. Per sfruttare l'override bisogna che sussista l'ereditarietà.
5. Per sfruttare l'overload bisogna che sussista l'ereditarietà.
   Supponiamo che in una classe B, la quale estende la classe A, ereditiamo il metodo:
   ```
   public int m(int a, String b) { . . . }
   ```
6. Se nella classe `B` scriviamo il metodo:
   ```
   public int m(int c, String b) { . . . }
   ```
   stiamo facendo overload e non override.
7. Se nella classe `B` scriviamo il metodo:
   ```
   public int m(String a, String b) { . . . }
   ```
   stiamo facendo overload e non override.
8. Se nella classe `B` scriviamo il metodo:
   ```
   public void m(int a, String b) { . . . }
   ```
   otterremo un errore in compilazione.
9. Se nella classe `B` scriviamo il metodo:
   ```
   protected int m(int a, String b) { . . . }
   ```
   otterremo un errore in compilazione.
10. Se nella classe `B` scriviamo il metodo:
    ```
    public int m(String a, int c) { . . . }
    ```
    otterremo un override.

Esercizio 6.b)
Polimorfismo per dati, Vero o Falso:

1. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
Veicolo v []= {new Auto(), new Aereo(), new
Veicolo()};
```

2. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Object o []= {new Veicolo(), new Aereo(), "ciao"};
```
3. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Aereo a []= {new Veicolo(), new Aereo(), new
Aereo()};
```
4. Considerando le classi introdotte in questo modulo, e se il metodo della classe viaggiatore fosse questo:
```
public void viaggia(Object o) {
  o.accelera();
}
```
potremmo passargli un oggetto di tipo `Veicolo` senza avere errori in compilazione. Per esempio:
```
claudio.viaggia(new Veicolo());
```
5. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
PuntoTridimensionale ogg = new Punto();
```
6. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
PuntoTridimensionale ogg = (PuntoTridimensionale)new
Punto();
```
7. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Punto ogg = new PuntoTridimensionale();
```
8. Considerando le classi introdotte in questo modulo, e se la classe `Piper` estende la classe `Aereo`, il seguente frammento di codice non produrrà errori in compilazione:
```
Veicolo a = new Piper();
```
9. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
String stringa = fiat500.toString();
```
10. Considerando le classi introdotte in questo modulo. Il seguente frammento di codice non produrrà errori in compilazione:
```
public void pagaDipendente(Dipendente dip) {
  if (dip instanceof Dipendente) {
```

```
  dip.stipendio = 1000;
}
else if (dip instanceof Programmatore) {
  . . .
}
```

Soluzioni esercizi modulo 6

Esercizio 6.a)
Polimorfismo per metodi, Vero o Falso:

1. **Falso** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
2. ***Falso*** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
3. **Vero.**
4. ***Vero.***
5. ***Falso*** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
6. **Falso** stiamo facendo override. L'unica differenza sta nel nome dell'identificatore di un parametro, che è ininfluente al fine di distinguere metodi.
7. **Vero** la lista dei parametri dei due metodi è diversa.
8. **Vero** in caso di override il tipo di ritorno non può essere differente.
9. **Vero** in caso di override il metodo riscritto non può essere meno accessibile del metodo originale.
10. **Falso** otterremo un overload. Infatti, le due liste di parametri differiscono per posizioni.

Esercizio 6.b)
Polimorfismo per dati, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso** non è possibile inserire in una collezione eterogenea di aerei un `Veicolo` che è superclasse di `Aereo`.
4. **Falso** la compilazione fallirebbe già dal momento in cui provassimo a compilare il metodo `viaggia()`. Infatti, non è possibile chiamare il metodo `accelera()` con un reference di tipo `Object`.
5. **Falso** c'è bisogno di un casting, perché il compilatore non sa a priori il tipo a cui punterà il reference al runtime.
6. **Vero.**
7. **Vero.**
8. **Vero** infatti `Veicolo` è superclasse di `Piper`.

9. **Vero** il metodo `toString()` appartiene a tutte le classi perché ereditato dalla superclasse `Object`**.**
10. **Vero** ma tutti i dipendenti verranno pagati allo stesso modo.

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere il significato del polimorfismo (unità 6.1)	□	
Saper utilizzare l'overload, l'override ed il polimorfismo per dati (unità 6.2 e 6.3)	□	
Comprendere e saper utilizzare le collezioni eterogenee, i parametri polimorfi ed i metodi virtuali (unità 6.3)	□	
Sapere utilizzare l'operatore instanceof ed il casting di oggetti (unità 6.3)	□	

Note:

7

Complessità: alta

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Sviluppare un'applicazione in Java utilizzando i paradigmi della programmazione ad oggetti (unità 7.1, 7.2, 7.3).

7 Un esempio guidato alla programmazione ad oggetti

In questo modulo verrà simulata la scrittura di un semplice programma, passo dopo passo, L'accento sarà posto sulle scelte e i ragionamenti che bisogna svolgere quando si programma ad oggetti. In questo modo forniremo un esempio di come affrontare, almeno per i primi tempi, i problemi della programmazione object oriented.

7.1 Perché questo modulo

Questo modulo è stato introdotto per dare al lettore una piccola ma importante esperienza, finalizzata alla corretta utilizzazione dei paradigmi della programmazione ad oggetti. Quando si approccia l'object orientation, l'obiettivo più difficile da raggiungere non è comprenderne le definizioni, che, come abbiamo visto sono derivate dal mondo reale, ma piuttosto apprendere il corretto utilizzo di esse all'interno di un'applicazione. Ciò che probabilmente potrebbe mancare al lettore è la capacità di scrivere un programma, che sicuramente non è cosa secondaria. Facciamo un esempio: se venisse richiesto di scrivere un'applicazione che simuli una rubrica, o un gioco di carte, od un'altra qualsiasi applicazione, il lettore si porrà ben presto alcune domande: "quali saranno le classi che faranno parte dell'applicazione?", "dove utilizzare l'ereditarietà?", "dove utilizzare il polimorfismo?" e così via. Sono domande cui è molto difficile rispondere, perché per ognuna di esse esistono tante risposte che sembrano tutte valide. Se dovessimo decidere quali classi comporranno l'applicazione che simuli una rubrica, potremmo decidere di codificare tre classi (`Rubrica`, `Persona`, e classe che contiene il metodo `main()`), oppure cinque (`Indirizzo`, `Ricerca`, `Rubrica`, `Persona`, e classe che contiene il metodo `main()`). Riflessioni approfondite sulle varie situazioni che si potrebbero presentare nell'utilizzare quest'applicazione (i cosiddetti "casi d'uso") probabilmente suggerirebbero la codifica di altre classi.

Una soluzione con molte classi sarebbe probabilmente più funzionale, ma richiederebbe troppo sforzo implementativo per un'applicazione che si può definire "semplice". D'altronde, un'implementazione che fa uso di un numero ridotto di classi costringerebbe lo sviluppatore ad inserire troppo codice in troppe poche classi. Ciò garantirebbe inoltre in misura minore l'utilizzo dei paradigmi della programmazione ad oggetti, giacché la nostra applicazione, più che simulare la realtà, cercherebbe di "arrangiarla".
La soluzione migliore sarà assolutamente personale, perché garantita dal buon senso e dall'esperienza.
E' già stato accennato che un importante supporto alla risoluzione di tali problemi, viene garantito dalla conoscenza di metodologie object oriented, o almeno dalla conoscenza di UML. In questa sede però, dove il nostro obiettivo principale è quello di apprendere un linguaggio di programmazione, non è consigliabile, né fattibile, introdurre anche altri argomenti tanto complessi. Tuttavia, l'appendice G di questo manuale contiene una reference sulla sintassi di UML. Ovviamente non può bastare a risolvere tutti i nostri problemi, ma può rappresentare un utile supporto allo studio dell'object orientation. Sarà presentato invece un esercizio-esempio, che il lettore può provare a risolvere, oppure direttamente studiarrne la soluzione. Con quest'esercizio ci poniamo lo scopo di operare attente osservazioni sulle scelte fatte, per poi trarne conclusioni importanti. La soluzione sarà presentata con una filosofia iterativa ed incrementale (ad ogni iterazione nel processo di sviluppo verrà incrementato il software), così com'è stata creata, sul modello delle moderne metodologie orientate agli oggetti. In questo modo saranno esposti al lettore tutti i passaggi eseguiti per arrivare alla soluzione.

7.2 Esercizio 7.a

Obiettivo:
Realizzare un'applicazione che possa calcolare la distanza geometrica tra punti. I punti possono trovarsi su riferimenti a due o tre dimensioni.

> **Con quest'esercizio non realizzeremo un'applicazione utile; lo scopo è puramente didattico.**

7.3 Risoluzione dell'esercizio 7.a

Di seguito è presentata una delle tante possibili soluzioni. Non bisogna considerarla come una soluzione da imitare, ma piuttosto come elemento di studio e riflessione per applicazioni future.

7.3.1 Passo 1

Individuiamo le classi di cui sicuramente l'applicazione non può fare a meno. Sembra evidente che componenti essenziali debbano essere le classi che devono costituire il "dominio" di quest'applicazione. Codifichiamo le classi `Punto` e `Punto3D` sfruttando incapsulamento, ereditarietà, overload di costruttori e riutilizzo del codice:

```
public class Punto {
    private int x, y;
    public Punto() {
      //Costruttore senza parametri
    }
    public Punto(int x, int y) {
        this.setXY(x, y); //Il this è facoltativo
        //riutilizziamo codice
    }
    public void setX(int x) {
        this.x = x; //Il this non è facoltativo
    }
    public void setY(int y) {
        this.y = y; //Il this non è facoltativo
    }
    public void setXY(int x, int y) {
        this.setX(x); //Il this è facoltativo
        this.setY(y);
    }
    public int getX() {
        return this.x; //Il this è facoltativo
    }
    public int getY() {
        return this.y; //Il this è facoltativo
    }
}

public class Punto3D extends Punto {
    private int z;
    public Punto3D() {
       //Costruttore senza parametri
    }
```

```
    public Punto3D(int x, int y, int z)
    {
       this.setXYZ(x, y, z); //Riuso di codice
    }
    public void setZ(int z) {
       this.z = z; //Il this non è facoltativo
    }
    public void setXYZ(int x, int y, int z) {
       this.setXY(x, y); //Riuso del codice
       this.setZ(z);     //Il this è facoltativo
    }
    public int getZ() {
       return this.z; //Il this è facoltativo
    }
}
```

Notiamo che, pur di utilizzare l'ereditarietà "legalmente", abbiamo violato la regola dell'astrazione. Infatti abbiamo assegnato l'identificatore `Punto` ad una classe che si sarebbe dovuta chiamare `Punto2D`. Avrebbe senso chiamare la classe `Punto` solo dove il contesto dell'applicazione è chiaro. Per esempio in un'applicazione che permetta di fare disegni, la classe `Punto` così definita avrebbe avuto senso. Nel momento in cui il contesto è invece generale come nel nostro esempio, non è detto che un `Punto` debba avere due dimensioni.
Ricordiamo che, per implementare il meccanismo dell'ereditarietà, lo sviluppatore deve testarne la validità, mediante la cosiddetta relazione "is a" ("è un"). Violando l'astrazione abbiamo potuto validare l'ereditarietà; ci siamo infatti chiesti: "un punto a tre dimensioni è un punto?" La risposta affermativa ci ha consentito la specializzazione. Se avessimo rispettato la regola dell'astrazione non avremmo potuto implementare l'ereditarietà tra queste classi, dal momento che, ci saremmo dovuti chiedere: "un punto a tre dimensioni è un punto a due dimensioni?" In questo caso la risposta sarebbe stata negativa. Questa scelta è stata compiuta non perché ci faciliti il prosieguo dell'applicazione, anzi, proprio per osservare come, in un'applicazione che subisce incrementi, il "violare le regole" porti a conseguenze negative. Nonostante tutto, la semplicità del problema e la potenza del linguaggio ci consentiranno di portare a termine il compito. Contemporaneamente vedremo quindi come forzare il codice affinché soddisfi i requisiti.

La codifica di due classi come queste è stata ottenuta apportando diverse modifiche. Il lettore non immagini di ottenere soluzioni ottimali al primo tentativo! Questa osservazione varrà anche relativamente alle codifiche realizzate nei prossimi passi.

7.3.2 Passo 2

Individuiamo le funzionalità del sistema. È stato richiesto che la nostra applicazione debba in qualche modo calcolare la distanza tra due punti. Facciamo alcune riflessioni prima di "buttarci sul codice". Distinguiamo due tipi di calcolo della distanza tra due punti: tra due punti bidimensionali e tra due punti tridimensionali. Escludiamo a priori la possibilità di calcolare la distanza tra due punti di cui uno sia bidimensionale e l'altro tridimensionale. A questo punto sembra sensato introdurre una nuova classe (per esempio la classe `Righello`) con la responsabilità di eseguire questi due tipi di calcoli. Nonostante questa appaia la soluzione più giusta, optiamo per un'altra strategia implementativa: assegniamo alle stesse classi `Punto` e `Punto3D` la responsabilità di calcolare le distanze relative ai "propri" oggetti. Notiamo che l'astrarre queste classi inserendo nelle loro definizioni metodi chiamati `dammiDistanza()` rappresenta una palese violazione alla regola dell'astrazione stessa. Infatti, in questo modo potremmo affermare che intendiamo un oggetto come un punto capace di "calcolarsi da solo" la distanza geometrica che lo separa da un altro punto. E tutto ciò non rappresenta affatto una situazione reale.
Quest'ulteriore violazione dell'astrazione di queste classi permetterà di valutarne le conseguenze e, contemporaneamente, di verificare la potenza e la coerenza della programmazione ad oggetti.

La distanza geometrica tra due punti bidimensionali è data dalla radice della somma del quadrato della differenza tra la prima coordinata del primo punto e la prima coordinata del secondo punto, e del quadrato della differenza tra la seconda coordinata del primo punto e la seconda coordinata del secondo punto.

Di seguito è presentata la nuova codifica della classe `Punto`, che dovrebbe poi essere estesa dalla classe `Punto3D`:

```
public class Punto {
    private int x, y;
    . . . //inutile riscrivere l'intera classe
```

```
    public double dammiDistanza(Punto p) {
    //quadrato della differenza delle x dei due punti
         int tmp1 = (x - p.x)*(x - p.x);
         //quadrato della differenza della y dei due
punti
         int tmp2 = (y - p.y)*(y - p.y);
         //radice quadrata della somma dei due quadrati
         return Math.sqrt(tmp1 + tmp2);
    }
}
```

Anche non essendo obbligati tecnicamente, per il riuso dovremmo chiamare comunque i metodi `p.getX()` e `p.getY()` piuttosto che `p.x` e `p.y`. Il lettore sa dare una spiegazione alla precedente affermazione (cfr. Modulo 5)?

Notiamo come in un eventuale metodo `main()` sarebbe possibile scrivere il seguente frammento codice:

```
Punto p1 = new Punto(5,6);
Punto p2 = new Punto(10,20);
double dist = p1.dammiDistanza(p2);
System.out.println("La distanza è " + dist);
```

7.3.3 Passo 3

Notiamo che questo metodo ereditato nella sottoclasse `Punto3D` ha bisogno di un override per avere un senso. Ma ecco che le precedenti violazioni della regola dell'astrazione iniziano a mostrarci le prime incoerenze. Infatti, il metodo `dammiDistanza()` nella classe `Punto3D`, dovendo calcolare la distanza tra due punti tridimensionali, dovrebbe prendere in input come parametro un oggetto di tipo `Punto3D`, nella maniera seguente:

```
public class Punto3D extends Punto {
    private int z;
    . . .
    //inutile riscrivere l'intera classe
    public double dammiDistanza(Punto3D p) {
```

```
    //quadrato della differenza della x dei due punti
        int tmp1=(x - p.x)*(x - p.x);
    //quadrato della differenza della y dei due punti
        int tmp2=(y - p.y)*(y - p.y);
    //quadrato della differenza della z dei due punti
        int tmp3=(z - p.z)*(z - p.z);
    //radice quadrata della somma dei tre quadrati
        return Math.sqrt(tmp1 + tpm2 + tpm3);
    }
}
```

in questa situazione, però, ci troveremmo di fronte ad un overload piuttosto che ad un override. Infatti, oltre al metodo `dammiDistanza(Punto3D p)`, sussiste in questa classe anche il metodo `dammiDistanza(Punto p)` ereditato dalla classe `Punto`. Quest'ultimo tuttavia, come abbiamo notato in precedenza, non dovrebbe sussistere in questa classe, giacché rappresenterebbe la possibilità di calcolare la distanza tra un punto bidimensionale ed uno tridimensionale.
La soluzione migliore sembra allora "forzare" un override. Possiamo riscrivere il metodo `dammiDistanza(Punto p)`. Dobbiamo però considerare il reference `p` come parametro polimorfo ed utilizzare il casting di oggetti all'interno del blocco di codice, per garantire il corretto funzionamento del metodo:

```
public class Punto3D extends Punto {
    private int z;
    . . . //inutile riscrivere l'intera classe
    public double dammiDistanza(Punto p) {
        if (p instanceof Punto3D) {
            Punto3D p1=(Punto3D)p; //Casting
            //quadrato della differenza della x dei due
            //punti
            int tmp1 = (getX()-p1.getX())*(getX()-
              p1.getX());
            //quadrato della differenza della y dei due
            //punti
            int tmp2 = (getY()-p1.getY())*(getY()-
              p1.getY());
            //quadrato della differenza della z dei due
```

```
            //punti
            int tmp3=(z-p1.z)*(z-p1.z);
            //radice quadrata della somma dei tre
            //quadrati
            return Math.sqrt(tmp1 + tpm2 + tpm3);
        }
        else {
          return -1; //distanza non valida!
        }
    }
}
```

Il metodo `dammiDistanza(Punto p)` dovrebbe ora funzionare correttamente. Infine è assolutamente consigliabile apportare una modifica stilistica al nostro codice, al fine di garantire una migliore astrazione funzionale:

```
public class Punto3D extends Punto {
    private int z;
    . . . //inutile riscrivere l'intera classe
    public double dammiDistanza(Punto p) {
      if (p instanceof Punto3D) {
      //Chiamata ad un metodo privato tramite casting
        return this.calcolaDistanza((Punto3D)p);
      }
      else {
        return -1; //distanza non valida!
      }
    }

    private double calcolaDistanza(Punto3D p1) {
        //quadrato della differenza della x dei due punti
        int tmp1=(getX()-p1.getX())*(getX()-p1.getX());
        //quadrato della differenza della y dei due punti
        int tmp2=(getY()-p1.getY())*(getY()-p1.getY());
        //quadrato della differenza della z dei due punti
        int tmp3=(z-p1.z)*(z-p1.z);
        //radice quadrata della somma dei tre quadrati
        return Math.sqrt(tmp1+tpm2+tpm3);
```

```
    }
}
```

Java ha tra le sue caratteristiche anche una potente gestione delle eccezioni, che costituisce uno dei punti di forza del linguaggio (cfr. Modulo 10). Sicuramente sarebbe meglio lanciare un'eccezione personalizzata piuttosto che ritornare un numero negativo in caso di errore.

7.3.4 Passo 4

Abbiamo adesso la possibilità di iniziare a scrivere la "classe del `main()`". Il nome da assegnarle dovrebbe coincidere con il nome dell'applicazione stessa. Optiamo per l'identificatore `TestGeometrico` giacché, piuttosto che considerare questa un'applicazione "finale", preferiamo pensare ad essa come un test per un nucleo di classi funzionanti (`Punto` e `Punto3D`) che potranno essere riusate in un'applicazione "vera".

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

Di solito, quando si inizia ad imparare un nuovo linguaggio di programmazione, uno dei primi argomenti che l'aspirante sviluppatore impara a gestire, è l'input/output nelle applicazioni. Quando invece s'approccia a Java, rimane misterioso per un certo periodo il "comando" di output:

```
System.out.println("Stringa da stampare");
```

e resta assolutamente sconosciuta per un lungo periodo una qualsiasi istruzione che permetta di acquisire dati in input! Ciò è dovuto ad una ragione ben precisa: le classi che permettono di realizzare operazioni di input/output fanno parte del package `java.io` della libreria standard. Questo package è stato progettato con una filosofia ben precisa, basata sul pattern "Decorator" (per informazioni sui pattern, cfr. Appendice H; per informazioni sul pattern Decorator, cfr. Modulo 13). Il risultato è un'iniziale difficoltà d'approccio all'argomento, compensata però da una semplicità ed efficacia "finale". Per

esempio, a un aspirante programmatore può risultare difficoltoso comprendere le ragioni per cui, per stampare una stringa a video, i creatori di Java hanno implementato un meccanismo tanto complesso (`System.out.println()`). Per un programmatore Java invece è molto semplice utilizzare gli stessi metodi per eseguire operazioni di output complesse, come scrivere in un file o mandare messaggi via socket. Rimandiamo il lettore al modulo relativo all'input/output (Modulo 13).
Per non anticipare troppo i tempi, presentiamo intanto un procedimento che permette di dotare di un minimo d'interattività le nostre prime applicazioni. Quando codifichiamo il metodo `main()`, il programmatore è obbligato a fornire una firma che definisca come parametro in input un array di stringhe (di solito chiamato `args`). Questo array immagazzinerà stringhe da riga di comando nel modo seguente. Se per lanciare la nostra applicazione, invece di scrivere a riga di comando:

```
java NomeClasseDelMain
```

scrivessimo:

```
java NomeClasseDelMain Claudio De Sio Cesari
```

all'interno della nostra applicazione avremmo a disposizione la stringa `args[0]` che ha come valore `Claudio`, la stringa `args[1]` che ha come valore `De`, la stringa `args[2]` che ha come valore `Sio`, e la stringa `args[3]` che ha come valore `Cesari`. Potremmo anche scrivere:

```
java nomeClasseDelMain Claudio "De Sio Cesari"
```

in questo modo, all'interno dell'applicazione potremmo utilizzare solamente la stringa `args[0]` che ha come valore `Claudio`, e la stringa `args[1]` che ha come valore `De Sio Cesari`.

Se come editor state utilizzando EJE, allora per poter passare parametri da riga di comando bisogna lanciare l'applicazione dal menu "sviluppo – esegui con argomenti" (in inglese "build - execute with args"). In alternativa è possibile usare la scorciatoia costituita dalla pressione dei tasti SHIFT-F9. Verrà presentata una maschera per inserire gli argomenti (e solo gli argomenti).

Codifichiamo finalmente la nostra classe `TestGeometrico`, sfruttando quanto detto, ed un metodo per trasformare una stringa in intero:

```
public class TestGeometrico {
    public static void main(String args[]) {
        //Conversione a tipo int di stringhe
        int p1X = Integer.parseInt(args[0]);
        int p1Y = Integer.parseInt(args[1]);
        int p2X = Integer.parseInt(args[2]);
        int p2Y = Integer.parseInt(args[3]);
        //Istanza dei due punti
        Punto p1 = new Punto(p1X, p1Y);
        Punto p2 = new Punto(p2X, p2Y);
        //Stampa della distanza
        System.out.println("i punti distano " +
        p1.dammiDistanza(p2));
    }
}
```

Possiamo ora lanciare l'applicazione (ovviamente dopo la compilazione), scrivendo a riga di comando per esempio:

```
java TestGeometrico 5 6 10 20
```

l'output del nostro programma sarà:

```
C:\ >java TestGeometrico 5 6 10 20
i punti distano 14.866068747318506
```

Per mandare in esecuzione quest'applicazione siamo obbligati a passare da riga di comando quattro parametri interi, per non ottenere un'eccezione. In ambito esecutivo, infatti, la Java Virtual Machine incontrerà variabili con valori indefiniti come `args[0]`.

7.3.5 Passo 5

Miglioriamo la nostra applicazione in modo tale che possa calcolare le distanze anche tra due punti tridimensionali. Introduciamo prima un test per verificare se è stato inserito il

giusto numero di parametri in input: se i parametri sono 4, viene calcolata la distanza tra due punti bidimensionali, se i parametri sono 6, si calcola la distanza tra due punti tridimensionali. In ogni altro caso viene lanciato un messaggio esplicativo e viene terminata l'esecuzione del programma, prevenendone eventuali eccezioni in fase di runtime.

```
public class TestGeometrico {
    public static void main(String args[]) {
    /* dichiariamo le variabili locali */
        Punto p1 = null, p2 = null;
    /* testiamo se sono stati inseriti il giusto numero
       di parametri */
        if (args.length == 4) {
        //Conversione a tipo int di stringhe
            int p1X = Integer.parseInt(args[0]);
            int p1Y = Integer.parseInt(args[1]);
            int p2X = Integer.parseInt(args[2]);
            int p2Y = Integer.parseInt(args[3]);
            //Istanza dei due punti
            p1 = new Punto(p1X, p1Y);
            p2 = new Punto(p2X, p2Y);
        }
        else if (args.length == 6) {
            //Conversione a tipo int di stringhe
            int p1X = Integer.parseInt(args[0]);
            int p1Y = Integer.parseInt(args[1]);
            int p1Z = Integer.parseInt(args[3]);
            int p2X = Integer.parseInt(args[4]);
            int p2Y = Integer.parseInt(args[5]);
            int p2Z = Integer.parseInt(args[6]);
            //Istanza dei due punti
            p1 = new Punto3D(p1X, p1Y, p1Z);
            p2 = new Punto3D(p2X, p2Y, p2Z);
        }
        else {
            System.out.println(
               "inserisci 4 o 6 parametri");
            System.exit(0); // Termina l'applicazione
        }
```

```
        //Stampa della distanza
        System.out.println("i punti distano "
            + p1.dammiDistanza(p2));
    }
}
```

Le classi `Punto` e `Punto3D` sono assolutamente riutilizzabili "dietro" altre applicazioni

Nei prossimi moduli, in base agli argomenti proposti, torneremo su questo esercizio sottoponendo osservazioni ed apportando modifiche.

Riepilogo

In questo modulo abbiamo dato un esempio di come si possa scrivere un'applicazione passo dopo passo. Lo scopo di questo modulo è simulare lo sviluppo di una semplice applicazione e dei problemi che si possono porre. I problemi sono stati risolti uno a uno, a volte anche forzando le soluzioni. Si è fatto notare che il non rispettare le regole (per esempio l'astrazione) porta naturalmente alla nascita di problemi. Inoltre si è cercato di dare un esempio su come si possano implementare con codice vero alcune caratteristiche object oriented.

Conclusioni Parte II

In questa sezione del manuale ci siamo dedicati esclusivamente al supporto che Java offre all'Object Orientation. Ovviamente non finisce qui! Ci sono tanti altri argomenti che vanno studiati, alcuni dei quali presentati nella prossima sezione, dedicata allo studio delle caratteristiche avanzate del linguaggio. È fondamentale capire che il lavoro di un programmatore che conosce Java ad un livello medio, ma in possesso di un buon metodo OO per lo sviluppo, vale molto più del lavoro di un programmatore con un livello di preparazione su Java formidabile, ma privo di un buon metodo OO. Il consiglio è quindi di approfondire le conoscenze sull'Object Orientation quanto prima. Ovviamente, trattandosi di argomenti di livello sicuramente più astratto, bisogna sentirsi pronti per poterli studiare con profitto. In realtà, il modo migliore per poter imparare certe argomentazioni è la pratica. Avere accanto una persona esperta (mentore) mentre si sviluppa è, dal nostro punto di vista, il modo migliore per crescere velocemente. La

teoria da imparare è infatti sterminata, , ma la cosa più complicata è applicare in pratica correttamente le tecniche descritte.
Un'ulteriore difficoltà da superare è il punto di vista degli autori delle varie metodologie. Non è raro trovare in due testi, di autori diversi, consigli opposti per risolvere la stessa tipologia di problematica. Come già asserito, quindi, il lettore deve sviluppare uno spirito critico nei confronti dei suoi studi. Ovviamente, un tale approccio allo studio dell'OO richiede una discreta esperienza di sviluppo.

Esercizi modulo 7

Esercizio 7.a)
Riprogettare l'applicazione dell'esempio cercando di rispettare le regole dell'object orientation.

Raccomandiamo al lettore di cercare una soluzione teorica prima di "buttarsi sul codice". Avere un metodo di approcciare il problema può fare risparmiare ore di debug. Questo metodo dovrà permettere quantomeno di:
1) Individuare le astrazioni chiave del progetto (le classi più importanti).
2) Assegnare loro responsabilità avendo cura dell'astrazione.
3) Individuare le relazioni tra esse.
Utilizzare UML potrebbe essere considerato (anche da chi non l'ha mai utilizzato) un modo per non iniziare a smanettare da subito con il codice…

Non viene presentata soluzione per quest'esercizio.

Esercizio 7.b)
Realizzare un'applicazione che simuli il funzionamento di una rubrica.

Il lettore si limiti a simulare la seguente situazione:
una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di persone (per esempio 5), prestabilito (le informazioni sono preintrodotte nel metodo `main()`). L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative alla persona. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona preintrodotta dall'applicazione, deve essere restituito un messaggio significativo.
Il lettore non ha altri vincoli.
Non è presentata soluzione per quest'esercizio.

Se il lettore ottiene un risultato implementativo funzionante e coerente con tutto ciò che ha studiato sino ad ora, può considerarsi veramente soddisfatto. Infatti l'esercizio proposto, è presentato spesso a corsi di formazione da noi erogati, nella giornata iniziale, per testare il livello della classe. Molto spesso, anche corsisti che si dichiarano "programmatori Java", con esperienza e/o conoscenze, non riescono ad ottenere un risultato accettabile. Ricordiamo una volta di più che questo testo vuole rendere il lettore capace di programmare in Java in

modo corretto e senza limiti. Non bisogna avere fretta! Con un po' di pazienza iniziale in più, si otterranno risultati sorprendenti. Una volta padroni del linguaggio, non esisteranno più ambiguità e misteri, e l'acquisizione di argomenti che oggi sembrano avanzati (Applet, Servlet etc…), risulterà semplice!

Soluzioni esercizi modulo 7

Esercizio 7.a)

Non è prevista una soluzione per questo esercizio.

Esercizio 7.b)

Non è prevista una soluzione per questo esercizio

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Sviluppare un'applicazione in Java, utilizzando i paradigmi della programmazione ad oggetti (unità 7.1, 7.2, 7.3)	□	

Note:

Parte III “Caratteristiche avanzate del linguaggio”

Nella parte 3 vengono trattati tutti gli argomenti che completano la conoscenza del linguaggio. Vengono quindi spiegate diverse parole chiave ed in particolare tutti i modificatori del linguaggio. Inoltre vengono presentate le definizioni di classe astratta e di interfaccia, argomenti complementari all’Object Orientation. Per completezza vengono esposti alcuni argomenti avanzati quali le classi innestate, le classi anonime, gli inizializzatori di istanze e statici. Un modulo a parte è dedicato alla gestione delle eccezioni, degli errori e delle asserzioni. Lo studio di questa sezione è in particolar modo consigliato a chi vuole conseguire la certificazione Sun Java Programmer. Più in generale, è consigliato a chi vuole conoscere ogni minima particolarità del linguaggio. Anche se vi considerate già programmatori Java, è altamente improbabile che abbiate già usato con profitto un inizializzatore d’istanza…
Al termine di questa parte, il lettore che ha studiato attentamente dovrebbe aver appreso tutti i “segreti di Java”.

8

Complessità: alta

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper definire ed utilizzare i costruttori sfruttando l'overload (unità 8.1).
2. Conoscere e saper sfruttare il rapporto tra i costruttori e il polimorfismo (unità 8.1).
3. Conoscere e saper sfruttare il rapporto tra i costruttori e l'ereditarietà (unità 8.2).
4. Saper definire ed utilizzare il reference super (unità 8.3)
5. Saper chiamare i costruttori con i reference this e super (unità 8.3).
6. Conoscere le classi interne e le classi anonime (unità 8.4).

8 Caratteristiche avanzate del linguaggio

Questo modulo è dedicato alle caratteristiche del linguaggio che solitamente sono poco conosciute, anche dai programmatori esperti. Tuttavia riteniamo molto importante la conoscenza di tali caratteristiche. Infatti, nella programmazione, a volte ci si trova di fronte a soluzioni complicate che possono diventare semplici o a bachi inspiegabili che invece possono essere risolti. Le caratteristiche avanzate di cui parleremo sono anche fondamentali per poter superare l'esame SCJP.

8.1 Costruttori e polimorfismo

Nel modulo 2 abbiamo introdotto i metodi costruttori. Essi sono stati definiti come metodi speciali, in quanto possiedono proprietà:

1. Hanno lo stesso nome della classe cui appartengono.
2. Non hanno tipo di ritorno.
3. Sono chiamati automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartengono, relativamente a quell'oggetto.
4. Sono presenti in ogni classe.

Relativamente all'ultimo punto abbiamo anche definito il costruttore di default come il costruttore che è introdotto in una classe dal compilatore al momento della

compilazione, nel caso il programmatore non gliene abbia fornito uno in maniera esplicita.
Abbiamo anche affermato che solitamente un costruttore è utilizzato per inizializzare le variabili degli oggetti al momento dell'istanza. Per esempio consideriamo la seguente classe, che fa uso dell'incapsulamento:

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    public void setNome(String n)
    {
        nome = n;
    }
    public void setIndirizzo(String ind)
    {
        indirizzo = ind;
    }
    public void setNumeroDiTelefono(String num)
    {
        numeroDiTelefono = num;
    }
    public String getNome()
    {
        return nome;
    }
    public String getIndirizzo()
    {
        return indirizzo;
    }
    public String getNumeroDiTelefono()
    {
        return numeroDiTelefono;
    }
}
```

Il lettore potrà facilmente intuire la difficoltà di utilizzare le istanze di questa classe. Infatti, per creare un cliente significativo, dopo averlo istanziato, dovremo chiamare tre metodi per inizializzare le relative variabili:

```
Cliente cliente1 = new Cliente();
cliente1.setNome("James Gosling");
cliente1.setIndirizzo("Palo Alto, California");
cliente1.setNumeroDiTelefono("0088993344556677L");
```

Nella classe `Cliente` possiamo però inserire il seguente costruttore:

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    //il seguente è un costruttore
    public Cliente(String n, String ind, String num)
    {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
    public void setNome(String n)
    {
        . . . . . .
```

Si noti che il costruttore ha lo stesso nome della classe e che non è stato dichiarato il tipo di ritorno. Per chiamare questo costruttore basta istanziare un oggetto dalla classe cliente passando i dovuti parametri, nel seguente modo:

```
Cliente cliente1 = new Cliente("James Gosling", "Palo
Alto, California",
  "0088993344556677");
```

Nell'esempio precedente, l'introduzione del costruttore esplicito implica la scomparsa del costruttore di default e la conseguente impossibilità d'istanziare oggetti con la seguente sintassi:

```
Cliente cliente1 = new Cliente();
```

infatti, in questo modo, si tenterebbe di chiamare un costruttore inesistente.
Ovviamente, come già visto nel modulo 5, è possibile sfruttare il reference `this` per migliorare la leggibilità del costruttore appena inserito:

```
public Cliente(String nome, String indirizzo, String
numeroDiTelefono)
{
    this.nome = nome;
    this.indirizzo = indirizzo;
    this.numeroDiTelefono = numeroDiTelefono;
}
```

Ancora meglio è delegare l'impostazione delle variabili d'istanza ai metodi giusti:

```
public Cliente(String nome, String indirizzo, String
numeroDiTelefono)
{
    this.setNome(nome);
    this.setIndirizzo(indirizzo);
    this.setNumeroDiTelefono(numeroDiTelefono);
}
```

8.1.1 Overload dei costruttori

Pur non essendo metodi ordinari, i costruttori sono comunque molto simili ai metodi. È quindi possibile, ed è fortemente consigliato, utilizzare l'overload dei costruttori. Infatti, nell'esempio precedente, dato che abbiamo inserito un costruttore in maniera esplicita, il compilatore non ha inserito il costruttore di default. A questo punto non sarà più possibile istanziare un oggetto senza passare parametri al costruttore. In compenso possiamo aggiungere altri costruttori alla classe, anche uno senza parametri, in maniera tale da poter costruire oggetti in modo differente in fase di runtime. Per esempio:

```
public class Cliente
{
    private String nome;
    private String indirizzo;
```

```
    private int numeroDiTelefono;
    public Cliente(String n, String ind, int num)
    {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
    public Cliente(String n)
    {
        nome = n;
    }
    public Cliente(String n,String ind)
    {
        nome = n;
        indirizzo = ind;
    }
    . . . . . .
}
```

8.1.2 Override dei costruttori

Ricordiamo che l'override consente di riscrivere un metodo ereditato da una superclasse. Ma, alle caratteristiche di un costruttore, dobbiamo aggiungerne un'altra:

5. non è ereditato dalle sottoclassi

Infatti, se per esempio la classe `Punto3D` ereditasse della classe `Punto` il costruttore `Punto()`, quest'ultimo non si potrebbe invocare, giacché la sintassi per istanziare un oggetto dalla classe `Punto3D`:

```
    Punto3D punto = new Punto3D();
```

chiamerà sempre un costruttore della classe `Punto3D`.
Possiamo concludere che, non potendo ereditare costruttori, non si può parlare di override di costruttori.

8.2 Costruttori ed ereditarietà

Il fatto che i costruttori non siano ereditati dalle sottoclassi è assolutamente in linea con la sintassi del linguaggio, ma contemporaneamente è in contraddizione con i principi della programmazione ad oggetti. In particolare sembra violata la regola dell'astrazione. Infatti, nel momento in cui lo sviluppatore ha deciso di implementare il meccanismo dell'ereditarietà, ha dovuto testarne la validità mediante la cosiddetta relazione "is a". Alla domanda: "un oggetto istanziato dalla candidata sottoclasse può considerarsi anche un oggetto della candidata superclasse?" ha infatti risposto affermativamente. Per esempio, nel modulo precedente avevamo deciso di violare l'astrazione pur di dare una risposta affermativa alla domanda "un punto tridimensionale è un punto?". Un punto tridimensionale, essendo quindi anche un punto, doveva avere tutte le caratteristiche di un punto. In particolare doveva riutilizzarne anche il costruttore. Non ereditandolo però, l'astrazione sembra violata. Invece è proprio in una situazione del genere che Java dimostra quanto sia importante utilizzare la programmazione ad oggetti in maniera corretta. Aggiungiamo infatti un'altra caratteristica ad un costruttore:

6. un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse.

Per esempio, rivediamo le classi del modulo precedente:

```
public class Punto
{
    private int x,y;
    public Punto()
    {
        System.out.println("Costruito punto
bidimensionale");
    }
    . . .
    // inutile riscrivere l'intera classe
}

public class Punto3D extends Punto
{
    private int z;
    public Punto3D()
```

```
    {
        System.out.println("Costruito punto " +
          "tridimensionale");
    }
    . . .
    // inutile riscrivere l'intera classe
}
```

Il lettore, avendo appreso che i costruttori non sono ereditati, dovrebbe concludere che l'istanza di un punto tridimensionale, mediante una sintassi del tipo:

```
new Punto3D(); /* N.B. :L'assegnazione di un reference
                  non è richiesta per istanziare un
                  oggetto */
```

produrrebbe in output la seguente stringa:

Costruito punto tridimensionale

L'output risultante sarà invece:

Costruito punto bidimensionale
Costruito punto tridimensionale

Il che rende evidente la validità della proprietà 6) dei costruttori.
La chiamata obbligatoria ad un costruttore di una superclasse viene effettuata tramite la parola chiave `super`, che viene introdotta di seguito.

8.3 super: un "super reference"

Nel modulo 5 abbiamo definito la parola chiave `this` come "reference implicito all'oggetto corrente".
Possiamo definire la parola chiave `super` come "reference implicito all'intersezione tra l'oggetto corrente e la sua superclasse".
Ricordiamo per l'ennesima volta che l'ereditarietà tra due classi si applica solo dopo aver utilizzato la relazione "is a". Consideriamo nuovamente le classi `Punto` e `Punto3D` del modulo precedente. Tramite il reference `super`, un oggetto della classe `Punto3D` potrà non solo accedere ai membri della superclasse `Punto` che sono stati

riscritti, ma addirittura ai costruttori della superclasse! Ciò significa che da un metodo della classe `Punto3D` si potrebbe invocare il metodo `dammiDistanza()` della superclasse `Punto`, mediante la sintassi:

```
super.dammiDistanza(p); /* chiamata al metodo
                           della superclasse riscritto
                           nella sottoclasse. Il
reference
                           p è di tipo Punto */
```

Ciò è possibile, ma nell'esempio specifico è improduttivo. L'esistenza del reference `super`, quale parola chiave del linguaggio, è assolutamente in linea con i paradigmi dell'object orientation. Infatti, se un punto tridimensionale è anche un punto (ereditarietà), deve poter eseguire tutto ciò che può eseguire un punto. L'inefficacia dell'utilizzo di `super` nell'esempio presentato è ancora una volta da imputare alla violazione dell'astrazione della classe `Punto`. Consideriamo il seguente codice, che si avvale di un'astrazione corretta dei dati:

```
public class Persona
{
    private String nome, cognome;
    public String toString()
    {
        return nome + " " + cognome;
    }
    . . .
    //accessor e mutator methods (set e get)
}

public class Cliente extends Persona
{
    private String indirizzo, telefono;
    public String toString()
    {
        return super.toString()+ "\n"+
        indirizzo + "\nTel:" + telefono;
    }
    . . .
```

```
    //accessor e mutator methods (set e get)
}
```

Per l'esistenza del reference `super`, preferiamo tradurre override con "riscrittura", piuttosto che con "sovrascrittura".

8.3.1 super e i costruttori

La parola chiave `super` è strettamente legata al concetto di costruttore. In ogni costruttore, infatti, è sempre presente una chiamata al costruttore della superclasse, tramite una sintassi speciale, che sfrutta il reference `super`.
Andiamo a riscrivere nuovamente le revisioni effettuate nel precedente paragrafo alle classi `Punto` e `Punto3D`, dove avevamo introdotto due costruttori senza parametri che stampano messaggi:

```
public class Punto
{
    private int x, y;
    public Punto()
    {
        System.out.println("Costruito punto
bidimensionale");
    }
    . . .
}
public class Punto3D extends Punto
{
    private int z;
    public Punto3D()
    {
        //Il compilatore inserirà qui "super();"
        System.out.println("Costruito punto
tridimensionale");
    }
    . . .
}
```

Precedentemente avevamo anche notato che un costruttore non può essere ereditato da una sottoclasse. Eppure l'ouput della seguente istruzione, in cui istanziamo un `Punto3D`:

```
    new Punto3D();
```

sarà:

```
Costruito punto bidimensionale
Costruito punto tridimensionale
```

Questo sorprendente risultato non implica che il costruttore della superclasse sia stato ereditato, ma solamente che sia stato invocato.
Infatti, al momento della compilazione, il compilatore ha inserito alla prima riga del costruttore l'istruzione `super()`. È stato quindi invocato il costruttore della classe `Punto`. La chiamata ad un costruttore della superclasse è inevitabile! L'unico modo in cui possiamo evitare che il compilatore introduca un'istruzione `super()` nei vari costruttori è introdurre esplicitamente un comando di tipo `super()`. Per esempio potremmo sfruttare al meglio la situazione modificando le due classi in questione nel seguente modo:

```
public class Punto
{
    private int x, y;
    public Punto()
    {
        super(); //inserito dal compilatore
    }
    public Punto(int x, int y)
    {
        super(); //inserito dal compilatore
        setX(x); //riuso di codice già scritto
        setY(y);
    }
    . . .
}
public class Punto3D extends Punto
{
```

```
        private int z;
        public Punto3D()
        {
            super(); //inserito dal compilatore
        }
        public Punto3D(int x, int y, int z)
        {
            super(x,y); //Chiamata esplicita al
            //costruttore con due parametri interi
            setZ(z);
        }
        . . .
}
```

In questo modo l'unica istruzione di tipo `super(x, y)` inserita esplicitamente darà luogo ad un risultato constatabile: l'impostazione delle variabili ereditate tramite costruttore della superclasse.
Attenzione: la chiamata al costruttore della superclasse mediante `super` deve essere la prima istruzione di un costruttore e, ovviamente, non potrà essere inserita all'interno di un metodo che non sia un costruttore. Anche il costruttore della classe `Punto` chiamerà il costruttore della sua superclasse `Object`.

Il rapporto tra `super` e i costruttori può essere considerato una manifestazione emblematica del fatto che Java obbliga lo sviluppatore a programmare ad oggetti.

Il lettore potrà notare la somiglianza tra il reference `super` e il reference `this`. Se tramite `super` abbiamo la possibilità (anzi l'obbligo) di chiamare un costruttore della superclasse, tramite `this` potremo invocare da un costruttore un altro costruttore della stessa classe. Presentiamo un esempio:

```
public class Persona
{
    private String nome, cognome;
    public Persona(String nome)
    {
        super();
        this.setNome(nome);
```

```
    }
    public Persona(String nome, String cognome)
    {
        this(nome); //chiamata al primo costruttore
        this.setCognome(cognome);
    }
    . . .
    //accessor e mutator methods (set e get)
}
```

La chiamata al costruttore della superclasse mediante `super` deve essere la prima istruzione di un costruttore e, ovviamente, non potrà essere inserita all'interno di un metodo che non sia un costruttore.

La chiamata ad un altro costruttore della stessa classe mediante `this` rimanda solamente la chiamata al costruttore della superclasse. Infatti questa è comunque effettuata all'interno del costruttore chiamato come prima istruzione.
Inoltre, ogni costruttore di ogni classe, una volta chiamato, direttamente o indirettamente andrà ad invocare il costruttore della classe `Object`, che non provocherà nessun risultato visibile agli occhi dello sviluppatore…

8.4 Altri componenti di un'applicazione Java: classi innestate, anonime

Nel modulo 2 abbiamo introdotto i principali componenti di un'applicazione Java. Ci sono alcuni concetti che volutamente non sono stati ancora introdotti, quali le classi astratte, le classi innestate, le classi anonime, le interfacce, le enumerazioni e le annotazioni. Di seguito introduciamo le classi innestate e le classi anonime.

I prossimi due argomenti (classi interne e classi anonime) sono presentati per completezza, giacché l'intento di questo testo è spiegare tutti i concetti fondamentali del linguaggio Java. Inoltre, tali argomenti sono indispensabili per superare l'esame di certificazione SCJP. Dobbiamo però avvertire il lettore che un utilizzo object oriented del linguaggio non richiede assolutamente l'impiego di classi interne e/o anonime. Si tratta solo di "comodità" che ci offre il linguaggio.

Il lettore può tranquillamente saltare il prossimo paragrafo se non è interessato alle ragioni "storiche" che hanno portato all'introduzione delle classi anonime e delle classi interne nel linguaggio.

8.4.1 Classi innestate: introduzione e storia

Quando nel 1995 la versione 1.0 di Java fu introdotta nel mondo della programmazione, si parlava di linguaggio orientato agli oggetti "puro". Ciò non era esatto. Un linguaggio orientato agli oggetti puro, come SmallTalk, non dispone di tipi di dati primitivi ma solo di classi da cui istanziare oggetti. Dunque non esistono operatori nella sintassi. Proviamo ad immaginare cosa significhi utilizzare un linguaggio che per sommare due numeri interi costringe ad istanziare due oggetti dalla classe `Integer` ed invocare il metodo `sum()` della classe `Math`, passandogli come parametri i due oggetti. È facile intuire perché SmallTalk non abbia avuto un clamoroso successo.
Java vuole essere un linguaggio orientato agli oggetti, ma anche semplice da apprendere. Di conseguenza non ha eliminato i tipi di dati primitivi e gli operatori. Ciononostante, il supporto che il nostro linguaggio offre ai paradigmi della programmazione ad oggetti, come abbiamo avuto modo di apprezzare, è notevole. Ricordiamo che l'object orientation è nata come scienza che vuole imitare il mondo reale, giacché i programmi rappresentano un tentativo di simulare concetti fisici e matematici importati dalla realtà che ci circonda. C'è però da evidenziare un aspetto importante delle moderne applicazioni object oriented. Solitamente dovremmo dividere un'applicazione in tre parti distinte:

- una parte rappresentata da ciò che è visibile all'utente (View), ovvero l'interfaccia grafica (in inglese "GUI": Graphical User Interface);

- una parte che rappresenta i dati e le funzionalità dell'applicazione (Model);
- una parte che gestisce la logica di controllo dell'applicazione (Controller).

Partizionare un'applicazione come descritto implica notevoli vantaggi per il programmatore. Per esempio, nel debug, semplifica la ricerca dell'errore. Quanto appena riportato non è altro che una banalizzazione di uno dei più importanti Design Pattern conosciuti, noto come Model–View–Controller Pattern o, più brevemente, MVC Pattern (una descrizione del pattern in questione è nell'appendice D di questo testo).
L'MVC propone questa soluzione architetturale, per motivi molto profondi ed interessanti, e la soluzione è molto meno banale di quanto si possa pensare.
L'applicazione di questo modello implica che l'utente utilizzi l'applicazione, generando

eventi (come il clic del mouse su un bottone) sulla View, che saranno gestiti dal Controller per l'accesso ai dati del Model. Il lettore può immaginare come in questo modello le classi che costituiscono il Model, la View ed il Controller abbiano ruoli ben definiti. L'Object Orientation ovviamente supporta l'intero processo d'implementazione dell'MVC, ma c'è un'eccezione: la View. Infatti, se un'applicazione rappresenta un'astrazione idealizzata della realtà, l'interfaccia grafica non costituisce imitazione della realtà. La GUI esiste solamente nel contesto dell'applicazione stessa, "all'interno del monitor".
In tutto questo discorso si inseriscono le ragioni della nascita delle classi innestate e delle classi anonime nel linguaggio. Nella versione 1.0 infatti, Java definiva un modello per la gestione degli eventi delle interfacce grafiche noto come "modello gerarchico". Esso effettivamente non distingueva in maniera netta i ruoli delle classi costituenti la View ed il Controller di un'applicazione. Di conseguenza avevamo la possibilità di scrivere classi che astraevano componenti grafici, i quali avevano anche la responsabilità di gestire gli eventi da essi generati. Per esempio, un bottone di un'interfaccia grafica poteva contenere il codice che doveva gestire gli eventi di cui era sorgente (bastava riscrivere il metodo `action()` ereditato dalla superclasse `Component`). Una situazione del genere non rendeva certo giustizia alle regole dell'astrazione. Ma la verità è che non esiste un'astrazione reale di un concetto che risiede all'interno delle applicazioni!
In quel periodo, se da una parte erano schierati con Sun per Java grandi società come Netscape e IBM, dall'altra parte era schierata Microsoft. Ovviamente un linguaggio indipendente dalla piattaforma non era (e non è) gradito al monopolista dei sistemi operativi. In quel periodo provennero attacchi da più fonti verso Java. Si mise in dubbio addirittura che Java fosse un linguaggio object oriented! Lo sforzo di Sun si concentrò allora nel risolvere ogni ambiguità, riscrivendo una nuova libreria di classi (e di interfacce).
Nella versione 1.1, fu definito un nuovo modello di gestione degli eventi, noto come "modello a delega" (in inglese "delegation model"). Questo nuovo modo per gestire gli eventi permette di rispettare ogni regola dell'object orientation. Il problema nuovo però, riguarda la complessità di implementazione del nuovo modello, che implica la scrittura di codice tutto sommato superfluo. Infatti, un'applicazione che deve gestire eventi con la filosofia della delega richiede la visibilità da parte di più classi sui componenti grafici della GUI. Ecco che allora, contemporaneamente alla nascita del modello a delegazione, fu definita anche una nuova struttura dati: la classe innestata ("nested class"). Il vantaggio principale delle classi innestate risiede proprio nel fatto che esse hanno visibilità "agevolata" sui membri della classe dove sono definite. Quindi, per quanto riguarda le interfacce grafiche, è possibile creare un gestore degli eventi di una certa

GUI come classe innestata all'interno della classe che rappresenta la GUI stessa. In questo modo verrà risparmiato tutto il codice di incapsulamento delle variabili d'istanza della GUI (di solito sono bottoni, pannelli etc…) che (solo in casi del genere) è superfluo. Non è molto utile incapsulare un bottone… Inoltre verrà risparmiato da parte dei gestori degli eventi il codice di accesso a tali variabili d'istanza,, che in molti casi potrebbe essere abbastanza noioso. Per approfondire il discorso, non ci resta che aspettare lo studio del Modulo 15.

8.4.2 Classe innestata: definizione

Una **classe innestata** non è altro che una classe definita all'interno di un'altra classe. Per esempio:

```
public class Outer
{
    private String messaggio = "Nella classe ";
    private void stampaMessaggio()
    {
        System.out.println(messaggio + "Esterna");
    }
/* la classe interna accede in maniera naturale ai membri
   della classe che la contiene */
    public class Inner // classe interna
    {
        public void metodo()
        {
            System.out.println(messaggio + "Interna");
        }
        public void chiamaMetodo()
        {
            stampaMessaggio();
        }
        . . .
    }
    . . .
}
```

Il vantaggio di implementare una classe all'interno di un'altra riguarda principalmente il risparmio di codice. Infatti la classe interna ha accesso alle variabili di istanza della

classe esterna. Nell'esempio, il metodo `metodo()`, della classe interna `Inner` utilizza senza problemi la variabile d'istanza `messaggio`, definita nella classe esterne `Outer`. Dal punto di vista Object Oriented, invece, di solito nessun vincolo o requisito dovrebbe consigliarci l'implementazione di una classe innestata. Questo significa che è sempre possibile evitarne l'utilizzo. Esistono dei casi dove però tali costrutti sono effettivamente molto comodi. Per esempio nella creazione di classi per la gestione degli eventi sulle interfacce grafiche. Tale argomento sarà affrontato in dettaglio nel modulo 15.

8.4.3 Classi innestate: proprietà

Le seguenti proprietà vengono riportate per completezzae perché sono importanti al fine di superare la certificazione SCJP. Tuttavia non accenneremo al fine di utilizzare le classi interne in situazioni tanto singolari. Dubitiamo fortemente che un lettore neofita abbia l'esigenza a breve scadenza di dichiarare una classe innestata statica. Sicuramente, però, il programmatore Java esperto può sfruttare con profitto tali proprietà per risolvere problemi di non facile soluzione.

Fino alla versione 1.3 di Java il termine "classe innestata" non era stato ancora adottato. Si parlava invece di "classe interna" ("inner class"). Dalla versione 1.4 in poi la Sun ha deciso di sostituire il termine "classe interna" con il termine "classe innestata" ("nested class"). Il termine "classe interna" deve ora essere utilizzato solo per le classi innestate che non sono dichiarate statiche. Quindi, anche se abbiamo parlato di classi innestate, in realtà la stragrande parte delle volte utilizzeremo classi interne.

Una classe innestata:

1) Deve avere un identificatore differente dalla classe che la contiene.
2) Si può utilizzare solo nello scope in cui è definita, a meno che non si utilizzi la sintassi: `NomeClasseEsterna.nomeClasseInterna`. In particolare, se volessimo istanziare una classe interna al di fuori della classe in cui è definita, bisogna eseguire i seguenti passi:
   a) istanziare la classe esterna (in cui è dichiarata la classe interna):
   (facendo riferimento all'esempio precedente)
   ```
   Outer outer = new Outer();
   ```
   b) dichiarare l'oggetto che si vuole istanziare dalla classe interna tramite la classe esterna:

```
    Outer.Inner inner;
```

c) istanziare l'oggetto che si vuole istanziare dalla classe interna tramite l'oggetto istanziato dalla classe esterna:

```
    inner = outer.new Inner();
```

Ha accesso sia alle variabili d'istanza sia a quelle statiche della classe in cui è dichiarata.

3) Si può dichiarare anche all'interno di un metodo, ma in quel caso, le variabili locali saranno accessibili solo se dichiarate `final`. Per esempio il seguente codice è legale:

```
public class Outer {
    private String stringaOuter = "JAVA";
    public void metodoOuter() {
        final String stringaMetodo = "5";
        class Inner{
            public void metodoInner(){
            System.out.println(stringaOuter + " " +
                stringaMetodo);
            }
        }
    }
}
```

Notare che se la variabile `stringaMetodo` non fosse stata dichiarata `final`, il codice precedente avrebbe provocato un errore in compilazione.
Notare inoltre che non è possibile dichiarare classe `Inner` con un modificatore come `private` o `public`. Trovandosi infatti all'interno di un metodo, tale dichiarazione non avrebbe senso.

4) Se viene dichiarata statica diventa automaticamente una "top-level class". In pratica non sarà più definibile come classe interna e non godrà della proprietà di poter accedere alle variabili d'istanza della classe in cui è definita. Il modificatore `static` sarà trattato esaustivamente nel prossimo modulo.
5) Solo se dichiarata statica può dichiarare membri statici.
6) Può essere dichiarata astratta (le classi astratte saranno argomento del prossimo modulo).
7) Può essere dichiarata `private` come se fosse un membro della classe che la contiene. Ovviamente una classe innestata privata non sarà istanziabile al di fuori delle classi in cui è definita.
8) Nei metodi di una classe interna è possibile utilizzare il reference `this`. Ovviamente con esso ci si può riferire ai membri della classe interna e non della classe esterna. Per referenziare un membro della classe esterna bisognerebbe

utilizzare la seguente sintassi:
9.a) `NomeClasseEsterna.this.nomeMembroDaReferenziare`
Questo è in realtà necessario solo nel caso ci sia possibilità d'ambiguità tra i membri della classe interna e i membri della classe esterna. Infatti, è possibile avere una classe interna ed una esterna che dichiarano un membro con lo stesso identificatore (supponiamo una variabile d'istanza pippo). All'interno dei metodi della classe interna, se non specifichiamo un reference per la variabile pippo, è scontato che verrà anteposto dal compilatore il reference this. Quindi verrà referenziata la variabile d'istanza della classe interna. Per referenziare la variabile della classe esterna bisogna utilizzare la sintassi 9.a).
Per esempio, l'output del seguente codice:

```
public class Outer2 {
    private String stringa = "esterna";
    public class Inner2 {
        private String stringa = "interna";
        public void metodoInner() {
            System.out.println(Outer2.this.stringa +
              " " +   this.stringa);
        }
    }
    public static void main(String [] args) {
        Outer2 outer = new Outer2();
        Outer2.Inner2 inner = outer.new Inner2();
        inner.metodoInner();
    }
}
```

sarà il seguente:

```
esterna interna
```

I modificatori `abstract` e `static`, saranno argomenti del prossimo modulo

Se compiliamo una classe che contiene una classe interna, verranno creati due file: "NomeClasseEsterna.class" e "NomeClasseEsterna$NomeClasseInterna.class".

8.4.4 Classi anonime: definizione

Come le classi innestate, le classi anonime sono state introdotte successivamente alla nascita del linguaggio: nella versione 1.2. Le classi anonime non sono altro che classi innestate, ma senza nome. Essendo classi innestate, godono delle stesse proprietà e sono utilizzate per gli stessi scopi (soprattutto per la gestione degli eventi sulle interfacce grafiche). La dichiarazione di una classe anonima richiede anche l'istanza di un suo oggetto e l'esistenza di una sua superclasse (o super interfaccia) di cui sfrutterà il costruttore (virtualmentenel caso di un'interfaccia). Se una classe non ha nome, non può avere un costruttore. La sintassi a prima vista può disorientare:

```
  public class Outer
  {
      private String messaggio = "Nella classe ";
      private void stampaMessaggio()
      {
          System.out.println(messaggio+"Esterna");
      }
      //Definizione della classe anonima e sua istanza
      ClasseEsistente ca = new ClasseEsistente()
      {
          public void metodo()
          {
              System.out.println(messaggio+"Interna");
          }
      }; //Notare il ";"
      . . .
  }
//Superclasse della classe anonima
public class ClasseEsistente
{
. . .
}
```

In pratica, quando si dichiara una classe anonima, la si deve anche istanziare. Come istanziare una classe senza nome? Una classe anonima estende sicuramente un'altra classe (nell'esempio estendeva `ClasseEsistente`) e ne sfrutta reference (grazie al polimorfismo) e costruttore (per definizione di classe anonima).

Se compiliamo una classe che contiene una classe interna, verranno creati due file: "NomeClasseEsterna.class" e "NomeClasseEsterna$1.class". Ovviamente, se introduciamo una seconda classe anonima, verrà creato anche il file "NomeClasseEsterna$2.class" e così via.

Siamo consapevoli che gli ultimi argomenti siano stati trattati con un po' troppa sinteticità. Ciò è dovuto al fatto che non abbiamo ancora introdotto librerie interessanti dove applicare tali argomenti e risulta difficile formulare esempi significativi senza disorientare il lettore. Tuttavia tali argomenti saranno ripresi ampiamente nel Modulo 15, dedicato alle interfacce grafiche.
Ricordiamo ancora al lettore che le classi innestate e le classi anonime, non sono mai necessarie per l'object orientation, ma in alcuni casi possono risultare molto comode.

Riepilogo

Abbiamo esplorato alcune caratteristiche avanzate del linguaggio. Gran parte del modulo è stato dedicato al complesso funzionamento dei costruttori quando si ha a che fare con i paradigmi dell'object orientation. A tal proposito è stato anche introdotto il reference `super` con le sue potenti possibilità d'utilizzo. Inoltre si è anche completato il discorso sull'altro reference implicito `this`, introdotto nel modulo 5. Questi due reference hanno anche uno stretto rapporto con i costruttori. Conoscere il comportamento dei costruttori è molto importante.
La parte finale del modulo è stata dedicata all'introduzione di altre tipologie di componenti del linguaggio: le classi innestate e le classi anonime. L'utilizzo di queste due ultime definizioni, è limitato a situazioni particolari, dove due classi si trovano in stretta dipendenza tra loro.

Esercizi modulo 8

Esercizio 8.a)
Caratteristiche avanzate del linguaggio, Vero o Falso:

1. Qualsiasi costruttore scritto da uno sviluppatore invocherà un costruttore della superclasse o un altro della stessa classe.
2. Qualsiasi costruttore di default invocherà un costruttore della superclasse o un altro della stessa classe.
3. Il reference `super`permette ad una sottoclasse di riferirsi ai membri della superclasse.
4. L'override di un costruttore non è possibile, perché i costruttori non sono ereditati. L'overload di un costruttore è invece sempre possibile.
5. Il comando `this([parametri])` permette ad un metodo di invocare un costruttore della stessa classe in cui è definito.
6. I comandi `this([parametri])` e `super([parametri])` sono mutuamente esclusivi e uno di loro deve essere per forza la prima istruzione di un costruttore.
7. Non è possibile estendere una classe con un unico costruttore dichiarato privato.
8. Una classe innestata è una classe che viene dichiarata all'interno di un'altra classe.
9. Una classe anonima è anche innestata, ma non ha nome. Inoltre, per essere dichiarata, deve per forza essere istanziata
10. Le classi innestate non sono molto importanti per programmare in Java e non sono necessarie per l'object orientation.

Soluzioni esercizi modulo 8

Esercizio 8.a)
Caratteristiche avanzate del linguaggio, Vero o Falso:

1. **Vero.**
2. **Falso** qualsiasi costruttore di default invocherà il costruttore della superclasse tramite il comando `super()` inserito dal compilatore automaticamente.
3. **Vero.**
4. **Vero.**
5. **Falso** il comando `this()` permette solo ad un costruttore di invocare un altro costruttore della stessa classe in cui è definito.
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper definire ed utilizzare i costruttori sfruttando l'overload (unità 8.1)	□	
Conoscere e saper sfruttare il rapporto tra i costruttori e il polimorfismo (unità 8.1)	□	
Conoscere e saper sfruttare il rapporto tra i costruttori ed ereditarietà (unità 8.2)	□	
Saper definire ed utilizzare il reference super (unità 8.3)	□	
Saper chiamare i costruttori con i reference this e super (unità 8.3)	□	
Conoscere le classi interne e le classi anonime (unità 8.4)	□	

Note:

9

Complessità: alta

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper utilizzare tutti i modificatori d'accesso (unità 9.1, 9.2) .
2. Saper dichiarare ed importare package (unità 9.3).
3. Saper utilizzare il modificatore final (unità 9.4).
4. Saper utilizzare il modificatore static (unità 9.5).
5. Saper utilizzare il modificatore abstract (unità 9.6).
6. Comprendere l'utilità di classi astratte ed interfacce (unità 9.6, 9.7).
7. Comprendere e saper utilizzare l'ereditarietà multipla (unità 9.7).
8. Comprendere e saper utilizzare le enumerazioni (unità 9.8).
9. Saper accennare alle definizione dei modificatori strictfp, volatile e native (unità 9.9).

9 Modificatori, package ed interfacce

In questo modulo dedicheremo la nostra attenzione ad argomenti particolarmente importanti per la programmazione Java. Per prima cosa definiremo tutti i modificatori del linguaggio: da quelli particolarmente importanti (per esempio `static`), a quelli che non si utilizzano praticamente mai (per esempio `volatile`). A partire dalla discussione sui vari modificatori, sconfineremo in tre argomenti importantissimi e correlati alla definizione di alcuni modificatori. Infatti introdurremo i package, le interfacce e le enumerazioni, argomenti cardine del linguaggio. Per quanto riguarda i package, la loro tardiva introduzione è dovuta ad un percorso didattico ben preciso che abbiamo voluto seguire. Nel paragrafo relativo verranno trattati in dettaglio e saranno esplicitate le ragioni per cui l'argomento è stato introdotto solo nel modulo 9. Con la trattazione delle interfacce, aggiungeremo un altro tassello importantissimo per programmare in maniera object oriented. Infine, sarà presentata una delle novità più importanti introdotte dalla versione 5 del linguaggio: le enumerazioni, argomento tutt'altro che semplice, ulteriormente approfondito nel modulo 17.

9.1 Modificatori fondamentali

Un modificatore è una parola chiave capace di cambiare il significato di un componente di un'applicazione Java. Nonostante si tratti di un concetto fondamentale, non sempre l'utilizzo di un modificatore ha un chiaro significato per il programmatore. Non è raro leggere programmi Java che abusano di modificatori senza un particolare motivo.
Come già asserito nel modulo 1, un modificatore sta ad un componente di un'applicazione Java come un aggettivo sta ad un sostantivo nel linguaggio umano.
Si possono anteporre alla dichiarazione di un componente di un'applicazione Java anche più modificatori alla volta, senza tener conto dell'ordine in cui vengono anteposti. Una variabile dichiarata `static` e `public` avrà quindi le stesse proprietà di una dichiarata `public` e `static`.
Il seguente schema riassume le associazioni tra la lista completa dei modificatori, e i relativi componenti cui si possono applicare:

MODIFICATORE	CLASSE	ATTRIBUTO	METODO	COSTRUTTORE	BLOCCO DI CODICE
`public`	sì	sì	sì	sì	no
`protected`	no	sì	sì	sì	no
`(default)`	sì	sì	sì	sì	sì
`private`	no	sì	sì	sì	no
`abstract`	sì	no	sì	no	no
`final`	sì	sì	sì	no	no
`native`	no	no	sì	no	no
`static`	no	sì	sì	no	sì
`strictfp`	sì	no	sì	no	no
`synchronized`	no	no	sì	no	no
`transient`	no	sì	no	no	no

Si noti che con la sintassi `(default)` intendiamo indicare la situazione in cui non anteponiamo alcun modificatore alla dichiarazione di un componente. In questo modulo approfondiremo solo alcuni dei modificatori fondamentali che Java mette a disposizione; tutti gli altri riguardano argomenti avanzati che sono trattati molto raramente. Di questi daremo giusto un'introduzione alla fine del modulo. Inoltre, i modificatori

`synchronized` e `transient` saranno trattati approfonditamente nei moduli 11 e 13, rispettivamente. Questo perché richiedono definizioni aggiuntive per poter essere compresi, riguardanti argomenti complessi come il multithreading e la serializzazione. Inizieremo la nostra carrellata dai cosiddetti modificatori d'accesso (detti anche specificatori d'accesso), con i quali abbiamo già una discreta familiarità. Ne approfitteremo per puntualizzare il discorso sull'utilizzo dei package.

9.2 Modificatori d'accesso

I **modificatori di accesso** regolano essenzialmente la visibilità e l'accesso ad un componente Java:

public:
Può essere utilizzato sia relativamente ad un membro (attributo o metodo) di una classe, sia relativamente ad una classe stessa. Abbiamo già abbondantemente analizzato l'utilizzo relativo ad un membro di una classe. Sappiamo oramai bene che un membro dichiarato pubblico sarà accessibile da una qualsiasi classe situata in qualsiasi package. Una classe dichiarata pubblica sarà anch'essa visibile da un qualsiasi package.

protected:
Questo modificatore definisce per un membro il grado più accessibile dopo quello definito da `public`. Un membro protetto sarà infatti accessibile all'interno dello stesso package ed in tutte le sottoclassi della classe in cui è definito, anche se non appartenenti allo stesso package.

Default:
Possiamo evitare di usare modificatori sia relativamente ad un membro (attributo o metodo) di una classe, sia relativamente ad una classe stessa. Se non anteponiamo modificatori d'accesso ad un membro di una classe, esso sarà accessibile solo da classi appartenenti al package dove è definito. Se dichiariamo una classe appartenente ad un package senza anteporre alla sua definizione il modificatore `public`, la classe stessa sarà visibile solo dalle classi appartenenti allo stesso package. Possiamo considerare allora anche un incapsulamento di secondo livello.

private:
Questo modificatore restringe la visibilità di un membro di una classe alla classe stessa (ma bisogna tener conto di quanto osservato nell'Unità Didattica 5.3, nel paragrafo intitolato "Seconda osservazione sull'incapsulamento").

Il tutto è riassunto nella seguente tabella riguardante i modificatori di accesso e la relativa visibilità (solo per i membri di una classe):

MODIFICATORE	STESSA CLASSE	STESSO PACKAGE	SOTTOCLASSE	DAPPERTUTTO
`public`	sì	sì	sì	sì
`protected`	sì	sì	sì	no
`(default)`	sì	sì	no	no
`private`	sì	no	no	no

9.2.1 Attenzione a `protected`

Il modificatore `protected` viene spesso utilizzato in maniera impropria. Infatti molti programmatori preferiscono ereditare nelle sottoclassi direttamente una variabile `protected` piuttosto che lasciarla `private`, ed ereditarne i metodi "set "e "get" pubblici. In realtà l'utilizzo di `protected` potrebbe essere limitato a casi abbastanza rari, nella maggior parte dei casi non c'è una vera necessità di utilizzare questo modificatore. Bisogna anche tener conto che l'utilizzo di tale modificatore, implica anche una strana limitazione. Infatti , se una classe `A` definisce un metodo `m()` protetto, un'eventuale sottoclasse `B` appartenente ad un package diverso da quello di `A`, può accedere al metodo della superclasse mediante il reference `super`, ma non può invocare tale metodo su altre istanze di altre classi che non siano di tipo `B`. Facciamo un esempio, consideriamo la classe seguente:

```
package package1;

public class Superclasse {
  protected void metodo(){

  }
}
```

Poi consideriamo la seguente sottoclasse:

```
package package2;

import package1.*;

public class Sottoclasse extends Superclasse {
  protected void metodo(){

  }

  public void chiamaMetodoValido1(){
    super.metodo();
  }

  public void chiamaMetodoValido2(Sottoclasse oggetto){
    oggetto.metodo();
  }

  public void chiamaMetodoNonValido(Superclasse oggetto){
    oggetto.metodo();
  }
}
```

Nella classe Sottoclasse abbiamo ridefinito il metodo chiamato `metodo()`. È possibile farlo, anche se siamo in una classe appartenente ad un package differente rispetto alla classe estesa. Il metodo denominato `chiamaMetodoValido1()`, invoca legalmente il metodo riscritto. Invece il metodo `chiamaMetodoNonValido(Superclasse oggetto)`, non è valido perché prova chiamare il metodo riscritto su un'istanza che non è a priori di tipo sottoclasse. L'errore risultante dalla compilazione sarà il seguente:

```
metodo() has protected access in package1.Superclasse
  oggetto.metodo();
      ^
1 error
```

Infine il metodo `chiamaMetodoValido2(Sottoclasse oggetto)`, è ovviamente legale perché prova ad invocare il metodo denominato `metodo()`, su

un'istanza della stessa classe (in questo caso avrebbe compilato anche se il metodo fosse stato dichiarato `private`; cfr. Modulo 5).
Abbiamo ora un motivo in più per scoraggiare l'utilizzo del modificatore `protected`!

9.3 Gestione dei package

Abbiamo visto come la libreria standard di Java sia organizzata in package. Questo permette di consultare la documentazione in maniera più funzionale, potendo ricercare le classi che ci interessano, limitandoci al package di appartenenza. Grazie ai package, il programmatore ha quindi la possibilità di organizzare anche le classi scritte da sé medesimo. È molto semplice infatti dichiarare una classe appartenente ad un package. La parola chiave `package` permette di specificare, prima della dichiarazione della classe (l'istruzione `package` deve essere assolutamente la prima in un file Java) il package di appartenenza. Ecco un frammento di codice che ci mostra la sintassi da utilizzare:

```
package programmi.gestioneClienti;
public class AssistenzaClienti
{
    . . . . .
```

In questo caso la classe `AssistenzaClienti` apparterrà al package `gestioneClienti`, che a sua volta appartiene al package `programmi`.
I package fisicamente sono cartelle (directory). Ciò significa che, dopo aver dichiarato la classe appartenente a questo package, dovremo inserire la classe compilata all'interno di una cartella chiamata `gestioneClienti`, situata a sua volta all'interno di una cartella chiamata `programmi`. Di solito il file sorgente va tenuto separato dalla classe compilata così come schematizzato in figura 9.1, dove abbiamo idealizzato i file come ovali e le directory come rettangoli:

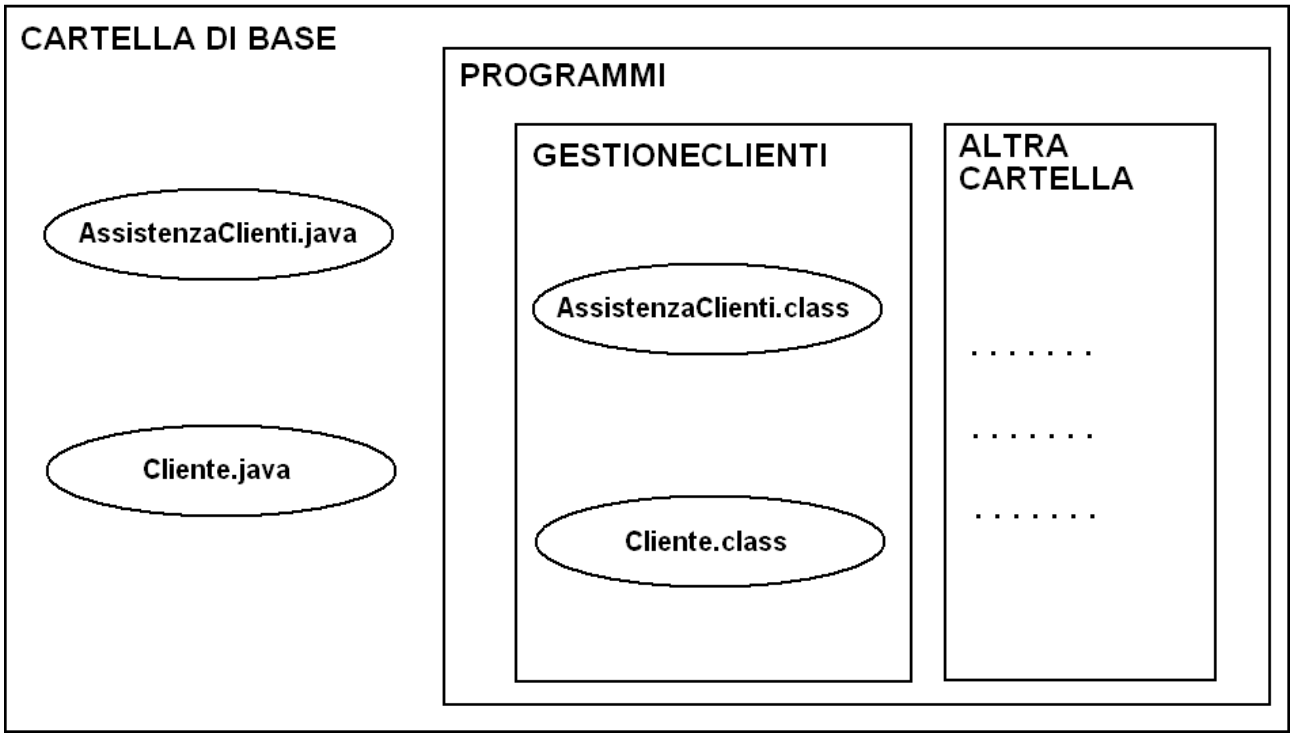


Figura 9.1: gestione dei package

Il Java Development Kit ci permette comunque di realizzare il giusto inserimento dei file nelle cartelle e la relativa creazione automatica delle cartelle stesse, mediante il comando:

```
javac –d . AssistenzaClienti.java
```

Lanciando il suddetto comando dalla cartella di base (dove è situato il file "AssistenzaClienti.java"), verranno automaticamente create le cartelle "programmi" e "gestioneClienti", ed il file "AssistenzaClienti.class" collocato nella cartella giusta, senza però spostare il file sorgente dalla cartella di base. A questo punto, supponendo che il file in questione contenga il metodo `main()`, potremmo mandarlo in esecuzione solo posizionandoci nella cartella di base e lanciando il comando:

```
java programmi.gestioneClienti.AssistenzaClienti
```

infatti, a questo punto, il file è associato ad un package e non potrà essere più chiamato spostandosi nella cartella dove esso è presente utilizzando il comando:

```
java AssistenzaClienti
```

Inoltre bisogna utilizzare un comando di `import` per utilizzare tale classe da un package differente.

Esistono altri problemi collegati all'utilizzo di package che l'aspirante programmatore potrà incontrare e risolvere con l'aiuto dei messaggi di errore ricevuti dal compilatore. Per esempio potrebbe risultare problematico utilizzare classi appartenenti ad un package, da una cartella non direttamente legata al package stesso. Ciò è risolvibile mediante l'impostazione della variabile di ambiente "classpath" del sistema operativo, che dovrebbe puntare alla cartella di base relativa al package in questione.

9.3.1 Classpath

La variabile d'ambiente CLASSPATH viene utilizzata al runtime dalla virtual machine per trovare le classi che il programma vuole utilizzare. Supponiamo di posizionare la nostra applicazione all'interno della cartella "C:\NostraApplicazione". Ora supponiamo che la nostra applicazione voglia utilizzare alcune classi situate all'interno di un'altra cartella, diciamo "C:\AltreClassiDaUtilizzare". Ovviamente la virtual machine non esplorerà tutte le cartelle del disco fisso per trovare le classi che gli servono, bensì cercherà solo nelle cartelle indicate dalla variabile d'ambiente CLASSPATH. In questo caso, quindi, è necessario impostare il classpath sia sulla cartella "C:\AltreClassiDaUtilizzare", sia sulla cartella "C:\NostraApplicazione", per poter lanciare correttamente la nostra applicazione.

1. Per raggiungere l'obiettivo ci sono due soluzioni:Impostare la variabile d'ambiente CLASSPATH direttamente da sistema operativo.
2. Impostare CLASSPATH solo per la nostra applicazione.

La prima soluzione è altamente sconsigliata. Si tratta di impostare in maniera permanente una variabile d'ambiente del sistema operativo (il procedimento è identico a quello descritto nell'appendice B per la variabile PATH). Questo implicherebbe l'impostazione permanente di questa variabile che impedirà il lancio di un'applicazione da un'altra cartella diversa da quelle referenziate dal classpath.

La seconda soluzione invece è più flessibile e si implementa specificando il valore della variabile CLASSPATH con una opzione del comando "java". Infatti il flag "–classpath" (oppure "–cp") del comando "java" permette di specificare proprio il classpath, ma solo per la durata dell'applicazione che stiamo lanciando. Per esempio, se ci troviamo nella cartella "C:\NostraApplicazione" e vogliamo lanciare la nostra applicazione, dovremo lanciare il seguente comando:

```
java –cp .; C:\AltreClassiDaUtilizzare miopackage.MiaClasseConMain
```

dove con l'opzione -cp abbiamo specificato che il classpath deve puntare alla directory corrente (specificata con ".") e alla directory "C:\AltreClassiDaUtilizzare". Ovviamente è anche possibile utilizzare percorsi relativi. Il seguente comando è equivalente al precedente:

```
java –cp .; ..\AltreClassiDaUtilizzare
miopackage.MiaClasseConMainSi noti che abbiamo usato come
separatore il ";", dato che stiamo supponendo di trovarci su un sistema
Windows. Su un sistema Unix il separatore sarebbe stato ":".
```

9.3.2 File JAR

Non è raro referenziare dalla propria applicazione librerie di classi esterne. È ovviamente possibile distribuire librerie di classi all'interno di cartelle, ma il formato più comune con cui vengono create librerie di classi è il formato JAR. Il termine JAR sta per Java ARchive (archivio java). Si tratta di un formato assolutamente equivalente al classico formato ZIP. L'unica differenza tra un formato ZIP e uno JAR è che un file JAR deve contenere una cartella chiamata META-INF, con all'interno una file di testo chiamato MANIFEST.MF. Questo file può essere utilizzato per aggiungere proprietà all'archivio JAR in cui è contenuto. Per creare un file JAR è quindi possibile utilizzare un utility come WinZIP, per poi aggiungere anche il file MANIFEST.MF in una cartella META-INF. Il JDK però offre un'alternativa più comoda: il comando jar. Con il seguente comando:

```
jar cvf libreria.jar MiaCartella
```

creeremo un file chiamato "libreria.jar" con all'interno la cartella "MiaCartella" e tutto il suo contenuto.

Sul sistema operativo Windows è possibile creare file "jar eseguibili". Questo significa che, una volta creato il file jar con all'interno la nostra applicazione, sarà possibile avviarla con un tipico doppio clic del mouse, così come se fosse un file .exe. Il discorso verrà approfondito nel Modulo 15.

9.3.3 Classpath e file JAR

Se volessimo utilizzare dalla nostra applicazione classi contenute all'interno di un file JAR, non direttamente disponibile nella stessa cartella dove è posizionata l'applicazione, è sempre possibile utilizzare il classpath. In tal caso, per lanciare la nostra applicazione, dovremo utilizzare un comando come:

```
java –cp .; C:\cartellaConFIleJAR\MioFile.jar miopackage.MiaClasseConMain
```

In pratica la JVM si occuperà di recuperare i file .class all'interno del file JAR, in maniera automatica ed ottimizzata.
Se volessimo utilizzare più file JAR, dovremo lanciare un comando come:

```
java –cp .; C:\cartellaConFileJAR\MioFile.jar;
C:\cartellaConFIleJAR\AltroMioFile.jar miopackage.MiaClasseConMain
```

Se non si vuole utilizzare una libreria JAR senza impostare il classpath è possibile inserire il file JAR nella cartella jre/lib/ext che si trova all'interno dell'installazione del JDK. Ovviamente, tutte le applicazioni che verranno lanciate tramite il JDK potranno accedere a tale libreria. Per quanto riguarda EJE, dalla versione 2.7 è integrato un gestore di classpath.

9.3.4 Gestione "a mano"

Sebbene semplice da spiegare, la gestione dei package può risultare difficoltosa e soprattutto noiosa se "fatta a mano". Se il lettore è arrivato sino a questo punto utilizzando uno strumento come il Blocco Note per esercitarsi, vuol dire che è pronto per passare ad un editor più sofisticato. Un qualsiasi tool Java gestisce l'organizzazione dei package in maniera automatica…

Attualmente EJE (versione 2.7) permette di specificare la directory di output dove devono essere generate le classi. Di default però EJE compila le classi mella stessa directory di dove si trovano i sorgenti. Ci sono casi in cui quindi per poter compilare correttamente i vostri file bisognerà compilare tutti i file che compongono la vostra applicazione. Non si dovrebbero presentare problemi però se si specifica come output directory una cartela diversa da quella di default.

Se il lettore vuole continuare con il Blocco Note (anche se sospettiamo parecchie diserzioni…) ed utilizzare i package, prenda in considerazione il seguente consiglio:
Quando si inizia un progetto, è opportuno creare una cartella con il nome del progetto stesso. Per esempio se il progetto si chiama `Bancomat`, chiamare la cartella "Bancomat".
Poi vanno create almeno due sottocartelle, chiamate "src" (dove metteremo tutti i sorgenti) e "classes" (dove metteremo tutti i file compilati). In questo modo divideremo l'ambiente di sviluppo dall'ambiente di distribuzione. Opzionalmente, consigliamo di creare altre cartelle parallele come "docs" (dove mettere tutta la documentazione), "config" (dove mettere i file di configurazione) etc… Per esempio, nella cartella "Bancomat" inserire le cartelle "src", "classes" etc…
Ogni volta che una classe deve appartenere ad un package, creare la cartella package a mano ed inserire il file sorgente all'interno. Per esempio, se la classe `ContoCorrente` deve appartenere al package `banca`, creare la cartella "banca" all'interno della cartella "src" e inserire il file "ContoCorrente.java" nella cartella "banca".
Ogni volta che dobbiamo compilare, bisogna posizionarsi tramite il prompt di DOS nella cartella "classes" (sempre) e da qui lanciare un comando del tipo:

```
javac –d . ..\src\nomepackage\*.java
```

Se volessimo compilare tutte le classi del package `banca`, dovremmo lanciare il seguente comando:

```
javac –d . ..\src\banca\*.java
```

Consigliamo al lettore di crearsi file batch (.bat) per immagazzinare comandi così lunghi o ancora meglio utilizzare un tool come Ant (per informazioni consultare http://ant.apache.org).

Questo tipo di organizzazione è consigliato dalla stessa Sun ed è implementato automaticamente dai tool più importanti (nonché in futuro anche da EJE).

9.4 Il modificatore final

Questo semplice modificatore ha un'importanza fondamentale. È applicabile sia a variabili, sia a metodi, sia a classi. Potremmo tradurre il termine `final` proprio con "finale", nel senso di "non modificabile". Infatti:

- una variabile dichiarata `final` diviene una costante
- un metodo dichiarato `final` non può essere riscritto in una sottoclasse (non è possibile applicare l'override)
- una classe dichiarata `final` non può essere estesa

Il modificatore `final` si può utilizzare anche per variabili locali e parametri locali di metodi. In tali casi, ovviamente, i valori di tali variabili non saranno modificabili localmente. Per esempio, il seguente codice viene compilato senza errori:

```
public class LocalVariables {
  public static void main(String args[]) {
    System.out.println(new
      LocalVariables().finalLocalVariablesMethod(5,6));
  }
  public int finalLocalVariablesMethod(final int i,
      final int j) {
    final int k = i + j;
    return k;
  }
}
```

In particolare, nel modulo precedente, abbiamo visto come le classi interne e le classi anonime si possano dichiarare anche all'interno di metodi. In tali casi queste possono accedere solo alle variabili locali e ai parametri dichiarati final. Supponendo che sia stata definita la classe `MyClass`, il seguente codice è valido:

```
public void methodWithClass(final int a) {
```

```
  new MyClass(){
    public void myMethod(){
      System.out.println(a);
    }
  };
}
```

Come volevasi dimostrare, esistono casi dove anche dichiarare un parametro di un metodo `final` ha senso.

9.5 Il modificatore static

`static` è forse il più potente modificatore di Java. Forse anche troppo! Con `static` la programmazione ad oggetti trova un punto di incontro con quella strutturata ed il suo uso deve essere quindi limitato a situazioni di reale e concreta utilità. Potremmo tradurre il termine `static` con "condiviso da tutte le istanze della classe", oppure "della classe". Per quanto detto, un membro statico ha la caratteristica di poter essere utilizzato mediante una sintassi del tipo:

```
NomeClasse.nomeMembro
```

in luogo di:

```
nomeOggetto.nomeMembro
```

Anche senza istanziare la classe, l'utilizzo di un membro statico provocherà il caricamento in memoria della classe contenente il membro in questione, che quindi, condividerà il ciclo di vita con quello della classe.

9.5.1 Metodi statici

Un esempio di metodo statico è il metodo `sqrt()` della classe `Math`, che viene chiamato tramite la sintassi:

```
Math.sqrt(numero)
```

`Math` è quindi il nome della classe e non il nome di un'istanza di quella classe. La ragione per cui la classe `Math` dichiara tutti i suoi metodi statici è facilmente

comprensibile. Infatti, se istanziassimo due oggetti differenti dalla classe `Math`, `ogg1` e `ogg2`, i due comandi:

```
ogg1.sqrt(4);
```

e

```
ogg2.sqrt(4);
```

produrrebbero esattamente lo stesso risultato (`2`). Effettivamente non ha senso istanziare due oggetti di tipo matematica, che come si sa è unica.

> **Un metodo dichiarato `static` e `public` può considerarsi una sorta di funzione e perciò questo tipo di approccio ai metodi dovrebbe essere accuratamente evitato.**

9.5.2 Variabili statiche (di classe)

Una variabile statica, essendo condivisa da tutte le istanze della classe, assumerà lo stesso valore per ogni oggetto di una classe.
Di seguito viene presentato un esempio:

```
public class ClasseDiEsempio
{
    public static int a = 0;
}
public class ClasseDiEsempioPrincipale
{
    public static void main (String args[])
    {
        System.out.println("a = "+ClasseDiEsempio.a);
        ClasseDiEsempio ogg1 = new ClasseDiEsempio();
        ClasseDiEsempio ogg2 = new ClasseDiEsempio();
        ogg1.a = 10;
        System.out.println("ogg1.a = " + ogg1.a);
        System.out.println("ogg2.a = " + ogg2.a);
        ogg2.a=20;
        System.out.println("ogg1.a = " + ogg1.a);
```

```
        System.out.println("ogg2.a = " + ogg2.a);
    }
}
```

L'output di questo semplice programma sarà:

```
a = 0
ogg1.a = 10
ogg2.a = 10
ogg1.a = 20
ogg2.a = 20
```

Come si può notare, se un'istanza modifica la variabile statica, essa risulterà modificata anche relativamente all'altra istanza. Infatti essa è condivisa dalle due istanze ed in realtà risiede nella classe. Una variabile di questo tipo potrebbe ad esempio essere utile per contare il numero di oggetti istanziati da una classe (per esempio incrementandola in un costruttore). Per esempio:

```
public class Counter {
  private static int counter = 0;
  private int number;
  public Counter() {
    counter++;
    setNumber(counter);
  }
  public void setNumber(int number) {
    this.number = number;
  }
  public int getNumber() {
    return number;
  }
}
```

Si noti come la variabile d'istanza `number` venga valorizzata nel costruttore di questa classe con il valore della variabile statica `counter`. Questo significa che `number` rappresenterà proprio il numero seriale di ogni oggetto istanziato. Per esempio, dopo questa istruzione:

```
Counter c1 = new Counter();
```

La variabile statica `counter` varrà `1` e la variabile `c1.number` varrà sempre `1`.
Se poi istanziamo un altro oggetto `Counter`:

```
Counter c2 = new Counter();
```

allora la variabile statica `counter` varrà `2`. Infatti, essendo condivisa da tutte le istanze della classe, non viene riazzerata ad ogni istanza. Invece la variabile `c1.number` varrà sempre `1`, mentre la variabile `c2.number` varrà `2`.
Il modificatore `static` quindi prescinde dal concetto di oggetto e lega strettamente le variabili al concetto di classe, che a sua volta si innalza a qualcosa più di un semplice mezzo per definire oggetti. Per questo motivo a volte ci si riferisce alle variabili d'istanza statiche come "variabili di classe".

Una variabile dichiarata `static` e `public` può considerarsi una sorta di variabile globale e perciò questo tipo di approccio alle variabili va accuratamente evitato. È però utile a volte utilizzare costanti globali, definite con `public`, `static` e `final`. Infatti la classe `Math` definisce due costanti statiche pubbliche: `PI` ed `E` (cfr. documentazione ufficiale).
Notiamo anche che una variabile statica non viene inizializzata al valore nullo del suo tipo al momento dell'istanza dell'oggetto. Non è una variabile d'istanza.

Un metodo statico non può ovviamente utilizzare variabili d'istanza senza referenziarle, ma solo variabili statiche (ed ovviamente variabili locali). Infatti, un metodo statico non appartiene a nessuna istanza in particolare, e quindi non potrebbe "scegliere" una variabile d'istanza di una istanza particolare senza referenziarla esplicitamente. Per esempio la seguente classe:

```
public class StaticMethod {
    private int variabileDiIstanza;
    private static int variabileDiClasse;
    public static void main(String args[]) {
        System.out.println(variabileDiIstanza);
    }
```

```
}
```

produrrà il seguente errore in compilazione:

```
non-static variable variabileDiIstanza cannot be referenced from a static context
        System.out.println(variabileDiIstanza);
                           ^
1 error
```

Ovviamente, se stampassimo la variabile `variabileDiClasse`, il problema non si porrebbe.

9.5.3 Inizializzatori statici (e d'istanza)

Il modificatore `static` può anche essere utilizzato per marcare un semplice blocco di codice, che viene a sua volta ribattezzato **inizializzatore statico**. Anche questo blocco, come nel caso dei metodi statici, potrà utilizzare variabili definite fuori da esso se e solo se dichiarate statiche.
In pratica un blocco statico definito all'interno di una classe avrà la caratteristica di essere chiamato al momento del caricamento in memoria della classe stessa, addirittura prima di un eventuale costruttore. La sintassi è semplice e di seguito è presentato un semplice esempio:

```
public class EsempioStatico
{
    private static int a = 10;
    public EsempioStatico()
    {
        a += 10;
    }
    static
    {
        System.out.println("valore statico = " + a);
    }
}
```

Supponiamo di istanziare un oggetto di questa classe, mediante la seguente sintassi:

```
EsempioStatico ogg = new EsempioStatico();
```

Questo frammento di codice produrrà il seguente output:

```
valore statico = 10
```

Infatti, quando si istanzia un oggetto da una classe, questa deve essere prima caricata in memoria. È in questa fase di caricamento che viene eseguito il blocco statico, e di conseguenza stampato il messaggio di output. Successivamente viene chiamato il costruttore che incrementerà il valore statico.

E' possibile inserire in una classe anche più di un inizializzatore statico. Ovviamente, questi verranno eseguiti in maniera sequenziale "dall'alto in basso".

L'uso di un inizializzatore statico può effettivamente essere considerato sporadico. Ma per esempio è alla base della possibilità di rendere codice Java indipendente dal database a cui si interfaccia tramite JDBC (cfr. Modulo 14). Concludendo, l'utilizzo di questo modificatore dovrebbe quindi essere legato a soluzioni di progettazione avanzate. Esiste anche un'altra tipologia di inizializzatore, ma non statico. Si chiama **inizializzatore d'istanza** (**instance initializer** o **object initializer**) e si implementa includendo codice in un blocco di parentesi graffe all'interno di una classe. La sua caratteristica è l'essere eseguito quando viene istanziato un oggetto, prima del costruttore. Per esempio, se istanziassimo la seguente classe:

```
public class InstanceInitializer {
    public InstanceInitializer() {
        System.out.println("Costruttore");
    }
    {
        System.out.println("Inizializzatore");
    }
}
```

l'output risultante sarebbe il seguente:

```
Inizializzatore
Costruttore
```

Ovviamente i casi in cui risulta necessario utilizzare un inizializzatore d'istanza sono molto rari. Un esempio potrebbe essere l'inizializzazione delle classi anonime dove non esistono i costruttori. Non è possibile però passare parametri ad un inizializzatore, come si potrebbe altrimenti fare con un costruttore.

E' possibile inserire nella stessa classe inizializzatori statici e d'istanza. Ovviamente gli inizializzatori statici avranno sempre la precedenza.

9.5.4 Static import

Solo dalla versione 5 in poi è possibile utilizzare anche i cosiddetti "**import statici**". Con il comando `import` siamo soliti importare nei nostri file classi ed interfacce direttamente da package esterni. In pratica, possiamo importare all'interno dei nostri file i nomi delle classi e delle interfacce, in modo tale che si possano poi usare senza dover specificare l'intero "fully qualified name" (nome completo di package). Per esempio, volendo utilizzare la classe `DOMSource` del package `javax.xml.transform.dom`, abbiamo due possibilità:

1. Importiamo la classe con un'istruzione come la seguente:

   ```
   import javax.xml.transform.dom.DOMSource;
   ```

   per poi sfruttarla nel nostro file sorgente utilizzandone solo il nome. Per esempio:

   ```
   DOMSource source = new DOMSource();
   ```

2. Oppure possiamo scrivere il nome completo di package ogni volta che la utilizziamo. Per esempio:

   ```
   javax.xml.transform.dom.DOMSource source = new
         javax.xml.transform.dom.DOMSource();
   ```

Ovviamente la prima soluzione è conveniente.
In alcuni casi, però, potremmo desiderare di importare nel file solo ciò che è dichiarato statico all'interno di una certa classe, e non la classe stessa. Per esempio, sapendo che la classe `Math` contiene solo metodi e costanti statiche, potremmo voler importare,

piuttosto che la classe `Math`, solo i suoi membri statici. Segue la sintassi per realizzare quanto detto:

```
import static java.lang.Math.*;
```

In questo caso abbiamo importato tutti i membri statici all'interno del file. Quindi sarà lecito scrivere, in luogo di:

```
double d = Math.sqrt(4);
```

direttamente:

```
double d = sqrt(4);
```

senza anteporre il nome della classe al metodo statico.
Ovviamente è anche possibile importare staticamente solo alcuni membri specifici, per esempio:

```
import static java.lang.Math.PI;
import static java.lang.Math.random;
import static java.sql.DriverManager.getConnection;
import static java.lang.System.out;
```

Si noti come l'`import` di nomi di metodi statici non specifichi le parentesi con i relativi argomenti.
Tutte le considerazioni e i consigli di utilizzo di questa nuova caratteristica di Java sono trattati approfonditamente nell'Unità Didattica 18.2.

9.6 Il modificatore `abstract`

Il modificatore `abstract` può essere applicato a classi e metodi.

9.6.1 Metodi astratti

Un metodo astratto non implementa un proprio blocco di codice e quindi il suo comportamento. In pratica, un metodo astratto non definisce parentesi graffe, ma termina con un punto e virgola. Un esempio di metodo astratto potrebbe essere il seguente:

```
public abstract void dipingiQuadro();
```

Ovviamente, questo metodo non potrà essere chiamato, ma potrà essere soggetto a riscrittura (override) in una sottoclasse.
Inoltre, un metodo astratto potrà essere definito solamente all'interno di una classe astratta. In altre parole, una classe che contiene anche un solo metodo astratto deve essere dichiarata astratta.

9.6.2 Classi astratte

Una classe dichiarata astratta non può essere istanziata. Il programmatore che ha intenzione di marcare una classe con il modificatore `abstract` deve essere consapevole a priori che da quella classe non saranno istanziabili oggetti. Consideriamo la seguente classe astratta:

```
public abstract class Pittore
{
   . . .
   public abstract void dipingiQuadro();
   . . .
}
```

Questa classe ha senso se inserita in un sistema in cui l'oggetto `Pittore` può essere considerato troppo generico per definire un nuovo tipo di dato da istanziare. Supponiamo che per il nostro sistema sia fondamentale conoscere lo stile pittorico di un oggetto `Pittore` e, siccome non esistono pittori capaci di dipingere con un qualsiasi tipo di stile tra tutti quelli esistenti, non ha senso istanziare una classe `Pittore`. Sarebbe corretto popolare il nostro sistema di sottoclassi non astratte di `Pittore` come `PittoreImpressionista` e `PittoreNeoRealista`. Queste sottoclassi devono ridefinire il metodo astratto `dipingiQuadro` (a meno che non si abbia l'intenzione di dichiarare astratte anche esse) e tutti gli altri eventuali metodi astratti ereditati.
Effettivamente, la classe `Pittore` deve dichiarare un metodo `dipingiQuadro`, ma a livello logico non sarebbe giusto definirlo favorendo uno stile piuttosto che un altro. Nulla vieta però ad una classe astratta di implementare tutti suoi metodi.

> **Il modificatore `abstract`, per quanto riguarda le classi, potrebbe essere considerato l'opposto del modificatore `final`. Infatti, una classe `final` non può essere estesa, mentre una classe dichiarata**

`abstract` può esserlo. Non è ovviamente possibile utilizzare congiuntamente i modificatori `abstract` e `final`, per chiari motivi logici.
Non è altrettanto possibile utilizzare congiuntamente i modificatori `abstract` e `static`.

Il grande vantaggio che offre l'implementazione di una classe astratta è che "obbliga" le sue sottoclassi ad implementare un comportamento. A livello di progettazione, le classi astratte costituiscono uno strumento fondamentale. Spesso è utile creare classi astratte tramite la generalizzazione, in modo da sfruttare il polimorfismo.
Nel modulo 6, dove abbiamo presentato i vari aspetti del polimorfismo, avevamo presentato un esempio in cui una classe `Veicolo` veniva estesa da varie sottoclassi (`Auto`, `Nave`…), che reimplementavano i metodi ereditati (`accelera()`, e `decelera()`) in maniera adatta al contesto.
La classe `Veicolo` era stata presentata nel seguente modo:

```
public class Veicolo
{
    public void accelera()
    {
        . . .
    }
    public void decelera()
    {
        . . .
    }
}
```

e forse il lettore più riflessivo si sarà anche chiesto cosa avrebbe mai potuto dichiarare all'interno di questi metodi.
Effettivamente, come accelera un veicolo? Come una nave o come un aereo?
La risposta più giusta è che non c'è risposta! E' indefinita. Un veicolo sicuramente può accelerare e decelerare ma, in questo contesto, non si può dire come. La soluzione più giusta appare dichiarare i metodi astratti:

```
public abstract class Veicolo
{
    public abstract void accelera();
```

```
    public abstract void decelera() ;
}
```

Il polimorfismo continua a funzionare in tutte le sue manifestazioni (cfr. Modulo 6).

9.7 Interfacce

Un'interfaccia è un'evoluzione del concetto di classe astratta. Per definizione, un'interfaccia possiede tutti i metodi dichiarati `public` e `abstract` e tutte le variabili dichiarate `public`, `static` e `final`.
Un'interfaccia, per essere utilizzata, ha bisogno di essere in qualche modo estesa, non potendo ovviamente essere istanziata. Si può fare qualcosa di più rispetto al semplice estendere un'interfaccia: la si può implementare. L'implementazione di un'interfaccia da parte di una classe consiste nell'ereditare tutti i metodi e fornire loro un blocco di codice e si realizza posponendo alla dichiarazione della classe la parola chiave `implements`, seguita dall'identificatore dell'interfaccia che si desidera implementare. La differenza tra l'implementare un'interfaccia ed estenderla consiste essenzialmente nel fatto che, mentre possiamo estendere una sola classe alla volta, possiamo invece implementare infinite interfacce, simulando di fatto l'ereditarietà multipla, ma senza i suoi effetti collaterali negativi.
Di seguito è presentato un esempio per familiarizzare con la sintassi:

```
public interface Saluto
{
    public static final String CIAO = "Ciao";
    public static final String BUONGIORNO = "Buongiorno";
    . . .
    public abstract void saluta();
}
```

Possiamo notare che siccome un'interfaccia non può dichiarare metodi non astratti, non c'è bisogno di marcarli con `abstract` (e con `public`); è scontato che lo siano. Stesso discorso per i modificatori delle costanti. Quindi la seguente interfaccia è assolutamente equivalente alla precedente:

```
public interface Saluto
{
```

```
    String CIAO = "Ciao";
    String BUONGIORNO = "Buongiorno";
    . . .
    void saluta();
}
```

Come le classi, le interfacce si devono scrivere all'interno di file che hanno esattamente lo stesso nome dell'interfaccia che definiscono, ed ovviamente con suffisso ".java". Quindi l'interfaccia `Saluto` dell'esempio deve essere salvata all'interno di un file di nome "Saluto.java".

Un'interfaccia, essendo un'evoluzione di una classe astratta, non può essere istanziata. Potremmo utilizzare l'interfaccia dell'esempio nel seguente modo:

```
public class SalutoImpl implements Saluto
{
    public void saluta()
    {
        System.out.println(CIAO);
    }
}
```

Una classe può implementare un'interfaccia. Un'interfaccia può estendere un'altra interfaccia.

9.7.1 Regole di conversione dei tipi

A proposito di ereditarietà, ora che conosciamo la definizione di interfaccia, possiamo anche definire le regole che sussistono per le conversioni dei tipi che fino ad ora abbiamo incontrato (ovvero le assegnazioni di reference delle superclassi che puntano a reference delle sottoclassi; cfr. Modulo 6 sul polimorfismo per dati):
Un'interfaccia può essere convertita ad un'interfaccia o ad `Object`. Se il nuovo tipo è un'interfaccia, questa deve essere estesa dal vecchio tipo. Per esempio, se `A` e `B` sono due interfacce, e `C` una classe che implementa `B`, è legale scrivere:

```
B b = new C();
A a = b;
```

se e solo se l'interfaccia `B` estende `A`.
Una classe può essere convertita ad un'altra classe o ad un'interfaccia. Se il nuovo tipo è una classe, il vecchio tipo deve ovviamente estendere il nuovo tipo. Se il nuovo tipo è un'interfaccia allora la vecchia classe deve implementare l'interfaccia.
Un array può essere convertito solo ad `Object`, o ad un'interfaccia `Cloneable` o `Serializable` (che evidentemente in qualche modo sono implementate dagli array) o ad un altro array.
Solo un array di tipi complessi può essere convertito ad un array, ed inoltre il vecchio tipo complesso deve essere convertibile al nuovo tipo. Per esempio è legale scrivere:

```
Auto[] auto = new Auto[100];
Veicolo[] veicoli = auto;
```

9.7.2 Ereditarietà multipla

Le interfacce sono il mezzo tramite il quale Java riesce a simulare l'ereditarietà multipla. L'implementazione di un'interfaccia da parte di una classe, richiede come condizione imprescindibile, l'implementazione di ogni metodo astratto ereditato (altrimenti bisognerà dichiarare la classe astratta). L'implementazione dell'ereditarietà multipla da parte di una classe, si realizza posponendo alla dichiarazione della classe la parola chiave `implements`, seguita dagli identificatori delle interfacce che si desiderano implementare, separati da virgole. Il lettore potrebbe non recepire immediatamente l'utilità e la modalità d'utilizzo delle interfacce quindi, di seguito, è presentato un significativo esempio forzatamente incompleto, e piuttosto complesso se si valutano le informazioni finora fornite:

```
public class MiaApplet extends Applet implements
MouseListener, Runnable
{
     . . .
}
```

Come il lettore avrà probabilmente intuito, abbiamo definito un'applet. In Java, un'applet è una qualsiasi classe che estende la classe `Applet`, e che quindi non può estendere altre classi. Nell'esempio abbiamo implementato due interfacce che contengono metodi particolari che ci permetteranno di gestire determinate situazioni. In particolare l'interfaccia `MouseListener`, contiene metodi che gestiscono gli eventi che hanno come generatore il mouse. Quindi, se implementiamo il metodo astratto

`mousePressed()`, ereditato dall'interfaccia, definiremo il comportamento di quel metodo, che gestirà gli eventi generati dalla pressione del pulsante del mouse. Notare che implementiamo anche un'altra interfaccia chiamata `Runnable`, tramite i cui metodi possiamo gestire eventuali comportamenti di processi paralleli che Java ha la possibilità di utilizzare con il multi-threading. In pratica, in questo esempio, con la sola dichiarazione stiamo inizializzando una classe non solo ad essere un'applet, ma anche a gestire eventi generati dal mouse, ed un eventuale gestione parallela dei thread (per la definizione di thread cfr. Modulo 11).

9.7.3 Differenze tra interfacce e classi astratte

Il vantaggio che offrono sia le classi astratte che le interfacce, risiede nel fatto che esse possono "obbligare" le sottoclassi ad implementare dei comportamenti. Una classe che eredita un metodo astratto infatti, deve fare override del metodo ereditato oppure essere dichiarata astratta. Dal punto di vista della progettazione quindi, questi strumenti supportano l'astrazione dei dati. Per esempio ogni veicolo accelera e decelera e quindi tutte le classi non astratte che estendono `Veicolo`, devono ri-scrivere i metodi `accelera()` e `decelera()`.
Una evidente differenza pratica è che possiamo simulare l'ereditarietà multipla, solo con l'utilizzo di interfacce. Infatti, è possibile estendere una sola classe alla volta, ma implementare più interfacce.
Tecnicamente la differenza più evidente che esiste tra una classe astratta ed una interfaccia e che quest'ultima non può dichiarare né variabili ne metodi concreti, ma solo costanti statiche e pubbliche e metodi astratti. È invece possibile dichiarare in maniera concreta un'intera classe astratta (senza metodi astratti). In quel caso il dichiararla astratta implica comunque che non possa essere istanziata.
Quindi una classe astratta solitamente non è altro che un'astrazione, che è troppo generica per essere istanziata nel contesto in cui si dichiara. Un buon esempio è la classe `Veicolo`.
Un'interfaccia invece, solitamente non è una vera astrazione troppo generica per il contesto, ma semmai un'"astrazione comportamentale", che non ha senso istanziare in un certo contesto.
Considerando sempre l'esempio del `Veicolo` superclasse di `Aereo` del modulo 6, possiamo introdurre un'interfaccia `Volante` (notare che il nome fa pensare ad un comportamento più che ad un oggetto astratto), che verrà implementata dalle classi che volano. Ora se l'interfaccia `Volante` è definita nel seguente modo:

```
public interface Volante
{
```

```
    void atterra();
    void decolla();
}
```

ogni classe che deve astrarre un concetto di "oggetto volante" (come un aereo o un elicottero) deve implementare l'interfaccia. Riscriviamo quindi la classe `Aereo` nel seguente modo:

```
public class Aereo extends Veicolo implements Volante
{
    public void atterra()
    {
         // override del metodo di Volante
    }
    public void decolla()
    {
        // override del metodo di Volante
    }
    public void accelera()
    {
        // override del metodo di Veicolo
    }
    public void decelera()
    {
        // override del metodo di Veicolo
    }
}
```

Qual è il vantaggio di tale scelta? Ovvia risposta: possibilità di utilizzo del polimorfismo. Infatti, sarà legale scrivere:

```
Volante volante = new Aereo();
```

oltre ovviamente a

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

e quindi si potrebbero sfruttare parametri polimorfi, collezioni eterogenee e invocazioni di metodi virtuali, in situazioni diverse. Potremmo anche fare implementare alla classe `Aereo` altre interfacce…

9.8 Tipi Enumerazioni

Una delle più importanti novità introdotte nella versione 5 del linguaggio sono le **Enumerazioni**, concetto già noto ai programmatori di altri linguaggi come il C e il Pascal. Si tratta di una nuova struttura dati che si va ad aggiungere alle classi, alle interfacce e alle annotazioni (che approfondiremo nel Modulo 19). Per poter introdurre le enumerazioni in Java, si dovuta definire una nuova parola chiave: `enum`. Quindi non sarà più possibile utilizzare `enum` come identificatore da ora in poi.
Le enumerazioni sono delle strutture dati molto particolari, somigliano alle classi, ma hanno delle proprietà particolari. Facciamo subito un esempio:

```
public enum MiaEnumerazione {
    UNO, DUE, TRE;
}
```

In pratica abbiamo definito un'enum di nome `MiaEnumerazione`, definendo tre suoi elementi che si chiamano `UNO`, `DUE`, `TRE`. Notare che il "`;`" in questo caso non è necessario ma consigliato. Diventerà infatti necessario, nel caso vengano definiti altri elementi nell'enumerazione, come per esempio metodi.
Un'enum non si può istanziare come una classe. Invece, le uniche istanze che esistono, sono proprio i suoi elementi, che vengono definiti insieme all'enumerazione stessa. Questo significa che nell'esempio `UNO`, `DUE` e `TRE` sono le (uniche) istanze di `MiaEnumerazione`, e quindi sono di tipo `MiaEnumerazione`.
Tutte le istanze di un'enumerazione sono implicitamente dichiarate `public`, `static` e `final`. Questo spiega anche il perché viene utilizzata la convenzione per i nomi delle costanti per gli elementi di un'enum.

9.8.1 Ereditarietà ed enum

Un'enum, non si può estendere, né può estendere un'altra enum o un'altra classe. Infatti le enumerazioni verranno trasformate in classi che estendono la classe `java.lang.Enum` dal compilatore. Questo significa che erediteremo dalla classe `java.lang.Enum`, diversi metodi, che possiamo tranquillamente invocare sugli elementi dell'enum. Per esempio il metodo `toString()`, è definito in modo tale da restituire il nome dell'elemento, quindi:

```
System.out.println(MiaEnumerazione.UNO);
```

produrrà il seguente output:

```
UNO
```

È invece possibile far implementare un'interfaccia ad un'enum. Infatti, possiamo dichiarare in un'enum tutti i metodi che vogliamo, compresi quelli da implementare di un'interfaccia. Ovviamente non potendo estendere un'enum, non si potrà dichiarare `abstract`. Quindi, nel momento in cui implementiamo un'interfaccia in un'enum, dovremo implementare obbligatoriamente tutti i metodi ereditati. Ovviamente, possiamo anche fare override dei metodi di `java.lang.Enum`.

9.8.2 Costruttori ed enum

E' possibile anche creare costruttori in una enum. Questi però sono implicitamente dichiarati `private`, e non è possibile utilizzarli se non nell'ambito dell'enumerazione stessa. Per esempio, con il seguente codice viene ridefinita l'enum `MiaEnumerazione`, in modo tale da definire un costruttore con un parametro intero, una variabile privata con relativi metodi setter e getter, e con un override del metodo `toString()`:

```
public enum MiaEnumerazione {
    UNO(1), DUE(2), TRE(3);
    private int valore;
    MiaEnumerazione(int valore) {
      setValore(valore);
    }
    public void setValore(int valore){
      this.valore = valore;
    }
    public int getValore(){
      return this.valore;
    }
    public String toString() {
      return ""+valore;
    }
}
```

Notare come il costruttore venga sfruttato quando vengono definiti gli elementi dell'enum.

Come per le classi, anche per le enumerazioni, se non inseriamo costruttori, il compilatore ne aggiungerà uno per noi senza parametri (il costruttore di default). E come per le classi, il costruttore di default non verrà inserito nel momento in cui ne inseriamo noi uno esplicitamente come nell'esempio precedente.

9.8.3 Quando utilizzare un'enum

È consigliato l'uso dell'enum, ogni qualvolta ci sia bisogno di dichiarare una numero finito di valori da utilizzare, per gestire il flusso di un'applicazione. Facciamo come al solito un esempio. Prima dell'avvento delle enumerazioni in Java, spesso si rendeva necessario creare delle costanti simboliche per rappresentare dei valori. Tali costanti venivano spesso definite di tipo intero oppure stringa. A volte si raccoglievano tali costanti all'interno di interfacce dedicate. Per esempio:

```
public interface Azione {
    public static final int AVANTI = 0;
    public static final int INDIETRO = 1;
    public static final int FERMO = 2;
}
```

Ci sono almeno due motivi per cui questo approccio è considerato sconsigliabile:
Il ruolo object oriented di un'interfaccia viene stravolto (cfr. paragrafo precedente)
A livello di compilazione, non si possono esplicitare vincoli che impediscano all'utente di utilizzare scorrettamente in tipo.
Per esempio consideriamo il seguente codice:

```
public void esegui(int azione) {
    switch (azione) {
        case AZIONE.AVANTI:
             vaiAvanti();
        break;
        case AZIONE.INDIETRO:
            vaiIndietro();
        break;
```

```
        case AZIONE.FERMO:
            fermati();
        break;
    }
}
```

nessun compilatore potrebbe rilevare che la seguente istruzione non è un'istruzione valida:

```
oggetto. esegui(3);
```

Ovviamente è possibile aggiungere una clausola `default` al costrutto `switch`, dove si gestisce in qualche modo il problema, ma questo sarà eseguito solo in fase di runtime dell'applicazione.
Un'implementazione più robusta della precedente, potrebbe richiedere l'utilizzo di una classe come la seguente:

```
public class Azione {
    private String nome;
    public static final Azione AVANTI = new
      Azione("AVANTI");
    public static final Azione INDIETRO = new
      Azione("INDIETRO");
    public static final Azione FERMO = new
      Azione("FERMO");
    public Azione(String nome){
        setNome(nome);
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getNome(){
        return this.nome;
    }
}
```

Ovviamente in tale caso il metodo `esegui()` dovrebbe essere modificato in modo tale da sostituire il costrutto `switch`, con un costrutto `if`:

```
public void esegui(Azione azione) {
    if (azione == Azione.AVANTI) {
        vaiAvanti();
    }
    else if (azione == Azione.INDIETRO) {
        vaiIndietro();
    }
    else if (azione == Azione.FERMO) {
        fermo();
    }
}
```

Purtroppo, anche in questo caso però, mentre sembra risolto il problema 1), il problema 2) rimane. Per esempio, è sempre possibile passare `null`, al metodo `esegui()`. Qualsiasi controllo volessimo inserire, potrebbe risolvere il problema solo in fase di runtime dell'applicazione.
In casi come questo, l'uso di un'enumerazione mette tutti d'accordo. Se infatti creiamo la seguente enumerazione:

```
public enum Azione {
    AVANTI, INDIETRO, FERMO;
}
```

e senza modificare il codice del metodo `esegui()` (infatti è possibile utilizzare un tipo enum anche come variabile di test di un costrutto `switch`), non avremo più i nessuno dei due lati negativi, definiti dai punti 1) e 2).
Esistono anche altre caratteristiche avanzate delle enumerazioni. Il lettore può approfondire lo studio delle enum, nell'Unità didattica 17.2.

Come le classi e le interfacce, le enumerazioni si devono scrivere all'interno di file che hanno esattamente lo stesso nome dell'enumerazione stessa, ed ovviamente con suffisso ".java". Quindi l'enum `Azione` dell'esempio, deve essere salvata all'interno di un file di nome "Azione.java".

9.9 Modificatori di uso raro: native, volatile e strictfp

I seguenti modificatori vengono utilizzati raramente. In particolare `strictfp` e `volatile` sono assolutamente sconosciuti anche alla maggior parte dei programmatori Java.

9.9.1 Il modificatore strictfp

Il modificatore `strictfp`, viene utilizzato rarissimamente. Questo serve per garantire nel caso in cui ci siano delle espressioni aritmetiche che coinvolgono valori esclusivamente `float` o `double`, non avvengano operazioni intermedie che producano overflow o underflow.
Marcando con `strictfp` una classe `X`, un'interfaccia (argomento di questo modulo) `Y`, o un metodo `Z`, allora qualsiasi classe, interfaccia, metodo, costruttore, variabile d'istanza, inizializzatore statico o d'istanza interno a `X`, `Y` o `Z`, godrà della proprietà di `strictfp`.

9.9.2 Il modificatore native

Il modificatore `native`, serve per marcare metodi che possono invocare funzioni "native". Questo significa che è possibile scrivere applicazioni Java, che invocano funzioni scritte in C/C++. Stiamo parlando di una delle tante tecnologie Java standard, nota come JNI, ovvero Java Native Interface.
Proponiamo come esempio la tipica applicazione che stampa la frase "Hello World", tratta direttamente dal Java Tutorial. In questo caso però, un programma Java utilizzerà una libreria scritta in C++, per chiamare una funzione che stampa la scritta "Hello World".

> **Se non si conosce il linguaggio C++, o almeno il C, si potrebbero avere delle difficoltà a comprendere il seguente esempio. Niente di grave comunque…**

Per prima cosa, bisogna creare il file sorgente Java, che dichiara un metodo nativo (che di default è anche astratto:

```
class HelloWorld {
    public native void displayHelloWorld();
    static {
        System.loadLibrary("hello");
    }
```

```
    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

notiamo come venga caricata una libreria in maniera statica prima di tutto il resto (vedi blocco statico). La libreria "`hello`", verrà caricata ricercata con estensione ".ddl" su sistemi Windows e verrà ricercato il file "libhello.so" su sistemi Unix. In particolare, il caricamento della libreria dipende dalla piattaforma utilizzata, e questo implica possibili problemi di portabilità.
Notiamo inoltre come il metodo `main()` vada a chiamare il metodo nativo `displayHelloWorld()`, che come un metodo astratto, non è definito.
Dopo aver compilato il file, bisogna utilizzare l'utility "javah" del JDK con la seguente sintassi:

javah -jni HelloWorld

in questo modo otterremo il seguente file header "HelloWorld.h":

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
  #endif
  /*
   * Class:     HelloWorld
   * Method:    displayHelloWorld
   * Signature: ()V
   */
  JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
```

```
#endif
```

L'unica cosa da notare è il nome della funzione C, che viene denominata `Java_HelloWorld_displayHelloWorld`, ovvero `Java_NomeClasse_nomeMetodo`.
Ovviamente poi, bisogna codificare il file C++, che viene chiamato "HelloWorldImpl.c":

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>
JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject
obj)
{
    printf("Hello World!\n");
    return;
}
```

Questo bisogna compilarlo in modo tale da renderlo una libreria. Per esempio su Windows, se utilizziamo Microsoft Visual C++ 4.0, allora bisognerà lanciare il comando:

```
cl -Ic:\java\include -Ic:\java\include\win32 -LD HelloWorldImpl.c -Fehello.dll
```

Per ottenere il file "hello.dll".
Invece su sistemi Solaris, è necessario specificare la seguente istruzione da una shell:

```
cc -G -I/usr/local/java/include - I/usr/local/java/include/solaris \
HelloWorldImpl.c -o libhello.so
```

per ottenere la libreria "libhello.so".
Non ci resta che mandare in esecuzione l'applicativo:

```
java HelloWorld
```

Per approfondimenti, è possibile dare uno sguardo al Java Tutorial della Sun.

Ovviamente JNI è una tecnologie che rende Java dipendente dal sistema operativo…

9.9.3 Il modificatore `volatile`

Il modificatore `volatile`, è un altro modificatore dall'utilizzo molto raro. Serve per marcare variabili d'istanza, in modo tale che la Java Virtual Machine, utilizzi una particolare ottimizzazione nel loro uso, in caso di accesso parallelo da più thread. Infatti, quando più thread condividono l'utilizzo di una variabile, la JVM, per ottimizzare le prestazioni, crea copie della variabile per ogni thread, preoccupandosi poi di sincronizzare i loro valori con la variabile vera, quando lo ritiene opportuno. Se dichiariamo la variabile condivisa `volatile`, faremo in modo che la JVM, sincronizzi il suo valore con le relative copie dopo ogni cambio di valore. L'utilità è quindi limitata a pochi casi, relativi ad ambiti molto complessi.

Fino ad ora non abbiamo ancora parlato di thread, ma il modulo 11, è interamente dedicato alla loro gestione. Quindi, è possibile ritornare a leggere queste poche righe, dopo aver studiato il modulo 11.

Riepilogo

In questo modulo abbiamo illustrato i package e la gestione di cui deve tenere conto uno sviluppatore. Inoltre abbiamo spiegato il significato dei modificatori più importanti e per ognuno, sono state illustrate le possibilità di utilizzo rispetto ai componenti di un'applicazione Java. Per quanto riguarda gli specificatori d'accesso e `final`, non ci dovrebbero essere state particolari difficoltà d'apprendimento. Per il modificatore `static` invece, abbiamo visto le sue caratteristiche e ne abbiamo evidenziato i problemi. Se si vuole veramente programmare ad oggetti, bisogna essere cauti con il suo utilizzo.
La nostra attenzione si è poi rivolta al modificatore `abstract` e, di conseguenza, all'importanza delle classi astratte e delle interfacce. Questi, due concetti rappresentano due tra le più importanti caratteristiche della programmazione ad oggetti. Inoltre, abbiamo anche evidenziato le differenze tra questi due concetti formalmente simili. Introducendo le enumerazioni, abbiamo poi aggiunto un altro importante tassello che mancava tra i componenti fondamentali di un programma Java. Infine abbiamo introdotto per completezza, anche i modificatori meno utilizzati: `native`, `volatile` e `strictfp`.

Esercizi modulo 9

Esercizio 9.a)
Modificatori e package, Vero o Falso:

1. Una classe dichiarata `private` non può essere utilizzata al di fuori del package in cui è dichiarata
2. La seguente dichiarazione di classe è scorretta:
```
public static class Classe {...}
```
3. La seguente dichiarazione di classe è scorretta:
```
public final class Classe extends AltraClasse {...}
```
4. La seguente dichiarazione di metodo è scorretta:
```
public final void metodo ();
```
5. Un metodo statico può utilizzare solo variabili statiche, e per essere utilizzato, non bisogna per forza istanziare un oggetto dalla classe in cui è definito.
6. Se un metodo è dichiarato `final`, non si può fare overload
7. Una classe `final`, non è accessibile al di fuori del package in cui è dichiarata
8. Un metodo `protected` viene ereditato in ogni sottoclasse qualsiasi sia il suo package
9. Una variabile `static`, viene condivisa da tutte le istanza della classe a cui appartiene
10. Se non anteponiamo modificatori ad un metodo, il metodo è accessibile solo all'interno dello stesso package

Esercizio 9.b)
Classi astratte ed interfacce, Vero o Falso:

1. La seguente dichiarazione di classe è scorretta:
```
public abstract final class Classe {...}
```
2. La seguente dichiarazione di classe è scorretta:
```
public abstract class Classe;
```
3. La seguente dichiarazione di interfaccia è scorretta:
```
public final interface Classe {...}
```
4. Una classe astratta contiene per forza metodi astratti
5. Un'interfaccia può essere estesa da un'altra interfaccia
6. Una classe può estendere una sola classe ma implementare più interfacce

7. Il pregio delle classi astratte e delle interfacce è che obbligano le sottoclassi ad implementare i metodi astratti ereditati. Quindi, rappresentano un ottimo strumento per la progettazione object oriented
8. Il polimorfismo può essere favorito dalla definizione di interfacce
9. Un'interfaccia può dichiarare solo costanti statiche e pubbliche
10. Una classe astratta può implementare un'interfaccia

Soluzioni esercizi modulo 9

Esercizio 9.a)
Modificatori e package, Vero o Falso:

1. **Falso** `private` non si può utilizzare con la dichiarazione di una classe
2. **Vero** `static` non si può utilizzare con la dichiarazione di una classe
3. **Falso**
4. **Vero** manca il blocco di codice (non è un metodo `abstract`)
5. **Vero**
6. **Falso** se un metodo è dichiarato `final`, non si può fare override
7. **Falso** una classe `final`, non si può estendere
8. **Vero**
9. **Vero**
10. **Vero**

Esercizio 9.b)
Classi astratte ed interfacce, Vero o Falso:

1. **Vero** i modificatori `abstract` e `final` sono in contraddizione
2. **Vero** manca il blocco di codice che definisce la classe
3. **Vero** un'interfaccia `final` non ha senso
4. **Falso**
5. **Vero**
6. **Vero**
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper utilizzare tutti i modificatori d'accesso (unità 9.1, 9.2)	☐	
Saper dichiarare ed importare package (unità 9.3)	☐	
Saper utilizzare il modificatore final (unità 9.4)	☐	
Saper utilizzare il modificatore static (unità 9.5)	☐	
Saper utilizzare il modificatore abstract (unità 9.6)	☐	
Conoscere cosa sono le classi interne e le classi anonime (unità 8.4)	☐	
Comprendere l'utilità di classi astratte ed interfacce (unità 9.6, 9.7)	☐	
Comprendere e saper utilizzare l'ereditarietà multipla(unità 9.7)	☐	
Comprendere e saper utilizzare le Enumerazioni (unità 9.8).	☐	
Saper accennare alle definizione dei modificatori strictfp, volatile e native (unità 9.9)	☐	

Note:

10

Complessità: alta

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere le varie tipologie di eccezioni, errori ed asserzioni (unità 10.1).
2. Saper gestire le varie tipologie di eccezioni con i blocchi try – catch (unità 10.2).
3. Saper creare tipi di eccezioni personalizzate e gestire il meccanismo di propagazione con le parole chiave throw e throws (unità 10.4).
4. Capire e saper utilizzare il meccanismo delle asserzioni (unità 10.5).

10 Eccezioni ed asserzioni

I concetti relativi ad eccezioni, errori ed asserzioni e le relative gestioni, permettono allo sviluppatore di scrivere del software robusto, ovvero che riesca a funzionare correttamente, anche in presenza di situazioni impreviste. Questo modulo non solo completa in qualche modo il discorso sulla sintassi fondamentale del linguaggio, ma è in pratica dedicato alla creazione di software robusto.

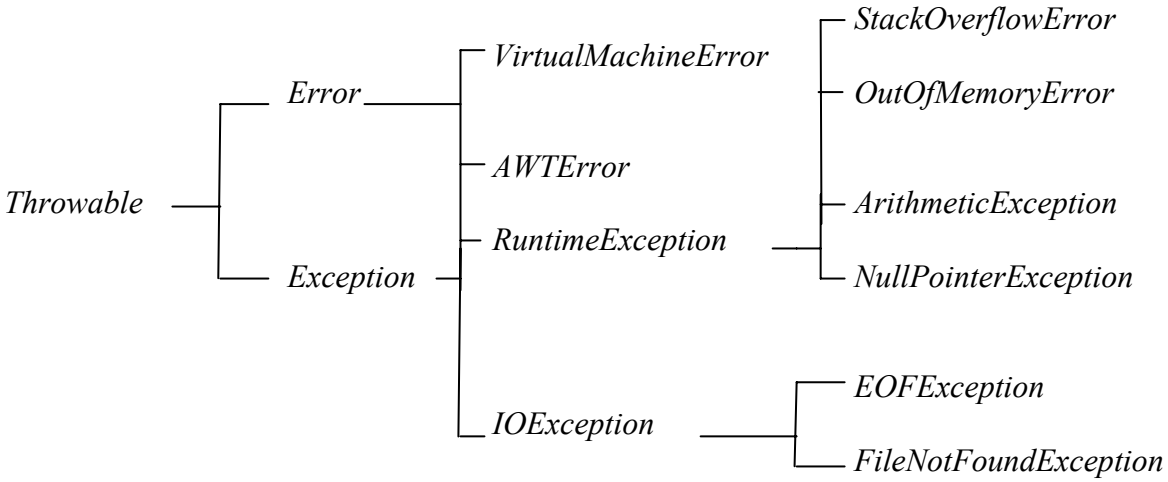
10.1 Eccezioni, errori ed asserzioni

Dei tre argomenti di cui tratta questo modulo, il più importante è sicuramente la gestione delle eccezioni, vero e proprio punto cardine del linguaggio.
E' possibile definire un'**eccezione** come un situazione imprevista che il flusso di un'applicazione può incontrare. È possibile gestire un'eccezione in Java, imparando ad utilizzare cinque semplici parole chiave: `try`, `catch`, `finally`, `throw` e `throws`. Sarà anche possibile creare eccezioni personalizzate e decidere non solo come, ma anche in quale parte del codice gestirle, grazie ad un meccanismo di propagazione estremamente potente. Questo concetto è implementato nella libreria Java mediante la classe `Exception` e le sue sottoclassi. Un esempio di eccezione che potrebbe verificarsi all'interno di un programma è quella relativo ad un divisione tra due variabili numeriche nella quale la variabile divisore ha valore `0`. Come è noto infatti, tale operazione non è fattibile.

È invece possibile definire un **errore** come una situazione imprevista non dipendente da un errore commesso dallo sviluppatore. A differenza delle eccezioni quindi, gli errori non sono gestibili. Questo concetto è implementato nella libreria Java mediante la classe `Error` e le sue sottoclassi. Un esempio di errore che potrebbe causare un programma è quello relativo alla terminazione delle risorse di memoria. Ovviamente, questa condizione non è gestibile.
Infine è possibile definire un'**asserzione** come una condizione che deve essere verificata affinché lo sviluppatore consideri corretta una parte di codice. A differenza delle eccezioni e degli errori, le asserzioni rappresentano uno strumento da abilitare per testare la robustezza del software, ed eventualmente disabilitare in fase di rilascio. In questo modo l'esecuzione del software non subirà nessun tipo di rallentamento. Questo concetto è implementato tramite la parola chiave `assert`. Un esempio d'asserzione potrebbe essere quello di asserire che la variabile che deve fare da divisore in una divisione, deve essere diversa da `0`. Se questa condizione non dovesse verificarsi allora l'esecuzione del codice sarà interrotta.

1.1. 10.2 Gerarchie e categorizzazioni

Nella libreria standard di Java, esiste una gerarchia di classi che mette in relazione, la classe `Exception` e la classe `Error`. Infatti, entrambe queste classi estendono la superclasse `Throwable`. Segue uno schema che mostra tale gerarchia:



Un'ulteriore categorizzazione delle eccezioni, è data dalla divisione delle eccezioni in checked ed unchecked exception. Ci si riferisce alle `RuntimeException` (e le sue sottoclassi) come unchecked exception.

Tutte le altre eccezioni (ovvero tutte quelle che non derivano da `RuntimeException`), vengono dette checked exception. Se si utilizza un metodo che lancia una checked exception senza gestirla da qualche parte, la compilazione non andrà a buon fine. Da qui il termine checked exception (in italiano "eccezioni controllate").

Come abbiamo già detto, non bisognerà fare confusione tra il concetto di errore (problema che un programma non può risolvere) e di eccezione (problema non critico gestibile). Il fatto che sia la classe `Exception` sia la classe `Error`, estendano una classe che si chiama "lanciabile" (`Throwable`), è dovuto al meccanismo con cui la Java Virtual Machine reagisce quando si imbatte in una eccezione-errore. Infatti, se il nostro programma genera un'eccezione durante il runtime, la JVM istanzia un oggetto dalla classe eccezione relativa al problema, e "lancia" l'eccezione appena istanziata (tramite la parola chiave `throw`). Se il nostro codice non "cattura" (tramite la parola chiave `catch`) l'eccezione, il gestore automatico della JVM interromperà il programma generando in output informazioni dettagliate su ciò che è accaduto. Per esempio, supponiamo che durante l'esecuzione un programma provi ad eseguire una divisione per zero tra interi. La JVM istanzierà un oggetto di tipo `ArithmeticException` (inizializzandolo opportunamente) e lo lancerà. In pratica è come se la JVM eseguisse le seguenti righe di codice:

```
ArithmeticException exc = new ArithmeticException();
throw exc;
```

Tutto avviene "dietro le quinte", e sarà trasparente al lettore.

10.3 Meccanismo per la gestione delle eccezioni

Come già asserito in precedenza, lo sviluppatore ha a disposizione alcune parole chiave per gestire le eccezioni: `try`, `catch`, `finally`, `throw` e `throws`. Se bisogna sviluppare una parte di codice che potenzialmente può scatenare un'eccezione è possibile circondarlo con un blocco `try` seguito da uno o più blocchi `catch`. Per esempio:

```
public class Ecc1 {
    public static void main(String args[]) {
        int a = 10;
```

```
        int b = 0;
        int c = a/b;
        System.out.println(c);
    }
}
```

Questa classe può essere compilata senza problemi, ma genererà un'eccezione durante la sua esecuzione, dovuta all'impossibilità di eseguire una divisione per zero. In tal caso, la JVM dopo aver interrotto il programma produrrà il seguente output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Ecc1.main(Ecc1.java:6)
```

Un messaggio di sicuro molto esplicativo, infatti sono stati evidenziati:

- il tipo di eccezione (java.lang.ArithmeticException)
- un messaggio descrittivo (/ by zero)
- il metodo in cui è stata lanciata l'eccezione (at Ecc1.main)
- il file in cui è stata lanciata l'eccezione (Ecc1.java)
- la riga in cui è stata lanciata l'eccezione (:6)

L'unico problema è che il programma è terminato prematuramente. Utilizzando le parole chiave `try` e `catch` sarà possibile gestire l'eccezione in maniera personalizzata:

```
public class Ecc2 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
    }
}
```

Quando la JVM eseguirà tale codice incontrerà la divisione per zero della prima riga del blocco `try`, lancerà l'eccezione `ArithmeticException` che verrà catturata nel blocco `catch` seguente. Quindi non sarà eseguita la riga che doveva stampare la variabile `c`, bensì la stringa "Divisione per zero...", con la quale abbiamo gestito l'eccezione, ed abbiamo permesso al nostro programma di terminare in maniera naturale. Come il lettore avrà sicuramente notato, la sintassi dei blocchi `try – catch` è piuttosto strana, ma presto ci si fa l'abitudine, perché è presente più volte praticamente in tutti i programmi Java. In particolare il blocco `catch` deve dichiarare un parametro (come se fosse un metodo) del tipo dell'eccezione che deve essere catturata. Nell'esempio precedente il reference `exc`, puntava proprio all'eccezione che la JVM aveva istanziato e lanciato. Infatti tramite esso, è possibile reperire informazioni proprio sull'eccezione stessa. Il modo più utilizzato e completo per ottenere informazioni su ciò che è successo, è invocare il metodo `printStackTrace()` sull'eccezione stessa:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    exc.printStackTrace();
}
```

Il metodo `printStackTrace()` produrrà in output i messaggi informativi (di cui sopra) che il programma avrebbe prodotto se l'eccezione non fosse stata gestita, ma senza interrompere il programma stesso.
È ovviamente fondamentale che si dichiari, tramite il blocco `catch`, un'eccezione del tipo giusto. Per esempio, il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (NullPointerException exc) {
```

```
    exc.printStackTrace();
}
```

produrrebbe un'eccezione non gestita, e, quindi, un'immediata terminazione del programma. Infatti, il blocco `try` non ha mai lanciato una `NullPointerException`, ma una `ArithmeticException`.
Come per i metodi, anche per i blocchi `catch` i parametri possono essere polimorfi. Per esempio, il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

contiene un blocco `catch` che gestirebbe qualsiasi tipo di eccezione, essendo `Exception`, la superclasse da cui discende ogni altra eccezione. Il reference `exc`, è in questo esempio, un parametro polimorfo.
È anche possibile far seguire ad un blocco `try`, più blocchi `catch`, come nel seguente esempio:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
```

```
    exc.printStackTrace();
}
```

In questo modo il nostro programma risulterebbe più robusto, e gestirebbe diversi tipi di eccezioni. Male che vada (ovvero il blocco `try` lanci un'eccezione non prevista), l'ultimo blocco `catch` gestirà il problema.
E' ovviamente fondamentale l'ordine dei blocchi `catch`. Se avessimo:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
```

allora gli ultimi due `catch` sarebbero superflui e il compilatore segnalerebbe l'errore nel seguente modo:

```
C:\Ecc2.java:12: exception java.lang.ArithmeticException has already been
caught
      catch (ArithmeticException exc) {
      ^
C:\Ecc2.java:15: exception java.lang.NullPointerException has already been
caught
      catch (NullPointerException exc) {
      ^
2 errors
```

È anche possibile far seguire ad un blocco `try`, oltre a blocchi `catch`, un altro blocco definito dalla parola chiave `finally`, per esempio:

```
public class Ecc4 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
            System.out.println("Tentativo di
operazione");
        }
    }
}
```

Ciò che è definito in un blocco `finally`, viene eseguito in qualsiasi caso, sia se viene lanciata l'eccezione, sia se non viene lanciata. Per esempio, è possibile utilizzare un blocco `finally` quando esistono operazioni critiche che devono essere eseguite in qualsiasi caso. L'output del precedente programma è:

```
Divisione per zero...
tentativo di operazione
```

Se invece la variabile `b` fosse settata a `2` piuttosto cha a `0`, allora l'output sarebbe:

```
5
tentativo di operazione
```

Un classico esempio (più significativo del precedente) in cui la parola `finally` è spesso utilizzata è il seguente:

```
public void insertInDB() {
   try {
        cmd.executeUpdate("INSERT INTO…");
     }
   catch (SQLException exc) {
        exc.printStackTrace();
   }
   finally {
        connection.close();
   }
}
```

Il metodo precedente, tenta di eseguire una "INSERT" in un database, tramite le interfacce JDBC offerte dal package `java.sql`. Nell'esempio `cmd` è un oggetto `Statement` e `connection` è un oggetto di tipo `Connection`. Il comando `executeUpdate()` specifica come parametro una stringa con codice SQL per inserire un certo record in una certa tabella di un certo database. Se ci sono problemi (per esempio sintassi SQL scorretta, chiave primaria già presente, etc…) la JVM lancerà una `SQLException`, che verrà catturata nel relativo blocco `catch`. In ogni caso, dopo il tentativo di inserimento, la connessione al database deve essere chiusa. (per maggiori approfondimenti su JDBC rimandiamo il lettore al modulo relativo oppure, per una breve introduzione all'indirizzo, http://www.claudiodesio.com/java/jdbc.htm).
Giusto per essere precisi, la chiamata al metodo `close()` dovrebbe essere gestita con un altro blocco `try-catch`, nel modo seguente:

```
finally {
    try{
        connection.close();
    } catch (SQLException exc) {
        exc.printStackTrace();
    }
}
```

E' possibile anche far seguire ad un blocco `try`, direttamente un blocco `finally`. Quest'ultimo verrà eseguito sicuramente dopo

l'esecuzione del blocco `try`, sia se l'eccezione viene lanciata, sia se non viene lanciata. Comunque, se l'eccezione venisse lanciata, non essendo gestita con un blocco `catch`, il programma terminerebbe anormalmente.

A questo punto in molti si potrebbero chiedere il perché gestire le eccezioni con blocchi `try – catch`, piuttosto che utilizzare dei semplici `if`. La risposta sarà implicitamente data nei prossimi paragrafi.

10.4 Eccezioni personalizzate e propagazione dell'eccezione

Ci sono alcune tipologie di eccezioni che sono più frequenti e quindi più conosciute dagli sviluppatori Java. Si tratta di:

- `NullPointerException` : probabilmente la più frequente tra le eccezioni. Viene lanciata dalla JVM, quando per esempio viene chiamato un metodo su di un reference che invece punta a `null`.
- `ArrayIndexOutOfBoundsException` : questa eccezione viene ovviamente lanciata quando si prova ad accedere ad un indice di un array troppo alto.
- `ClassCastException` : eccezione particolarmente insidiosa. Viene lanciata al runtime quando si prova ad effettuare un cast ad un tipo di classe sbagliato.

Queste eccezioni appartengono tutte al package `java.lang`. Inoltre, se si utilizzano altri package come `java.io`, bisognerà gestire spesso le eccezioni come `IOException` e le sue sottoclassi (`FileNotFoundException`, `EOFException`, etc…). Stesso discorso con la libreria `java.sql` e l'eccezione `SQLException`, il package `java.net` e la `ConnectException` e così via. Lo sviluppatore imparerà con l'esperienza come gestire tutte queste eccezioni.
È però altrettanto probabile che qualche volta occorra definire nuovi tipi di eccezioni. Infatti, per un particolare programma. potrebbe essere una eccezione anche una divisione per 5. Più verosimilmente, un programma che deve gestire in maniera automatica le prenotazioni per un teatro, potrebbe voler lanciare un'eccezione nel momento in cui si tenti di prenotare un posto non più disponibile. In tal caso la soluzione è estendere la classe `Exception`, ed eventualmente aggiungere membri e fare override di metodi come `toString()`. Segue un esempio:

```
public class PrenotazioneException extends Exception {
    public PrenotazioneException() {
```

```
// Il costruttore di Exception chiamato inizializza la
// variabile privata message
        super("Problema con la prenotazione");
    }
    public String toString() {
        return getMessage() + ": posti esauriti!";
    }
}
```

La "nostra" eccezione, contiene informazioni sul problema, e rappresenta una astrazione corretta. Tuttavia la JVM, non può lanciare automaticamente una `PrenotazioneException` nel caso si tenti di prenotare quando non ci sono più posti disponibile. La JVM infatti, sa quando lanciare una `ArithmeticException` ma non sa quando lanciare una `PrenotazioneException`. In tal caso sarà compito dello sviluppatore lanciare l'eccezione. Esiste infatti la parola chiave `throw` (in inglese "lancia"), che permette il lancio di un'eccezione tramite la seguente sintassi:

```
PrenotazioneException exc = new PrenotazioneException();
throw exc;
```

o equivalentemente (dato che il reference `exc` poi non sarebbe più utilizzabile):

```
throw new PrenotazioneException();
```

Ovviamente il lancio dell'eccezione dovrebbe seguire un controllo condizionale come il seguente:

```
if (postiDisponibili == 0) {
    throw new PrenotazioneException();
}
```

Il codice precedente ovviamente farebbe terminare prematuramente il programma a meno di gestire l'eccezione come segue:

```
try {
        //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
```

```
        throw new PrenotazioneException();
    }
        //istruzione eseguita
        // se non viene lanciata l'eccezione
    postiDisponibili--;
}
catch (PrenotazioneException exc){
    System.out.println(exc.toString());
}
```

Il lettore avrà sicuramente notato che il codice precedente non rappresenta un buon esempio di gestione dell'eccezione: dovendo utilizzare la condizione `if`, sembra infatti superfluo l'utilizzo dell'eccezione. In effetti è così! Ma ci deve essere una ragione per la quale esiste la possibilità di creare eccezioni personalizzate e di poterle lanciare. Questa ragione è la "**propagazione dell'eccezione**" per i metodi chiamanti. La potenza della gestione delle eccezioni è dovuta essenzialmente a questo meccanismo di propagazione. Per comprenderlo bene, affidiamoci come al solito ad un esempio.
Supponiamo di avere la seguente classe:

```
public class Botteghino {
  private int postiDisponibili;

  public Botteghino() {
    postiDisponibili = 100;
  }

  public void prenota() {
    try {
        //controllo sulla disponibilità dei posti
      if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
      }
        //metodo che realizza la prenotazione
        // se non viene lanciata l'eccezione
      postiDisponibili--;
    }
    catch (PrenotazioneException exc){
      System.out.println(exc.toString());
```

```
    }
  }
}
```

La classe `Botteghino` astrae in maniera semplicistica, un botteghino virtuale che permette di prenotare i posti in un teatro. Ora consideriamo la seguente classe eseguibile (con metodo `main()`) che utilizza la classe `Botteghino`:

```
public class GestorePrenotazioni {
  public static void main(String [] args) {
    Botteghino botteghino = new Botteghino();
    for (int i = 1; i <= 101; ++i){
      botteghino.prenota();
      System.out.println("Prenotato posto n° " + i);
    }
  }
}
```

Per una classe del genere, il fatto che l'eccezione sia gestita all'interno della classe `Botteghino`, rappresenta un problema. Infatti l'output del programma sarà:

```
Prenotato posto n° 1
Prenotato posto n° 2
...
Prenotato posto n° 99
Prenotato posto n° 100
Problema con la prenotazione: posti esauriti!
Prenotato posto n° 101
```

che ovviamente contiene una contraddizione. Gestire eccezioni è sempre una operazione da fare, ma non sempre bisogna gestire eccezioni laddove si presentano. In questo caso, l'ideale sarebbe gestire l'eccezione nella classe `GestorePrenotazioni`, piuttosto che nella classe `Botteghino`:

```
public class GestorePrenotazioni {
  public static void main(String [] args) {
    Botteghino botteghino = new Botteghino();
    try {
```

```
      for (int i = 1; i <= 101; ++i){
        botteghino.prenota();
        System.out.println("Prenotato posto n° " + i);

      }
    }
    catch (PrenotazioneException exc) {
      System.out.println(exc.toString());
    }
  }
}
```

Tutto ciò è fattibile grazie al meccanismo di propagazione dell'eccezione di Java. Per compilare la classe `Botteghino` però, non basta rimuovere il blocco `try – catch` dal metodo `prenota()`, ma bisogna anche utilizzare la parola chiave `throws` nel seguente modo:

```
public void prenota() throws PrelievoException {
      //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
      //lancio dell'eccezione
      throw new PrenotazioneException();
    }
      //metodo che realizza la prenotazione
      // se non viene lanciata l'eccezione
    postiDisponibili--;
}
```

In questo modo otteremo il seguente desiderabile output:

```
Prenotato posto n° 1
Prenotato posto n° 2
. . .
Prenotato posto n°99
Prenotato posto n°100
Problema con la prenotazione: posti esauriti!
```

Se non utilizzassimo la clausola `throws` nella dichiarazione del metodo, il compilatore non compilerebbe il codice precedente. Infatti, segnalerebbe che il metodo `prenota()` potrebbe lanciare l'eccezione `PrenotazioneException` (che è evidente al compilatore per la parola chiave `throw`), e che questa, non viene gestita. In particolare il messaggio di errore restituito sarebbe simile al seguente:

Botteghino.java:5: unreported exception PrenotazioneException; must be caught or declared to be thrown

Questo messaggio è una ulteriore prova delle caratteristiche di robustezza di Java.
Con la clausola `throws` nella dichiarazione del metodo, in pratica è come se avvertissimo il compilatore che siamo consapevoli che il metodo possa lanciare al runtime la `PrelievoException`, e di non "preoccuparsi", perché gestiremo in un'altra parte del codice l'eccezione.

Se un metodo "chiamante" vuole utilizzare un altro metodo "daChiamare" che dichiara con una clausola `throws` il possibile lancio di un certo tipo di eccezione, allora, il metodo "chiamante", o deve gestire l'eccezione con un blocco `try - catch` che include la chiamata al metodo "daChiamare", o deve dichiarare anch'esso una clausola `throws` alla stessa eccezione. Ad esempio, ciò vale per il metodo `main()` della classe `GestorePrenotazioni`.

Molti metodi della libreria standard sono dichiarati con clausola `throws` a qualche eccezione. Per esempio molti metodi delle classi del package `java.io`, dichiarano clausole `throws` alla `IOException` (eccezione di input - output). Appare ancora più chiaro ora la categorizzazione tra eccezioni checked ed unchecked: le checked exception devono essere per forza gestite per poter compilare, le unchecked no, dato che si presentano solo al runtime.

E' possibile dichiarare nella clausola `throws` anche più di una eccezione, separando le varie tipologie con virgole, come nel seguente esempio:

```
public void prenota() throws PrenotazioneException,
NullPointerException { . . .
```

Dalla versione 1.4 del linguaggio, è stata introdotta una nuova caratteristica alle eccezioni. La classe `Throwable` infatti, è stata modificata per supportare un semplice meccanismo di wrapping. Spesso infatti, si rende necessario per lo sviluppatore catturare una certa eccezione per lanciarne un'altra. Per esempio:

```
try {
    ...
} catch(UnaCertaEccezione e) {
    throw new UnAltraEccezione();
}
```

In tali casi però, l'informazione della prima eccezione (nell'esempio `UnaCertaEccezione`) viene persa. Si era quindi costretti a creare un'eccezione personalizzata che poteva contenerne un'altra come variabile d'istanza. Per esempio:

```
public class WrapperException {
    private Exception altraEccezione;
    public WrapperException(Exception altraEccezione){
        this.setAltraEccezione(altraEccezione);
    }
    public void setAltraEccezione(Exception
          altraEccezione) {
        this.altraEccezione = altraEccezione;
    }
    public Exception getAltraEccezione() {
        return altraEccezione;
    }
    ...
}
```

Con questo tipo di eccezione è possibile includere un'eccezione in un'altra in questo modo:

```
try {
    ...
} catch(UnaCertaEccezione e) {
```

```
    throw new WrapperException(e);
}
```

Ma dalla versione 1.4, non bisogna più creare un'eccezione personalizzata per ottenere eccezioni wrapper. Nella classe `Throwable` sono stati introdotti i metodi `getCause()` e `initCause(Throwable)`, e due nuovi costruttori, `Throwable(Throwable)` e `Throwable(String, Throwable)`. È quindi ora possibile per esempio codificare le seguenti istruzioni:

```
try {
    ...
} catch(ArithmeticException e) {
    throw new SecurityException(e);
}
```

E' ovviamente possibile, concatenare un numero arbitrario di eccezioni.

10.4.1 Precisazione sull'override

Quando si fa override di un metodo, non è possibile specificare clausole `throws` ad eccezioni che il metodo base non ha nella propria clausola `throws`. È comunque possibile da parte del metodo che fa override, dichiarare una clausola `throws` ad eccezioni che sono sottotipi di eccezioni che il metodo base, ha nella sua clausola `throws`. Per esempio:

```
public class ClasseBase {
    public void metodo() throws java.io.IOException { }
}

class SottoClasseCorretta1 extends ClasseBase {
    public void metodo() throws java.io.IOException {}
}

 class SottoClasseCorretta2 extends ClasseBase {
    public void metodo() throws
java.io.FileNotFoundException {}
}

 class SottoClasseCorretta3 extends ClasseBase {
```

```
    public void metodo() {}
}

 class SottoClasseScorretta extends ClasseBase {
   public void metodo() throws java.sql.SQLException {}
}
```

La classe `ClasseBase` ha un metodo che dichiara nella sua clausola `throws` una `IOException`. La classe `SottoClasseCorretta1` fa override del metodo e dichiara la stessa `IOException` nella sua clausola `throws`. La classe `SottoClasseCorretta2` fa override del metodo e dichiara una `FileNotFoundException`, che è sottoclasse di `IOException` nella sua clausola `throws`. La classe `SottoClasseCorretta3` fa override del metodo e non dichiara clausole `throws`. Infine la classe `SottoClasseScorretta`, fa override del metodo e dichiara una `SQLException` nella sua clausola `throws`, e ciò è illegale.

10.5 Introduzione alle Asserzioni

Dalla versione 1.4 di java, è stata introdotta una nuova e clamorosa caratteristica al linguaggio. Clamorosa perché si è dovuto addirittura modificare la lista delle parole chiave con una nuova: `assert`, cosa mai accaduta nelle precedenti major release. Un'asserzione è un'istruzione che permette di testare eventuali comportamenti che un'applicazione deve avere. Ogni asserzione richiede che sia verificata un'espressione booleana che lo sviluppatore ritiene debba essere verificata, nel punto in cui viene dichiarata. Se questa non è verificata, allora si deve parlare di bug. Le asserzioni possono quindi rappresentare un'utile strumento per accertarsi che il codice scritto si comporti così come ci si aspetta. Lo sviluppatore può disseminare il codice di asserzioni, in modo tale da testare la robustezza del codice in maniera semplice ed efficace. Lo sviluppatore può infine disabilitare la lettura delle asserzioni da parte della JVM, in fase di rilascio del software, in modo tale che l'esecuzione non venga in nessun modo rallentata. Moltissimi sviluppatori pensano che l'utilizzo delle asserzioni sia una delle tecniche di maggior successo per scovare bug. Inoltre, le asserzioni rappresentano anche un ottimo strumento per documentare il comportamento interno di un programma, favorendo la manutenibilità dello stesso.

10.5.1 Sintassi

Esistono due tipi di sintassi per poter utilizzare le asserzioni:

1. `assert espressione_booleana;`
2. `assert espressione_booleana: espressione_stampabile;`

Con la sintassi 1) quando l'applicazione esegue l'asserzione valuta il valore dell'`espressione_booleana`. Se questo è `true`, il programma prosegue normalmente, ma se il valore è `false` viene lanciato l'errore `AssertionError`. Per esempio l'istruzione:

```
assert b > 0;
```

è semanticamente equivalente a:

```
if (!(b>0)) {
   throw new AssertionError();
}
```

A parte l'eleganza e la compattezza del costrutto `assert`, la differenza tra le precedenti due espressioni è notevole. Le asserzioni rappresentano più che un'istruzione applicativa classica, uno strumento per testare la veridicità delle assunzioni che lo sviluppatore fa della propria applicazione. Se la condizione che viene asserita dal programmatore è falsa, l'applicazione terminerà immediatamente mostrando le ragioni tramite uno stack-trace (metodo `printStackTrace()` di cui sopra). Infatti si è verificato qualcosa che non era previsto dallo sviluppatore stesso. È possibile disabilitare la lettura delle asserzioni da parte della JVM, una volta rilasciato il proprio prodotto, al fine di non rallentarne l'esecuzione. Ciò evidenzia la differenza tra le asserzioni e tutte le altre istruzioni applicative.
Rispetto alla sintassi 1), la sintassi 2) permette di specificare anche un messaggio esplicativo tramite l'`espressione_stampabile`. Per esempio

```
assert b > 0: b;
```

oppure

```
assert b > 0: "il valore di b è " + b;
```

oppure

```
assert b > 0: getMessage();
```

o anche

```
assert b > 0: "assert b > 0 = " + (b > 0);
```

l'`espressione_stampabile` può essere una qualsiasi espressione che ritorni un qualche valore (quindi non è possibile invocare un metodo con tipo di ritorno `void`). La sintassi 2) permette quindi di migliorare lo stack-trace delle asserzioni.

10.5.2 Progettazione per contratto

Il meccanismo delle asserzioni, deve il suo successo ad una tecnica di progettazione nota con il nome di "Progettazione per contratto" ("Design by contract"), sviluppata da Bertrand Meyer. Tale tecnica è una caratteristica fondamentale del linguaggio di programmazione sviluppato da Meyer stesso: l'Eiffel (per informazioni http://www.eiffel.com). Ma è possibile progettare per contratto, più o meno agevolmente, con qualsiasi linguaggio di programmazione. La tecnica si basa in particolare su tre tipologie di asserzioni: pre-condizioni, post-condizioni ed invarianti (le invarianti a loro volta si dividono in interne, di classe, sul flusso di esecuzione, etc…).
Con una **pre-condizione** lo sviluppatore può specificare quale deve essere lo stato dell'applicazione nel momento in cui viene invocata un'operazione. In questo modo si rende esplicito chi ha la responsabilità di testare la correttezza dei dati. L'utilizzo dell'asserzione riduce sia il pericolo di dimenticare completamente il controllo, sia quello di fare troppi controlli (perché si possono abilitare e disabilitare). Dal momento che si tende ad utilizzare le asserzioni in fase di test e debugging, non bisogna mai confondere l'utilizzo delle asserzioni con quello della gestione delle eccezioni. Nel unità didattica 10.7 verranno esplicitate delle regole da seguire per l'utilizzo delle asserzioni.
Con una **post-condizione** lo sviluppatore può specificare quale deve essere lo stato dell'applicazione nel momento in cui un'operazione viene completata. Le post-condizioni rappresentano un modo utile per dire cosa fare senza dire come. In altre parole è un altro metodo per separare interfaccia ed implementazione interna.
È infine possibile utilizzare il concetto di **invariante**, che se applicato ad una classe, permette di specificare vincoli per tutti gli oggetti istanziati. Questi possono trovarsi in un stato che non rispetta il vincolo specificato (detto "stato inconsistente"), solo temporaneamente durante l'esecuzione di qualche metodo, al termine del quale lo stato deve ritornare "consistente".
La progettazione per contratto, è appunto una tecnica di progettazione, e non di programmazione. Essa permette per esempio anche di testare la consistenza

dell'ereditarietà. Una sottoclasse infatti, potrebbe indebolire le pre-condizioni, e fortificare le post-condizioni e le invarianti di classe, al fine di convalidare l'estensione. Al lettore interessato ad approfondire le sue conoscenze sulla progettazione per contratto, consigliamo di dare uno sguardo alla bibliografia.

10.5.3 Uso delle asserzioni

Per poter sfruttare l'utilità delle asserzioni all'interno dei nostri programmi bisogna compilarli e mandarli in esecuzione utilizzando particolari accorgimenti. L'introduzione della parola chiave `assert` infatti, ha per la prima volta sollevato il problema della compatibilità all'indietro con le precedenti versioni di Java. Non è raro infatti trovare applicazioni scritte precedentemente all'uscita della versione 1.4 di Java, che utilizzano come identificatori di variabili o metodi la parola `assert`. Spesso questo è dovuto proprio alla necessità di alcuni sviluppatori di simulare in Java il meccanismo delle asserzioni, fino ad allora mancante. Quindi, per compilare un'applicazione che fa uso delle asserzioni, bisogna stare attenti anche alla versione di Java che stiamo utilizzando:

10.5.4 Note per la compilazione di programmi che utilizzano la parola assert

Dal momento che attualmente, ancora molti sviluppatori, utilizzano versioni del JDK come la 1.4 o addirittura la 1.3, è doveroso in questa sede fare delle precisazioni

1. Se si utilizza una versione di Java precedente alla 1.4, non è possibile utilizzare le asserzioni e `assert` non è nemmeno una parola chiave.
2. Se si utilizza la versione di Java 1.4 e si vuole sfruttare il meccanismo delle asserzioni in un programma, allora bisogna compilarlo con il flag "–source 1.4", come nel seguente esempio:

   ```
   javac –source 1.4 MioProgrammaConAsserzioni.java
   ```

3. Se non utilizziamo il flag suddetto, allora il compilatore non considererà `assert` come parola chiave. Conseguentemente, programmi che utilizzano `assert` come costrutto non saranno compilati (perché il costrutto non sarà riconosciuto), e allo sviluppatore verrà segnalato con un warning, dato che dalla versione 1.4 `assert` è una parola chiave del linguaggio. I programmi che invece utilizzano `assert` come identificatore di variabili o metodi, saranno compilati correttamente, ma sarà segnalato lo stesso warning di cui sopra.
4. Se si utilizza una versione 1.5 o di Java allora la situazione cambia nuovamente. Infatti se non si specifica il flag "-source", sarà implicitamente utilizzato il flag "-source 1.5" per la versione 5. Per esempio, se si vuole sfruttare il meccanismo

delle asserzioni all'interno del programma, utilizzando un JDK 1.5, basterà quindi compilare senza utilizzare flag, come nel seguente esempio:

javac MioProgrammaConAsserzioni.java

che è equivalente a:

javac -source 1.5 MioProgrammaConAsserzioni.java

ed anche a:

javac -source 5 MioProgrammaConAsserzioni.java

visto che la versione 1.5 di Java è stata pubblicizzata come "Java 5".
Se invece si vuole sfruttare la parola `assert` come identificatore di un metodo o di una variabile (magari perché il codice era stato scritto antecedentemente alla versione 1.4), bisognerà sfruttare il flag "-source" specificando una versione precedente alla 1.4. Per esempio:
javac –source 1.3 MioVecchioProgramma.java
Purtroppo la situazione è questa, e bisogna stare attenti.

10.5.5 Note per l'esecuzione di programmi che utilizzano la parola assert

Come più volte detto, è possibile in fase di esecuzione abilitare o disabilitare le asserzioni. Come al solito bisogna utilizzare dei flag, questa volta applicandoli al comando "java", ovvero "–enableassertions" (o più brevemente "-ea") per abilitare le asserzioni, e "–disableassertions" (o "-da") per disabilitare le asserzioni. Per esempio:

```
java –ea MioProgrammaConAsserzioni
```

Abilita da parte della JVM la lettura dei costrutti `assert`. Mentre

```
java –da MioProgrammaConAsserzioni
```

disabilita le asserzioni, in modo tale da non rallentare in alcun modo l'applicazione. Siccome le asserzioni sono di default disabilitate, il precedente codice è esattamente equivalente al seguente:

```
java MioProgrammaConAsserzioni
```

Sia per l'abilitazione sia per la disabilitazione valgono le seguenti regole:

1) Se non si specificano argomenti dopo i flag di abilitazione o disabilitazione delle asserzioni, allora saranno abilitate o disabilitate le asserzioni in tutte le classi del nostro programma (ma non nelle classi della libreria standard utilizzate). Questo è il caso dei precedenti esempi.
2) Specificando invece il nome di un package seguito da tre puntini, si abilitano o si disabilitano le asserzioni in quel package e in tutti i sotto package. Per esempio il comando:

   java –ea –da:miopackage... MioProgramma

   abiliterà le asserzioni in tutte le classi tranne quelle del package `miopackage`,
3) Specificando solo i tre puntini invece si abilitano o si disabilitano le asserzioni nel package di default (ovvero la cartella da dove parte il comando)
4) Specificando solo un nome di una classe invece si abilitano o si disabilitano le asserzioni in quella classe. Per esempio il comando:

   java –ea:... –da:MiaClasse MioProgramma

   abiliterà le asserzioni in tutte le classi del package di default, tranne che nella classe `MiaClasse`.
5) E' anche possibile eventualmente abilitare o disabilitare, le asserzioni delle classi della libreria standard che si vuole utilizzare mediante i flag "-enablesystemassertions" (o più brevemente "-esa"), e "-disablesystemassertions" (o "-dsa"). Anche per questi flag valgono le regole di cui sopra.
6) Per quanto riguarda la fase di esecuzione, non esistono differenze sul come sfruttare le asserzioni tra la versione 1.4 e le versioni successive… fortunatamente…

In alcuni programmi critici, è possibile che lo sviluppatore si voglia assicurare che le asserzioni siano abilitate. Con il seguente blocco di codice statico:

```
  static {
    boolean assertsEnabled = false;
    assert assertsEnabled = true;
    if (!assertsEnabled)
        throw new RuntimeException("Asserts must "
           + "be enabled! ");
  }
```

è possibile garantire che il programma sia eseguibile solo se le asserzioni sono abilitate. Il blocco infatti, prima dichiara ed inizializza la variabile booleana `assertsEnabled` a `false`, per poi cambiare il suo valore a `true` se le asserzioni sono abilitate. Quindi se le asserzioni non sono abilitate, il programma termina con il lancio della `RuntimeException`, altrimenti continua. Ricordiamo che il blocco statico, viene eseguito (cfr. Modulo 9) un'unica volta nel momento in cui la classe che lo contiene viene caricata. Per questa ragione il blocco statico dovrebbe essere inserito nella classe del `main()` per essere sicuri di ottenere il risultato voluto.

10.5.6 Quando usare le asserzioni

Non tutti gli sviluppatori possono essere interessati all'utilizzo delle asserzioni. Un'asserzione non può ridursi ad essere un modo coinciso di esprimere una condizione regolare. Un'asserzione è invece il concetto fondamentale di una metodologia di progettazione per rendere i programmi più robusti. Nel momento in cui però, lo sviluppatore decide di utilizzare tale strumento, dovrebbe essere suo interesse utilizzarlo correttamente. I seguenti consigli derivano dall'esperienza e dallo studio dei testi relativi alle asserzioni dell'autore.

1) È spesso consigliato (anche nella documentazione ufficiale Sun), non utilizzare pre-condizioni, per testare la correttezza dei parametri di metodi pubblici. È invece raccomandato l'utilizzo delle pre-condizioni per testare la correttezza dei parametri di metodi privati, protetti o con visibilità a livello di package. Questo dipende dal fatto che un metodo non pubblico, ha la possibilità di essere chiamato da un contesto limitato, corretto e funzionante. Ciò implica che assumiamo che le nostre chiamate al metodo in questione sono corrette, ed è quindi lecito rinforzare tale concetto con un'asserzione. Per esempio supponiamo di avere un metodo con visibilità di package come il seguente:

```
public class InstancesFactory {
    Object getInstance(int index) {
        assert (index == 1 || index == 2);
        switch (index) {
            case 1:
                return new Instance1();
            case 2:
                 return new Instance2();
        }
    }
}
```

La classe precedente, implementa una soluzione personalizzata basata sul pattern denominato "Factory Method" (per informazioni sul concetto di pattern, cfr. Appendice H).
Se questo metodo può essere chiamato solo da classi cha appartengono allo stesso package della classe `InstancesFactory`, allora non deve mai accadere che il parametro `index` sia diverso da `1` o `2`, perchè tale situazione rappresenterebbe un bug.
Se invece il metodo `getInstance()`, fosse dichiarato `public`, allora la situazione sarebbe diversa. Infatti, un eventuale controllo del parametro `index`, dovrebbe essere considerato ordinario, e quindi da gestire magari mediante il lancio di un'eccezione:

```
public class InstancesFactory {
    public Object getInstance(int index) throws Exception
{
        if (!(index == 1 || index == 2)) {
            throw new Exception("Indice errato: " +
index);
        }
        switch (index) {
            case 1:
                return new Instance1();
            case 2:
                return new Instance2();
        }
```

```
    }
}
```

L'uso di un'asserzione in tal caso, non garantirebbe la robustezza del programma, ma solo la sua eventuale interruzione, se fossero abilitate le asserzioni al runtime, non potendo a priori controllare la chiamata al metodo. In pratica una pre-condizione di questo tipo violerebbe il concetto object oriented di metodo pubblico.

2) È sconsigliato l'utilizzo di asserzioni laddove si vuole testare la correttezza di dati che sono inseriti da un utente. Le asserzioni dovrebbero testare la consistenza del programma con se stesso, non la consistenza dell'utente con il programma. L'eventuale input non corretto da parte di un utente è giusto che sia gestito mediante eccezioni, non asserzioni. Per esempio, modifichiamo la classe `Data` di cui abbiamo parlato nel modulo 5 per spiegare l'incapsulamento:

```
public class Data {
   private int giorno;
   . . .
    public void setGiorno(int g) {
       assert (g > 0 && g <= 31): "Giorno non valido";
       giorno = g;
    }
    ...
```

dove il parametro `g` del metodo `setGiorno()` veniva passato da un utente mediante un oggetto interfaccia, che rappresentava un interfaccia grafica (codice 5.2 bis):

```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Come il lettore avrà intuito, l'utilizzo della parola chiave `assert` non è corretto. Infatti nel caso le asserzioni fossero abilitate in fase di esecuzione

dell'applicazione, e l'utente inserisse un valore errato per inizializzare la variabile giorno, l'applicazione si interromperebbe con un `AssertError`! Ovviamente se le asserzioni non fossero abilitate allora nessun controllo impedirebbe all'utente di inserire valori errati. La soluzione ideale sarebbe quella di gestire la situazione tramite un'eccezione, per esempio:

```
public void setGiorno(int g) throws RuntimeException {
    if (!(g > 0 && g <= 31)) {
         throw new RuntimeException("Giorno non valido");
    }
    giorno = g;
}
```

Ovviamente la condizione è ampiamente migliorabile…

3) L'uso delle asserzioni invece, ben si adatta alle post-condizioni ed alle invarianti. Per **post-condizione** intendiamo una condizione che viene verificata appena prima che termini l'esecuzione di un metodo (ultima istruzione). Segue un esempio:

```
public class Connection {
    private boolean isOpen = false;
    public void open() {
        // ...
        isOpen = true;
        // ...
        assert isOpen;
    }
    public void close() throws ConnectionException {
        if (!isOpen) {
            throw new ConnectionException(
              "Impossibile chiudere connessioni non" +
                 "aperte!" );
        }
        // ...
        isOpen = false;
        // ...
        assert !isOpen;
```

```
    }
}
```

Dividiamo le **invarianti** in **interne**, **di classe** e **sul flusso di esecuzione**.
Per **invarianti interne** intendiamo asserzioni che testano la correttezza dei flussi del nostro codice. Per esempio il seguente blocco di codice:

```
if (i == 0) {
    ...
} else if (i == 1) {
    ...
} else { // ma sicuramente (i == 2)
    ...
}
```

può diventare più robusto con l'uso di un'asserzione:

```
if (i == 0) {
    ...
} else if (i == 1) {
    ...
} else {
    assert i == 2 : "Attenzione i = " + i + "!";
    ...
}
```

Maggiore probabilità di utilizzo di un tale tipo di invariante, è all'interno di una clausola `default` di un costrutto `switch`. Spesso lo sviluppatore sottovaluta il costrutto omettendo la clausola `default`, perchè suppone che il flusso passi sicuramente per un certo `case`. Per convalidare le nostre supposizioni, sono molto utili le asserzioni. Per esempio il seguente blocco di codice:

```
switch(tipoAuto) {
    case Auto.SPORTIVA:
        ...
    break;
    case Auto.LUSSO:
        ...
```

```
    break;
    case Auto.UTILITARIA:
        ...
    break;
}
```

può diventare più robusto con l'uso di un'asserzione:

```
switch(tipoAuto) {
    case Auto.SPORTIVA:
        ...
    break;
    case Auto.LUSSO:
      ...
    break;
    case Auto.UTILITARIA:
        ...
    break;
    default:
        assert false : "Tipo auto non previsto : " +
        tipoAuto;
}
```

Per **invarianti di classe**, intendiamo particolari invarianti interne che devono essere vere per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi. All'inizio ed al termine di ogni metodo però, lo stato dell'oggetto deve tornare "consistente". Per esempio un oggetto della seguente classe:

```
public class Bilancia {
  private double peso;
  public Bilancia() {
    azzeraLancetta();
    assert lancettaAzzerata();// invariante di classe
  }
  private void setPeso(double grammi) {
    assert grammi > 0; // pre-condizione
    peso = grammi;
  }
```

```
  private double getPeso() {
    return peso;
  }
  public void pesa(double grammi) {
    if (grammi <= 0) {
      throw new RuntimeException("Grammi <= 0! ");
    }
    setPeso(grammi);
    mostraPeso();
    azzeraLancetta();
    assert lancettaAzzerata(); // invariante di classe
  }
  private void mostraPeso() {
    System.out.println("Il peso è di " + peso + "
grammi");
  }
  private void azzeraLancetta() {
    setPeso(0);
  }
  private boolean lancettaAzzerata () {
    return peso == 0;
  }
}
```

potrebbe dopo ogni pesatura, azzerare la lancetta (notare che i due soli metodi pubblici terminano con un'asserzione).
Per **invarianti sul flusso di esecuzione** intendiamo asserzioni che vengono posizionate in posti del codice che non dovrebbero mai essere raggiunte. Per esempio, se abbiamo un pezzo di codice che viene commentato in tal modo:

```
public void metodo() {
    if (flag == true) {
        return;
    }
    // L'esecuzione non dovrebbe mai arrivare qui!
}
```

Potremmo sostituire il commento con un asserzione sicuramente false:

```
public void metodo() {
    if (flag == true) {
         return;
    }
    assert false;
}
```

10.5.7 Conclusioni

In questo modulo abbiamo raggruppato argomenti che possono sembrare simili, proprio per esplicitarne le differenze. Dopo aver studiato questo capitolo il lettore dovrebbe avere le idee chiare a riguardo. Il concetto di fondo è che l'utilizzo della gestione delle eccezioni è fondamentale per la creazione di applicazioni robuste. Le asserzioni invece rappresentano un comodo meccanismo per testare la robustezza delle nostre applicazioni. La gestione delle eccezioni, non è consigliabile, è obbligatoria! Le asserzioni rappresentano un potente meccanismo per testare la robustezza delle nostre applicazioni. Tuttavia bisogna avere un po' di esperienza per sfruttarne a pieno le potenzialità. Infatti, la progettazione per contratto è un argomento complesso che va studiato a fondo per poter ottenere risultati corretti. Ciononostante anche l'utilizzo dei concetti più semplici come le post-condizioni, possono migliorare le nostre applicazioni.

Riepilogo

In questo modulo, abbiamo dapprima distinto i concetti di eccezione, errore ed asserzione. Poi abbiamo categorizzato le eccezioni e gli errori con una panoramica sulla classi principali. Inoltre abbiamo ulteriormente suddiviso le tipologie di eccezioni in checked ed unchecked. Il meccanismo che è alla base della gestione delle eccezioni è stata presentato, parallelamente alle cinque parole chiave che ne permettono la gestione. I blocchi `try-catch-finally`, permettono di gestire localmente le eccezioni. Le coppie `throw-throws` supportano invece la propagazione (in maniera robusta) delle eccezioni. Abbiamo anche mostrato come creare eccezioni personalizzate. La possibilità di astrarre il concetto di eccezione con gli oggetti e la possibilità di sfruttare il meccanismo di call-stack (propagazione dell'errore), permettono di creare applicazioni contemporaneamente object oriented, semplici e robuste.
Le asserzioni hanno la caratteristica di potere essere abilitate o disabilitate al momento dell'esecuzione del programma. Abbiamo esplicitato ogni tipo di flag che deve essere utilizzato a tal proposito. Abbiamo anche introdotto il loro utilizzo all'interno della progettazione per contratto, introducendo i concetti di pre-condizioni, post-condizioni ed

invarianti. Infine sono stati dati alcuni consigli sui casi in cui è opportuno utilizzare le asserzioni.

Esercizi modulo 10

Esercizio 10.a)
Gestione delle eccezioni e degli errori, Vero o Falso:

1. Ogni eccezione che estende in qualche modo una `ArithmeticException` è una checked excepion
2. Un `Error` si differenzia da una `Exception` perché non può essere lanciato, infatti non estende la classe `Throwable`
3. Il seguente frammento di codice:
```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    System.out.println("Eccezione generica...");
}
finally {
    System.out.println("Finally!");
}
```
produrrà il seguente output:
Divisione per zero...
Eccezione generica..
Finally!
4. Il seguente frammento di codice:
```
int a = 10;
int b = 0;
try {
    int c = a/b;
```

```
        System.out.println(c);
    }
    catch (Exception exc) {
        System.out.println("Eccezione generica...");
    }
    catch (ArithmeticException exc) {
        System.out.println("Divisione per zero...");
    }
    catch (NullPointerException exc) {
        System.out.println("Reference nullo...");
    }
    finally {
        System.out.println("Finally!");
    }
```

produrrà un errore al runtime

5. La parola chiave `throw` permette di lanciare "a mano" solo le sottoclassi di `Exception` che crea il programmatore
6. La parola chiave `throw` permette di lanciare "a mano" solo le sottoclassi di `Exception`
7. Se un metodo fa uso della parola chiave `throw`, affinché la la compilazione abbia buon esito allora nello stesso metodo o deve essere gestita l'eccezione che si vuole lanciare, o il metodo stesso deve utilizzare una clausola `throws`
8. Non è possibile estendere la classe `Error`
9. Se un metodo `m2` fa ovverride di un altro metodo `m2` posto nella superclasse, allora non potrà dichiarare con la clausola `throws` eccezioni nuove che non siano sottoclassi rispetto a quelle che dichiara il metodo `m2`
10. Dalla versione 1.4 di Java, è possibile "includere" in un'eccezione un'altra eccezione

Esercizio 10.b)
Gestione delle asserzioni, Vero o Falso:

1. Se in un'applicazione un'asserzione non viene verificata, si deve parlare di bug
2. Un'asserzione che non viene verificata, provoca il lancio da parte della JVM di un `AssertionError`
3. Le pre-condizioni servono per testare la correttezza dei parametri di metodi pubblici

4. È sconsigliato l'utilizzo di asserzioni laddove si vuole testare la correttezza di dati che sono inseriti da un utente
5. Una post-condizione serve per verificare che al termine di un metodo, sia verificata un'asserzione
6. Un'invariante interna, permette di testare la correttezza dei flussi all'interno dei metodi
7. Un'invariante di classe è una particolare invariante interna che deve essere verificata per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi
8. Un'invariante sul flusso di esecuzione, è solitamente un'asserzione con una sintassi del tipo:
   ```
   assert false;
   ```
9. Non è in nessun modo possibile compilare un programma che fa uso di asserzioni con il jdk 1.3
10. Non è in nessun modo possibile eseguire un programma che fa uso di asserzioni con il jdk 1.3

Soluzioni esercizi modulo 10

Esercizio 10.a)
Gestione delle eccezioni e degli errori, Vero o Falso:

1. **Vero** perché `ArithmeticException` è sottoclasse di `RuntimeException`
2. **Falso**
3. **Falso** produrrà il seguente output:
   Divisione per zero...
   Finally!
4. **Falso** produrrà un errore in compilazione
5. **Falso**
6. **Falso** solo le sottoclassi di `Throwable`
7. **Vero**
8. **Falso**
9. **Vero**
10. **Vero**

Esercizio 10.b)
Gestione delle asserzioni, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere le varie tipologie di eccezioni, errori ed asserzioni (unità 10.1)	☐	
Saper gestire le varie tipologie di eccezioni con i blocchi `try – catch` (unità 10.2)	☐	
Saper creare tipi di eccezioni personalizzate e gestire il meccanismo di propagazione con le parole chiave throw e throws (unità 10.3)	☐	
Capire e saper utilizzare il meccanismo delle asserzioni (unità 10.4)	☐	

Note:

Parte IV “Le librerie fondamentali”

La parte 4 è dedicata all’introduzione delle principali librerie. Restiamo fermamente convinti che la piena autonomia su certi argomenti possa derivare solo dallo studio della documentazione ufficiale. Questa sezione quindi non potrà assolutamente sostituire la documentazione fornita dalla Sun. Tuttavia, si propone di semplificare l’approccio alle librerie fondamentali, senza nessuna pretesa di essere esauriente. In particolare vengono introdotte le classi più importanti, con le quali prima o poi bisognerà fare i conti. Dopo una intensa immersione nel mondo dei thread, saranno introdotti i package `java.lang` e `java.util`. Tra le classi presentate per esempio, la classe `System`, `StringTokenizer`, le classi Wrapper, le classi per internazionalizzare le nostre applicazioni e quelle del framework “Collections”. Introdurremo anche le principali caratteristiche dell’input-output e del networking in Java. Inoltre, esploreremo il supporto che Java offre a due altri linguaggi cardine dell’informatica dei nostri giorni, per la gestione dei dati: l’SQL e l’XML. Inoltre introdurremo le applet, impareremo a creare interfacce grafiche con le librerie AWT e Swing, e gestire gli eventi su di esse. Argomenti come i Thread, le Collections o i package `java.lang` e `java.util` fanno parte anche del programma di certificazione.
Il lettore che avrà studiato attentamente anche questa quarta parte, dovrebbe oramai essere in grado di superare l’esame SCJP, e di avere delle solide basi teoriche su cui basare i propri sforzi di programmazione.

11

Complessità: alta

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper definire multi-threading e multi-tasking (unità 11.1).
2. Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 11.2)
3. Saper creare ed utilizzare thread tramite la classe Thread e l'interfaccia Runnable (unità11.2).
4. Definire cos'è uno scheduler e i suoi comportamenti riguardo le priorità dei thread (unità11.3).
5. Sincronizzare thread (unità11.4).
6. Far comunicare i thread (unità11.5).

11 Gestione dei thread

Questo modulo è dedicato alla gestione della concorrenza dei processi nella programmazione Java. Non capita spesso di dover gestire i thread "a mano" nella programmazione giornaliera, perché, fortunatamente, spesso la tecnologia Java (per esempio nelle tecnologie Web implementate dalla Enterprise Edition), i thread vengono gestiti automaticamente. Tuttavia, riteniamo indispensabile la conoscenza di tali concetti, per padroneggiare la programmazione Java, e per comprendere molti concetti presenti nelle altre librerie. Per esempio, nella documentazione troveremo spesso la dicitura che una classe della libreria standard è thread-safe (per esempio la classe `java.util.Vector` che sarà trattata nel prossimo modulo). Questo modulo è quindi molto importante.

Avvertiamo il lettore che questo modulo è ben più complesso degli altri e richiede un impegno ed una concentrazione particolari. Chi troverà l'argomento troppo ostico potrà eventualmente tornare a rileggerlo in un secondo momento quando si sentirà pronto. Tuttavia, non consigliamo a nessuno di saltare completamente lo studio di questo modulo. Infatti, anche se riguarda un argomento tanto avanzato e tutto

sommato poco utilizzato, come già asserito, rappresenta la base di molti concetti chiave che si incontreranno in futuro nello studio di Java.

11.1 Introduzione ai thread

I thread, rappresentano il mezzo mediante il quale, Java fa eseguire un'applicazione da più Virtual Machine contemporaneamente, allo scopo di ottimizzare i tempi del runtime. Ovviamente, si tratta di un'illusione: per ogni programma solitamente esiste un'unica JVM ed un'unica CPU. Ma la CPU può eseguire codice tramite più processi all'interno della gestione della JVM per dare l'impressione di avere più processori.
L'esperienza ci ha insegnato che l'apprendimento di un concetto complesso come la gestione dei thread in Java, richiede un approccio graduale sia per quanto riguarda le definizioni, sia per quanto riguarda le tecniche di utilizzo. Tale convincimento, oltre ad avere una natura empirica, nasce dall'esigenza di dover definire un thread quale oggetto della classe `Thread`, cosa che può portare il lettore facilmente a confondersi.
Didatticamente inoltre, è spesso utile adoperare schemi grafici per aiutare il discente nella comprensione. Ciò risulta meno fattibile quando si cerca di spiegare il comportamento di più thread al runtime, poiché un schema statico non è sufficiente. Per aiutare il lettore nella comprensione degli esempi presentati in questo tutorial, viene quindi fornita una semplice applet (ma bisogna collegarsi al sito http://www.claudiodesio.com) che permette di navigare tra le schematizzazioni grafiche dei momenti topici di ogni esempio.

11.1.1 Definizione provvisoria di Thread

Quando lanciamo un'applicazione Java, vengono eseguite le istruzioni contenute in essa in maniera sequenziale, a partire dal codice del metodo `main()`. Spesso, soprattutto in fase di debug, allo scopo di simulare l'esecuzione dell'applicazione, il programmatore immagina un cursore che scorre sequenzialmente le istruzioni, magari, simulando il suo movimento con un dito che punta sul monitor. Un tool di sviluppo che dispone di un debugger grafico invece, evidenzia concretamente questo cursore. Consapevoli che il concetto risulterà familiare ad un qualsiasi programmatore, possiamo per il momento identificare un thread proprio con questo cursore immaginario. In tal modo, affronteremo le difficoltà dell'apprendimento in maniera graduale.

11.1.2 Cosa significa “multi-threading”

L'idea di base è semplice: immaginiamo di lanciare un'applicazione Java. Il nostro cursore immaginario, scorrerà ed eseguirà sequenzialmente le istruzioni partendo dal codice del metodo `main()`. Quindi, a runtime, esiste almeno un thread in esecuzione, ed il suo compito è quello di eseguire il codice, seguendo il flusso definito dall'applicazione stessa.
Per “multi-threading” si intende il processo che porterà un'applicazione, a definire più di un thread, assegnando ad ognuno compiti da eseguire parallelamente. Il vantaggio che può portare un'applicazione multi-threaded, è relativo soprattutto alle prestazioni della stessa. Infatti i thread possono “dialogare” allo scopo di spartirsi nella maniera ottimale l'utilizzo delle risorse del sistema. D'altronde, l'esecuzione parallela di più thread all'interno della stessa applicazione è vincolata all'architettura della macchina su cui gira. In altre parole, se la macchina ha un unico processore, in un determinato momento `x`, può essere in esecuzione un unico thread. Ciò significa che un'applicazione non multi-threaded che richiede in un determinato momento alcuni input (da un utente, una rete, da un database etc...) per poter proseguire nell'esecuzione, quando si trova in stato di attesa non può eseguire nulla. Un'applicazione multi-threaded invece, potrebbe eseguire altro codice mediante un altro thread, “avvertito” dal thread che è in stato di attesa.
Solitamente, il multi-threading, è una caratteristica dei sistemi operativi (per esempio Unix), piuttosto che dei linguaggi di programmazione. La tecnologia Java, tramite la Virtual Machine, ci offre uno strato d'astrazione per poter gestire il multi-threading direttamente dal linguaggio. Altri linguaggi (come il C/C++), solitamente sfruttano le complicate librerie del sistema operativo per gestire il multi-threading, lì dove possibile. Infatti tali linguaggi, non essendo stati progettati per lo scopo, non supportano un meccanismo chiaro per gestire i thread.

Il multi-threading non deve essere confuso con il multi-tasking. Possiamo definire “task” i processi che possono essere definiti “pesanti”, per esempio Word ed Excel. In un sistema operativo che supporta il multi-tasking, è possibile lanciare più task contemporaneamente. La precedente affermazione, può risultare scontata per molti lettori, ma negli anni '80 l'”Home Computer”, utilizzava spesso DOS, un sistema chiaramente non multi-tasking. I task hanno comunque spazi di indirizzi separati, e la comunicazione fra loro è limitata. I thread possono essere definiti come processi “leggeri”, che condividono lo stesso spazio degli indirizzi e lo stesso processo pesante in cooperazione. I thread hanno quindi la

caratteristica fondamentale di poter "comunicare" al fine di ottimizzare l'esecuzione dell'applicazione in cui sono definiti.

In Java, i meccanismi della gestione dei thread, risiedono essenzialmente:

1. Nella classe `Thread` e l' interfaccia `Runnable` (package `java.lang`)
2. Nella classe `Object` (ovviamente package `java.lang`)
3. Nella JVM e nella keyword `synchronized`

11.2 La classe `Thread` e la dimensione temporale

Come abbiamo precedentemente affermato, quando si avvia un'applicazione Java, c' è almeno un thread in esecuzione, appositamente creato dalla JVM per eseguire il codice dell'applicazione. Nel seguente esempio introdurremo la classe `Thread` e vedremo che anche con un unico thread è possibile creare situazioni interessanti.

```
1   public class ThreadExists {
2     public static void main(String args[]) {
3       Thread t = Thread.currentThread();
4       t.setName("Thread principale");
5       t.setPriority(10);
6       System.out.println("Thread in esecuzione: " + t);
7       try {
8         for (int n = 5; n > 0; n--) {
9           System.out.println("" + n);
10          t.sleep(1000);
11        }
12      }
13      catch (InterruptedException e) {
14        System.out.println("Thread interrotto");
15      }
16    }
17  }
```

Segue l'output dell'applicazione:

```
C:\TutorialJavaThread\Code>java ThreadExists
```

```
Thread in esecuzione: Thread[Thread principale,10,main]
5
4
3
2
1
```

11.2.1 Analisi di `ThreadExists`

Questa semplice classe produce un risultato tutt'altro che trascurabile: la gestione del tempo. La durata dell'esecuzione del programma è infatti quantificabile in circa 5 secondi. In questo testo, il precedente è il primo esempio che gestisce in qualche modo la durata dell'applicazione stessa. Analizziamo il codice nei dettagli:
Alla riga 3, viene chiamato il metodo statico `currentThread()` della classe `Thread`. Questo restituisce l'indirizzo dell'oggetto `Thread` che sta eseguendo l'istruzione, che viene assegnato al reference `t`. Questo passaggio è particolarmente delicato: abbiamo identificato un thread con il "cursore immaginario" che processa il codice. Inoltre abbiamo anche ottenuto un reference a questo cursore. Una volta ottenuto un reference, è possibile gestire il thread, controllando così l'esecuzione (temporale) dell'applicazione!

> **Notiamo la profonda differenza tra la classe `Thread`, e tutte le altre classi della libreria standard. La classe `Thread` astrae un concetto che non solo è dinamico, ma addirittura rappresenta l'esecuzione stessa dell'applicazione! Il `currentThread` infatti non è l'"oggetto corrente" , che è solitamente individuato dalla parola chiave `this`, ma l'oggetto (thread corrente) che esegue l'oggetto corrente. Potremmo affermare che un oggetto `Thread` si trova in un'altra dimensione rispetto agli altri oggetti: la dimensione temporale.**

Continuiamo con l'analisi della classe `ThreadExists`. Alle righe 4 e 5, scopriamo che è possibile non solo assegnare un nome al thread, ma anche una priorità. La scala delle priorità dei thread in Java, va dalla priorità minima `1` alla massima `10`, e la priorità di default è `5`.

> **Come vedremo più avanti, il concetto di priorità NON è la chiave per gestire i thread. Infatti, limitandoci alla sola gestione delle priorità, la nostra applicazione multi-threaded potrebbe comportarsi in maniera**

**differente su sistemi diversi. Pensiamo solo al fatto che non tutti i sistemi operativi utilizzano una scala da 1 a 10 per le priorità, e che per esempio Unix e Windows hanno
thread scheduler con filosofie completamente differenti.**

Alla riga 6, viene stampato l'oggetto `t` (ovvero `t.toString()`). Dall'output notiamo che vengono stampate informazioni sul nome e la priorità del thread, oltre che sulla sua appartenenza al gruppo dei thread denominato `main`. I thread infatti appartengono a dei `ThreadGroup`, ma non ci occuperemo di questo argomento in dettaglio, (consultare la documentazione ufficiale). Tra la riga 7 e la riga 12 viene dichiarato un blocco `try` contenente un ciclo `for` che esegue un conto alla rovescia da `5` ad `1`. Tra una stampa di un numero ed un altra c'è una chiamata al metodo `sleep()` sull'oggetto `t`, a cui viene passato l'intero `1000`. In questo modo il thread che esegue il codice, farà un pausa di un secondo (1000 millisecondi) tra la stampa di un numero ed un altro. Tra la riga 13 e la riga 15 viene definito il blocco `catch` che gestisce una `InterruptedException`, che il metodo `sleep()` dichiara nella sua clausola `throws`. Questa eccezione scatterebbe nel caso in cui il thread non riuscisse ad eseguire "il suo codice" perchè stoppato da un altro thread. Nel nostro esempio però, non vi è che un unico thread, e quindi la gestione dell'eccezione non ha molto senso. Nel prossimo paragrafo vedremo come creare altri thread, sfruttando il thread principale.

11.2.2 L'interfaccia Runnable e la creazione dei thread

Per avere più thread basta istanziarne altri dalla classe `Thread`. Nel prossimo esempio noteremo che quando si istanzia un oggetto `Thread`, bisogna passare al costruttore un'istanza di una classe che implementa l'interfaccia `Runnable`. I questo modo infatti, il nuovo thread, quando sarà fatto partire (mediante la chiamata al metodo `start()`), andrà ad eseguire il codice del metodo `run()` dell'istanza associata. L' interfaccia `Runnable` quindi, richiede l' implementazione del solo metodo `run()` che definisce il comportamento di un thread, e l' avvio di un thread si ottiene con la chiamata del metodo `start()`. Dopo aver analizzato il prossimo esempio, le idee dovrebbero risultare più chiare.

```
1   public class ThreadCreation implements Runnable {
2     public ThreadCreation () {
3       Thread ct = Thread.currentThread();
4       ct.setName("Thread principale");
5       Thread t = new Thread(this, "Thread figlio");
```

```
6       System.out.println("Thread attuale: " + ct);
7       System.out.println("Thread creato: " + t);
8       t.start();
9       try {
10        Thread.sleep(3000);
11      }
12      catch (InterruptedException e) {
13        System.out.println("principale interrotto");
14      }
15      System.out.println("uscita Thread principale");
16    }
17    public void run() {
18      try {
19        for (int i = 5; i > 0; i--) {
20          System.out.println("" + i);
21          Thread.sleep(1000);
22        }
23      }
24      catch (InterruptedException e) {
25        System.out.println("Thread figlio interrotto");
26      }
27      System.out.println("uscita Thread figlio");
28    }
29    public static void main(String args[]) {
30      new ThreadCreation();
31    }
32  }
```

Segue l'output del precedente codice:

```
C:\TutorialJavaThread\Code>java ThreadCreation
Thread attuale: Thread[Thread principale,5,main]
Thread creato: Thread[Thread figlio,5,main]
5
4
3
uscita Thread principale
2
1
```

uscita Thread figlio

11.2.3 Analisi di ThreadCreation

Nel precedente esempio oltre al thread principale, ne è stato istanziato un secondo. La durata dell'esecuzione del programma è anche in questo caso quantificabile in circa 5 secondi. Analizziamo il codice nei dettagli:

A supporto della descrizione dettagliata del runtime dell'esempio, viene fornita un'applet che schematizza graficamente la situazione dei due thread, nei momenti topici del runtime. L'applet è disponibile all'indirizzo http://www.claudiodesio.com/java/TutorialJavaThread/Applet/ThreadCreation.html. Gli schemi vengono riportati in copia anche tra queste pagine per semplicità. Tutte le figure fanno riferimento alla legenda della Fig.11.0.

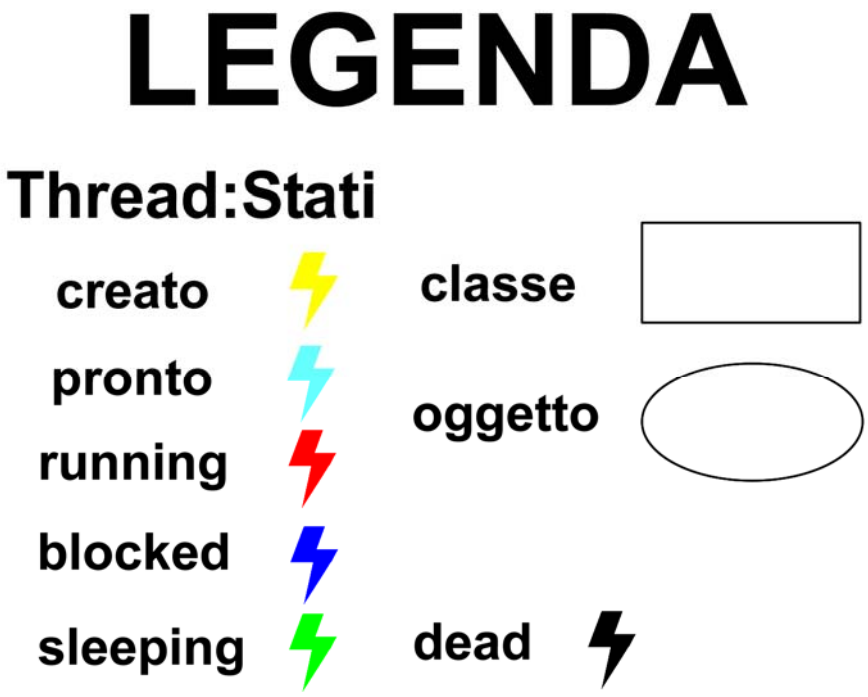


Figura 11.0 – "Elementi della notazione degli schemi"

L'applicazione al runtime viene eseguita a partire dal metodo `main()` alla riga 29. Alla riga 30 viene istanziato un oggetto della classe `ThreadCreation` poi il nostro cursore immaginario si sposta ad eseguire il costruttore dell'oggetto appena creato, alla riga 3. Qui il nostro cursore immaginario (il thread corrente), ottiene un reference `ct`. Notiamo che "`ct` esegue `this`" . A `ct` viene poi assegnato il nome "thread principale". Alla riga 5 viene finalmente istanziato un altro thread, dal thread corrente `ct` che sta eseguendo la riga 5 [Fig. 1 dell'applet e Fig. 11.1 di questo manuale].

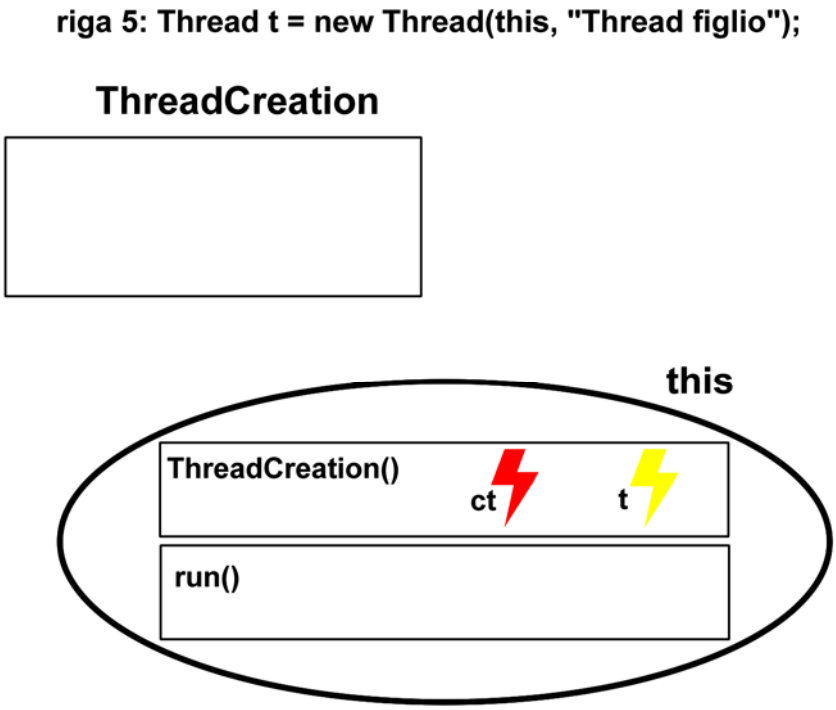


Figura 11.1 – "Istanza del Thread"

Viene utilizzato un costruttore che prende in input due parametri. Il primo (`this`) è un oggetto `Runnable` (ovvero un'istanza di una classe che implementa l'interfaccia `Runnable`), il secondo è ovviamente il nome del thread. L'oggetto `Runnable` contiene il metodo `run()`, che diventa l'obiettivo dell'esecuzione del thread `t`. Quindi, mentre per il thread principale l'obiettivo dell'esecuzione è scontato, per i thread che vengono istanziati bisogna specificarlo passando al costruttore un oggetto `Runnable`. Alle righe 6 e 7 vengono stampati messaggi descrittivi dei due thread. Alla riga 8 viene finalmente fatto partire il thread `t` mediante il metodo `start()`.

La chiamata al metodo `start()` per eseguire il metodo `run()`, fa sì che il thread `t` vada prima o poi ad eseguire il metodo `run()`. Invece, una eventuale chiamata del metodo `run()`, non produrrebbe altro che una normale esecuzione dello stesso da parte del thread principale `ct` non ci sarebbe multi-threading.

La "partenza" (tramite l'invocazione del metodo `start()`) di un thread **non** implica che il thread inizi immediatamente ad eseguire il suo codice, ma solo che è stato reso eleggibile per l'esecuzione [Fig. 11.2].

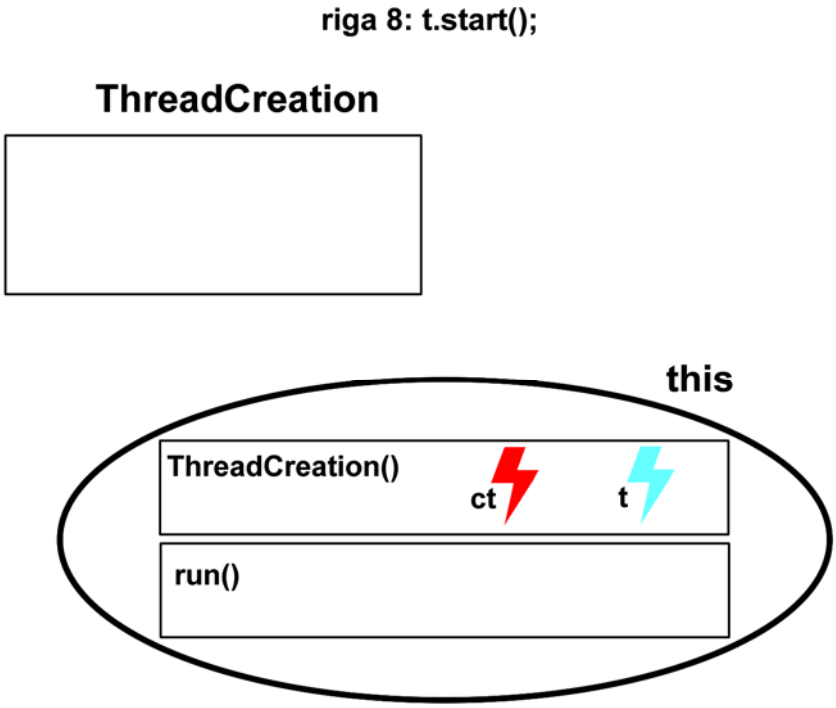


Figura 11.2 – "Chiamata al metodo `start()`"

Quindi, il thread `ct`, dopo aver istanziato e reso eleggibile per l'esecuzione il thread `t`, continua nella sua esecuzione fino a quando, giunto ad eseguire la riga 10, incontra il metodo `sleep()` che lo ferma per 3 secondi.

Notiamo come il metodo `sleep()` sia statico. Infatti, viene mandato "a dormire" il thread che esegue il metodo.

A questo punto il processore è libero dal thread `ct`, e viene utilizzato dal thread `t`, che finalmente può eseguire il metodo `run()`. Ecco che allora il thread `t` va ad eseguire un ciclo, che come nell'esempio precedente, realizza un conto alla rovescia, facendo pausa da un secondo. Esaminando l'output verifichiamo che il codice fa sì che, il thread `t` stampi il `5` [fig. 11.3], faccia una pausa di un secondo [fig. 11.4], stampi `4`, pausa di un secondo, stampi `3`, pausa di un secondo.

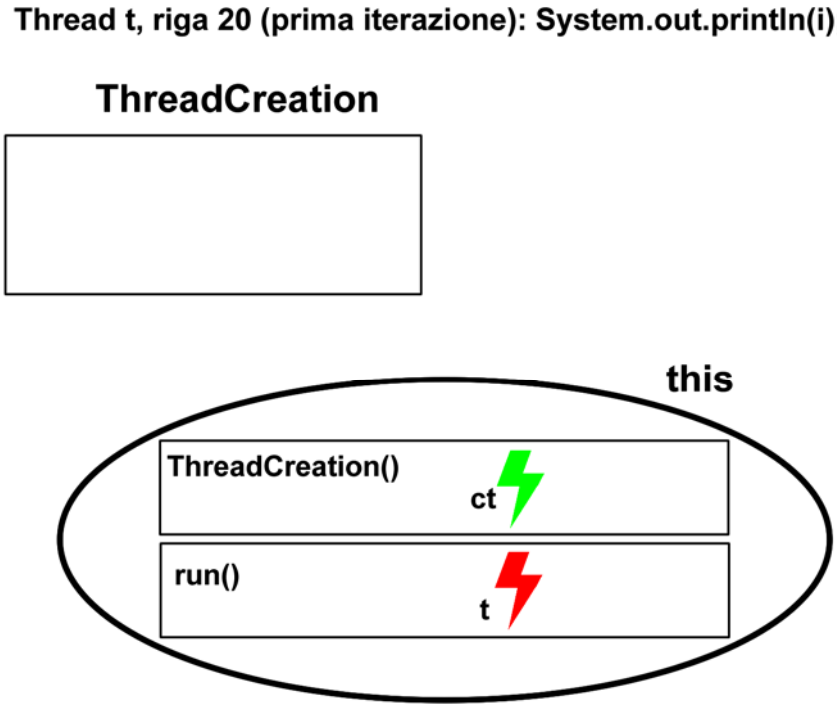


Figura 11.3 – "Prima iterazione, stampa 5"

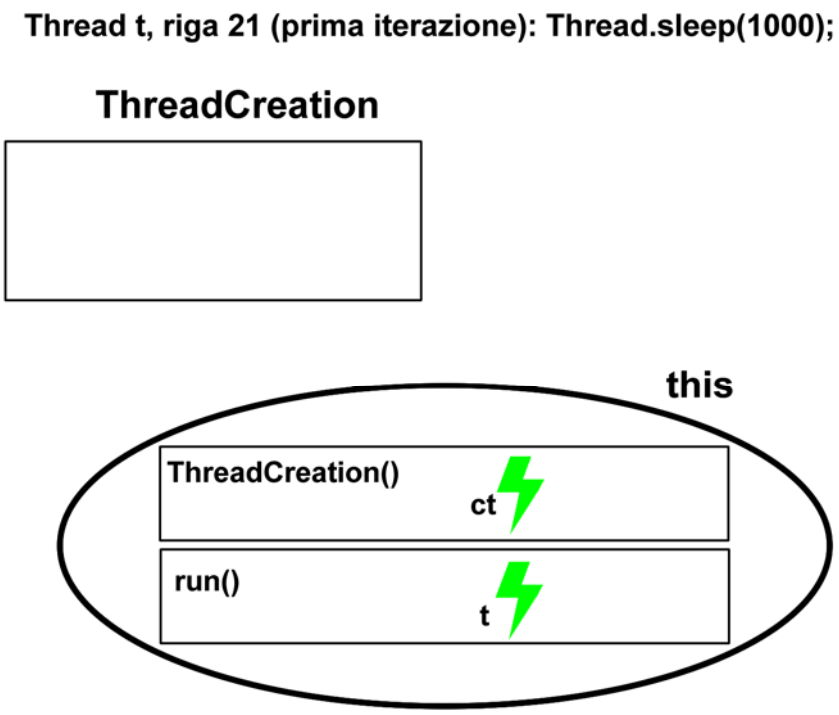


Figura 11.4 – "Prima iterazione, tutti i thread in pausa"

Poi si risveglia il thread `ct` che stampa la frase "uscita thread principale" [fig. 11.5], e poi "muore".

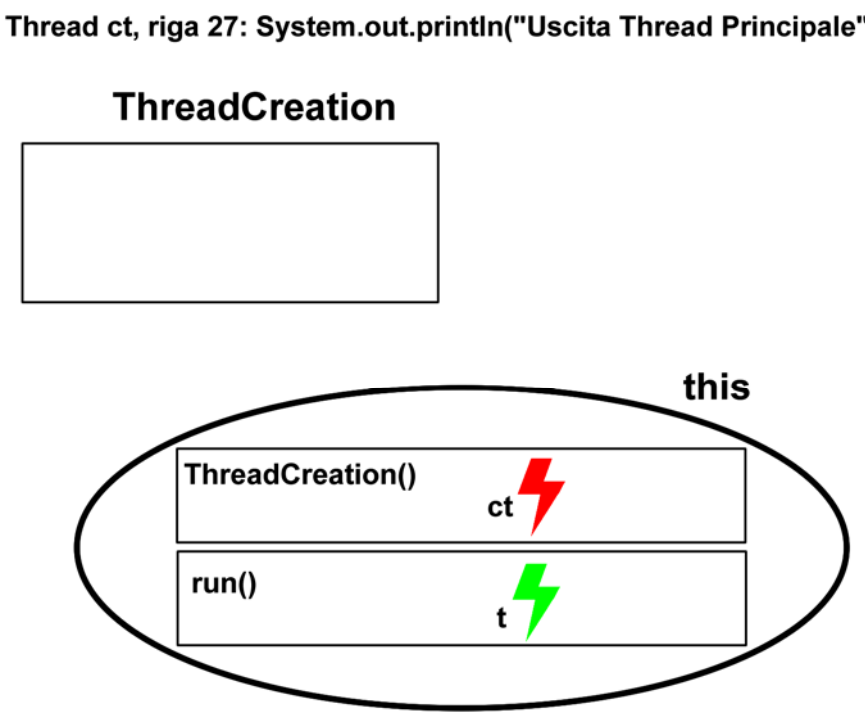


Figura 11.5 – "Uscita thread principale"

Quasi contemporaneamente viene stampato `2` [fig. 11.6], pausa di un secondo, stampa di `1`, pausa di un secondo, stampa di "uscita thread figlio".

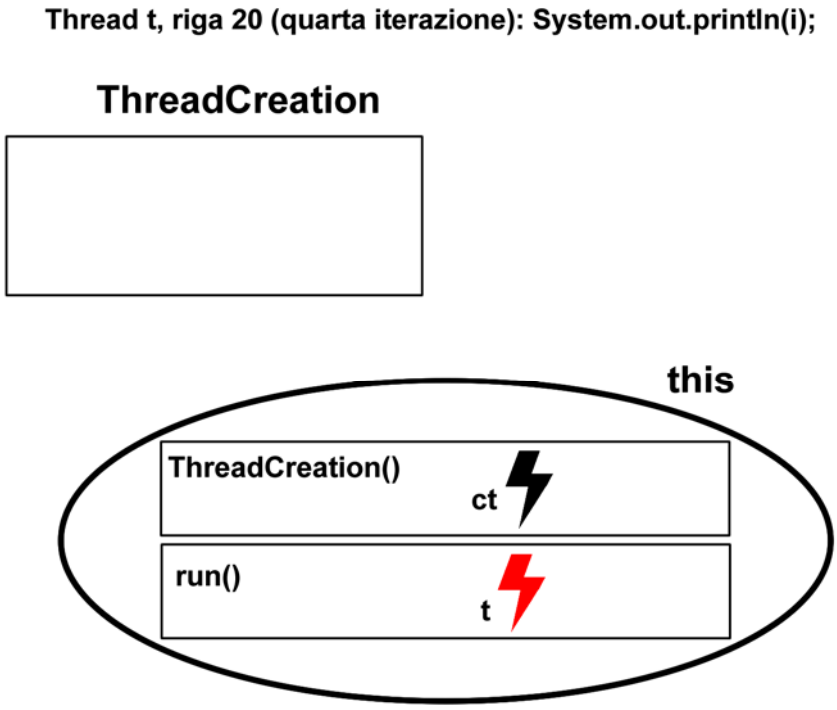


Figura 11.6 – "Quarta iterazione"

Nell'esempio quindi, l'applicazione ha un ciclo di vita superiore a quello del thread principale, grazie al thread creato.

11.2.4 La classe `Thread` e la creazione dei thread

Abbiamo appena visto come un thread creato deve eseguire codice di un oggetto istanziato da una classe che implementa l'interfaccia `Runnable` (l'oggetto `Runnable`). Ma la classe `Thread` stessa implementa l'interfaccia `Runnable`, fornendo un implementazione vuota del metodo `run()`. È quindi possibile fare eseguire ad un thread il metodo `run()` definito all'interno dello stesso oggetto thread. Per esempio:

```
1   public class CounterThread extends Thread {
2     public void run() {
3       for (int i = 0; i<10; ++i)
4         System.out.println(i);
5     }
6   }
```

è possibile istanziare un thread senza specificare l'oggetto `Runnable` al costruttore e farlo partire con il solito metodo `start()`:

```
1   CounterThread thread = new CounterThread ();
2   thread.start();
```

E' anche possibile (ma non consigliato) creare un thread che utilizza un `CounterThread` come oggetto `Runnable`:

```
Thread t = new Thread(new CounterThread());
t.start();
```

Sicuramente la strategia di fare eseguire il metodo `run()` all'interno dell'oggetto thread stesso, è più semplice rispetto a quella vista nel paragrafo precedente. Tuttavia ci sono almeno tre buone ragioni per preferire il passaggio di un oggetto `Runnable`:
In Java una classe non può estendere più di una classe alla volta. Quindi, implementando l'interfaccia `Runnable`, piuttosto che estendere `Thread`, permetterà di utilizzare l'estensione per un altra classe.
Solitamente un oggetto della classe `Thread` non dovrebbe possedere variabili d'istanza private che rappresentano i dati da gestire. Quindi il metodo `run()` nella sottoclasse di `Thread`, non potrà accedere, o non potrà accedere in una maniera "pulita", a tali dati.
Dal punto di vista della programmazione object oriented, una sottoclasse di `Thread` che definisce il metodo `run()`, combina due funzionalità poco relazionate tra loro: il supporto del multi–threading ereditato dalla classe `Thread`, e l'ambiente esecutivo fornito dal metodo `run()`. Quindi in questo caso l'oggetto creato è un thread, ed è associato con se stesso, e questa non è una soluzione molto object oriented.

11.3 Priorità, scheduler e sistemi operativi

Abbiamo visto come il metodo `start()` chiamato su di un thread non implichi che questo inizi immediatamente ad eseguire il suo codice (contenuto nel metodo `run()` dell'oggetto associato). In realtà la JVM, definisce un thread scheduler, che si occuperà di decidere in ogni momento quale thread deve trovarsi in esecuzione. Il problema che la JVM stessa è un software che gira su di un determinato sistema operativo, e la sua implementazione, dipende dal sistema. Quando si gestisce il multi-threading ciò può apparire evidente come nel prossimo esempio. Infatti, lo scheduler della JVM, deve

comunque rispettare la filosofia dello scheduler del sistema operativo, e questa può cambiare notevolmente tra sistema e sistema. Prendiamo in considerazione due tra i più importanti sistemi attualmente in circolazione: Unix e Windows (qualsiasi versione). Il seguente esempio produce output completamente diversi, su i due sistemi:

```
1   public class Clicker implements Runnable {
2     private int click = 0;
3     private Thread t;
4     private boolean running = true;
5     public Clicker(int p) {
6       t = new Thread(this);
7       t.setPriority(p);
8     }
9     public int getClick(){
10      return click;
11    }
12    public void run() {
13      while (running) {
14        click++;
15      }
16    }
17    public void stopThread() {
18      running = false;
19    }
20    public void startThread() {
21      t.start();
22    }
23  }
24  public class ThreadRace {
25    public static void main(String args[]) {
26      Thread.currentThread().setPriority(
          Thread.MAX_PRIORITY);
27      Clicker hi = new Clicker(Thread.NORM_PRIORITY +
2);
28      Clicker lo = new Clicker(Thread.NORM_PRIORITY -
2);
29      lo.startThread();
30      hi.startThread();
31      try {
```

```
32          Thread.sleep(10000);
33        }
34        catch (Exception e){}
35        lo.stopThread();
36        hi.stopThread();
37        System.out.println(lo.getClick()+" vs." +
            hi.getClick());
38      }
39    }
```

Segue l'output su Sun Solaris 8**:**

```
solaris% java ThreadRace
0 vs. 1963283920
```

Vari Output Windows 2000**:**

```
C:\TutorialJavaThread\Code>java ThreadRace
15827423 vs. 894204424
C:\TutorialJavaThread\Code>java ThreadRace
32799521 vs. 887708192
C:\TutorialJavaThread\Code>java ThreadRace
15775911 vs. 890338874
C:\TutorialJavaThread\Code>java ThreadRace
15775275 vs. 891672686
```

11.3.1 Analisi di ThreadRace

Il precedente esempio è composto da due classi: `ThreadRace` e `Clicker`. `ThreadRace` contiene il metodo `main()` e rappresenta quindi la classe principale. Immaginiamo di lanciare l'applicazione, (è possibile lanciare l'applet che si trova all'indirizzo http://www.claudiodesio.com/java/TutorialJavaThread/Applet/ThreadRace.html): Alla riga 27 , viene assegnata al thread corrente, la priorità massima tramite la costante statica intera della classe `Thread MAX_PRIORITY`, che ovviamente vale 10. Alle righe 28 e 29, vengono istanziati due oggetti dalla classe `Clicker`, `hi` e `lo`, ai cui costruttori vengono passati i valori interi `7` e `3`. Gli oggetti `hi` e `lo`, tramite costruttori creano due thread associati ai propri metodi `run()` rispettivamente con priorità `7` e priorità `3`. Il thread principale, continua nella sua esecuzione chiamando su entrambi gli

oggetti `hi` e `lo`, il metodo `startThread()`, che ovviamente rende eleggibili per l'esecuzione i due thread a priorità `7` e `3` nei rispettivi oggetti [Fig. 11.7].

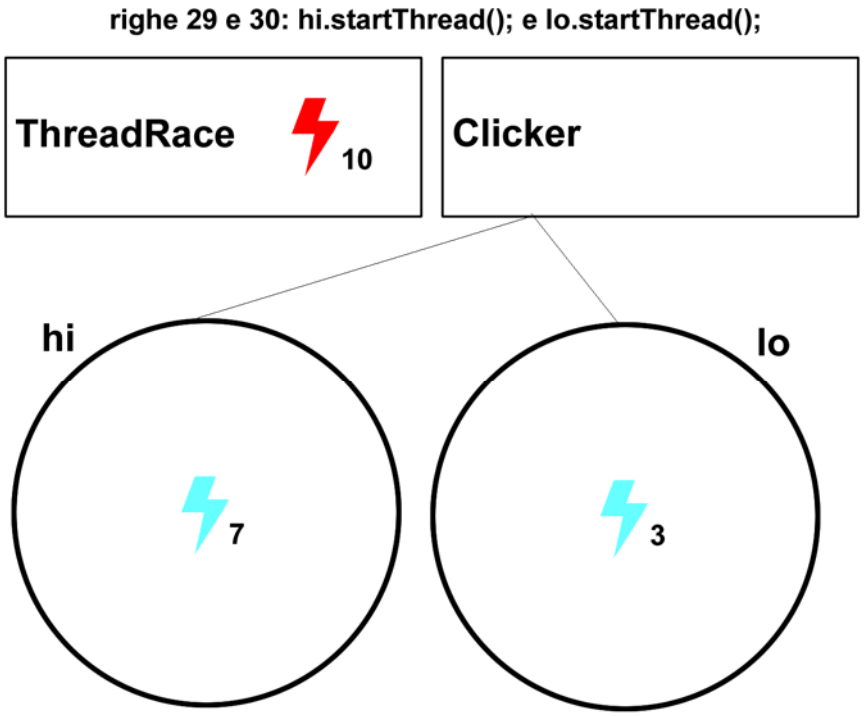


Figura 11.7 – "Start dei thread"

Alla riga 33, il thread principale a priorità `10` va a "dormire" per una decina di secondi, lasciando disponibile la CPU agli altri due thread [fig. 11.8].

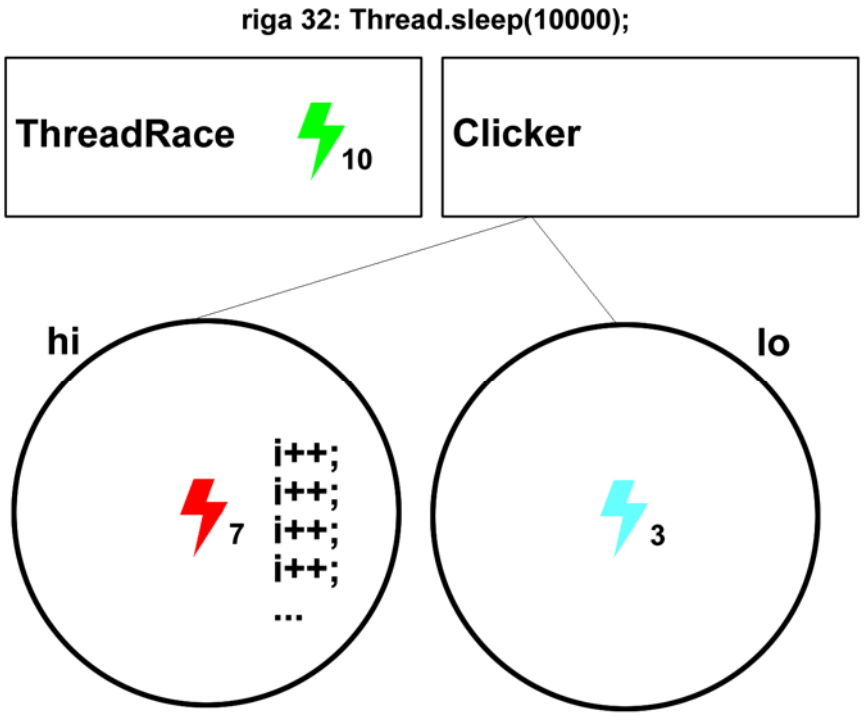


Fig. 11.8 – "Il thread principale si 'addormenta'"

In questo momento dell'esecuzione dell'applicazione che lo scheduler avrà un comportamento dipendente dalla piattaforma.

11.3.2 Comportamento Windows (Time-Slicing o Round-Robin scheduling):

Un thread può trovarsi in esecuzione solo per un certo periodo di tempo, poi deve lasciare ad altri thread la possibilità di essere eseguiti. Ecco che allora l'output di Windows evidenzia che entrambi i thread hanno avuto la possibilità di eseguire codice. Il thread a priorità `7`, ha avuto a disposizione per molto più tempo il processore, rispetto al thread a priorità `3`.
Tale comportamento è però non deterministico, e quindi l'output prodotto, cambierà anche radicalmente ad ogni esecuzione dell'applicazione.

11.3.3 Comportamento Unix (Preemptive scheduling)

Un thread in esecuzione, può uscire da questo stato solo nelle seguenti situazioni:

1. Viene chiamato un metodo di scheduling come `wait()` o `suspend()`
2. Viene chiamato un metodo di blocking, come quelli dell' I/O

3. Può essere "buttato fuori" dalla CPU da un altro thread a priorità più alta che diviene eleggibile per l'esecuzione
4. Termina la sua esecuzione (il suo metodo `run()`).

Quindi il thread a priorità 7, ha occupato la CPU, per tutti i 10 secondi che il thread principale (a priorità 10), è in pausa. Quando quest'ultimo si risveglia rioccupa di forza la CPU.
Su entrambi i sistemi poi l'applicazione continua con il thread principale che, chiamando il metodo `stopThread()` (righe 35 e 36) su entrambi gli oggetti `hi` e `lo`, setta le variabili `running` a `false` [fig. 11.9 e fig.11.10].

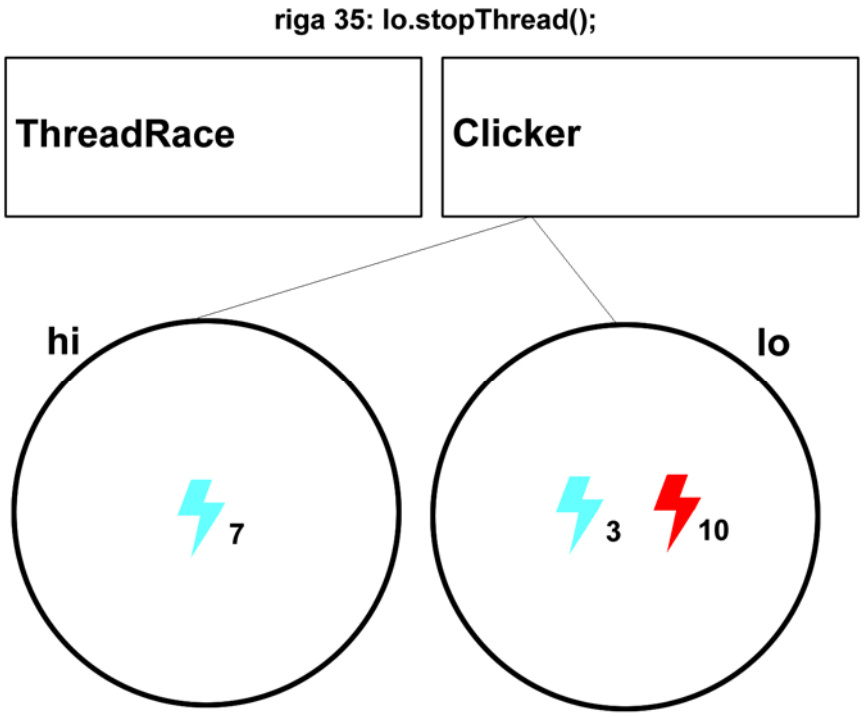


Figura 11.9 – "Stop del thread lo"

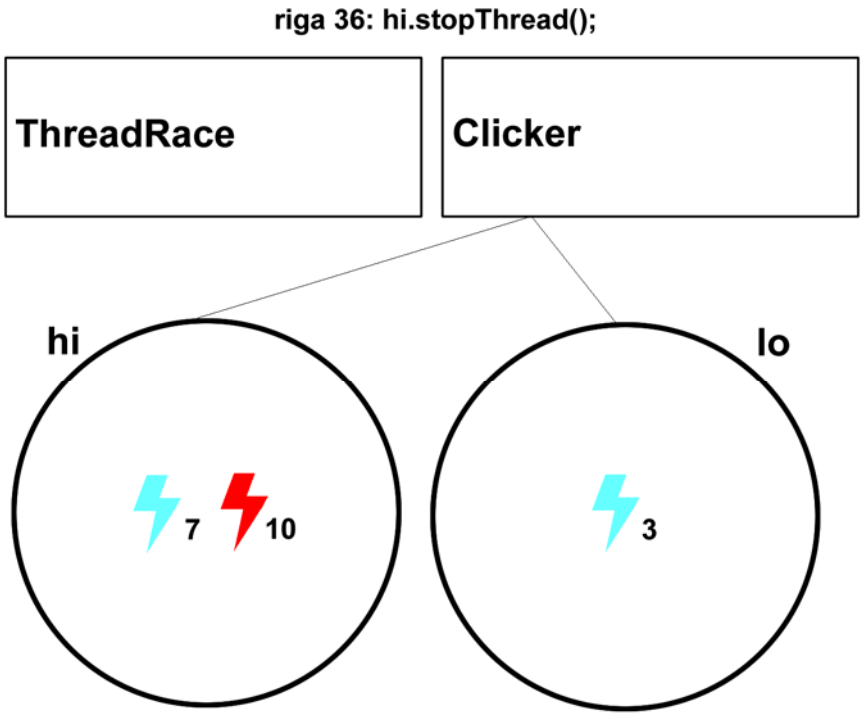


Figura 11.10 – "Stop del thread hi"

Il thread principale termina la sua esecuzione stampando il "risultato finale" [fig. 11.11 e 11.12].

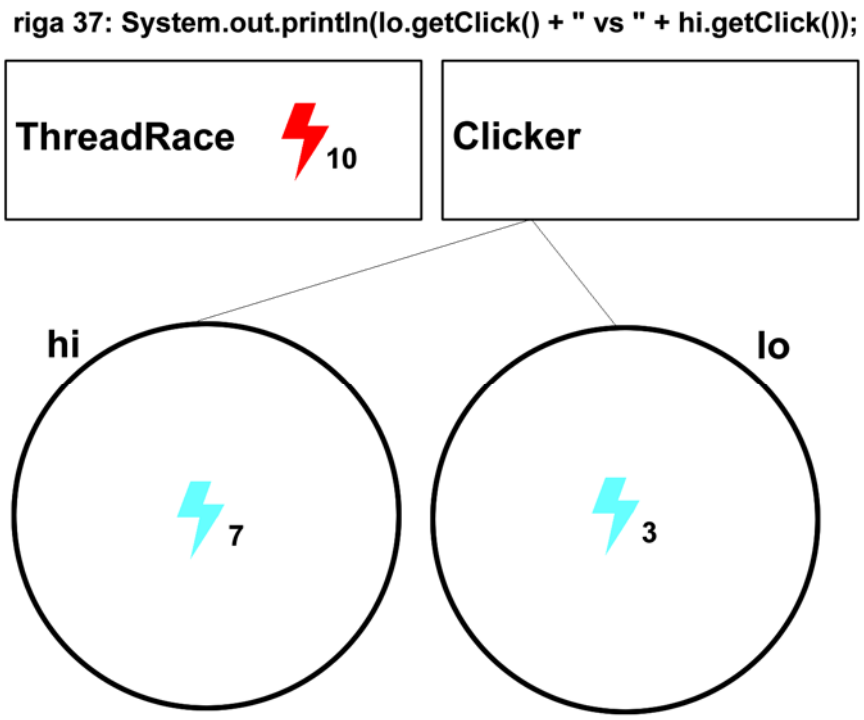


Figura 11.11 – "Stampa risultato finale"

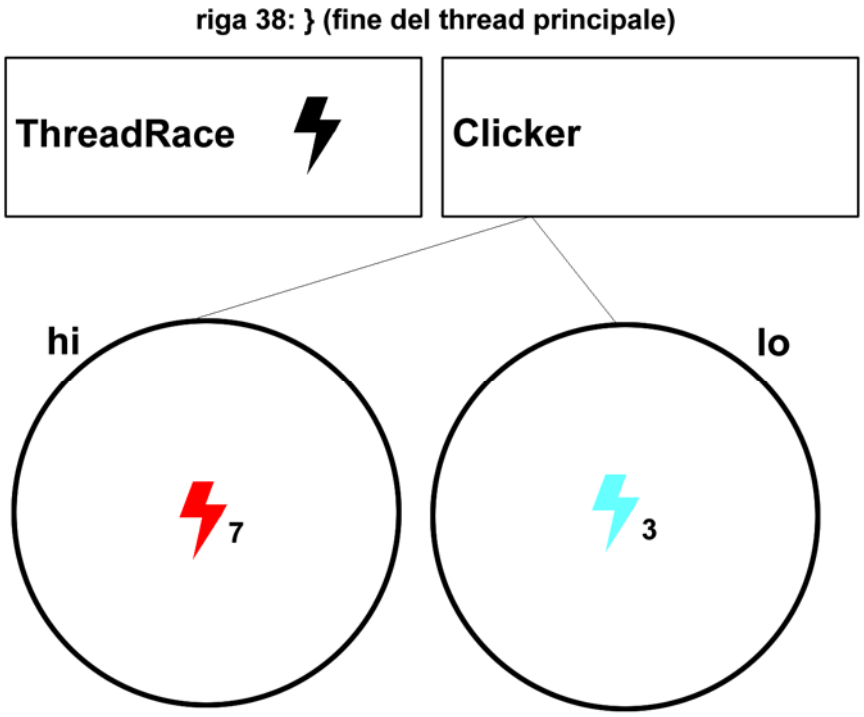


Figura 11.12 – “Fine del thread principale”

A questo punto parte il thread a priorità 7, la cui esecuzione si era bloccata alla riga 13 o 14 o 15. Se riparte dalla riga 14, la variabile `click` viene incrementata un'altra ultima volta, senza però influenzare l'output dell'applicazione. Quando si troverà alla riga 13, la condizione del ciclo `while` non verrà verificata, e quindi il thread a priorità 7 terminerà la sua esecuzione [fig. 11.13]. Stessa sorte toccherà al thread a priorità 3 [fig. 11.14].

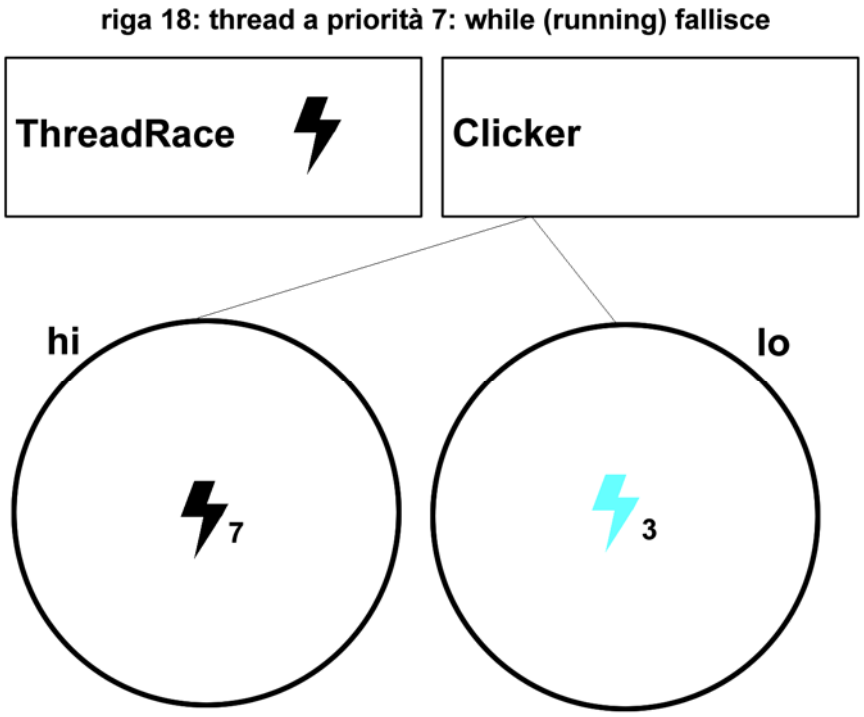


Figura 11.13 – “Termina il thread hi”

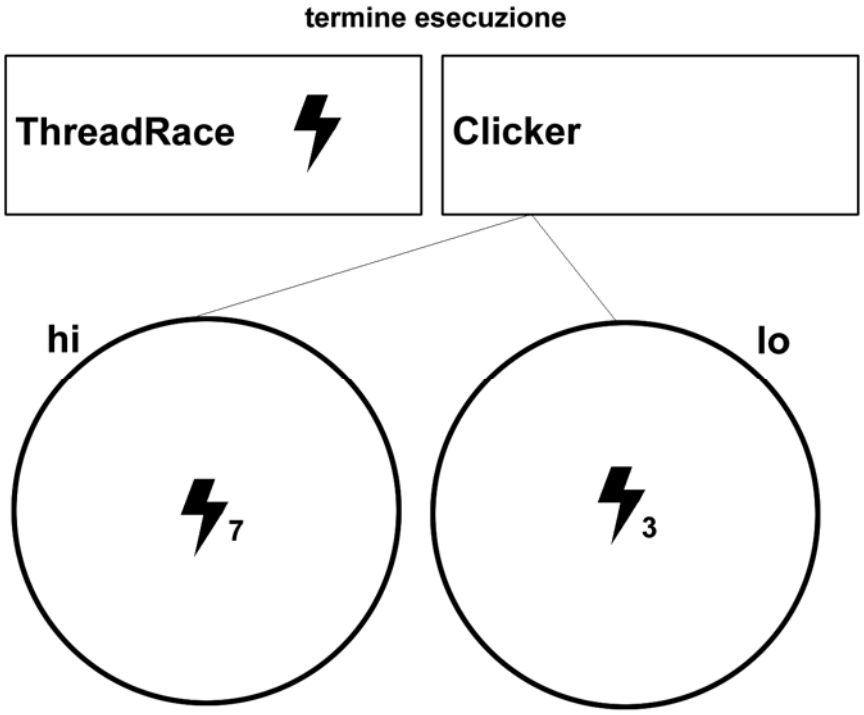


Figura 11.14 – "Termina il thread lo"

11.4 Thread e Sincronizzazione

Abbiamo sino ad ora identificato un thread come un cursore immaginario. Per quanto sia utile poter pensare ad un thread in questi termini, riteniamo il momento maturo per poter dare una definizione più "scientifica" di thread.

Definizione: Un thread è un **processore virtuale**, che esegue **codice** su determinati **dati**.

Nell'esempio precedente `ThreadRace`, i due thread creati (quelli a priorità 7 e 3), eseguono lo stesso codice (il metodo `run()` della classe `Clicker`), utilizzando dati diversi (le variabili `lo.click` ed `hi.click`). Quando però due o più thread necessitano contemporaneamente dell'accesso ad una fonte di dati condivisa, bisogna che accedano ai dati uno alla volta, cioè i loro metodi vanno sincronizzati (`synchronized`). Consideriamo il seguente esempio:

```
1   class CallMe {
2     /*synchronized*/ public void call(String msg) {
3       System.out.print("[" + msg);
```

```
4         try {
5           Thread.sleep(1000);
6         }
7         catch (Exception e){}
8         System.out.println("]");
9       }
10    }

11    class Caller implements Runnable {
12      private String msg;
13      private CallMe target;
14      public Caller(CallMe t, String s) {
15        target = t;
16        msg = s;
17        new Thread(this).start();
18      }
19      public void run() {
20        //synchronized(target){
21        target.call(msg);
22        //}
23      }
24    }

25    public class Synch {
26      public static void main(String args[]) {
27        CallMe target = new CallMe();
28        new Caller(target, "Hello");
29        new Caller(target, "Synchronized");
30        new Caller(target, "World");
31      }
32    }
```

Output senza sincronizzazione

```
[Hello[Synchronized[World]
]
]
```

Output con sincronizzazione

```
[Hello]
[Synchronized]
[World]
```

11.4.1 Analisi di Synch

Nell'esempio ci sono tre classi: `Synch`, `Caller`, `CallMe`. Immaginiamo il runtime dell'applicazione, e partiamo dalla classe `Synch` che contiene il metodo `main()` (questa volta l'applet di riferimento è all'indirizzo http://www.claudiodesio.com/java/TutorialJavaThread/Applet/Synch.html). Il thread principale, alla riga 27 istanzia un oggetto chiamato `target` dalla classe `CallMe`. Alla riga 28 istanzia (senza referenziarlo), un oggetto della classe `Caller`, al cui costruttore passa l'istanza `target` e la stringa `Hello`. A questo punto il thread principale si è spostato nell'istanza appena creata dalla classe `Caller`, per eseguirne il costruttore (righe da 14 a 18). In particolare setta come variabili d'istanza, sia l'oggetto `target` sia la stringa `Hello` , quest'ultima referenziata come `msg`. Inoltre crea e fa partire un thread al cui costruttore viene passato l'oggetto `this`. Poi il thread principale, ritorna alla riga 29 istanziando un altro oggetto della classe `Caller`. Al costruttore di questo secondo oggetto viene passato la stessa istanza `target`, che era stata passata al primo, e la stringa `msg`. Prevedibilmente, il costruttore di questo oggetto appena istanziato, setterà le variabili d'istanza `target` e `msg` (con la stringa `Synchronized`), e creerà e farà partire un thread, il cui campo d'azione sarà il metodo `run()` di questo secondo oggetto `Caller`. Infine, il thread principale creerà un terzo oggetto `Caller`, che avrà come variabili d'istanza, sempre lo stesso oggetto `target` e la stringa `world` . Anche in questo oggetto viene creato e fatto partire un thread il cui campo d'azione sarà il metodo `run()` di questo terzo oggetto `Caller`. Subito dopo, il thread principale "muore", e lo scheduler dovrà scegliere quali dei tre thread in stato ready (pronto per l'esecuzione), dovrà essere eseguito [fig. 11.15].

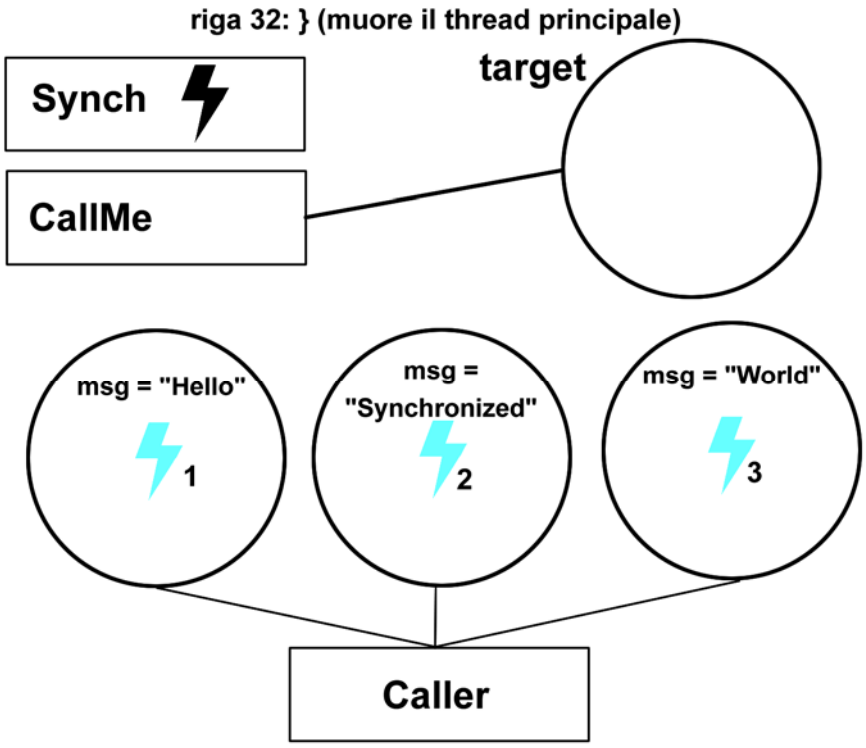


Figura 11.15 – "Muore il thread principale"

Avendo tutti i thread la stessa priorità 5 di default, verrà scelto quello che è da più tempo in attesa. Ecco che allora il primo thread creato (che si trova nell'oggetto dove `msg = "Hello"`), eseguirà il proprio metodo `run()`, chiamando il metodo `call()` sull'istanza `target`. Quindi, il primo thread si sposta nel metodo `call()` dell'oggetto `target` (righe da 2 a 9), mettendosi a "dormire" per un secondo [fig. 11.17], dopo aver stampato una parentesi quadra d'apertura e la stringa `Hello` [fig. 11.16].

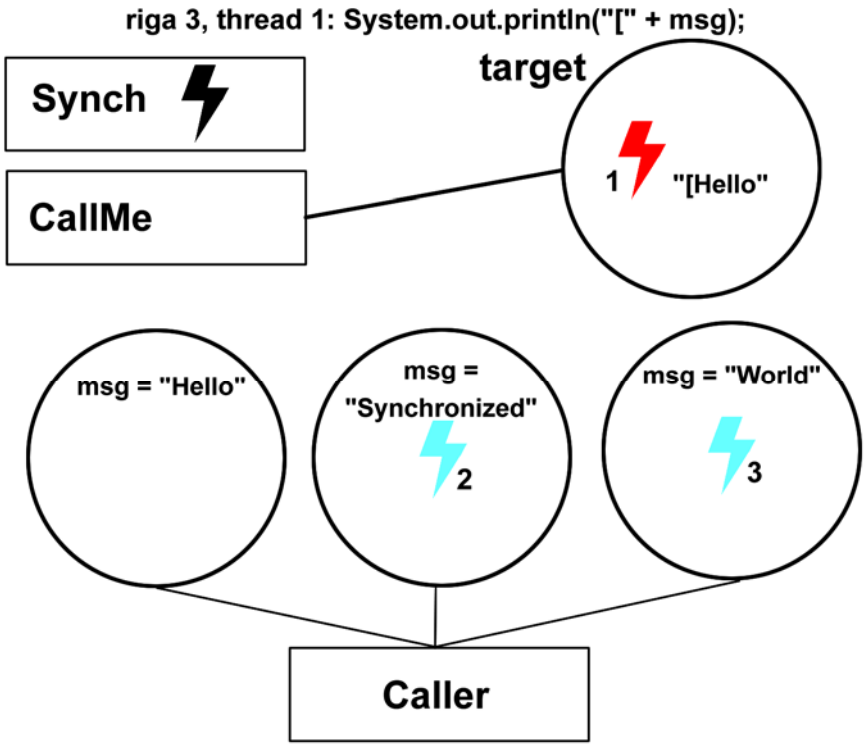


Figura 11.16 – "Stampa della prima parentesi e del primo messaggio"

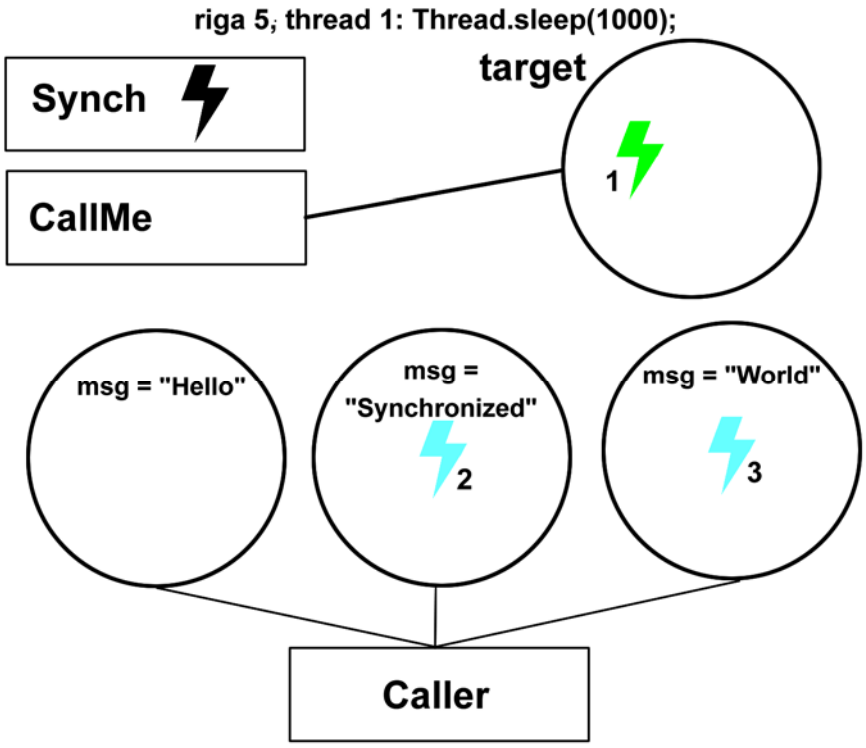


Figura 11.17 – "Il thread 1 'va a dormire'"

A questo punto il secondo thread si impossessa del processore ripetendo in manera speculare le azioni del primo thread. Dopo la chiamata al metodo `call()`, anche il secondo thread si sposta nel metodo `call()` dell'oggetto `target`, mettendosi a "dormire" per un secondo [fig. 11.19], dopo aver stampato una parentesi quadra d'apertura e la stringa `Synchronized` [fig. 11.18].

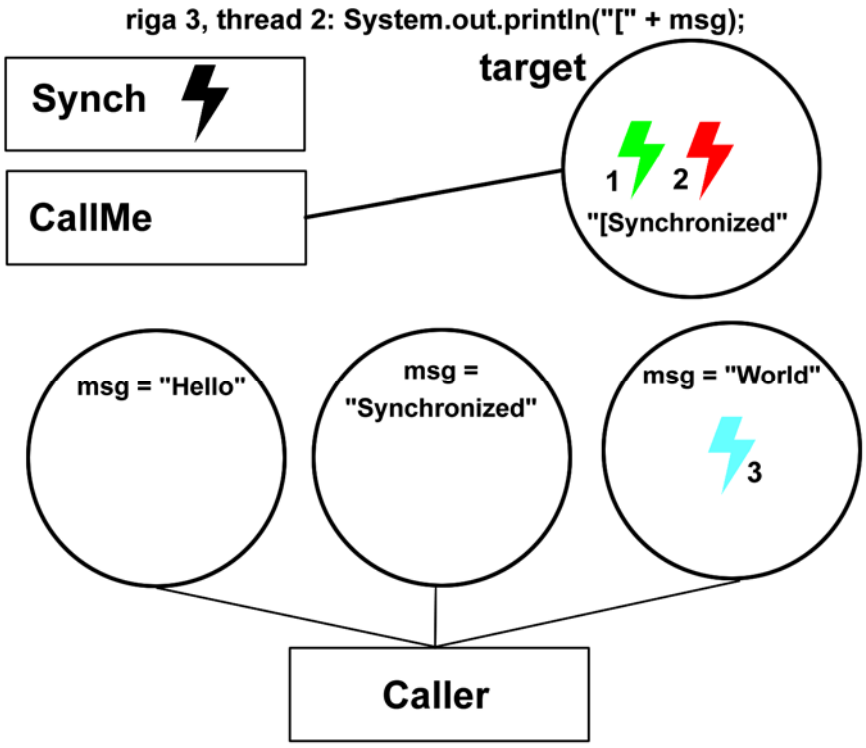


Figura 11.18 – “Stampa della seconda parentesi e del secondo messaggio”

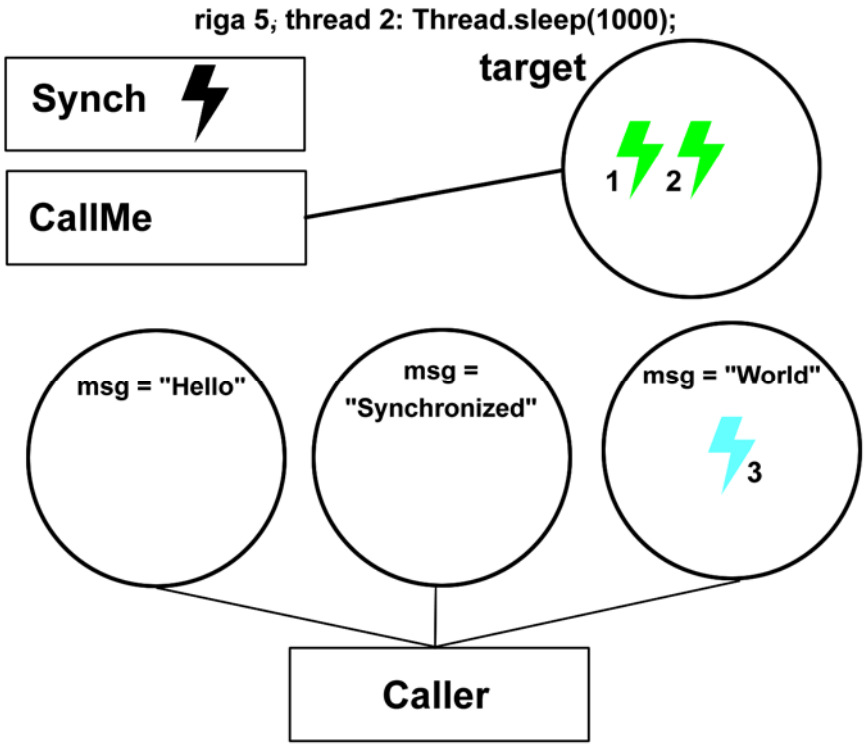


Figura 11.19 – "Il thread 2 'va a dormire'"

Il terzo thread quindi, si sposterà nel metodo `call()` dell'oggetto `target`, mettendosi a "dormire" per un secondo [fig. 11.21], dopo aver stampato una parentesi quadra d'apertura e la stringa `World` [fig. 11.20].

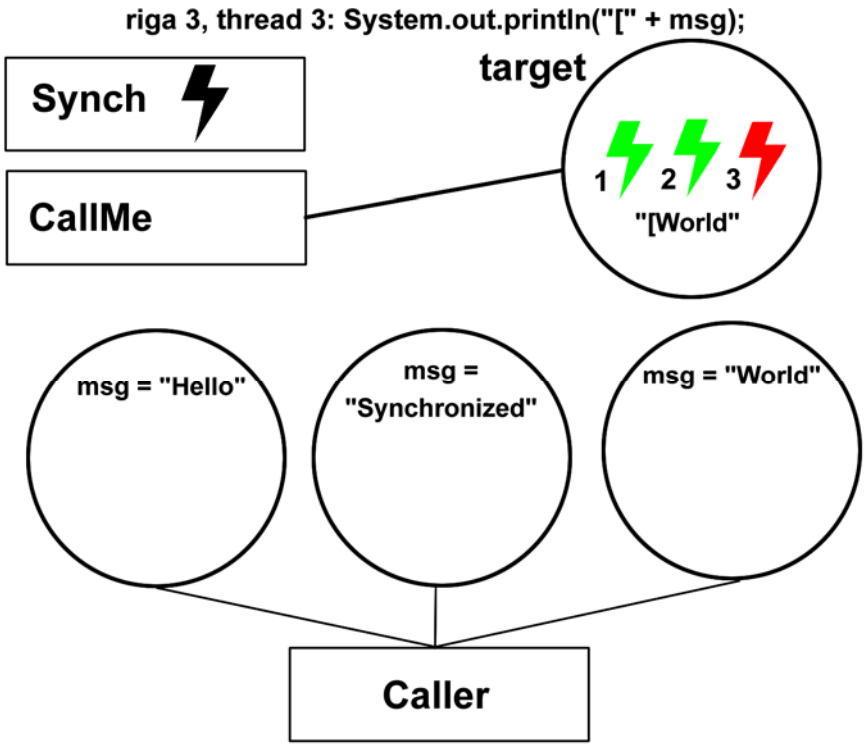


Figura 11.20 – “Stampa della terza parentesi e del terzo messaggio”

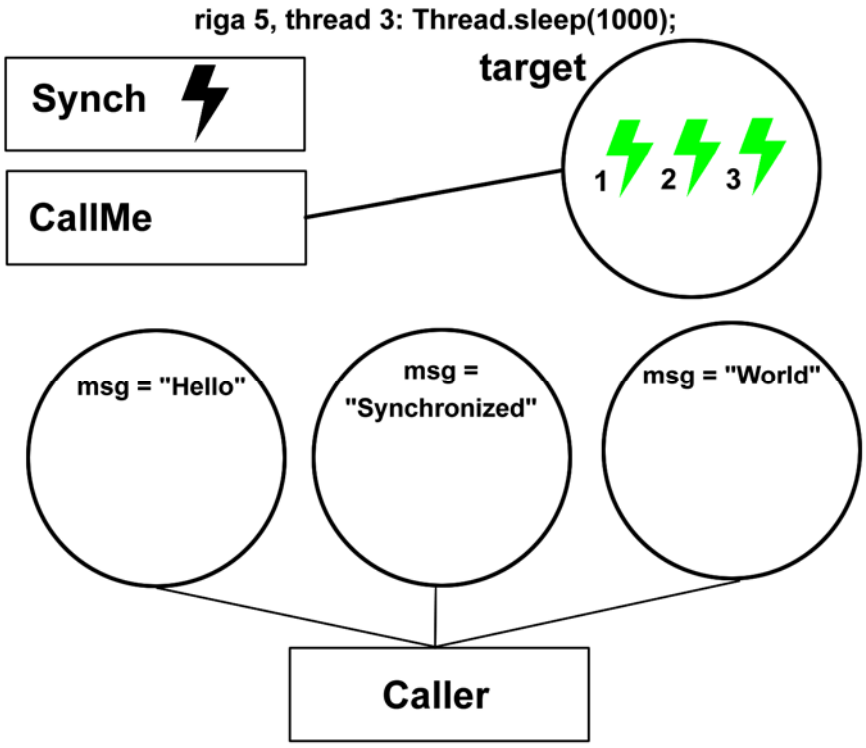


Figura 11.21 – "Il thread 3 'va a dormire'"

Dopo poco meno di un secondo, si risveglierà il primo thread che terminerà la sua esecuzione stampando una parentesi quadra di chiusura ed andando a capo (metodo `println()` alla riga 8) [fig. 11.22].

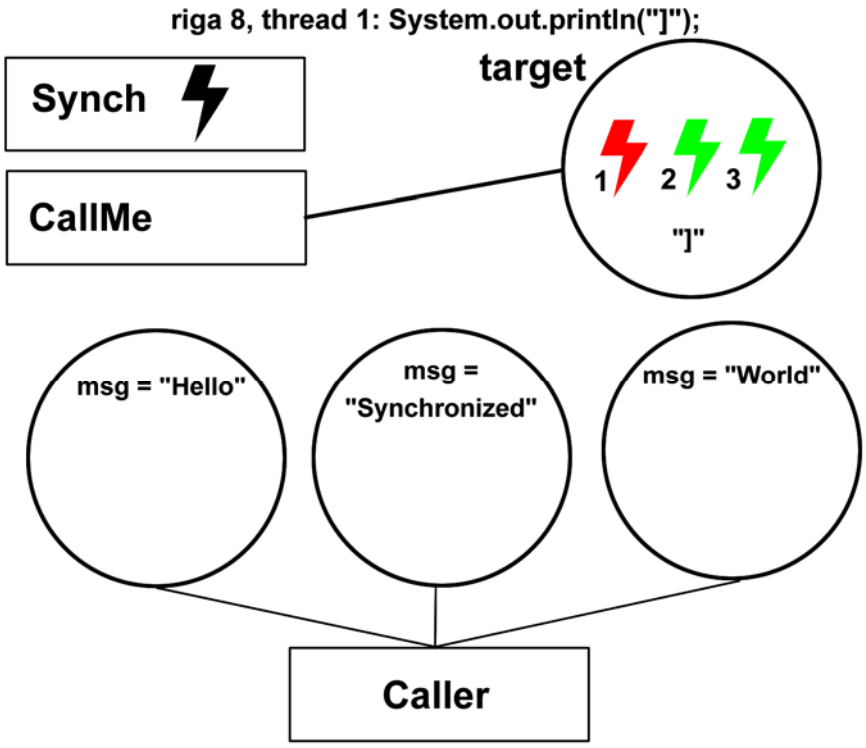


Figura 11.22 – "Stampa prima parentesi di chiusura da parte del thread 1"

Anche gli altri due thread si comporteranno allo stesso modo, negli attimi successivi [fig. 11.23 e 11.24], producendo l'output non sincronizzato.

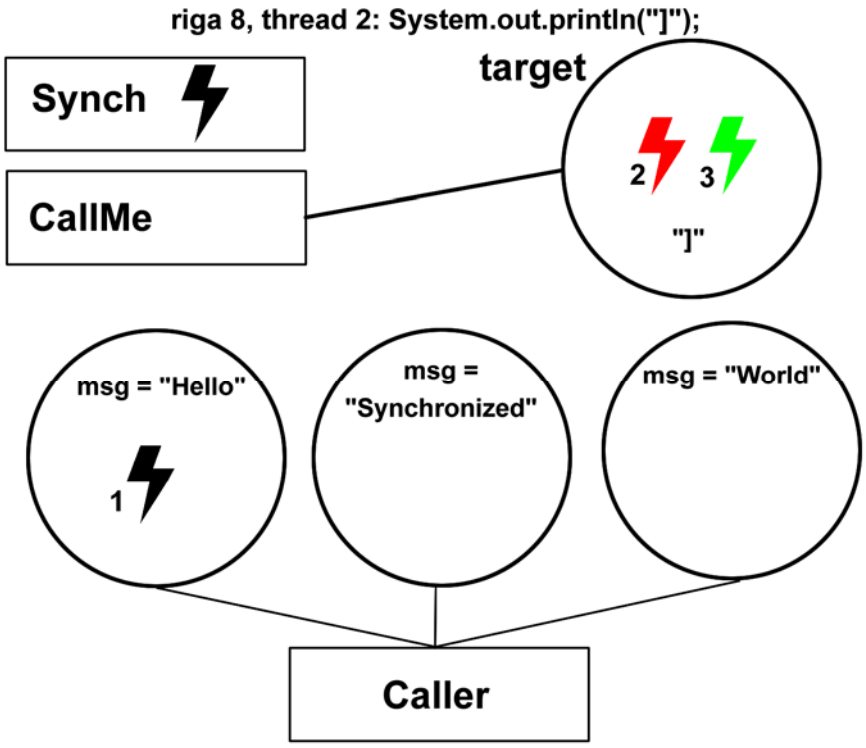


Figura 11.23 – “Stampa seconda parentesi di chiusura da parte del thread 2”

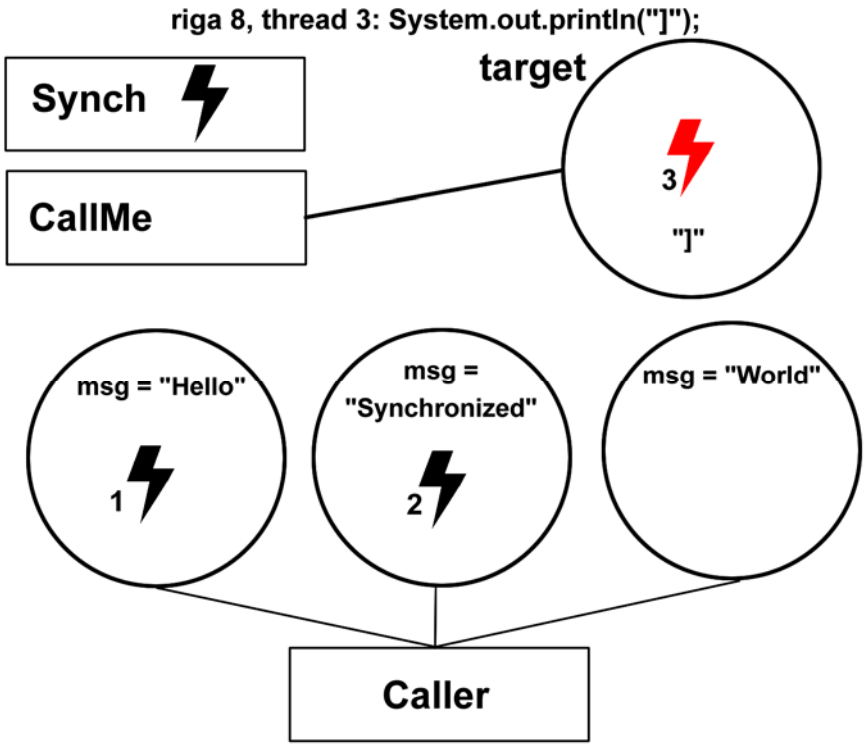


Figura 11.24 – "Stampa terza parentesi di chiusura da parte del thread 3"

L'applicazione ha una durata che si può quantificare in circa un secondo.
Per ottenere l'output sincronizzato, basta decommentare il modificatore `synchronized` anteposto alla dichiarazione del metodo `call()` (riga 2). Infatti, quando un thread inizia ad eseguire un metodo dichiarato sincronizzato, anche in caso di chiamata al metodo `sleep()`, non lascia il codice a disposizione di altri thread. Quindi, sino a quando non termina l'esecuzione del metodo sincronizzato, il secondo thread non può eseguire il codice dello stesso metodo. Ovviamente lo stesso discorso si ripete con il secondo ed il terzo thread. L'applicazione quindi ha una durata quantificabile in circa tre secondi e produce un output sincronizzato.
In alternativa, lo stesso risultato si può ottenere decommentando le righe 20 e 22. In questo caso la keyword `synchronized` assume il ruolo di comando, tramite la sintassi:

```
synchronized (nomeOggetto) {
...blocco di codice sincronizzato...
}
```

e quando un thread si trova all'interno del blocco di codice, valgono le regole di sincronizzazione sopra menzionate.
Si tratta di un modo di sincronizzare gli oggetti che da un certo punto di vista può risultare più flessibile, anche se più complesso e meno chiaro. Infatti è possibile utilizzare un metodo di un oggetto in maniera sincronizzata o meno a seconda del contesto.

11.4.2 Monitor e Lock

Esiste una terminologia ben precisa riguardo la sincronizzazione dei thread. Nei vari testi che riguardano l'argomento, viene definito il concetto di monitor di un oggetto. In Java ogni oggetto ha associato il proprio monitor, se contiene del codice sincronizzato. A livello concettuale un monitor è un oggetto utilizzato come blocco di mutua esclusione per i thread, il che significa che solo un thread può "entrare" in un monitor in un determinato istante.
Java non implementa fisicamente il concetto di monitor di un oggetto, ma questo è facilmente associabile alla parte sincronizzata dell'oggetto stesso. In pratica, se un thread `t1` entra in un metodo sincronizzato `ms1()` di un determinato oggetto `o1`, nessun altro thread potrà entrare in nessun metodo sincronizzato dell'oggetto `o1`, sino a quando `t1`, non avrà terminato l'esecuzione del metodo `ms1` (ovvero non avrà abbandonato il monitor dell'oggetto).
In particolare si dice che il thread `t1`, ha il "lock" dell'oggetto `o1`, quando è entrato nel suo monitor (parte sincronizzata).
È bene conoscere questa terminologia, per interpretare correttamente la documentazione ufficiale. Tuttavia, l'unica preoccupazione che deve avere il programmatore è l'utilizzo della keyword `synchronized`.

11.5 La comunicazione fra thread

Nell'esempio precedente abbiamo visto come la sincronizzazione di thread che condividono gli stessi dati sia facilmente implementabile. Purtroppo, le situazioni che si presenteranno dove bisognerà gestire la sincronizzazione di thread, non saranno sempre così semplici. Come vedremo nel prossimo esempio, la sola keyword `synchronized` non sempre basta a risolvere i problemi di sincronizzazione fra thread. Descriviamo lo scenario del prossimo esempio. Vogliamo creare una semplice applicazione che simuli la situazione economica ideale, dove c'è un produttore che produce un prodotto, e un consumatore che lo consuma. In questo modo, il produttore non avrà bisogno di un

magazzino. Ovviamente le attività del produttore e del consumatore saranno eseguite da due thread lanciati in attività parallele.

```
// Classe Magazzino ***************************
1   public class WareHouse{
2     private int numberOfProducts;
3     private int idProduct;
4     public synchronized void put(int idProduct) {
5       this.idProduct = idProduct;
6       numberOfProducts++;
7       printSituation("Produced " + idProduct);
8     }
9     public synchronized int get() {
10      numberOfProducts--;
11      printSituation("Consumed " + idProduct);
12      return idProduct;
13    }
14    private synchronized void printSituation(String
msg)
15      System.out.println(msg +"\n" + numberOfProducts
16      + " Product in Warehouse");
17    }
18  }
19
20  //classe Produttore ***************************
21
22    public class Producer implements Runnable {
23      private WareHouse wareHouse;
24      public Producer(WareHouse wareHouse) {
25        this.wareHouse = wareHouse;
26        new Thread(this, "Producer").start();
27      }
28      public void run() {
29        for (int i = 1; i <= 10; i++) {
30          wareHouse.put(i);
31        }
32      }
33    }
34
```

```
35  //classe Consumatore ************************
36  public class Consumer implements Runnable {
37    private WareHouse wareHouse;
38    public Consumer(WareHouse wareHouse) {
39      this.wareHouse = wareHouse;
40      new Thread(this, "Consumer").start();
41    }
42    public void run() {
43      for (int i = 0; i < 10;) {
44        i = wareHouse.get();
45      }
46    }
48  }
49
50  //classe del main ************************
51  public class IdealEconomy {
52    public static void main(String args[]) {
53      WareHouse wareHouse = new WareHouse();
54      new Producer(wareHouse);
55      new Consumer(wareHouse);
56    }
57  }
```

Ouput su Sun Solaris 8:

```
solaris% java IdealEconomy
Produced 1
1 Product in Warehouse
Produced 2
2 Product in Warehouse
Produced 3
3 Product in Warehouse
Produced 4
4 Product in Warehouse
Produced 5
5 Product in Warehouse
Produced 6
6 Product in Warehouse
Produced 7
```

```
7 Product in Warehouse
Produced 8
8 Product in Warehouse
Produced 9
9 Product in Warehouse
Produced 10
10 Product in Warehouse
Consumed 10
9 Product in Warehouse
```

11.5.1 Analisi di IdealEconomy

Visto l'output prodotto dal codice l'identificatore della classe `IdealEconomy`, suona un tantino ironico...

> **L'output in questione è stato generato su di un sistema Unix, (comportamento preemptive), ed è l'output obbligato per ogni esecuzione dell'applicazione. C'è da dire che se l'applicazione fosse stata lanciata su Windows, l'output avrebbe potuto variare drasticamente da esecuzione a esecuzione. Su Windows l'output "migliore", sarebbe proprio quello che è standard su Unix. Il lettore può lanciare l'applicazione più volte per conferma su un sistema Windows.**

Come al solito cerchiamo di immaginare il runtime dell'applicazione, supponendo di lanciare l'applicazione su di un sistema Unix (caso più semplice). La classe `IdealEconomy`, fornisce il metodo `main()`, quindi partiremo dalla riga 53, dove viene istanziato un oggetto `WareHouse` (letteralmente "magazzino").

> **La scelta dell'identificatore `magazzino`, può anche non essere condivisa da qualcuno. Abbiamo deciso di pensare ad un magazzino, perchè il nostro obiettivo è quello di tenerlo sempre vuoto.**

Nelle successive due righe, vengono istanziati un oggetto `Producer` ed un oggetto `Consumer`, ai cui costruttori viene passato lo stesso oggetto `WareHouse`, (già si intuisce che il magazzino sarà condiviso fra i due thread). Sia il costruttore di `Producer` sia il costruttore di `Consumer`, dopo aver settato come variabile d'istanza l'istanza comune di `WareHouse`, creano un thread (con nome rispettivamente

`Producer` e `Consumer`) e lo fanno partire. Una volta che il thread principale ha eseguito entrambi i costruttori, muore, e poiché è stato fatto partire per primo, il thread `Producer`, passa in stato di esecuzione all'interno del suo metodo `run()`. Da questo metodo viene chiamato il metodo sincronizzato `put()`, sull'oggetto `wareHouse`. Essendo questo metodo sincronizzato, ne è garantita l'atomicità dell'esecuzione, ma nel contesto corrente ciò non basta a garantire un corretto comportamento dell'applicazione. Infatti, il thread `Producer` chiamerà il metodo `put()` per 10 volte [riga 29], per poi terminare il suo ciclo di vita, e lasciare l'esecuzione al thread `Consumer`. Questo eseguirà un'unica volta il metodo `get()` dell'oggetto `wareHouse`, per poi terminare il suo ciclo di vita e con esso il ciclo di vita dell'applicazione.
Sino ad ora, non abbiamo visto ancora dei meccanismi chiari per far comunicare i thread. Il metodo `sleep()`, le priorità e la parola chiave `synchronized`, non rappresentano meccanismi sufficienti per fare comunicare i thread. Contrariamente a quanto ci si possa aspettare, questi meccanismi non sono definiti nella classe `Thread`, bensì nella classe `Object`.
Trattasi di metodi dichiarati `final` nella classe `Object` e pertanto ereditati e da tutte le classi e non modificabili (applicando l'override). Possono essere invocati in un qualsiasi oggetto all'interno di codice sincronizzato. Questi metodi sono:

- `wait()`: dice al thread corrente (cioè che legge la chiamata a questo metodo) di abbandonare il monitor e porsi in pausa finché qualche altro thread non entra nello stesso monitor e chiama `notify()`
- `notify()`: richiama dallo stato di pausa il primo thread che ha chiamato `wait()` nello stesso oggetto
- `notifyAll()`: richiama dalla pausa tutti i thread che hanno chiamato `wait()` in quello stesso oggetto. Viene fra questi eseguito per primo quello a più alta priorità. Quest'ultima affermazione potrebbe non essere verificata da differenti versioni della JVM.

In realtà esistono dei metodi nella classe `Thread` che realizzano una comunicazione tra thread: `suspend()` e `resume()`. Questi però sono attualmente deprecati, per colpa dell'alta probabilità di produrre "DeadLock". Il Deadlock è una condizione di errore difficile da risolvere, in cui due thread stanno in reciproca dipendenza in due oggetti sincronizzati. (Esempio veloce: il thread `t1` si trova nel metodo sincronizzato `m1` dell'oggetto `o1` (di cui quindi possiede il lock), il

thread `t2` si trova nel metodo sincronizzato `m2` dell'oggetto `o2` (di cui quindi possiede il lock). Se il thread `t1` prova a chiamare il metodo `m2`, si bloccherà in attesa che il thread `t2` rilasci il lock dell'oggetto `o2`. Se poi anche il thread `t2` prova a chiamare il metodo `m1`, si bloccherà in attesa che il thread `t1` rilasci il lock dell'oggetto `o1`. L'applicazione rimarrà bloccata in attesa di una interruzione da parte dell'utente). Come si può intuire anche dai soli identificatori, i metodi `suspend()` e `resume()`, più che realizzare una comunicazione tra thread di pari dignità, implicavano l'esistenza di thread "superiori" che avevano il compito di gestirne altri. Poiché si tratta di metodi deprecati, non aggiungeremo altro.

Allo scopo di far correttamente comportare il runtime dell'applicazione `IdealEconomy`, andiamo a modificare solamente il codice della classe `WareHouse`. In questo modo sarà la stessa istanza di questa classe (ovvero il contesto di esecuzione condiviso dai due thread), a stabilire il comportamento corretto dell'applicazione. In pratica l'oggetto `wareHouse` bloccherà (`wait()`), il thread `Producer` una volta realizzato un `put()` del prodotto, dando via libera al `get()` del `Consumer`. Poi notificherà (`notify()`) l'avvenuta consumazione del prodotto al `Producer`, e bloccherà (`wait()`) il `get()` di un prodotto (che non esiste ancora) da parte del thread `Consumer`. Poi, dopo che il `Producer` avrà realizzato un secondo `put()` del prodotto, verrà prima avvertito (`notify()`) il `Consumer`, e poi bloccato il `Producer` (`wait()`). Ovviamente il ciclo si ripete per 10 iterazioni. Per realizzare l'obiettivo si introduce la classica tecnica del flag (boolean `empty` che vale `true` se il magazzino è vuoto). Di seguito troviamo il codice della nuova classe `WareHouse`:

```
1   public class WareHouse{
2     private int numberOfProducts;
3     private int idProduct;
4     private boolean empty = true; // magazzino vuoto
5     public synchronized void put(int idProduct) {
6       if (!empty) // se il magazzino non è vuoto...
7         try {
8           wait(); // fermati Producer
9         }
10        catch (InterruptedException exc) {
11          exc.printStackTrace();
```

```
12        }
13      this.idProduct = idProduct;
14      numberOfProducts++;
15      printSituation("Produced " + idProduct);
16      empty = false;
17      notify(); // svegliati Consumer
18    }
19    public synchronized int get() {
20      if (empty) // se il magazzino è vuoto...
21        try {
22          wait(); // bloccati Consumer
23        }
24        catch (InterruptedException exc) {
25          exc.printStackTrace();
26        }
27      numberOfProducts--;
28      printSituation("Consumed " + idProduct);
29      empty = true; // il magazzino ora è vuoto
30      notify(); // svegliati Producer
31      return idProduct;
32    }
33    private synchronized void printSituation(String
msg)
34      System.out.println(msg +"\n" + numberOfProducts +
35      " Product in Warehouse");
36    }
37  }
```

Immaginiamo il runtime relativo a questo codice. Il thread `Producer` chiamerà il metodo `put()` passandogli `1` alla prima iterazione. Alla riga 6 viene controllato il flag `empty`, siccome il suo valore è `true`, non sarà chiamato il metodo `wait()`. Quindi viene settato l'`idProduct`, incrementato il `numberOfProducts`, e chiamato il metodo `printSituation()`. Poi, il flag `empty` viene settato a `false` (dato che il magazzino non è più vuoto) e viene invocato il metodo `notify()`. Quest'ultimo non ha nessun effetto dal momento che non ci sono altri thread in attesa nel monitor di questo oggetto. Poi riviene chiamato il metodo `put()` dal thread `Producer` con argomento `2`. Questa volta il controllo alla riga 6 è verificato (`empty` è `false` cioè il magazzino non è vuoto), e quindi il thread `Producer` va a chiamare il metodo `wait()` rilasciando il

lock dell'oggetto. A questo punto il thread `Consumer` va ad eseguire il metodo `get()`. Il controllo alla riga 20, fallisce perchè `empty` è `false`, e quindi non viene chiamato il metodo `wait()`. Viene decrementato il `numberOfProducts` a `0`, chiamato il metodo `printSituation()`, viene settato il flag `empty` a `true`. Poi viene chiamato il metodo `notify()` che toglie il thread `Producer` dallo stato di `wait()`. Infine viene ritornato il valore di `idProduct`, che è ancora `1`. Poi viene richiamato il metodo `get()`, ma il controllo alla riga 20 è verificato e il thread `Consumer` si mette in stato di `wait()`. Quindi il thread `Producer` continua la sua esecuzione da dove si era bloccato cioè dalla riga 8. Quindi viene settato l'`idProduct` a `2`, incrementato di nuovo ad `1` il `numberOfProducts`, e chiamato il metodo `printSituation()`. Poi, il flag `empty` viene settato a `false` e viene chiamato il metodo `notify()` che risveglia il thread `Consumer`...

11.5.2 Conclusioni

La gestione dei thread in Java, può considerarsi semplice se paragonata alla gestione dei thread in altri linguaggi. Inoltre, una classe (`Thread`), un'interfaccia (`Runnable`), una parola chiave (`synchronized`) e tre metodi (`wait()`, `notify()` e `notifyAll()`), rappresentano il nucleo di conoscenza fondamentale per gestire applicazioni multi-threaded. In realtà le situazioni multi-threaded che bisognerà gestire nelle applicazioni reali, non saranno tutte così semplici come nell'ultimo esempio. Tuttavia la tecnologia Java fa ampio uso del multi-threading. La notizia positiva, è che la complessità del multi-threading, è gestita dalla tecnologia stessa. Consideriamo ad esempio la tecnologia Java Servlet (http://java.sun.com/products/servlet/). La tecnologia Java Servlet nasce come alternativa alla tecnologia C.G.I. (Common Gateway Interface) ma è in grado di gestire il multi-threading in maniera automatica, allocando thread diversi per servire ogni richiesta del client. Quindi, lo sviluppatore è completamente esonerato dal dover scrivere codice per la gestione dei thread. Anche altre tecnologie Java (Java Server Pages, Enterprise JavaBeans…), gestiscono il multi-threading automaticamente, ed alcune classi della libreria Java (`java.nio.channels.SocketChannel`, `java.util.Timer...`), semplificano enormemente il rapporto tra lo sviluppatore ed i thread. Ma senza conoscere l'argomento, è difficile utilizzare questi strumenti in maniera “consapevole” .

Dalla versione 5 di Java, il supporto al multithreading è stato ulteriormente ampliato, con l’introduzione di classi molto potenti che semplificano il lavoro dello sviluppatore. Per esempio, è stato introdotto il concetto di semaforo, argomento già familiare a chi

conosce la gestione dei thread in altri ambienti. Esistono ora anche classi che permettono con poche righe di codice la creazione di pool di thread (cfr. documentazione `java.util.concurrent.ThreadPoolExcecutor`)! Inoltre, oltre all'interfaccia `Runnable` che ci permette riscrivere il metodo `run()` e definire il comportamento dei nostri thread, è stata introdotta l'interfaccia `Future` e la classe `FutureTask`, che permettono di superare il limite dei metodi `run()` degli oggetti `Runnable`, di non poter tornare parametri in output. Infatti il metodo `run()` è definito con tipo di ritorno `void`, e, come sappiamo, nell'override non è permesso cambiare il tipo di ritorno del metodo. Invece `Future`, è una interfaccia definita come Generic, che ci permetterà di ottenere tipi di ritorno dall'esecuzione dei nostri thread. Un vantaggio non da poco.

Riepilogo

Abbiamo distinto le definizioni di multi-threading e multi-tasking. Abbiamo gradualmente dato la definizione di thread che è piuttosto complessa (processore virtuale, che esegue codice su determinati dati). Abbiamo utilizzato la classe `Thread` e l'interfaccia `Runnable` per avere del codice da far eseguire ai nuovi thread. Abbiamo parlato di scheduler e priorità, ed abbiamo visto come questo concetto non sia quello determinante per gestire più thread contemporaneamente. Abbiamo imparato a sincronizzare più thread introducendo i concetti di monitor e lock di un oggetto in due modi diversi: sincronizzando un metodo, o un'istanza in un determinato blocco di codice. Abbiamo visto che è spesso necessario sincronizzare alcune parti di codice quando esiste del codice "eseguibile" da più thread. La sincronizzazione si limita a sincronizzare il codice, non i dati che vengono utilizzati dal codice. Abbiamo quindi infine esplorato con un esempio, la comunicazione tra thread. Questa si gestisce direttamente dal codice da eseguire in comune mediante i metodi `wait()`, `notify()` e `notifyAll()` della classe `Object`.

Esercizi modulo 11

Esercizio 11.a)
Creazione di Thread, Vero o Falso:

1. Un thread è un oggetto istanziato dalla classe `Thread` o dalla classe `Runnable`
2. Il multi-threading è solitamente una caratteristica dei sistemi operativi e non dei linguaggi di programmazione
3. In ogni applicazione al runtime esiste almeno un thread in esecuzione
4. A parte il thread principale, un thread ha bisogno di eseguire del codice all'interno di un oggetto la cui classe o estende `Runnable` o estende `Thread`
5. Il metodo `run()` è il metodo che deve essere chiamato dal programmatore per attivare un thread
6. Il "thread corrente" non si identifica solitamente con il reference `this`
7. Chiamando il metodo `start()` su di un thread, questo viene immediatamente eseguito
8. Il metodo `sleep()`, è statico e permette di far dormire il thread che legge tale istruzione, per un numero specificato di millisecondi
9. Assegnare le priorità ai thread è una attività che può produrre risultati diversi su piattaforme diverse
10. Lo scheduler della JVM, non dipende dalla piattaforma su cui viene lanciato

Esercizio 11.b)
Gestione del multi-threading, Vero o Falso:

1. Un thread astrae un processore virtuale, che esegue codice su determinati dati
2. La parola chiave `synchronized` può essere utilizzata sia come modificatore di un metodo, sia come modificatore di una variabile
3. Il monitor di un oggetto può essere identificato con la parte sincronizzata dell'oggetto stesso
4. Affinchè due thread che eseguono lo stesso codice e condividono gli stessi dati, non abbiano problemi di concorrenza, basta sincronizzare il codice comune
5. Un thread si dice che ha il lock di un oggetto se entra nel suo monitor
6. I metodi `wait()`, `notify()` e `notifyAll()`, rappresentano il principale strumento per far comunicare più thread

7. I metodi `sospend()` e `resume()` sono attualmente deprecati
8. Il metodo `notityAll()` invocato su di un certo oggetto `o1`, risveglia dallo stato di pausa tutti i thread che hanno invocato `wait()` sullo stesso oggetto. Tra questi verrà eseguito quello che era stato fatto partire per primo con il metodo `start()`
9. Il deadlock è una condizione di errore bloccante che viene generata da due thread che stanno in reciproca dipendenza in due oggetti sincronizzati
10. Se un thread `t1` esegue il metodo `run()` nell'oggetto `o1` della classe `C1`, e un thread `t2` esegue il metodo `run()` nell'oggetto `o2` della stessa classe `C1`, la parola chiave `synchronized` non serve a niente

Soluzioni esercizi modulo 11

Esercizio 11.a)
Creazione di Thread, Vero o Falso:

1. **Falso** `Runnable` è un'interfaccia
2. **Vero**
3. **Vero** il cosiddetto thread "main"
4. **Vero**
5. **Falso** il programmatore può invocare il metodo `start()` e lo scheduler invocherà il metodo `run()`
6. **Vero**
7. **Falso**
8. **Vero**
9. **Vero**
10. **Falso**

Esercizio 11.b)
Gestione del multi-threading, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Vero**
4. **Falso**
5. **Vero**
6. **Vero**
7. **Vero**
8. **Falso** il primo thread che partirà sarà quello a priorità più alta
9. **Vero**
10. **Vero**

Obiettivi del modulo)

Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper definire multi-threading e multi-tasking (unità 11.1)	☐	
Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 11.2)	☐	
Saper creare ed utilizzare thread tramite la classe Thread e l'interfaccia Runnable (unità11.2)	☐	
Definire cos'è uno scheduler e i suoi comportamenti riguardo le priorità dei thread (unità11.3)	☐	
Sincronizzare thread (unità11.4)	☐	
Far comunicare i thread (unità11.5)	☐	

Note:

12

Complessità: bassa

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere l'utilità e saper utilizzare il framework Collection (unità 12.1) .
2. Saper implementare programmi con l'internazionalizzazione (unità 12.1).
3. Saper implementare programmi configurabili mediante file di properties (unità 12.1).
4. Saper utilizzare la classe StringTokenizer per "splittare" stringhe (unità 12.1).
5. Saper utilizzare la Reflection per l'introspezione delle classi (unità 12.2).
6. Saper introdurre le classi System, Math e Runtime (unità 12.1).

12 Le librerie alla base del linguaggio: java.lang e java.util

Questo modulo è interamente dedicato ai package che probabilmente sono i più utilizzati in assoluto: `java.lang` e `java.util`. Il primo ricordiamo che è l'unico package importato in automatico in tutti i nostri programmi. Il secondo contiene classi di cui il programmatore Java non può proprio fare a meno. Ovviamente, questo modulo non coprirà tutte le circa 200 classi presenti in questi package, bensì cercheremo di introdurre i principali concetti e la filosofia con cui utilizzare questi package.

12.1 Package java.util

Il package `java.util`, contiene una serie di classi di utilità come il framework "Collections" per gestire collezioni eterogenee di ogni tipo, il modello ad eventi, classi per la gestione facilitata delle date e degli orari, classi per la gestione dell'internazionalizzazione, e tante altre utilità come un separatore di stringhe, un generatore di numeri casuali e così via…

Con il termine "framework" in questo caso intendiamo un'insieme di classi e di interfacce riutilizzabili ed estendibili. Questa definizione

dovrebbe essere un po' più complessa me per i nostri scopi dovrebbe andar bene così.

Per chi non ne ha ancora abbastanza di thread, consigliamo anche di dare uno sguardo alla documentazione relativa alle classi `Timer` e `TimerTask`.

12.1.1 Framework Collections

Il framework noto come "Collections", è costituito da una serie di classi ed interfacce che permettono la gestione di collezioni eterogenee (cfr. Modulo 6) di ogni tipo. I vantaggi di avere a disposizione questo framework per la programmazione sono tanti: possibilità di scrivere meno codice, incremento della performance, interoperabilità tra classi non relazionate tra loro, riusabilità, algoritmi complessi già a disposizione (per esempio per l'ordinamento) etc…
Il framework è basato su 6 interfacce principali (poi diventate 9 nella versione 5), che vengono poi estese da tante altre classi astratte e concrete. Nella figura 12.1) viene evidenziata la gerarchia di queste interfacce con un class diagram, più avanti definiremo anche le restanti altre 3 interfacce base `Queue`, `BlockingQueue` e `ConcurrentMap`. Ad un primo livello ci sono le interfacce `Collection` (la più importante), `Map` e `SortedMap`. `Collection` è estesa dalle interfacce `Set`, `List`, e `SortedSet`. Lo scopo principale di queste interfacce è quello di permettere la manipolazioni delle implementazioni indipendentemente dai dettagli di rappresentazione. Questo implica che capire a cosa servono queste interfacce, significa capire la maggior parte dell'utilizzo del framework. Ognuna di queste interfacce ha delle proprietà che le sottoclassi ereditano. Per esempio una classe che implementa `Set` (in italiano "Insieme") è un tipo di collezione non ordinata che non accetta elementi duplicati. Quindi descriviamo un po' meglio queste interfacce:

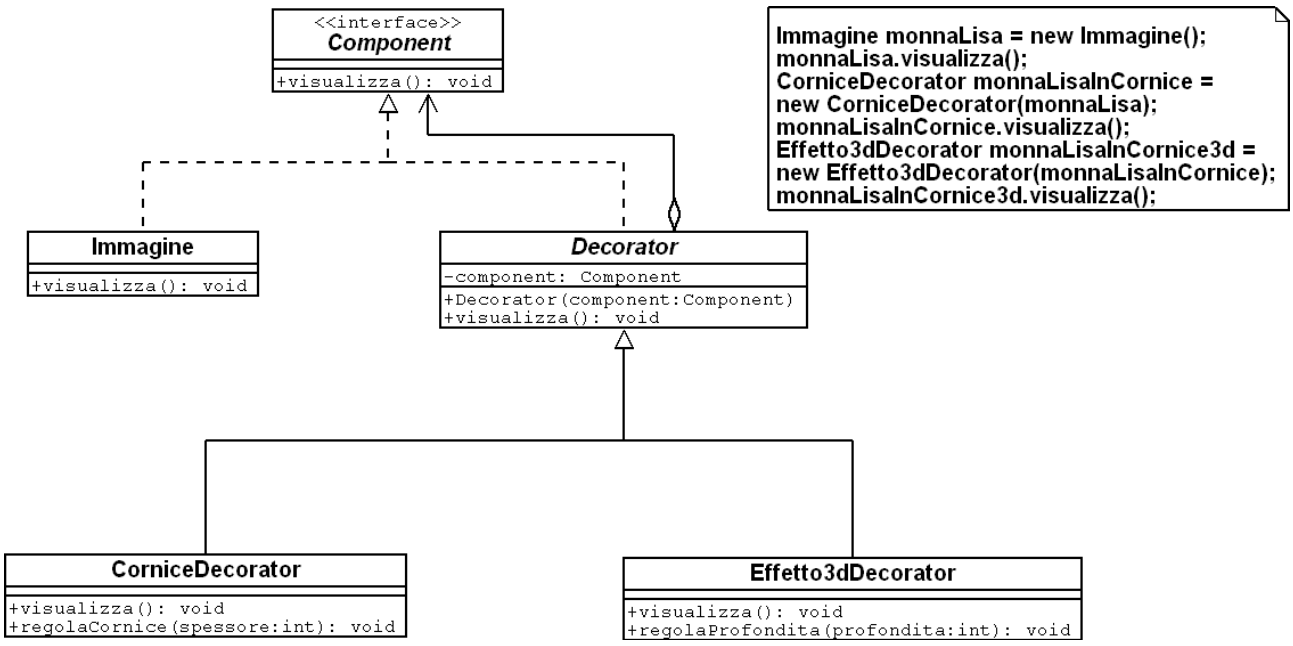


Figura 12.1 – "La gerarchie tra le interfacce fondamentali del framework Collections"

L'interfaccia **`Collection`** è la radice di tutta la gerarchia del framework. Essa astrae il concetto di "insieme di oggetti", detti "elementi". Esistono implementazioni che ammettono elementi duplicati ed altre che non lo permettono, collezioni ordinate e non ordinate. La libreria non mette a disposizione nessuna implementazione diretta di `Collection` ma solo delle sue dirette sotto-interfacce come `Set` e `List`.
L'interfaccia `Collection` viene definita come "il minimo comun denominatore" che tutte le collection devono implementare.
Un **`Set`** (in italiano "Insieme") è un tipo di collection che astraendo il concetto di insieme matematico, non ammette elementi duplicati.
Una **`List`** è una collezione ordinata (detta anche "sequence"). In una lista viene sempre associato un indice ad ogni elemento, che equivale alla posizione dell'elemento stesso all'interno della lista. Una lista ammette elementi duplicati (distinguibili fra di loro per la posizione).
Una **`Map`** è una collezione che associa delle chiavi ai suoi elementi. Le mappe non possono contenere chiavi duplicate, ed ogni chiave può essere associata ad un solo valore.
Le ultime due interfacce **`SortedSet`** e **`SortedMap`** rappresentano le versioni ordinate di `Set` e `Map`. Esse aggiungono anche diverse nuove funzionalità relative all'ordinamento.

Esistono due modi per ordinare oggetti: sfruttare l'interfaccia `Comparable` o l'interfaccia `Comparator`. La prima fornisce un "ordinamento naturale" alle classi che la implementano, ed è implementata da molte classi della libreria standard come `String`. L'interfaccia `Comparator` invece, permette al programmatore di implementare soluzioni di ordinamento personalizzate, mediante l'override del metodo `compare()`. È estremamente utile leggere la documentazione ufficiale di queste due interfacce.

Di seguito introduciamo alcuni esempi di implementazioni di collection.

12.1.2 Implementazioni di Map e SortedMap

Una tipica implementazione di una mappa è l'**`Hashtable`**. Questa classe permette di associare ad ogni elemento della collezione una chiave univoca. Sia la chiave sia l'elemento associato sono di tipo `Object` (quindi per il polimorfismo, un qualsiasi oggetto). Si aggiungono elementi mediante il metodo `put(Object key, Object value)`, e si recuperano tramite il metodo `get(Object key)`. In particolare il metodo `get()` permette un recupero molto performante dell'elemento della collezione, mediante la specifica della chiave. Per esempio:

```
Hashtable table = new Hashtable();
table.put("1", "la data attuale è ");
table.put("2", new Date());
table.put("3", table);
for (int i = 0; i <= table.size(); i++){
    System.out.println(table.get(""+i));
}
```

Come tutte le collection, anche l'`Hashtable` è una collezione eterogenea. Non sono ammesse chiavi duplicate, né elementi `null`. Inoltre un oggetto `Hashtable` è sincronizzato di default.
Una classe non sincronizzata (e quindi ancora più performante) e del tutto simile ad `Hashtable`, è la classe **`HashMap`**.

Per quanto riguarda le classi `Hashtable` e `HashMap` esistono delle regole per gestirne al meglio la performance, che si basano su due

parametri detti load factor (fattore di carico) e buckets (capacità). Per chi volesse approfondire questo discorso consigliamo di rifarsi alla documentazione ufficiale.

Un'altra classe molto simile ma questa volta implementazione di `SortedMap` è invece **`TreeMap`**. La classe `HashMap` è notevolmente più performante rispetto a `TreeMap`. Quest'ultima però gestisce l'ordinamento, che non è una caratteristica di tutte le mappe. Ovviamente, l'ordinamento è gestito sulle chiavi e non sui valori.

12.1.3 Implementazioni di Set e SortedSet

Un'implementazione di `Set` è **`HashSet`**, mentre un'implementazione di `SortedSet` è **`TreeSet`**. `HashSet` è più performante rispetto a `TreeSet`, ma non gestisce l'ordinamento. Entrambe queste classi, non ammettono elementi duplicati. Per il resto, anche per queste due classi valgono esattamente le stesse regole di `HashMap` e `TreeMap`.
Segue un esempio di utilizzo di `TreeSet` dove si aggiungono degli elementi e poi si cicla su di esso per stamparli:

```
TreeSet set = new TreeSet();
set.add("c");
set.add("a");
set.add("b");
set.add("b");
Iterator iter = set.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

L'output sarà

```
a
b
c
```

Infatti l'elemento duplicato (`b`) non è stato aggiunto e gli elementi sono stati ordinati secondo la loro natura di stringhe.
Nell'esempio abbiamo utilizzato anche un'implementazione dell'interfaccia **`Iterator`**, che permette mediante i suoi metodi di iterare sugli elementi della

collection, in maniera standard e molto intuitiva. La stessa interfaccia `Collection` definisce il metodo `iterator()`.
Un'altra interfaccia che bisogna menzionare è **`Enumeration`**. Essa è molto simile all'`Iterator` ed ha esattamente la stessa funzionalità. Ci sono solo due differenze tra queste due interfacce:

- i nomi dei metodi (che rimangono però molto simili)
- un `Iterator` può durante l'iterazione anche rimuovere elementi mediante il metodo `remove()`

In effetti l'interfaccia `Iterator` è nata solo nella versione 1.2 di Java proprio per sostituire `Enumeration` nel framework Collections.

12.1.4 Implementazioni di List

Le principali implementazioni di `List`, sono `ArrayList`, `Vector` e `LinkedList`. `Vector` e `ArrayList` hanno la stessa funzionalità ma quest'ultima è più performante perchè non è sincronizzata. **`ArrayList`** ha prestazioni nettamente superiori anche a **`LinkedList`**, che conviene utilizzare solo per gestire collezioni di tipo "coda". Infatti mette a disposizione metodi come `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()`, e `removeLast()`. È quindi opportuno scegliere l'utilizzo di una `LinkedList` in luogo di `ArrayList`, solo quando si devono spesso aggiungere elementi all'inizio della lista, oppure cancellare elementi all'interno della lista durante le iterazioni. Queste operazioni hanno infatti un tempo costante nelle `LinkedList`, e un tempo "lineare" (ovvero che dipende dal numero degli elementi) in un `ArrayList`. In compenso però, l'accesso posizionale in una `LinkedList` è lineare mentre è costante in un `ArrayList`. Questo implica una performance superiore da parte dell'`ArrayList` nella maggior parte dei casi.

La classe `LinkedList` attualmente è stata rivisitata per implementare anche l'interfaccia `Queue`, una delle nuove interfacce base (che come asserito precedentemente sono 9 a partire dalla versione 1.5).

Anche l'`ArrayList` possiede un parametro di configurazione: la capacità iniziale. Se istanziamo un `ArrayList` con capacità iniziale `20`, avremo un oggetto che ha `20` posizioni vuote (ovvero in ogni posizione c'è un reference che punta a `null`), che

possono essere riempite. Quando sarà aggiunto il ventunesimo elemento, l'`ArrayList` si ridimensionerà automaticamente per avere capacità ventuno, e questo avverrà per ogni nuovo elemento. Ovviamente, per ogni nuovo elemento che si vuole aggiungere oltre la capacità iniziale, l'`ArrayList` dovrà prima ridimensionarsi e poi aggiungere l'elemento. Questa doppia operazione porterà ad un decadimento delle prestazioni.
È però possibile ottimizzare le prestazioni di una `ArrayList` nel caso si vogliano aggiungere nuovi elementi superata la capacità iniziale. Infatti, quest'ultima si può modificare a nostro piacimento "al volo", in modo tale che l'`ArrayList` non sia costretto a ridimensionarsi per ogni nuovo elemento. Per fare ciò, basta utilizzare il metodo `ensureCapacity()` passandogli la nuova capacità, prima di chiamare il metodo `add()`. Per avere un'idea di quanto sia importante ottimizzare viene presentato un semplice esempio. Sfruttiamo il metodo statico `currentTimeMillis()` della classe `System` (cfr. prossimo paragrafo) per calcolare i millisecondi che impiega un ciclo a riempire l'`ArrayList`:

```
//capacità iniziale 1
ArrayList list = new ArrayList(1);
long startTime = System.currentTimeMillis();
list.ensureCapacity(100000);
for (int i = 0; i < 100000; i++) {
    list.add("nuovo elemento");
}
long endTime = System.currentTimeMillis();
System.out.println("Tempo = " + (endTime - startTime));
```

L'output sul mio portatile (processore Pentium 4 a 1.9 Ghz) è

```
Tempo = 20
```

Commentando la riga :

```
list.ensureCapacity(100000);
```

l'output cambierà:

```
Tempo = 50
```

La performance è nettamente peggiore, e la differenza sarà ancora più evidente aumentando il numero di elementi.
Se rimuoviamo un elemento da un `ArrayList`, la sua capacità non diminuisce. Esiste il metodo `trimToSize()`, per ridurre la capacità dell'`ArrayList` al numero degli elementi effettivi.
In generale la classe **`Vector`** offre prestazioni inferiori rispetto ad un `ArrayList` essendo sincronizzata. Per questo utilizza due parametri per configurare l'ottimizzazione delle prestazioni: la capacità iniziale e la capacità d'incremento. Per quanto riguarda la capacità iniziale vale quanto detto per `ArrayList`. La capacità d'incremento (specificabile tramite un costruttore), permette di stabilire di quanti posti si deve incrementare la capacità del vettore, ogni qual volta, si aggiunga un elemento che "sfora" il numero di posizioni disponibili. Se per esempio istanziamo un `Vector` nel seguente modo:

```
Vector v = new Vector(10, 10);
```

dove il primo parametro è la capacità iniziale e il secondo la capacità di incremento, quando aggiungeremo l'undicesimo elemento, la capacità del vettore sarà resettata a `20`. Quando aggiungeremo il ventunesimo elemento, la capacità del vettore sarà resettata a `30` e così via. Il seguente codice per esempio:

```
Vector list = new Vector(10,10);
for (int i = 0; i < 11; i++) {
    list.add("1");
}
System.out.println("Capacità = " + list.capacity());
for (int i = 0; i < 11; i++) {
    list.add("1");
}
System.out.println("Capacità = " + list.capacity());
```

produrrà il seguente output:

```
Capacità = 20
Capacità = 30
```

Se istanziamo un vettore senza specificare la capacità di incremento, oppure assegnandogli come valore un intero minore o uguale a zero, la capacità verrà

raddoppiata ad ogni "sforamento". Se quindi per esempio istanziamo un `Vector` nel seguente modo:

```
Vector v = new Vector();
```

dove non sono state specificate capacità iniziale (che di default viene settata a `10`) e capacità di incremento (che di default viene settata a `0`), quando aggiungeremo l'undicesimo elemento, la capacità del vettore sarà resettata a `20`. Quando aggiungeremo il ventunesimo elemento, la capacità del vettore sarà resettata a `40` e così via, raddoppiando la capacità del `Vector` tutte le volte che occorre ampliarlo.

12.1.5 Le interfacce Queue, BlockingQueue e ConcurrentMap

Le interfacce principali del framework Collections, come già detto sono diventate 9 dalla versione 1.5 di Java. Le ultime tre di cui non ci siamo ancora occupati sono `Queue`, `BlockingQueue` e `ConcurrentMap` (queste ultime due appartenenti al package `java.util.concurrent`).
L'interfaccia **`Queue`** (in italiano "coda"), estende l'interfaccia `Collection` definendo nuovi metodi per l'inserimento, la rimozione e l'utilizzo dei dati. Ognuno di questi metodi è presente in due forme: se l'operazione fallisce una forma lancia un'eccezione, e l'altra restituisce un valore speciale (per esempio `null` o `false`). Quindi, a seconda dell'esigenza, lo sviluppatore può usufruire di un metodo piuttosto che un altro. La seguente tabella riassume quanto appena asserito:

	Lancia eccezione	Ritorna valore speciale
Inserimento	`add(e)`	`offer(e)`
Rimozione	`remove()`	`poll()`
Recupero	`element()`	`peek()`

Le collection sono tutte ridimensionabili, e quindi l'inserimento è sempre possibile in generale. L'operazione potrebbe fallire però nel caso di implementazioni di `Queue` con dimensione limitata. In particolare il metodo `offer()` inserisce un elemento ritornando `true` o `false` qualora l'operazione di inserimento riesca oppure no. La differenza essenziale con il metodo `add()` (già definito nell'interfaccia `Collection`), è che quest'ultimo può fallire nell'aggiungere un

elemento solo lanciando un'unchecked exception. Il metodo `offer()` è invece progettato per essere utilizzato quando il fallimento di un inserimento non rappresenta un evento eccezionale, per esempio proprio nel caso di code con dimensione massima.

I metodi `remove()` e `poll()` ritornano e rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `remove()` ritorna `null`, mentre `poll()` lancia un'eccezione.
I metodi `element()` e `peek()` invece ritornano ma non rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `element()` ritorna `null`, mentre `peek()` lancia un'eccezione.
La "testa della coda" è definita dall'implementazione della coda. Esistono code FIFO (che sta per "First In First Out") che definiscono come testa della coda, l'ultimo elemento inserito. Un'implementazione di coda FIFO l'abbiamo già vista: la classe `LinkedList`, che mette a disposizione i metodi `addLast()`, `getLast()`, e `removeLast()`. In realtà `LinkedList` implementa anche la classe `Deque` e quindi può essere utilizzata come coda LIFO (che sta per "Last In First Out") che definiscono come testa della coda, il primo elemento inserito. Infatti mette a disposizione anche i metodi `addFirst()`, `getFirst()`, `removeFirst()`.
Un'altra tipologia di coda è definita dalle priority queue (classe `PriorityQueue`) che ordina i propri elementi a seconda del proprio ordinamento naturale o a seconda di un oggetto `Comparator` associato al momento della creazione.

Nel package `java.util.concurrent`, sono definite le interfacce `BlockingQueue` e `ConcurrentMap`. L'interfaccia **`BlockingQueue`** estende `Queue` e definisce nuovi metodi che bloccano con un thread apposito il recupero e la rimozione di un elemento fino a quando la coda diventa non vuota, e bloccano un inserimento fino a quando lo spazio nella blocking queue diventa disponibile. Infatti, è possibile limitare la capacità di una `BlockingQueue`, di solito mediante un costruttore (come accade per la classe `ArrayBlockingQueue`).

Se non si limita esplicitamente la capacità di una `BlockingQueue`, la capacità massima sarà pari al valore di `Integer.MAX_VALUE`, ovvero il più grande numero intero.

Oltre ai metodi che vengono ereditati dall'interfaccia `Queue`, `BlockingQueue` definisce altre due categorie di metodi, alcuni che appunto che bloccano l'inserimento e

la rimozione degli elementi secondo quanto detto prima, oppure che bloccano per un tempo massimo specificato (time out). Nel caso il tempo specificato passi prima che l'operazione di inserimento o rimozione sia possibile, allora viene restituito un valore booleano `false`. Segue una tabella esplicativa:

	Lancia eccezione	Ritorna valore speciale	Blocca	Time out
Inserimento	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Rimozione	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Recupero	`element()`	`peek()`	`N/A`	`N/A`

Per i metodi che gestiscono il time out, bisogna specificare come secondo parametro un `long` (`time` nella tabella) che rappresenta il tempo massimo per eseguire l'operazione. L'unità di tempo viene però specificata con il terzo parametro (`unit` nella tabella) che è di tipo `TimeUnit`. Si tratta di un'enumerazione che definisce come suoi elementi appunto delle unità temporali: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `NANOSECONDS`, `SECONDS` e `MINUTES`.
Un'interessante implementazione dell'interfaccia `BlockingQueue`, è `LinkedBlockingQueue`.
Infine l'interfaccia **`ConcurrentMap`**, estende l'interfaccia `Map` definendo metodi come `putIfAbsent()`, `remove()` e `replace()`. Tutte le azioni di inserimento e di rimozione degli elementi sono thread safe, e le sue implementazioni come `ConcurrentHashMap`, sono adatte per sistemi dove c'è l'esigenza di avere mappe condivise.

12.1.6 Algoritmi e utilità

Le classi `Collections` ed `Arrays`, offrono una serie di metodi statici che rappresentano utilità ed algoritmi che possono essere utili con le collection.
In particolare la classe **`Collections`**, offre particolari metodi (detti wrapper) per operare su collezioni in modo tale da modificare il loro stato. In realtà tali metodi prendono in input una collezione e ne restituiscono un'altra con le proprietà cambiate. Per esempio i seguenti metodi:

- `public static Collection unmodifiableCollection(Collection c)`
- `public static Set unmodifiableSet(Set s)`
- `public static List unmodifiableList(List list)`
- `public static Map unmodifiableMap(Map m)`
- `public static SortedSet unmodifiableSortedSet(SortedSet s)`
- `public static SortedMap unmodifiableSortedMap(SortedMap m)`

restituiscono delle copie immutabili di ogni collection. Infatti se si prova a modificare la collezione scatterà una `UnsupportedOperationException`.
I seguenti metodi invece:

- `public static Collection synchronizedCollection(Collection c)`
- `public static Set synchronizedSet(Set s)`
- `public static List synchronizedList(List list)`
- `public static Map synchronizedMap(Map m)`
- `public static SortedSet synchronizedSortedSet(SortedSet s)`
- `public static SortedMap synchronizedSortedMap(SortedMap m)`

sincronizzano tutte le tipologie di collection.
Per esempio anche l'`ArrayList` si può sincronizzare sfruttando uno dei tanti metodi statici di utilità della classe `Collections` (da non confondere con `Collection`):

```
List list = Collections.synchronizedList(new
ArrayList(...));
```

Abbiamo già asserito che in generale le prestazioni di un `Vector` sono inferiori rispetto a quelle di un `ArrayList`. La situazione in questo caso però si capovolge: il `Vector` ha performance superiori a quelle di un `ArrayList` sincronizzato.

Sincronizzare una collection non significa sincronizzare anche il rispettivo `Iterator`. È quindi obbligatorio ciclare con un `Iterator` quantomeno all'interno di un blocco sincronizzato come nel seguente esempio, (pratica consigliata direttamente dal Java Tutorial della Sun):

```
Collection c =
Collections.synchronizedCollection(myCollection);
synchronized(c) {
    Iterator i = c.iterator();
    while (i.hasNext())
        faQualcosa(i.next());
}
```

Questo perché l'iterazione avviene con chiamate multiple sulla collezione che vengono composte in un'unica operazione.
Discorso simile anche per le mappe, ma da studiare direttamente dalla documentazione del metodo `synchronizedMap()`.

I metodi wrapper della classe `Collections` hanno come lato negativo il fatto che catturano con un reference di tipo interfaccia, un oggetto di tipo implementazione. Questo implica che non sarà più possibile utilizzare tutti i metodi dell'implementazione. Per esempio se abbiamo:
`List list = Collections.synchronizedList(new ArrayList());`
con il reference `list` (di tipo `List`) non sarà più possibile chiamare il metodo `ensureCapacity()` dell'`ArrayList`, perché non definito dell'interfaccia `List`. Ovviamente è sempre possibile ricorrere al casting quando è possibile.

Oltre ai metodi wrapper, `Collections` ha metodi che implementano per noi complicati algoritmi come `sort()` (ordina), `shuffle()` (mischia – il contrario di ordina), `max()` (resituisci l'elemento massimo), `reverse()` (inverti l'ordine degli elementi), `bynarySearch()` (ricerca binaria) e così via…
Altri metodi (detti "di convenienza"), permettono la creazione di collection immutabili di un numero definito di oggetti identici (metodo `nCopies()`), o di un oggetto singleton, che si può istanziare una sola volta (metodo `singleton()`).

La classe **`Arrays`**, contiene alcuni dei metodi-algoritmi di `Collections`, ma relativi ad array, come `sort()` e `bynarySearch()`. Inoltre possiede il metodo `asList()` che può trasformare un array in una `List` (precisamente un oggetto di tipo `Arrays.ArrayList`… vedi documentazione…). Per esempio:

```
List l = Arrays.asList(new String[] { "1", "2", "3"});
```

crea un'implementazione di `List` a partire dall'array specificato al volo come parametro (di cui non rimarrà nessun puntamento e quindi verrà poi "garbage collected").
Con le versione 5 sono stati aggiunti nuovi metodi molto interessanti per la classe `Arrays`.
Per esempio dalla versione 5 esistono 9 versioni del metodo `toString(array)`, overridato per prendere in input tutte le possibili combinazioni di array di tipi primitivi e di `Object`. In pratica si potrà stampare all'interno di parentesi quadre tutti gli elementi di un array (separati da virgole), senza ricorrere ad un ciclo.

12.1.7 Collection personalizzate

Infine bisogna dire che ogni collection si può estendere per creare collection personalizzate. Se volete estendere qualcosa di meno definito rispetto ad un `Vector` o un `ArrayList`, avete a disposizione una serie di classi astratte:

- `AbstractCollection`: implementazione minimale di `Collection`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractSet`: implementazione di un `Set`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractList`: implementazione minimale di una `list`. Bisogna definire i metodi `get(int)` e, opzionalmente, `set(int)`, `remove(int)`, `add(int)` e `size()`.
- `AbstractSequentialList`: implementazione di un `LinkedList`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractMap`: implementazione di una `Map`. Bisogna definire il metodo `entrySet()` (che è solitamente implementato con la classe `AbstractSet`), e se la mappa deve essere modificabile occorre definire anche il metodo `put()`.

Esistono tantissime altre implementazioni di `Collection`. Sarà compito del lettore approfondire l'argomento con la documentazione ufficiale.

12.1.8 Collections e Generics

Se il lettore ha provato ad utilizzare qualche `Collection` con un JDK 1.5 o superiore, avrà sicuramente notato alcuni messaggi di warning dopo aver compilato. Questi sono dovuti all'introduzione di una rivoluzionaria novità del linguaggio: i **Generics**. Probabilmente si tratta della caratteristica con il più alto impatto sul linguaggio, tra quelle introdotte in Tiger.
I Generics, offrono la loro più classica utilità nell'uso delle `Collection`. Abbiamo visto come il fatto che le `Collection` siano delle collezioni eterogenee, diano loro grande flessibilità nel contenere dati diversi. Purtroppo però, l'utilità di una collezione "molto" eterogenea come la seguente:

```
List list = new ArrayList();
list.add("Stringa");
list.add(new Date());
list.add(new ArrayList());
```

è ridotta a pochissimi casi. Inoltre l'estrazione dei dati dalla precedente `Collection`, è una operazione particolarmente delicata ed incline all'errore (per via del casting):

```
String string = (String)list.get(0);
Date date = (Date) list.get(1);
ArrayList arrayList = (ArrayList)list.get(2);
```

In particolare può facilmente portare al lancio di una `ClassCastException` al runtime. Per evitare ciò, oltre che al casting bisogna prima testarne la validità del tipo con l'operatore `instanceof`.

```
String string = null;
Date date = null;
ArrayList arrayList = null;
for (int i = 0; i < list.size(); ++i){
    Object object = list.get(i);
    if (object instanceof String) {
      string = (String)object;
    } else if (object instanceof Date) {
```

```
      date = (Date)object;
    } else if (object instanceof ArrayList) {
      arrayList = (ArrayList)object;
    }
}
```

È certo che in casi come questo non si può parlare né di efficienza, né di codice elegante. In realtà, nella maggior parte dei casi, le collezioni eterogenee, sono composte da oggetti che appartengono alla stessa gerarchia. Per esempio, è facile immaginare che una lista come la seguente:

```
List list = new ArrayList();
list.add(new Auto("Fiat Punto"));
list.add(new Auto("Ford Fiesta"));
list.add(new Moto("Ducati 999"));
. . .
```

possa essere utile all'interno di un'applicazione gestionale per un concessionario di veicoli generico.
Essendo questo l'utilizzo più frequente di una `Collection`, rimane sempre il problema dell'estrazione dei dati dalla `List`, che potrebbe richiedere (laddove il polimorfismo non dovesse bastare, cfr. Modulo 6) l'utilizzo del casting e dell'operatore `instanceof`.
I cosiddetti "**Tipi Generics**", permettono di fare in modo che una particolare `Collection`, sia parametrizzata con un certo tipo. La sintassi fa uso di parentesi angolari. Segue un primo esempio:

```
Vector<String> vector = new Vector<String>();
```

In questo caso abbiamo dichiarato un vettore parametrizzato con la classe `String`. Questo significa che la nostra `Collection` potrà contenere solo e solamente stringhe.

La classe `String` non ha sottoclassi essendo stata dichiarata `final`.

Quindi se provassimo ad aggiungere a `vector` un eventuale oggetto che non sia di tipo `String` come nel seguente esempio:

```
vector.add(new StringBuffer("Tiger"));
```

Otterremo un errore in compilazione. Quindi, ogni eventuale tentativo di uso scorretto della nostra `Collection`, verrà rilevato in fase di compilazione, evitandoci di perdere ore in debug evitabili. Java diventa così un linguaggio ancora più robusto, sicuro e fortemente tipizzato.
Questo è un vantaggio assolutamente non trascurabile, ecco perché il compilatore ci segnalerà un warning nel caso utilizzassimo "raw type" (così vengono definite a partire da Java 5 le `Collection` che non sono parametrizzate tramite generics).
È assolutamente indispensabile quindi, che il lettore inizi ad utilizzare da subito i Generics, anche perché l'approccio alla sintassi, e soprattutto alla mentalità, non è sicuramente dei più semplici.
Inoltre, dal momento che sappiamo che il nostro oggetto `vector`, conterrà solo e solamente stringhe, allora ogni casting è assolutamente superfluo. È quindi possibile estrarre i dati dall'oggetto `vector` come nel seguente esempio:

```
String stringa = null;
for (int i = 0; i < vector.size(); ++i) {
    stringa = vector.elementAt(i);
    . . .
}
```

In questo modo non andremo incontro a nessun tipo di `ClassCastException`.

12.1.9 La classe Properties

Questa classe rappresenta un insieme di proprietà che può diventare persistente in un file. Estende la classe `Hashtable`, e quindi ne eredita i metodi ma ha la caratteristica di poter leggere e salvare tutte le coppie chiave-valore, in un file, mediante i metodi `load()` e `store()`. Ovviamente, se il lettore non ha ancora studiato il modulo relativo all'input-output, deve rimandare ad un secondo momento la comprensione del meccanismo per accedere ad un file. Intanto vediamo un esempio di utilizzo. Se dovesse risultare troppo complicato (improbabile) è sempre possibile rileggere questo paragrafo in un secondo momento.
Come esempio prendiamo proprio i file sorgenti di EJE (disponibili per il download all'indirizzo http://www.claudiodesio.com/eje.htm e http://sourceforge.net/projects/eje). Il "file di properties" chiamato "EJE_options.properties", viene infatti utilizzato per leggere e salvare le preferenze dell'utente come la lingua, la versione di Java, la dimensione dei caratteri e così via. Con il seguente codice (file "EJE.java"), tramite un blocco statico viene chiamato il metodo `loadProperties()`:

```
static {
  try {
    properties = new Properties();
    try {
      loadProperties();
      . . .
    } catch (FileNotFoundException e) {
      . . .
    }
}
```

Ovviamente la variabile `properties` era stata dichiarata come variabile d'istanza.
Segue la dichiarazione del metodo `loadProperties()`:

```
public static void loadProperties() throws
      FileNotFoundException {
    InputStream is = null;
    try {
        is = new
FileInputStream("resources/EJE_options.properties");
        properties.load(is);
    } catch (FileNotFoundException e) {
      . . .
    }
}
```

Per esempio per settare la lingua, successivamente viene letta la proprietà con chiave "eje.lang":

```
String language = EJE.properties.getProperty("eje.lang");
```

dove `EJE.properties` è un oggetto di tipo `Properties`.
Quando invece si vuole salvare un proprietà appena settata, viene chiamato il seguente metodo:

```
public static void saveProperties() {
    OutputStream os = null;
```

```
    try {
        os = new
FileOutputStream("resources/EJE_options.properties");
        EJE.properties.store(os,
        "EJE OPTIONS - DO NOT EDIT");
    } catch (FileNotFoundException e) {
      . . .
    }
}
```

Per esempio dopo che è stato aperto un file, il suo nome viene salvato nella lista dei file recenti come primo (ovviamente dopo aver shiftato tutti gli altri di un posto):

```
EJE.properties.setProperty("file.recent1", file);
saveProperties();
```

Il file di properties, a volte viene salvato dopo averlo modificato "a mano". Nel caso un valore di una certa chiave sia molto lungo, e lo si desideri riportare su più righe, bisogna utilizzare il simbolo di slash "/" alla fine di ogni riga quando si vuole andare da capo. Per esempio:
```
eje.message=This message is very very very very /
very long
```

12.1.10 La classe Locale ed internazionalizzazione

Con il termine internazionalizzazione, intendiamo il processo di progettare un'applicazione che si possa adattare a vari linguaggi e zone, senza modificare il software. In inglese, il termine "internationalization" è spesso abbreviato in i18n, poiché è un termine difficile da scrivere e pronunciare, e ci sono 18 lettere tra la "i"iniziale e la "n" finale.
Per gestire l'internazionalizzazione non si può fare a meno di utilizzare la classe `Locale`. La classe `Locale` astrae il concetto di "zona". Molte rappresentazioni di molte altre classi Java, dipendono da `Locale`. Per esempio, per la rappresentazione di un numero decimale, si utilizza la virgola come separatore tra il numero intero e le cifre decimali in Italia, mentre in America si utilizza il punto. Ecco che allora la classe `NumberFormat` (package `java.text`) ha dei metodi che utilizzano `Locale` per individuare la rappresentazione di numeri decimali. Altri esempi di concetti che dipendono da `Locale` sono date, orari e valute (vedi classe `Currency`).
La classe `Locale`, è basata essenzialmente su tre variabili: `language`, `country` e

`variant` (vedi costruttori). Inoltre possiede alcuni metodi che restituiscono informazioni sulla zona. Ma probabilmente, più che interessarci dei metodi di `Locale`, ci dovremmo interessare all'utilizzo che altre classi fanno di `Locale`.
Infatti, nella maggior parte dei casi, non andremo ad istanziare `Locale`, bensì ad utilizzare le diverse costanti statiche di tipo `Locale` che individuano le zone considerate più facilmente utilizzabili. Esempi sono `Locale.US`, o `Locale.ITALY`.
A volte potrebbe risultare utile il metodo `getDefault()`, che restituisce l'oggetto `Locale` prelevato dalle informazioni del sistema operativo.
Come esempio segue il codice con cui EJE va a riconoscere il `Locale` da utilizzare per settare la lingua (cfr. prossimo paragrafo sul `ResourceBundle` per i dettagli del settaggio della lingua)

```
static {
  . . .
    Locale locale = null;
    String language =
      EJE.properties.getProperty("eje.lang");
    locale = (language != null && !language.equals("")
      ? new Locale(language) : Locale.getDefault());
     . . .
}
```

In questo caso viene prima letto dal file di properties (cfr. paragrafo precedente) il valore di `eje.lang`. Poi viene controllato che sia stato valorizzato in qualche modo. Se è stato valorizzato, tramite un operatore ternario, viene assegnato alla variabile `locale` un istanza di `Locale` inizializzata con il valore della variabile `language`. Se `language` non è mai stato settato, (condizione che dovrebbe presentarsi appena scaricato EJE), la variabile `locale` viene settata al suo valore di default.

12.1.11 La classe ResourceBundle

La classe `ResourceBundle`, rappresenta un contenitore di risorse dipendente da `Locale`. Per esempio, EJE "parlerà" inglese o italiano sfruttando un semplice meccanismo basato sul `ResourceBundle`. Infatti, EJE utilizza tre file che si chiamano EJE.properties , EJE_en.properties, EJE_it.properties che come al solito contengono coppie del tipo chiave-valore. I primi due contengono tutte le stringhe personalizzabili in inglese (sono identici), il terzo le stesse stringhe in italiano.

Con l'ultimo esempio di codice, abbiamo visto come viene scelto il `Locale`. Ci sono tre possibilità per EJE:

1) Il `Locale` è esplicitamente italiano perché la variabile `language` vale "`it`"
2) Il `Locale` è inglese perché la variabile `language` vale "`en`"
3) Il `Locale` è quello di default (quindi potrebbe essere anche francese), perché la variabile `language` non è stata valorizzata.

Per EJE il default è comunque inglese, infatti i file EJE.properties ed EJE_en.properties, come già asserito, sono identici.

Il codice che setta il `ResourceBundle` (che si chiama `resources`) da utilizzare è il seguente:

```
static {
    . . .
    if (language != null && !language.equals("")) {
        locale = . . .
        . . .
        resources =
ResourceBundle.getBundle("resources.EJE", locale);
    } catch (MissingResourceException mre) {
            . . .
        }
    }
}
```

Dove la variabile `locale` è stata settata come abbiamo visto nell'esempio precedente. In pratica, con il metodo `getBundle()`, stiamo chiedendo di caricare un file che si chiama EJE, nella directory `resources`, con il `locale` specificato. In maniera automatica, se il `locale` specificato è italiano viene caricato il file EJE_it.properties, se il `locale` è inglese viene caricato il file EJE_en.properties, se il `locale` è per esempio francese EJE prova a caricare un file che si chiama EJE_fr.properties che però non esiste e quindi viene caricato il file di default EJE.properties! Tutto sommato il codice scritto è minimo.

Nel resto dell'applicazione quando bisogna visualizzare una stringa, viene semplicemente chiesto al `ResourceBundle` mediante il metodo `getString()`. Per esempio nel seguente modo:

```
newMenuItem = new
JMenuItem(resources.getString("file.new"),
  new ImageIcon("resources" + File.separator + "images" +
  File.separator + "new.png"));
```

la voce di menu relativa al comando "nuovo file" carica la sua stringa direttamente dal `ResourceBundle resources`.

Se il lettore è interessato a scrivere un file di properties per EJE diverso da italiano ed inglese (per esempio francese, spagnolo, tedesco etc…). può tranquillamente contattare l'autore all'indirizzo eje@claudiodesio.com. Sarà ovviamente citato nella documentazione in linea del software.

12.1.12 Date, orari e valute

Le classi **`Date`**, **`Calendar`**, **`GregorianCalendar`**, **`TimeZone`**, e **`SimpleTimeZone`**, permettono di gestire ogni tipologia di data ed orario. È possibile anche gestire tali concetti sfruttando l'internazionalizzazione. Non sempre sarà semplice gestire date ed orari, e questo è dovuto proprio alla complessità dei concetti (a tal proposito basta leggere l'introduzione della classe `Date` nella documentazione ufficiale). La complessità è aumentata dal fatto che oltre a queste classi, è molto probabile che serva utilizzarne altre quali `DateFormat` e `SimpleDateFormat` (package `java.text`), che permettono la trasformazione da stringa a data e viceversa. Vista la vastità del discorso, preferiamo solo accennare qualche esempio di utilizzo comune di tali concetti. Per il resto si rimanda direttamente alla documentazione ufficiale.
Per esempio, il seguente codice crea un oggetto `Date` (contenente la data corrente) e ne stampa il contenuto, formattandolo con un oggetto `SimpleDateFormat`, che utilizza il pattern `"dd-MM-yyyy"`, ovvero due cifre per il giorno, due per il mese, e quattro per l'anno:

```
Date date = new Date();
SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
System.out.println(df.format(date));
```

I due seguenti frammenti di codice invece, producono esattamente lo stesso output di formattazione di tempi. Nel primo caso viene utilizzato un pattern personalizzato:

```
DateFormat formatter = new SimpleDateFormat("HH:mm:ss");
String s = formatter.format(new Date());
s = DateFormat.getTimeInstance(DateFormat.MEDIUM,
locale).format(new Date());
System.out.println(s);
```

nel seguente codice invece raggiungiamo lo stesso risultato chiedendo di formattare allo stesso in maniera breve (`DateFormat.SHORT`), secondo lo stile Italiano:

```
Locale locale = Locale.ITALY;
DateFormat formatter =
DateFormat.getTimeInstance(DateFormat.SHORT, locale);
String s = formatter.format(new Date());
s = DateFormat.getTimeInstance(DateFormat.MEDIUM,
locale).format(new Date());
System.out.println(s);
```

Nel prossimo esempio formattiamo due valute (o numeri) secondo gli standard americano ed italiano:

```
double number = 55667788.12345;
Locale localeUsa = Locale.US;
Locale localeIta = Locale.ITALY;

NumberFormat usaFormat =
NumberFormat.getInstance(localeUsa);
String usaNumber = usaFormat.format(number);
System.out.println(localeUsa.getDisplayCountry() + " " +
usaNumber);

NumberFormat itaFormat =
NumberFormat.getInstance(localeIta);
```

```
String itaNumber = itaFormat.format(number);
System.out.println(localeIta.getDisplayCountry() + " " +
itaNumber);
```

Ouput:

```
Stati Uniti 55,667,788.123
Italia 55.667.788,123
```

Notare come può essere utile l'oggetto `Locale` per certe formattazioni.
Nel prossimo esempio, utlizziamo una classe molto importante, non dal punto di vista della formattazione delle date, ma proprio per il calcolo: `Calendar` (e la sua sottoclasse `GregorianCalendar`). Se non trovate i metodi che vi servono nella classe `Date`, date un occhiata alla documentazione di queste classi… potreste risolvere i vostri dubbi. Per esempio, il metodo `get()` utilizzando le costanti di `Calendar` ci permette recuperare i "pezzi" delle date. Per esempio:

```
public int getNumeroSettimana(Date date){
  Calendar calendar = new GregorianCalendar();
  calendar.setTime(date);
  int settimana = calendar.get(Calendar.WEEK_OF_YEAR);
  return settimana;
}
```

La classe `GregorianCalendar`, possiede anche un costruttore che prende come argomento un oggetto di tipo `TimeZone`. Con questo è possibile per esempio ottenere orari relativi ad altre parti del mondo. Per esempio:

```
Calendar cal = new
GregorianCalendar(TimeZone.getTimeZone("America/Denver"));
int hourOfDay = cal.get(Calendar.HOUR_OF_DAY);
```

permette di vedere l'ora attuale a Denver negli Stati Uniti. Per ottenere tutti gli id validi per il metodo `getTimeZone()`, è possibile invocare il metodo statico `getAvailableIDs()` della stessa classe `TimeZone`, che restituisce un array di stringhe.

12.1.13 La classe StringTokenizer

`StringTokenizer` è una semplice classe che permette di separare i contenuti di una stringa in più parti ("token"). Solitamente si utilizza per estrarre le parole in una stringa. I costruttori vogliono in input una stringa, e permettono di "navigare su di essa" per estrarne i token. Ma uno string tokenizer ha bisogno di sapere anche come identificare i token. Per questo si possono esplicitare anche quali sono i delimitatori dei token. Un token è quindi in generale, la sequenza massima di caratteri consecutivi che non sono delimitatori. Per esempio:

```
StringTokenizer st = new StringTokenizer("questo è un
test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Stampa il seguente output:

```
questo
è
un
test
```

Questo perché con il costruttore non abbiamo specificato i delimitatori, che per default saranno i seguenti: " \t\n\r\f" (notare che il primo delimitatore è uno "spazio"). Se utilizzassimo il seguente costruttore:

```
StringTokenizer st = new StringTokenizer("questo è un
test", ";");
```

l'output sarebbe costituito da un unico token, dato che non esiste quel delimitatore nella stringa specificata:

```
questo è un test
```

Esiste anche un terzo costruttore, che prende in input un booleano. Se questo booleano vale `true` allora lo string tokenizer considererà token anche gli stessi delimitatori. Quindi l'output del seguente codice:

```
StringTokenizer st = new StringTokenizer("questo è un
test", "t", true);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà:

```
ques
t
o è un
t
es
t
```

mentre se utilizzassimo questo costruttore:

```
StringTokenizer st = new StringTokenizer("questo è un
test", "t", true);
```

l'output sarebbe:

```
ques
o è un
es
```

12.1.14 Espressioni regolari

Esiste un modo però molto più potente di analizzare del testo in Java. Infatti, dalla versione 1.4 Java supporta le cosiddette "**espressioni regolari**" (in inglese "**regular expressions**"). Si tratta di una potente tecnica che esiste anche in altri ambienti primi tra tutti, Unix e il linguaggio Perl, per la ricerca e l'analisi del testo.
In pratica si basa su di un linguaggio sintetico e a volte criptico, fatto appunto di espressioni, con una sintassi non ambigua, per poter individuare determinate aree di testo. Per esempio, l'espressione

```
[aeiou]
```

Permette di individuare una vocale. La sintassi è abbastanza vasta e quindi ci limiteremo a riportare solo quella necessaria a superare l'esame SCJP.
Essenzialmente, ci occorre conoscere tre tipologie di espressioni: i **gruppi di caratteri** (character classes), le **classi predefinite** e i **quantificatori** (greedy quantifier). Un esempio di gruppi di caratteri l'abbiamo appena visto:

```
[aeiou]
```

Che individua una tra le vocali. Per esempio l'espressione:

```
alunn[ao]
```

permette di individuare all'interno di un testo le occorrenze sia della parola "alunno" che della parola "alunna". L'operatore `^` è detto "operatore di negazione", e per esempio l'espressione:

```
[^aeiou]
```

Individua un qualsiasi carattere escluse le vocali (quindi una consonante).
Con i gruppi di caratteri è anche possibile specificare range di caratteri con il separatore `-`. Per esempio l'espressione:

```
[a-zA-Z]
```

permette di individuare una qualsiasi lettera maiuscola o minuscola.
Le classi predefinite sono particolari classi che permettono con una sintassi abbreviata di definire alcune espressioni. Segue uno schema con tutte le classi predefinite e le relative descrizioni:

- `.` Un qualsiasi carattere
- `\d` Una cifra (equivalente a `[0-9]`)
- `\D` Una non-cifra (equivalente a `[^0-9]`)
- `\s` Un carattere spazio bianco (equivalente a `[\t\n\x0B\f\r]`)
- `\S` Un caratter non-spazio bianco (equivalente a `[^\s]`)
- `\w` Un carattere (equivalente a `[a-zA-Z_0-9]`)
- `\W` Un non carattere (equivalente a `[^\w]`)

Per quanto riguarda i quantificatori, questi servono a indicare la molteplicità di caratteri, parole, espressioni o gruppi di caratteri. In particolare:

- `*` significa zero o più occorrenze del pattern precedente
- `+` significa una o più occorrenze del pattern precedente
- `?` significa zero o una o più occorrenze del pattern precedente

> **In realtà stiamo parlando solo di una tipologia di quantificatori (i greedy quantifier) ma ne esistono altre con potenzialità notevoli…anche troppo.**

La libreria Java supporta con tante classi le regular expression. Per esempio il metodo `replace()` della classe `String`, oppure la classe `Scanner` che introdurremo nel prossimo modulo (che potrebbe tranquillamente sostituirsi anche alla classe `StringTokenizer`, ma usando regular expression). Ma in realtà esiste il package `java.util.regex`, che definisce due semplici classi, sulla quali si basa tutta la libreria che utilizza le espressioni regolari. Si tratta della classe `Pattern` e `Matcher`. La classe `Pattern`, serve proprio a definire le espressioni regolari mediante il suo metodo statico `compile()`. La classe `Matcher` invece definisce diversi metodi per la ricerca e l'analisi del testo, come i metodi `matches()`, `find()`, `start()`, `end()`, `replaceFirst()`, `replaceAll()`, etc…
Per esempio EJE fa grande uso di regular expression per interpretare il codice digitato dallo sviluppatore. Per esempio il seguente codice:

```
Pattern p = Pattern.compile("\\bpackage\\b");
String content = EJEArea.this.getText();
Matcher m = p.matcher(content);
while (m.find()) {
    int start = m.start();
    int end = m.end();
    . . .
```

Permette di ricercare la posizione della parola "package" all'interno del testo ciclando su ogni occorrenza trovata. Tenere presente che la variabile `content` contiene il testo digitato dall'utente su EJE, e che il simbolo `\b` rappresenta il delimitatore dell'espressione.

Notare come sia stato necessario utilizzare due simboli di backslash invece che uno solo. Infatti il primo è necessario alla sintassi Java per interpretare il secondo come simbolo di backslash!

12.2 Introduzione al package java.lang

Il package `java.lang`, oltre a contenere la classe `Thread` e l'interfaccia `Runnable`, di cui abbiamo ampiamente parlato nel modulo precedente, contiene altre decine di classi che riguardano da vicino il linguaggio stesso. Fortunatamente, nessuna di esse darà luogo ad un discorso complesso come quello sui `Thread`! Per esempio, il package `java.lang` contiene la classe `String`, la classe `Object` (cfr. Modulo 3), e la classe `System` (cfr. prossimo paragrafo), che sono già state ampiamente utilizzate in questo testo.
La descrizione della documentazione ufficiale descrive così questo package: "Provides classes that are fundamental to the design of the Java programming language", ovvero "fornisce classi fondamentali per la progettazione del linguaggio Java". Ricordiamo che stiamo parlando dell'unico package automaticamente importato in ogni programma Java. In questo modulo introdurremo le classi più utilizzate e famose, con lo scopo di dare una visione di insieme del package.

12.2.1 La classe String

La classe `String`, è già stata esaminata più volte in questo testo, in particolare nel Modulo 3. Abbiamo già visto come `String` sia l'unica classe che sia possibile istanziare come se fosse un tipo di dato primitivo. Inoltre, abbiamo anche visto che tali istanze vengono inserite in una pool di stringhe allo scopo di migliorare le performance, e che tutti gli oggetti siano immutabili. Da momento che la certificazione SCJP lo richiede, vengono di seguito elencati i metodi più importanti:

- `char charAt(int index)` ritorna il carattere all'indice specificato (indice iniziale `0`)
- `String concat(String other)` ritorna un nuova stringa che concatena la vecchia con la nuova (`other`)
- `int compareTo(String other)` esegue una comparazione lessicale: ritorna un `int < 0` se la stringa corrente è minore della stringa `other`, un intero `= 0` se le due stringhe sono identiche, e un intero `> 0` se la stringa corrente è maggiore di `other`

- `boolean endsWith(String suffix)` ritorna `true` se e solo se la stringa corrente termina con `suffix`
- `boolean equals(Object ob)` di questo metodo abbiamo già discusso ampiamente
- `boolean equalsIgnoreCase(String s)` metodo equivalente ad `equals()` che ignora la differenza tra lettere maiuscole e minuscole.
- `int indexOf(int ch)` ritorna l'indice del carattere specificato. Notare che in realtà potremmo passare anche un intero come parametro. Questo perché così sarà possibile anche passare la rappresentazione Unicode del carattere da cercare (per esempio `0XABCD`). Ritorna `-1` nel caso non venga trovato il carattere richiesto. Se esistessero più occorrenze nella stringa del carattere richiesto, verrebbe ritornato l'indice della prima occorrenza.
- `int indexOf(int ch, int fromIndex)` equivalente al metodo precedente, ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi. Di questi due ultimi metodi esistono versioni con `String` al posto di `int`.
- `int lastIndexOf(int ch)` come `indexOf()`, ma viene restituito l'indice dell'ultima occorrenza trovata del carattere specificato.
- `int length()` restituisce il numero di caratteri di cui è costituita la stringa corrente.
- `String replace(char oldChar, char newChar)` restituisce una nuova stringa, dove tutte le occorrenze di `oldChar`, sono rimpiazzate con `newChar`
- `boolean startsWith(String prefix)` restituisce `true` se e solo se la stringa corrente inizia con la stringa `prefix`
- `boolean startsWith(String prefix, int fromIndex)` equivalente al metodo precedente, ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi.
- `String substring(int startIndex)` restituisce una sotto-stringa della stringa corrente, composta dai caratteri che partono dall'indice `startIndex` alla fine
- `String substring(int startIndex, int number)` restituisce una sotto-stringa della stringa corrente, composta dal numero `number` di caratteri che partono dall'indice `startIndex`
- `String toLowerCase()` restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri minuscoli

- `String toString()` restituisce la stringa corrente (!)… comunque era ereditato dalla classe `Object`…
- `String toUpperCase()` restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri maiuscoli
- `String trim()` restituisce una nuova stringa privata dei caratteri spazio, '`\n`' e '`\r`', se questi si trovano all'inizio o alla fine della stringa corrente

12.2.2 La classe System

La classe `System` astrae il sistema su cui si esegue il programma Java. Tutto ciò che esiste nella classe `System` è dichiarato statico. Abbiamo più volte utilizzato tale classe quando abbiamo usato l'istruzione per stampare `System.out.println()`. In realtà per stampare abbiamo utilizzato il metodo `println()` della classe `PrintStream`. Infatti il metodo è invocato sull'oggetto statico `out` della classe `System`, che è di tipo `PrintStream`. L'oggetto `out` rappresenta l'output di default del sistema che dipende ovviamente dal sistema dove gira l'applicativo.
Esiste anche l'oggetto `err` (`System.err`) che rappresenta l'error output di default del sistema. Anche esso è di tipo `PrintStream` e di solito coincide con l'output di default del sistema. Infatti, su di un sistema Windows, sia l'output di un programma che gli eventuali errori prodotti dal programma stesso vengono visualizzati su la finestra DOS, da dove si è lanciato l'applicativo.
Esiste anche un oggetto statico `in` (`System.in`), che astrae il concetto di input di default del sistema. È di tipo `InputStream` (package `java.io`) e solitamente individua la tastiera del vostro computer.
È anche possibile modificare il puntamento di queste variabili verso altre fonti di input o di output. Esistono infatti i metodi statici `setOut()`, `setErr()` e `setIn()`. Nel modulo relativo al package `java.io` (input-output), vedremo anche un esempio di utilizzo dell'oggetto `System.in` (leggeremo i caratteri digitati sulla tastiera) e di come si può alterare l'indirizzamento di questi oggetti.
Dopo aver parlato delle variabili membro della classe `System`, diamo un sguardo ai metodi più interessanti.
Abbiamo già accennato al metodo `arraycopy()` (cfr. Modulo 3), che permetteva di copiare il contenuto di un array in un altro.
Sicuramente più interessante è il metodo statico `exit(int code)`, che permette di stoppare istantaneamente l'esecuzione del programma. Il codice che viene specificato come parametro, potrebbe servire al programmatore per capire il perché si è interrotto il

programma. Un sistema piuttosto rudimentale dopo aver visto gestione delle eccezioni ed asserzioni… segue un esempio di utilizzo di questo metodo:

```
if (continua == false) {
    System.err.println("Si è verificato un problema!");
    System.exit(0);
}
```

I metodi `runFinalization()` e `gc()`, richiedono alla virtual machine rispettivamente la "finalizzazione" degli oggetti inutilizzati e di eventualmente liberarne la memoria. La "finalizzazione" consiste nel testare se esistono oggetti non più "raggiungibili" da qualche reference, e quindi non utilizzabili. Questo viene fatto mediante la chiamata al metodo `finalize()` della classe `Object` (e quindi ereditato in ogni oggetto). Nel caso in cui l'oggetto non sia più reputato utilizzabile dalla Java Virtual machine, questo viene "segnato", e il successivo passaggio della garbage collection (la chiamata al metodo `gc()`) dovrebbe deallocare la memoria allocata per l'oggetto in questione.
Tuttavia, è sconsigliato l'utilizzo di questa coppia di metodi perché la JVM stessa, ha un meccanismo ottimizzato per gestire il momento giusto per chiamare questa coppia di metodi. Per di più, questi due metodi fanno partire dei thread tramite il metodo `start()`, che, come abbiamo visto nel modulo precedente, non garantisce l'esecuzione immediata del metodo.
Altri metodi interessanti sono i metodi `setProperty(String key, String value), getProperty(String key)`. Come abbiamo già affermato, la classe `System` astrae il sistema dove girà l'applicazione. Il sistema ha determinate proprietà prestabilite. Per esempio:

```
System.out.println(System.getProperty("java.version"));
```

Restituirà la versione di Java che si sta utilizzando. È anche possibile impostare nuove proprietà all'interno della classe `System`, per esempio:

```
System.setProperty("claudio.cognome", "De Sio Cesari")
```

Per un elenco completo di tutte le proprietà disponibili di default all'interno della classe `System`, rimandiamo il lettore al modulo relativo all'Input-Ouput.

12.2.3 La classe Runtime

La classe `Runtime` invece, astrae il concetto di runtime (esecuzione) del programma. Non ha costruttori pubblici ed una sua istanza si ottiene chiamando il metodo factory `getRuntime()`.
Caratteristica interessante di questa classe, è quella di permettere di eseguire comandi del sistema operativo direttamente da programma Java. Per esempio, il metodo `exec()` (di cui esistono più versioni), utilizzato in questo modo:

```
Runtime r = Runtime.getRuntime();
try {
    r.exec("calc");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Lancerà su di un sistema Windows la famosa calcolatrice.

Ovviamente, bisogna tener conto che l'utilizzo della classe `Runtime` potrebbe ovviamente compromettere la portabilità delle nostre applicazioni. Infatti, la classe `Runtime` è strettamente dipendente all'interprete Java e quindi dal sistema operativo. Ovviamente cercare di eseguire un programma come "calc" su Windows va bene, ma non su altri sistemi.

EJE sfrutta proprio la classe `Runtime` per lanciare la compilazione, l'esecuzione, la creazione di javadoc etc…Ma il primo prototipo di EJE è stato sviluppato nell'anno 1999.
Esistono tanti altri metodi nella classe Runtime. Per esempio il metodo `availableProcessors()`, restituisce il numero di processori disponibili. I metodi `freeMemory()`, `maxMemory()` e `totalMemory()` restituiranno informazioni al runtime sullo stato della memoria.
Lo studio degli altri metodi è lasciato al lettore se interessato.

12.2.4 La classe Class e Reflection

La classe `Class`, astrae il concetto di classe Java. Questo per esempio ci permetterà di creare oggetti Java dinamicamente all'interno di programmi Java. In particolare ci sono tre modi per avere un oggetto della classe `Class`, che astrae un certo tipo di classe: Utilizzare il metodo statico `forName(String name)` in questo modo:

```
try {
   Class stringa = Class.forName("java.lang.String");
} catch (ClassNotFoundException exc) {
        . . .
}
```

Ottenerla da un oggetto:

```
String a = "MiaStringa";
Class stringa = a.getClass();
```

Mediante un cosiddetto "class literal":

```
Class stringa = java.lang.String.class;
```

La caratteristica più interessante di questa classe, è che da la possibilità di utilizzare una tecnica che è conosciuta con il nome di reflection: l'introspezione delle classi. Infatti la classe `Class` mette a disposizioni dei metodi che si chiamano `getConstructor()`, `getMethods()`, `getFields()`, `getSuperClass()` etc… che restituiscono oggetti di tipo `Costructor`, `Field`, `Method` e così via. Queste classi astraggono i concetti di costruttore, variabile e metodo e si trovano all'interno del package `java.lang.reflect`. Ognuna di esse definisce metodi per ricavare informazioni specifiche. Per esempio la classe `Field`, dichiara il metodo `getModifiers()`…
Il seguente esempio permette di stampare i nomi di tutti i metodi della classe passata da riga di comando (se viene lanciata senza specificare parametri esplora la classe `Object`):

```
import java.util.*;
import java.lang.reflect.*;

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

```
        String className = "java.lang.Object";
        if (args.length > 0) {
            className = args[0];
        }
        Class objectClass= null;
        try {
            objectClass = Class.forName(className);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        Method[] methods = objectClass.getMethods();
        for (int i = 0; i < methods.length; i++) {
            String name = methods[i].getName();
            Class[] classParameters =
                methods[i].getParameterTypes();
            String stringClassParameters = "";
            for (int j = 0; j < classParameters.length;
                    ++j) {
                stringClassParameters +=
                    classParameters[j].toString();
            }
            String methodReturnType =
                methods[i].getReturnType().getName();
            String methodString = methodReturnType + " "
            + name + " (" + stringClassParameters + ")";
            System.out.println(methodString);
        }
    }
}
```

EJE fa uso di reflection quando mostra i membri della classe, dopo aver inserito l'operatore dot, dopo un reference (o una classe, o un array…). Se siete interessati è possibile scaricare il codice del progetto open source EJE, agli indirizzi: http://sourceforge.net/projects/eje/, http://eje.sourceforge.net o http://www.claudiodesio.com/eje.htm.

Tramite la classe `Class` è anche possibile istanziare oggetti di una certa classe, conoscendone solo il nome. Per esempio:

```
try {
    Class miaClasse = Class.forName("MiaClasse");
    Object ref = miaClasse.newInstance();
    . . .
} catch (ClassNotFoundException exc) {
    . . .
}
```

Spesso questa tecnica può diventare molto utile.

12.2.5 Le classi wrapper

Sono dette classi wrapper (in italiano "involucro") le classi che fanno da contenitore ad un tipo di dato primitivo, astraendo proprio il concetto di tipo. In Java esistono infatti le seguenti classi **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, **`Boolean`**, **`Character`**. Ognuna delle quali può contenere il relativo tipo primitivo. Queste sono utili ed indispensabili soprattutto nei casi in cui dobbiamo in qualche modo utilizzare un tipo di dato primitivo laddove è richiesto un oggetto. Per esempio, supponiamo di avere il seguente metodo che tramite un parametro polimorfo di tipo `Object` setta l'età di un oggetto di tipo `Persona`:

```
public class Persona {
    private int age;
    . . .
    public void setAge(Object age){
        // codice che riesce a settare l'attributo age
        //qualunque sia il formato del parametro
    }
    . . .
```

l'obiettivo di un tale metodo è quello di settare in qualche modo l'età, sia che questa venga passata sotto forma di data di nascita come oggetto `Date`, `Calendar` o `String`. Ora supponiamo che a seguito di un cambio di requisiti del programma, sia richiesto che questo metodo accetti anche numeri interi che rappresentino il numero di anni piuttosto che la data di nascita. Per come è definito il metodo `setAge()`, supponendo che `pippo` sia un oggetto della classe `Persona`, non è possibile scrivere:

```
pippo.setAge(30);
```

senza ottenere un errore in compilazione. Però è possibile scrivere:

```
Integer anni = new Integer(30);
pippo.setAge(anni);
```

Ovviamente bisogna poi accomodare il metodo `setAge()`, in modo tale che gestisca anche questo nuovo caso, con un codice simile al seguente:

```
public void setAge(Object age){
    // codice che riesce a settare la data qualsiasi
    // sia il formato
    . . .
    if (age instanceof Integer) {
       Integer integerAge = (Integer)age;
       this.age = integerAge.intValue();
    }
    . . .
}
```

Un altro tipico esempio in cui è necessario utilizzare le classi wrapper, è quando si utilizzano le Collections. Infatti, esse accettano come elementi solo oggetti e non tipi di dati primitivi. Quindi è illegale scrivere:

```
Vector v = new Vector();
v.add(5.3F);
```

mentre è del tutto legale scrivere:

```
Vector v = new Vector();
v.add(new Float(5.3F));
```

Inoltre per riottenere il tipo di dato primitivo originale, bisogna dapprima estrarre il tipo wrapper dalla `Collection` (sfruttando un casting), per poi estrarre a sua volta il tipo primitivo dal tipo wrapper, come è possibile notare nel seguente codice:

```
Float objectFloat = (Float)v.get(0);
float primitiveFloat = objectFloat.floatValue();
```

Tutto questo non è fortunatamente più necessario dalla versione 5 di Java!
Infatti, grazie alla nuova caratteristica del linguaggio detta "**Autoboxing e Auto-Unboxing**", è ora possibile anche aggiungere tipi primitivi direttamente alle collections! Infatti, la nuova caratteristica permette di fatto di equiparare i tipi primitivi con i relativi tipi wrapper. Quindi sarà possibile tranquillamente anche aggiungere direttamente un tipo primitivo ad una `Collection`, come nel seguente esempio:

```
Vector v = new Vector();
v.add(5.3F);
```

Infatti, il compilatore di Java, convertirà il codice precedente in modo tale che il valore primitivo `5.3F`, venga dapprima inscatolato automaticamente (da qui il termine "autoboxing") nel relativo tipo wrapper (in questo caso un `Float`). Ovvero, è come se il codice dell'esempio precedente venga trasformato dal compilatore nel seguente codice:

```
Vector v = new Vector();
v.add(new Float(5.3F));
```

Sarà anche possibile estrarre direttamente il tipo primitivo dalla `Collection`, senza nessun passaggio intermedio né casting, grazie all'auto-unboxing:

```
float primitiveFloat = v.get(0);
```

Dal momento che i tipi primitivi ed i relativi tipi wrapper sono di fatto equiparati, allora sarà anche per esempio possibile eseguire operazioni aritmetiche incrociate come nei seguenti esempi:

```
Integer i = new Integer(22);
int j = i++;
Integer k = (new Integer(10) + j);
int t = k + j + i;
```

Per conoscere tutti i dettagli dell'autoboxing e dell'auto-unboxing, il lettore è rimandato all'Unità Didattica 16.2.

12.2.6 La classe Math

Questa classe appartiene al package `java.lang` (da non confondere con il package `java.math`). Ha la caratteristica di avere solo membri statici: 2 costanti ("PI greco" e la "E" base dei logaritmi naturali), e 31 metodi.
I metodi rappresentano:

- Le funzioni matematiche valore assoluto (`abs()`), tangente (`tan()`), logaritmo (`log()`), potenza (`pow()`), massimo (`max()`), minimo (`min()`), seno (`sin()`), coseno (`cos()`), esponenziale (`exp()`), radice quadrata (`sqrt()`)
- arrotondamenti per eccesso (`ceil()`), per difetto (`floor()`), e classico (`round()`)
- generazione di numeri casuali (`random()`)

Questa classe non ha un costruttore pubblico ma privato, quindi non si può istanziare. Non è neanche possibile estendere questa classe per due ragioni:

1. Ha un costruttore privato (cfr. Modulo 8)
2. È dichiarata `final` (cfr. Modulo 9)

Un esempio di utilizzo della classe `Math` è il seguente:

```
System.out.println(Math.floor(5.6));
```

In questo caso sarà stampato il valore `5.0`. Infatti la "funzione" `floor()`, restituisce un numero `double` che arrotonda per difetto il valore del parametro in input, sino al più vicino numero intero.

Riepilogo

In questo modulo abbiamo introdotto le classi principali dei package `java.lang` e `java.util`.
La conoscenza approfondita del framework Collections, è fondamentale per gestire le performance dei nostri programmi. L'introduzione dei generics inoltre, ha aumentato notevolmente la potenza del framework.
Le classi wrapper rappresentano un importantissimo strumento quando si utilizzano le collections e non solo. Con l'Autoboxing però, è ora possibile utilizzare tipi wrapper e

tipi primitivi (cfr. Unità Didattica 16.2) quasi allo stesso modo.
Abbiamo anche visto come l'internazionalizzazione in Java sia semplice, potente e facilmente implementabile in programmi già scritti seguendo pochi passi.
Un altro argomento particolarmente semplice ed utile è la gestione della configurazione mediante file di properties. Meno semplice è utilizzare le espressioni regolari, non tanto per le classi da utilizzare ma per la sintassi stessa delle espressioni. Ma le regular expressions rappresentano uno strumento molto potente per l'analisi dei testi, e potremmo quasi parlare di standard.
La Reflection inoltre, rappresenta uno strumento essenziale per applicazioni come EJE, ma non solo. Ma la possibilità di creare oggetti a partire da stringhe (nome della classe), è un lusso che è facile permettersi in Java. Abbiamo anche accennato alla nuova "Java Compiler API", che probabilmente permetterà la nascita di nuovi tool di sviluppo in futuro.
Abbiamo anche introdotto le classi `Math`, `System`, `Runtime` e `StringTokenizer` ed accennato alla gestione delle date, delle valute e degli orari.

Esercizi modulo 12

Esercizio 12.a)
Framework Collections, Vero o Falso:

1. `Collection` , `Map`, `SortedMap`, `Set`, `List` e `SortedSet` sono interfacce e non possono essere istanziate
2. Un `Set` è una collezione ordinata di oggetti, una `List` non ammette elementi duplicati ed è ordinata
3. Le mappe non possono contenere chiavi duplicate, ed ogni chiave può essere associata ad un solo valore
4. Esistono diverse implementazioni astratte da personalizzare nel framework come `AbstractMap`
5. Un'`HashMap` è più performante rispetto ad un `Hashtable` perché non è sincronizzata
6. Un'`HashMap` è più performante rispetto ad un `TreeMap` ma quest'ultima, essendo un'implementazione di `SortedMap` gestisce l'ordinamento
7. `HashSet` è più performante rispetto a `TreeSet`, ma non gestisce l'ordinamento
8. `Iterator` ed `Enumeration` hanno lo stesso ruolo ma quest'ultima permette durante le iterazioni di rimuovere anche elementi
9. `ArrayList` ha prestazioni migliori rispetto a `Vector` perché non è sincronizzato, ma entrambi hanno meccanismi per ottimizzare le prestazioni
10. La classe `Collections` è un lista di `Collection`

Esercizio 12.b)
Package java.util e java.lang, Vero o Falso:

1. La classe `Properties` estende `Hashtable` ma permette di salvare su di un file le coppie chiave-valore rendendole persistenti
2. La classe `Locale` astrae il concetto di "zona"
3. La classe `ResourceBundle` rappresenta un file di properties che permette di gestire l'internazionalizzazione. Il rapporto tra nome del file e `Locale` specificato per individuare tale file, permetterà di gestire per esempio la configurazione della lingua delle nostre applicazioni
4. L'output del seguente codice:
```
StringTokenizer st = new StringTokenizer(
```

```
            "Il linguaggio object oriented Java", "t",
false);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà

Il linguaggio objec
t
orien
t
ed Java

5. Il seguente codice non è valido:
```
Pattern p = Pattern.compile("\bb");
Matcher m = p.matcher("blablabla…");
boolean b = m.find();
System.out.println(b);
```
6. La classe `Runtime` dipende strettamente dal sistema operativo su cui gira
7. La classe `Class` ci permette di leggere i membri di una classe (ma anche le superclassi ed altre informazioni) partendo semplicemente dal nome della classe grazie al metodo `forName()`
8. Tramite la classe `Class` è possibile istanziare oggetti di una certa classe, conoscendone solo il nome
9. È possibile dalla versione 1.4 di Java sommare un tipo primitivo e un oggetto della relativa classe wrapper come nel seguente esempio:
```
Integer a = new Integer(30);
int b = 1;
int c = a+b;
```
10. La classe `Math` non si può istanziare perché dichiarata `abstract`

Soluzioni esercizi modulo 12

Esercizio 12.a)
Framework Collections, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Vero**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Vero**
8. **Falso**
9. **Vero**
10. **Falso**

Esercizio 12.b)
Package java.util e java.lang, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Vero**
4. **Falso** tutte le "t" non dovrebbero esserci
5. **Falso** è valido ma stamperà `false`. Affinché stampi `true` l'espressione si deve modificare in "`\\bb`"
6. **Vero**
7. **Vero**
8. **Vero**
9. **Falso** dalla versione 1.5
10. **Falso** non si può istanziare perché ha un costruttore privato ed è dichiarata `final` per non poter essere estesa

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere l'utilità e saper utilizzare il framework Collection (unità 12.1)	☐	
Saper implementare programmi con l'internazionalizzazione (unità 12.1)	☐	
Saper implementare programmi configurabili mediante file di properties (unità 12.1)	☐	
Saper utilizzare la classe StringTokenizer per "splittare" stringhe (unità 12.1)	☐	
Saper utilizzare la Reflection per l'introspezione delle classi (unità 12.2)	☐	
Saper introdurre le classi System, Math e Runtime (unità 12.1)	☐	

Note:

13

Complessità: media

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Aver compreso il pattern Decorator (unità 13.1, 13.2).
2. Saper riconoscere nelle classi del package java.io i ruoli definiti nel pattern Decorator (unità 13.3).
3. Capire le fondamentali gerarchie del package java.io (unità 13.3).
4. Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 13.4).
5. Avere un'idea di base del networking in Java, dei concetti di socket e del metodo accept (unità 13.5).

13 Comunicare con Java: Input, Output e Networking

Gli argomenti di questo modulo sono considerati complessi da molti sviluppatori. Tuttavia, conoscendo la filosofia di base del networking e soprattutto dell'Input Output in Java, tali argomenti possono essere considerati alla portata di tutti.

13.1 Introduzione all'input-output

Spesso le applicazioni hanno bisogno di utilizzare informazioni lette da fonti esterne, o spedire informazioni a destinazioni esterne. Per **informazioni** intendiamo non solo stringhe, ma anche oggetti, immagini, suoni, etc…. Per **fonti** o **destinazioni** esterne all'applicazione invece intendiamo file, dischi, reti, memorie o altri programmi. In questo modulo vedremo come Java permette di gestire la lettura (**input**) da fonti esterne e la scrittura su destinazioni esterne (**output**). In particolare introdurremo il package `java.io`, croce e delizia dei programmatori Java. Il package in questione è molto vasto, ed anche abbastanza complesso. Conoscere ogni singola classe del package è un'impresa ardua e soprattutto inutile. Per poter gestire l'input - output in Java conviene piuttosto capirne la filosofia che ne è alla base, che è regolata dal design pattern noto come Decorator (cfr. Appendice H per la definizione di pattern). Non comprendere il pattern Decorator implicherà fare sempre fatica nel districarsi tra le classi di

`java.io`... al lettore quindi si raccomanda la massima concentrazione, visto che anche il pattern stesso è abbastanza complesso.

13.2 Pattern Decorator

È facile riconoscere nella gerarchia delle classi di `java.io`, il modello di classi definito dal pattern Decorator. Si tratta di un pattern GoF strutturale (cfr. Appendice H), decisamente complesso ma incredibilmente potente.
Nelle prossime righe viene proposto un esempio del pattern Decorator, per poter dare un'idea al lettore della sua utilità. È però possibile (ma non consigliabile) passare direttamente a leggere la descrizione del package se l'argomento pattern non interessa o lo si ritiene troppo complesso.

13.2.1 Descrizione del pattern

Il pattern **Decorator**, permette al programmatore di implementare tramite una particolare gerarchia di ruoli, una relazione tra classi che rappresenta un'alternativa dinamica alla statica ereditarietà. In pratica, sarà possibile aggiungere responsabilità addizionali agli oggetti al runtime, piuttosto che creare una sottoclasse per ogni nuova responsabilità. Sarà anche possibile ovviare a questo tipo di problema: supponiamo di voler aggiungere ad una certa classe `ClasseBase`, delle responsabilità (chiamiamole `r1`, `r2`, `r3`, e `r4`) di pari dignità, ovvero ognuna dovrebbe essere aggiunta indipendentemente alle altre.

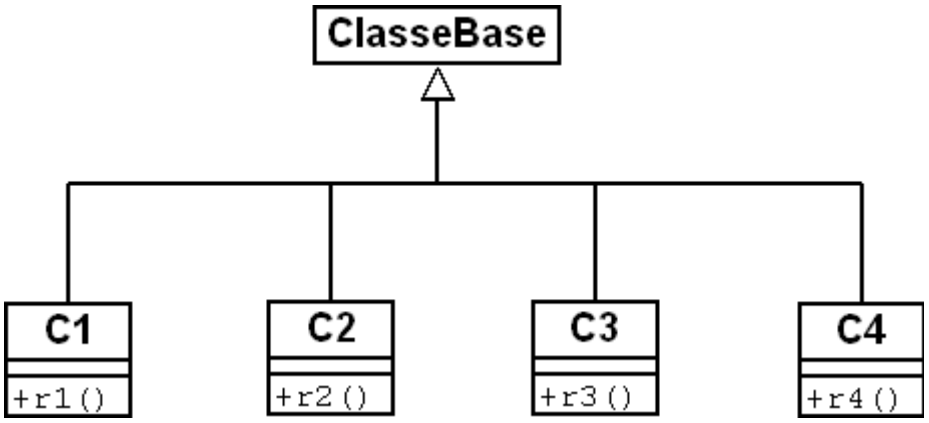


Figura 13.1 – "Ereditarietà"

Potrebbe però servire anche creare classi che hanno più di una di queste responsabilità per esempio `r1`, `r2` e `r4`.

In questo caso particolare l'ereditarietà da sola non riesce a soddisfare i nostri bisogni. Partendo dalla gerarchia in Figura 13.1, il lettore può provare a cercare una soluzione al nostro problema. Probabilmente ci sarà un numero di classi pari a 2 elevato al numero delle varie responsabilità, nel nostro esempio 2 elevato alla quarta potenza, ovvero 16. Inoltre, causa l'impossibilità di implementare l'ereditarietà multipla, non sarà possibile ottenere un risultato che sia accettabile dal punto di vista object oriented.
Consigliamo al lettore però di non impegnarsi troppo per trovare la soluzione a questo problema, perché ne esiste una geniale già a disposizione di tutti.
Nel pattern Decorator esistono i seguenti ruoli:

- `Component`: si tratta di un'interfaccia (o classe astratta), che definisce una o più operazioni astratte da implementare nelle sottoclassi.
- `ConcreteComponent`: questo ruolo è interpretato da una o più classi non astratte, che implementano `Component`.
- `Decorator`: si tratta di un'altra estensione di `Component` che potrebbe essere anche astratta. Questa deve semplicemente obbligare le sue sottoclassi (`ConcreteDecorator`), non solo ad implementare `Component`, ma anche a referenziare un `Component` con un'aggregazione (cfr. App. G per la definizione di aggregazione) (vedi Fig. 13.2). Questo ruolo può essere considerato opzionale.
- `ConcreteDecorator`: è l'implementazione di `Decorator`, che come abbiamo già asserito, dovrà implementare `Component` e le sue operazioni e mantenere un reference verso un `Component`.

Tutto ciò viene riassunto con il diagramma in Fig. 13.2:

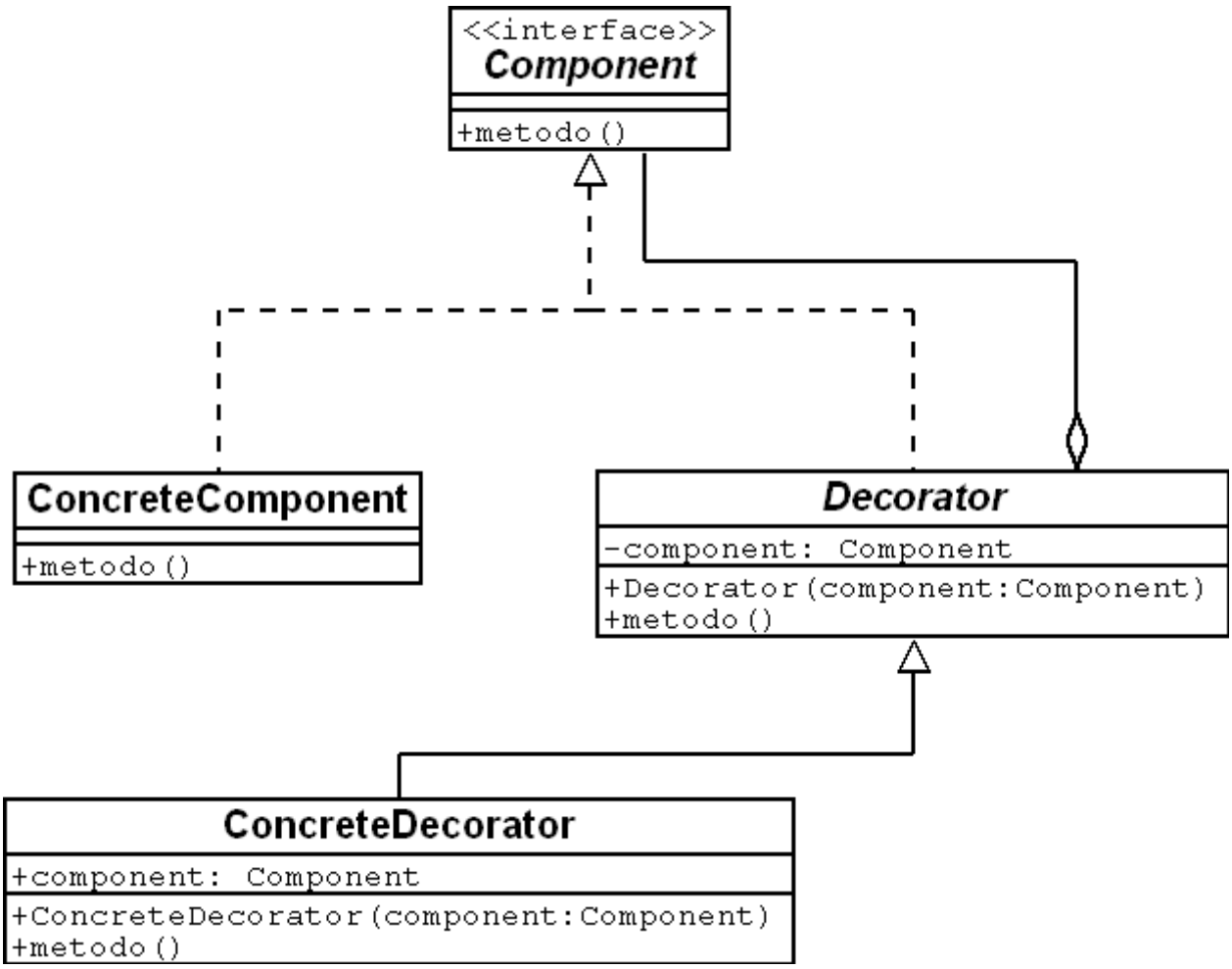


Figura 13.2 – "Modello del pattern Decorator"

Facciamo un esempio per comprendere come "funziona" questo pattern.
Come al solito è necessario calarsi in un contesto con un po' di fantasia e flessibilità. Supponiamo di voler realizzare un'applicazione grafica che permette di aggiungere effetti speciali (come un effetto 3D e un effetto trasparente) alle nostre immagini. Per realizzare ciò, occorre identificare la classe `Immagine` come `ConcreteComponent`, e la classe `Effetto3DDecorator` e `TrasparenteDecorator` come `ConcreteDecorator` come mostrato in Figura 13.3.

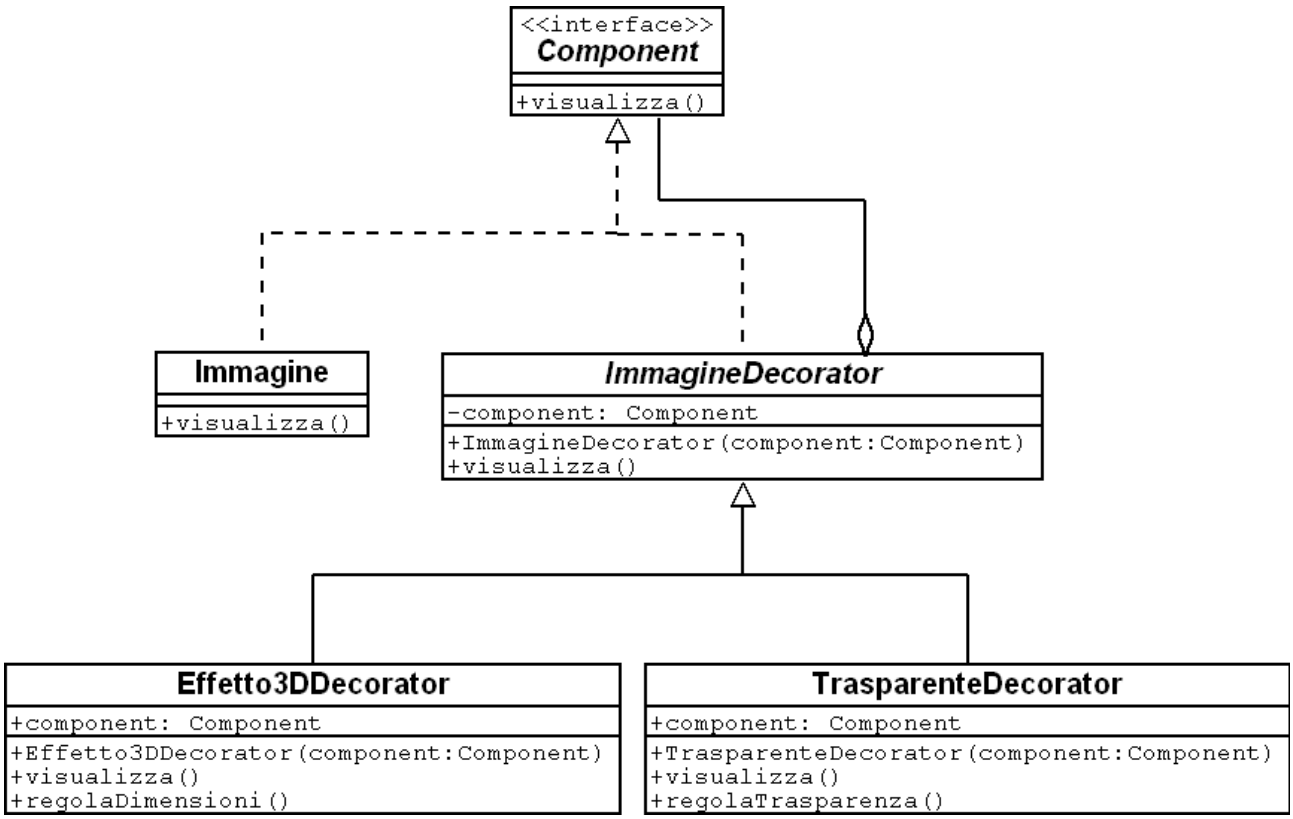


Figura 13.3 – "Esempio di implementazione di Decorator"

A questo punto analizziamo il seguente codice:

```
1 Immagine monnaLisa = new Immagine();
2 monnaLisa.visualizza();
3 Effetto3DDecorator monnaLisa3D = new
Effetto3DDecorator(monnaLisa);
4 monnaLisa3D.visualizza();
5 TrasparenteDecorator monnaLisa3DTrasparente = new
    TrasparenteDecorator(monnaLisa3D);
6 monnaLisa3DTrasparente.visualizza();
```

Alle righe 1 e 2 viene istanziata l'immagine `monnaLisa` e visualizzata. Alla riga 3 entra in scena il primo decorator che viene istanziato aggregandogli l'oggetto `monnaLisa` appena istanziato. Questo è il punto più complesso e ha bisogno di essere ben analizzato. Per prima cosa notiamo che ogni decorator, ha un unico costruttore che prende in input obbligatoriamente (concetto di aggregazione cfr. App. G) un

`Component`. Ovvero non può esistere un oggetto `Decorator` senza che aggreghi un `Component` (in questo caso `Immagine` ha ruolo di `ConcreteComponent`). In effetti un effetto speciale non può esistere senza un oggetto da decorare. È questo il punto chiave del pattern.
Alla riga 4 viene invocato il metodo `visualizza()` direttamente sull'oggetto `Effetto3DDecorator (monnaLisa3D)` che contiene l'immagine. Notiamo come l'oggetto `monnaLisa3D` (di tipo `Effetto3DDecorator`), se da un lato rappresenta solo una decorazione dell'immagine, in questo pattern diventa proprio l'oggetto immagine decorato. Ecco perché abbiamo preferito chiamarlo `monnaLisa3D` nonostante sia di tipo `Effetto3DDecorator`. Ovviamente, sarebbe più naturale che un oggetto di tipo `Immagine` (o una sua sottoclasse) rimanesse l'oggetto contenitore, ma bisogna abituarsi all'idea per ottenere i benefici di questo pattern.
Notiamo inoltre che il metodo `visualizza()` di un `Decorator`, sicuramente farà uso anche del metodo `visualizza()` dell'`Immagine`. Per esempio il metodo `visualizza()` della classe `Effetto3DDecorator` potrebbe essere codificato più o meno nel modo seguente:

```
public void visualizza() {
    //codice per generare l'effetto 3D . . .
    component.visualizza();
}
```

Ovviamente il reference `component` punta all'oggetto `monnaLisa` di tipo `Immagine`.
Alla riga 5 si ripete un'altra "decorazione" ma stavolta il decoratore (`monnaLisa3DTrasparente` di tipo `TrasparenteDecorator`) invece di decorare un oggetto di tipo immagine, decora un altro decoratore (`monnaLisa3D` di tipo `Effetto3DDecorator`) che ha già decorato un oggetto di tipo `Immagine` (`monnaLisa`). Ciò è possibile grazie al fatto che ogni `Decorator` deve decorare per forza un `Component`, e `Component`, è implementato anche dai `Decorator`.
Alla riga 6 viene invocato il metodo `visualizza()` direttamente sull'oggetto `TrasparenteDecorator` (`monnaLisa3DTrasparente`) che contiene il decoratore che contiene l'immagine.
Anche in questo caso il metodo `visualizza()` di `TrasparenteDecorator`, sicuramente farà uso anche del metodo `visualizza()` del suo `component` che aggrega. In questo caso però il `component` aggregato non è una semplice immagine

ma un oggetto di tipo `Effetto3DDecorator` (`monnaLisa3D`) che a sua volta aggregava l'oggetto `monnaLisa` di tipo `Immagine`.
Concludendo, il pattern è sicuramente complesso, e questo non gioca a suo favore, ma è incredibilmente potente. Basta pensare che, facendo riferimento all'esempio appena presentato ci sono almeno i seguenti vantaggi:

1. Il numero delle classi da creare è molto minore a qualsiasi soluzione basata sulla ereditarietà. Per esempio non deve esistere la classe `Immagine3DTrasparente`.
2. Se nascono nuovi effetti speciali, basterà aggiungerli come decorator senza che tutto il resto del codice ne risenta.
3. Al runtime si può creare qualsiasi combinazione basata sui decoratori senza sforzi eccessivi.

13.3 Descrizione del package

L'implementazione del pattern Decorator per la gestione dell'input-output in Java, è sicuramente stata la soluzione ideale. Infatti, come già asserito nell'introduzione, tale package deve mettere a disposizione dell'utente, classi per realizzare un qualsiasi tipo di lettura (input) e un qualsiasi tipo di scrittura output. Le fonti di lettura e le destinazioni di scrittura sono molte, e in futuro potrebbero nascerne di nuove. Il pattern Decorator permette quindi di realizzare qualsiasi tipo di comunicazione con fonti di destinazioni esterne, con un limitato numero di classi. Nonostante ciò, il numero di classi del package `java.io` rimane comunque alto. Fortunatamente però, non bisogna conoscerle tutte, la documentazione ufficiale serve proprio a questo, e conoscendo il pattern Decorator, potremo riconoscere i ruoli di ogni classe.
Partiamo dal concetto fondamentale che è alla base del discorso: lo **stream** (in italiano flusso).
Per prelevare informazioni da una fonte esterna (un file, una rete etc…), un programma deve aprire uno stream su di essa e leggere le informazioni in maniera sequenziale. La figura 13.4 mostra graficamente l'idea.

Figura 13.4 – "Rappresentazione grafica di un input"

Allo stesso modo, un programma può spedire ad una destinazione esterna aprendo uno stream su di essa, e scrivendo le informazioni sequenzialmente, come mostrato in figura 13.5.

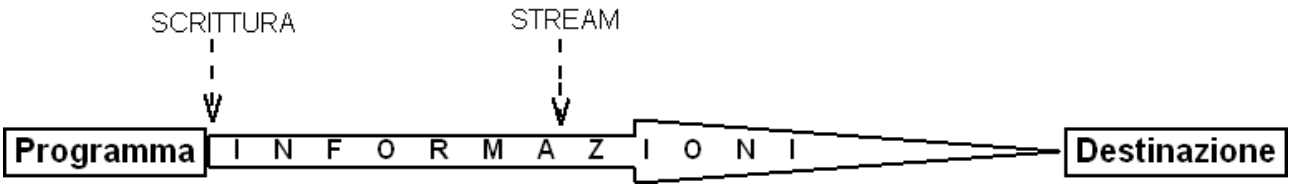


Figura 13.5 – "Rappresentazione grafica di un output"

In pratica aprire uno stream da una fonte o verso una destinazione, significa aprire verso questi punti terminali un canale di comunicazione, dove far passare le informazioni un "pezzo dopo l'altro". Con `java.io`, non importa di che tipo sono le informazioni, né con che fonti o destinazioni abbiamo a che fare. Gli algoritmi per scrivere o leggere infatti, sono sostanzialmente sempre gli stessi.
Per l'input bisogna:

1. aprire lo stream
2. leggere tutte le informazioni dallo stream fino a quando non terminano
3. chiudere lo stream

Per l'output bisogna:

1. aprire lo stream
2. scrivere tutte le informazioni tramite lo stream fino a quando non terminano
3. chiudere lo stream

Il package `java.io`, contiene una collezione di classi che supportano tali algoritmi per leggere e scrivere. Le classi di tipo stream sono divise in due gerarchie separate (anche se simili) in base al tipo di dato di informazione che devono trasportare (byte o caratteri).

13.1.1 I Character Stream

`Reader` e `Writer` sono le due superclassi astratte per i "character stream" ("flussi di caratteri") in `java.io`. Queste due classi hanno la caratteristica di obbligare le

sottoclassi a leggere e scrivere dividendo i dati in "pezzi" di 16 bit ognuno, quindi compatibili con il tipo `char` di Java. Le sottoclassi di `Reader` e `Writer` hanno i ruoli del Decorator pattern. Il ruolo di `Component` è interpretato proprio da `Reader` (e `Writer`). Le sottoclassi di `Reader` (e `Writer`) implementano stream speciali. Alcune di queste hanno il ruolo di `ConcreteComponent` (dovremmo dire `ConcreteReader` e `ConcreteWriter`) e da soli possono attaccarsi ad una fonte (o ad un destinazione) e leggere (o scrivere) subito, quantomeno con un algoritmo sequenziale implementato nei metodi `read()` (o `write()`) ereditati da `Reader` (o `Writer`). Questi stream vengono anche detti "Node Stream" ("flussi nodo").
Altre sottoclassi di `Reader` (e `Writer`) invece, interpretano il ruoli di `ConcreteDecorator`. Tali stream sono anche detti "Processing Stream", e, senza aggregazioni (cfr. App. G) a `ConcreteReader` (o `ConcreteWriter`) non si possono neanche istanziare. Il più delle volte, lo scopo dei decoratori di stream è quello di migliorare le prestazioni o facilitare la lettura (o la scrittura) delle informazioni negli stream, fornendo metodi adeguati allo scopo, ed evitando così cicli noiosi, e poco eleganti. Nelle figure 13.6 e 13.7, sono mostrate le gerarchie delle classi principali per quanto riguarda rispettivamente `Reader` e `Writer`, con i decoratori colorati in grigio.

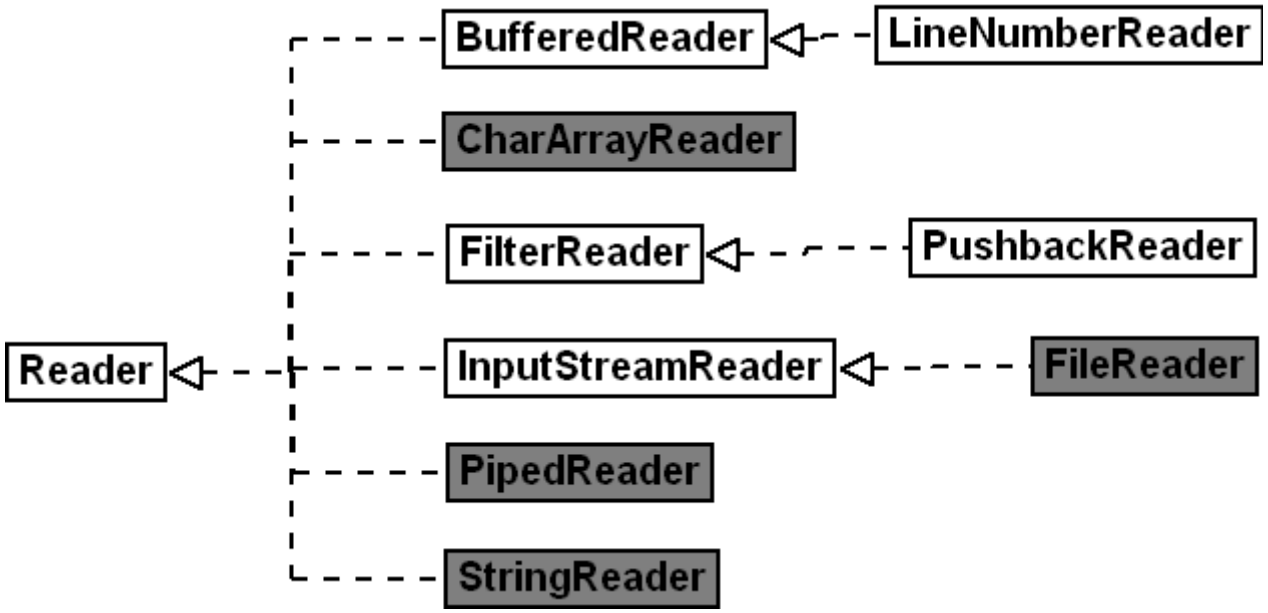


Figura 13.6 – "Gerarchia dei Reader"

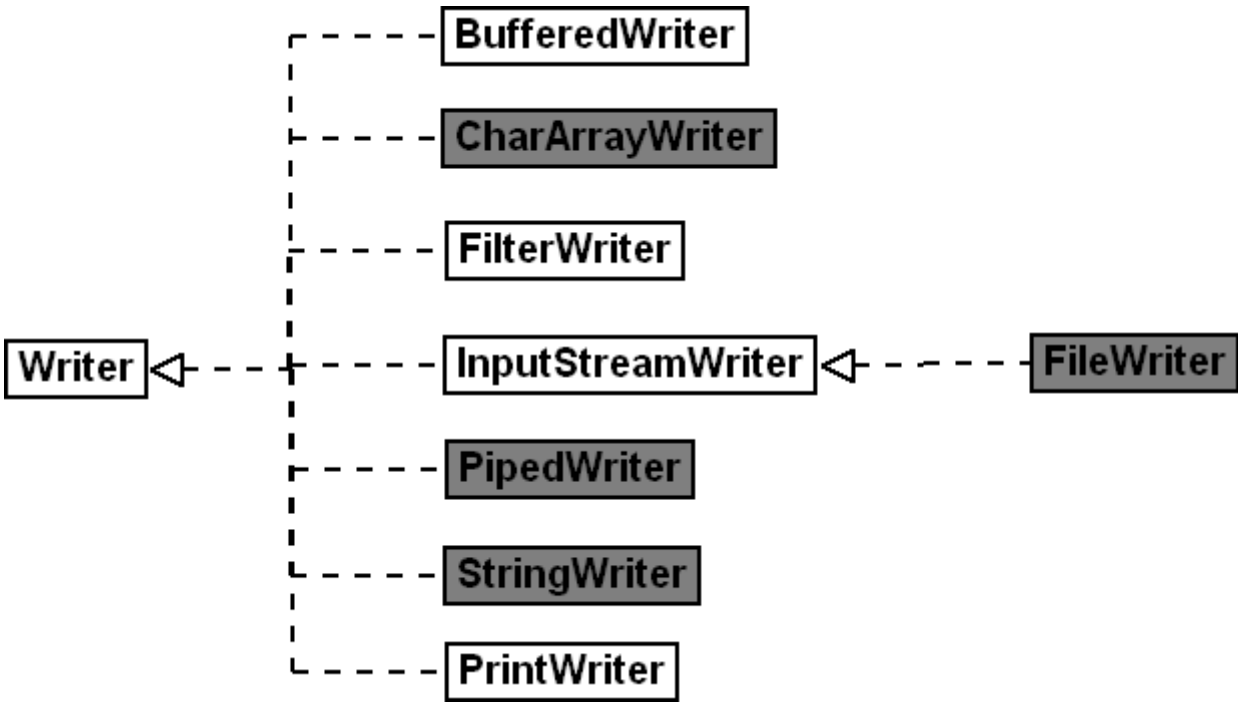


Figura 13.7 – "Gerarchia dei Writer"

I programmi dovrebbero utilizzare i character stream (con i reader e i writer) per le informazioni di tipo testuale. Infatti, questi permettono di leggere qualsiasi tipo di carattere Unicode.

13.3.2 I Byte Stream

Nel package `java.io`, esistono delle gerarchie di classi parallele ai reader e i writer che però sono destinati alla lettura e la scrittura di informazioni non testuali (per esempio file binari come immagini o suoni). Queste classi infatti leggono (e scrivono) dividendo i dati in "pezzi" di 8 bit ognuno. In pratica vale quanto detto per `Reader` e `Writer` anche per le classi `InputStream` e `OutputStream`. Le figure 13.8 e 13.9 mostrano le gerarchie delle classi principali per quanto riguarda rispettivamente `InputStream` e `OutputStream`, con i decoratori colorati in grigio.

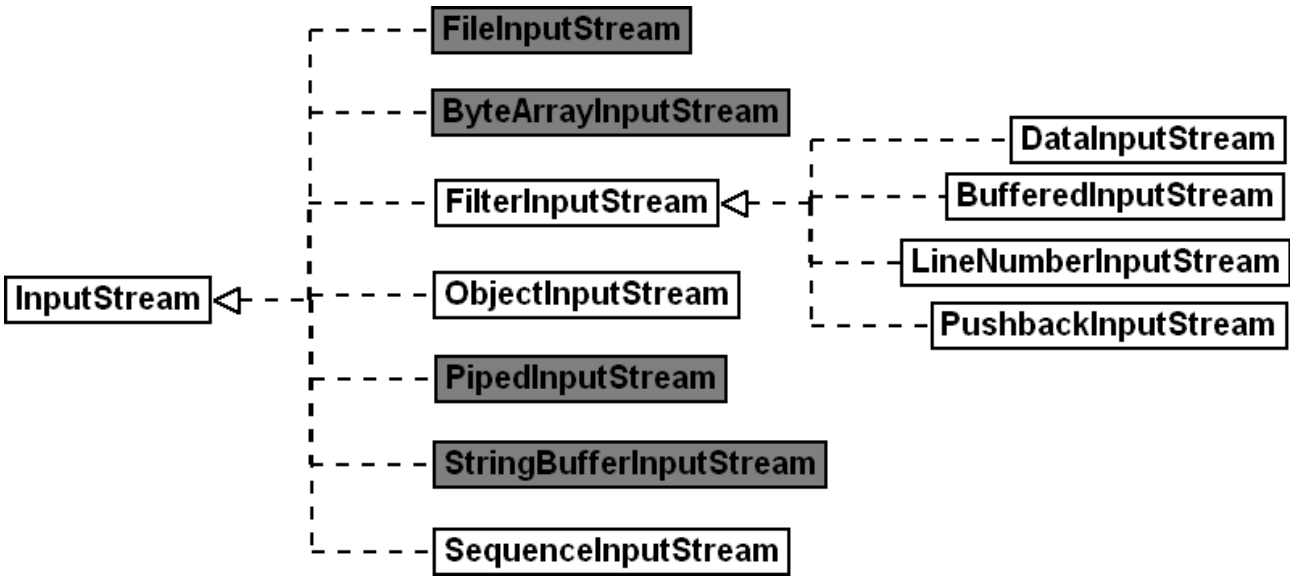


Figura 13.8 – "Gerarchia degli InputStream"

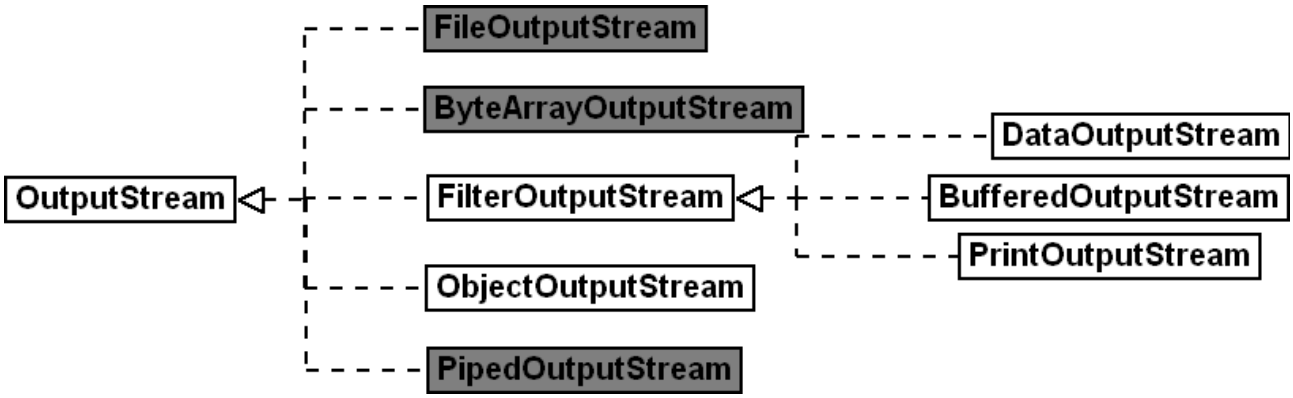


Figura 13.9 – "Gerarchia degli OuputStream"

Due di queste classi `ObjectInputStream` e `ObjectOutputStream`, sono utilizzate per la cosiddetta serializzazione di oggetti di cui vedremo un esempio nel prossimo paragrafo.

13.3.3 Le super-interfacce principali

`Reader` e `InputStream` definiscono praticamente gli stessi metodi ma per differenti tipi di dati. Per esempio, `Reader` contiene i seguenti metodi per leggere caratteri ed array di caratteri:

- `int read() throws IOException`
- `int read(char cbuf[]) throws IOException`
- `int read(char cbuf[], int offset, int length) throws IOException`

`InputStream` definisce gli stessi metodi ma per leggere byte ed array di byte:

- `int read()throws IOException`
- `int read(byte cbuf[])throws IOException`
- `int read(byte cbuf[], int offset, int length) throws IOException`

Inoltre, sia `Reader` che `InputStream` forniscono i seguenti metodi d'utilità:

- `int available() throws IOException` **:** ritorna il numero di byte che possono essere letti all'interno dello stream
- `void close() throws IOException` **:** rilascia le risorse si sistema associate allo stream (ma non provoca una deallocazione immediata). Nonostante il garbage collector può implicitamente chiudere uno stream quando non è più possibile referenziare l'oggetto, conviene comunque chiudere gli stream esplicitamente ogni volta che è possibile, per migliorare le prestazioni dell'applicazione
- `long skip(long nbytes) throws IOException` **:** prova a leggere e a scartare `nbytes`. Ritorna il numero di byte scartati.

Anche `Writer` e `OutputStream` sono da considerarsi classi parallele. `Writer` definisce i seguenti metodi per scrivere caratteri ed array di caratteri characters:

- `int write(int c)`
- `int write(char cbuf[])`
- `int write(char cbuf[], int offset, int length)`

`OutputStream` invece definisce gli stessi metodi ma per i byte:

- `int write(int c)`
- `int write(byte cbuf[])`

- `int write(byte cbuf[], int offset, int length)`

Tutti gli stream (reader, writer, input stream, e output stream) sono automaticamente aperti quando creati. Ogni stream si può chiudere esplicitamente chiamando il metodo `close()`. Il garbage collector può implicitamente chiudere uno stream quando non è più possibile referenziare l'oggetto.

Conviene comunque chiudere gli stream esplicitamente ogni volta che è possibile, per migliorare le prestazioni dell'applicazione. Una buona tecnica è quella di chiudere lo stream (ovviamente una volta utilizzato), nella clausola `finally` di un blocco `try-catch`.

13.4 Input ed output "classici"

In questa unità didattica verranno presentati alcuni esempi di classiche situazioni di input output.

13.4.1 Lettura di input da tastiera

Il seguente codice, definisce un programma che riesce a leggere ciò che l'utente scrive sulla tastiera, e dopo aver premuto invio, ristampa quanto letto. Per terminare il programma bisogna digitare "fine" e premere "invio".

Se si lancia quest'applicazione con EJE, per potere iniziare a digitare con la tastiera, bisognerà assicurarsi di posizionarsi sull'area di output dell'editor (che infatti si chiama `IOArea`).

```
import java.io.*;

public class KeyboardInput {
  public static void main (String args[]) {
    String stringa = null;
    InputStreamReader isr = new
      InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(isr);
    System.out.println("Digita qualcosa e premi " +
      "invio...\nPer terminare il programma " +
```

```
      "digitare\"fine\"");
    try {
      stringa = in.readLine();
      while ( stringa != null ) {
        if (stringa.equals("fine")) {
          System.out.println("Programma terminato");
          break;
        }
        System.out.println("Hai scritto: " + stringa);
        stringa = in.readLine();
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        in.close();
      }
      catch (IOException exc) {
        exc.printStackTrace();
      }
    }
  }
}
```

Un `InputStreamReader` viene creato ed agganciato all'oggetto `System.in`, che, come asserito nel modulo precedente, rappresenta l'input di default del nostro sistema (quasi sicuramente la tastiera).

> **La classe `InputStreamReader` è molto particolare. Permette di trasformare una fonte dati di tipo `byte`, in una fonte dati di tipo `char`.**

L'oggetto `isr`, da solo ci avrebbe già permesso di leggere con uno dei metodi `read()` il contenuto dallo stream proveniente dalla tastiera, ma per noi sarebbe stato molto più complesso realizzare il nostro obiettivo. Infatti avremmo dovuto concretizzare il seguente algoritmo:

1. leggere ogni byte con un ciclo

2. per ogni carattere letto con il metodo `read()`
3. controllare se il carattere coincide con "`\n`" (ovvero il tasto "invio")
4. se è verificato il passo 3 stampare i caratteri precedenti e svuotare lo stream. Se i caratteri precedenti formavano la parola "fine" terminare l'applicazione. Se la condizione 3 non è verificata allora leggere il prossimo carattere.

Notiamo come invece decorando con un `BufferedReader` l'oggetto `isr`, la situazione si sia notevolmente semplificata. Il `BufferedReader` infatti, mette a disposizione del programmatore il metodo `readLine()`, che restituisce il contenuto dello stream sotto forma di stringa fino al tasto "invio".
Quanta strada abbiamo dovuto fare con Java per leggere finalmente un carattere da tastiera! Se stessimo imparando il linguaggio C, già nel primo modulo avremmo introdotto la funzione `scanf()` per leggere e `printf()` per scrivere…quanti progressi con questi linguaggi moderni!
Ritornando seri, l'input-output di Java è effettivamente un argomento complesso, ma come constateremo presto, la complessità è costante per qualsiasi operazione di input-output. Per esempio il metodo `println()` viene utilizzato sia per stampare sulla prompt DOS, sia per stampare contenuto dinamico in una risposta HTTP…
A proposito di `printf()`, dalla versione 5 esiste anche in Java! Ritrova nella classe `PrintStream` (quella di `System.out`) e basa la sua implementazione sul metodo `format()` della classe `java.util.Formatter`. Per esempio è ora possibile scrivere il seguente codice in Java:

```
boolean b = true;
System.out.printf("Ecco un valore booleano: %b, ora si
formatta come in C!", b);
```

ottenendo il seguente output:

```
Ecco un valore booleano: true, ora si formatta come in C!
```

Questo argomento sarà trattato approfonditamente nel Modulo 18.

Inoltre la classe `Scanner` del package `java.util`, permette di semplificare la lettura di sorgenti di input, siano esse stringhe, tipi primitivi, file o altro. Per esempio, è ora possibile leggere da tastiera tramite `Scanner` con il seguente codice:

```
Scanner sc = new Scanner(System.in);
String testoDigitato = null;
while (sc.hasNext()) {
    testoDigitato += sc.next();
}
```

in realtà `Scanner` può utilizzare anche le espressioni regolari per realizzare complesse operazioni di riconoscimento di testo.

13.4.2 Gestione dei file

Nel package `java.io`, esiste la classe `File` che astrae il concetto di file generico.

Anche una directory è un file: un file che contiene altri file.

Seguono i dettagli dei metodi più interessanti di questa classe:

- `boolean exists()` restituisce `true` se l'oggetto file coincide con un file esistente sul file system
- `String getAbsolutePath()` ritorna il path assoluto del file
- `String getCanonicalPath()` **c**ome il metodo precedente ritorna il path assoluto ma senza utilizzare i simboli "`.`" e "`..`"
- `String getName()` ritorna il nome del file o della directory
- `String getParent()` ritorna il nome del file della directory che contiene il file
- `boolean isDirectory()` ritorna `true` se l'oggetto file coincide con una directory esistente sul file system
- `boolean isFile()` ritorna `true` se l'oggetto file coincide con un file esistente sul file system
- `String[] list()` ritorna una array di stringhe contenente i nomi dei file contenuti nella directory su cui viene chiamato il metodo. Se questo metodo viene invocato su un file che non è una directory restituisce `null`
- `boolean delete()` tenta di cancellare il file corrente
- `long length()` ritorna la lunghezza del file
- `boolean mkdir()` tenta la creazione di una directory il cui path è descritto dall'oggetto `File` corrente

- `boolean renameTo(File newName)` tenta di rinominare il file corrente. Tale metodo ritorna `true` e solo se ha successo
- `boolean canRead()` ritorna `true` se il file o la directory può essere letta dall'utente corrente (ha permesso in lettura)
- `boolean canWrite()` ritorna `true` se il file o la directory può essere può essere modificata
- `boolean createNewFile()` crea un nuovo file vuoto come descritto dall'oggetto corrente se tale file non esiste già. Ritorna `true` se e solo se tale file viene creato

I costruttori della classe File sono i seguenti:

- `File(String pathname)`
- `File(String dir, String subpath)`
- `File(File dir, String subpath)`

Esempi:

- ```
  File dir = new File("/usr", "local");
  File file = new File(dir, "Abc.java");
  ```
- ```
  File dir2 = new File("C:\\directory");
  File file2 = new File(dir2, "Abc.java");
  ```

Notare come nel primo caso abbiamo istanziato una directory e un file su un sistema Unix, e nel secondo invece abbiamo eseguito le stesse operazioni su un sistema Windows. È possibile utilizzare quindi il separatore per il path dei file per i vari sistemi operativi in maniera dipendente, ma è possibile anche utilizzare come separatore "/" anche su sistemi Windows. La migliore idea, è però quella di utilizzare la costante statica della classe File (dipendente dal sistema operativo): `File.pathSeparator`. Questa varrà su sistemi Windows "\" e su sistemi Unix "/". Per esempio:
`File file = new File(".." + File.pathSeparator + "Abc.java")`
Istanziare un file non significa però crearlo fisicamente sul file system, ovviamente bisogna utilizzare gli stream…
Il seguente codice permette di creare una copia di backup di un file specificato da riga di comando:

```
import java.io.*;

public class BackupFile {
  public static void main(String[] args) {
    FileInputStream fis = null;
    FileOuputStream fos = null;
    try {
      if (args.length == 0) {
        System.out.println("Specificare nome del file!");
        System.exit(0);
      }
      File inputFile = new File(args[0]);
      File outputFile = new File(args[0]+".backup");
      fis = new FileInputStream (inputFile);
      fos = new FileOuputStream (outputFile);
      int b = 0;
      while ((b = fis.read()) != -1) {
        fos.write(b);
      }
      System.out.println("Eseguito backup in " + args[0]
        + ".backup!");
    } catch (IOException exc) {
      exc.printStackTrace();
    } finally {
      try {
        fis.close();
        fos.close();
      } catch (IOException exc) {
        exc.printStackTrace();
      }
    }
  }
}
```

Il codice precedente ha bisogno che sia specificato come argomento da riga di commando il nome del file di cui fare la copia, altrimenti termina stampando un messaggio esplicativo. Il codice è abbastanza semplice. Vengono dapprima istanziati due oggetti `File`: `inputFile`, che rappresenta il file da copiare, ed `outputFile` che rappresenta il file copia. Subito dopo vengono istanziati i relativi canali di input e di

output verso questi due file, con la creazione dei due oggetti `fis` (di tipo `FileInputStream`) e `fos` (di tipo `FileOutputStream`). Infine, prima di stampare un messaggio di successo, l'applicazione cicla mediante un ciclo `while`, su tutti i byte che possono essere letti tramite l'oggetto `fis`, immagazzinandoli al volo all'interno della variabile `b`, per poi scriverli all'interno di `fos`. Notare come il ciclo termini solo quando il metodo `read()` del `FileInputStream`, restituisce `-1` (ovvero quando non ci sono più byte da leggere).

In questo caso non sono stati utilizzati decoratori. In questo modo abbiamo fatto una copia binaria del file, copiando i byte nella maniera più generica possibile. Questo fa sì che sia possibile specificare qualsiasi tipologia di file in input, per ottenerne una copia perfetta, sia esso un file di testo, sia esso un'immagine, un documento Word o qualsiasi altro tipo di file binario.

Notare che nel caso il file da copiare si trovi in una cartella diversa da quella in cui si sta lanciando l'applicazione, è necessario specificare il nome del file comprensivo del suo path relativo od assoluto. Un esempio di istruzione valida per lanciare la precedente applicazione mediante la specifica di un path relativo potrebbe essere il seguente:

```
java BackupFile ..\..\MioFileDacopiare
```

un esempio che sfrutta invece un path assoluto, potrebbe essere il successivo:

```
java BackupFile  C:\MiaCartella\MioFileDaCopiare
```

13.4.3 Serializzazione di oggetti

Con il termine **serializzazione di oggetti** intendiamo il processo di rendere persistente un oggetto Java. Rendere **persistente** un oggetto, significa far sopravvivere l'oggetto oltre lo shutdown del programma. Solitamente questo significa salvare lo **stato dell'oggetto**, ovvero le variabili d'istanza con i relativi valori, all'interno di un file (o all'interno di un database… ma questo sarà argomento del prossimo modulo).
In Java è possibile serializzare oggetti a patto che implementino l'interfaccia `Serializable` del package `java.io`. Tale interfaccia non contiene metodi ma serve solo a distinguere ciò che è serializzabile da ciò che non lo è. Esistono per esempio delle classi della libreria standard che, per come sono concepite, non possono essere

serializzate. Un esempio potrebbe essere la classe `Thread`. Un thread non ha uno stato, rappresenta un concetto dinamico, un "processo che esegue del codice su dei dati", e non può essere serializzato. La classe `Thread` infatti, non implementa l'interfaccia `Serializable`. Altri esempi di classi non serializzabili sono proprio tutte le classi di tipo stream.
Ora, tenendo presente che serializzare un oggetto significa salvare il suo stato interno, ovvero salvare il valore delle proprie variabili d'istanza, se vogliamo creare un classe da cui istanziare oggetti da serializzare, bisogna stare attenti alle variabili d'istanza. Se infatti una delle variabili d'istanza è di tipo `Thread` o di tipo `Writer`, se provassimo a serializzare l'oggetto, al runtime otterremmo una `java.io.NotSerializableException`. La stessa eccezione scatterebbe nel caso in cui la nostra classe non implementi `Serializable`.
Per ovviare a questo problema esiste il modificatore `transient` (cfr. Modulo 9). Esso può essere anteposto solo a variabili d'istanza, ed avvertirà al runtime la JVM che la variabile marcata `transient` non deve essere serializzata. È obbligatorio quindi marcare `transient` le variabili non serializzabili (per esempio di tipo `Thread`), ma potremmo anche desiderare non serializzare volutamente alcune variabili per ragioni di sicurezza.
Come esercizio consideriamo la seguente semplice classe da serializzare:

```
public class Persona implements java.io.Serializable{
    private String nome;
    private String cognome;
    private transient Thread t = new Thread();
    private transient String codiceSegreto;

    public Persona(String nome, String cognome, String
cs){
        this.setNome(nome);
        this.setCognome(cognome);
        this.setCodiceSegreto(cs);
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
```

```
    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public String getCognome() {
        return cognome;
    }

    public void setCodiceSegreto (String codiceSegreto) {
        this.codiceSegreto = codiceSegreto;
    }

    public String getCodiceSegreto () {
        return codiceSegreto;
    }

    public String toString() {
        return "Nome: " + getNome() + "\nCognome: " +
          getCognome() + "\nCodice Segreto: " +
          getCodiceSegreto();
    }
}
```

Abbiamo marcato `transient` una variabile `Thread` (eravamo obbligati), e la variabile `codiceSegreto` di tipo `String`. Ovviamente serializzare su di un file il codice segreto potrebbe essere una mossa poco previdente.
Per serializzare l'oggetto basta lanciare la seguente classe :

```
import java.io.*;

public class SerializeObject {
    public static void main (String args[]) {
        Persona p = new Persona ("Claudio",
          "De Sio Cesari", "xxx");
        try {
            FileOutputStream f = new FileOutputStream (
              new File("persona.ser"));
            ObjectOutputStream s =
              new ObjectOutputStream (f);
```

```
            s.writeObject (p);
            s.close ();
            System.out.println("Oggetto serializzato!");
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }
}
```

Il codice è molto semplice perché non molto diverso dagli altri esempi che finora abbiamo visto. La particolarità sta nella decorazione eseguita con l'oggetto di tipo `ObjectOutputStream`, e l'utilizzo del metodo `writeObject()` che ci fornisce quest'ultimo per realizzare il nostro scopo.
Per deserializzare l'oggetto basta lanciare la seguente classe:

```
import java.io.*;

public class DeSerializeObject {
    public static void main (String args[]) {
        Persona p = null;
        try {
            FileInputStream f =
            new FileInputStream (new
              File("persona.ser"));
            ObjectInputStream s =
            new ObjectInputStream (f);
            p = (Persona)s.readObject();
            s.close ();
            System.out.println("Oggetto " +
              "deserializzato!");
            System.out.println(p);
        } catch (Exception e) {
            e.printStackTrace ();
        }
    }
}
```

L'output della precedente classe sarà:

```
Oggetto deserializzato!
Nome: Claudio
Cognome: De Sio Cesari
Codice Segreto: null
```

Anche le variabili dichiarate `static` non saranno serializzate. Infatti, non sarebbe giusto cambiare il valore di una variabile statica, che è condivisa da tutte le istanze della classe, solo perché è possibile deserializzare una particolare istanza.

Esiste una strano vincolo per realizzare con successo una deserializzazione. Bisogna inserire un costruttore senza argomenti, nella prima superclasse della classe da serializzare, che non implementa `Serializable`, altrimenti andremo incontro ad una `java.io.InvalidClassException`, con un messaggio:

```
no valid constructor
```

quando verrà invocato il metodo `readObject()`.

13.5 Introduzione al networking

Il networking in Java è un argomento diverso dall'input-output ma dipendente da esso. Come abbiamo già asserito infatti, è possibile usare come fonte di lettura o come destinazione di scrittura una rete. Fortunatamente il discorso in questo caso è incredibilmente semplice! Infatti, per poter creare applicazioni che comunicano in rete occorrono solo poche righe di codice. Esiste però un package nuovo da importare: `java.net`.
Iniziamo con il dare però qualche informazione di base sui concetti che dovremo trattare. Ovviamente esistono interi libri che parlano di questi concetti, ma questa non è la sede adatta per poter approfondire troppo il discorso. Si consiglia al lettore interessato un approfondimento sulle migliaia di documenti disponibili in Internet.
Il networking in Java si basa essenzialmente sul concetto di **socket**. Un socket potrebbe essere definito come il punto terminale di comunicazione di rete. Ovviamente, per comunicare tramite rete, occorrono due socket che alternativamente si scambiano informazioni in input ed in output.
L'architettura di rete più utilizzata ai nostri giorni viene detta **client-server**. In questa

architettura esistono almeno due programmi (e quindi due socket): appunto un server e un client.
Un **Server** nella sua definizione più generale, è un'applicazione che una volta lanciata, si mette in attesa di connessioni da parte di altri programmi (detti appunto client), con cui comunica per fornire loro servizi. Un server una volta lanciato gira 24 ore su 24.
Un **Client** invece, è un'applicazione real-time, ovvero ha un ciclo di vita normale, si lancia e si termina senza problemi. Gode della facoltà di potersi connettere ad un server, per usufruire dei servizi messi a disposizione da quest'ultimo.

Nella maggior parte dei casi esiste un unico server per più client.

Giusto per dare un'idea della vastità dell'utilizzo di questa architettura, basti pensare che il mondo di Internet, è basato su client (per esempio browser come Internet Explorer) e server (per esempio server come Apache o IIS) e sulla suite di protocolli nota come **TCP-IP**. Potremmo parlare dell'argomento per centinaia di pagine ovviamente, ma in questa sede ci limiteremo a dare solo un'idea di cosa si tratta. Ignorando completamente la struttura ISO-OSI, possiamo dire che TCP-IP, contiene tutti i **protocolli di comunicazione** che normalmente utilizziamo in Internet. Quando navighiamo con il nostro browser per esempio, utilizziamo (nella stragrande maggior parte dei casi) il protocollo noto come **HTTP** (HyperText Transfer Protocol). L'HTTP (come tutti i protocolli) ha delle regole che definiscono non solo i tipi di informazioni che si possono scambiare client e server, ma anche come gestire le connessioni tra essi. Per esempio, un Client HTTP (il browser) può visitare un certo sito, e quindi richiedere una pagina HTML al server del sito in questione. Questo può restituire o meno la pagina richiesta. A questo punto la connessione tra client e server è già chiusa. Alla prossima richiesta del client si aprirà un'altra connessione con il server.
Quando invece mandiamo una e-mail, utilizziamo un altro protocollo che si chiama **POP3**, che ha delle regole completamente diverse dall'HTTP. Stesso discorso quando riceviamo una e-mail con il protocollo **SMTP**, quando scambiamo file con l'**FTP** o comunichiamo con **Telnet**. Ogni protocollo ha una sua funzione specifica ed una sua particolare politica di gestione delle connessioni.
Un client ed un server comunicano tramite un canale di comunicazione univoco, basato su essenzialmente tre concetti: il numero di porta, l'indirizzo IP, e il tipo di protocollo. Per quanto riguarda il **numero di porta**, abbiamo uno standard a 16 bit tra cui scegliere (quindi le porte vanno dalla 0 alla 65535). Le prime 1024 dovrebbero essere dedicate ai protocolli standard. Per esempio HTTP agisce solitamente sulla porta 80, FTP sulla 21 e così via. È possibile anche però utilizzare i vari protocolli su altre porte diverse da quelle di default. Il concetto di porta è solo virtuale, non fisico.

L'**indirizzo IP** è la chiave per raggiungere una certa macchina che si trova in rete. Ha una struttura costituita da 4 valori interi compresi a 8 bit (quindi compresi tra 0 e 255) separati da punti. Per esempio sono indirizzi IP validi 192.168.0.1, 255.255.255.0 e 127.0.0.1 (quest'ultimo è l'indirizzo che individua la macchina locale dove si lancia l'applicazione). Ogni macchina che si connette in rete ha un proprio indirizzo IP.
Quando viene lanciato un server, questo deve dichiarare su che numero di porta ascolterà le connessioni da parte dei client, aprendo un canale di input. Il server inoltre definirà il protocollo accettando e rispondendo ai client.
Passando subito alla pratica, creeremo con poche righe di codice un server e un client, che sfruttano un semplicissimo protocollo inventato da noi. La nostra coppia di programmi vuole essere solo un esempio iniziale, ma renderà bene l'idea di cosa significa comunicare in rete con Java.
Scriviamo un semplice server che si mette in ascolto sulla porta 9999, e restituisce ai client che si connettono una stringa di saluto per poi interrompere la comunicazione:

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

public class SimpleServer {
  public static void main(String args[]) {
    ServerSocket s = null;
    try {
      s = new ServerSocket(9999);
      System.out.println("Server avviato, in ascolto"
        + " sulla porta 9999");
    } catch (IOException e) {
      e.printStackTrace();
    }
    while (true) {
      try {
        Socket s1 = s.accept();
        OutputStream s1out = s1.getOutputStream();
        BufferedWriter bw = new BufferedWriter(new
          OutputStreamWriter(s1out));
        bw.write("Ciao client sono il server!");
           System.out.println("Messaggio spedito a " +
          s1.getInetAddress());
        bw.close();
        s1.close();
```

```
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

L'analisi del codice è davvero banale. Istanziando un `ServerSocket` con la porta `9999`, l'applicazione si dichiara server. Poi inizia un ciclo infinito che però si blocca sul metodo `accept()`, che mette in stato di "attesa di connessioni" l'applicazione. Una volta ricevuta una connessione da parte di un client, il metodo `accept()` viene eseguito e restituisce il socket che rappresenta il client con tutte le informazioni necessarie. A questo punto ci serve un canale di output verso il client che otteniamo semplicemente con il metodo `getOutputStream()` chiamato sull'oggetto socket `s1`. Poi per comodità decoriamo questo `OutputStream`, con un `BufferedWriter` che mettendoci a disposizione il metodo `writeLine()`, ci consente di spedire il messaggio al client in maniera banale. Subito dopo stampiamo l'indirizzo del client che riceverà il messaggio. Tutto qui!
Scrivere un client che utilizza il server appena descritto è ancora più semplice! Il seguente client se non specificato da riga di comando suppone che il server sia stato lanciato sulla stessa macchina. Poi dopo essersi connesso scrive la frase che riceve dal server. Segue il codice:

```
import java.net.*;
import java.io.*;
public class SimpleClient {
  public static void main(String args[]) {
    try {
      String host = "127.0.0.1";
      if (args.length != 0) {
         host = args[0];
      }
      Socket s = new Socket(host, 9999);
      BufferedReader br = new BufferedReader(new
        InputStreamReader(s.getInputStream()));
      System.out.println(br.readLine());
      br.close();
      s.close();
```

```
    } catch (ConnectException connExc) {
      System.err.println("Non riesco a connettermi "
        + "al server");
    } catch (IOException e) {
      System.err.println("Problemi...");
    }
  }
}
```

Questa applicazione quando istanzia l'oggetto socket si dichiara client del server che si trova all'indirizzo `host` e ascolta sulla porta `9999`. Il socket istanziato, quindi rappresenta il server a cui ci si vuole connettere. Poi decoriamo con un `BufferedReader` l'`InputStream` che ricava dal socket mediante il metodo `getInputStream()`. Infine stampa ciò che viene letto dallo stream mediante il metodo `readLine()`.
Come volevasi dimostrare, scrivere una semplice coppia di client-server è particolarmente facile in Java. Per i programmatori che hanno avuto a che fare con linguaggi come il C++, il package `java.net` potrebbe avere un suono dolce...
Certamente lo sviluppo di server più complessi, richiederebbe molto più impegno, dovendo utilizzare anche il multi-threading per la gestione parallela di più client e magari protocolli ben più complessi. Ma ci piace sottolineare che quello che abbiamo appena visto, è il modo più complesso di affrontare il networking in Java. Dalla versione 1.4 in poi infatti, esiste un nuovo package chiamato `java.nio` (ovvero "New Input Output"), e i suoi sottopackage, che semplificano enormemente il lavoro degli sviluppatori per creare server e client multi-threaded più complessi ed efficienti. I concetti di channel, buffers e charset permettono di creare con poco sforzo applicazioni molto complesse (cfr. documentazione). Addirittura dalla versione 5 in poi esistono nuove classi (per esempio la classe `ThreadPoolExecutor`) che con poche righe di codice permettono di creare pool di thread, per creare applicazioni server con performance notevoli.
Se poi vogliamo entrare nel mondo delle tecnologie Java, il lettore può provare a dare uno sguardo ad RMI (Remote Method Invocation) (cfr. guida del JDK). Sarà possibile invocare metodi di oggetti che si trovano in rete su altre macchine, senza scrivere veri e propri server e client…praticamente gli oggetti remoti vengono trattati come se fossero in locale…

Riepilogo

In questo modulo abbiamo essenzialmente parlato della comunicazione delle nostre applicazioni con l'esterno. Abbiamo visto come un package complesso come `java.io`, sia governato dai rapporti tra classi definiti dal pattern Decorator. Abbiamo quindi cercato di dare un'idea dell'utilità di tale pattern e lo abbiamo riconosciuto all'interno del package. Inoltre sono stati forniti degli esempi per le problematiche di input-output più comuni come l'accesso ai file, la lettura da tastiera e la serializzazione di oggetti. Infine il nostro discorso si è concluso con una descrizione sommaria di un argomento strettamente legato all'input-output: il networking. Abbiamo apprezzato la semplicità del codice necessario a soddisfare le esigenze dei nostri programmi di comunicare con altri programmi in rete, presentando due semplici esempi di client e server. Tutto ciò è stato preceduto da un velocissima introduzione ai concetti base del networking. Per concludere abbiamo accennato alla possibilità di utilizzare librerie più avanzate che il lettore può cercare di approfondire.

Esercizi modulo 13

Esercizio 13.a)
Input - Output, Vero o Falso:

1. Il pattern Decorator permette di implementare una sorta di ereditarietà dinamica. Questo significa che invece di creare tante classi quanti sono i concetti da astrarre, al runtime sarà possibile concretizzare uno di questi concetti direttamente con un oggetto
2. I reader e i writer, permettono di leggere e scrivere caratteri. Per tale ragione sono detti Character Strema
3. All'interno del package `java.io`, l'interfaccia `Reader` ha il ruolo di `ConcreteComponent`
4. All'interno del package `java.io`, l'interfaccia `InputStream`, ha il ruolo di `ConcreteDecorator`
5. Un `BufferedWriter` è un `ConcreteDecorator`
6. Gli stream che possono realizzare una comunicazione direttamente con una fonte o una destinazione, vengono detti "node stream"
7. I node stream di tipo `OutputStream` possono utilizzare il metodo
   `int write(byte cbuf[])`
   per scrivere su una destinazione
8. Il seguente oggetto `in`:
   ```
   BufferedReader in = new BufferedReader(
     new InputStreamReader(System.in));
   ```
   permette di usufruire di un metodo `readLine()` che leggerà frasi scritte con la tastiera delimitate dalla battitura del tasto "invio"
9. Il seguente codice:
   `File outputFile = new File("pippo.txt");`
   crea un file di chiamato "pippo.txt" nella cartella corrente
10. Non è possibile decorare un `FileReader`

Esercizio 13.b)
Serializzazione e networking, Vero o Falso:

1. Lo stato di un oggetto è definito dal valore delle sue variabili d'istanza (ovviamente in un certo momento)
2. L'interfaccia `Serializable` non ha metodi

3. `transient` è un modificatore applicabile a variabili e classi. Una variabile `transient` non viene serializzata con le altre variabili, una classe `transient` non è serializzabile
4. `transient` è un modificatore applicabile a metodi e variabili. Una variabile `transient` non viene serializzata con le altre variabili, un metodo `transient` non è serializzabile
5. Se si prova a serializzare un oggetto che ha tra le sue variabili d'istanza una variabile di tipo `Reader` dichiarata `transient`, otteremo un `NotSerializableException` al runtime
6. In una comunicazione di rete devono esistere almeno due socket
7. Un client per connettersi ad un server deve conoscere almeno il suo indirizzo IP e la porta su cui si è posto in ascolto
8. Un server si può mettere in ascolto anche sulla porta 80, la porta di default dell'HTTP, senza per forza utilizzare quel protocollo. È infatti possibile anche che si comunichi con il protocollo HTTP, su di una porta diversa dalla 80
9. Il metodo `accept()`, blocca il server in uno statto di "attesa di connessioni". Quando un client si connette, il metodo `accept()` viene eseguito per raccogliere tutte le informazioni del client in un oggetto di tipo `Socket`
10. Un `ServerSocket` non ha bisogno di dichiarare l'indirizzo IP, ma deve solo dichiarare la porta su cui si metterà in ascolto

Soluzioni esercizi modulo 13

Esercizio 13.a)
Input - Output, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso**
4. **Falso**
5. **Vero**
6. **Vero**
7. **Vero**
8. **Vero**
9. **Falso**
10. **Falso**

Esercizio 13.b)
Serializzazione e networking, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso**
4. **Falso**
5. **Falso**
6. **Vero**
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Aver compreso il pattern Decorator (unità 13.1, 13.2)	☐	
Saper riconoscere nelle classi del package java.io, i ruoli definiti nel pattern Decorator (unità 13.3)	☐	
Capire le fondamentali gerarchie del package java.io (unità 13.3)	☐	
Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 13.4)	☐	
Avere un'idea di base del networking in Java, dei concetti di socket e del metodo accept (unità 13.5)	☐	

Note:

14

Complessità: media

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper spiegare la struttura dell'interfaccia JDBC (unità 14.1, 14.2).
2. Saper scrivere codice che si connette a qualsiasi tipo di database (unità 14.2, 14.3).
3. Saper scrivere codice che aggiorna, interroga e gestisce i risultati qualsiasi sia il database in uso (unità 14.2, 14.3).
4. Avere confidenza con le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità 13.4).
5. Saper gestire i concetti della libreria JAXP, per la gestione dei documenti XML (unità 14.4).
6. Saper risolvere i problemi di utilizzo delle interfacce DOM e SAX per l'analisi dei documenti XML (unità 14.4).
7. Saper trasformare con XSLT i documenti XML (unità 14.4).

14 Java e la gestione dei dati: supporto a SQL e XML

Una delle difficoltà con cui si scontrano tutti i giorni gli sviluppatori Java, è quella di dover confrontarsi anche con aree dell'informatica. Infatti, l'apertura di Java verso le altre tecnologie è totale, e quindi bisogna spesso imparare altri linguaggi. Oggigiorno per esempio, è praticamente impossibile fare a meno dell'SQL e dell'XML. L'SQL (Structured Query Language), è il linguaggio standard per l'interrogazione (e non solo) dei database relazionali. È un linguaggio "leader" da tanti anni, che non ha ancora trovato un competitore (e chissà se lo troverà mai).
L'XML, è invece un linguaggio giovane, diventato in breve tempo "l'esperanto dell'informatica". Tramite esso, sistemi diversi riescono a dialogare, e questo è molto importante nell'era digitale. Oramai, è difficile trovare un'applicazione professionale che non faccia uso in qualche modo di XML. Non ci resta quindi che affrontare lo studio di questo modulo, con la consapevolezza che tratta argomenti estremamente importanti.

14.1 Introduzione a JDBC

JDBC viene spesso inteso come l'acronimo di “Java DataBase Connectivity”. Si tratta dello strato di astrazione software che permette alle applicazioni Java di connettersi a database. La potenza, la semplicità, la stabilità e le prestazioni delle interfacce JDBC, in questi anni hanno portato uno standard affermato come ODBC a mettere a serio rischio la propria supremazia. Rispetto ad ODBC, JDBC permette ad un’applicazione di accedere a diversi database senza dover essere modificata in nessun modo! Ciò implica che ad un'applicazione Java, di per sé indipendente dalla piattaforma, può essere aggiunta anche l'indipendenza dal database engine.
Caliamoci in uno scenario: supponiamo che una società crei un'applicazione Java che utilizza un RDBMS come DB2, lo storico prodotto della IBM. La stessa applicazione gira su diverse piattaforme come server Solaris e client Windows e Linux. Ad un certo punto, per strategie aziendali, i responsabili decidono di sostituire DB2, con un altro RDBMS questa volta di natura open source: MySQL. A questo punto, l'unico sforzo da fare, è far migrare i dati da DB2 a MySQL, ma l'applicazione Java, continuerà a funzionare come prima...
Questo vantaggio è molto importante. Basti pensare alle banche o agli enti statali, che decine di anni fa si affidavano completamente al trittico Cobol-CICS-DB2 offerto da IBM. Adesso, con l’enorme mole di dati accumulati negli anni, hanno difficoltà a migrare verso nuove piattaforme. In futuro con Java e JDBC, le migrazioni saranno molto meno costose…

14.2 Le basi di JDBC

Come già asserito, si tratta di uno strato di astrazione software tra un'applicazione Java ed un database. La sua struttura a due livelli, permette di accedere a database engine differenti, a patto che questi supportino l'ANSI SQL 2 standard.

La stragrande maggioranza dei database engine in circolazione supporta come linguaggio di interrogazione un soprainsieme dell'ANSI SQL 2. Ciò significa che esistono comandi che funzionano specificamente solo sui RDBMS su cui sono stati definiti (comandi proprietari) e che non sono parte dell'SQL standard. Questi comandi, semplificano l'interazione tra l'utente e il database, sostituendosi a

comandi SQL standard più complessi. Ciò implica che è sempre possibile sostituire ad un comando proprietario del RDBMS utilizzato con un comando SQL standard, anche se l'implementazione potrebbe essere più complessa. Un'applicazione Java-JDBC, che vuole mantenere una completa indipendenza dal database engine dovrebbe utilizzare solo comandi SQL standard, oppure prevedere appositi controlli laddove si vuole necessariamente adoperare un comando proprietario.

I due fondamentali componenti di JDBC sono:

1. Un'implementazione del vendor del RDBMS (o di terze parti) conforme alle specifiche delle API `java.sql`.
2. Un'implementazione da parte dello sviluppatore dell'applicazione.

14.2.1 Implementazione del vendor (Driver JDBC)

Il vendor deve fornire l'implementazione di una serie di interfacce definite dal package `java.sql`, ovvero `Driver`, `Connection`, `Statement`, `PreparedStatement`, `CallableStatement`, `ResultSet`, `DatabaseMetaData`, `ResultSetMetaData`. Ciò significa che saranno fornite alcune classi magari impacchettate in un unico file archivio JAR, che implementano i metodi delle interfacce appena citate. Solitamente tali classi appartengono a package specifici, ed i loro nomi sono spesso del tipo:

```
nomeDBNomeInterfacciaImplementata
```

(per esempio: `DB2Driver`, `DB2Connection`...). Inoltre il vendor dovrebbe fornire allo sviluppatore anche una minima documentazione. Attualmente tutti i più importanti database engine, supportano driver JDBC.

Esistono quattro tipologie diverse di driver JDBC caratterizzati da differenti potenzialità e strutture. Per una lista aggiornata dei driver disponibili è possibile consultare l'indirizzo http://java.sun.com/products/jdbc/jdbc.drivers.html.

14.2.2 Implementazione dello sviluppatore (Applicazione JDBC)

Lo sviluppatore ha un compito piuttosto semplice: implementare del codice che sfrutta l'implementazione del vendor, seguendo pochi semplici passi.
Un'applicazione JDBC deve:

1. Caricare un driver
2. Aprire una connessione con il database
3. Creare un oggetto `Statement` per interrogare il database
4. Interagire con il database
5. Gestire i risultati ottenuti

Viene presentato di seguito una semplice applicazione che interroga un database. Viene sfruttato come driver l'implementazione della Sun del jdbc-odbc bridge, presente nella libreria standard di Java (JDK1.1 in poi).

```
0 import java.sql.*;
1
2 public class JDBCApp {
3     public static void main (String args[]) {
4       try {
5 // Carichiamo un driver di tipo 1 (bridge jdbc-odbc)
6         String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
7         Class.forName(driver);
8 // Creiamo la stringa di connessione
9         String url = "jdbc:odbc:myDataSource";
10 // Otteniamo una connessione con username e password
11        Connection con =
12        DriverManager.getConnection (url, "myUserName",
            "myPassword");
13 // Creiamo un oggetto Statement per interrogare il db
14        Statement cmd = con.createStatement ();
15 // Eseguiamo una query e immagazziniamone i risultati
16 // in un oggetto ResultSet
17        String qry = "SELECT * FROM myTable";
18        ResultSet res = cmd.executeQuery(qry);
19 // Stampiamone i risultati riga per riga
20        while (res.next()) {
21        System.out.println(res.getString("columnName1")
            );
22        System.out.println(res.getString("columnName2")
            );
23        }
24        res.close();
25        cmd.close();
```

```
26          con.close();
27      } catch (SQLException e) {
28          e.printStackTrace();
29      } catch (ClassNotFoundException e) {
30          e.printStackTrace();
31        }
32      }
33}
```

14.2.3 Analisi dell'esempio JDBCApp

La nostra applicazione è costituita da un'unica classe contenente il metodo `main()`, non perché sia la soluzione migliore, bensì per evidenziare la sequenzialità delle azioni da eseguire.
Alla riga 0, viene importato l'intero package `java.sql`. In questo modo è possibile utilizzare i reference relativi alle interfacce definito in esso. Questi, sfruttando il polimorfismo, saranno utilizzati per puntare ad oggetti istanziati dalle classi che implementano tali interfacce, ovvero, le classi che sono fornite dal vendor. In questo modo l'applicazione non utilizzerà mai il nome di una classe fornita dal vendor, rendendo in questo modo l'applicazione indipendente dal database utilizzato. Infatti, gli unici riferimenti espliciti all'implementazione del vendor risiedono all'interno di stringhe, che sono ovviamente facilmente parametrizzabili in svariati modi (come vedremo presto).
Alla riga 7, utilizziamo il metodo statico `forName()` della classe `Class` (cfr. Modulo 12) per caricare in memoria l'implementazione del driver JDBC, il cui nome completo viene specificato nella stringa argomento di tale metodo. A questo punto il driver è caricato in memoria e si auto-registra con il `DriverManager` grazie ad un inizializzatore statico, anche se questo processo è assolutamente trasparente allo sviluppatore.

> **Il lettore può anche verificare quanto appena affermato scaricando il codice sorgente di un driver jdbc open source (per una lista aggiornata dei driver JDBC consultare l'indirizzo http://java.sun.com/products/jdbc/jdbc.drivers.html)**

Tra la riga 9 e la riga 12 viene aperta una connessione al database mediante la definizione di una stringa `url`, che deve essere disponibile nella documentazione fornita dal vendor (che però ha sempre una struttura del tipo `jdbc:subProtocol:subName` dove `jdbc` è una stringa fissa, `subProtocol` è

un identificativo del driver, e `subName` è un identificativo del database), e la chiamata al metodo statico `getConnection()` sulla classe `DriverManager`.
Alla riga 14 viene creato un oggetto `Statement` che servirà da involucro per trasportare le eventuali interrogazioni o aggiornamenti al database.
Tra la riga 17 e la riga 18 viene formattata una query SQL in una stringa chiamata `qry`, eseguita sul database, ed immagazzinati i risultati all'interno di un oggetto `ResultSet`. Quest'ultimo corrisponde ad una tabella formata da righe e colonne dove è possibile estrarre risultati facendo scorrere un puntatore tra le varie righe mediante il metodo `next()`.
Infatti tra la riga 20 e 23, un ciclo `while` chiama ripetutamente il metodo `next()`, il quale restituirà `false` se non esiste una riga successiva a cui accedere. Quindi, fin quando ci sono righe da esplorare vengono stampati a video i risultati presenti alle colonne di nome `columnName1` e `columnName2`.
Tra la riga 24 e la riga 26 vengono chiusi il `ResultSet res`, lo `Statement cmd`, e la `Connection con`.
Tra la riga 27 e la riga 28 viene gestita l'eccezione `SQLException`, lanciabile da qualsiasi problema relativo a JDBC, come una connessione non possibile o una query SQL errata.

Tra la riga 29 e la riga 32 invece, viene gestita la `ClassNotFoundException` (lanciabile nel caso fallisca il caricamento del driver mediante il metodo `forName()`)

I driver di tipo jdbc-odbc bridge (in italiano "ponte jdbc-odbc") sono detti di "tipo 1" e sono i meno evoluti. Richiedono che sia installato anche ODBC sulla nostra macchina (se avete installato Access sulla vostra macchina avete installato anche ODBC) e che sia configurata la fonte dati. In questo modulo useremo tale driver solo perché esiste nella libreria standard. Per poter lanciare quest'applicazione, dopo aver creato un semplice database (per esempio con Access), bisogna configurare la fonte data (data source), affinché punti al database appena creato. Il lettore può consultare la documentazione del database per il processo di configurazione della fonte dati (di solito è un processo molto semplice).
Il lettore può comunque procurarsi un altro driver che non sia di tipo 1 e che quindi non richieda tale configurazione (che però è piuttosto semplice). I driver di tipo 2 hanno la caratteristica di essere scritti in Java e in C/C++. Questo significa che devono essere installati per poter

funzionare. I driver di tipo 3 e 4, sono i più evoluti e sono scritti interamente in Java. Questo implica che non hanno bisogno di installazioni, ma solo di essere disponibili all'applicazione.

Ovviamente inoltre, bisogna apportare le seguenti modifiche al codice:
Riga 6: è possibile assegnare alla stringa `driver` il nome di un altro driver disponibile (opzionale)
Riga 9: è possibile assegnare alla stringa `url` il nome di un'altra stringa di connessione (dipende dal driver JDBC del database utilizzato e si legge dalla documentazione del driver). Nel nostro caso, disponendo di una fonte dati (data source) ODBC installata, basta sostituire il nome nella stringa il nome `myDataSource` con quello della fonte dati.
Riga 12: è possibile sostituire le stringhe `myUserName` e `myPassword` rispettivamente con la username e la password per accedere alla fonte dei dati. Se non esistono username e password per la fonte dati in questione, basterà utilizzare il metodo `DriverManager.getConnection(url)`.
Riga 17: sostituire nella stringa `myTable` con il nome di una tabella valida
Righe 21 e 22: sostituire le stringhe `columnName1` e `columnName2` con nomi di colonne valide per la tabella in questione.

14.3 Altre caratteristiche di JDBC

Esistono tante altre caratteristiche di JDBC, che è bene conoscere.

14.3.1 Indipendenza dal database

Avevamo asserito che la caratteristica più importante di un programma JDBC, è che è possibile cambiare il database da interrogare senza cambiare il codice dell'applicazione. Nell'esempio precedente però questa affermazione non trova riscontro. Infatti, se deve cambiare database, deve cambiare anche il nome del driver. Inoltre potrebbero cambiare anche la stringa di connessione (che solitamente contiene anche l'indirizzo IP della macchina dove gira il database), la username e la password. Come il lettore può notare però, queste 4 variabili non sono altro che stringhe, e una stringa è facilmente configurabile dall'esterno. Segue il codice dell'applicazione precedente, rivisto in modo tale da sfruttare un file di properties (cfr. Modulo 12) per leggere le variabili "incriminate":

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

public class JDBCApp {
    public static void main (String args[]) {
        try {
             Properties p = new Properties();
             p.load(new
               FileInputStream("config.properties"));
             String driver = p.getProperty("jdbcDriver");
             Class.forName(driver);
             String url = p.getProperty("jdbcUrl");
             Connection con =
             DriverManager.getConnection (url,
             p.getProperty("jdbcUsername"),
             p.getProperty("jdbcPassword"));
             Statement cmd = con.createStatement ();
             String qry = "SELECT * FROM myTable";
             ResultSet res = cmd.executeQuery(qry);
             while (res.next()) {

        System.out.println(res.getString("columnName1"));

        System.out.println(res.getString("columnName2"));
             }
             res.close();
             cmd.close();
             con.close();
        } catch (SQLException e) {
             e.printStackTrace();
        } catch (ClassNotFoundException e) {
             e.printStackTrace();
        }
    }
}
```

14.3.2 Altre operazioni JDBC (CRUD)

Ovviamente, con JDBC è possibile eseguire qualsiasi tipo di comando CRUD (Create Retrieve, Update, Delete) verso il database, non solo interrogazioni. Per esempio se volessimo inserire un nuovo record in una tabella potremmo scrivere:

```
String insertStatement = "INSERT INTO MyTable …";
int ris = cmd.executeUpdate(insertStatement);
```

Noi italiani, siamo soliti chiamare "query", un qualsiasi comando inoltrato al database. In realtà in inglese il termine "query" (che si può tradurre come "interrogazione"), viene utilizzato solamente per i comandi di tipo `SELECT …` Tutti i comandi che in qualche modo aggiornano il database vengono detti "update". Ecco perché con JDBC, è necessario invocare il metodo `executeUpdate()` per le operazioni di `INSERT`, `UPDATE` e `DELETE`, e `executeQuery()` per le operazioni di `SELECT`.

Ovviamente un aggiornamento del database non restituisce un `ResultSet`, ma solo un numero intero che specifica il numero dei record aggiornati.

14.3.3 Statement parametrizzati

Esiste una sotto-interfaccia di `Statement` chiamata `PreparedStatement`. Questa permette di parametrizzare gli statement, ed è molto utile laddove esiste un pezzo di codice che utilizza statement uguali che differiscono solo per i parametri. Segue un esempio:

```
PreparedStatement stmt = conn.prepareStatement("UPDATE
Tabella3 SET m = ? WHERE x = ?");
stmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
    stmt.setInt(2, i);
    int j = stmt.executeUpdate();
    System.out.println(j +" righe aggiornate quando i=" +
i);
}
```

Un `PreparedStatement` si ottiene mediante la chiamata al metodo `prepareStatement()` specificando anche la query che viene parametrizzata con dei punti interrogativi. I metodi `setString()` (ma ovviamente esistono anche i metodi

`setInt()`, `setDate()` e così via), vengono usati per settare i parametri. Si deve specificare come primo argomento un numero intero che individua la posizione del punto interrogativo all'interno del `PreparedStatement`, e come secondo argomento il valore che deve essere settato.
Ovviamente, il metodo `executeUpdate()` (o eventualmente il metodo `executeQuery()`), in questo caso non ha bisogno di specificare query.

1.1.1. 14.3.4 Stored procedure

JDBC offre anche il supporto alle stored procedure, mediante la sotto-intefaccia `CallableStatement` di `PreparedStatement`. Segue un esempio:

```
String spettacolo = "JCS";
CallableStatement query = msqlConn.prepareCall(
  "{call return_biglietti[?, ?, ?]}");
try {
    query.setString(1, spettacolo);
    query.registerOutParameter(2,
       java.sql.Types.INTEGER);
    query.registerOutParameter(3,
       java.sql.Types.INTEGER);
    query.execute();
    int bigliettiSala = query.getInt(2);
    int bigliettiPlatea = query.getInt(3);
} catch (SQLException SQLEx){
     System.out.println("Query fallita");
     SQLEx.printStackTrace();
}
```

In pratica nel caso delle stored procedure, i parametri potrebbero anche essere di output. In tal caso vengono registrati con il metodo `registerOutParameter()`, specificando la posizione nella query e il tipo SQL.

14.3.5 Mappature Java – SQL

Esistono delle tabelle da tener presente per sapere come mappare i tipi Java con i corrispettivi tipi SQL. La seguente mappa i tipi Java con i tipi SQL:

Tipo SQL Type	Tipo Java
`CHAR`	`String`

VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.TimeStamp

La prossima tabella invece serve per avere sempre ben presente cosa ritornano I metodi `getXXX()` di `ResultSet`:

Metodo	Tipo Java ritornato
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	boolean
getByte	byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getTime	java.sql.Time

`getTimestamp`	`java.sql.Timestamp`

Infine è utile tener presente anche la seguente tabella che mostra che tipi SQL sono associatia i metodi `setXXX()` di `Statement`:

Method	SQL Types
`setASCIIStream`	`LONGVARCHAR` prodotto da un ASCII stream
`setBigDecimal`	`NUMERIC`
`setBinaryStream`	`LONGVARBINARY`
`setBoolean`	`BIT`
`setByte`	`TINYINT`
`setBytes`	`VARBINARY` o `LONGVARBINARY` (dipende dai limiti relativi del `VARBINARY`)
`setDate`	`DATE`
`setDouble`	`DOUBLE`
`setFloat`	`FLOAT`
`setInt`	`INTEGER`
`setLong`	`BIGINT`
`setNull`	`NULL`
`setObject`	L'oggetto passato è convertito al tipo SQL corrispondente prima di essere mandato
`setShort`	`SMALLINT`
`setString`	`VARCHAR` o `LONGVARCHAR` (dipende dalla dimensione relativa ai limiti del driver sul `VARCHAR`)
`setTime`	`TIME`
`setTimestamp`	`TIMESTAMP`

14.3.6 Transazioni

JDBC ovviamente supporta anche le transazioni. Per usufruirne bisogna prima disabilitare l'auto commit nel seguente modo:

```
connection.setAutoCommit (false);
```

in seguito è poi possibile utilizzare i metodi `commit()` e `rollback()` sull'oggetto `connection`. Per esempio

```
try {
    . . .
    cmd.executeUpdate(INSERT_STATEMENT);
    . . .
    cmd.executeUpdate(UPDATE_STATEMENT);
    . . .
    cmd.executeUpdate(DELETE_STATEMENT);
    conn.commit();
}
catch (SQLException sqle) {
    sqle.printStackTrace();
    try {
        conn.rollback();
    }
    catch (SQLException ex) {
        throw new MyException("Commit fallito " +
          "- Rollback fallito!", ex);
    }
    throw new MyException("Commit fallito " +
        "- Effettuato rollback", sqle);
}
finally {
    // chiusura connessione…
}
```

14.3.7 Evoluzione di JDBC

Da quando JDBC è diventata un punto cardine della tecnologia Java, ha subito continui miglioramenti. Bisogna tener presente che anche il package `javax.sql` (definito da Sun come "JDBC Optional Package API"), fa parte dell'interfaccia JDBC, sin dalla versione 1.4 di Java. Ecco i nomi che hanno contraddistinto le versioni caratterizzanti lìevoluzione di JDBC:

- JDBC 3.0
- JDBC 2.1 core API
- JDBC 2.0 Optional Package API

- JDBC 1.2 API
- JDBC 1.0 API

Notare che JDBC 2.1 core API insieme a JDBC 2.0 Optional Package API insieme di solito vengono definite come JDBC 2.0 API.
Sostanzialmente sino ad ora abbiamo parlato della versione 1.0. È importante conoscere le versioni di JDBC perché così possiamo conoscere le caratteristiche supportate da un certo driver, solamente riferendoci al supporto della versione dichiarata. In particolare, le classi, le interfacce, i metodi, i campi, i costruttori e le eccezioni, dei package `java.sql` e `javax.sql`, sono documentate con un tag javadoc "since" che specifica la versione. In inglese "since" significa "da" nel senso "esiste dalla versione". Possiamo sfruttare tale tag per capire a quale versione di JDBC l'elemento documentato appartiene, tenendo presenti la seguente tabella:

Tag	Versione JDBC	Versione JDK
Since 1.4	JDBC 3.0	JDK 1.4
Since 1.2	JDBC 2.0	JDK 1.2
Since 1.1 oppure nessun tag	JDBC 2.0	JDK 1.1

Molte caratteristiche sono opzionali e non è detto che un driver le debba supportare. Per non avere brutte sorprese, è bene quindi consultare preventivamente la documentazione del driver da utilizzare.

14.3.8 JDBC 2.0

Oramai praticamente tutti i vendor hanno creato driver che supportano JDBC 2.0. Si tratta di un'estensione migliorata di JDBC che permette tra l'altro di scorrere il `ResultSet` anche al contrario, o di ottenere una connection mediante un oggetto di tipo `DataSource` (package `javax.sql`) in maniera molto performante mediante una "connection pool".

Per "connection pool" intendiamo quella tecnica di ottimizzazione delle performance che permette di ottenere delle istanze già pronte per l'uso (non si devono istanziare) di oggetti di tipo connessione. In particolare questa tecnica è fondamentale in ambienti enterprise dove bisogna servire contemporaneamente in maniera multi-threaded, diversi client che chiedono connessioni.

In particolare i `DataSource` sono lo standard da utilizzare per le applicazioni lato server in ambienti J2EE, e giusto per avere un'idea di come si utilizzano, riportiamo il seguente frammento di codice:

```
InitialContext context = new InitialContext();
DataSource ds =
(DataSource)context.lookup("jdbc/myDataSource");
Connection connection = ds.getConnection();
```

In questo caso abbiamo sfruttato la tecnologia JNDI per ottenere un istanza dell'oggetto `DataSource` per sfruttare un eventuale connection pool "offertaci" direttamente da un server J2EE.
Dopo avere ottenuto un oggetto `Connection`, i nostri passi per interagire con il database non cambiano.

14.3.9 JDBC 3.0

Ovviamente, oggi però non è assolutamente raro utilizzare driver JDBC 3.0. Tra le novità introdotte da JDBC 3.0, ricordiamo la capacità di creare oggetti di tipo `ResultSet` aggiornabili. Questo significa che abbiamo la possibilità ottenere dei risultati e di poterli modificare al volo in modalità "connessa". Per esempio, con le seguenti istruzioni:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM
TABLE2");
```

il `ResultSet` sarà scrollabile senza che vengano mostrati eventuali cambiamenti ai dati nel db (con cui il `ResulSet` rimane connesso), e contemporaneamente sarà possibile modificarne i dati al volo. Per esempio con le seguenti righe di codice:

```
rs.absolute(3);
rs.updateString("NAME", "Claudio");
rs.updateRow();
```

andiamo ad aggiornare nella terza riga del `ResultSet`, con il valore "Claudio" la colonna "NAME".

Notare come quando un `ResultSet` è scrollabile, è possibile non solo navigare in indietro e in avanti, ma anche spostarsi ad un determinata riga con il metodo `absolute()`.

Oppure con il seguente frammento di codice:

```
rs.moveToInsertRow();
rs.updateString(1, "Claudio");
rs.updateString("COGNOME", "De Sio Cesari");
rs.updateBoolean(3, true);
rs.updateInt(4, 32);
rs.insertRow();
rs.moveToCurrentRow();
```

abbiamo inserito una nuova riga nel `ResultSet`.
È interessante anche studiare la sotto-interfaccia di `ResultSet` `RowSet` (package `javax.sql`). In particolare, `RowSet` è a sua volta estesa da `CachedRowSet` (package `javax.sql.rowset`), che è un tipo di oggetto che lavora in maniera disconnessa dal db, ma che può anche sincronizzarsi con un'istruzione esplicita. Per esempio con le seguenti istruzioni:

```
rs.updateString(2, "Claudio");
rs.updateInt(4, 300);
rs.updateRow();
rs.acceptChanges();
```

il `CachedRowSet` si sincronizza con il db.

L'utilizzo di un `RowSet`, non è così intuitivo come per `ResultSet`, e si rimanda il lettore interessato ad eventuali approfondimenti sulla documentazione ufficiale e il relativo tutorial contenuto in essa.

14.4 Supporto a XML: JAXP

XML (acronimo di eXstensible Markup Language) è un linguaggio che oramai è in qualche modo parte di qualsiasi tecnologia moderna. Trattasi del naturale complemento

a Java per quanto riguarda il trasporto dei dati. Ovvero, come Java si distingue per l'indipendenza dalla piattaforma, XML può gestire il formato dei dati di un'applicazione, qualsiasi sia il linguaggio di programmazione utilizzato. Gestire i dati in semplici strutture XML, significa infatti gestirli tramite semplici file testuali (o flussi di testo), che sono indipendenti dal linguaggio di programmazione o dalla piattaforma che li utilizza. Per informazioni su XML è possibile consultare migliaia di documenti su Internet. La forza di questo linguaggio, è essenzialmente dovuta alla sua semplicità.
Java offre supporto a tutti i linguaggi o tecnologie basate su XML, da XSL a XPATH, da XSL-FO ai Web Services, dalla validazione con XML-schema a quella con il DTD, dall'esplorazione SAX a quella DOM e così via.
In questa sede però, dovendo parlare di dati, ci concentreremo essenzialmente al supporto di Java all'utilizzo di XML come tecnologia di trasporto informazioni. Infine introdurremo anche il supporto alle trasformazioni XSLT.
Le librerie che soddisfano ai nostri bisogni sono note come **JAXP** (Java API for XML Processing), e i package che ci interesseranno di più sono `org.w3c.dom`, `org.xml.sax`, `javax.xml.parsers`, `javax.xml.xpath` e `javax.xml.transform`, con i rispettivi sotto-package. I primi due package definiscono essenzialmente le interfacce delle due principali interfacce per il parsing (l'analisi) dei documenti XML.
La libreria **DOM**, acronimo di Document Object Model, è basata sull'esplorazione del documento XML in maniera sequenziale partendo dal primo tag e scendendo nelle varie ramificazioni (si parla di "albero DOM)". È implementata essenzialmente nelle interfacce del package `org.w3c.dom` e i suoi sotto-package.
La libreria **SAX**, acronimo di Simple API for XML invece, supporta l'analisi di un documento XML basata su eventi. È implementata essenzialmente dalle interfacce del package `org.xml.sax` e i suoi sotto-package.
Il package `javax.xml.parsers` invece, oltre ad un'eccezione (`ParserConfigurationException`) e un errore (`FactoryConfigurationError`), definisce solo 4 classi (`DocumentBuilder`, `DocumentBuilderFactory`, `SAXParser` e `SAXParserFactory`) che rappresentano essenzialmente delle factory per i principali concetti di XML. La situazione è simile a quella di JDBC, dove dal `DriverManager` ricavavamo una `Connection`, dalla `Connection` uno `Statement` e così via. Con JAXP otterremo da un `DocumentBuilderFactory` un `DocumentBuilder`, da un `DocumentBuilder` un oggetto `Document` e dal `Document` vari altri concetti fondamentali di XML come nodi e liste di nodi…

Il package `javax.xml.xpath`, ci permetterà di rimpiazzare spesso righe di codice DOM, grazie al linguaggio XPATH. Sicuramente si tratta del modo più veloce per recuperare i nodi di un documento XML.
Il package `javax.xml.transform` infine, offre le classi e le interfacce che supportano le trasformazioni XSLT.
Visto che la teoria è abbastanza vasta e la libreria non è delle più semplici, questa unità didattica sarà basata su semplici esempi pratici per risolvere i problemi più comuni. Inoltre la libreria è in continua evoluzione e quindi si raccomanda una seria consultazione della documentazione ufficiale.

14.4.1 Creare un documento DOM a partire da un file XML

Segue un esempio:

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
  . . .
  try {
    DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
    factory.setValidating(false);
    Document doc = factory.newDocumentBuilder().parse(new
    File("nomeFile.xml"));
    . . .
  } catch (Exception e) {
  . . .
  }
```

In pratica, bisogna utilizzare una factory per istanziare un `DocumentBuilder` per poter fare il parse del file XML. Il metodo `setValidating()` è stato utilizzato per comandare all'oggetto factory di non validare il documento verso l'eventuale DTD associato.

Se volessimo creare un documento vuoto, basterebbe sostituire la chiamata al metodo `parse()` con la chiamata al metodo `newDocument()`.

14.4.2 Recuperare la lista dei nodi da un documento DOM

Segue un esempio:

```
NodeList list = doc.getElementsByTagName("*");
for (int i=0; i<list.getLength(); i++) {
    Element element = (Element)list.item(i);
}
```

Con il precedente codice è possibile "toccare" solo i nodi principali, non i sottonodi. Se vogliamo invece accedere a tutti i nodi, anche quelli innestati in profondità allora non ci resta altro che creare una funzione ricorsiva come la seguente:

```
public void findNode(Node node, int level) {
    NodeList list = node.getChildNodes();
    for (int i=0; i<list.getLength(); i++) {
        Node childNode = list.item(i);
        findNode (childNode, level+1);
    }
}
```

Questo metodo va invocato con la seguente istruzione:

```
findNode(doc,0);
```

Purtroppo la libreria è un po' complessa, anche se dalla versione 5 di Java ci sono state delle evoluzioni…Tale complessità è dovuta all'evoluzione delle specifiche DOM (http://www.w3c.org per informazioni), che al momento del rilascio di questa libreria è al livello 3.

Per le specifiche DOM, ogni nodo è equivalente ad un altro. Questo significa che anche i commenti e il testo di un nodo (viene detto `TextNode`), sono a loro volta nodi.

14.4.3 Recuperare particolari nodi

Per recuperare l'elemento root di un documento XML ci sono due modi. Il primo metodo consiste nello scorrere l'albero DOM del documento (come nel precedente esempio), e fermarsi al primo elemento che sia di tipo `Element`. Questo controllo è

obbligatorio, altrimenti, a seconda del documento XML, si potrebbe recuperare un commento o la dichiarazione del DocumentType. Segue il codice per recuperare l'elemento root:

```
Element root = null;
NodeList list = doc.getChildNodes();
for (int i=0; i<list.getLength(); i++) {
    if (list.item(i) instanceof Element) {
        root = (Element)list.item(i);
        break;
    }
}
```

Il secondo metodo per recuperare l'elemento root di un documento è estremamente più semplice, ed equivalente al precedente:

```
Element root = doc.getDocumentElement();
```

Ovviamente se la necessità è quella di recuperare solo l'elemento root, il secondo metodo è consigliabile. Il primo metodo è preferibile per esempio solo se si vuole accedere anche ad altri nodi (ma il codice si deve un po' modificare).

L'interfaccia `Element` implementa `Node`.

Ottenuto un determinato nodo, l'interfaccia `Node` ci offre diversi metodi per esplorare altri nodi relativi al nodo in questione.
Per ottenere il nodo "padre" relativo al nodo in questione esiste il metodo `getParentNode()`:

```
Node parent = node.getParentNode();
```

Per ottenere la lista dei nodi "figlio" relativa al nodo in questione, esiste il metodo `getChildNodes()` (come già visto negli esempi precedenti):

```
NodeList children = node.getChildNodes();
```

Ma è anche possibile ottenere solo il primo o solo l'ultimo dei nodi "figlio", grazie ai metodi `getFirstChild()` e `getLastChild()`:

```
Node child = node.getFirstChild();
```

e

```
Node child = node.getLastChild();
```

I metodi `getNextSibling()` e `getPreviousSibling()`, permettono invece di accedere ai nodi "fratelli", ovvero i nodi che si trovano allo stesso livello di ramificazione. Con il seguente codice si accede al nodo "fratello" successivo:

```
Node sibling = node.getNextSibling();
```

Con il seguente codice invece, si accede al nodo "fratello" precedente:

```
Node sibling = node.getPreviousSibling();
```

I metodi `getFirstChild()`, `getLastChild()`, `getNextSibling()` e `getPreviousSibling()` restituiranno null nel caso in cui non trovino quanto richiesto.

Purtroppo lo sviluppatore con questi metodi alcune volta deve un po' ingegnarsi per poter accedere a determinati nodi. Per esempio con i prossimi due frammenti di codice si accede rispettivamente al primo e all'ultimo nodo "fratello".

```
Node sibling = node.getParentNode().getFirstChild();
```

e

```
Node sibling = node.getParentNode().getLastChild();
```

14.4.4 XPATH

Come è facile immaginare, se il documento da analizzare ha una struttura molto ramificata, non sarà sempre agevole riuscire ad analizzare uno specifico nodo, visto che la ricerca potrebbe essere anche molto complicata.
Per semplificare la ricerca dei nodi XML, esiste un linguaggio appositamente creato che si chiama XPath (le specifiche si trovano all'indirizzo http://www.w3c.org/TR/xpath). In realtà si tratta di uno dei linguaggi definiti dalla tecnologia XSLT insieme a XSL e XSL-FO, allo scopo di trasformare i documenti da XML in altri formati. Ma XPath ha trovato applicazione anche in altre tecnologie XML-based, ed è relativamente conosciuto da

molti sviluppatori. Java supporta XPath ufficialmente solo dalla versione 5 mediante la definizione del package `javax.xml.xpath`.
Il linguaggio XPATH, permette di raggiungere un determinato nodo (o attributo o testo etc…) mediante una sintassi semplice, senza dover obbligatoriamente esplorare l'intero albero DOM del documento XML. Per esempio anteporre due slash prima del nome del nodo all'interno di un'espressione XPATH, significa voler cercare il primo nodo con quell'identificatore, indipendentemente dalla sua posizione nel documento XML. Post-porre poi parentesi quadre che circondano un numero intero `i` al nome di un nodo, significa volere l'i-esimo nodo con quell'identificatore. Il simbolo di chiocciola invece serve per specificare gli attributi dei nodi.
Per esempio il frammento di codice è stato estratto e modificato dal mio progetto open-source XMVC (per informazioni http://sourceforge.net/projects/xmvc):

```
final static String FILE_XML = "resources/file.xml";
XPath xpath = XPathFactory.newInstance().newXPath();
InputSource inputSource = new InputSource(FILE_XML);
for (int i = 1; true; i++) {
    String expression = String.format(
      "//http-request[%d]", i);
    Node node = null;
    try {
        node = (Node) xpath.evaluate(expression,
          inputSource, XPathConstants.NODE);
        if (node == null) {
            break;
        }
        String requestURI = (String)
          xpath.evaluate("@requesturi", node);
        System.out.println("node " + node);
        System.out.println("requestURI " + requestURI);
    } catch (XPathExpressionException exc) {
        exc.printStackTrace();
   }
}
```

In pratica si legge un file XML, andandone a stampare tutti i nodi "http-request", e i relativi attributi "requesturi". Ma le potenzialità di XPATH non so fermano qui. La seguente tabella mostra una serie di espressioni XPATH e il relativo significato:

Espressione	Significato
`expression = "/*";`	Elemento root senza specificare il nome
`expression = "/root";`	Elemento root specificando il nome
`expression = "/root/*";`	Tutti gli elementi subito sotto root
`expression = "/root/node1";`	Tutti gli elementi node1 subito sotto root
`expression = "//node1";`	Tutti gli elementi node1 subito nel documento
`expression = "//node1[4]";`	Il quarto elemento node1 trovato nel documento
`expression = "//*[name()!='node1'";`	Tutti gli elementi che non siano node1
`expression = "//node1/node2";`	Tutti i node2 che sono figli di node1
`expression = "//*[not(node1)]";`	Tutti gli elementi il cui figli non è node1
`expression = "//*[*]";`	Tutti gli elementi con almeno un elemento figlio
`expression = "//*[count(node1) > 3]";`	Tutti gli elementi con almeno tre node1 figli
`expression = "/*/*/node1";`	Tutti i node1 al terzo livello

14.4.4 Modifica di un documento XML

Il codice seguente crea un documento XML da zero e aggiunge vari tipi di nodi, sfruttando diversi metodi:

```
1  DocumentBuilderFactory factory =
2  DocumentBuilderFactory.newInstance();
3  factory.setValidating(false);
4  Document doc = factory.newDocument();
5  Element root = doc.createElement("prova");
6  doc.appendChild(root);
7  Comment comment = doc.createComment("prova commento");
8  doc.insertBefore(comment, root);
9  Text text = doc.createTextNode("prova testo");
10 root.appendChild(text);
```

Con le prime quattro righe, creiamo un documento XML vuoto.
Con le righe 5 e 6, prima creiamo l'elemento root, che chiamiamo `prova`, e poi lo aggiungiamo al documento con il metodo `appendChild()`.
Con le righe 7 e 8 invece, creiamo un commento che poi andiamo ad anteporre al documento root.
Con le righe 9 e 10 infine, viene creato del testo e aggiunto con il metodo `appendChild()` all'unico elemento del documento.

Come già asserito in precedenza, qui è possibile notare come anche il testo di un nodo sia considerato un nodo esso stesso. Infatti l'interfaccia `Text` implementa `Node`.

Alla fine il documento finale sarà il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!—prova commento-->
<prova>
Prova testo
</prova>
```

Se come testo inserissimo caratteri che per XML sono considerati speciali come "<" o ">", sarebbero automaticamente convertiti dall'XML writer che si occupa di creare il documento nelle relative entità : rispettivamente in "<" e ">".

Per rimuovere un nodo è possibile utilizzare il metodo `removeChild()` nel seguente modo:

```
NodeList list = doc.getElementsByTagName("prova");
Element element = (Element)list.item(0);
element.getParentNode().removeChild(element);
```

Nell'esempio abbiamo dapprima individuato un nodo specifico all'interno del documento, grazie al metodo `getElementsByTagName()` su cui è stato chiamato i metodo `item(0)`. Infatti, `getElementsByTagName()` restituisce un oggetto `NodeList`, che contiene tutti i tag con il nome specificato. Con `item(0)` ovviamente ci viene restituito il primo della lista. Infine, per rimuovere il nodo individuato, abbiamo

dapprima dovuto ritornare al nodo "padre", per poi cancellarne il figlio con il metodo `removeChild()`.

Rimuovere un nodo non significa rimuovere i suoi sotto-nodi, e quindi neanche gli eventuali text node.

14.4.5 Analisi di un documento tramite parsing SAX

Sino ad ora abbiamo utilizzato DOM per poter analizzare il file perché sicuramente è il metodo preferito dagli sviluppatori. Nel seguente esempio invece utilizziamo un parsing SAX per esplorare un file xml, ottenendo la stampa di tutti i tag del documento:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
public class SaxParse {
  public static void main(String[] args) {
    DefaultHandler myHandler = new MyHandler();
    try {
      SAXParserFactory factory =
         SAXParserFactory.newInstance();
      factory.setValidating(false);
      factory.newSAXParser().parse(
        new File("nomeFile.xml"), myHandler);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  static class MyHandler extends DefaultHandler {

    public void startDocument() {
      System.out.println("---Inizio parsing---");
    }

    public void startElement(String uri, String
localName,
        String qName, Attributes attributes) {
     System.out.println("<"+qName+ ">");
```

```
    }

    public void endElement(String uri, String localName,
        String qName) {
      System.out.println("</"+ qName+ ">");
    }

    public void endDocument() {
      System.out.println("---Fine parsing---");
    }
  }
}
```

SAX, come già asserito in precedenza, si basa sul concetto di handler (gestore di eventi). Quando viene lanciato un parsing di un documento tramite SAX, la lettura di ogni nodo rappresenta un evento da gestire. Analizzando l'esempio, concentriamoci dapprima sulla classe interna (cfr. Modulo 8) `MyHandler`. Come è possibile notare `MyHandler` è sottoclasse di `DefaultHandler`, e ne ridefinisce alcuni metodi. Questi, come già si può intuire dai loro nomi, vengono chiamati in base agli eventi che rappresentano. Infatti, il corpo del metodo `main()` è piuttosto semplice. Si istanzia prima `MyHandler`, poi in un blocco `try-catch`, viene istanziato un oggetto di tipo `SAXParserFactory`, a cui viene imposto di ignorare una eventuale validazione del documento con il metodo `setValidating()`. Infine viene lanciato il parsing del file "nomeFile.xml" su di un nuovo oggetto `SAXParser`, specificando come gestore di eventi l'oggetto `myHandler`. Da questo punto in poi sarà la stessa Java Virtual Machine ad invocare i metodi della classe `MyHandler` sull'oggetto `myHandler`, in base agli eventi scatenati dalla lettura sequenziale del file XML.

14.4.6 Trasformazioni XSLT

Il package `javax.xml.transform` e i suoi sotto-package, permettono di utilizzare le trasformazioni dei documenti XML in base alla tecnologia XSLT (per informazioni su XSLT: http://www.w3c.org). Questo package definisce due interfacce chiamate `Source` (sorgente da trasformare) e `Result` (risultato della trasformazione), che vengono utilizzate per le trasformazioni. Ovviamente si devono utilizzare classi concrete per poter utilizzare le trasformazioni. Esistono tre implementazioni di coppie `Source - Result`:

- `StreamSource` e `StreamResult`
- `SAXSource` e `SAXResult`
- `DOMSource` e `DOMResult`

`StreamSource` e gli `StreamResult`, essenzialmente servono per avere rispettivamente in input e output flussi di dati, per esempio file. Quindi, se volessimo per esempio scrivere un documento XML in un file, potremmo utilizzare il seguente codice:

```
try {
    Source source = new DOMSource(doc);
    File file = new File("fileName.xml");
    Result result = new StreamResult(file);
    Transformer transformer =
      TransformerFactory.newInstance().newTransformer();
    transformer.transform(source, result);
} catch (Exception e) {
    e.printStackTrace();
}
```

Dando per scontato che `doc` sia un documento DOM, come sorgente abbiamo utilizzato un oggetto di tipo `DOMSource`. Siccome vogliamo scrivere in un file allora abbiamo utilizzato come oggetto di output uno `StreamResult`. Il metodo statico `newInstance()` invece, in base a determinati criteri (cfr. documentazione) istanzierà un oggetto di tipo `TransformerFactory`, che poi istanzierà un oggetti di tipo `Transformer`. L'oggetto `transformer` infine, grazie al metodo `transform()`, trasformerà il contenuto del documento DOM nel file.

Gli oggetti di tipo `Source` e `Result` possono essere utilizzati una sola volta (cfr. documentazione), dopodichè bisogna istanziarne degli altri.

Inoltre la classe `Transformer` definisce il metodo `setOutputProperty()` che si può sfruttare per esempio per garantirsi degli output personalizzati (cfr. documentazione). Per esempio, il seguente codice, se venisse eseguito prima del metodo `transform()` dell'esempio precedente, provocherebbe che nel file XML, venga scritto solamente il testo e non i tag del documento.
transformer.setOutputProperty(OutputKeys.METHOD, "text");

Ovviamente, con XSLT, è possibile fare molto di più. Con il seguente codice per esempio, è possibile ottenere un transformer basato su un file XSL per una trasformazione del documento DOM:

```
TransformerFactory tf = TransformerFactory.newInstance();
Templates template = tf.newTemplates(new StreamSource(
    new FileInputStream("fileName.xsl")));
Transformer transformer = template.newTransformer();
```

il resto del codice rimane identico a quello dell'esempio precedente.

Riepilogo

Questo modulo è stato dedicato alla gestione dei dati con Java. Attualmente, le due modalità di gestione dati applicativi, sono basati su database e file. In particolare, su database relazionali che sfruttano il linguaggio SQL, e file formattati con il linguaggio XML. È inevitabile che gli sviluppatori, prima o poi abbiano a che fare con questi due linguaggi universali e quindi si raccomanda al lettore inesperto, quantomeno qualche lettura ed esercizio di base su entrambi gli argomenti. Risorse gratuite su SQL e XML su Internet sono fortunatamente diffusissime.
In questo modulo abbiamo dapprima esplorato la struttura e le basi dell'interfaccia JDBC. Poi abbiamo introdotto con degli esempi alcune delle caratteristiche più importanti ed avanzate come le stored procedure (con i `CallableStatement`), gli statement parametrizzati (con i `PreparedStatement`), e la gestione delle transazioni.
Infine abbiamo cercato di introdurre la libreria JAXP, con un approccio basato su esempi. Sono stati affrontati i principali problemi che solitamente si incontrano con i metodi di analisi DOM e SAX, e toccato il mondo delle trasformazioni XSLT e di XPATH.

Esercizi modulo 14

Esercizio 14.a)
JDBC, Vero o Falso:

1. L'implementazione del driver JDBC da parte del vendor, è costituita solitamente dalla implementazioni delle interfacce del package `java.sql`
2. `Connection` è solo un'interfaccia
3. Un'applicazione JDBC, è indipendente dal database solo se si parametrizzano le stringhe relative al driver, la url di connessione, la username e la password
4. Se si inoltra un comando ad un particolare database, che non sia standard SQL 2, questo comando funzionerà solo su quel database. In questo modo si perde l'indipendenza dal database, a meno di controlli o parametrizzazioni
5. Se si inoltra un comando ad un particolare database, che non sia standard SQL 2, l'implementazione JDBC lancerà un'eccezione
6. Per cancellare un record bisogna utilizzare il metodo `executeQuery()`
7. Per aggiornare un record bisogna utilizzare il metodo `executeUpdate()`
8. `CallableStatement` è una sotto-interfaccia di `PreparedStatement` `PreparedStatement` è una sotto-interfaccia di `Statement`
9. Per eseguire una stored procedure, bisogna utilizzare il metodo `execute()`
10. L'auto-commit è settato a `true` di default

Esercizio 14.b)
JAXP, Vero o Falso:

1. Per le specifiche DOM, ogni nodo è equivalente ad un altro e un commento viene visto come un oggetto di tipo `Node`
2. Infatti l'interfaccia `Node` implementa `Text`
3. Per poter analizzare un documento nella sua interezza con DOM, bisogna utilizzare un metodo ricorsivo
4. Con il seguente codice:
   `Node n = node.getParentNode().getFirstChild();`
   si raggiunge il primo nodo "figlio" di `node`
5. Con il seguente codice:
   `Node n = node.getParentNode().getPreviousSibling();`
   si raggiunge il nodo "fratello" precedente di `node`

6. Con il seguente codice:
   ```
   NodeList list = node.getChildNodes();
     Node n = list.item(0);
   ```
   si raggiunge il primo nodo "figlio" di `node`
7. Con il seguente codice:
   ```
   Element element = doc.createElement("nuovo");
     doc.appendChild(element);
     Text text = doc.createTextNode("prova testo");
     doc.insertBefore(text, element);
   ```
   viene creato un nodo chiamato `nuovo`, e gli viene aggiunto al suo interno del testo
8. La rimozione di un nodo provoca la rimozione di tutti i suoi nodi "figli"
9. Per analizzare un documento tramite l'interfaccia SAX, bisogna estendere la classe `DefaultHandler` e fare override dei suoi metodi
10. Per trasformare un file XML e serializzarlo in un altro file dopo una trasformazione mediante un file XSL, è possibile utilizzare il seguente codice:

```
try {
    TransformerFactory factory =
TransformerFactory.newInstance();
    Source source = new StreamSource(new
File("input.xml"));
    Result result = new StreamResult(new
File("output.xml"));
    Templates template = factory.newTemplates(
      new StreamSource(new
FileInputStream("transformer.xsl")));
    Transformer transformer = template.newTransformer();
    transformer.transform(source, result);
} catch (Exception e) {
    e.printStackTrace();
}
```

Soluzioni esercizi modulo 14

Esercizio 14.a)
JDBC, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Vero**
4. **Vero**
5. **Falso**
6. **Falso**
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Esercizio 14.b)
JAXP, Vero o Falso:

1. **Vero**
2. **Falso** l'interfaccia `Text` implementa `Node`
3. **Vero**
4. **Falso** si raggiunge il primo nodo "fratello" di `node`
5. **Falso**
6. **Vero**
7. **Falso** il testo viene aggiunto prima del tag con il metodo `insertBefore()`. Sarebbe invece opportuno utilizzare la seguente istruzione per aggiungere il testo all'interno del tag `nuovo`:
   `element.appendChild(text);`
8. **Falso**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper scrivere codice che si connette a qualsiasi tipo di database (unità 14.2, 14.3)	☐	
Saper scrivere codice che aggiorna, interroga e gestisce i risultati qualsiasi sia il database in uso (unità 14.2, 14,3)	☐	
Avere confidenza con le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità 13.4)	☐	
Saper gestire i concetti della libreria JAXP, per la gestione dei documenti XML (unità 14.4)	☐	
Saper risolvere i problemi di utilizzo delle interfacce DOM e SAX per l'analisi dei documenti XML (unità 14.4)	☐	
Saper trasformare con XSLT i documenti XML (unità 14.4)	☐	

Note:

15

Complessità: media

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper elencare le principali caratteristiche che deve avere una GUI (unità 15.1).
2. Saper descrivere le caratteristiche della libreria AWT (unità 15.2).
3. Saper gestire i principali Layout Manager per costruire GUI complesse (unità 15.3).
4. Saper gestire gli eventi con il modello a delega (unità 15.4).
5. Saper creare semplici applet (unità 15.5).
6. Saper descrivere le caratteristiche della libreria Swing (unità 15.6).

15 Interfacce grafiche (GUI) con AWT, Applet e Swing

Siamo ora pronti per parlare di grafica…

15.1 Introduzione alle Graphical User Interface (GUI)

In questo modulo finalmente impareremo a creare le interfacce grafiche che solitamente hanno i programmi moderni. Oggigiorno, le GUI sono una parte molto importante delle nostre applicazioni per varie ragioni. Basti pensare solo al fatto che la maggior parte degli utenti di software applicativo, giudica lo stesso software, principalmente dalla GUI con cui si interfaccia. Ovviamente, questo non vale (o vale di meno) per gli utenti che utilizzano gli applicativi più tecnici come i tool di sviluppo, dove magari si preferisce avere una brutta GUI, ma delle funzionalità che aiutano nello sviluppo. Esistono dei principi per giudicare un'interfaccia che elenchiamo di seguito:

- Utenza: è fondamentale tenere ben conto, che tipologie di utenza usufruirà della GUI che creeremo. Per esempio, utenti esperti potrebbero non gradire di dover utilizzare dei wizard per effettuare alcune procedure. Utenti meno esperti invece, potrebbero invece gradire di essere guidati.
- Semplicità: non bisogna mai dimenticare che l'obiettivo di una GUI, è quella di facilitare l'uso dell'applicazione all'utente non di complicarglielo. Creare GUI semplici quindi, è una priorità. Bisogna sempre far sì che l'utente non pensi mai di "essersi perso".

- Usabilità: occorre avere un occhio di riguardo in fase di progettazione di una GUI, affinché essa offra un utilizzo semplice ed immediato. Per esempio offrire delle scorciatoie con la tastiera potrebbe aiutare molto alcune tipologie di utenti.
- Estetica: ovviamente un ruolo importante lo gioca la piacevolezza dello stile che diamo alla nostra GUI. Essendo questa soggettiva, non bisognerebbe mai scostarsi troppo dagli standard conosciuti, oppure offrire la possibilità di personalizzazione.
- Riuso: una buona GUI, seguendo le regole dell'object orientation, dovrebbe anche offrire dei componenti riutilizzabili. Come già accennato precedentemente però, bisogna scendere a compromessi con l'object orientation quando si progettano GUI.
- Gusti personali e standard: quando bisogna scegliere come creare una certa interfaccia, non bisogna mai dimenticare che i gusti personali sono sempre trascurabili rispetto agli standard a cui gli utenti sono abituati.
- Consistenza: le GUI devono sempre essere consistenti dovunque vengano eseguite. Questo significa anche che è fondamentale sempre esporre all'utente delle informazioni interessanti. Inoltre le differenze che ci sono tra una vista ed un'altra, devono essere significative. Infine, qualsiasi sia l'azione (per esempio il ridimensionamento) che l'utente effettua sulla GUI, quest'ultima deve sempre rimanere significativa e utile.
- Internazionalizzazione: nel modulo relativo al package `java.util`, abbiamo già parlato di questo argomento. Nel caso di creazione di una GUI, l'internazionalizzazione può diventare molto importante.
- Model-View-Controller: nella creazione delle GUI moderne, esiste uno schema architetturale che viene utilizzato molto spesso: il pattern Model View Controller (MVC). In verità non viene sempre utilizzato nel modo migliore seguendo tutte le sue linee guida, ma almeno uno dei suoi principi deve assolutamente essere considerato: la separazione dei ruoli. Infatti nell'MVC, si separano tre componenti a seconda del loro ruolo nell'applicazione:

    1. il Model che implementa la vera applicazione, ovvero non solo i dati ma anche le funzionalità. In pratica quello che abbiamo studiato sino ad ora, serve per creare Model (ovvero applicazioni). Si dice che il Model, all'interno dell'MVC, implementa la "logica di business" (logica applicativa).
    2. la View che è composta da tutta la parte dell'applicazione con cui si interfaccia l'utente. Questa parte, solitamente costituita da GUI multiple, deve implementare la logica che viene detta "logica di presentazione", ovvero la logica per organizzare se stessa. Questo tipo di logica, come

vedremo in questo modulo, è tutt'altro che banale. La View non contiene nessun tipo di funzionalità o dato applicativo, "espone" solamente all'utente le funzionalità dell'applicazione.

3. il Controller che implementa la "logica di controllo". Questo è il componente più difficile da immaginare in maniera astratta, anche perché non si sente parlare spesso di "logica di controllo". Giusto per dare un'idea, diciamo solo che questo componente deve avere almeno queste responsabilità: controllare gli input che l'utente immette nella GUI, decidere quale sarà la prossima pagina della View che sarà visualizzata, mappare gli input utente nelle funzionalità del Model.

I vantaggi dell'applicazione dell'MVC sono diversi. Uno di questi, il più evidente, è quello che se deve cambiare l'interfaccia grafica, non deve cambiare l'applicazione. Concludendo, raccomandiamo al lettore quantomeno di non confondere mai il codice che riguarda la logica di business, con il codice che riguarda la logica di presentazione.

Per maggiori dettagli ed esempi di codice rimandiamo all'appendice D, interamente dedicata all'MVC. Per la definizione di pattern invece il lettore può consultare l'appendice H. Se invece il lettore è intenzionato a proseguire il suo percorso formativo Java proprio con lo studio dei Design Pattern fondamentali, consigliamo da questo stesso editore, il libro di Massimiliano Bigatti "UML e Design Pattern".

Alcuni IDE (Integrated Development Editor) per Java, come JBuilder della Borland o WSAD di IBM, permettono la creazione di GUI in maniera grafica tramite il trascinamento dei componenti, come avviene in altri linguaggi come Visual Basic e Delphi. Questo approccio, ovviamente abbatte i tempi di sviluppo della GUI, ma concede poco spazio alle "modifiche a mano" e porta a "dimenticare" il riuso. Inoltre, il codice scritto da un IDE non è assolutamente paragonabile a quello scritto da un programmatore. Per tali ragioni chi vi scrive, ha sempre evitato di scrivere codice ufficiale con tali strumenti. Comunque, in questo modulo gli argomenti verranno presentati come se si dovesse scrivere ogni singola riga.

Avvertiamo il lettore inoltre che, quando si dota un programma di una GUI, cambia completamente il ciclo di vita del programma stesso. Infatti, mentre tutte le applicazioni sviluppate fino ad adesso, duravano il "tempo di eseguire un `main()`" (e tutti i thread creati), adesso le cose cambiano radicalmente. Una volta che viene visualizzata

una GUI, la Java Virtual Machine fa partire un nuovo thread che si chiama "AWT thread", mantiene sullo schermo la GUI stessa, cattura eventuali eventi su di essa. Quindi un'applicazione che fa uso di GUI, una volta lanciata, rimane in attesa dell'input dell'utente e termina solo in base a un determinato input.

15.2 Introduzione ad Abstract Window Toolkit (AWT)

AWT è una libreria per creare interfacce grafiche utente sfruttando componenti dipendenti dal sistema operativo. Ciò significa che eseguendo la stessa applicazione grafica su sistemi operativi differenti, lo stile dei componenti grafici (in inglese detto "Look & Feel") sarà imposto dal sistema operativo. Nella figura 15.1) si può notare la stessa semplice GUI visualizzata su due sistemi operativi diversi (in questo caso Windows XP e Linux Fedora Core 3 con interfaccia Motif di KDE)

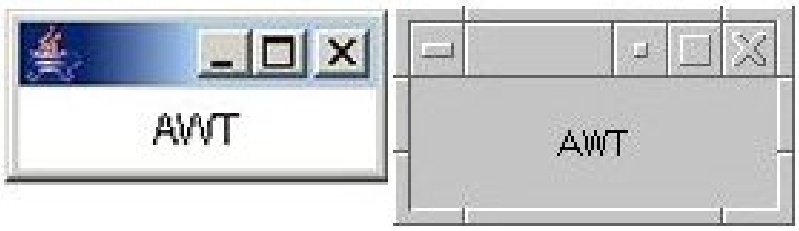


Figura 15.1 – "La stessa GUI visualizzata su Windows e Linux"

Il codice per generare la GUI in figura 15.1), è stata generata dal seguente codice:

```
import java.awt.*;
public class AWTGUI {
    public static void main(String[] args) {
        Frame frame = new Frame();
        Label l = new Label("AWT", Label.CENTER);
        frame.add(l);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Basta conoscere un po' di inglese per analizzare il codice precedente…

Un'applicazione grafica AWT, non si può terminare chiudendo il frame!. Infatti, il click sulla "X" della finestra, per la JVM è un "evento da gestire". Quindi bisognerà studiare l'unità didattica relativa alla gestione degli eventi prima di poter chiudere le nostre applicazioni come siamo abituati. Per adesso bisognerà interrompere il processo in uno dei seguenti modi:

1. Se abbiamo lanciato la nostra applicazione da riga di comando, basta premere contemporaneamente `CTRL-C` avendo in primo piano la prompt dei comandi.
2. Se abbiamo lanciato la nostra applicazione utilizzando EJE o un qualsiasi altro IDE, troverete un bottone o una voce di menu che interrompe il processo.
3. Se proprio non ci riuscite potrete accedere sicuramente in qualche modo alla lista dei processi (Task Manager su sistemi Windows moderni) sul vostro sistema operativo, ed interrompere quello relativo all'applicazione Java.

15.2.1 Struttura della libreria AWT ed esempi

La libreria AWT offre comunque una serie abbastanza ampia di classi ed interfacce per la creazione di GUI. È possibile utilizzare bottoni, checkbox, liste, combo box (classe `Choice`), label, radio button (utilizzando checkbox raggruppati mediante la classe `CheckboxGroup`), aree e campi di testo, scrollbar, finestre di dialogo (classe `Dialog`), finestre per navigare sul file system (classe `FileDialog`) etc….
Per esempio il seguente codice crea un'area di testo con testo iniziale "`Java AWT`", 4 righe, 10 colonne e con la caratteristica di andare da capo automaticamente. La costante statica `SCROLLBARS_VERTICAL_ONLY` infatti, verrà interpretata dal costruttore in modo tale da utilizzare solo scrollbar verticali e non orizzontali in caso di "sforamento".

```
TextArea ta = new TextArea("Java AWT", 4,
10,TextArea.SCROLLBARS_VERTICAL_ONLY);
```

Il numero di colonne è puramente indicativo. Per un certo tipo di font, una "w" potrebbe occupare lo spazio di tre "i".

La libreria AWT, dipendendo strettamente dal sistema operativo, definisce solamente i componenti grafici che appartengono all'intersezione comune dei sistemi operativi più diffusi. Per esempio, l'albero (in inglese "tree") di windows (vedi "esplora risorse"), non esistendo su tutti i sistemi operativi, non è contenuto in AWT.
È molto semplice creare anche ,menu personalizzati tramite le classi `MenuBar`, `Menu`, `MenuItem`, `CheckboxMenuItem` ed eventualmente `MenuShortcut` per utilizzarli

direttamente con la tastiera mediante le cosiddette "scorciatoie".
Segue un semplice frammento di codice che crea un piccolo menu:

```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
MenuItem mi1 = new MenuItem("New");
MenuItem mi2 = new MenuItem("Open");
MenuItem mi3 = new MenuItem("Save");
MenuItem mi4 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```

Non ci dovrebbe essere bisogno di spiegazioni…

Notare come il menu "`Help`", sia stato aggiunto diversamente dagli altri mediante il metodo `setHelpMenu()`. Questo perché su un ambiente grafico come quello di Solaris (sistema operativo creato da Sun), il menu di Help viene piazzato all'estrema destra della barra dei menu, e su altri sistemi potrebbe avere posizionamenti differenti. Per quanto semplice sia l'argomento quindi, è comunque necessario dare sempre uno sguardo alla documentazione prima di utilizzare una nuova classe.

La classe `Toolkit` ci permette di accedere a varie caratteristiche grafiche e non grafiche, del sistema su cui ci troviamo. Per esempio il metodo `getScreenSize()` ci restituisce un oggetto `Dimension` con all'interno la dimensione del nostro schermo. Inoltre offre il supporto per la stampa tramite il metodo `getPrintJob()`. Per ottenere un oggetto `Toolkit` possiamo utilizzare il metodo `getDefaultToolkit()`:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
```

AWT ci offre anche la possibilità di utilizzare i font di sistema o personalizzati, tramite la classe `Font`. Per esempio quando deve stampare un file, EJE utilizza il seguente `Font`:

```
Font font = new Font("Monospaced", Font.BOLD, 14);
```

Con AWT è anche possibile disegnare. Infatti ogni `Component` può essere esteso e si può overridare il metodo `paint(Graphics g)`, che ha la responsabilità di disegnare il componente stesso. In particolare, la classe `Canvas` (in italiano "tela"), è stata creata proprio per diventare un componente da estendere allo scopo di disegnarci sopra. Come esempio quindi creeremo proprio un oggetto `Canvas`:

```
public class MyCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawString("java",10,10);
        g.setColor(Color.red);
        g.drawLine(10,5, 35,5);
    }
}
```

La classe precedente stampa la scritta "`java`" con una linea rossa che l'attraversa. Se poi guardiamo la documentazione di `Graphics`, troveremo diversi metodi per disegnare ovali, rettangoli, poligoni etc…
Con la classe `Color` si possono anche creare colori ad hoc con lo standard RGB (red, green e blue), specificando l'intensità di ciascun colore con valori compresi tra 0 e 255:

```
Color c = new Color (255, 10 ,110 );
```

In figura 15.2, viene presentata un parte significativa della gerarchia di classi di AWT.

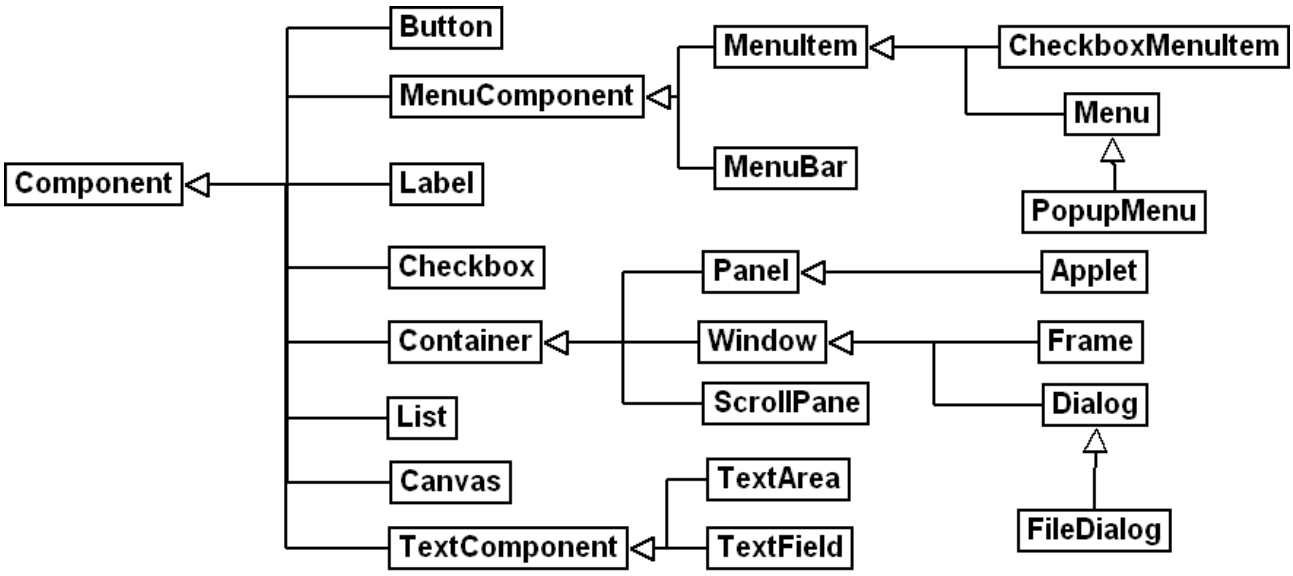


Figura 15.2 – "Gerarchia di classi di AWT (Composite Pattern)"

In pratica si tratta di un'implementazione del pattern strutturale GoF chiamato "Composite".
La classe astratta `Component` astrae il concetto di componente generico astratto. Quindi, ogni componente grafico è sottoclasse di `Component` (è un component). Per esempio `Button`, e `Checkbox` sono sottoclassi dirette di `Component` e ridefiniscono il metodo `paint()`. Come già asserito, questo ha il compito di disegnare il componente stesso e quindi viene ovviamente reimplementato in tutte le sottoclassi di `Component`.
Tra le sottoclassi di `Component`, bisogna però notarne una un po' particolare: la classe `Container`. Questa astrae il concetto di componente grafico astratto che può contenere altri componenti. Non è una classe astratta, ma solitamente vengono utilizzate le sue sottoclassi `Frame` e `Panel`.

Le applicazioni grafiche si basano sempre su di un "top level container", ovvero un container di primo livello. Infatti, avete mai visto un'applicazione dove ci sono dei bottoni, ma non c'è una finestra che li contiene? Questo significa che in ogni applicazione Java con interfaccia grafica, è necessario quantomeno istanziare un top level container, di solito un Frame. Un caso speciale è rappresentato dalle applet, di cui parleremo tra poco.

La caratteristica chiave dei container è di avere un metodo `add(Component c)`, che permette
di aggiungere altri componenti come `Button`, `Checkbox`, ma anche altri container (che essendo sottoclasse di `Component` sono anch'essi component). Per esempio, è possibile aggiungere `Panel` a `Frame` .
Il primo problema, che si pone è: dove posiziona il componente aggiunto il container, se ciò non viene specificato esplicitamente? La risposta è nella prossima unità didattica.

15.3 Creazione di interfacce complesse con i layout manager

La posizione di un componente aggiunto a un container dipende essenzialmente dall'oggetto che è associato al container, detto "layout manager". In ogni container infatti, esiste un layout manager associato di default. Un layout manager è un'istanza di una classe che implementa l'interfaccia `LayoutManager`. Esistono decine di implementazioni di `LayoutManager`, ma tutto sommato le più importanti sono solo cinque:

- `FlowLayout`
- `BorderLayout`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

In questo modulo introdurremo le prime quattro, accennando solo al `GridBagLayout`. Come al solito il lettore interessato potrà approfondire lo studio di quest'ultima classe con la documentazione Sun.

Anche la dimensione dei componenti aggiunti dipenderà dal layout manager.

La Figura 15.3, mostra come tutte le sottoclassi di `Window` (quindi anche `Frame`), abbiano associato per default il `BorderLayout`, mentre tutta la gerarchia di `Panel` utilizzi il `FlowLayout` per il posizionamento dei componenti.

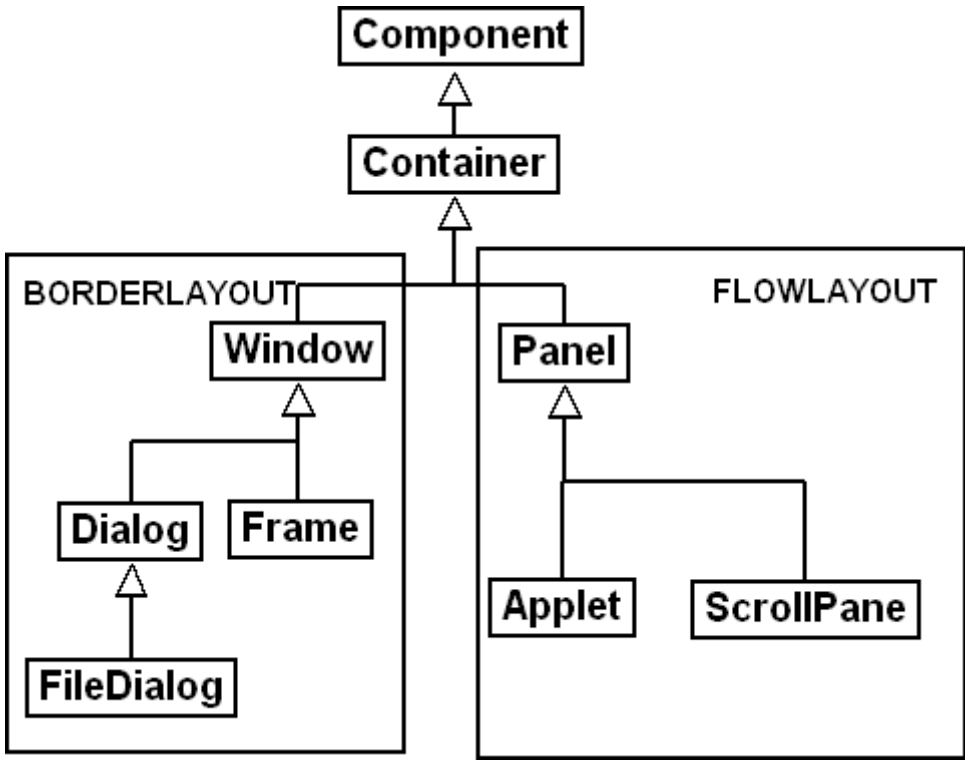


Figura 15.3 – "Layout manager associati di default"

In realtà è anche possibile non utilizzare layout manager per gestire interfacce grafiche. Tale tecnica però comprometterebbe la consistenza e la portabilità della GUI stessa. Il lettore interessato può provare per esempio ad annullare il layout di un container (per esempio un `Frame`) con l'istruzione `setLayout(null)`, per poi usare i metodi `setLocation()`, `setBounds()` e `setSize()` per gestire il posizionamento dei componenti.

15.3.1 Il FlowLayout

Il `FlowLayout` è il layout manager di default di `Panel`, come vedremo una delle classi principali del package AWT. `FlowLayout` dispone i componenti aggiunti in un flusso ordinato che va da sinistra a destra con un allineamento centrato verso l'alto. Per esempio il codice seguente (da inserire all'interno di un metodo `main()`):

```
Frame f = new Frame("FlowLayout");
Panel p = new Panel();
Button button1 = new Button("Java");
```

```
Button button2 = new Button("Windows");
Button button3 = new Button("Motif");
p.add(button1);
p.add(button2);
p.add(button3);
f.add(p);
f.pack();
f.setVisible(true);
```

Produrrebbe come output quanto mostrato in figura 15.4. Notare che il metodo `pack()`, semplicemente ridimensiona il frame in modo tale da mostrarsi abbastanza grande da visualizzare il suo contenuto.

Figura 15.4 – "Il FlowLayout in azione"

In particolare, le figura 15.5 e 15.6 mostrano anche come si dispongono i bottoni dopo avere ridimensionato la finestra che contiene il `Panel`. Nella figura 15.5 è possibile vedere come l'allargamento della finestra non alteri la posizione dei bottoni sul `Panel`.

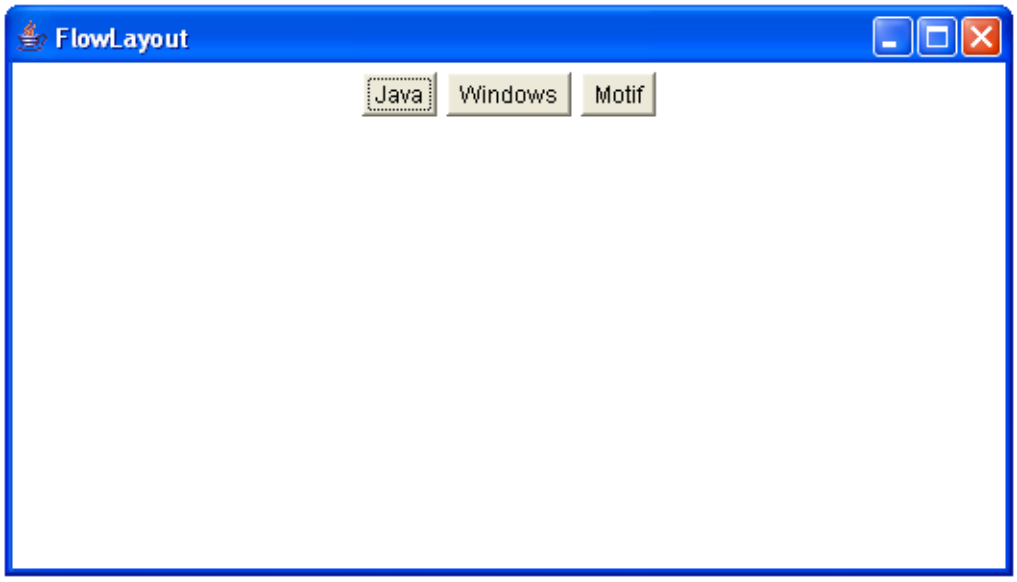


Figura 15.5 – "Il FlowLayout dopo aver allargato il frame"

Mentre nella figura 15.6, è possibile vedere come i bottoni si posizionino in maniera coerente con la filosofia del `FlowLayout`, in posizioni diverse dopo aver ristretto molto il frame.

Figura 15.6 – "Il FlowLayout dopo aver ristretto il frame"

Il `FlowLayout` utilizza per i componenti aggiunti la loro "dimensione preferita". Infatti, tutti i componenti ereditano dalla classe `Component` il metodo `getPreferredSize()` (in italiano "dammi la dimensione preferita"). Il `FlowLayout` chiama questo metodo per ridimensionare i componenti prima di aggiungerli al container. Per esempio il metodo `getPreferredSize()` della classe `Button`, dipende dall'etichetta che gli viene settata. Un bottone con etichetta "`OK`", avrà dimensioni molto più piccole rispetto ad un bottone con etichetta "`Ciao io sono un bottone AWT`".

15.3.2 Il BorderLayout

Il `BorderLayout` è il layout manager di default per i `Frame`, il top level container per eccellenza. I componenti sono disposti solamente in cinque posizioni specifiche che si ridimensionano automaticamente:

- `NORTH`, `SOUTH` che si ridimensionano orizzontalmente
- `EAST`, `WEST` che si ridimensionano verticalmente
- `CENTER` che si ridimensiona orizzontalmente e verticalmente

Questo significa che un componente aggiunto in una certa area, si deformerà per occupare l'intera area. Segue un esempio:

```
import java.awt.*;

public class BorderExample {
    private Frame f;
    private Button b[]={new Button("b1"),new
      Button("b2"),new Button("b3"),
      new Button("b4"), new Button("b5")};

    public BorderExample() {
        f = new Frame("Border Layout Example");
    }

    public void setup() {
        f.add(b[0], BorderLayout.NORTH);
```

```
        f.add(b[1], BorderLayout.SOUTH);
        f.add(b[2], BorderLayout.WEST);
        f.add(b[3], BorderLayout.EAST);
        f.add(b[4], BorderLayout.CENTER);
        f.setSize(200,200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        new BorderExample().setup();
    }
}
```

La figura 15.7, mostra l'output della precedente applicazione.

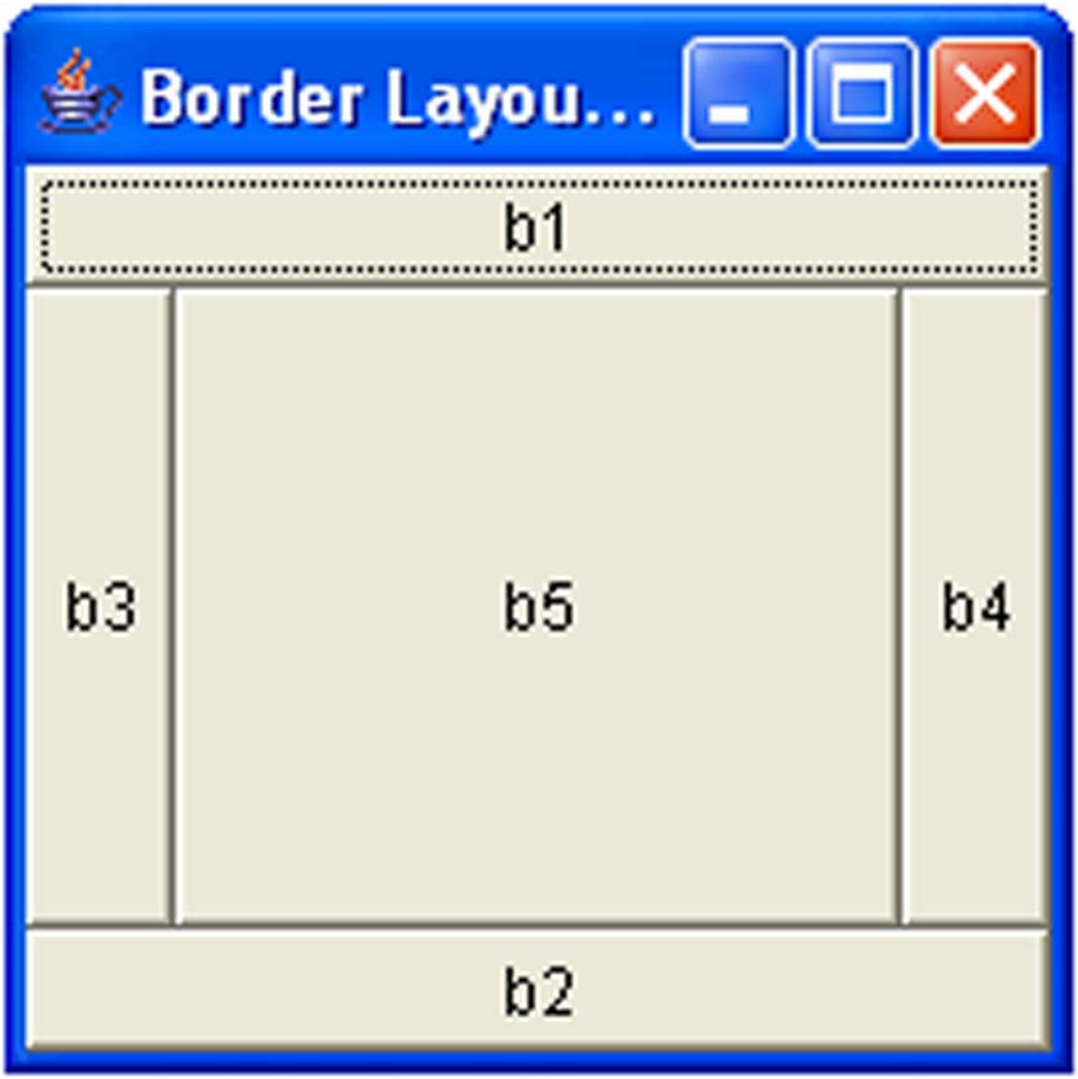


Figura 15.7 – "Il BorderLayout in azione"

Quando si aggiungono componenti con il `BorderLayout` quindi, si utilizzano il metodo `add(Component c, int position)` o `add(Component c, String position)`. Se si utilizza il metodo `add(Component c)`, il componente verrà aggiunto al centro dal `BorderLayout`.

1.1.2. 15.3.3 Il GridLayout

Il `GridLayout` dispone i componenti da sinistra verso destra e dall'alto verso il basso all'interno di una griglia. Tutte le regioni della griglia hanno sempre la stessa dimensione (anche se vengono modificate le dimensioni del frame), e i componenti occuperanno tutto lo spazio possibile all'interno delle varie regioni. Il costruttore del

`GridLayout` permette di specificare righe e colonne della griglia. Come per il `BorderLayout`, i componenti occuperanno interamente le celle in cui vengono aggiunti.
Il seguente codice mostra come può essere utilizzato il `GridLayout`:

```
import java.awt.*;

public class GridExample {
    private Frame f;
    private Button b[]={new Button("b1"),new
      Button("b2"),new Button("b3"), new Button("b4"),
      new Button("b5"), new Button("b6")};

    public GridExample() {
        f = new Frame("Grid Layout Example");
    }

    public void setup() {
        f.setLayout(new GridLayout(3,2));
        for (int i=0; i<6;++i)
            f.add(b[i]);
        f.setSize(200,200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        new GridExample().setup();
    }
}
```

La figura 15.8, mostra l'output della precedente applicazione.

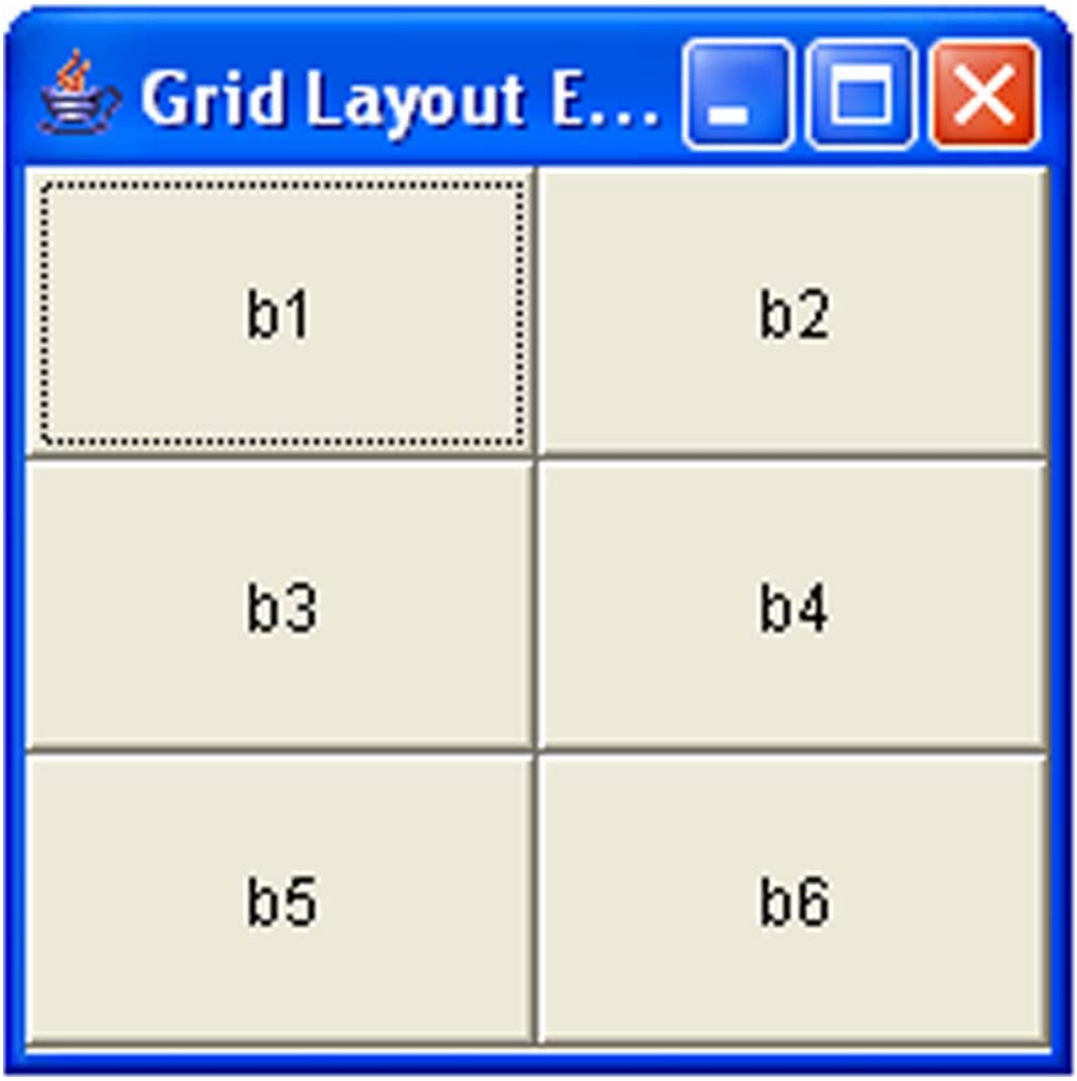


Figura 15.8 – "Il GridLayout in azione"

Per quanto riguarda il costruttore di `GridLayout` che abbiamo appena utilizzato nell'esempio, è possibile specificare che il numero delle righe o delle colonne può essere zero (ma non può essere zero per entrambi). Lo zero viene inteso come "un numero qualsiasi di oggetti che può essere piazzato in una riga o una colonna".

15.3.4 Creazione di interfacce grafiche complesse

Cerchiamo ora di capire come creare GUI con layout più complessi. È possibile infatti sfruttare i vari layout in un'unica GUI, creando così un layout composito, complesso e stratificato. In un `Frame` per esempio, possiamo inserire molti container (come i

`Panel`), che a loro volta possono disporre i componenti mediante il proprio layout manager.
Il seguente codice mostra come creare una semplice interfaccia per uno "strano" editor:

```
import java.awt.*;
public class CompositionExample {
    private Frame f;
    private TextArea ta;
    private Panel p;
    private Button b[]={new Button("Open"),new
      Button("Save"), new Button("Load"), new
      Button("Exit")};

    public CompositionExample() {
        f = new Frame("Composition Layout Example");
        p = new Panel();
        ta = new TextArea();
    }

    public void setup() {
        for (int i=0; i<4;++i)
            p.add(b[i]);
        f.add(p,BorderLayout.NORTH);
        f.add(ta,BorderLayout.CENTER);
        f.setSize(350,200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        new CompositionExample().setup();
    }
}
```

La figura 15.9, mostra l'output della precedente applicazione.

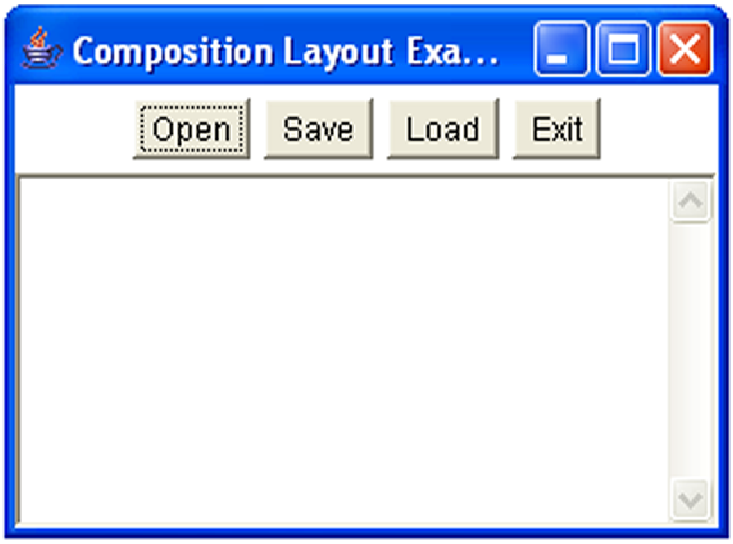


Figura 15.9 – "L'interfaccia per un semplice editor"

In pratica, componendo i layout tramite questa tecnica, è possibile creare un qualsiasi tipo di interfaccia.

Il consiglio in questo caso è di progettare con degli schizzi su di un foglio di carta, tutti gli strati che dovranno comporre l'interfaccia grafica. È difficile creare una GUI senza utilizzare questa tecnica, che tra l'altro, suggerisce anche eventuali container riutilizzabili.

15.3.5 Il GridBagLayout

Il `GridBagLayout` può organizzare interfacce grafiche complesse da solo. Infatti, anch'esso è capace di dividere il container in una griglia, ma, a differenza del `GridLayout`, può disporre i suoi componenti in modo tale che si estendano anche oltre un'unica cella. Quindi, anche nella più complicata dell'interfacce, è idealmente possibile dividere in tante celle il container quanto sono i pixel dello schermo e piazzare i componenti a proprio piacimento. Anche se quella appena descritta non è una soluzione praticabile, rende l'idea della potenza del `GridBagLayout`.
Si può tranquillamente affermare che da solo il `GridBagLayout` può sostituire i tre precedenti layout manager di cui abbiamo parlato. In compenso però, la difficoltà di utilizzo è notevole (vedi documentazione).

1.1.3. 15.3.6 Il CardLayout

Il `CardLayout`, è un layout manager particolare che permetterà di posizionare i vari componenti uno sopra l'altro, come le carte in un mazzo.
Il seguente esempio mostra come è possibile disporre i componenti utilizzando un `CardLayout`:

```
import java.awt.*;
public class CardTest {
    private Panel p1, p2, p3;
    private Label lb1, lb2, lb3;
    private CardLayout cardLayout;
    private Frame f;

    public CardTest() {
        f = new Frame ("CardLayout");
        cardLayout = new CardLayout();
        p1 = new Panel();
        p2 = new Panel();
        p3 = new Panel();
        lb1 = new Label("Primo pannello rosso");
        p1.setBackground(Color.red);
        lb2 = new Label("Secondo pannello verde");
        p2.setBackground(Color.green);
        lb3 = new Label("Terzo pannello blue");
        p3.setBackground(Color.blue);
    }

    public void setup() {
        f.setLayout(cardLayout);
        p1.add(lb1);
        p2.add(lb2);
        p3.add(lb3);
        f.add(p1, "uno");
        f.add(p2, "due");
        f.add(p3, "tre");
        cardLayout.show(f, "uno");
        f.setSize(200,200);
        f.setVisible(true);
```

```
    }

    private void slideShow() {
        while (true) {
            try {
                Thread.sleep(3000);
                cardLayout.next(f);
            }
            catch (InterruptedException exc) {
                exc.printStackTrace();
            }
        }
    }

    public static void main (String args[]) {
        CardTest cardTest = new CardTest();
        cardTest.setup();
        cardTest.slideShow();
    }
}
```

In pratica tre pannelli vengono aggiunti sfruttando un `CardLayout`, e ad ogni panel viene assegnato un alias ("`uno`", "`due`", e "`tre`"). Viene settato il primo pannello da visualizzare con l'istruzione:

```
cardLayout.show(f, "uno");
```

e dopo aver visualizzato la GUI, viene invocato il metodo `slideShow()` che tramite il metodo `next()`, mostra con intervalli di tre secondi i vari panel. Per far questo viene utilizzato il metodo `sleep()` della class `Thread`, già incontrato nel modulo dedicato ai thread.

Figura 15.10 – "Il card layout che alterna i tre pannelli"

Solitamente l'alternanza di pannelli realizzata nel precedente esempio, non viene governata da un thread in maniera temporale. Piuttosto sembra sempre più evidente che occorre un modo per interagire con le GUI. Questo "modo di interagire" è descritto nella prossima unità didattica.

15.4 Gestione degli eventi

Per gestione degli eventi, intendiamo la possibilità di associare l'esecuzione di una certa parte di codice, in corrispondenza ad un certo evento sulla GUI. Un esempio di evento potrebbe essere la pressione di un bottone.
Nel modulo 8, quando sono state introdotte le classi innestate e le classi anonime, si è data anche una descrizione della storia di come si è arrivati alla definizione della gestione degli eventi in Java. Si parla di "modello a delega", e si tratta di un'implementazione nativa del pattern GoF noto come "Observer".

15.4.1 Observer e Listener

Anche se il pattern si chiama Observer (osservatore), in questo modulo parleremo soprattutto di Listener (ascoltatore). Il concetto è lo stesso e sembra che il nome sia diverso, perché quando è stato creato il nuovo modello a delega nella versione 1.1 di Java, gia esisteva una classe `Observer` (che serviva proprio per implementare "a mano" il pattern). Con il modello a delega, esistono almeno tre oggetti per gestire gli eventi su una GUI:

1) il componente sorgente dell'evento (in inglese "event source")

2) l'evento stesso
3) il gestore dell'evento, detto "listener".

Per esempio se premiamo su di un bottone e vogliamo che appaia una scritta su una `Label` della stessa interfaccia, allora:

1) il bottone è la sorgente dell'evento
2) l'evento è la pressione del bottone, che sarà un oggetto istanziato direttamente dalla JVM dalla classe `ActionEvent`
3) il gestore dell'evento sarà un oggetto istanziato da una classe a parte che implementa un interfaccia `ActionListener` (in italiano "ascoltatore d'azioni"). Quest'ultima ridefinirà il metodo `actionPerformed(ActionEvent ev)`, con il quale sarà gestito l'evento. Infatti la JVM, invocherà automaticamente questo metodo su quest'oggetto, quando l'utente premerà il bottone.

Ovviamente bisognerà anche effettuare un'operazione supplementare: "registrare" il bottone con il suo ascoltatore. È necessario infatti avvertire la JVM, su quale oggetto invocare il metodo di gestione dell'evento.
Ma vediamo in dettaglio come questo sia possibile con un esempio:

```
import java.awt.*;
public class DelegationModel {
    private Frame f;
    private Button b;

    public DelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
    }

    public void setup() {
        b.addActionListener(new ButtonHandler());
        f.add(b,BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
```

```
    public static void main(String args[]) {
        DelegationModel delegationModel = new
          DelegationModel();
        delegationModel.setup();
    }

}
```

L'unica istruzione che ha bisogno di essere commentata è :

```
b.addActionListener(new ButtonHandler());
```

trattasi della "registrazione" tra il bottone ed il suo gestore. Dopo tale istruzione la JVM sa su quale oggetto di tipo Listener chiamare il metodo `actionPerformed()`. Ovviamente il metodo `addActionListener()`, si aspetta come parametro un oggetto di tipo `ActionListener`. Essendo `ActionListener` un'interfaccia, questo significa che si aspetta un oggetto istanziato da una classe che implementa tale interfaccia.
La classe di cui stiamo parlando e che gestisce l'evento (il listener) è la seguente:

```
import java.awt.event.*;
public class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("È stato premuto il bottone");
        System.out.println("E la sua etichetta è: "
        + e.getActionCommand());
    }
}
```

in pratica ogni volta che viene premuto il bottone viene stampato sulla riga di comando l'etichetta ("`Press Me!`") del bottone, mediante il metodo `getActionCommand()`. Vista così non sembra una grossa impresa gestire gli eventi. Basta:

1) creare la GUI
2) creare un listener
3) registrare il componente interessato con il rispettivo listener.

Al resto ci pensa la JVM. Infatti, alla pressione del bottone viene istanziato un oggetto di tipo `ActionEvent` (che viene riempito di informazioni riguardanti l'evento), e passato in input al metodo `actionPerformed()` dell'oggetto listener associato. Un meccanismo che ricorda da vicino quello già studiato delle eccezioni. In quel caso, l'evento era l'eccezione (anch'essa veniva riempita di informazioni su ciò che era avvenuto), ed al posto del metodo `actionPerformed()`, c'era un blocco `catch`.
In realtà nell'esempio appena visto c'è una enorme e vistosa semplificazione. La scritta, piuttosto che venire stampata sulla stessa GUI, viene stampata sulla riga di comando. Qualcosa di veramente originale…creare un'interfaccia grafica per stampare sulla prompt dei comandi…
Ma cosa dobbiamo fare se vogliamo stampare la frase in una label della stessa GUI? Come può procurarsi i "reference giusti" la classe `ButtonHandler`? Proviamo a fare un esempio.
Stampiamo la frase su un oggetto `Label` della stessa GUI:

```
import java.awt.*;
public class TrueDelegationModel {
    private Frame f;
    private Button b;
    private Label l;

    public TrueDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(new TrueButtonHandler(l));
        f.add(b,BorderLayout.CENTER);
        f.add(l,BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String args[]) {
        TrueDelegationModel delegationModel = new
          TrueDelegationModel();
```

```
        delegationModel.setup();
    }
}
```

Notiamo come ci sia un cambiamento notevole: quando viene istanziato l'oggetto listener `TrueButtonHandler`, viene passato al costruttore la label.

```
import java.awt.event.*;
import java.awt.*;
public class TrueButtonHandler implements ActionListener
{
    private Label l;
    private int counter;

    public TrueButtonHandler(Label l) {
        this.l = l;
    }

    public void actionPerformed(ActionEvent e) {
        l.setText(e.getActionCommand() + " - " +
          (++counter));
    }
}
```

Come è facile osservare il codice si è notevolmente complicato. La variabile `counter` è stata utilizzata per rendere evidente l'evento di pressione sul bottone.

Questo tipo di approccio ovviamente può scoraggiare lo sviluppatore. Troppo codice per fare qualcosa di semplice, e tutta colpa dell'incapsulamento! Ma come affermato precedentemente in questo testo (cfr. Modulo 8), quando si programmano le GUI, si possono prendere delle profonde licenze rispetto all'object orientation. Non è un caso infatti, che le classi innestate siano nate insieme al modello a delega (versione 1.1 di Java), mentre le classi anonime ancora più tardi (versione 1.2).

15.4.2 Classi innestate e classi anonime

Nel modulo 8 sono stati introdotti due argomenti, le classi innestate e le classi anonime, ed in questo modulo probabilmente ne apprezzeremo di più l'utilità. Ricordiamo brevemente che una classe innestata è definita come una classe definita all'interno di un'altra classe. Per quanto riguarda la gestione degli eventi, l'implementazione del gestore dell'evento tramite una classe innestata rappresenta una soluzione molto vantaggiosa. Segue il codice dell'esempio precedente rivisitato con una classe innestata:

```
import java.awt.*;
import java.awt.event.*;
public class InnerDelegationModel {
    private Frame f;
    private Button b;
    private Label l;

    public InnerDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(new InnerButtonHandler(l));
        f.add(b,BorderLayout.CENTER);
        f.add(l,BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }

    public class InnerButtonHandler implements
      ActionListener {
        private int counter;
        public void actionPerformed(ActionEvent e) {
            l.setText(e.getActionCommand() + " - " +
            (++counter));
        }
    }
    public static void main(String args[]) {
        InnerDelegationModel delegationModel = new
```

```
            InnerDelegationModel();
        delegationModel.setup();
    }
}
```

Si può notare come la proprietà delle classi anonime di vedere le variabili della classe esterna come se fosse pubblica abbia semplificato il codice. Infatti, nella classe `InnerButtonHandler`, non è più presente il reference alla `Label l`, nè il costruttore che serviva per settarla, visto che è disponibile direttamente il reference "originale".
Una soluzione ancora più potente è rappresentata dall'utilizzo di una classe anonima per implementare il gestore dell'evento:

```
import java.awt.*;
import java.awt.event.*;
public class AnonymousDelegationModel {
    private Frame f;
    private Button b;
    private Label l;

    public AnonymousDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(    new ActionListener() {
            private int counter;
            public void actionPerformed(ActionEvent e) {
                l.setText(e.getActionCommand() + " - "
                  +(++counter));
            }
        } );
        f.add(b,BorderLayout.CENTER);
        f.add(l,BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
```

```
    }

    public static void main(String args[]) {
        AnomynousDelegationModel delegationModel = new
        AnomynousDelegationModel();
        delegationModel.setup();
    }
}
```

Come si può notare, la classe anonima ha una sintassi sicuramente più compatta. La sintassi delle classe anonima va fuori dai soliti standard (cfr. Modulo 8), ma quando ci si abitua è poi difficile rinunciarvi. Inoltre, rispetto ad una classe innestata, una classe anonima è sempre un singleton. Infatti la sua sintassi obbliga ad istanziare una ed una sola istanza, il che è una soluzione ottimale per un gestore degli eventi. Una buona programmazione ad oggetti richiederebbe che ogni evento abbia il suo "gestore personale".

Una limitazione che hanno le classi anonime è quella di non poter avere un costruttore (il costruttore ha lo stesso nome della classe). È possibile però utilizzare un inizializzatore d'istanza (cfr. Modulo 9) per inizializzare una classe anonima. Ovviamente, non si possono passare parametri ad un inizializzatore…Per un esempio d'utilizzo reale di un inizializzatore d'istanza, si possono studiare le classi anonime del file EJE.java (file sorgenti di EJE scaricabili agli indirizzi: http:// sourceforge.net/projects/eje/, http://eje.sourceforge.net o http://www.claudiodesio.com/eje.htm). In particolare le classi che rappresentano le azioni ed estendono la classe `AbstractAction` (che poi implementa `ActionListener`, cfr. documentazione di AbstractAction).

15.4.3 Altri tipi di eventi

Come già accennato precedentemente, esistono vari tipi di eventi che possono essere generati dai componenti e dai container di una GUI. Per esempio si potrebbe gestire l'evento di una pressione di un tasto della tastiera, il rilascio del bottone del mouse, l'acquisizione del focus da parte di un certo componente e soprattutto la chiusura della finestra principale. Esiste nella libreria una gerarchia di classi di tipo evento che viene riportata sommariamente nella figura 15.11.

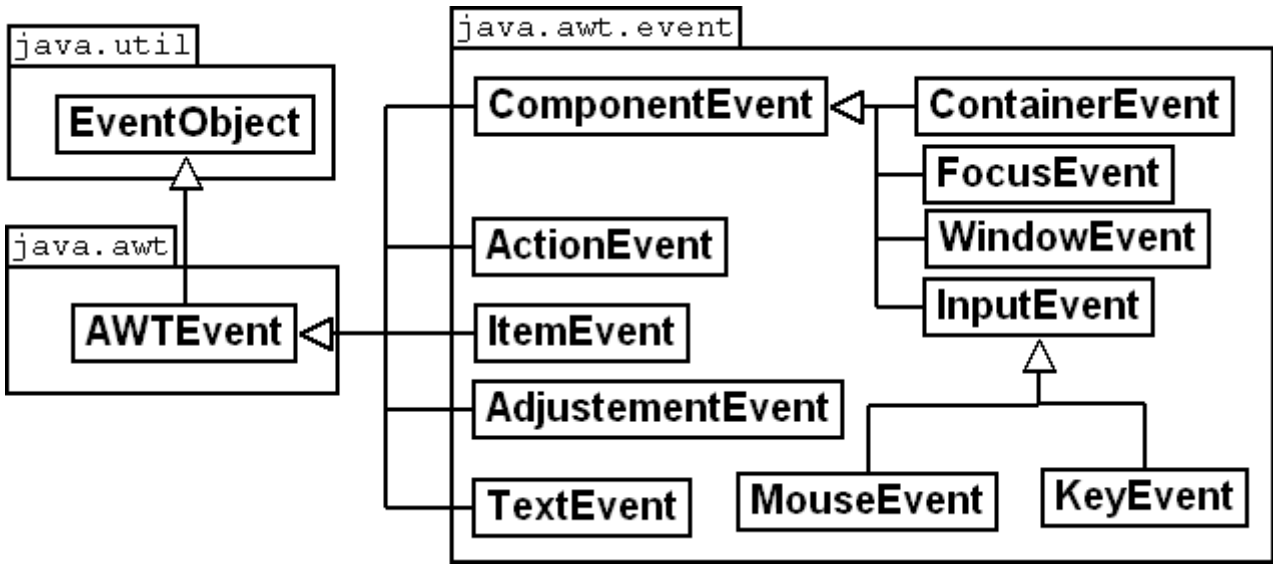


Figura 15.11 – "La gerarchia delle classi evento"

Nella seguente tabella invece, viene riportato una schematizzazione del tipo di evento, una veloce descrizione, l'interfaccia per la sua gestione e i metodi che sono dichiarati da essa. Ovviamente ci limiteremo solo agli eventi più importanti.

Evento	Descrizione	Interfaccia	Metodi
`ActionEvent`	Azione (generica)	`ActionListener`	`actionPerformed`
`ItemEvent`	Selezione	`ItemListener`	`itemStateChanged`

`MouseEvent`	Azioni effettuate con il mouse	`MouseListener`	`mousePressed`
			`mouseReleased`
			`mouseEntered`
			`mouseExited`
			`mouseClicked`

`MouseEvent`	Movimenti del mouse	`MouseMotionListener`	`mouseDragged` `mouseMoved`

`KeyEvent`	Pressione di tasti	`KeyListener`	`keyPressed`
			`keyReleased`
			`keyTyped`

`WindowEvent`	Azioni effettuate su finestre	`WindowListener`	`windowClosing`
			`windowOpened`
			`windowIconified`
			`windowDeiconified`

			`windowClosed`
			`windowActivated`
			`windowDeactivated`

Ovviamente, non tutti i componenti possono generare tutte le tipologie di eventi. Per esempio un bottone non può generare un `WindowEvent`.
Come primo esempio vediamo finalmente come chiudere un'applicazione grafica, semplicemente chiudendo il frame principale. Supponiamo che `frame`, sia il reference del frame principale, il seguente frammento di codice, implementa una classe anonima che definendo il metodo `windowClosing()` (cfr. documentazione), permette all'applicazione di terminare l'applicazione:

```
frame.addWindowListener( new WindowListener() {

    public void windowClosing (WindowEvent ev) {
        System.exit(0);
    }

    public void windowClosed (WindowEvent ev) {}

    public void windowOpened (WindowEvent ev) {}

    public void windowActivated (WindowEvent ev) {}

    public void windowDeactivated (WindowEvent ev) {}

    public void windowIconified (WindowEvent ev) {}

    public void windowDeiconified (WindowEvent ev) {}

} );
```

purtroppo, implementando l'interfaccia `WindowListener`, abbiamo ereditato ben sette metodi. Anche se in realtà ne abbiamo utilizzato uno solo, abbiamo comunque dovuto riscriverli tutti!
Questo problema evidente, è mitigato in parte dall'esistenza delle classi dette "Adapter". Un adapter è una classe che implementa un listener, riscrivendo ogni metodo ereditato

senza codice applicativo. Per esempio la classe `WindowAdapter` è implementata più o meno come segue:

```
public abstract class WindowAdapter implements
WindowListener {

    public void windowClosing (WindowEvent ev) {}

    public void windowClosed (WindowEvent ev) {}

    public void windowOpened (WindowEvent ev) {}

    public void windowActivated (WindowEvent ev) {}

    public void windowDeactivated (WindowEvent ev) {}

    public void windowIconified (WindowEvent ev) {}

    public void windowDeiconified (WindowEvent ev) {}
}
```

Quindi se invece di implementare l'interfaccia listener si estende la classe adapter, non abbiamo più bisogno di riscrivere tutti i metodi ereditati, ma solo quelli che ci interessano. Quindi la nostra classe anonima diventerà molto più compatta:

```
frame.addWindowListener( new WindowAdapter() {
    public void windowClosing (WindowEvent ev) {
        System.exit(0);
    }
} );
```

L'estensione di un adapter rispetto alla implementazione di un listener, è un vantaggio solo per il numero di righe da scrivere. Infatti, comunque il nostro gestore di eventi eredita i "metodi vuoti", il che non rappresenta una soluzione object oriented molto corretta.

Non sempre è possibile sostituire l'estensione di un adapter all'implementazione di un listener. Infatti, un adapter è comunque una classe, quindi impedirebbe l'estensione di

eventuali altre classi. Come vedremo nella prossima unità didattica per esempio, per un'applet non è possibile utilizzare un adapter, dato che per definizione si deve già estendere la classe `Applet`. Ricordiamo che è invece possibile implementare più interfacce e quindi qualsiasi numero di listener. Per esempio se dovessimo creare un classe che gestisca sia la chiusura della finestra che la pressione di un certo bottone, si potrebbe utilizzare una soluzione mista adapter-listener come la seguente:

```
public class MixHandler extends WindowAdapter implements
ActionListener {
    public void actionPerformed(ActionEvent e){…}
    public void windowClosing(WindowEvent e){…}
}
```

Ultime raccomandazione per evitare errori che chi vi scrive ha visto commettere molte volte.

1) `Listener` si scrive con le `e` tra la `t` e la `n`. Purtroppo, spesso noi italiani scriviamo i termini inglesi non molto noti, così come li pronunciamo.
2) Se avete scritto del codice ma l'evento non viene minimamente gestito, allora come primo tentativo di correzione, controllate se avete registrato il componente con il relativo gestore. Spesso infatti, presi dalla logica della gestione, ci si dimentica del passaggio della registrazione.
3) `WindowEvent`, `WindowListener` e tutti i suoi metodi (`windowClosing()`, `windowActivated()` etc…) si riferiscono al concetto di finestra (in inglese "window"), e non al sistema operativo più famoso! Quindi attenzione a non cadere nell'abitudine di scrivere `windowsClosing()` al posto di `windowClosing()`, perché vi potrebbe portar via molto tempo in debug. Se per esempio state utilizzando un `WindowAdapter` e sbagliate l'override di uno dei suoi metodi come appena descritto, il metodo che avete scritto, semplicemente non verrà mai chiamato. Verrà invece chiamato il metodo vuoto della superclasse adapter che ovviamente non farà niente, neanche avvertirvi del problema.

15.5 La classe Applet

Si tratta di una tecnologia che ha dato grande impulso e pubblicità a Java nei primi tempi, ed ancora oggi molto utilizzata sulle pagine web (cfr. introduzione al Mod.1).

In inglese applet potrebbe essere tradotta come "applicazioncina". Questo perché un'applet deve essere un'applicazione leggera dovendo essere scaricata dal web insieme alle pagine HTML. Pensiamo anche al fatto che nel 1995, anno di nascita di Java ma anche delle applet, non esistevano connessioni a banda larga. Bisogna tener conto anche che oltre alla dimensione, un'applet deve anche subire il caricamento, i controlli di sicurezza e l'interpretazione della JVM. Quindi è bene che sia "piccola".
In queste pagine si parlerà delle applet "al femminile". Questo perché noi traduciamo applet come "applicazioncina". Altri autori preferiscono parlare di applet al maschile, ed avranno le loro ragioni…

Finalmente potremo lanciare le nostre applicazioni direttamente da pagine web. In pratica un'applet è un applicazione Java, che può essere direttamente linkata ad una pagina HTML, mediante un tag speciale: il tag "`<APPLET>`".
Un'applet per definizione deve estendere la classe `Applet` del package `java.applet`. Ne eredita i metodi e può ovverridarli. L'applet non ha infatti un metodo `main()`, ma i metodi ereditati sono invocati direttamente dalla JVM del browser, con una certa filosofia. Quindi se il programmatore riscrive i metodi opportunamente riuscirà a far eseguire all'applet il codice che vuole. Segue una prima banale applet con dei commenti esplicativi:

```
import java.applet.*;
import java.awt.*;
public class BasicApplet extends Applet {

    public void init() {
    // Chiamato una sola volta dal browser appena viene
    // eseguita l'applet
    }

    public void start() {
    // Chiamato ogni volta che la pagina che contiene
    // l'applet diviene visibile
    }

    public void stop() {
    // Chiamato ogni volta che la pagina che contiene
```

```
    // l'applet deve essere disegnata
    }

    public void destroy() {
    // Chiamato una sola volta dal browser quando viene
    // distrutta l'applet
    }

    public void paint(Graphics g) {
    // Chiamato ogni volta che la pagina che contiene
    // l'applet diviene non visibile
    }
}
```

Tenendo conto dei commenti, il programmatore deve gestire l'esecuzione dell'applet. Non è obbligatorio scrivere tutti i cinque metodi, ma almeno uno si dovrebbe overridare. Per esempio la seguente applet stampa una frase:

```
import java.applet.*;
import java.awt.*;
public class StringApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Applet", 10, 10);
    }
}
```

In pratica l'oggetto `Graphics`, come accennato precedentemente, mette a disposizione molti metodi per il disegno. Per poter eseguire un'applet però, bisogna anche creare una pagina HTML che inglobi l'applet. Basterà creare un semplice file con suffisso ".htm" o ".html", che contenga il seguente testo.

```
<applet code='StringApplet' width='100' height='100'>
</applet>
```

Ovviamente al variare degli attributi `width` ed `height`, varierà la dimensione dell'area che la pagina HTML dedicherà all'applet.
L'HTML (HyperText Markup Language) è il linguaggio standard per la formattazione delle pagine WEB. Non è un linguaggio di programmazione e le sue regole (come la sua

robustezza e la sua portabilità) sono alquanto limitate. Rimandiamo il lettore allo studio delle regole base dell'HTML ad una delle migliaia di fonti che si trovano in Internet. Una piccola introduzione all'HTML allo scopo di dare la terminologia di base, si può trovare nella appendice E di questo manuale.
È anche possibile passare dalla pagina HTML, parametri che verranno letti al runtime dall'applet sfruttando il meccanismo esplicitato nel seguente esempio. Aggiungendo il seguente override del metodo `init()` all'esempio precedente, è possibile parametrizzare per esempio la frase da stampare:

```
import java.applet.*;
import java.awt.*;
public class ParameterApplet extends Applet {
    String s;

    public void init() {
        String parameterName = "p";
        s = getParameter(parameterName);
    }

    public void paint(Graphics g) {
        g.drawString(s, 10, 10);
    }
}
```

Ovviamente il codice della pagina HTML che deve caricare l'applet precedente cambierà leggermente.

```
<applet code='ParameterApplet' width='100'
  height='100'>
<param name='p' value='Java'>
</applet>
```

Per approfondire l'argomento rimandiamo il lettore interessato al Java Tutorial della Sun (cfr. bibliografia), o alle migliaia di esempi disponibili in rete.

15.6 Introduzione a Swing

Swing è il nome della libreria grafica di seconda generazione di Java.

Se il lettore cercherà una panoramica sui componenti più importanti di Swing in questo paragrafo non la troverà. Infatti, per quanto si possa essere precisi nella descrizione di un componente Swing, la documentazione rappresenterà comunque la guida migliore per lo sviluppatore. La complessità di questa libreria dovrebbe inoltre obbligare il programmatore ad utilizzare la documentazione. Per esempio, alcuni componenti di Swing come le tabelle o le liste (classi `JTable` e `JList`), utilizzano una sorta di pattern MVC per separare i dati dalla logica di accesso ad essi.

Swing fa parte di un gruppo di librerie note come Java Foundation Classes (JFC). Le JFC, ovviamente oltre a Swing, includono:

1) **Java 2D**: una libreria che permette agli sviluppatori di incorporare grafici di alta qualità, testo, effetti speciali ed immagini all'interno di applet ed applicazioni.
2) **Accessibility**: una libreria che permette a strumenti diversi dai soliti monitor (per esempio schermi Braille) di accedere alle informazioni sulle GUI
3) **Supporto al Drag and Drop** (DnD): una serie di classi che permettono di gestire il trascinamento dei componenti grafici.
4) **Supporto al Pluggable Look and Feel**: offre la possibilità di cambiare lo stile delle GUI che utilizzano Swing, e più avanti ne vedremo un esempio…

I componenti AWT sono stati forniti già dalla prima versione di Java (JDK 1.0), mentre Swing è stata inglobata come libreria ufficiale solo dalla versione 1.2 in poi. La Sun raccomanda fortemente l'utilizzo di Swing piuttosto che di AWT nelle applicazioni Java. Swing ha infatti molti "pro" ed un solo "contro" rispetto ad AWT. Essendo quest'ultimo abbastanza importante, introdurremo questa libreria partendo proprio da questo argomento.

15.6.1 Swing vs AWT

A differenza di AWT, Swing non fa chiamate native al sistema operativo dirette per sfruttarne i componenti grafici, bensì li ricostruisce da zero. Ovviamente, questa caratteristica di Swing, rende AWT nettamente più performante. Swing, è di sicuro la causa principale per cui Java gode della fama di "linguaggio lento". Esempi di applicazioni che utilizzano questa libreria sono proprio i più famosi strumenti di sviluppo come JBuilder, NetBeans o Together. Per esempio, lanciando JBuilder, passeranno diversi secondi, (o addirittura minuti se non si dispone di una macchina con

risorse sufficienti) per poter finalmente vedere la GUI. Una volta caricata però, l'applicazione gira ad una velocità sufficientemente performante (ovviamente sempre dando per scontato di avere una macchina al passo con i tempi). Questo perché il problema principale risiede proprio nel caricamento dei componenti della GUI, e bisogna tenere conto che una GUI complessa potrebbe essere costituita da centinaia di componenti. Una volta caricati tali componenti, le azioni su di essi non richiedono lo stesso sforzo da parte del sistema. Per esempio, se l'apertura di un menu scritto in C++ (linguaggio considerato altamente performante) richiede un tempo nell'ordine di millesimi di secondo, la stessa azione effettuata su di un'equivalente menu scritto in Java (ovviamente a parità di macchina), potrebbe al massimo essere eseguita nell'ordine del decimo di secondo. Questa differenza è poco percettibile all'occhio umano.
Rimane comunque il problema di dover aspettare un po'di tempo prima di vedere le GUI Swing.

Come già asserito nel Modulo 1, il problema può essere risolto solamente con il tempo, ed i miglioramenti degli hardware e della Virtual Machine. Ma per chi ha visto (come noi) Java in azione già nel 1995 con gli hardware di allora, la situazione attuale è più che soddisfacente. D'altronde, la programmazione dei nostri giorni (anche con altri linguaggi compresi quelli di Dot-Net, e C++), tende a badare meno alle performance e sempre più alla robustezza del software… in questo senso Java ha precorso i tempi.

La performance con la Virtual Machine di Java 5 è stata chiaramente migliorata. Una delle caratteristiche più importanti della release 5, sembra essere proprio la performance. Questa, stando al performance tuning di alcuni siti importanti e credibili, pare migliorata in maniera notevole. Ma noi non ci siamo fidati, ed abbiamo lanciato la stessa applicazione (SwingSet2 di cui parleremo più avanti) con due diversi JDK: l'1.4.2, 1.5.0_06.I risultati sono stati i seguenti:

1) La versione 1.4.2 ha avviato completamente l'applicazione in 11.1 secondi
2) La versione 1.5.0_06 ha avviato completamente l'applicazione in 10.3 secondi

Insomma andiamo nella direzione giusta…

Esiste anche una terza libreria grafica non ufficiale che fu sviluppata originariamente da IBM e poi donata al mondo Open Source: la SWT. È una libreria sviluppata in C++ altamente performante e dallo stile

piacevole, che bisogna inglobare nelle nostre applicazioni. Essendo scritta in C++, esistono versioni diverse per sistemi operativi diversi. Un esempio di utilizzo di SWT, è l'interfaccia grafica del più importante strumento di sviluppo Open Source: Eclipse (http://www.eclipse.org).

Anche EJE utilizza Swing, ma a differenza di JBuilder per esempio, ha una GUI estremamente più semplice e leggera.
Il fatto che Swing non utilizzi codice nativo però, porta anche diversi vantaggi rispetto ad AWT. Abbiamo già asserito come AWT possa definire solo un minimo comune denominatore dei componenti grafici presenti sui vari sistemi operativi. Questa non è più una limitazione per Swing. Swing definisce qualsiasi tipo di componente di qualsiasi sistema operativo, ed addirittura ne inventa alcuni nuovi! Questo significa che sarà possibile per esempio vedere sul sistema operativo Solaris il componente tree (il famoso albero di "esplora risorse") di Windows.
Inoltre, ogni componente di Swing estende la classe `Container` di AWT, e quindi può contenere altri componenti. Quindi ci sono tante limitazioni di AWT che vengono superate. Per esempio:

- I bottoni e le label di Swing possono visualizzare anche immagini oltre che semplice testo.
- Grazie al supporto al pluggable look and feel, è possibile vedere GUI con stili di diversi sistemi operativi, sullo stesso sistema operativo. Per esempio, EJE su di un sistema operativo Windows XP, permetterà di scegliere nelle opzioni (premere F12 e fare clic sul Tab "EJE") lo stile tra Metal (uno stile personalizzato di Java), stile Windows, o stile CDE/Motif (stile familiare agli utenti Unix). Con AWT siamo costretti ad utilizzare lo stile della piattaforma nativa.
- È possibile facilmente cambiare l'aspetto o il comportamento di un componente Swing o invocando metodi o creando sottoclassi.
- I componenti Swing non devono per forza essere rettangolari. I `Button`, possono per esempio essere circolari.
- Con il supporto della libreria Accessibility è semplice per esempio leggere con uno strumento come uno schermo braille l'etichetta di una label o di un bottone.
- È facile cambiare anche i bordi dei componenti, con una serie di bordi predefiniti o personalizzati.
- È anche molto semplice utilizzare i tooltip (i messaggi descrittivi che appaiono quando si posiziona il puntatore su un componente) sui componenti Swing

mediante il metodo `setToolTip()`, e gestire il controllo delle azioni direttamente da tastiera.

Le classi di Swing quindi, sono molto più complicate di quelle di AWT, e permettono di creare interfacce grafiche senza nessun limite di fantasia. Rimane il limite delle prestazioni che però, grazie ai progressi degli hardware, sarà nel tempo sempre meno un problema.

La Sun raccomanda di non mixare all'interno della stessa GUI, componenti Swing con componenti "heavyweight" ("pesanti") di AWT. Per componenti pesanti si intendono tutti i componenti "pronti all'uso" come `Menu` e `ScrollPane`, e tutti i componenti AWT che estendono le classi `Canvas` e `Panel`. Questa restrizione esiste perchè quando si sovrappongono i componenti Swing (e tutti gli altri componenti "lightweight") ai componenti pesanti, questi ultimi vengono sempre disegnati sopra.

I componenti Swing non sono "thread safe". Infatti, se si modifica un componente Swing visibile per esempio invocando su di una label il metodo `setText()`, da una qualsiasi parte di codice eccetto un gestore di eventi, allora bisogna prendere alcuni accorgimenti per rendere visibile la modifica. In realtà questo problema non si presenta spesso perchè nella maggior parte dei casi sono proprio i gestori degli eventi ad implementare il codice per modificare i componenti.
Le classi di Swing si distinguono da quelle di AWT principalmente perchè i loro nomi iniziano con una " J". Per esempio, la classe di Swing equivalente alla classe `Button` di AWT si chiama `JButton`. Ovviamente, il package di riferimento non è più `java.awt` ma `javax.swing`.

Notare che, a differenza delle altre librerie finora incontrate (eccetto JAXP), il package principale non si chiama "`java`" ma "`javax`". La "`x`" finale sta per "eXtension" (gli americani fanno così le abbreviazioni…e noi ci adeguiamo…) perché inizialmente (JDK 1.0 e 1.1) Swing era solo un'estensione della libreria ufficiale.

Ci sono altre differenze che possono far perdere un po' di tempo per chi è abituato ad AWT. Per esempio, per aggiungere un componente ad un `JFrame`, si deve utilizzare il metodo `add()`, non direttamente sul `JFrame`, ma su di un `Container` interno al

`JFrame` che si chiama Content Pane (cfr. documentazione per i dettagli). Quindi se vi eravate abituati con AWT ad aggiungere il panel `p` al frame `f` nel modo seguente:

```
f.add(p);
```

se avete `f` come `JFrame` e `p` come `JPanel` allora il codice precedente potrebbe essere sostituito in uno dei seguenti modi:

```
f.getContetPane().add(p);
```

o anche:

```
f.setContentPane(p);
```

In questo ultimo caso abbiamo sostituito il content pane con il nostro pannello. Stesso discorso si applica anche ad altri metodi che appartengono ora al "ContentPane" come il metodo `setLayout()`.

Dalla versione 5 in poi però è possibile utilizzare il metodo `add()` direttamente con il `JFrame`, come in AWT…

Inoltre, ci sono alcune situazioni in cui per ottenere un componente Swing equivalente a quello AWT non basterà aggiungere una `J` davanti al nome AWT. Per esempio la classe equivalente a `Choice` di AWT si chiama `JComboBox` in Swing. Oppure l'equivalente di `Checkbox` è `JCheckBox` con la "`B`" maiuscola…
Il miglior modo per rendersi conto delle incredibili potenzialità di Swing, è quello di dare un'occhiata all'applicazione `SwingSet2`, che trovate nella cartella "demo/jfc" del JDK. È possibile lanciare l'applicazione o come applet (lanciando il file "SwingSet2.html"), o come applicazione con un doppio click sul file "SwingSet2.jar" su sistemi Windows. Se avete associato qualche altro programma ai file con suffisso ".jar", o non avete un sistema Windows, è sempre possibile lanciare l'applicazione nel modo tradizionale da riga di comando nel seguente modo:

```
java –jar SwingSet2.jar
```

Sono disponibili anche tutti i sorgenti, ed il loro studio potrebbe diventare molto fruttuoso.

15.6.2 File JAR eseguibile

I file JAR (Java ARchive) come abbiamo visto nel modulo 9, sono spesso utilizzati per creare librerie da utilizzare nei nostri programmi. C'è però un utilizzo meno noto dei file JAR, ma molto di effetto: la creazione di un file JAR eseguibile. Per fare questo bisogna semplicemente scrivere nel file "Manifest.mf", una riga che permetta alla virtual machine di capire qual è la classe che definisce il metodo `main()`. Per esempio il file manifest compreso nel file JAR di SwingSet2 è il seguente:

```
Manifest-Version: 1.0
Created-By: 1.5.0 (Sun Microsystems Inc.)
Main-Class: SwingSet2
```

con l'ultima riga si istruisce la JVM, in modo tale che possa avviare la nostra applicazione.
Su di un sistema Windows quindi, un doppio click sul file JAR, avvierà l'applicazione come se fosse un normale eseguibile.

> **È possibile che abbiate installato sul vostro sistema windows un programma come Winrar, associato all'apertura del file JAR. In tal caso dovete associare al comando "javaw" (compreso nella cartella "bin" del JDK) l'apertura del file JAR, per poter sfruttare questa utilità.**

Riepilogo

In questo modulo abbiamo introdotto i principi per creare GUI al passo con i tempi, ed in particolare abbiamo sottolineato l'importanza del pattern MVC.
Abbiamo in seguito descritto la libreria AWT, sia elencando le sue caratteristiche principali, sia con una panoramica su alcuni dei suoi principali componenti. In particolare abbiamo sottolineato come tale libreria sia fondata sul pattern Composite, sui ruoli di Component e Container, senza però soffermarci troppo sui dettagli.
La gestione del posizionamento dei componenti sul container, è basata sul concetto di layout manager, di cui abbiamo introdotto i più importanti rappresentanti.
La gestione degli eventi invece, è basata sul modello a delega, a integrazione del quale abbiamo introdotto diversi concetti quali adapter, classi innestate e anonime.

Non poteva mancare anche una introduzione alle applet, per essere operativi da subito, supportata anche da una piccola introduzione al linguaggio HTML.
Infine, abbiamo introdotto anche le principali caratteristiche della libreria Swing, per poterne apprezzare la potenza.

Esercizi modulo 15

Esercizio 15.a)
GUI, AWT e Layout Manager, Vero o Falso:

1. Nella progettazione di una GUI, è preferibile scegliere soluzioni standard, per facilitare l'utilizzo all'utente
2. Nell'MVC, il Model rappresenta i dati, il controller le operazioni, e la view l'interfaccia grafica
3. Le GUI AWT, sono invisibili di default
4. Per ridefinire l'aspetto grafico di un componente AWT, è possibile estenderlo e ridefinire il metodo `paint()`
5. AWT è basata sul pattern Decorator
6. In un'applicazione basata su AWT, è necessario sempre avere un top-level container
7. È impossibile creare GUI senza layout manager, otterremmo solo eccezioni al runtime
8. Il `FlowLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento
9. Il `BorderLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento
10. Il `GridLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento

Esercizio 15.b)
Gestione degli eventi, Applet e Swing, Vero o Falso:

1. Il modello a delega è basato sul pattern Observer
2. Senza la registrazione tra la sorgente dell'evento e il gestore dell'evento, l'evento non sarà gestito
3. Le classi innestate e le classi anonime non sono adatte per implementare gestori di eventi
4. Una classe innestata può gestire eventi se e solo se è statica
5. Una classe anonima per essere definita si deve per forza istanziare
6. Un `ActionListener` può gestire eventi di tipo `MouseListener`
7. Un bottone può chiudere un finestra

8. È possibile (ma non consigliabile) per un gestore di eventi estendere tanti adapter per evitare di scrivere troppo codice
9. La classe `Applet` estendendo `Panel`, potrebbe anche essere aggiunta direttamente ad un `Frame`. In tal caso però, i metodi overridati non verranno chiamati automaticamente
10. I componenti di Swing (`JComponent`) estendono la classe `Container` di AWT

Soluzioni esercizi modulo 15

Esercizio 15.a)
GUI, AWT e Layout Manager, Vero o Falso:

1. **Vero**
2. **Falso** in particolare il Model rappresenta l'intera applicazione composta da dati e funzionalità
3. **Vero**
4. **Vero**
5. **Falso** è basata sul pattern Composite, che, nonostante abbia alcuni punti di contatto con il Decorator, si può tranquillamente definire completamente diverso
6. **Vero**
7. **Falso** ma perderemmo la robustezza a la consistenza della GUI
8. **Vero**
9. **Falso**
10. **Falso**

Esercizio 15.b)
Gestione degli eventi, Applet e Swing, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso**
4. **Falso**
5. **Vero**
6. **Falso** solo di tipo `ActionListener`
7. **Vero** può sfruttare il metodo `System.exit(0)`, ma non c'entra niente con gli eventi di tipo `WindowEvent`
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper elencare le principali caratteristiche che deve avere una GUI (unità 15.1)	☐	
Saper descrivere le caratteristiche della libreria AWT (unità 15.2)	☐	
Saper gestire i principali Layout Manager per costruire GUI complesse (unità 15.3)	☐	
Saper gestire gli eventi con il modello a delega (unità 15.4)	☐	
Saper creare semplici applet (unità 15.5)	☐	
Saper descrivere le caratteristiche della libreria Swing (unità 15.6)	☐	

Note:

Parte V “Approfondimento sulle novità di Java 5”

La parte 5 è interamente dedicata alle rivoluzionarie caratteristiche della release 1.5 del linguaggio. Tiger infatti ha sconvolto completamente il linguaggio, mettendone in discussione anche caratteristiche oramai assodate come la semplicità. Tiger è stata la più importante tra le revisioni che ha subito Java, e a quanto pare lo rimarrà per sempre. Passare da un release ad un'altra è sempre stato agevole per il programmatore Java, ma il salto dal JDK 1.4 al JDK 1.5, è diverso. Nonostante oramai stiamo parlando di una versione stabile e consolidata, la maggior parte dei programmatori Java, non ha ancora compiuto la conversione, anzi tende ad evitarla. Ma Tiger non è stata pensata come il capolinea della programmazione Java, ma si tratta in realtà di una svolta. Acquisite le conoscenze necessarie come i cicli for migliorati, i generics, le enumerazioni e le annotazioni, lo sviluppatore entrerà in un nuovo mondo Java, molto più potente rispetto al precedente. Tutti gli argomenti che tratteremo in questa sezione, sono già stati introdotti precedentemente in questo testo. In quest’ultima parte del libro però, andremo a studiare a fondo le caratteristiche più importanti introdotte con la versione 5 di Java. È stata fatta la scelta di separare questa sezione dalle altre, per rendere più graduale l’apprendimento per chi inizia, e più semplice la consultazione a chi è già pratico.
Ogni unità didattica si occuperà di un’unica nuova feature. La struttura di ogni unità didattica sarà sempre suddivisa quindi in due parti. Nella prima parte sarà introdotta la feature, con osservazioni, spiegazioni e il supporto di numerosi esempi. Qui cercheremo anche di spiegare cosa tecnicamente realizza il compilatore Java per i nostri scopi. In questo modo, come consuetudine di questo testo, scendendo nei dettagli ci porremo nuovi quesiti e provvederemo a dare loro una risposta. Infine ogni unità didattica terminerà con un paragrafo dal titolo fisso “Impatto su Java”. Questo ha il compito di evidenziare le conseguenze dell’introduzione della feature, sul linguaggio. Ovviamente, il contenuto di questo paragrafo è direttamente correlato con il contenuto del paragrafo precedente.

Premessa alla Parte V

Introduzione a Tiger

Come affermato sin dal primo modulo, Java è un linguaggio in continua evoluzione. Questo obbliga gli utenti a continui aggiornamenti per non rischiare di rimanere indietro. Oramai siamo rassegnati all’idea, che dovremo sempre aggiornarci, non ci si può più

fermare. Solitamente però, i cambiamenti tra una nuova release ed un'altra, erano principalmente concentrate sulle librerie, sulle prestazioni del compilatore e della Virtual Machine. Nella storia di Java i cambiamenti che ricordiamo, che non riguardano librerie e prestazioni, sono pochi. Nella versione 1.1 l'introduzione delle classi interne, nella versione 1.2 l'introduzione delle classi anonime, nella versione 1.4 l'introduzione della parola chiave `assert` e poco altro. Nella release 1.5, Sun ha rivoluzionato il linguaggio con una serie di nuove caratteristiche. I cambiamenti sono talmente importanti che si è subito parlato di Java 5 invece che di Java 1.5 (d'altronde Sun non è nuova a questo tipo di "salti di release", vedi sistema operativo Solaris). La versione 5, ha anche un nome simbolico "Tiger" (in italiano "tigre"), dovuto ad esigenze di marketing.
Java è sempre stato pubblicizzato come un linguaggio che possiede tra le proprie caratteristiche la semplicità. Abbiamo precisato che tale caratteristica è relativa a paragoni con linguaggi di "pari livello" quali il C++. In questo testo, abbiamo spesso cercato di giustificare questa affermazione più o meno con successo. In effetti, il linguaggio in sé, a parte che per qualche concetto più avanzato come le classi anonime, mantiene una sintassi chiara, semplice e coerente. La difficoltà vera e propria semmai, consiste nel fatto che bisogna programmare ad oggetti per ottenere buoni risultati. Lo studio dell'object orientation non si è dimostrato privo di difficoltà. Ecco perché in molti considerano Java un linguaggio complesso.
Con la versione 1.5 del linguaggio Java, Sun ha scelto una strada molto precisa: mettere da parte la semplicità, per evolvere il linguaggio con nuove caratteristiche. Questo cambio di strategia, è stato probabilmente inevitabile, vista la concorrenza dei nuovi linguaggi di Microsoft, come VB.Net e C#. In pratica con Java 5 ci sono stati enormi cambiamenti che rendono necessario un nuovo e accurato studio di aggiornamento.
In realtà Sun, pubblicizzando Tiger, ha sostituito il termine "simplicity" ("semplicità"), con "ease of development" ("facilità di sviluppo"). Infatti, alcune delle caratteristiche di Java 5, permettono di scrivere meno codice, e di risolvere i bug già in fase di compilazione. Tuttavia è innegabile che questo costi in termini di semplicità.

Perché Java 5?

Non è detto che uno sviluppatore debba obbligatoriamente apprezzare le novità di Tiger. Complicare il linguaggio non è una mossa che tutti possono apprezzare. Anche i programmatori più esperti potrebbero storcere il naso, visto che il linguaggio sta perdendo la sua natura originaria. La release 5 di Java, ha però una caratteristica fondamentale che dovrebbe incoraggiarne lo studio: la compatibilità con le versioni precedenti. Infatti, anche se sono definite delle novità clamorose come un nuovo ciclo

`for`, e nuove keyword, queste saranno trasformate dal compilatore in istruzioni "vecchie", che saranno interpretabili anche da Virtual Machine meno aggiornate. Nonostante al momento esistono alcune incompatibilità tra versioni, queste sono alquanto trascurabili.
Inoltre, nessuno è obbligato a cambiare il proprio stile di programmazione se non vuole. Possiamo programmare alla veccia maniera con il Java Development Kit 1.5, ignorando le novità del linguaggio. Dovremo subire solo qualche warning, che però è anche possibile disabilitare. Prima o poi però, ci accorgeremo che possiamo programmare "meglio" sfruttando le caratteristiche di Tiger. Ed è meglio aggiornarsi prima che dopo… la strada è oramai segnata…anzi siamo in ritardo per Java 6!
Non ci resta che iniziare a studiare…

Utilizzare Java 5

Per utilizzare le nuove caratteristiche di Tiger, in primo luogo bisogna ovviamente avere a disposizione un Java Development Kit aggiornato almeno alla versione 1.5. Come al solito, se il lettore non ha già provveduto, è possibile effettuare il download direttamente dal sito della Sun (http://java.sun.com). Altra raccomandazione scontata è quella di scaricare anche la relativa documentazione.
Per compilare i nostri file con Java 5, è assolutamente indispensabile che il lettore consulti l'Appendice I di questo manuale, nel caso non l'abbia ancora fatto.

16

Complessità: alta

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere le semplificazioni che ci offre la nuova (doppia) feature autoboxing e auto-unboxing (unità 16.1).
2. Conoscere le conseguenze e i problemi che genera l'introduzione dell'autoboxinge dell'auto-unboxing nel linguaggio Java (unità 16.1).
3. Capire cos'è un tipo generic (unità 16.2).
4. Saper utilizzare i tipi generic (unità 16.2).
5. Aver presente l'impatto su Java dell'introduzione dei generics (unità 16.2)

16 Autoboxing, Auto-Unboxing e Generics

In questo modulo affronteremo una caratteristica semplice (autoboxing) ed una molto complicata (Generics).

16.1 Autoboxing e Auto-Unboxing

Questa nuova feature di Java 5, è sicuramente molto utile. Si tratta di una semplificazione evidente per la gestione delle cosiddette classi wrapper (cfr. Modulo 12) ed i relativi tipi primitivi. Infatti, è possibile trattare i tipi primitivi come se fossero oggetti e viceversa. Questo significa che non dovremo più ogni volta creare l'oggetto wrapper che contiene il relativo dato primitivo, per poi eventualmente riestrarre in un secondo momento il dato primitivo. Questa pratica era per esempio necessaria quando si volevano introdurre all'interno di una collection, tipi primitivi. Inoltre sarà possibile per esempio sommare due oggetti `Integer` o anche un `int` ed un `Integer` nel seguente modo:

```
int i = 1;
Integer integer = new Integer(2);
int somma = i + integer;
```

ma anche:

```
Integer somma = i + integer;
```

funziona!
Seguono una serie di altri esempi di codice Java 5 valido:

```
Integer i = 0;
Double d = 2.2;
char c = new Character('c');
```

In pratica un `int` ed un `Integer` sono equivalenti e così tutti i tipi primitivi con i rispettivi wrapper. Questo ovviamente favorisce la risoluzione di alcune criticità come l'inserimento di dati primitivi nelle collections… Prima di Java 5 per esempio il codice:

```
Vector v = new Vector();
v.add(1);
v.add(false);
v.add('c');
```

avrebbe provocato errori in compilazione. Infatti, non era possibile aggiungere tipi primitivi laddove ci si aspetta un tipo complesso. L'equivalente codice prima di Java 5 doveva essere il seguente:

```
Vector v = new Vector();
v.add(new Integer(1));
v.add(new Boolean(false));
v.add(new Character('c'));
```

Per poi dover recuperare i dati primitivi successivamente mediante il seguente codice:

```
Integer i = (Integer)v.elementAt(0);
Boolean b = (Boolean) v.elementAt(1);
Character c = (Character) v.elementAt(2);
int intero = i.intValue();
boolean booleano = b.booleanValue();
character c = c.charValue();
```

ma perchè fare tanta fatica?

In realtà il codice precedente darà luogo ad un warning in compilazione se non specifichiamo il flag "-source 1.4" (cfr. Appendice I).
Ora è possibile scrivere direttamente:

```
Vector v = new Vector();
v.add(1);
v.add(false);
v.add('c');
int intero = (Integer)v.elementAt(0);
boolean booleano = (Boolean) v.elementAt(1);
character c = (Character) v.elementAt(2);
```

e questo non può che semplificare la vita del programmatore.

Il termine "boxing", equivale al termine italiano "inscatolare". In pratica l'inscatolamento dei tipi primitivi nei relativi tipi wrapper con Java 5 è automatico (ci pensa il compilatore in realtà). Ecco perché si parla di "autoboxing". Ovviamente l'auto-unboxing è il processo inverso. Si tratta quindi di una doppia feature. Il compilatore non fa altro che inscatolare i tipi primitivi nei relativi tipi wrapper, o estrarre tipi primitivi dai tipi wrapper, quando ce n'è bisogno.
In pratica, l'autoboxing e l'auto-unboxing, sono delle comodità che ci fornisce il compilatore, che fa il lavoro per noi.

Le specifiche di Java 5, asseriscono che alcuni tipi di dati primitivi, vengono sempre inscatolati nelle stesse istanze wrapper immutabili. Tali istanze vengono poi poste in una speciale cache dalla Virtual Machine e riusate, perché ritenute di frequente utilizzo. Tutto questo ovviamente, al fine di migliorare le performance.
In pratica, godono di questa caratteristica:

1. tutti i tipi `byte`
2. i tipi `short` e `int` con valori compresi nell'ordine dei `byte` (da `-128` a `127`)
3. i tipi `char` con range compreso da `\u0000` a `\u007F` (cioè da `0` a `127`)
4. i tipi `boolean`

Vedremo presto che, benché questa osservazione sembri fine a se stessa, ha un'importante conseguenza di cui i nostri programmi devono tener conto.

Attenzione che è illegale per esempio la seguente istruzione:
```
Double d = 2;
```
infatti, il valore `2` di tipo `int`. L'autoboxing invece richiede che ci sia una coincidenza perfetta tra il tipo primitivo e il relativo tipo wrapper. Non importa che un `int` possa essere contenuto in un `double`. Il problema è facilmente risolvibile con il seguente cast (cfr. Modulo 3):
```
Double d = 2D;
```

16.1.1 Impatto su Java

Purtroppo, non sono sempre tutte rose e fiori. Una volta introdotta una nuova feature in un linguaggio, bisogna valutarne l'impatto su ciò che c'era in precedenza.

Assegnazione di un valore `null` al tipo wrapper

È sempre possibile assegnare il valore `null` ad un oggetto della classe wrapper. Ma bisogna tener presente che `null`, non è un valore valido per un tipo di dato primitivo. Quindi il seguente codice:

```
Boolean b = null;
boolean bb = b;
```

compila tranquillamente, ma lancia una `NullPointerException` al runtime.

Costrutti del linguaggio ed operatori relazionali

Tutti i costrutti del linguaggio (`if`, `for`, `while`, `do`, `switch` e operatore ternario), e tutti gli operatori di confronto (<,>, ==, `!`=, etc…), si basano su tipi booleani. Con Tiger ovviamente, potremo sostituire un tipo `boolean` con il rispettivo tipo wrapper `Boolean` senza problemi, anche in tutti i costrutti e operatori.

Il costrutto `switch` (cfr. Modulo 4) inoltre, si basava su una variabile di test che poteva essere solo di tipo `byte`, `short`, `int`, o `char`. Con Tiger tale variabile potrebbe anche essere di tipo `Byte`, `Short`, `Integer`, o `Character` (oltre che un'enum come vedremo nel modulo successivo).

L'operatore ==, è utilizzabile per confrontare sia tipi di dati primitivi, sia reference. In ogni caso va a confrontare i valori delle variabili coinvolte, che, nel caso di tipi

reference, sono indirizzi in memoria (cfr. Modulo 4). Questo implica che due oggetti che vengono confrontati con l'operatore ==, saranno uguali se e solo se risiedono allo stesso indirizzo, ovvero se sono lo stesso oggetto. L'introduzione della doppia feature autoboxing e auto-unboxing, ci obbliga ad alcune riflessioni. Visto che i tipi primitivi e i tipi wrapper sono equivalenti, che risultati darà l'operatore ==? Il risultato è quello che ci si aspetta sempre, le regole non sono cambiate. Ma c'è un'eccezione. Consideriamo il seguente codice:

```
public class Comparison {
    public static void main(String args[]) {
        Integer a = 1000;
        Integer b = 1000;
        System.out.println(a==b);
        Integer c = 100;
        Integer d = 100;
        System.out.println(c==d);
        int e = 1000;
        Integer f = 1000;
        System.out.println(e==f);
        int g = 100;
        Integer h = 100;
        System.out.println(g==h);
    }
}
```

L'output del precedente programma sarà:

```
false
true
true
true
```

In pratica, tranne che nella comparazione tra `c` e `d`, tutto funziona in maniera normale. Infatti `c` e `d`, sono due `Integer` con valori compresi nel range del tipo `byte` (`-128` a `127`), e quindi, come abbiamo asserito nel paragrafo precedente, vengono trattati in maniera speciale dalla JVM. Questi due `Integer` vengono inscatolati nelle stesse istanze wrapper immutabili, per essere memorizzate e riusate. Quindi i due `Integer`, risultano essere allo stesso indirizzo perché in realtà sono lo stesso oggetto.

Nonostante, l'ultima considerazione è da considerarsi indubbiamente un problema, se scriveremo del codice che utilizza il metodo `equals()` con i tipi complessi e l'operatore `==` con i tipi primitivi, questo problema non si presenterà mai. In fondo, questa dovrebbe essere la regola da seguire sempre.

Overload

L'overload (cfr. Modulo 6), è una caratteristica di Java molto semplice e potente. Grazie ad essa, è possibile far coesistere in una stessa classe, più metodi con lo stesso nome ma con differente lista di parametri. Precedentemente a Tiger, era molto semplice capire la chiamata ad un metodo oveloadato, quale metodo avrebbe realmente invocato. La lista dei parametri, non genera nessun dubbio. Ma consideriamo il seguente codice:

```
public void metodo(Integer i) { . . . }
public void metodo(float f) { . . . }
```

quale metodo verrà chiamato dalla seguente istruzione in Tiger?

```
metodo(123);
```

La risposta è semplice, lo stesso che veniva chiamato con le versioni precedenti (ovvero `metodo(float f)`. In questo modo, è stato evitata una scelta che avrebbe avuto conseguenze inaccettabili per gli sviluppatori.

16.2 Generics

Le novità di Java 5 sono molte, ma uno dei primi argomenti da trattare non può non essere questo. L'argomento "**Generics**" infatti, influenza in qualche modo anche altri argomenti quali: varargs, enumerazioni, collections, e qualcosa che riguarda la nuova utility dei thread. Quindi sconsigliamo vivamente al lettore di saltare questo paragrafo, anche se lo troverà abbastanza complicato…

Questa nuova feature aggiunge nuova robustezza a Java e lo rende un linguaggio ancora più fortemente tipizzato di quanto già non lo fosse in precedenza. Permetterà di creare collection (e non solo) che accettano solo determinati tipi di dati che si possono specificare con una nuova sintassi. In questo modo, eviteremo noiosi e pericolosi controlli e casting di oggetti.

Uno dei punti di forza di Java, è la sua chiarezza nel definire i tipi. La gerarchia delle classi è rigida e non ambigua: tutto è un oggetto. Questo, per esempio ci garantisce l'utilizzo di collezioni eterogenee (cfr. Modulo 6 e Modulo 12). Queste, se da un lato garantiscono grande flessibilità e potenza, da un altro creano qualche difficoltà allo sviluppatore. Per esempio, supponiamo di aver creato un `ArrayList` in cui vogliamo che siano inserite solo stringhe. È abbastanza noioso e poco performante, essere obbligati ad utilizzare un casting di oggetti, quando si estrae un oggetto di cui si sa già a priori il tipo. Per esempio, consideriamo il seguente codice:

```
ArrayList list = getListOfStrings();
for (int i = 0; i < list.size(); i++)  {
    String stringa = (String)list.get(i);
}
```

Se rimuovessimo il cast, otterremmo un errore in compilazione come il seguente:

```
. . .
incompatible types
found : java.lang.Object
required: java.lang.String
String stringa = list.get(i);
^
```

Per avere del codice robusto inoltre, dovremmo comunque garantirci a priori che siano inserite solo stringhe, magari con un controllo come il seguente:

```
if (input istanceof String) {
    list.add(input);
}
```

Senza un controllo come il precedente, potremmo andare incontro ad una delle più insidiose delle unchecked exception: la `ClassCastException`.
I generics ci permettono di dichiarare una lista specificando che essa accetterà solo stringhe con la seguente sintassi:

```
List<String> strings;
```

Questa nuova sintassi, dove appaiono per la prima volta le parentesi angolari, può disorientare in un primo momento. Non c'è scelta, in futuro faremo un uso massiccio di parentesi angolari. Inoltre, bisogna anche assegnare a `strings` un'istanza che accetti lo stesso tipo di elementi (stringhe).

```
List<String> strings = new ArrayList<String>();
```

A questo punto abbiamo una lista che accetta solo stringhe, e nessun altro tipo di oggetto. Per esempio il seguente codice:

```
List<String> strings = new ArrayList<String>( );
strings.add("è possibile aggiungere String");
```

compilerà tranquillamente mentre la seguente istruzione:

```
strings.add(new StringBuffer("è impossibile aggiungere" +
" StringBuffer"));
```

provocherà il seguente output di errore:

```
. . .
cannot find symbol
symbol : method add(java.lang.StringBuffer)
location: interface java.util.List<java.lang.String>
strings.add(new StringBuffer("è impossibile aggiungere" +
   " StringBuffer"));
^
```

Quindi, a differenza di prima, abbiamo ora tra le mani uno strumento per controllare le nostre collezioni con grande robustezza. I problemi che verranno evidenziati in fase di compilazione, eviteranno problemi ben più seri al runtime.
Chiaramente i tipi generici sono utilizzati anche come parametri sia di input che di output dei metodi. Segue un esempio di dichiarazione di un metodo che prende un tipo generic in input e ne restituisce un altro in output:

```
public List<String> getListOfMapValues (Map<Integer ,
String> map){
    List <String> list = new ArrayList <String>();
```

```
    for (int i = 0; i < map.size(); i++){
        list.add(map.get(i));
    }
    return list;
}
```

tutto come previsto, e nessun casting pericoloso.

16.2.1 Dietro le quinte

Se andiamo a dare uno sguardo alla documentazione ufficiale dalla versione 1.5 in poi, troveremo delle misteriose novità nelle dichiarazioni delle Collections (cfr. Modulo 12). Per esempio, l'interfaccia `List` è dichiarata nel seguente modo:

```
public interface List<E> extends Collection<E>,
Iterable<E>
```

È inutile cercare la classe `E`. Si tratta in realtà solo di una nuova terminologia, che sta a significare che `List` è un tipo "generic", ed `E` rappresenta un parametro. Questo implica che quando utilizziamo `List`, è possibile parametrizzare `E` con un tipo particolare, come abbiamo fatto nell'esempio (dovrebbe essere l'iniziale di "Element").
Anche alcuni metodi hanno cambiato la loro dichiarazione sfruttando parametri. Per esempio, il metodo `add()`, che nelle versioni precedenti alla versione 5 era dichiarato nel seguente modo:

```
public boolean add(Object obj);
```

attualmente è stato rivisitato per gestire i generics, ed ora è dichiarato nel seguente modo:

```
public boolean add(E o);
```

Possiamo immaginare che il compilatore rimpiazzi tutte le occorrenze di `E` con il tipo specificato tra le parentesi angolari. Infatti, per l'oggetto `strings` dell'ultimo esempio, non viene riconosciuto il metodo `add(Object obj)` ma `add(String s)`. Ecco spiegato il messaggio di errore.
Ovviamente, questo avviene non per l'intera classe, ma solo per quella particolare istanza trattata. Quindi è possibile creare più liste con differenti parametri. Come nel seguente esempio:

```
List<String> strings = new ArrayList<String>( );
List<Integer> ints = new ArrayList<Integer>( );
List<Date> dates = new ArrayList<Date>( );
```

16.2.2 Tipi primitivi

I generics non si possono applicare ai tipi di dati primitivi. Quindi, il seguente codice:

```
List<int> ints = new ArrayList<int>();
```

Restituirà due messaggi di errore del tipo:

```
. . .
found   : int
required: reference
      List<int> ints = new ArrayList<int>( );
                ^
. . .
```

Fortunatamente però, la seguente sintassi:

```
  List<Integer> ints = new ArrayList<Integer>( );
```

permetterà tranquillamente di aggiungere interi primitivi, grazie alla nuova doppia feature autoboxing e auto-unboxing (cfr. Unità Didattica 16.1).

16.2.3 Interfaccia `Iterator`

Oltre a `List`, tutte le classi e tutte le interfacce Collections supportano ora i generics. Più o meno, quanto visto per `List`, vale per tutte le altre Collections (con l'eccezione per `Map` come vedremo tra poco), ed anche per `Iterator` ed `Enumeration`. Per esempio il seguente codice:

```
List<String> strings = new ArrayList<String>( );
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
```

```
strings.add("Enhanced for loop");
. . .
Iterator i = strings.iterator();
while (i.hasNext()){
    String string = i.next();
    System.out.println(string);
}
```

produrrà il seguente output di errore:

```
found   : java.lang.Object
required: java.lang.String

        String string = i.next();
                        ^
1 error
```

Il problema è che bisogna dichiarare anche l'`Iterator` come generic nel seguente modo:

```
Iterator <String> i = strings.Iterator();
while (i.hasNext()){
    String string = i.next();
    System.out.println(string);
}
```

Attenzione a non utilizzare `Iterator` come generic su una `Collection` non generic…si rischia un'evitabile eccezione al runtime se la `Collection` non è stata riempita come ci si aspetta.

16.2.4 Interfaccia `Map`

L'interfaccia `Map` dichiara due parametri invece. Segue la sua dichiarazione:

```
public interface Map<K,V>
```

Questa volta i due parametri si chiamano `K` e `V`, rispettivamente iniziali di "Key" ("chiave" in italiano) e "Value" ("valore" in italiano). Infatti, per la mappe possono essere parametrizzati sia le chiavi che i valori. Segue un frammento di codice che

dichiara un mappa di tipo generic, con parametro chiave di tipo `Integer` e parametro valore di tipo `String`:

```
Map<Integer, String> map = new HashMap<Integer,
String>();
```

Ovviamente, grazie all'autoboxing e all'auto-unboxing (cfr. Unità didattica 16.1) sarà possibile utilizzare interi primitivi per valorizzare la chiave. Per esempio il seguente codice:

```
map.put(0, "generics");
map.put(1, "metadata");
map.put(2, "enums");
map.put(3, "varargs");
for (int i = 0; i < 4; i++) {
    System.out.println(map.get(i));
}
```

inizializza la mappa e ne stampa i valori con un ciclo sulle sue chiavi.

16.2.5 Ereditarietà di generics

Anche i tipi generic formano gerarchie. Tali gerarchie si basano sui tipi generici non sui tipi parametri dichiarati. Questo significa che bisogna stare attenti a fare casting con i tipi generici. Per esempio il seguente codice è valido:

```
Vector <Integer> vector = new Vector<Integer>();
List <Integer> list = vector;
```

il seguente codice invece, non è valido:

```
Vector <Number> list = vector;
```

Infatti, il fatto che `Number` sia superclasse di `Integer`, non autorizza a considerare il tipo generic `Vector <Number>`, superclasse di `Vector <Integer>`.
Ovviamente non è legale neanche la seguente istruzione:

```
ArrayList <Number> list = new ArrayList<Integer>;
```

Il perchè sia improponibile che il tipo parametro sia la base di una gerarchia, diventa evidente con un semplice esempio. Supponiamo sia legale il seguente codice:

```
List<Integer> ints = new ArrayList<Integer>();
List<Object> objs = ints;
objs.add("Stringa in un generic di Integer?");
```

Ovviamente l'ereditarietà non può che essere determinata dal tipo generic.
Un'altra ragione per cui l'ereditarietà si basa sul tipo generic, è basata sul concetto "erasure" (in italiano "cancellazione"). La gestione dei tipi generic, è gestita solo a livello di compilazione. È il compilatore a trasformare il codice Tiger in codice Java tradizionale prima di trasformarlo in bytecode. Quindi, a livello di runtime, le istruzioni:

```
Vector <Integer> vector = new Vector<Integer>();
Vector <Number> list = vector;
```

saranno lette dalla JVM come se fossero stati cancellati (da qui il termine "erasure") i tipi generic:

```
Vector vector = new Vector();
Vector list = vector;
```

ma a questo punto non si potrebbero più avere informazioni sulla compatibilità degli elementi dei due vettori. Anche per questo il compilatore non permette una ereditarietà basata sui parametri: al runtime, i parametri non esistono più.

16.2.6 Wildcards

Alla luce di quanto appena visto, bisogna fermarci un attimo per riflettere su quali possono essere le conseguenze di un tale comportamento. Supponiamo di avere il seguente metodo:

```
public void print(ArrayList al) {
    Iterator i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
    }
}
```

la compilazione di questo metodo, andrà a buon fine con Tiger, ma includerà dei warning (argomento che affronteremo tra breve). Nonostante sia possibile disabilitare i warning con un'opzione di compilazione, sarebbe sicuramente meglio evitare che ci siano piuttosto che sopprimerli. La soluzione che più facilmente può venire in mente è la seguente:

```
public void print(ArrayList<Object> al) {
    Iterator<Object> i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
    }
}
```

purtroppo però, l'utilizzo del tipo generic, per quanto appena visto sull'ereditarietà, implicherà semplicemente che questo metodo accetterà in input solo tipi generic con parametro `Object` (e non per esempio `String`). Quello che potrebbe essere un parametro polimorfo quindi, rischia di essere un clamoroso errore di programmazione. Ma come risolvere la situazione? A tale scopo esiste una sintassi speciale per i generic che fa uso di **wildcards** (caratteri jolly), in questo caso rappresentati da punti interrogativi. Il seguente codice rappresenta l'unica reale soluzione al problema presentato:

```
public void print(ArrayList <?>al) {
    Iterator<?> i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
    }
}
```

facendo uso di generic, tale metodo non genererà nessun tipo di warning in fase di compilazione, ed accetterà qualsiasi tipo di parametro per l'arraylist in input.

Siccome il compilatore non può controllare la correttezza del tipo di parametro quando viene utilizzato una wildcard, esso rifiuterà di compilare qualsiasi istruzione che tenta di aggiungere o settare elementi nell'arraylist. In pratica, l'utilizzo delle wildcard, trasforma i tipi generic "in sola lettura".

16.2.7 Creare propri tipi generic

Non esistono solo le collezioni che si possono parametrizzare tramite generics, una qualsiasi classe è parametrizzabile. Tanto per fare un esempio la classe `java.lang.Class`, e dichiarata nel seguente modo:

```
public final class Class<T> . . .
```

La parametrizzazione della classe `Class` ci permette di scrivere metodi complessi come il seguente:

```
public static <T> T createObjectFromClass(Class<T> type)
throws . . .{
  T object = null;
  try {
    object = type.newInstance();
  } catch (Exception exc) {
  throw new . . .
  }
  return object;
}
```

questo metodo (che è a sua volta parametrizzato per la definizione del parametro `T`), appartenente alla classe `XMVCUtils` del mio progetto Open Source XMVC (per informazioni http://sourceforge.net/projects/xmvc), e permette di creare istanze da una classe tramite reflection (cfr. Unità Didattica 12.2), senza adoperare la tecnica del casting. Segue un esempio di utilizzo di tale metodo:

```
MiaClasse miaClasse =
XMVCUtils.createObjectFromClass(MiaClasse.class);
```

Nessun casting…

Notare che in questo caso abbiamo parametrizzato anche il metodo `createObjectFromClass()`, con il parametro `<T>`. La sintassi purtroppo non è delle più semplici.

È anche possibile definire i propri tipi generic, come nel seguente esempio:

```
public class OwnGeneric <E> {
  private List<E> list;
  public OwnGeneric () {
    list = new ArrayList<E>();
  }
  public void add(E e) {
    list.add(e);
  }
  public void remove(int i) {
    list.remove(i);
  }
  public E get(int i) {
    return list.get(i);
  }
  public int size( ) {
    return list.size( );
  }

  public boolean isEmpty( ) {
    return list.size( ) == 0;
  }
  public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < size(); i++) {
      sb.append(get(i) + (i != size() - 1 ? "-" : ""));
    }
    return sb.toString();
  }
}
```

Probabilmente l'implementazione del metodo `toString()`, ha bisogno di qualche osservazione supplementare. Per prima cosa notiamo l'utilizzo della classe (introdotta con Tiger) `StringBuilder`. Questa è del tutto simile alla classe `StringBuffer`, ma è più performante perché non è thread-safe.
Inoltre, l'implementazione del ciclo `for` può risultare criptica a prima vista. In effetti, l'utilizzo dell'operatore ternario non aiuta la leggibilità. Analizzando con calma il ciclo

`for` in questione, è per prima cosa possibile notare che è costituito da un'unica espressione. Questa, aggiunge una stringa all'oggetto `sb` di tipo `StringBuilder`, tramite il metodo `append()`. Il parametro di questo metodo è costituito dal ritorno del metodo `get()`, concatenato con il risultato dell'operatore ternario compreso tra parentesi tonde. Tale operatore ritornerà un trattino (`"-"`) per separare i vari elementi estratti dalla collezione, se e solo se il valore di `i` è diverso la dimensione della collezione `-1`. In questo modo l'output risultante avrà una formattazione corretta. Notiamo come il parametro è stato definito con una `E`, come si usa nella documentazione. Tuttavia, benché sia preferibile utilizzare un'unica lettera maiuscola per definire il parametro, è possibile utilizzare una qualsiasi parola valida per la sintassi Java. Con il seguente codice invece, andiamo ad utilizzare il nostro tipo generic:

```
OwnGeneric <String> own = new OwnGeneric <String>();
for (int i = 0; i < 10; ++i) {
    own.add(""+ (i));
}
System.out.println(own);
```

L'output sarà:

0-1-2-3-4-5-6-7-8-9

Per quanto riguarda la classe generic che abbiamo appena creato, non è possibile dichiarare statica la variabile d'istanza `list`, nel seguente modo:
`private static List<E> list = new ArrayList<E>();`
Infatti, questo impedirebbe alle varie istanze della classe di utilizzare parametri differenti (visto che devono essere condivisi…). È comunque possibile creare metodi statici come il seguente:
`public static boolean confronta(OwnGeneric<E> o1, OwnGeneric<E> o2) {. . .}`

È possibile anche creare propri tipi generici con parametri "ristretti" a determinati tipi. Per esempio, se definiamo la classe precedente nel seguente modo:

```
public class OwnGeneric <E extends Number> {
```

allora potremo utilizzare come parametri solo sottoclassi di `Number` (per esempio `Float` o `Integer`...)
Supponiamo, di non trovarci all'interno di un tipo generic creato da noi. Inoltre supponiamo di voler creare un metodo che prende come parametro un tipo generic, che a sua volta deve avere un parametro ristretto ad un altro tipo. La sintassi per implementare tale metodo è la seguente:

```
public void print(List <? extends Number> list) {
    for (Iterator<? extends Number> i = list.iterator();
i.hasNext( ); ) {
      System.out.println(i.next());
    }
}
```

l'utilizzo della wildcard, è obbligata se non ci troviamo in una classe di tipo generic creata da noi (come la classe `OwnGeneric`), che già dichiara un parametro (nel caso di `OwnGeneric` si chiamava `E`).
In realtà esiste anche una sintassi molto speciale che permette di non utilizzare la wildcard:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<A> i = list.iterator(); i.hasNext( ); )
{
        System.out.println(i.next());
    }
}
```

in pratica con l'istruzione:

```
<N extends Number>
```

prima del tipo di ritorno del metodo, stiamo in pratica dichiarando localmente, un parametro chiamato `N`, che deve avere la caratteristica di estendere `Number`. Questo parametro sarà utilizzabile all'interno dell'intero metodo. Questa sintassi può essere preferibile quando per esempio, all'interno del codice del metodo viene spesso utilizzato il parametro `<N>` (altrimenti siamo obbligati a scrivere più volte "`<? Extends Number>`").

È possibile anche creare "innesti di generics", per esempio il seguente codice è valido:

```
Map<Integer, ArrayList<String>> map =
  new HashMap<Integer, ArrayList<String>>();
```

Per ricavare un elemento dell'arraylist innestato è possibile utilizzare il seguente codice:

```
String s = map.get(chiave).get(numero);
```

Possiamo notare come non abbiamo utilizzato neanche un casting.

16.2.8 Impatto su Java

Si potrebbero scrivere interi libri sui generics e tutte le conseguenze del loro utilizzo, ma in questa sede non è opportuno proseguire oltre. In fondo, molti argomenti del precedente paragrafo, potrebbero essere considerate impatti sul linguaggio. Si pensi per esempio, all'impatto sulla sintassi, sulla documentazione e sull'ereditarietà.

16.2.9 Compilazione

Un impatto molto evidente dei generics su Java, di cui non si può non parlare, è relativo ai messaggi del compilatore. Abbiamo più volte evidenziato che saranno lanciati dei warning, se compiliamo file sorgenti, che dichiarino collection che potrebbero essere parametrizzate ma non lo sono (da Java 5 in poi queste vengono dette "raw type"). Per esempio, il seguente codice:

```
List strings = new ArrayList( );
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
strings.add("Enhanced for loop");
Iterator  i = strings.iterator();
while (i.hasNext()) {
    String string = (String)i.next();
    System.out.println(string);
}
```

avrà un esito di compilazione positivo, ma provocherà il seguente output:

```
Note: Generics1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

In pratica tali warning (o "note", o "lint" come vengono definiti nelle specifiche), avvertono lo sviluppatore che esistono delle operazioni non sicure o non controllate, e viene richiesto di ricompilare il file con l'opzione "–Xlint", per avere ulteriori dettagli. Seguendo il suggerimento del compilatore, ricompiliamo il file con l'opzione richiesta. L'output sarà:

```
Generics1.java:16: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.List
        strings.add("Autoboxing & Auto-unboxing");
                ^
Generics1.java:17: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.List
        strings.add("Generics");
                ^
Generics1.java:18: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.List
        strings.add("Static imports");
                ^
Generics1.java:19: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.List
        strings.add("Enhanced for loop");
                ^
4 warnings
```

Questa volta vengono segnalati come warning le righe di codice "incriminate", con una breve spiegazione, nel solito stile Java.
Come è possibile notare questi warning vengono definiti "`[unchecked]`". Infatti esistono vari tipi di warning, di cui gli unchecked sono considerati i più rilevanti. Infatti, lanciando la compilazione con l'opzione "-X" (che serve a dare una sinossi delle opzioni non standard, ovvero che potrebbero cambiare anche domani), scopriremo che riguardo a Xlint esistono le seguenti opzioni:

```
-Xlint: {all, deprecation, unchecked, fallthrough, path, serial, finally, -deprecation,
-unchecked,
-fallthrough, -path, -serial, -finally}
```

Quindi, se per esempio compilando il nostro file con l'opzione "–Xlint", otteniamo troppi warning (magari perché compiliamo con Tiger, codice Java "vecchio"), abbiamo

la possibilità di leggere solo la tipologia di warning che ci interessa. Se specifichiamo solo l'opzione "Xlint" senza sotto-opzioni, di default verrà utilizzata la sotto-opzione "all".
Per dettagli sulle possibili ulteriori opzioni di "Xlint", il lettore è invitato a consultare la documentazione dei tool del Java Development Kit (cartella docs\tooldocs\index.html della documentazione ufficiale).

Esistono due soluzioni per non ricevere warning dal compilatore. La prima è quella di compilare con il flag "–source 1.4", ma questo non è sempre possibile. Infatti, potremmo avere nello stesso codice delle collection parametrizzate ed altre no. La seconda soluzione, riguarda l'utilizzo di una cosiddetta annotazione standard (cfr. Modulo 19) chiamata `SuppressWarnings`.
Ovviamente, la migliore soluzione è applicare la parametrizzazione ovunque sia possibile.

16.2.10 Cambiamento di mentalità

Con l'avvento dei generics, anche il programmatore più bravo dovrà in qualche modo fare i conti con la sua radicata mentalità Object Oriented. L'abitudine a creare gerarchie complicate per astrarre al meglio il sistema, o ad implementare complicati controlli per astrarre i giusti vincoli, a volte può essere sostituita da codice che sfrutta generics, risparmiando diverse righe di codice, e probabili bachi dell'applicazione. Purtroppo, risparmiare righe di codice con i parametri dei generics, è di solito conseguenza di un processo mentale molto complesso. Anche la leggibilità del codice, già di per sé non immediata quando esistono gerarchie di classi, si riduce notevolmente, rendendo più ardua la comprensione del codice stesso.
Bisogna in pratica astrarsi dalla Object Orientation stessa in alcuni casi!
Un esempio pratico viene presentato nel prossimo paragrafo sui parametri covarianti.

1.1.4. 16.2.11 Parametri covarianti

Nel modulo 6 abbiamo affermato che dalla versione 5 di Java in poi sono stati definiti i cosiddetti tipi di ritorno covarianti. In pratica è possibile ora creare override di metodi il cui tipo di ritorno è una sottoclasse del tipo di ritorno del metodo originale. Per esempio, sempre considerando il rapporto di ereditarietà che sussiste tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto`, fosse presente il seguente metodo:

```
public Punto getClone() throws
CloneNotSupportedException{
```

```
    return (Punto)this.clone();
}
```

allora sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
public PuntoTridimensionale getClone() throws
CloneNotSupportedException {
    return (PuntoTridimensionale)this.clone();
}
```

Le regole dell'override però non sono cambiate relativamente ai parametri del metodo da overridare: questi devono coincidere. Nel caso non coincidessero si dovrebbe parlare di overload e non di override (cfr. Modulo 6). Non si può parlare quindi dell'esistenza di "parametri covarianti" in senso stretto.
Con l'avvento dei generics però, è possibile implementare praticamente un parametro covariante. Ovviamente parliamo di uno stratagemma (o sarebbe meglio definirlo pattern) non banale.
Come al solito, facciamo un esempio. Consideriamo le due seguenti interfacce:

```
interface Cibo {
    String getColore();
}
interface Animale {
    void mangia(Cibo cibo);
}
```

è facile implementare l'interfaccia `Cibo` nella classe `Erba`:

```
public class Erba implements Cibo {
    public String getColore(){
        return "verde";
    }
}
```

come è facile implementare l'interfaccia `Animale` nella classe `Carnivoro`

```
public class Carnivoro implements Animale {
    public void mangia(Cibo cibo) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
```

e, sia l'interfaccia `Animale` che l'interfaccia `Cibo` nella classe `Erbivoro` (dato che potrebbe essere il cibo di un carnivoro):

```
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) {
        //un erbivoro mangia erba
    }

    public String getColore() {
      . . .
    }
}
```

Il problema è che in questo modo sia un carnivoro sia un erbivoro potrebbero mangiare qualsiasi cosa, per esempio un carnivoro potrebbe cibarsi d'erba, e questo è inverosimile. Potremmo risolvere la situazione sfruttando dei controlli internamente ai metodi, che tramite l'operatore `instanceof`, e l'eventuale lancio di un'eccezione personalizzata (`CiboException`) al runtime, impongano i giusti vincoli alle nostre classi. Per esempio:

```
public class Carnivoro implements Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erbivoro)) {
            throw new CiboException("Un carnivoro deve "
              + "mangiare erbivori!");
        }
        . . .
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erba)) {
```

```
            throw new CiboException("Un erbivoro deve " +
              "mangiare erba!");
        }
        . . .
    }

    public String getColore() {
        . . .
    }
}
public class CiboException extends Exception {
    public CiboException(String msg){
        super(msg);
    }
    . . .
}
```

Inoltre per le regole dell'override relative alle eccezioni (cfr. Modulo 10), per poter compilare correttamente le nostre classi dovremo anche ridefinire l'interfaccia `Animale`, in modo tale che il metodo `mangia()` overridato, definisca una clausola `throws` per il lancio di una `CiboException`. Segue la ridefinizione dell'interfaccia `Animale`:

```
interface Animale {
    void mangia(Cibo cibo) throws CiboException;
}
```

Le nostre classi sono ora astratte più correttamente, ma un eventuale problema, verrà rilevato solo durante l'esecuzione dell'applicazione. Per esempio il seguente codice compilerà correttamente, salvo poi lanciare un'eccezione al runtime:

```
public class TestAnimali {
    public static void main(String[] args) {
        try {
            Animale tigre = new Carnivoro();
            Cibo erba = new Erba();
            tigre.mangia(erba);
        } catch (CiboException exc) {
```

```
            exc.printStackTrace();
        }
    }
}
```

Ecco l'output:

```
CiboException: Un carnivoro deve mangiare erbivori!
    at Carnivoro.mangia(Carnivoro.java:4)
    at TestAnimali.main(TestAnimali.java:6)
```

L'ideale però, sarebbe quello di "restringere" il parametro polimorfo del metodo `mangia()` ad `Erba` per la classe `Erbivoro`, e ad `Erbivoro` per la classe `Carnivoro`. In pratica ci piacerebbe utilizzare i parametri covarianti nel metodo `mangia()`. Così infatti, il codice precedente non sarebbe neanche compilabile! E questo sarebbe un vantaggio non da poco. Segue un primo tentativo di soluzione:

```
public class Carnivoro implements Animale {
    public void mangia(Erbivoro erbivoro) {
        . . .
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Erba erba) {
        . . .
    }

    public String getColore() {
        . . .
    }
}
```

Purtroppo questo codice non compilerà, perché appunto non avremo implementato in nessuna delle due classi il metodo `mangia()` che prende come parametro un oggetto di tipo `Cibo`. Quindi, ereditando metodi astratti in classi non astratte otterremo errori in compilazione.
Per raggiungere il nostro scopo possiamo però sfruttare appunto i generics. Ridefiniamo l'interfaccia `Animale`, parametrizzandola nel seguente modo:

```
interface Animale <C extends Cibo> {
    void mangia(C cibo);
}
```

Ora possiamo ridefinire le classi `Carnivoro` ed `Erbivoro` nel seguente modo:

```
public class Carnivoro implements Animale<Erbivoro> {
    public void mangia(Erbivoro erbivoro) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
public class Erbivoro<E extends Erba> implements Cibo,
Animale<E> {
    public void mangia(E erba) {
      //un erbivoro mangia erba
    }

    public String getColore() {
      . . .
    }
}
```

Il codice precedente viene compilato correttamente e impone i giusti vincoli senza problemi per le gerarchie create. Rimangono ovviamente le evidenti difficoltà di approccio al codice precedente. Segue una classe con metodo `main()`, che utilizza correttamente le precedenti:

```
public class TestAnimali {
    public static void main(String[] args) {
      Animale<Erbivoro> tigre = new
        Carnivoro<Erbivoro>();
      Erbivoro<Erba> erbivoro = new Erbivoro<Erba>();
      tigre.mangia(erbivoro);
    }
}
```

Ovviamente il metodo `mangia()` dell'interfaccia `Animale` non deve più definire la clausola `throws` ad una `CiboException`. Anzi, la classe `CiboException` non è più necessaria.

16.2.12 Casting automatico di reference al loro tipo "intersezione" nelle operazioni condizionali

Una conseguenza del discorso sui generics, è il casting automatico dei reference al loro tipo "intersezione". Stiamo parlando della proprietà che hanno introdotto i generics in Tiger, che rende compatibili due tipi che hanno una superclasse comune, senza obbligare lo sviluppatore ad esplicitare casting. Per esempio, se consideriamo il seguente codice:

```
Vector <Number> v = new Vector<Number>();
v.add(1);
v.add(2.0F);
v.add(2.0D);
v.add(2L);
Iterator<Number> i = v.iterator();
while(i.hasNext()){
    Number n = i.next();
}
```

possiamo notare come non ci sia stato bisogno di esplicitare nessun tipo di casting. Infatti, anche avendo aggiunto a `v` elementi di tipo `Integer`, `Float`, `Double` e `Long` (grazie all'autoboxing), e nonostante il metodo `next()` di `Iterator` restituisca un `Object`, nell'estrazione tramite `Iterator` parametrizzato, gli elementi del vettore sono stati automaticamente castati a `Number`. Questo perché il compilatore ha fatto il lavoro per noi.
Bisogna quindi osservare un nuovo comportamento dell'operatore ternario (cfr. modulo 4), e più in generale di alcune situazioni condizionali. Precedentemente a Tiger infatti, un codice come il seguente provocava un errore in compilazione:

```
import java.util.*;

public class Ternary {
    public static void main(String args[]) {
        String s = "14/04/04";
        Date today = new Date();
        boolean flag = true;
```

```
        Object obj = flag ? s : today;
    }
}
```

Segue l'errore evidenziato dal compilatore:

```
incompatible types for ?: neither is a subtype of the other
second operand: java.lang.String
third operand : java.util.Date
      Object obj = flag ? s : today;
                   ^
1 error
```

In pratica il compilatore non eseguiva nessun casting esplicito ad `Object`, nonostante sembri scontato che l'espressione non sia ambigua. Il problema si poteva risolvere solo grazie ad un esplicito casting ad `Object` delle due espressioni. Ovvero bisognava sostituire la precedente espressione riguardante l'operatore ternario con la seguente:

```
Object obj = flag ? (Object)s : (Object)today;
```

Il che sembra più una complicazione che una dimostrazione di robustezza di Java. In Tiger invece, il problema viene implicitamente risolto dal compilatore, senza sforzi del programmatore. Infatti `s` e `today`, vengono automaticamente castati ad `Object`.

Riepilogo

In questo modulo abbiamo affrontato due nuovi argomenti, entrambi molto utili ed interessanti. Abbiamo visto come l'autoboxing e l'auto-unboxing semplifichino la vita al programmatore, facendogli risparmiare noiose righe di codice. Nonostante questa nuova feature sia particolarmente semplice da utilizzare, nasconde comunque alcune insidie. Nel paragrafo dedicato agli impatti della nuova feature sul linguaggio infatti, abbiamo isolato alcuni casi particolari.
Nella seconda parte del modulo abbiamo trattato uno degli argomenti più interessanti di Tiger: i tipi generici. Dopo aver valutato la parametrizzazione di liste, mappe e iteratori con i generics, abbiamo cercato di capire qual è il lavoro del compilatore dietro le quinte. Abbiamo visto come l'ereditarietà non si basi sul tipo parametro, e come utilizzare le wildcard, quando è necessario generalizzare i parametri. Abbiamo anche visto come

creare tipi generici personalizzati. Infine abbiamo valutato l'impatto di questa nuova feature sul linguaggio e sul vecchio modo di compilare i nostri file sorgenti.

Esercizi modulo 16

Esercizio 16.a)
Autoboxing, Auto-unboxing e Generics, Vero o Falso:

1. Il seguente codice compila senza errori:
```
char c = new String("Pippo");
```
2. Il seguente codice compila senza errori:
```
int c = new Integer(1) + 1 + new Character('a');
```
3. Il seguente codice compila senza errori:
```
Integer i = 0;
switch(i){
  case 0:
    System.out.println();
  break;
}
```
4. Le regole dell'overload non cambiano con l'introduzione dell'autoboxing e dell'auto-unboxing
5. Per confrontare correttamente il contenuto di due `Integer` è necessario utilizzare il metodo `equals()`. L'operatore == infatti, potrebbe aver un comportamento anomalo su istanze che hanno un range limitato, causa ottimizzazioni delle prestazioni di Java
6. Il seguente codice:
```
List<String> strings = new ArrayList<String>( );
strings.add(new Character('A'));
```
   compila senza errori

7. Il seguente codice:
```
List<int> ints = new ArrayList<int>();
```
   compila senza errori
8. Il seguente codice:
```
List<int> ints = new ArrayList<Integer>();
```
   compila senza errori
9. Il seguente codice:
```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
```
   compila senza errori

10. Il seguente codice:

```
List ints = new ArrayList<Integer>();
```

compila senza erroriù

Esercizio 16.b)
Generics, Vero o Falso:

1. Se compiliamo file che utilizzano collection senza utilizzare generics con un JDK 1.5, otterremo errori in compilazione
2. La sintassi:

```
public interface Collection<E>
```

non sottintende l'esistenza di una classe `E`. Si tratta di una nuova terminologia che sta ad indicare che `Collection`, supporti la parametrizzazione tramite generics

3. Non è possibile compilare del codice che utilizza Generics con l'opzione "–source 1.4"
4. Non è possibile compilare del codice che utilizza Generics con l'opzione "–Xlint"
5. L'esecuzione di un file che non utilizza generics con un JVM 1.5, darà luogo a warning
6. Il seguente codice:

```
Collection <java.awt.Component> comps = new
Vector<java.awt.Component>();
    comps.add(new java.awt.Button());
    Iterator i = comps.iterator();
    while (i.hasNext()){
        Button button = i.next();
        System.out.println(button.getLabel());
}
```

compila senza errori

7. Il seguente codice:

```
Collection <Object> objs = new Vector<String>();
```

compila senza errori

8. Il seguente codice:

```
public void print(ArrayList<Object> al) {
    Iterator<Object> i = al.iterator();
    while (i.hasNext()) {
      Object o = i.next();
```

```
    }
  }
```

9. Il seguente codice:

```
public class MyGeneric <Pippo extends Number> {
    private List<Pippo> list;
public MyGeneric () {
      list = new ArrayList<Pippo>();
    }
public void add(Pippo pippo) {
      list.add(pippo);
    }
public void remove(int i) {
      list.remove(i);
    }
public Pippo get(int i) {
      return list.get(i);
    }
public void copy(ArrayList<?> al) {
      Iterator<?> i = al.iterator();
      while (i.hasNext()) {
          Object o = i.next();
          add(o);
      }
    }
  }
```

10. Il seguente codice:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<A> i = list.iterator();
        i.hasNext( ); ) {
      System.out.println(i.next());
    }
  }
```

compila senza errori

Soluzioni esercizi modulo 16

Esercizio 16.a)
Autoboxing, auto-unboxing e Generics, Vero o Falso:

1. **Falso**
2. **Vero**
3. **Vero**
4. **Vero**
5. **Vero**
6. **Falso**
7. **Falso**
8. **Falso**
9. **Falso**
10. **Vero**

Esercizio 16.b)
Generics, Vero o Falso:

1. **Falso** l'`Iterator` deve essere parametrizzato
2. **Vero**
3. **Vero**
4. **Falso**
5. **Falso**
6. **Falso** solo di tipo `ActionListener`
7. **Falso**
8. **Vero**

9. **Falso** otterremo il seguente errore:
   (Pippo) in MyGeneric<Pippo> cannot be applied to (java.lang.Object)
   add(o);
   ^
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere le semplificazioni che ci offre la nuova (doppia) feature autoboxing e auto-unboxing (unità 16.1)	☐	
Conoscere le conseguenze e i problemi che genera l'introduzione dell'autoboxing e dell'auto-unboxing nel linguaggio Java (unità 16.1)	☐	
Capire cos'è un tipo generic (unità 16.2)	☐	
Saper utilizzare i tipi generic (unità 16.2)	☐	
Aver presente l'impatto su Java dell'introduzione dei generics (unità 16.2)	☐	

Note:

17

Complessità: alta

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper utilizzare il ciclo for-migliorato (unità 17.1).
2. Comprendere i limiti e quando applicare il ciclo for migliorato (unità 17.1).
3. Comprendere e saper utilizzare le enumerazioni (unità 17.2).
4. Comprendere le caratteristiche avanzate e quando utilizzare le enumerazioni (unità 17.2).

17 Ciclo for migliorato ed Enumerazioni

Anche in questo modulo affronteremo dapprima un argomento semplice come il ciclo for migliorato, e dopo un argomento complesso come le enumerazioni.

17.1 Ciclo for migliorato

Nel modulo 4, abbiamo già visto che Java 5 ha definito una nuova tipologia di ciclo, chiamata **enhanced for loop**, (in italiano **ciclo for migliorato**).

> **Altri nomi che vengono utilizzati per questo ciclo sono "ciclo for/in" o anche "ciclo foreach".**

Questo, più che migliorato, rispetto agli altri cicli, dovrebbe chiamarsi "semplificato". Infatti, il compilatore tramuterà la sintassi del ciclo for migliorato, in un ciclo `for` "normale" al momento della compilazione.
La sintassi è effettivamente più compatta:

```
for (dichiarazione : espressione)
    statement
```

dove:

1. l'`espressione` deve coincidere o con un array o con un oggetto che implementa la nuova interfaccia `java.lang.Iterable` (questa viene ovviamente implementata da tutte le collection).
2. La `dichiarazione` deve dichiarare un reference ad un oggetto compatibile con il tipo dell'array (o dell'oggetto `Iterable`) dichiarato nell'`espressione`. Non è possibile utilizzare variabili dichiarate esternamente al ciclo.

Come sempre, se ci sono più statement all'interno del ciclo, è possibile utilizzare le parentesi graffe per circondarli. E come sempre, è consigliabile utilizzare le parentesi anche nel caso lo statement sia unico.

Per esempio, dando per scontato che da ora in poi faremo largo uso dei tipi generici, consideriamo il seguente codice:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Iterator <Integer> i = list.iterator();
i.hasNext();) {
    Integer value = i.next();
    System.out.println(value);
}
```

è possibile sostituire il precedente codice con il seguente:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Integer integer : list) {
    System.out.println(integer);
}
```

Che è sicuramente più semplice e compatto. Ad ogni iterazione, ogni elemento di ***`list`*** viene semplicemente assegnato alla variabile `integer`, dichiarata proprio per puntare localmente al valore estratto dalla `list`. Come si può notare subito, utilizzando il ciclo for migliorato, non si è dovuto utilizzare nessun oggetto `Iterator`.

Probabilmente il nome più appropriato per questo nuovo ciclo è "ciclo foreach", come si usa in altri linguaggi. Immaginiamo :

```
for (Integer integer : list) {
    . . .
}
```
è un po' come dire: per ogni `integer` in `list`...
In altri linguaggi `foreach` è una parola chiave, ma in Tiger (per una volta) si è scelto di non aggiungere un'ennesima complicazione.

Con un ciclo for migliorato, è possibile ciclare non solo su oggetti `Iterable`, ma anche su normali array. Per esempio:

```
int[] ints = new int[10];
. . .
for (int n : ints) {
    System.out.println(n);
}
```

Nella dichiarazione, è possibile utilizzare anche le annotazioni (cfr. Modulo 19) e il modificatore `final`. In particolare l'utilizzo di `final`, rafforzerebbe il ruolo della variabile dichiarata, di dover assumere solo il valore dell'elemento assegnatogli all'interno dell'iterazione. Inoltre enfatizzerebbe l'immutabilità di questa variabile all'interno delle iterazioni.

17.1.1 Limiti del ciclo for migliorato

Il nuovo ciclo non può sostituire in tutte le situazioni il ciclo `for` tradizionale. Ci sono delle operazioni che si possono svolgere con i "vecchi" cicli, che non risultano possibili con l'enhanced for.

1. Non è possibile accedere all'interno delle iterazioni, al numero di iterazione corrente. Questa sembra essere una grossa limitazione che ridimensiona la possibilità di utilizzo del nuovo ciclo. Per esempio, nell'unità didattica 16.2 relativa ai generics, nell'esempio relativo alla creazione di tipi generici, abbiamo creato la classe `OwnGeneric`, che dichiarava il seguente metodo `toString()` (già analizzato nella relativa unità didattica):
   ```
   public String toString() {
       StringBuilder sb = new StringBuilder();
   ```

```
        for (int i = 0; i < size(); i++) {
            sb.append(get(i) + (i!=size()-1?"-":""));
        }
        return sb.toString();
    }
```

In questo caso, un ciclo for migliorato, non potrebbe sostituire il ciclo `for` precedente.

2. Non è possibile ciclare all'indietro.
3. Non è possibile ciclare contemporaneamente su due array o oggetti `Iterable` sfruttando un unico ciclo.
4. Non potendo accedere all'indice corrente, non è possibile sfruttare alcuni metodi delle collection, come per esempio il metodo `get()` per un `ArrayList`. Tuttavia, è possibile dichiarare esternamente al ciclo un indice da incrementare all'interno del ciclo, così come si fa solitamente per un ciclo ***`while`***. Per esempio:

```
Vector <String>strings = new Vector<String>();
. . .
int i = 0;
for (Object o : strings) {
     System.out.println(strings.get(i++));
}
```

5. Non potendo accedere all'oggetto `Iterator` (almeno non in maniera corretta), non è possibile sfruttare metodi di `Iterator` come `remove()`.

17.1.2 Implementazione di un tipo Iterable

È possibile, e alcune volte auspicabile, creare collezioni personalizzate, magari estendendo una collection già pronta e completa. Per esempio possiamo facilmente estendere la classe `Vector`, ed aggiungere nuovi metodi. In tali casi, sarà già possibile sfruttare il nostro nuovo tipo all'interno di un ciclo for migliorato. Infatti, la condizione necessaria affinché una classe sia parte di un costrutto `foreach`, è che implementi la nuova interfaccia `Iterable`. `Vector`, come tutte le altre collection, implementa tale interfaccia.
Implementare l'interfaccia `Iterable`, significa implementare un unico metodo: il metodo `iterator()`. Segue la definizione dell'interfaccia `Iterable`:

```
package java.lang;
public interface Iterable<E> {
```

```
  public java.util.Iterator<E> iterator( );
}
```

Questo metodo, che dovrebbe già essere familiare al lettore, restituisce un'implementazione dell'interfaccia `Iterator`, più volte utilizzata in questo testo. Non è così quindi banale implementare "da zero" `Iterable`, perché significa anche definire un `Iterator` personalizzato. Non si tratta quindi di definire solo il metodo `iterator()`, ma anche implementare tutti i metodi dell'interfaccia `Iterator`, che è così definita:

```
package java.util;
public interface Iterator<E> {
    public boolean hasNext( );
    public E next( );
    public void remove( );
}
```

17.1.3 Impatto su Java

In effetti l'introduzione di questo nuovo ciclo, non dovrebbe sconvolgere il mondo della programmazione Java. Non è stata introdotta infatti, una caratteristica che aggiunge potenza al linguaggio. Tutto quello che si può fare con questo nuovo ciclo, già si poteva fare in precedenza.
Inoltre non c'è niente di nuovo rispetto ad altri linguaggi. Però ora si può dire che anche Java ha il suo ciclo `foreach`. In futuro questo segnerà un punto a favore nell'indice di gradimento di coloro che effettueranno una migrazione da un linguaggio che definisce tale ciclo, a Java. Finalmente non sentirò più dire ai miei corsisti: "ma come, Visual Basic ha qualcosa in più di Java?"… non sopporto queste affermazioni provocatorie, sappiatelo miei prossimi corsisti…
Il principio fondamentale che ha portato all'introduzione di questo nuovo ciclo in Tiger, è quello che è parte integrante di alcuni software (come per esempio lo storico "VI"), ovvero "meno scrivi, meglio è"…
In effetti, uno dei benefici più evidenti dell'utilizzo del nuovo ciclo è proprio quello di poter evitare di utilizzare gli `Iterator` (o le `Enumeration`), e quindi di risparmiare righe di codice con eleganza.

17.2 Tipi Enumerazioni

Questo argomento (già introdotto nell'Unità Didattica 9.8) è particolarmente importante. Si tratta di introdurre una nuova tipologia di struttura dati, che si va ad aggiungere alle classi, alle interfacce ed alle annotazioni (come vedremo nel Modulo 19). Inoltre, viene anche introdotta una nuova parola chiave, con conseguenti problemi di compatibilità all'indietro del codice Java. Java subisce una vera e propria evoluzione con tale costrutto, anche se i puristi dell'object orientation potrebbero storcere il naso.
Gli **enumerated types**, che traduciamo come **tipi enumerazioni**, o più semplicemente solo con **enumerazioni** (o volendo **enum**), esistono anche in altri linguaggi (primo fra tutti il C). Quindi, anche questa è una di quelle caratteristiche che favorirà le migrazioni da altri linguaggi in futuro, o almeno così spera Sun. Non si può dire che le enum, permettano di creare codice che prima non si poteva creare con le classi, ma, semmai, si può affermare che permetteranno di creare codice più robusto.
Passiamo subito ad un esempio per comprendere cosa sono e soprattutto a cosa servono le enumerazioni. Supponiamo di voler creare una serie di costanti, ognuna delle quali rappresenti una nuova feature di Java 5. È possibile fare questo con una classe (o con un'interfaccia), ma è sicuramente preferibile utilizzare un'enumerazione, che è stata introdotta nel linguaggio, proprio a questo scopo. Segue un esempio di codice:

```
public enum TigerNewFeature {
    ANNOTATIONS, AUTOBOXING, ENUMERATIONS, FOREACH,
    FORMATTING, GENERICS, STATIC_IMPORTS, VARARGS
}
```

In pratica è come se avessimo definito un nuovo tipo. Infatti, come vedremo tra breve, un'enumerazione viene trasformata dal compilatore in una classe che estende la classe astratta `Enum` (package `java.lang`). Gli **elementi** (detti anche **valori**) di questa enum, sono implicitamente di tipo `TigerNewFeature`, e quindi non va specificato il tipo. Essi vengono semplicemente definiti separandoli con delle virgole. Si tratta di costanti statiche, ma non bisogna dichiararle né `final` né `public` né `static`, anche in questo caso, già lo sono implicitamente.
Ogni elemento di `TigerNewFeature`, è di tipo `TigerNewFeature`. È questa la caratteristica delle enum più difficile da digerire all'inizio. In pratica, definita un'enumerazione, si definiscono anche tutte le sue possibili istanze. Non si possono istanziare altre `TigerNewFeature`, oltre a quelle definite da `TigerNewFeature`

stessa. Per tale motivi, gli elementi vengono spesso chiamati anche "valori"dell'enumerazione.

Trattandosi di costanti, la convenzione Java (cfr. Modulo 2), consiglia di definire gli elementi di un'enumerazione con caratteri maiuscoli. Ricordiamo che come separatore di parole, può essere utilizzato il carattere underscore ("_"). Inoltre ovviamente le enumerazioni vanno definite con la stessa convenzione delle classi.

La sintassi di un'enum, purtroppo, non si limita solo a quanto appena visto. Esistono tante altre caratteristiche che può avere un'enumerazione, come implementare interfacce, definire costruttori, metodi etc….
Ora che abbiamo visto la sintassi di un'enum, vediamo come si utilizza. Consideriamo le seguenti classi, `Programmatore` e `ProgrammatoreJava`:

```
public class Programmatore {
  private String nome;
  public Programmatore() {
    nome = "";
  }
  public Programmatore(String nome) {
    this.setNome(nome);
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public String getNome() {
    return nome;
  }
}

import java.util.ArrayList;

public class ProgrammatoreJava extends Programmatore {
  ArrayList <TigerNewFeature> aggiornamenti;
  public ProgrammatoreJava () {
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
```

```
  public ProgrammatoreJava (String nome) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
  public void aggiungiAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.add(aggiornamento);
  }
  public void rimuoviAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.remove(aggiornamento);
  }
  public String toString() {
    StringBuilder sb = new StringBuilder(getNome());
    sb.append(" è aggiornato a :" + aggiornamenti);
    return sb.toString();
  }
}
```

Come è possibile notare la classe `ProgrammatoreJava`, definisce un `ArrayList` generico (`aggiornamenti`) parametrizzato con l'enumerazione appena dichiarata. Inoltre definisce due metodi (`aggiungiAggiornamento()` e `rimuoviAggiornamento()`) che gestiscono il contenuto dell'`ArrayList`. L'enumerazione viene quindi trattata come un tipo qualsiasi. La seguente classe mostra un esempio di utilizzo di enum:

```
public class TestEnum1 {
    public static void main(String args[]) {
        ProgrammatoreJava pro = new
          ProgrammatoreJava("Pippo");

pro.aggiungiAggiornamento(TigerNewFeature.ANNOTATIONS);

pro.aggiungiAggiornamento(TigerNewFeature.AUTOBOXING);

pro.aggiungiAggiornamento(TigerNewFeature.ENUMERATIONS);

pro.aggiungiAggiornamento(TigerNewFeature.FOREACH);
```

```
pro.aggiungiAggiornamento(TigerNewFeature.FORMATTING);

pro.aggiungiAggiornamento(TigerNewFeature.GENERICS);

pro.aggiungiAggiornamento(TigerNewFeature.STATIC_IMPORTS)
;

pro.aggiungiAggiornamento(TigerNewFeature.VARARGS);
        System.out.println(pro);
    }
}
```

l'output di tale file sarà:

```
Pippo è aggiornato a :[ANNOTATIONS, AUTOBOXING,
 ENUMERATIONS, FOREACH, FORMATTING, GENERICS,
 STATIC_IMPORTS, VARARGS]
```

Come è possibile notare, i vari elementi dell'enumerazione `TigerNewFeature`, sono di tipo `TigerNewFeature`, altrimenti non avremmo potuto passare tali elementi come parametri al metodo `aggiungiAggiornamento()`.

17.2.1 Perché usare le enumerazioni

Nonostante un possibile rifiuto iniziale verso questa nuova feature, da parte dei programmatori Java classici (senza background C), le enum rappresentano una comodità notevole per lo sviluppatore. Principalmente infatti, evitano qualsiasi tipo di controllo sul tipo, che a volte può diventare abbastanza pesante. Ma proviamo a riscrivere l'esempio precedente, senza utilizzare le enumerazioni (ma sfruttando i generics, autoboxing e auto-unboxing):

```
public class TigerNewFeature {
    public static final int ANNOTATIONS = 0;
    public static final int AUTOBOXING = 1;
    public static final int  ENUMERATIONS = 2;
    public static final int FOREACH = 3;
    public static final int FORMATTING = 4;
    public static final int GENERICS = 5;
```

```
    public static final int  STATIC_IMPORTS = 6;
    public static final int VARARGS = 7;
}
```

forse siamo più abituati ad un'interfaccia (anche se così non ci sarà possibile evolvere il tipo più di tanto):

```
public interface TigerNewFeature {
    int ANNOTATIONS = 0;
    int AUTOBOXING = 1;
    int ENUMERATIONS = 2;
    int FOREACH = 3;
    int FORMATTING = 4;
    int GENERICS = 5;
    int STATIC_IMPORTS = 6;
    int VARARGS = 7;
}
```

Ora la variabile `ArrayList` di `ProgrammatoreJava` sarà riscritta per essere parametrizzata con `Integer`, come segue:

```
ArrayList <Integer> aggiornamenti;
```

e i metodi `aggiungiAggiornamento()` e `rimuoviAggiornamento()`, avranno come parametri degli interi:

```
public void aggiungiAggiornamento(Integer aggiornamento){
    aggiornamenti.add(aggiornamento);
}
public void rimuoviAggiornamento(Integer aggiornamento){
    aggiornamenti.remove(aggiornamento);
}
```

Dando per scontato che l'autoboxing e auto-unboxing (cfr. Unità Didattica 16.1), eseguirà per noi tutte le operazioni di conversioni dal tipo primitivi `int` al relativo tipo wrapper `Integer`, ovviamente la classe `TestEnum1`, non si dovrà modificare e funzionerà correttamente. Il problema è che anche le seguenti istruzioni sono valide:

```
ProgrammatoreJava pro = new ProgrammatoreJava("Pippo");
pro.aggiungiAggiornamento(0);
pro.aggiungiAggiornamento(1);
pro.aggiungiAggiornamento(2);
pro.aggiungiAggiornamento(3);
pro.aggiungiAggiornamento(4);
pro.aggiungiAggiornamento(5);
pro.aggiungiAggiornamento(6);
pro.aggiungiAggiornamento(7);
```

Ma non c'è modo di sapere se l'utilizzo dei valori delle costanti è stato fatto consapevolmente o meno. Supponiamo poi che sia definita anche la seguente interfaccia:

```
public interface CSharpFeature {
    int DELEGATES = 0;
    int UNSAFE_CODE = 1;
    int  ENUMERATIONS = 2;
    int ADO_DOT_NET = 3;
    int WINDOWS_FORMS = 4;
    int DATA_STREAMS = 5;
    int  REFLECTION = 6;
    int COM_PLUS = 7;
}
```

a questo punto anche il seguente codice sarà valido:

```
ProgrammatoreJava pro = new ProgrammatoreJava("Pippo");
pro.aggiungiAggiornamento(CSharpFeature.DELEGATES);
pro.aggiungiAggiornamento(CSharpFeature.UNSAFE_CODE);
pro.aggiungiAggiornamento(CSharpFeature.ENUMERATIONS);
pro.aggiungiAggiornamento(CSharpFeature.ADO_DOT_NET);
pro.aggiungiAggiornamento(CSharpFeature.WINDOWS_FORMS);
pro.aggiungiAggiornamento(CSharpFeature.DATA_STREAMS);
pro.aggiungiAggiornamento(CSharpFeature.REFLECTION);
pro.aggiungiAggiornamento(CSharpFeature.COM_PLUS);
```

e sarà impossibile distinguere le caratteristiche dei due linguaggi.

17.2.2 Proprietà delle enumerazioni

Le enumerazioni sono trasformate in classi dal compilatore
In particolare, ogni enum estende implicitamente la nuova classe astratta `java.lang.Enum` (che non è un'enumerazione). E possono usufruire o overridare i metodi di `Enum`.

> **Attenzione che il compilatore non permetterà allo sviluppatore di creare classi che estendono direttamente la classe Enum. Enum è una classe speciale creata appositamente per supportare il concetto di enumerazione.**

Un'enumerazione non può estendere altre enumerazioni né altre classi, ma può implementare interfacce
Infatti, se dovesse estendere un'altra enumerazione o un'altra classe, il compilatore non potrebbe fare estendere ad essa la classe `java.lang.Enum`, per le regole dell'ereditarietà singola. Ovviamente invece, è lecito per un'enum implementare interfacce.

Gli elementi di un'enumerazione sono istanze dell'enumerazione stessa
Essendo istanze, subiscono controlli sul tipo in fase di compilazione. È questa la ragione per cui non sono paragonabili alle solite costanti intere.

Gli elementi di un'enumerazione sono implicitamente public, static e final
Si tratta quindi di costanti statiche il cui valore non è possibile cambiare.

Un'enumerazione è implicitamente dichiarata final
Questo implica che non è possibile estendere un'enum.

> **Esiste un caso in cui le enum non sono `final`: quando hanno dei metodi specifici per gli elementi (Cfr. paragrafo relativo).**

Le enumerazioni non possono dichiarare costruttori public
Questo non permette di creare al runtime nuove istanze di enum che non siano state definite in fase di compilazione. Quindi è possibile usufruire solamente di istanze definite dalla stessa enumerazione.

La classe java.lang.Enum
La classe `Enum` è così dichiarata:

```
package java.lang;
public class Enum<E extends Enum<E>> implements
Comparable<E>, Serializable {
  protected Enum(String name, int ordinal){. . .}
  protected Object clone( ) {. . .}
  public int compareTo(E o) {. . .}
  public boolean equals(Object other) {. . .}
  public Class<E> getDeclaringClass( ) {. . .}
  public int hashCode( ) {. . .}
  public String name( ) {. . .}
  public int ordinal( ) {. . .}
  public String toString( ) {. . .}
  public static <T extends Enum<T>> T valueOf(Class<T>
      enumType, String name) {. . .}
}
```

Analizziamola con calma. Per prima cosa possiamo notare che implementa `Comparable` e `Serializable`. Ciò significa che è possibile utilizzare il metodo `compareTo()`, per ordinare enum, e che tutte le enumerazioni sono serializzabili (cfr. Modulo 13).
È definito anche il metodo `equals()`, che si può quindi utilizzare per confrontare più valori.

Questo metodo è più che altro utile alle nuove collection di enum: `EnumMap` e `EnumSet`. Si tratta di collection specifiche per gestire enumerazioni, molto utili ma anche dall'utilizzo non molto intuitivo (cfr. Documentazione ufficiale).

`Enum` fa override del metodo `toString()`, e questo lo si poteva già notare dall'esempio sopra riportato. Per esempio `TigerNewFeature.ANNOTATIONS.toString()`, ritorna la stringa "`ANNOTATIONS`". È comunque possibile fare override di questo metodo.
Enum dichiara anche un metodo complementare a `toString()`: il metodo statico `valueOf()`. Per esempio `TigerNewFeature.valueOf("ANNOTATIONS")`

ritorna `TigerNewFeature.ANNOTATIONS`.
`toString()` e `valueOf()` sono complementari nel senso che, se facciamo override di uno dei metodi, dovremmo fare override anche dell'altro.
`Enum` definisce il metodo `final ordinal()`. Tale metodo ritorna la posizione all'interno dell'enum di un suo elemento. Come al solito l'indice parte da zero. Anche questo metodo dovrebbe essere utilizzato più che altro dalle collection di enum per particolari metodi.
Le enumerazioni inoltre definiscono il metodo `values()`. Questo metodo permette di iterare sui valori di un'enumerazione. Tale pratica potrebbe servire quando non conosciamo l'enumerazione in questione, un po'come quando si utilizza la reflection per capire il contenuto di una classe. Per esempio, sfruttando un ciclo foreach, possiamo stampare i contenuti dell'enumerazione `TigerNewFeature`:

```
for (TigerNewFeature t : TigerNewFeature.values()) {
    System.out.println(t);
}
```

17.2.3 Caratteristiche avanzate di un'enumerazione

Sino ad ora ci siamo fatti un'idea di cosa sia un'enum e a cosa serve. Ma esistono ancora tanti altri utilizzi di un'enum, e tante altre precisazioni da fare. Per esempio è possibile creare costruttori privati, metodi interni, metodi specifici per ogni valore, enumerazioni innestate etc…

Enumerazioni innestate (in classi) o Enumerazioni membro

L'argomento classi innestate, è stato trattato in due moduli in particolare: il modulo 8 (caratteristiche avanzate del linguaggio) e il modulo 15 (applicazioni grafiche). Anche le enumerazioni si possono innestare nelle classi (ma non in altre enumerazioni), ed è una pratica molto utile, che sarà utilizzata spesso in futuro. Sarà per esempio possibile scrivere codice come il seguente:

```
public class Volume {
    public enum Livello {ALTO, MEDIO, BASSO};
    // implementazione della classe . . .
}
```

ovviamente se volessimo stampare il metodo `toString()` di un elemento dell'enumerazione all'interno della classe è possibile utilizzare la sintassi:

```
  System.out.println(Livello.ALTO);
```

mentre se ci si trova al di fuori della classe, bisognerà utilizzare la sintassi:

```
  System.out.println(Volume.Livello.ALTO);
```

è anche possibile utilizzare l'incapsulamento, ma trattandosi di costanti statiche non si corrono grossi pericoli.
Una enum innestata è statica implicitamente. Infatti, il seguente codice è valido:

```
public class Volume {
    public enum Livello {ALTO, MEDIO, BASSO};
    // implementazione della classe . . .
    public static void main(String args[]) {
        System.out.println(Livello.ALTO);
    }
}
```

se `Livello` non fosse statica, non avremmo potuto utilizzarla direttamente in un metodo statico come il `main()`.

Su altri testi le enumerazioni innestate sono chiamate semplicemente enumerazioni membro (membro di una classe cfr. Modulo 2). Per non avviare una sterile discussione su come sia più corretto chiamare tale costrutto, affermiamo che "sono punti di vista".

Enumerazioni e metodi
È possibile aggiungere alle enumerazioni variabili, metodi e costruttori. Consideriamo la seguente versione rivisitata dell'enumerazione `TigerNewFeature`:

```
public enum TigerNewFeature {
    ENUMERATIONS(1), FOREACH(), ANNOTATIONS, GENERICS,
    AUTOBOXING, STATIC_IMPORTS, FORMATTING, VARARGS;
    private TigerNewFeature() {
    }
    private int ordinal;
```

```
    public void setOrdinal(int ordinal) {
        this.ordinal = ordinal;
    }
    public int getOrdinal() {
        return ordinal;
    }
    TigerNewFeature(int ordinal) {
        setOrdinal(ordinal);
    }
}
```

il precedente codice è corretto (anche se non ha un'utilità se non dal punto di vista didattico). Iniziamo ad analizzare il codice:

1. **Qualsiasi dichiarazione deve seguire la dichiarazione degli elementi dell'enumerazione**
   Se anteponessimo una qualsiasi delle dichiarazioni fatte alla lista degli elementi, otterremo un errore in compilazione.
   In questo caso la dichiarazione degli elementi deve terminare esplicitamente con un ";", ed è buona abitudine che sia sempre così.
2. **È possibile dichiarare un qualsiasi numero di costruttori, che implicitamente saranno considerati private**
   Nell'esempio abbiamo due costruttori, di cui uno abbiamo esplicitato (ma è ridondante) il modificatore private. Se avessimo dichiarato esplicitamente un costruttore public, avremmo ottenuto un errore in compilazione. Per il resto valgono le regole che valgono per i costruttori delle classi (cfr. Modulo 8).
3. **Quando sono esplicitati i costruttori come in questo caso, i valori dell'enum possono utilizzarli**
   Basta osservare il codice dell'esempio. Il valore `ENUMERATIONS(1)`, utilizza il costruttore che prende in input un intero. Tutti gli altri valori invece, utilizzano il costruttore senza parametri. In particolare è possibile come il valore `FOREACH` sia dichiarato in maniera diversa rispetto agli altri valori. Infatti, esplicita due parentesi vuote, che sottolineano come stia utilizzando il costruttore senza parametri.
   Tale sintassi è assolutamente superflua, ed è stata riportata solo per preparare il lettore a strane sorprese.
   Se nell'esempio avessimo avuto un unico costruttore (quello che prende come

parametro in input un intero, allora tutti gli elementi dell'enum avrebbero obbligatoriamente dovuto utilizzare quell'unico costruttore.

4. **I metodi e le variabili sono esattamente dichiarati come si fa nelle classi.** Non bisogna fare nessuna attenzione particolare a questi membri, se non come già asserito, a dichiararli dopo i valori dell'enum. È anche possibile ovverridare i metodi di un'enum in una classe. Infine, è possibile anche dichiarare un'enum in un'enum come segue:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  public enum MyEnum2 {a,b,c}
}
```

Enumerazioni e metodi specifici degli elementi

È in effetti in qualche modo possibile estendere un'enum? In un certo senso la risposta è sì. È possibile infatti che una certa enumerazione, sia estesa dai suoi stessi elementi. È infatti possibile definire dei metodi nell'enumerazione, e fare override di essi con i suoi elementi. Consideriamo la seguente versione della enumerazione `TigerNewFeature`:

```
public enum TigerNewFeature {
    ENUMERATIONS {
        public void metodo() {
            System.out.println("metodo di ENUMERATIONS");
        }
    },
    FOREACH, ANNOTATIONS, GENERICS, AUTOBOXING,
    STATIC_IMPORTS, FORMATTING, VARARGS;
    public void metodo() {
        System.out.println("metodo dell'enum");
    }
  }
```

è stato definito un metodo che abbiamo chiamato `metodo()`, che dovrebbe stampare la stringa "metodo dell'enum". Però, l'elemento `ENUMERATIONS`, con una sintassi simile a quella delle classi anonime (cfr. Modulo 8), dichiara anch'esso lo stesso metodo, overridandolo in qualche modo. Infatti, il compilatore tramuterà `ENUMERATIONS` proprio in una classe anonima, che estenderà `TigerNewFeature`. Quindi l'istruzione:

```
TigerNewFeature.ENUMERATIONS.metodo();
```

stamperà:

metodo di ENUMERATIONS

mentre l'istruzione:

```
TigerNewFeature.VARARGS.metodo();
```

stamperà:

metodo dell'enum

perché `VARARGS` non ha fatto override di `metodo()`.
Per poter invocare dall'esterno di `TigerNewFeature` il metodo `metodo()`, è necessario che venga dichiarato anche da `TigerNewFeature` stessa. Altrimenti il compilatore segnalerebbe un errore per un'istruzione come la seguente:

```
TigerNewFeature.ENUMERATIONS.metodo();
```

In effetti per il compilatore il metodo `metodo()`, in questo caso semplicemente non esiste. Per tale ragione, potrebbe essere anche dichiarato astratto. Questo però obbligherebbe tutti gli elementi a ridefinirlo.

17.2.4 Impatto su Java

Nuova parola chiave enum
L'impatto che l'introduzione delle enumerazioni ha su Java, è notevole. Tanto per cominciare, l'introduzione della nuova parola chiave `enum`, comporta necessariamente delle attenzioni particolari. Se per esempio volete compilare con un JDK 1.5 o superiore, il vostro vecchio codice, dovete stare attenti. È per esempio usanza comune utilizzare `enum` come reference per dichiarare una `Enumeration` (interfaccia del framework Collections del package `java.util`). A questo punto avete da fare una scelta:

1. compilare con il flag "–source 1.4"
2. rivisitare il vostro codice per eliminare i reference non più validi.

Nel secondo caso però, probabilmente dovrete anche fare i conti con i lint warning dei generics (cfr. Unità Didattica 16.2), e la rivisitazione potrebbe essere anche molto impegnativa…

Costrutti

Come già asserito precedentemente, con i vari cicli di Java ora, è possibile ciclare sui valori di un'enumerazione, grazie al metodo `values()`. Abbiamo già visto un esempio del ciclo foreach, nel paragrafo precedente.
Il costrutto più "colpito" dall'introduzione delle enumerazioni, è però sicuramente il costrutto `switch`. Infatti, dopo l'impatto dell'Autoboxong e Auto-Unboxing (cfr. Unità Didattica16.1), anche le enumerazioni allargano il numero di tipi che la variabile di test del costrutto può accettare. Infatti, se un costrutto `switch` definisce come variabile di test un'enumerazione, allora tutte le istanze di tale enumerazione, possono essere possibili costanti per i case. Per esempio, tenendo presente l'enumerazione `Livello` di cui sopra, consideriamo il seguente frammento di codice:

```
switch (getLivello()){
    case ALTO:
        System.out.println(Livello.ALTO);
    break;
    case MEDIO:
        System.out.println(Livello.MEDIO);
    break;
    case BASSO:
        System.out.println(Livello.BASSO);
    break;
}
```

la variabile di test è di tipo `Livello`, e le costanti dei case sono gli elementi dell'enumerazione stessa. Notare come gli elementi non abbiano bisogno di essere referenziati con il nome dell'enumerazione in seguente modo:

```
case Livello.ALTO:
    System.out.println(Livello.ALTO);
    break;
case Livello.MEDIO:
    System.out.println(Livello.MEDIO);
```

```
    break;
case Livello.BASSO:
    System.out.println(Livello.BASSO);
    break;
```

infatti, la variabile di test, fornisce già la sicurezza del tipo.
Nonostante in un costrutto `switch` che si basa su un'enum, è escluso che la clausola `default` possa essere eseguita durante il runtime (questo è uno dei vantaggi delle enum rispetto ai "vecchi" approcci), è comunque buona norma utilizzarne una. Infatti, è facile che l'enumerazione subisca nel tempo delle aggiunte. Questo è particolarmente vero se il codice è condiviso tra più programmatori. In tal caso ci sono due approcci da consigliare. Il primo è più "soft", e consiste nel gestire comunque eventuali nuovi tipi in maniera generica. Per esempio:

```
default:
     System.out.println(getLivello());
```

Il secondo metodo è senz'altro più robusto, perché basato sulle asserzioni. Per esempio

```
default:
     assert false: "valore dell'enumerazione nuovo: " +
getLivello();
```

Notiamo che l'asserzione dovrebbe fare emergere il problema durante i test, se l'enumerazione `Livello` è stata ampliata con nuovi valori.
Nel caso che non si voglia comunque implementare la clausola `default`, possiamo comunque farci aiutare del compilatore nel caso che l'enumerazione coinvolta nello `switch` si evolva. Infatti, se in uno `switch` non vengono contemplati tutti i case dell'enumerazione, per esempio:

```
case ALTO:
    System.out.println(Livello.ALTO);
break;
case BASSO:
    System.out.println(Livello.BASSO);
break;
```

allora in compilazione il compilatore ci segnalerà un warning.

Interfacce
Ora le interfacce possono dichiarare oltre che metodi implicitamente astratti e variabili implicitamente `static`, `final` e `public`, anche enumerazioni essendo queste implicitamente `public`, `static` e `final`. Il seguente codice quindi, è valido:

```
public interface ProvaEnum {
    enum Prova {UNO, DUE, TRE};
}
```

Riepilogo

In questo modulo dopo aver definito la sintassi del nuovo ciclo for migliorato, ne abbiamo sottolineato i limiti. Abbiamo anche visto come il ciclo sia applicabile a qualsiasi oggetto che implementi l'interfaccia `Iterable`. Inoltre, nel paragrafo dedicato agli impatti su Java, abbiamo analizzato il nuovo costrutto, ma senza troppo entusiasmo…
La seconda parte del modulo, è sicuramente molto più interessante. Infatti, abbiamo definito le enumerazioni con il supporto di semplici esempi. Dopo aver presentato le ragioni per cui le enumerazioni costituiscono un vero punto di forza di Java, ne abbiamo presentato le proprietà. Inoltre, abbiamo cercato di capire che lavoro svolge il compilatore per noi. Abbiamo infatti notato come tutte le enumerazioni infine estendano la classe `java.lang.Enum`, ereditandone i metodi. Dopo aver presentato anche le caratteristiche avanzate delle enumerazioni come per esempio i metodi specifici degli elementi, abbiamo analizzato gli impatti sul linguaggio. In questo paragrafo, abbiamo visto come costrutti come lo `switch` e il `for` possano far uso delle enumerazioni, e come la nuova parola chiave possa introdurre problemi di compatibilità all'indietro, con codice pre-Java 5. Infine, abbiamo notato come le interfacce ora possano dichiarare oltre a metodi astratti e a costanti `public`, `static` e `final`, anche enumerazioni.

Esercizi modulo 17

Esercizio 17.a)
Ciclo for migliorato ed enumerazioni, Vero o Falso:

1. Il ciclo for migliorato può in ogni caso sostituire un ciclo `for`
2. Il ciclo for migliorato Il ciclo for migliorato può essere utilizzato con gli array, e con le classi che implementano `Iterable`.
3. Il ciclo for migliorato sostituisce l'utilizzo di `Iterator`
4. Il ciclo for migliorato non può sfruttare correttamente i metodi di `Iterator`
5. In un ciclo for migliorato non è possibile ciclare all'indietro
6. La classe `java.lang.Enum` implementa `Iterable`, altrimenti non sarebbe possibile utilizzare il ciclo for migliorato con le enumerazioni
7. Il seguente codice viene compilato senza errori:

```
Vector <Integer> integers = new Vector<Integer>();
. . .
for (int i : integers) {
    System.out.println(i);
}
```

8. Il seguente codice viene compilato senza errori:

```
  int i = new int[100];
  int j = new int[100];

  . . .
  for (int index1, int index2 : i, j) {
    System.out.println(i+j);
  }
```

9. Il seguente codice viene compilato senza errori:

```
    Vector <Integer> i = new <Integer>Vector;
    Vector <Integer> j = new <Integer>Vector;

    . . .
    for (int index1, int index2 : i, j) {
      System.out.println(i+j);
    }
```

10. Facendo riferimento all'enumerazione definite in questo modulo, il seguente codice viene compilato senza errori

```
for (TigerNewFeature t : TigerNewFeature.values()) {
  System.out.println(t);
}
```

Esercizio 17.b)
Enumerazioni, Vero o Falso:

1. Le enumerazioni, non si possono istanziare, se non all'interno della definizione dell'enumerazione stessa. Infatti, possono avere solamente costruttori `private`
2. Le enumerazioni possono dichiarare metodi, possono essere estese da classi che possono ovveridarne i metodi. Non è però possibile che un'enum estenda un'altra enum
3. Il metodo `values()` appartiene ad ogni enumerazione ma non alla classe `java.lang.Enum`
4. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
    public void metodo1() {

    }
    public void metodo2() {

    }
    ENUM1, ENUM2;
}
```

5. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 {
    public void metodo() {

    }
  }, ENUM2;
  public void metodo2() {

  }
}
```

6. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  private MyEnum(int i) {

  }
}
```

7. Il seguente codice viene compilato senza errori:

```
public class Volume {
    public enum Livello {
        ALTO, MEDIO, BASSO
 } ;
// implementazione della classe . . .
  public static void main(String args[]) {
    switch (getLivello()) {
      case ALTO:
      System.out.println(Livello.ALTO);
      break;
      case MEDIO:
      System.out.println(Livello.MEDIO);
      break;
      case BASSO:
      System.out.println(Livello.BASSO);
      break;
    }
  }
  public static Livello getLivello() {
    return Livello.ALTO;
  }
}
```

8. Se dichiariamo la seguente enumerazione:

```
public enum MyEnum {
  ENUM1 {
    public void metodo1() {

    }
  },
  ENUM2 {
    public void metodo2() {
```

```
    }
  }
}
```

il seguente codice potrebbe essere correttamente compilato:

```
MyEnum.ENUM1.metodo1();
```

9. Non è possibile dichiarare enumerazioni con un unico elemento
10. Si possono innestare enumerazioni in enumerazioni in questo modo

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  public enum MyEnum2 {a,b,c}
}
```

ed il seguente codice viene compilato senza errori:

```
System.out.println(MyEnum.MyEnum2.a);
```

Soluzioni esercizi modulo 17

Esercizio 17.a)
Ciclo for migliorato ed enumerazioni, Vero o Falso:

1. **Falso**
2. **Vero**
3. **Vero**
4. **Vero**
5. **Vero**
6. **Falso**
7. **Vero**
8. **Falso**
9. **Falso**
10. **Vero**

Esercizio 17.b)
Enumerazioni, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso**
4. **Falso**
5. **Vero**
6. **Falso** non è possibile utilizzare il costruttore di default se ne viene dichiarato uno esplicitamente
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper utilizzare il ciclo for-migliorato (unità 17.1)	☐	
Comprendere i limiti e quando applicare il ciclo for migliorato (unità 17.1)	☐	
Comprendere e saper utilizzare le enumerazioni (unità 17.2)	☐	
Comprendere le caratteristiche avanzate e quando utilizzare le enumerazioni (unità 17.2)	☐	

Note:

18

Complessità: bassa

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Saper utilizzare i varargs e comprenderne le proprietà (unità 18.1) .
2. Saper utilizzare gli static imports e comprenderne le conseguenze del loro utilizzo (unità 18.2)

18 Static imports e Varargs

Entrambi gli argomenti principali di questo modulo non dovrebbero presentare grossi problemi per il lettore.

18.1 Varargs

L'overload (cfr. Modulo 6) è la caratteristica polimorfica di Java basata sul concetto che, un metodo è univocamente individuato dalla sua firma (ovvero dal nome e dalla lista dei parametri). Si tratta di una caratteristica molto importante, perché permette allo sviluppatore di utilizzare lo stesso nome per più metodi, variando la lista degli argomenti. L'utilizzo è molto semplice e le regole molto chiare.
Ci sono dei casi in cui però l'overload non basta a soddisfare pienamente le esigenze di uno sviluppatore. Stiamo parlando del caso in cui ci sia bisogno di una lista di argomenti variabile in numero. In tali casi la soluzione pre - Java 5, era costituita da array e collezioni. Supponiamo di voler evolvere la classe `ProgrammatoreJava`, già definita nell'unità didattica relativa alle enumerazioni. Ricordiamo che la classe in questione definiva in particolare due metodi chiamati `aggiungiAggiornamento(TigerNewFeature aggiornamento)` e `rimuoviAggiornamento(TigerNewFeature aggiornamento)`, dove `TigerNewFeature` era un'enumerazione che definiva con i suoi elementi le nuove feature introdotte da Java 5. Ora, supponiamo di voler introdurre due metodi equivalenti a questi, che però devono permettere al programmatore Java, di aggiornarsi su più feature contemporaneamente. Per realizzare i nostri intenti, possiamo per esempio utilizzare un array, come mostra il codice seguente:

```
import java.util.ArrayList;
```

```
public class ProgrammatoreJava extends Programmatore {

  ArrayList <TigerNewFeature> aggiornamenti;

  public ProgrammatoreJava () {
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
  public ProgrammatoreJava (String nome) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
  public ProgrammatoreJava (String nome,
      TigerNewFeature[] features) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
    aggiungiAggiornamenti(features);
  }
  public void aggiungiAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.add(aggiornamento);
  }
  public void rimuoviAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.remove(aggiornamento);
  }
  public void aggiungiAggiornamenti(
      TigerNewFeature[] features) {
    for (TigerNewFeature aggiornamento : features) {
      aggiornamenti.add(aggiornamento);
    }
  }
  public void rimuoviAggiornamenti(
      TigerNewFeature[] features) {
    for (TigerNewFeature aggiornamento : features) {
      aggiornamenti.remove(aggiornamento);
    }
  }
  public String toString() {
    StringBuilder sb = new StringBuilder(getNome());
```

```
    sb.append(" è aggiornato a :" + aggiornamenti);
    return sb.toString();
  }
}
```

In questo caso, dato che il numero degli argomenti del costruttore poteva essere di grandezza variabile, si è preferito creare un costruttore che prende in input un array di `TigerNewFeature`, piuttosto che creare tanti costruttori, ognuno che aggiungeva un nuovo parametro al precedente. Ovviamente sarebbe possibile anche utilizzare una collection come `ArrayList`, nel caso la lista si debba evolvere.
Per poter creare un programmatore Java già aggiornato alle feature `VARARGS`, `FOREACH`, `ENUMERATIONS` e `GENERICS`, è possibile utilizzare la seguente istruzione:

```
Programmatore pro = new ProgrammatoreJava("Pippo", new
TigerNewFeature[]{
      TigerNewFeature.VARARGS, TigerNewFeature.FOREACH,
    TigerNewFeature.ENUMERATIONS,TigerNewFeature.GENERICS
    });
```

Notare che viene definito al volo un array come secondo parametro del costruttore. Nessun problema nel fare questo, ma sicuramente la sintassi precedente non è molto leggibile.
Esiste ora una nuova sintassi definita da i cosiddetti **varargs**, abbreviativo di **variable arguments** (argomenti variabili). La sintassi fa uso di un'**ellissi** (nel senso di omissione) costituita da tre puntini sospensivi ("..."), che seguono il tipo di dato, di cui non si conoscono la quantità degli argomenti. Segue la classe `ProgrammatoreJava` rivisitata con i varargs:

```
import java.util.ArrayList;
public class ProgrammatoreJava extends Programmatore {
  ArrayList <TigerNewFeature> aggiornamenti;
  public ProgrammatoreJava () {
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
  public ProgrammatoreJava (String nome) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
  }
```

```
  public ProgrammatoreJava (String nome,
      TigerNewFeature… features) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
    aggiungiAggiornamenti(features);
  }
  public void aggiungiAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.add(aggiornamento);
  }
  public void rimuoviAggiornamento(
      TigerNewFeature aggiornamento){
    aggiornamenti.remove(aggiornamento);
  }
  public void aggiungiAggiornamenti(
      TigerNewFeature… features) {
    for (TigerNewFeature aggiornamento : features) {
      aggiornamenti.add(aggiornamento);
    }
  }
  public void rimuoviAggiornamenti(
      TigerNewFeature… features) {
    for (TigerNewFeature aggiornamento : features) {
      aggiornamenti.remove(aggiornamento);
    }
  }
  public String toString() {
    StringBuilder sb = new StringBuilder(getNome());
    sb.append(" è aggiornato a :" + aggiornamenti);
    return sb.toString();
  }
}
```

Come è possibile notare questa classe è identica alla precedente, tranne per il fatto che sostituisce con delle ellissi ("…") le parentesi degli array ("`[]`").

Effettivamente i varargs, all'interno del metodo dove sono dichiarati, sono considerati a tutti gli effetti degli array. Quindi, come per gli array, ci si può

ricavare la dimensione con la variabile `length`, e ciclare su di essi. Nell'esempio abbiamo sfruttato il nuovo costrutto `foreach`. (cfr. Modulo 17.1).

Il vantaggio di avere varargs in luogo di un array o di una collection, risiede essenzialmente nel fatto che per chiamare un metodo che dichiara argomenti variabili, non bisogna creare array o collection. Un metodo con varargs, viene semplicemente invocato come si fa con un qualsiasi overload. Per esempio, ora per istanziare il programmatore aggiornato alle quattro feature di cui sopra, è possibile scrivere:

```
  Programmatore pro = new ProgrammatoreJava("Pippo",
      TigerNewFeature.VARARGS, TigerNewFeature.FOREACH,
      TigerNewFeature.ENUMUMERATIONS,
TigerNewFeature.GENERICS);
```

che è un modo molto più naturale di passare parametri.
Ma anche le seguenti istruzioni sono tutte valide:

```
  ProgrammatoreJava pro = new ProgrammatoreJava("Pippo",
    TigerNewFeature.GENERICS);
  pro.aggiungiAggiornamenti(TigerNewFeature.FOREACH,
    TigerNewFeature.STATIC_IMPORTS);
  pro.aggiungiAggiornamenti(TigerNewFeature.VARARGS,
    TigerNewFeature.ENUMERATIONS,
TigerNewFeature.ANNOTATIONS);
```

In pratica è come se esistesse un overload infinito dei costruttori e dei metodi `aggiungiAggiornamenti()` e `rimuoviAggiornamenti()`. Per tale ragione, i metodi originari `aggiungiAggiornamento()` e `rimuoviAggiornamento()`, che permettevano di aggiungere un unico aggiornamento alla volta, sono semplicemente superflui, e potrebbero anche essere eliminati. I varargs quindi, possono essere considerati anche uno strumento per risparmiare righe di codice.

18.1.1 Proprietà dei varargs

Il significato dei varargs, va interpretato come "zero o più argomenti". Infatti, è possibile anche invocare il metodo senza passare argomenti. Per esempio, anche la seguente istruzione è perfettamente valida:

```
pro.aggiungiAggiornamenti();
```

Essendo questa istruzione inutile, sarebbe preferibile modificare il codice dei metodi `aggiungiAggiornamenti()` e `rimuoviAggiornamenti()`, in modo tale che gestiscano nel modo più corretto l'anomalia. Per esempio, una buona strategia, sarebbe quella di lanciare una `IllegalArgumentException`, come nel seguente esempio:

```
public void aggiungiAggiornamenti(TigerNewFeature…
features) {
    if (features.length == 0){
        throw new IllegalArgumentException("Nessun " +
          "aggiornamento specificato!");
    }
    for (TigerNewFeature aggiornamento : features) {
        aggiornamenti.add(aggiornamento);
    }
}
```

Essendo `IllegalArgumentException` una unchecked exception (sottoclasse di `RuntimeException`, cfr. Modulo 10), il metodo può evitare di dichiararla nella sua clausola `throws`. Ovviamente, lo sviluppatore può comunque irrobustire il codice con una clausola `throws` nel seguente modo:

```
public void aggiungiAggiornamenti(
  TigerNewFeature… features)
    throws IllegalArgumentException { . . .
```

Quest'ultima caratteristica dei varargs in compenso, rende del tutto inutile il costruttore:

```
public ProgrammatoreJava (String nome) {
    super(nome);
    aggiornamenti = new ArrayList<TigerNewFeature>();
}
```

infatti, eliminando tale costruttore sarà sempre possibile istanziare un programmatore Java con la seguente sintassi:

```
ProgrammatoreJava pro = new ProgrammatoreJava("Pippo");
```

In una firma di un metodo, non è possibile utilizzare una dichiarazione di varargs, se non come ultimo parametro. Per tale ragione non è neanche possibile dichiarare più di un varargs, per ogni metodo.

Quindi non è possibile dichiarare più varargs nello stesso metodo. Per esempio, il codice:

```
public void aggiungiAggiornamenti(TigerNewFeature...
features, int… h)
```

non verrà "capito" dal compilatore. Ma neanche la seguente istruzione risulterà compilabile:

```
public void aggiungiAggiornamenti(TigerNewFeature...
features, int h)
```

18.1.2 Impatto su Java

Flessibilità con il polimorfismo
L'avvento dei varargs su Java, dovrebbe essenzialmente, permettere di avere del codice meno complesso e flessibile. Inoltre, in molti casi, sarà possibile eliminare diverse righe di codice. Tenendo conto che è ovviamente valido il polimorfismo anche con i varargs, potremmo anche creare metodi ultra-generici come il seguente:

```
public void metodo(Object… o) { . . . }
```

Ovviamente, un metodo come il precedente può prendere in input non solo qualsiasi tipo di oggetto, ma anche un qualsiasi numero di oggetti. Tenendo anche conto che un tipo primitivo dalla versione 5 viene all'occorrenza convertito nel relativo tipo wrapper (cfr. Unità Didattica 16.1 relativo all'autoboxing e auto-unboxing), allora sarà possibile anche passare a tale metodo qualsiasi tipo primitivo. Ovviamente è possibile utilizzare come argomenti anche enumerazioni (cfr. Modulo 17), e tipi generici (cfr. Modulo 16). Inoltre, è anche possibile non passargli niente! Esistono altre possibilità?
In pratica, i varargs, ampliano in qualche modo la potenza del linguaggio, ma non bisognerà abusarne, come nell'ultimo esempio, bisogna ricordare che con il polimorfismo si possono fare cose più utili…(questi sono solo puntini sospensivi!)

Override
L'override funziona esattamente come dovrebbe funzionare: se la super classe `Programmatore` di `ProgrammatoreJava`, definisse il seguente metodo:

```
public void aggiungiAggiornamenti(TigerNewFeature
features)
```

il metodo :

```
public void aggiungiAggiornamenti(TigerNewFeature…
features)
```

di `ProgrammatoreJava` non ne rappresenterebbe un override, ma solo un overload.
Formattazioni di output
Uno degli esempi più riusciti di utilizzo dei varargs, è sicuramente il metodo `format()` della classe `java.util.Formatter`. Il metodo ha anche un overload che permette di specificare un `Locale`, per effettuare eventuali formattazioni basate sull'internazionalizzazione (cfr. Modulo 12). Seguono le dichiarazioni del metodo `format()`:

```
public Formatter format(String format, Object... args)
public Formatter format(Locale l, String format,
Object... args)
```

Il metodo `format()` serve per stampare un output, con la possibilità di formattare correttamente tutti gli input che gli vengono passati. Consideriamo il seguente esempio:

```
  Formatter formatter = new Formatter(System.out);
  formatter.format(Locale.ITALY, "e = %+10.4f", Math.E);
```

Stamperà il seguente output:

```
e =    +2,7183
```

In pratica, con la sintassi:

```
  %+10.4f
```

abbiamo specificato secondo la sintassi di formattazione definita dalla classe `Formatter` il valore `double` della costante `E` della classe `Math`. In particolare con il simbolo "`%`" avvertiamo il formatter che stiamo definendo un valore da formattare, che in questo caso è il primo (e l'unico) valore del parametro varargs, ovvero `Math.E`. Con il simbolo "+" invece, abbiamo specificato che l'output deve obbligatoriamente specificare il segno (positivo o negativo che sia). La "`f`" finale serve per specificare che l'output deve essere formattato come numero decimale (floating point). Il numero "`10`" che precede il "`.`", fa in modo che l'output sia formattato in almeno 10 posizioni. Se l'output è costituito da meno di 10 posizioni come nel nostro caso, allora il valore viene emesso in 10 caratteri, allineato a destra, come è possibile notare dall'esempio. Infine, il numero "`4`" specificato dopo il "`.`", specifica che il valore deve essere emesso con 4 decimali (con eventuale arrotondamento).
La classe offre la possibilità di formattare gli output con un numero enorme di varianti (cfr. Documentazione classe `Formatter`). L'argomento sembra piuttosto complesso… ma esiste una categoria di programmatori, che non sarebbe d'accordo: i programmatori C/C++. Essi infatti, hanno dovuto imparare ben presto le regole che governano la formattazione degli output. Infatti, la principale funzione di stampa del C, si chiama `printf()`, e formatta gli output proprio con le regole a cui abbiamo appena accennato. Ma le novità non sono finite qui! Infatti, Tiger introduce nella classe `PrintStream`, un metodo chiamato proprio `printf()`. Questo metodo ricalca la firma del metodo `format()` di `Formatter`, sfruttando come secondo parametro un varargs di `Object`. Infatti, `printf()`, non fa altro che chiamare a sua volta il metodo `format()`. Ma qual è l'oggetto di tipo `PrintStream` più famoso? Ovviamente `System.out`. Da ora in poi sarà possibile quindi utilizzare la seguente sintassi per stampare:

```
System.out.printf("Data %d", new Date());
```

Quindi siamo di fronte ad un modo "più familiare" (per i programmatori che vengono dal C), di utilizzare il metodo `format()`. Anche questa è una di quelle caratteristiche che dovrebbe favorire la migrazione dei programmatori C a Java. Dieci anni di evoluzione per tornare ad un'istruzione degli anni '70…continuiamo così…
Come al solito il lettore può avere ogni dettaglio consultando la documentazione ufficiale.

Se come parametro varargs del metodo `format()` viene passato un array di `Object`, ci troveremo davanti ad una situazione imprevista. Consideriamo il seguente esempio:

```
Object [] o = {"Java 1.5","Java 5","Tiger"};
System.out.printf("%s", o);
```

l'output del precedente codice sarà:

```
Java 1.5
```

Infatti, il formatter, considererà come elemento del varargs, non l'array di `Object` in quanto oggetto da formattare, bensì come array di `Object`! Quindi considererà parte del parametro varargs, tutti i singoli elementi dell'array `o`.
Per far sì che l'output sia formattato come previsto, e che l'array di `Object` sia considerato come singolo oggetto, è possibile utilizzare la seguente sintassi:

```
System.out.printf("%s", new Object[]{ o });
```

o equivalentemente:

```
System.out.printf("%s", (Object)o);
```

18.2 Static import

Riteniamo questa nuova feature del linguaggio la più evitabile delle novità. A parte la nostra personale avversione all'utilizzo della parola chiave `static` (cfr. modulo 9), l'utilità di questa nuova caratteristica è limitata a poche situazioni. In compenso l'interpretazione errata dell'utilizzo di questo meccanismo da parte di un programmatore, può facilmente rendere le cose più complicate!
Gli import statici permettono al programmatore di importare solo ciò che è statico all'interno di una classe. La sintassi è:

```
import static nomePackage.nomeClasse.nomeMembroStatico
```

È consigliabile utilizzare gli import statici quando si è interessati solo ai membri statici di una certa classe, ma non alle sue istanze. Il beneficio immediato che riceve il programmatore, è quello di poter utilizzare i membri statici importati staticamente, senza referenziarli con il nome della classe. Per esempio:

```
import static java.lang.System.out;
```

In questo modo per esempio, sarà possibile scrivere all'interno del nostro codice:

```
out.println("Ciao");
```

in luogo di:

```
System.out.println("Ciao");
```

Nel caso in cui nel nostro file non si dichiarino o importino staticamente altre variabili `out`, non c'è nessun problema tecnico che ci impedisce di utilizzare questa nuova caratteristica. Ovviamente potremo così scrivere molto meno codice noioso e ripetitivo come `System.out`.

Ovviamente, l'utilità di tale feature è reale, se e solo se il membro statico importato staticamente è utilizzato più volte all'interno del codice.

È possibile importare anche tutti i membri statici di una classe utilizzando il solito simbolo di asterisco. Per esempio, con il seguente codice:

```
import static java.lang.Math.*;
```

abbiamo importato tutti i membri statici della classe `Math`. Quindi, all'interno del nostro codice, potremo chiamare i vari metodi statici di `Math`, senza referenziarli con il nome della classe che li contiene. Per esempio, supponendo che `x` ed `y` siano le coordinate di un punto bidimensionale, il calcolo della distanza del punto dall'origine, prima dell'avvento degli import statici si poteva codificare nel seguente modo:

```
Math.sqrt(Math.pow(x,2) + Math.pow(y,2));
```

Con Tiger, è possibile ottenere lo stesso risultato con minor sforzo:

```
sqrt(pow(x,2) + pow(y,2));
```

È possibile anche importare staticamente solamente metodi statici. In tal caso vengono specificati solo i nomi di tali metodi, e non l'eventuale lista di argomenti. Per esempio:

```
import static java.sql.DriverManager.getConnection();
```

È anche possibile importare staticamente classi innestate e classi anonime (cfr. modulo 8). Ovviamente, questo non è possibile se queste sono dichiarate all'interno di metodi. Per esempio è possibile importare la classe innestata statica `LookAndFeelInfo` della classe `UIManager` (una classe che contiene informazione sul look and feel di un'applicazione swing, cfr. modulo 15), con la seguente sintassi:

```
import static javax.swing.UIManager.LookAndFeelInfo;
```

In realtà in questo caso, trattandosi di una classe innestata statica, e quindi referenziabile con il nome della classe che la contiene, sarebbe possibile importarla anche nella maniera tradizionale con la seguente sintassi:

```
import javax.swing.UIManager.LookAndFeelInfo;
```

L'import statico però, evidenzia il fatto che la classe innestata sia statica, e forse, in tali situazioni, il suo utilizzo è più appropriato.

18.2.1 Un parere personale

La domanda è: "siamo sicuri che l'utilizzo degli import statici rappresenti davvero un vantaggio?".
La risposta è "dipende"!
Gli stili e "i gusti" della programmazione sono soggettivi. C'è chi trova utile evitare di referenziare le variabili statiche perchè scrive meno codice, e trova questa pratica sufficientemente espressiva. Come appare evidente, personalmente preferisco evitare l'utilizzo degli import statici. Come più volte sottolineato in questo testo, preferiamo un nome lungo ed esplicativo ad uno breve ed ambiguo.
Tuttavia esistono alcune situazioni però, che ne giustificano pienamente l'utilizzo.
Un caso dove, l'uso degli import statici risulta effettivamente utile, è relativo all'utilizzo delle enumerazioni. Per esempio, consideriamo l'enum `TigerNewFeature`, definita nell'unità didattica relativa all'enumerazioni e che riportiamo nuovamente di seguito:

```
public enum TigerNewFeature {
  ANNOTATIONS, AUTOBOXING, ENUMERATIONS, FOREACH,
  FORMATTING, GENERICS, STATIC_IMPORTS, VARARGS
}
```

Inoltre consideriamo il codice di esempio utilizzato nell'unità didattica relativa ai varargs, dove utilizzavamo il seguente metodo della classe `ProgrammatoreJava`:

```
public void aggiungiAggiornamenti(TigerNewFeature…
features) {
    for (TigerNewFeature aggiornamento : features) {
        aggiornamenti.add(aggiornamento);
    }
}
```

Per aggiungere più feature dell'enumerazione `TigerNewFeature` sfruttando il metodo `aggiungiAggiornamenti(TigerNewFeature… features)`, avevamo dovuto utilizzare il seguente codice:

```
  ProgrammatoreJava pro = new ProgrammatoreJava("Pippo",
      TigerNewFeature.VARARGS, TigerNewFeature.FOREACH,
      TigerNewFeature.ENUMERATIONS,
TigerNewFeature.GENERICS);
```

In questo caso l'utilizzo di un import statico, è senz'altro appropriato:

```
import static nomePackage.TigerNewFeature.*;
. . .
ProgrammatoreJava pro = new ProgrammatoreJava("Pippo",
      VARARGS, FOREACH, ENUMERATIONS, GENERICS);
```

infatti, abbiamo evitato inutili ripetizioni e snellito il codice. Contemporaneamente, la leggibilità non sembra essere peggiorata.
Un altra situazione dove è pienamente giustificato l'utilizzo degli import statici, è prettamente legata al concetto di astrazione (cfr. modulo 5). Spesso capita di avere a disposizione un'interfaccia che definisce diverse costanti statiche. Per esempio consideriamo la seguente interfaccia:

```
package applicazione.db.utility;
public interface ConstantiSQL {
    String GET_ALL_USERS = "SELECT * FROM USERS";
    String GET_USER = "SELECT * FROM USERS WHERE ID = ?";
    // Altre costanti...
}
```

Questa interfaccia può aver un senso in alcuni contesti. Infatti, essa definisce delle costanti di tipo stringa che contengono tutti i comandi SQL che una certa applicazione definisce. Così si favorisce il riuso di tali comandi da varie classi.
Ora supponiamo di dover creare una classe che utilizza ripetutamente le costanti di tale interfaccia. La soluzione solitamente più utilizzata in questi casi, è quella di implementare tale interfaccia:

```
package applicazione.db.logic;
import applicazione.db.utility.*;
import java.sql.*;
public class GestoreDB implements CostantiSQL {
    public Collection getUsers() {
        ...
        ResultSet rs = statement.execute(GET_ALL_USERS);
        ...
    }
    // Altri metodi ...
}
```

Tuttavia, la soluzione più corretta sarebbe quella di utilizzare l'interfaccia, non implementarla. Infatti, se la implementassimo, è come se stessimo dicendo che `GestoreDB` "è un" `CostantiSQL` (cfr. modulo 5, paragrafo sull'ereditarietà). La soluzione corretta potrebbe essere la seguente:

```
package applicazione.db.logic;
import applicazione.utility.db.*;
public class GestoreDB {
    public Collection getUsers() {
        ...
        ResultSet rs =
statement.execute(CostantiSQL.GET_ALL_USERS);
        ...
    }
    // Altri metodi ...
}
```

Considerando che non abbiamo riportato tutti i metodi (potrebbero essere decine), l'ultima soluzione ovviamente obbliga il programmatore a dover scrivere del codice un

po' troppo ripetitivo. Benché inesatta inoltre, la prima soluzione ha un notevole vantaggio programmatico, e un basso impatto di errore analitico. In fondo stiamo ereditando delle costanti statiche, non metodi concreti. Quindi la situazione è questa: seconda soluzione più corretta, prima soluzione più conveniente!
Bene, in questo caso gli import statici risolvono ogni dubbio:

```
package applicazione.db.logic;
import applicazione.db.utility.CostantiSQL.*;
public class GestoreDB {
    public Collection getUsers() {
        ...
        ResultSet rs = statement.execute(GET_ALL_USERS);
        ...
    }
    // Altri metodi ...
}
```

La terza soluzione é corretta e conveniente.

18.2.2 Impatto su Java

Gli import statici, come già asserito, rappresentano una novità marginale per Tiger. Non sono paragonabili ad argomenti come le enumerazioni, i generics, il ciclo for migliorato o i varargs. La loro introduzione è probabilmente dovuta all'introduzione delle enumerazioni. Sembra sia quasi una caratteristica fatta a posta per dire: "le enumerazioni sono fantastiche, e se utilizzate gli import statici, eviterete anche di scrivere il codice ripetitivo che ne caratterizza la sintassi…".
Una conseguenza negativa dell'utilizzo non ponderato degli import statici, è quella della perdita dell'identità dei membri importati staticamente. L'eliminazione del reference, se da un lato può semplificare il codice da scrivere, dall'altro potrebbe dare luogo ad ambiguità. Questo può avvenire essenzialmente in due situazioni: importando membri con lo stesso nome, o con il fenomeno dello shadowing delle variabili importate con le variabili locali.

Reference ambigui

Nel caso in cui importassimo staticamente metodi con lo stesso nome all'interno delle nostre classi, ovviamente varrebbero le regole dell'overload. Quindi, se importiamo staticamente metodi con lo stesso nome nelle nostre classi, dobbiamo essere sicuri di avere per essi firme (in particolare liste di parametri) differenti. In caso contrario il

compilatore ci segnalerà errore di "reference ai metodi ambigui", se essi vengono utilizzati all'interno del nostro codice senza reference, così come permettono gli import statici. In tal caso, per risolvere il problema di compilazione bisogna obbligatoriamente referenziare i metodi. Per esempio, consideriamo il seguente esempio:

```
import static java.lang.Math.*;
import static
javax.print.attribute.standard.MediaSizeName.*;

. . .
System.out.println(E);
```

ovviamente bisogna obbligatoriamente referenziare la variabile `E`, perché presente come variabile statica sia nella classe `Math` che nella classe `MediaSizeName`. per non ottenere un errore in compilazione, nel seguente modo:

```
System.out.println(Math.E);
```

Anche nel caso avessimo importato esplicitamente solo le due variabili `E`, come segue:

```
import static java.lang.Math.E;
import static
javax.print.attribute.standard.MediaSizeName.E;
```

il compilatore avrebbe segnalato l'errore solo nel caso di utilizzo nel codice.

Valendo le regole tradizionali dell'overload, nel caso importassimo staticamente i metodi `sort()` della classe `Arrays` (dove viene overloadato ben 18 volte in Tiger) e della classe `Collections`, non avremmo nessun problema nel loro utilizzo. Infatti, le firme dei metodi sono tutte diverse e l'overload risulterebbe "ampliato".

Shadowing

Un altro problema che introduce il non referenziare le variabili importate staticamente, è noto come "shadowing". Il fenomeno dello shadowing si manifesta quando dichiariamo una variabile locale con lo stesso nome di una variabile che ha uno scope più ampio, come una variabile d'istanza. Come già visto nel modulo 2, all'interno del metodo dove è dichiarata la variabile locale, il compilatore considera "più importante" la variabile

locale. In tali casi, per utilizzare la variabile d'istanza, bisogna obbligatoriamente referenziarla (nel caso di variabili d'istanza con il reference `this`). Lo shadowing affligge non solo le variabili d'istanza ma anche quelle importate staticamente, come mostra il seguente esempio:

```
import java.lang.System.out;
. . .
public void stampa (PrintWriter out, String text) {
    out.println(text);
}
```

all'interno del metodo `stampa()`, il reference `out` non è `System.out`, ma il parametro di tipo `PrintWriter`.

Riepilogo

In questo modulo sono stati introdotti due argomenti molto semplici: gli static import e i varargs. Dopo aver definito la sintassi dei varargs e presentato qualche esempio, ne abbiamo esplorato le proprietà. Abbiamo anche valutato l'impatto di questa nuova feature sul polimorfismo. Come esempio di utilizzo di varargs, abbiamo presentato un'introduzione alla classe `java.util.Formatter` e il suo metodo `format()`. Ne abbiamo approfittato per descrivere anche il metodo `printf()` di `java.io.PrintWriter`.
Per quanto riguarda gli import statici, ne abbiamo sottolineato la sintassi e l'applicabilità. Infine, nel paragrafo riguardante gli impatti su Java, abbiamo messo in evidenza due possibili problemi che si possono presentare quando si utilizzano gli static import: reference ambigui e shadowing.

Esercizi modulo 18

Esercizio 18.a)
Varargs, Vero o Falso:

1. I varargs permettono di utilizzare i metodi come se fossero overloadati
2. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(String… s, Date d) {
  . . .
}
```
3. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(String… s, Date d…) {
  . . .
}
```
4. Considerando il seguente metodo:
```
public void myMethod(Object… o) {
  . . .
}
```
la seguente invocazione è corretta:
```
oggetto.myMethod();
```
5. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(Object o, Object os…) {
  . . .
}
```
6. Considerando il seguente metodo:
```
public void myMethod(int i, int… is) {
  . . .
}
```
La seguente invocazione è corretta:
```
oggetto.myMethod(new Integer(1));
```
7. Le regole dell'override cambiano con l'introduzione dei varargs
8. Il metodo di `java.io.PrintStream` `printf()`, è basato sul metodo `format()` della classe `java.util.Formatter`
9. Il metodo `format()` di `java.util.Formatter` non ha overload perché definito con un varargs

10. Nel caso in cui si passi un array come varargs al metodo `printf()` di `java.io.PrintStream`, questo verrò trattato non come oggetto singolo, ma come se fossero stati passati ad uno ad uno, ogni suo elemento

Esercizio 18.b)
Static import, Vero o Falso:

1. Gli static import permettono di non referenziare i membri statici importati
2. Non è possibile dopo avere importato staticamente una variabile, referenziarla all'interno del codice
3. La seguente importazione non è corretta, perché `java.lang` è sempre importato implicitamente:
   ```
   import static java.lang.System.out;
   ```
4. Non è possibile importare staticamente classi innestate e/o anonime
5. In alcuni casi gli import statici, potrebbero peggiorare la leggibilità dei nostri file
6. Considerando la seguente enumerazione:
   ```
   package mypackage;
   public enum MyEnum {
     A,B,C
   }
   ```
   il seguente codice è compilabile correttamente:
   ```
   import static mypackage.MyEnum.*;
   public class MyClass {
     public MyClass(){
       out.println(A);
     }
   }
   ```
7. Se utilizziamo gli import statici, si potrebbero importare anche due membri statici con lo stesso nome. Il loro utilizzo all'interno del codice, darebbe luogo ad errori in compilazione, se non referenziati
8. Lo shadowing è un fenomeno che potrebbe verificarsi se si utilizzano gli import statici
9. Essenzialmente l'utilità degli import statici, risiede nella possibilità di scrivere meno codice probabilmente superfluo
10. Non ha senso importare staticamente una variabile, se poi viene utilizzata una sola volta all'interno del codice

Soluzioni esercizi modulo 18

Esercizio 18.a)
Varargs, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Falso**
8. **Vero**
9. **Falso**
10. **Vero**

Esercizio 18.b)
Static import, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Falso**
5. **Vero**
6. **Falso** `out` non è importato staticamente
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Saper utilizzare i varargs e comprenderne le proprietà (unità 18.1)	☐	
Saper utilizzare gli static imports e comprenderne le conseguenze del loro utilizzo (unità 18.2)	☐	

Note:

19

Complessità: alta

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

1. Comprendere cosa sono i metadati e la loro relatività (unità 19.1, 19.2) .
2. Comprendere l'utilità delle annotazioni (unità 19.1, 19.2, 19.3, 19.4) .
3. Saper definire nuove annotazioni (unità 19.2) .
4. Saper annotare elementi Java ed altre annotazioni (unità 19.2, 19.3) .
5. Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta annotazioni (unità 19.3, 19.4).

19 Annotazioni (Metadata)

L'argomento di questo modulo, è estremamente complesso. Dal mio personale punto di vista, si tratta dell'argomento più impegnativo tra tutti quelli trattati in questo testo.

19.1 Introduzione al modulo

La difficoltà di formulare correttamente le definizioni essenziali, la complessità di una sintassi assolutamente non standard, il livello elevato di astrazione dal linguaggio, l'introduzione di una nuova parola chiave, e la relativa applicabilità delle annotazioni, richiederanno al lettore un livello di concentrazione elevato. In compenso, le annotazioni rappresentano forse la novità potenzialmente più importante tra quelle introdotte da Tiger. Sicuramente argomenti come le enumerazioni e i generics, hanno rivoluzionato il linguaggio, rendendolo praticamente un linguaggio nuovo. Ma le annotazioni aprono uno scenario futuro a Java incredibilmente vasto. Grazie ad esse, nasceranno sicuramente nuovi strumenti di sviluppo, che renderanno la programmazione Java semplicemente migliore.
In ogni caso, riteniamo utile avvertire il lettore meno esperto, che difficilmente riuscirà ad apprezzare alcuni aspetti di questo argomento. È improbabile per esempio, che si definiscano da subito le proprie annotazioni e i propri processori di annotazioni. Ma esistono alcune parti senz'altro più "abbordabili" (vedi annotazioni standard), che potrebbero risultare utili da subito.

La prima parte di questo modulo sarà dedicata alla definizione delle annotazioni. Questa parte è abbastanza complessa, ma importante. Infatti impareremo a creare le nostre annotazioni e ne studieremo la sintassi.
La seconda parte del modulo è dedicata allo studio delle annotazioni standard, che potranno risultare utili da subito, anche al programmatore meno esperto. Ne approfitteremo per analizzare anche aspetti più complessi che potrebbero non interessare al neo programmatore, ma potrebbe essere illuminante per lo sviluppatore esperto.
Quindi, leggete questo modulo solo se:

1. siete concentrati e motivati
2. siete consapevoli che le annotazioni non saranno semplici da utilizzare subito (come magari avete fatto con gli static import)

Altrimenti, il nostro consiglio è quello di ritornare a queste pagine nel momento in cui ci si sente pronti. Ma è giunto il momento di passare ai fatti…

19.2 Definizione di Annotazione (Metadata)

Con il termine "**metadati**", solitamente si intendono le "informazioni sulle informazioni". La frase precedente non è poi così chiara però… Dovremmo filosofeggiare un po' per poter dare una definizione corretta, ma questa non ci pare la sede più opportuna. Quindi cercheremo di capire il concetto con degli esempi prima di entrare nel cuore dell'argomento.
Nel modulo 2, abbiamo definito il concetto di classe, come "un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità". Ma proviamo a guardare l'argomento da un'altra angolazione. Proviamo a pensare cos'è una classe per il compilatore o la virtual machine. Per far questo, passiamo subito a fare un esempio.
Se volessimo spiegare a qualcuno in lingua italiana cos'è una persona, cosa ovviamente molto complessa, utilizzeremmo una serie di frasi del tipo "ha un nome, un cognome, un'età…". In pratica definiremmo il concetto di persona, tramite l'elencazione delle sue caratteristiche e delle sue funzionalità. Tutto questo va bene, ma è comprensibile solo se si hanno già in mente le definizioni di nome, cognome, età etc… Ovvero, se il nostro interlocutore non sa cosa è un nome, potrebbe pensare che un nome valido sia "831". Ugualmente, se non ha mai sentito parlare di età, potrebbe pensare che un'età possa avere questa forma: "- 99".
Ecco allora il bisogno dei metadati: le informazioni sulle informazioni. Se definiamo un'età come un numero intero di anni non minore di zero, allora diventa tutto più chiaro.

Per qualcuno l'esempio precedente potrebbe essere contestabile. Nel modo reale non bisogna (fortunatamente) definire per forza tutti i metadati per farsi capire, almeno in una situazione così banale. Ma se volessimo definire cosa è per la geometria differenziale la "superficie romana di Steiner" (uno degli argomenti della mia tesi di Laurea in Matematica), non credo che senza una quantità notevole di metadati, potremmo capirci.

Molto spesso i metadati non sono altro che **vincoli** (in inglese "**constraints**"). Per esempio, nella definizione di età abbiamo specificato come vincolo che il numero degli anni non può essere minore di zero.
Un vincolo, è considerato una delle definizioni più importanti per molte metodologie moderne. Esiste infatti anche una sintassi in UML per specificare vincoli (se interessa cfr. appendice G). Addirittura, la sintassi UML stessa, è definita tramite un linguaggio che si basa proprio sul concetto di vincolo. Tale linguaggio infatti si chiama Object Constraint Language (OCL).
I metadati sono quindi informazioni su informazioni. Nel nostro esempio, la definizione di età, era un metadato relativamente alla definizione di persona. Ma relativamente alla definizione di età, potrebbe essere un metadato la definizione di anno. Quindi, il concetto di metadato, è sempre relativo a ciò che si sta definendo. (OK… un bel respiro e rileggiamo con più calma…)
Inoltre, un metadato, è anche relativo all'interlocutore che ne ha bisogno. Se per esempio spieghiamo ad un adulto un concetto come quello di persona, non dovrebbe essere necessario specificare metadati. Se il nostro interlocutore è un bambino invece, la situazione cambia.
Se poi il linguaggio in cui bisogna astrarre il concetto di persona è Java, allora non avremo di fronte un interlocutore umano, ma solo un freddo software. Che questo software sia il compilatore, la Java Virtual Machine, Javadoc o JAR, non cambia molto. Questi software non possono capire se stiamo definendo correttamente la classe `Persona`, a meno che non specifichiamo dei metadati.

Per le specifiche del linguaggio Java una classe è definita tecnicamente come "meta-modello Java".

Per esempio definiamo la classe `Persona` nel seguente modo:

```
public class Persona {
    . . .
    private int anni;
    public void setAnni(int anni) {
        if (anni < 0)
            throw new IllegalArgumentException(anni +
              " anni, non è una età valida!";
            this.anni = anni;
        }
    public int getAnni() {
        return anni;
    }
}
```

Con il controllo nel metodo `setAnni()`, abbiamo specificato un vincolo destinato alla JVM.
Un altro vincolo specificato dal codice precedente è invece riservato al compilatore: il modificatore `private`. Questo, rappresenta un metadato per il compilatore, che specifica che la variabile `anni`, non ha visibilità all'esterno della classe.
Esistono in Java anche altri meccanismi per specificare metadati. Per esempio, potremmo specificare con il tag `@deprecated` interno ad un commento javadoc, che un certo metodo è deprecato. Questo tipo di metadato è invece destinato all'utility Javadoc e al compilatore.
Su tag interni a commenti javadoc come `@deprecated`, è basato una tecnologia chiamata doclet, la cui più famosa implementazione è un progetto open source che trovate all'indirizzo http://xdoclet.sourceforge.net. Un "motore doclet" è un software che crea file sorgenti Java in base a tag di questo tipo. Motori simili sono molto utilizzati dai tool di sviluppo più famosi, per automatizzare la creazione di componenti complessi come gli EJB. I tag doclet sono destinati al motore doclet, un software per la generazione di codice creato ad hoc.
Quindi Java fornisce diversi modi di specificare metadati, ma prima di Tiger, non esisteva una sintassi univoca per poter avere diversi "interlocutori" come il compilatore, Javadoc e la JVM. Inoltre, ognuno di questi modi (a parte la tecnologia doclet), sono stati creati senza pensare al concetto di metadato. Quindi, ognuno di questi modi, ha dei limiti palesi (che non elencheremo per semplicità) per specificare i metadati.
Giusto per fare un esempio, accenniamo solo al paragone con l'unica tecnologia attualmente paragonabile: xdoclet. Quest'ultima, ha un limite essenziale: non ci sono

controlli sulla correttezza dei tag. Se per esempio viene specificato un tag come il seguente in un commento javadoc:

```
/** …
*
*  @overrride . . .
*
*/
```

(notare le tre "r"), xdoclet semplicemente ignorerà il tag, senza segnalare nessun tipo di errore. Le annotazioni invece, possono avere il pieno supporto del compilatore Java, e quindi il confronto termina qui.
Con l'introduzione del meccanismo delle "**annotazioni**" (in inglese "**annotations**"), Java ha ora un modo standard per definire i metadati di un programma. Per esempio è possibile specificare vincoli definiti dallo sviluppatore, che possono essere "interpretati" da un software come la JVM, il compilatore, Javadoc etc…. È anche possibile, creare un "processore di annotazioni" ad hoc, che interpreta le annotazioni, per esempio creando nuovi file sorgenti ausiliari a quelli già esistenti. Ma le annotazioni, essendo delle meta-informazioni dirette verso un software, hanno un orizzonte colmo di possibilità. Siamo sicuri che il tempo ci darà ragione…

19.1.1 Primo esempio

Tecnicamente le annotazioni sono un altro tipo di struttura dati del linguaggio, che si va ad aggiungere alle classi, alle interfacce ed alle enumerazioni. Si dichiarano in maniera simile alle interfacce, ma si definiscono con una sintassi che va spesso fuori dallo standard Java. Probabilmente, la sintassi delle annotazioni, è ancora più inusuale di quella delle classi anonime. Di seguito riportiamo un primo esempio di definizione di annotazione:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "[da assegnare]";
}
```

Come si può notare, c'è una nuova parola chiave in Java: `@interface`.
In questo caso l'annotazione si chiama `DaCompletare`, e definisce due "strani" metodi astratti: `descrizione()` e `assegnataA()`. In realtà questi due metodi, hanno una sintassi abbreviata, equivalente alla dichiarazione di una variabile con relativo

metodo omonimo che restituisce il suo valore. Notare anche come lo "strano" metodo `assegnataA()`, utilizzi anche la parola chiave `default`. Questa serve per specificare un valore di default per il metodo nel caso che, utilizzando l'annotazione, lo sviluppatore non fornisca un valore per la variabile "invisibile" `assegnataA`. Ma approfondiremo tra poco i dettagli della sintassi.

`DaCompletare` viene detto quindi "tipo annotazione" ("annotation type").

Dopo aver visto come si dichiarano le annotazioni, cerchiamo di capire come si utilizzano. È possibile utilizzare un'annotazione come si fa con un modificatore, per esempio per un metodo, utilizzando la seguente sintassi:

```
public class Test {
    @DaCompletare(
        descrizione = "Bisogna fare qualcosa…",
        assegnataA = "Claudio"
    )
    public void faQualcosa() {

    }
}
```

Analizziamo brevemente la sintassi utilizzata. L'annotazione viene dichiarata come se fosse un modificatore del metodo `faQualcosa()`. La sintassi però è molto particolare. Si utilizza il simbolo di chiocciola `@` (si dovrebbe leggere "AT"), che si antepone al nome dell'annotazione. Poi si aprono delle parentesi tonde per specificare dei parametri, quasi l'annotazione fosse un metodo… Specificare però i parametri di un'annotazione, significa specificare delle coppie del tipo chiave = valore, dove le chiavi, corrispondono alle variabili (mai dichiarate) relative al metodo omonimo. Quindi, se nell'annotazione abbiamo definito i metodi `descrizione()` e `assegnataA()`, abbiamo anche in qualche modo definito le variabili "invisibili" `descrizione` e `assegnataA`. Il loro tipo, corrisponde esattamente al tipo di ritorno del metodo omonimo. Infatti, tale metodo funzionerà da metodo getter (o accessor), ovvero restituirà il valore della relativa variabile.
Il metodo `faQualcosa()` è a questo punto stato annotato, ma manca ancora qualcosa. Bisogna creare un software che interpreti la annotazione implementando un

comportamento. Infatti, se anche noi possiamo intuire a cosa serva l'annotazione `DaCompletare`, nessun software avrà mai la nostra stessa perspicacia.
Per esempio, potremmo creare un'applicazione la quale riceva in input una classe, legga le eventuali annotazioni, e pubblichi su di una bacheca in Intranet i compiti da assegnare ai vari programmatori.

```
import java.lang.reflect.*;
import java.util.*;
public class AnnotationsPublisher {

    public static void main(String[] args) throws
      Exception {
        Map<String,String> map = new
          HashMap<String,String>();
        for (Method m :
          Class.forName("Test").getMethods()) {
            DaCompletare dc = null;
            if ((dc = m.getAnnotation
                (DaCompletare.class)) != null) {
                String descrizione = dc.descrizione();
                String assegnataA = dc.assegnataA();
                map.put(descrizione, assegnataA);
            }
        }
        pubblicaInIntranet(map);
    }
    public static void pubblicaInIntranet(
      Map<String,String> map) {
        Set <String>keys = map.keySet();
        for (String key : keys) {
            . . .
        }
    }
}
```

La classe `AnnotionsPublisher`, dichiara una mappa parametrizzata. Poi con un ciclo foreach, estrae tramite reflection i metodi della classe `Test`. Sfruttando i nuovi metodi della classe `Method`, ed in particolare il metodo `getAnnotation()`,

vengono scelti solo i metodi che sono stati annotati con `DaCompletare`. Per essi vengono memorizzate nella mappa le informazioni. Finito il ciclo, viene invocato il metodo `pubblicaInIntranet()`, che si occuperà di estrarre le informazioni dalla mappa per pubblicarle in Intranet…

Questo era solo un esempio ma le potenzialità delle annotazioni sono potenzialmente infinite.
Per semplicità abbiamo omesso la prima parte della definizione della annotazione `DaCompletare`. Senza di essa, non sarà possibile far funzionare correttamente l'esempio. Segue la definizione completa dell'annotazione:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

in pratica l'annotazione è stata a sua volta annotata… tra poco capiremo perché…

19.1.2 Tipologie di annotazioni e sintassi

Esistono tre tipologie di annotazioni: le annotazioni ordinarie, le annotazioni a valore singolo e le annotazioni segnalibro.
L'unica annotazione che abbiamo già visto è un esempio di **annotazione ordinaria**. Si tratta del tipo di annotazione più complesso. Infatti viene detto anche **full annotation** (**annotazione completa**).
Ri-analizziamo l'annotazione dell'esempio:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

L'annotazione `DaCompletare`, definisce due metodi astratti o, almeno tecnicamente, hanno una sintassi simile a quella dei metodi astratti. Come già asserito in precedenza però, in realtà hanno un'implicita implementazione. Infatti, dichiarare un metodo, equivale a dichiarare una coppia costituita da una variabile ed un metodo omonimi. Quest'ultimo ritorna il valore della variabile.

Un'altra novità nella sintassi, è l'utilizzo della parola chiave `default` per assegnare un valore predefinito alla variabile "nascosta" `assegnataA`.
Possiamo quindi immaginare che un'annotazione sia un specie di interfaccia che viene implementata da una classe creata al volo dal compilatore, simile alla seguente:

```
public class DaCompletareImpl implements DaCompletare {
    private String descrizione;
    private String assegnataA = "da assegnare";
    public String descrizione() {
        return descrizione;
    }
    public String assegnataA() {
        return assegnataA;
    }
}
```

Per un metodo di un'annotazione, non è possibile né specificare parametri di input, né `void` come valore di ritorno. Il compilatore altrimenti segnalerà errori espliciti.

Essendo come un'interfaccia, è possibile dichiarare all'interno di un'annotazione, oltre a metodi (implicitamente astratti), anche costanti ed enumerazioni (implicitamente `public`, `static` e `final`). Per esempio possiamo arricchire l'annotazione in questo modo:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
    enum Priorita {ALTA, MEDIA, BASSA};
    Priorita priorita() default Priorita.ALTA;
}
```

La convenzione per gli identificatori delle annotazioni, è ovviamente identica a quella delle classi, le interfacce e le enumerazioni.

Dopo studiato la sintassi della dichiarazione di un'annotazione, analizziamo ora la sintassi dell'utilizzo di un'annotazione.
Un'annotazione, come visto nell'esempio precedente, viene utilizzata come se fosse un

modificatore. Per convenzione un'annotazione precede gli altri modificatori, ma non è obbligatorio. La sintassi per modificare un elemento di codice Java (classe, metodo, variabile locale, parametro etc…) è sempre del tipo:

```
@NomeAnnotazione ([lista di coppie] nome=valore)
```

Nel precedente esempio avevamo modificato un metodo nel seguente modo:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa…",
    assegnataA = "Claudio"
)
public void faQualcosa() {

}
```

È obbligatorio settare tutte le variabili i cui metodi non dichiarano un `default`. Quindi è legale scrivere:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa…",
)
public void faQualcosa() {

}
```

mentre il seguente codice:

```
@DaCompletare(
    assegnataA = "Claudio"
)
public void faQualcosa() {

}
```

provocherà il seguente errore in compilazione:

```
Test.java:4: annotation DaCompletare is missing
descrizione
  assegnataA = "Claudio"
               ^
1 error
```

Se invece consideriamo la versione "arricchita" dall'enumerazione dell'annotazione `DaCompletare`, allora potremmo utilizzarla nel seguente modo:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa…",
    priorita = DaCompletare.Priorita.BASSA
)
public void faQualcosa() {

}
```

Avendo fornito un `default`, anche al metodo `priorita()`, è possibile ometterne il settaggio.

Il secondo tipo di annotazione è detto **annotazione a valore unico** ("**single value annotation**"). Si tratta di un'annotazione che contiene un unico metodo che viene chiamato `value()`. Per esempio la seguente annotazione è di tipo a valore unico:

```
public @interface Serie {
    Alfabeto value();
    enum Alfabeto {A,B,C};
}
```

Notare che è possibile dichiarare un qualsiasi tipo di ritorno per il metodo `value()`, tranne il tipo `java.lang.Enum`.

Ovviamente anche la seguente è un'annotazione a valore unico:

```
public @interface SingleValue {
    int value();
}
```

La seguente annotazione:

```
public @interface SerieOrdinaria {
    Alfabeto alfabeto();
    enum Alfabeto {A,B,C};
}
```

è un'annotazione ordinaria, perché non definisce come unico metodo il metodo `value()`.
La differenza con altri tipi di annotazioni è nella sintassi dell'utilizzo. Per utilizzare una annotazione a valore unico, è infatti possibile scrivere all'interno delle parentesi dell'annotazione, solamente il valore da assegnare. Per esempio potremmo scrivere in luogo di:

```
@Serie(value = Serie.Alfabeto.A)
public void faQualcosa() {

}
```

più semplicemente:

```
@Serie(Serie.alfabeto.A)
public void faQualcosa() {

}
```

con lo stesso risultato.
La terza tipologia di annotazioni, è chiamata **annotazione segnalibro** (in inglese "**marker annotation**"). Si tratta della tipologia più semplice: un'annotazione che non ha metodi. Per esempio la seguente è un esempio di annotazione segnalibro:

```
public @interface Marker {}
```

Questo tipo di annotazioni vengono ovviamente utilizzate con la sintassi che ci si aspetterebbe:

```
@Marker()
public void faQualcosa() {
```

```
}
```

e possono risultare molto più utili di quanto non ci si aspetti.
È anche possibile evitare di specificare le parentesi tonde come mostra il seguente codice:

```
@Marker public void faQualcosa() {

}
```

È anche possibile creare annotazioni innestate in classi come nel seguente esempio:

```
public class Test {
    public @interface Serie {
        alfabeto value();
        enum alfabeto {
            A,B,C
        } ;
    }
    @Serie(Serie.Alfabeto.A)
    public void faQualcosa() {

    }
}
```

Anche se in realtà è più probabile che si creino librerie pubbliche di annotazioni.
Un'annotazione implementerà l'interfaccia `java.lang.annotation.Annotation`. Notare che implementando "manualmente" tale interfaccia, non definiremo un'annotazione. Praticamente tra le annotazioni e l'interfaccia `Annotation`, c'è lo stesso rapporto che c'è tra le enumerazioni e la classe `java.lang.Enum`.

Un'annotazione non può utilizzare né la parola chiave `implements` né la parola chiave `extends`. Quindi, in nessun modo un'annotazione può estendere un'altra annotazione, una classe, o implementare un'interfaccia.

Vi sarete sicuramente chiesti: “un’annotazione a quali elementi di programmazione si può applicare?”. Infatti è possibile specificare a quali elementi può essere applicata un’annotazione come vedremo più avanti. Se non esplicitamente specificato comunque, un’annotazione è applicabile a qualsiasi tipo di elemento di programmazione a cui è applicabile un qualsiasi modificatore.
È possibile anche decidere se un’annotazione deve essere destinata alla lettura del solo compilatore, o anche al runtime. Di default, l’annotazione viene riportata nel file “.class”, ma non considerata dalla JVM.
Le ultime due osservazioni saranno chiarite nella prossima unità didattica.

19.2 Annotare annotazioni (meta-annotazioni)

Nel package `java.lang.annotation`, sono definite alcune “meta-annotazioni”. Si tratta di quattro tipi annotazioni, che sono destinate ad annotare solamente altre annotazioni. Queste quattro annotazioni si chiamano `Retention`, `Target`, `Documented`, `Inherited`.
In realtà abbiamo già visto un esempio di meta-annotazione, nell’unità didattica precedente. Infatti, dopo aver presentato il primo esempio, abbiamo fatto notare come l’annotazione `DaCompletare`, per funzionare correttamente, doveva essere modificata nel seguente modo:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

in pratica `Retention`, è un’annotazione (a valore unico) destinata ad annotare altre annotazioni. In questo caso, passando a `Retention` il valore unico `RetentionPolicy.RUNTIME`, si specificava che l’annotazione `DaCompletare`, era destinata alla lettura da parte del runtime Java. Senza l’utilizzo della meta-annotazione, l’annotazione `DaCompletare`, di default sarebbe stata inclusa nel bytecode dopo la compilazione, ma non considerata dal runtime Java.

19.2.1 Target

La prima meta-annotazione che andiamo a studiare è `java.lang.annotation.Target`. Il suo scopo è quella di specificare gli elementi

del linguaggio a cui è applicabile l'annotazione che si sta definendo. Infatti, in italiano `Target` significa "obiettivo".
Questa meta-annotazione è di tipo a singolo valore, e prende come parametro un array di `java.lang.annotation.ElementType`. `ElementType` è un'enumerazione definita come segue:

```
package java.lang.annotation;
public enum ElementType {
  TYPE, // Classi, interfacce, o enumerazioni
  FIELD, // variabili d'istanza (anche se enum)
  METHOD, // Metodi
  PARAMETER, // Parametri di metodi
  CONSTRUCTOR, // Costruttori
  LOCAL_VARIABLE, // Variabili locali o clausola catch
  ANNOTATION_TYPE, // Tipi Annotazioni
  PACKAGE // Package
}
```

Essa specifica con i suoi elementi i vari elementi del linguaggio Java, a cui è possibile applicare un'annotazione.
Per esempio potremmo utilizzare questa meta-annotazione, per limitare l'applicabilità dell'annotazione `DaCompletare`:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType
@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE,
ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Ora, l'annotazione `DaCompletare`, è applicabile solo a classi, interfacce, enumerazioni, annotazioni, metodi, costruttori e package.
Notare come gli elementi dell'array di `ElementType` siano specificati mediante la sintassi breve dell'array. Inoltre, tali elementi, sono stati utilizzati senza essere referenziati grazie all'import statico.

Notiamo inoltre, che se si vuole passare come parametro a `Target` un unico elemento, è possibile utilizzare anche la sintassi:

```
@Target(TYPE);
```

infatti il compilatore in questo caso è capace di capire le nostre intenzioni. Invece di darci la solita lezione di robustezza, questa volta il compilatore è piuttosto accomodante…

Se definiamo un'annotazione senza utilizzare la meta-annotazione `Target`, allora la nostra annotazione sarà applicabile di default a tutti gli elementi possibili.

Può risultare interessante, notare come viene definita l'annotazione `Target`:

```
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME);
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
  ElementType[] value( );
}
```

In pratica `Target` viene annotata da se stessa, affinché sia applicabile solo ad altre annotazioni.

19.2.2 Retention

La meta-annotazione `Retention` (che in italiano possiamo tradurre come "conservazione"), è anch'essa molto importante. Serve per specificare come deve essere conservata dall'ambiente Java, l'annotazione a cui viene applicata. Come `Target` anche `Retention` è di tipo a singolo valore, ma prende come parametro un valore dell'enumerazione `java.lang.annotation.RetentionPolicy`. Segue la definizione di `RetentionPolicy`, con dei commenti esplicativi sull'uso dei suoi valori:

```
package java.lang.annotation;
public enum RetentionPolicy {
  SOURCE, // l'annotazione è eliminata dal compilatore
```

```
  CLASS, /* l'annotazione viene conservata anche nel file
            ".class", ma ignorata dall JVM */
  RUNTIME /* l'annotazione viene conservata anche nel
             file  ".class", e letta dalla JVM */
}
```

Come già affermato precedentemente, la meta-annotazione `Retention` applicata alla annotazione `DaCompletare`, farà in modo che le annotazioni di tipo `DaCompletare`, siano conservate nei file compilati, per essere infine letti anche dalla virtual machine. Riportiamo nuovamente il codice di seguito:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Nel paragrafo "primo esempio" infatti, lo scopo di questa annotazione era quella di essere letta tramite reflection, e quindi era obbligatorio specificare tale meta-annotazione.

19.2.3 Documented

Questa semplice meta-annotazione, ha il compito di includere nella documentazione generata da Javadoc, anche le annotazioni a cui è applicata. Per esempio, la meta-annotazione `Target`, è a sua volta annotata da `Documented` come mostra la sua dichiarazione:

```
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME);
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
  ElementType[] value( );
}
```

Questo significa che se generiamo la documentazione del nostro codice tramite il comando javadoc, sui file dove è utilizzato `Target`, allora anche l'annotazione verrà riportata. Per esempio se generiamo la documentazione della seguente annotazione:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface MaxLength {
    int value();
}
```

che specifica la lunghezza massima di un certo campo (e che si può verificare a Runtime), avremo come risultato quanto mostrato in figura 19.1:

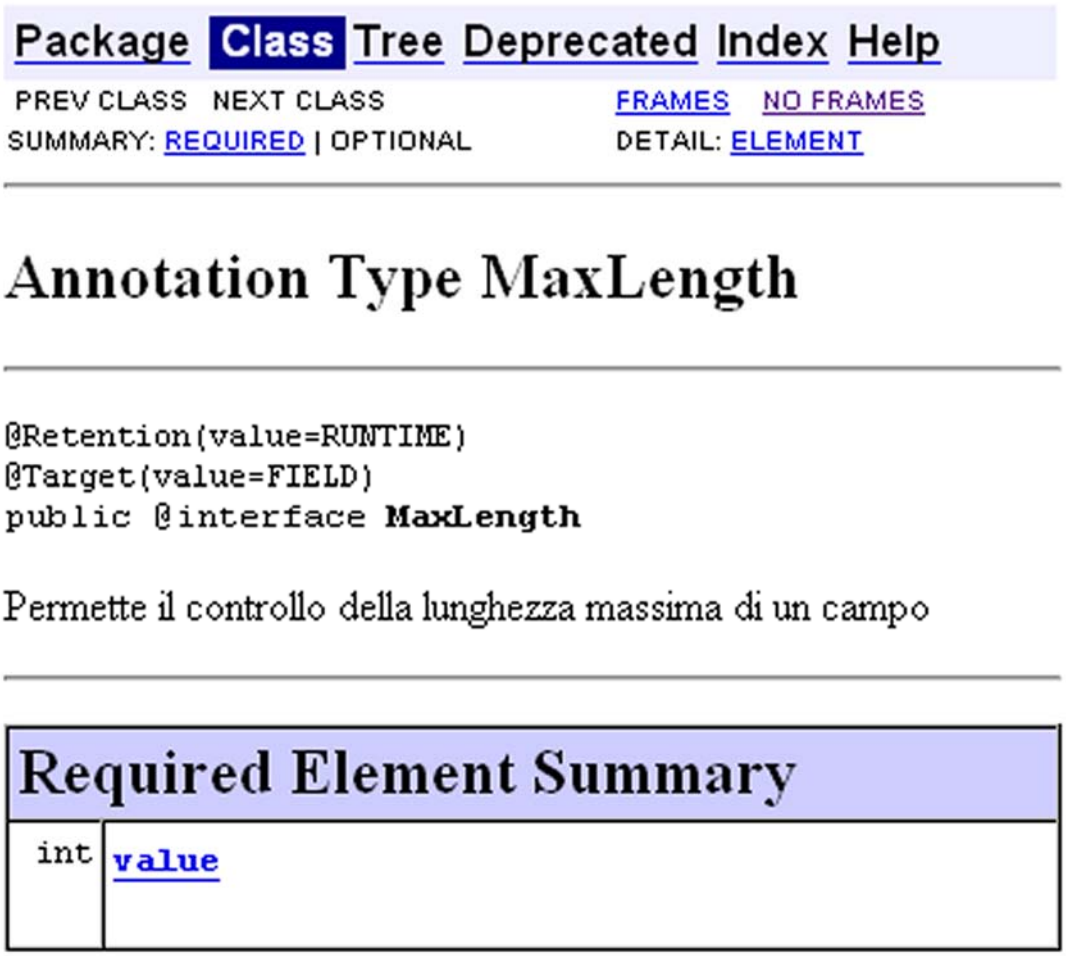


Figura 19.1 – "Un particolare della documentazione generata"

Nella figura 19.1 è possibile notare come anche `Retention` sia annotata con `Documented`

19.2.4 Inherited

Questa meta-annotazione permette alle annotazioni applicate a classi (e solo a classi), di essere ereditate. Questo significa che se abbiamo la seguente annotazione:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE,
ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

annotata a sua volta da `Inherited`, e l'applichiamo alla seguente classe:

```
@DaCompletare (
  descrizione = "Da descrivere…"
)
public class SuperClasse {
  . . .
}
```

che a sua volta viene estesa dalla seguente sottoclasse:

```
public class SottoClasse extends SuperClasse {
  . . .
}
```

Allora anche quest'ultima sarà annotata allo stesso modo della superclasse. Basterà lanciare la seguente classe per verificarlo:

```
import java.lang.reflect.*;
import java.util.*;
import java.lang.annotation.*;
public class AnnotationsReflection2 {

    public static void main(String[] args) throws
Exception
    {
        Annotation[]
dcs=SottoClasse.class.getAnnotations();
        for (Annotation dc : dcs) {
            System.out.println(dc);
        }
    }
}
```

Attualmente sussiste uno strano comportamento da parte del JDK, nell'utilizzo congiunto di `Inherited` e `Documented`. Infatti, se generiamo la documentazione tramite Javadoc della classe `SuperClasse`, sarà documentata correttamente anche l'annotazione `DaCompletare`, come è possibile notare dalla figura 19.2.

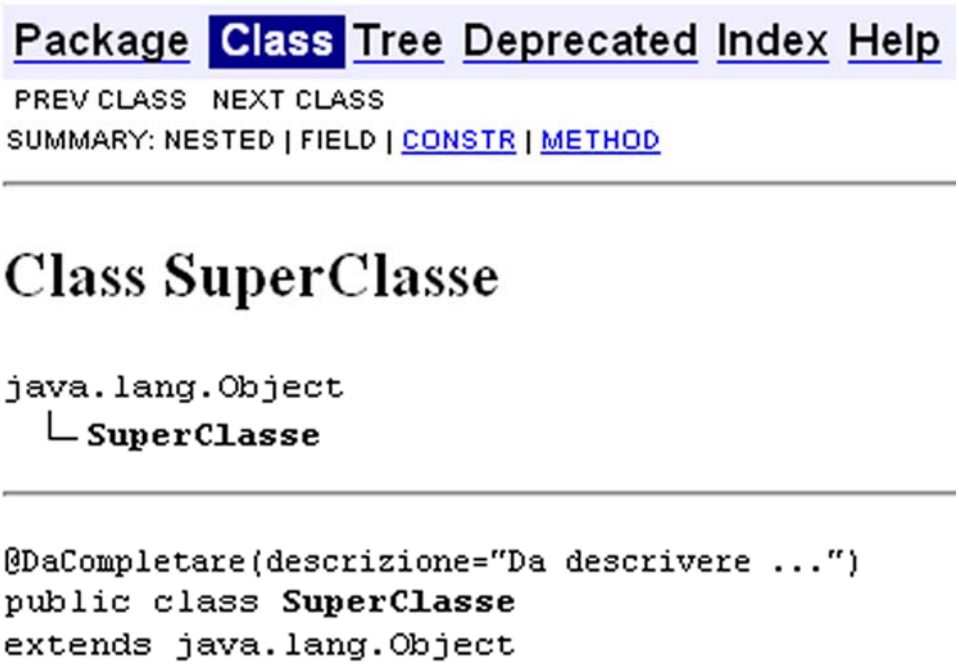


Figura 19.2 – "Un particolare della documentazione generata di SuperClasse"

Questo, non avverrà, come ci si aspetterebbe, per la classe `SottoClasse`, come è possibile notare dalla nella figura 19.3.

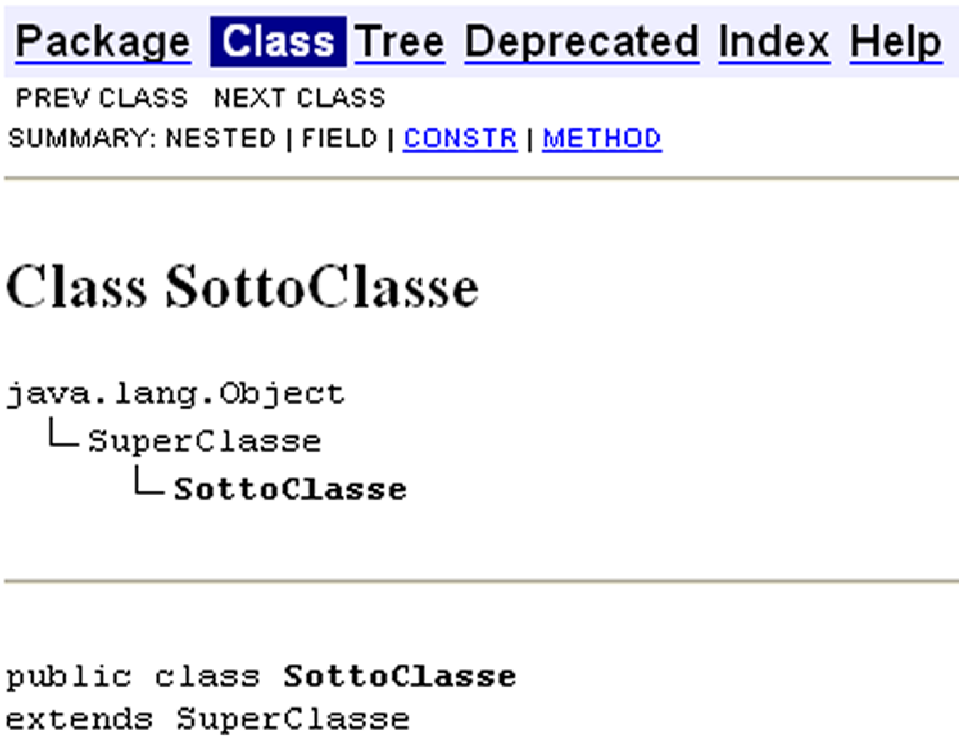


Figura 19.3 – "Un particolare della documentazione generata di SottoClasse"

Questo significa che nonostante, `SottoClasse` abbia sicuramente ereditato l'annotazione della `SuperClasse`, il tool Javadoc, non è in grado (per il momento) di riportarlo.

> **Le annotazioni annotate con `Inherited`, sono ereditate solo se applicate a classi. Tali annotazioni, se applicate a metodi, interfacce, o qualsiasi altro elemento Java che non sia una classe, non saranno ereditate.**

19.4 Annotazioni standard

Nel package `java.lang`, sono definite le uniche tre annotazioni "normali" attualmente definite nella libreria: `Override`, `Deprecated`, e `SuppressWarnings`.

19.4.1 Override

L'annotazione `java.lang.Override`, può essere utilizzata per indicare al compilatore che un metodo in una classe, è un override di un altro metodo della sua superclasse. Di seguito è riportata la sua dichiarazione:

```
package java.lang;
import java.lang.annotation.*;
@Target(value=ElementType.METHOD)
@Retention(value=RetentionPolicy.SOURCE)
public @interface Override {}
```

come è possibile notare, si tratta di un'annotazione di tipo marker, e quindi non si devono specificare valori quando si utilizza. Inoltre, è ovviamente applicabile solo a metodi (notare il valore della meta-annotazione `Target`). Infine, tramite il valore di `Retention` settato a `RetentionPolicy.SOURCE`, viene specificato che `Override`, è una annotazione interpretabile solo dal compilatore, e che non sarà inserita nel bytecode relativo.
L'utilità di questa annotazione è piuttosto intuitiva. Per esempio un tipico errore abbastanza difficile da scoprire, che i neo-programmatori a volte commettono, è relativo, alla gestione degli eventi sulle interfacce grafiche (cfr. Modulo 15). Ricordiamo brevemente che il codice necessario per gestire l'evento di chiusura di una finestra AWT, deve obbligatoriamente far uso o dell'interfaccia `WindowListener`, o della classe `WindowAdapter`. Per esempio, potremmo creare con una classe anonima al volo il gestore di tale evento con il seguente codice:

```
frame.setWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
      System.exit(0);
    }
});
```

Tale approccio è consigliabile per la brevità del codice. Purtroppo però, uno degli svantaggi che comporta l'utilizzo di una classe adapter in luogo di un'interfaccia listener, è che se il nome del metodo, viene in qualche modo alterato, il compilatore non segnalerà nessun errore. Uno dei tipici errori che abbiamo visto commettere in questi casi, è quello di chiamare il metodo "`windowsClosing`"… ma è tutta colpa del monopolio Microsoft! Al runtime invece, in risposta all'evento della chiusura della finestra, verrà invocato il metodo `windowClosing()`, e non il metodo che pensavamo

di aver riscritto.
In questi casi, l'utilizzo dell'annotazione `Override`:

```
frame.setWindowListener( new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent ev) {
        System.exit(0);
    }
});
```

ci potrebbe evitare noiosi debug, segnalando il problema in fase di compilazione.

19.4.2 Deprecated

L'annotazione standard `java.lang.Deprecated` ovviamente serve per indicare al compilatore e al runtime di Java, che un metodo o un qualsiasi altro elemento di codice Java è deprecato. Quindi, l'annotazione `Deprecated` ha per il compilatore, la stessa funzione che il tag `@deprecated` ha per l'utility Javadoc. Sono due "istruzioni" complementari e vanno utilizzate contemporaneamente. Infatti, se utilizzassimo solo il tag javadoc `@deprecated`, il compilatore ci restituirebbe un warning simile al seguente:

warning:deprecated name isnt annotated with @Deprecated

Anche il tag `Deprecated`, è di tipo marker. Può essere utilizzato per annotare qualsiasi elemento Java. Segue la sua dichiarazione:

```
import java.lang.annotation.*;
import static
java.lang.annotation.RetentionPolicy.RUNTIME;
@Documented
@Retention(value=RUNTIME)
public @interface Deprecated {}
```

Il seguente codice rappresenta un esempio di utilizzo di `Deprecated`:

```
@DaCompletare (
    descrizione = "Da descrivere ..."
)
```

```
public class SuperClasse {
    /**
    * Questo metodo è stato deprecato
    * @deprecated utilizza un altro metodo per favore
    */
    @Deprecated public void metodo() {
        . . .
    }
}
```

Le opzioni del compilatore "–deprecation" e "–Xlint:deprecation", sono equivalenti. Se viene per esempio utilizzato il metodo di `SuperClasse`, otterremo in fase di compilazione un warning simile al seguente:

```
TestAnnotation.java:4: warning: [deprecation] metodo() in SuperClasse has been
deprecated
   sc.metodo();
     ^
1 warning
```

sia che venga esplicitata l'opzione "–deprecation", sia che venga utilizzata l'opzione "–Xlint:deprecated".

EJE, se viene utilizzato con un JDK 1.5, e nelle opzione viene settato come valore di versione Java "1.5", utilizza di default il tag "Xlint", e quindi tutti i messaggi di warning vengono specificati.

19.4.3 SuppressWarnings

Ci sono tante ragioni per cui uno sviluppatore può scegliere di lavorare con le nuove caratteristiche introdotte da Tiger. Per esempio, si potrebbe avere voglia di iniziare ad utilizzare le nuove feature in maniera graduale, magari su del codice già sviluppato con versioni precedenti di Java. In tali casi, il compilatore di Tiger, molto probabilmente compilerà ugualmente l'applicazione, ma segnalerà dei warning di tipo Xlint (cfr Unità Didattica relativa ai Generics). Per esempio, il seguente codice:

```
Vector strings = new Vector();
strings.add("a");
strings.add("b");
strings.add("c");
```

```
for (String stringa : strings)
    System.out.println(stringa);
```

genererà un output simile al seguente:

```
Note: Warnings.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Se ricompiliamo specificando l'opzione "-Xlint", otterremo il seguente output che ci evidenzia il problema.

```
Warnings.java:102: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.Vector
      strings.add("a");
              ^
Warnings.java:103: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.Vector
      strings.add("b");
              ^
Warnings.java:104: warning: [unchecked] unchecked call to add(E) as a member
of the raw type java.util.Vector
      strings.add("c");
              ^
```

Ovviamente, la soluzione del nostro problema, non può risiedere nel compilare con l'opzione "–source 1.4". Infatti, in tal caso, non potremmo utilizzare nessuna delle nuove feature di Tiger nel nostro codice. In questo caso per esempio, il codice genererebbe un errore per dovuto all'uso del ciclo foreach.
Inoltre, ovviamente, il "buon senso dello sviluppatore", non ci permette di ignorare i warning del compilatore.
Sono questi i casi in cui può essere utile sfruttare l'annotazione `SuppressWarnings`. Questa infatti può svolgere il ruolo di modificatore per classi, metodi, costruttori, variabili d'istanza, parametri e variabili locali, affinché non generino warning. Si tratta questa volta di un'annotazione a valore unico, il cui parametro è di tipo array di stringhe. Questo serve per specificare la tipologia di warning Xlint (per esempio "unchecked"), che deve essere soppressa dal compilatore. Segue la sua dichiarazione:

```
package java.lang;
```

```
import java.lang.annotation.*;
import java.lang.annotation.ElementType;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

Per esempio, se vogliamo che il compilatore non generi warning di tipo `unchecked` per il metodo dell'esempio precedente, basterà annotare tale metodo nel seguente modo:

```
@SuppressWarnings({"unchecked"})
public void stampa() {
    Vector strings = new Vector();
    strings.add("a");
    strings.add("b");
    strings.add("c");
    for (Object o : strings) {
        System.out.println(o);
    }
}
```

In particolare, tale annotazione permette di sopprimere warning riguardanti l'elemento annotato. Se l'elemento annotato contiene anche altri elementi che possono provocare warning (dello stesso tipo specificato), anche questi saranno soppressi dal compilatore. Per esempio se una classe è annotata per sopprimere i warning di tipo deprecated, saranno soppressi eventuali altri warning dello stesso tipo, relativi ai metodi della classe.

Come consiglio stilistico proveniente direttamente da Joshua Block (lo sviluppatore di tale annotazione), i programmatori dovrebbero sempre annotare l'elemento più innestato. Quindi, è sconsigliato annotare una classe per annotare tutti i suoi metodi. Sarebbe meglio annotare ogni metodo.

Se si specifica come parametro due volte lo stesso parametro, la seconda occorrenza sarà ignorata, così come tutte le occorrenze non

valide, secondo la sintassi Xlint (cfr. Unità Didattica relativa ai Generics).

19.4.4 Impatto su Java

L'impatto sul linguaggio delle annotazioni è notevole. La sintassi di Java si è arricchita di una nuova tipologia di tipo (i tipi annotazione), di una nuova parola chiave (`@interface`) e di una nuova sintassi per dichiarare ed utilizzare le annotazioni. Ma probabilmente l'impatto su Java più rilevante sarà più visibile in futuro. Come già asserito, infatti, probabilmente presto vedremo nuove applicazioni che permetteranno di programmare meglio. Non si può avere idea di tutto quello che ora gli sviluppatori hanno a disposizione. Personalmente ho da tempo iniziato a sviluppare un nuovo framework per J2EE, basato proprio sulle annotazioni. Si chiama XMVC ed è ospitato attualmente all'indirizzo http://sourceforge.net/projects/xmvc, ma ogni tanto date uno sguardo anche al mio sito personale (http://www.claudiodesio.com)… potrebbero esserci delle novità…

Riepilogo

In questo modulo, è stato introdotto un unico nuovo argomento. La sua complessità però, è particolarmente alta, e, solo per dare una prima definizione di metadato, abbiamo dovuto ricorrere a numerosi esempi. Dopo averne presentato la complessa sintassi, abbiamo visto come sia possibile creare proprie annotazioni. Abbiamo anche visto cosa sono le meta-annotazioni, andando anche a studiarne le più importanti che si trovano nella documentazione standard. Infine abbiamo analizzato le uniche annotazioni standard della libreria. Nel paragrafo dedicato agli impatti su Java abbiamo essenzialmente sottolineato la potenza di questa nuova feature di Java 5.

Esercizi modulo 19

Esercizio 19.a)
Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. Un'annotazione è un modificatore
2. Un'annotazione è un'interfaccia
3. I metodi di un'annotazione, sembrano metodi astratti, ma in realtà sottintendono un'implementazione implicita
4. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    void metodo();
}
```
5. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    int metodo(int valore) default 5;
}
```
6. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    int metodo() default -99;
    enum MiaEnum{VERO, FALSO};
    MiaEnum miaEnum();
}
```
7. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6, sia corretta. Con il seguente codice essa, viene utilizzata correttamente:
```
public @MiaAnnotazione (
    MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
    return MiaAnnotazione.MiaEnum.VERO;
}
```
8. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6, sia corretta. Con il seguente codice essa, viene utilizzata correttamente:
```
public @MiaAnnotazione (
    miaEnum=MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
```

```
    return @MiaAnnotazione.miaEnum;
}
```

9. Consideriamo la seguente annotazione.
   ```
   public @interface MiaAnnotazione {
       int valore();
   }
   ```
   Con il seguente codice essa, viene utilizzata correttamente:
   ```
   public @MiaAnnotazione (
       5
   )
   void  m()
       ...
   }
   ```
10. Consideriamo la seguente annotazione:
    ```
    public @interface MiaAnnotazione {}
    ```
    Con il seguente codice essa, viene utilizzata correttamente:
    ```
    public @MiaAnnotazione void  m() {
        ...
    }
    ```

Esercizio 19.b)
Annotazioni e libreria, Vero o Falso:

1. La seguente annotazione è anche una meta annotazione:
   ```
   public @interface MiaAnnotazione ()
   ```
2. La seguente annotazione è anche una meta annotazione:
   ```
   @Target (ElementType.SOURCE)
   public @interface MiaAnnotazione ()
   ```
3. La seguente annotazione è anche una meta annotazione:
   ```
   @Target (ElementType.@INTERFACE)
   public @interface MiaAnnotazione ()
   ```
4. La seguente annotazione se applicata ad un metodo sarà documentata nella relativa documentazione Javadoc:
   ```
   @Documented
   @Target (ElementType.ANNOTATION_TYPE)
   public @interface MiaAnnotazione ()
   ```
5. La seguente annotazione sarà ereditata, se e solo se applicata ad una classe:
   ```
   @Inherited
   ```

```
@Target (ElementType.METHOD)
public @interface MiaAnnotazione ()
```

6. Per la seguente annotazione, è anche possibile creare un processore di annotazioni che riconosca al runtime il tipo di annotazione, per implementare un particolare comportamento:
```
@Documented
@Target (ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MiaAnnotazione ()
```
7. `Override` è un'annotazione standard per segnalare al runtime di Java che un metodo fa override di un altro
8. `Deprecated`, in fondo può essere considerata anche una meta-annotazione, perché applicabile ad altre annotazioni
9. `SuppressWarnings` è una annotazione a valore singolo. `Deprecated` e `Override` invece, sono entrambe annotazioni segnalibro
10. Non è possibile utilizzare contemporaneamente le tre annotazioni standard su di un'unica classe.

Soluzioni esercizi modulo 19

Esercizio 19.a)
Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. **Falso** è un tipo annotazione
2. **Falso** è un tipo annotazione
3. **Vero**
4. **Falso** un metodo di un annotazione, non può avere come tipo di ritorno `void`
5. **Falso** un metodo di un annotazione, non può avere parametri in input
6. **Vero**
7. **Falso** infatti, è legale sia il codice del metodo m(), sia il dichiarare public come primo modificatore. Non è legale però passare in input all'annotazione il valore `MiaAnnotazione.MiaEnum.VERO`, senza specificare una sintassi del tipo chiave= valore.
8. **Falso** infatti, la sintassi:
   `return @MiaAnnotazione.miaEnum;`
   non è valida. Non si può utilizzare un'annotazione come se fosse una classe con variabili statiche pubbliche…
9. **Falso** infatti l'annotazione in questione non è a valore singolo, perché il suo unico elemento non si chiama `value()`
10. **Vero**

Esercizio 19.b)
Annotazioni e libreria, Vero o Falso:

1. **Vero** infatti se non si specifica con la meta annotazione `Target`, quale sono gli elementi a cui è applicabile l'annotazione in questione, l'annotazione sarà di default applicabile a qualsiasi elemento
2. **Falso** il valore `ElementType.SOURCE` non esiste
3. **Falso** il valore `ElementType.@INTERFACE` *non esiste*
4. **Falso** non è neanche applicabile a metodi per causa del valore di `Target` che è `ElementType.ANNOTATION_TYPE`
5. **Falso** infatti non può essere applicata ad una classe se è annotata con `@Target (ElementType.METHOD)`
6. **Vero**

7. **Falso** al compilatore non al runtime
8. **Vero**
9. **Vero**
10. **Vero** `Override` non è applicabile a classi

Obiettivi del modulo)
Sono stati raggiunti i seguenti obiettivi?:

Obiettivo	Raggiunto	In Data
Comprendere cosa sono i metadati e la loro relatività (unità 19.1, 19.2)	☐	
Comprendere l'utilità delle annotazioni (unità 19.1, 19.2, 19.3, 19.4)	☐	
Saper definire nuove annotazioni (unità 19.2)	☐	
Saper annotare elementi Java ed altre annotazioni (unità 19.2, 19.3)	☐	
Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta annotazioni (unità 19.3, 19.4)	☐	

Note:

Ed ora?

Ora il lettore dovrebbe avere solide radici per immergersi nella tecnologia Java. Infatti esistono mille altri argomenti da studiare! Ma non c'è da scoraggiarsi, anzi, possiamo affermare che la parte più difficile è stata superata! Conoscere "seriamente" il linguaggio e il supporto che offre all'object orientation significa non avere problemi ad affrontare argomenti "avanzati", ovvero le tecnologie. La maggior parte degli argomenti avanzati, infatti, è semplice da imparare! I problemi potrebbero semmai arrivare non da Java, ma magari dalla non conoscenza delle tecnologie correlate (per esempio XML, SQL, Javascript e così via…).
Non vi resta che rimboccarvi le maniche…
Ultimo N.B. : Se il lettore sta leggendo queste ultime righe, probabilmente ha trovato interessante (nel bene e nel male) questo manuale! L'autore gradirebbe ricevere un suo feedback, al fine di apportare modifiche migliorative in futuro. Nessun obbligo ovviamente, ma se lo ritenesse opportuno è possibile utilizzare l'indirizzo e-mail claudio@claudiodesio.com.
Grazie e ancora buon lavoro!

Claudio De Sio Cesari

Appendice A

Comandi base per interagire con la riga di comando di Windows

Il sistema operativo Dos, non ha interfaccia grafica a finestre. Per navigare tra le cartelle, o per fare un qualsiasi altro tipo di operazione sui file, bisogna digitare un comando. Segue una sotto-lista di comandi basilari che dovrebbero permettere al lettore di affrontare lo studio del testo in questione senza problemi.

Ogni comando descritto deve essere seguito dalla pressione del tasto "invio" ("enter") sulla tastiera, per avere effetto.

Comando	Spiegazione
cd nomeCartella	Spostamento in una cartella contenuta nella cartella in cui ci si trova
cd..	Spostamento nella cartella che contiene la cartella in cui ci si trova.
dir	Lista il contenuto della cartella in cui ci si trova con un allineamento verticale
dir/w	Lista il contenuto della cartella in cui ci si trova con un allineamento orizzontale.
dir/p	Lista il contenuto della cartella in cui ci si trova con un allineamento verticale. Se i file da listare eccedono la disponibilità visiva della finestra vengono visualizzati solo i primi file che rientrano nella prima finestra. Alla pressione di un qualsiasi ulteriore tasto, il sistema visualizzerà la schermata successiva.
Control c	(pressione del tasto "ctrl" contemporaneamente al tasto "c") Interrompe il processo in corso (utile nel caso di processo con ciclo di vita infinito).

Appendice B

Preparazione dell'ambiente operativo su sistemi operativi Microsoft Windows: installazione del Java Development Kit

1. Scaricare il JDK Standard Edition da http://java.sun.com, possibilmente la versione più recente. Il nome del file varia da un rilascio all'altro, ma non di molto. Quello della versione 1.6.0 beta è "jdk-6-beta-windows-i586.exe", ed il file è ampio circa 51 Mb.
2. Scaricare anche la documentazione (si tratta di un file .zip). Anche qui il nome del file varia da un rilascio all'altro; per la versione 1.6.0 beta è "jdk-6-beta-doc.zip", ed il file è ampio circa 50 Mb. Una volta ottenuto il file di installazione, eseguirlo e rispondere sempre positivamente a tutte le domande poste (accetti la licenza? Deve installare qui il JDK?…)
3. Impostare la variabile d'ambiente PATH, facendola puntare alla cartella "bin" del JDK:
    a. per i sistemi Windows 2000/XP/NT, eseguire i seguenti passi:
    Tasto destro su "Risorse del Computer", fare clic su "Proprietà". Selezionare il tab "Avanzate" e cliccare su "Variabili d'ambiente" (su Windows NT selezionare solo il tab "Variabili d'ambiente"). Tra le "Variabili di sistema" (o se preferite tra le "Variabili utente"), selezionare la variabile PATH e fare clic su "Modifica". Spostarsi nella casella "Valore variabile" e portarsi con il cursore alla fine della riga. Se non c'è già, aggiungere un ";". Infine aggiungere il percorso alla cartella "bin" del JDK, che dovrebbe essere simile a :
    C:\programmi\java\jdk1.6.0\bin
    Fare clic su "OK" e l'installazione è terminata.
    b. Per i sistemi Windows 95/98, eseguire i seguenti passi:
    Fare clic su "Start" e poi su "Esegui". Inserire il comando "sysedit" e poi premere Invio. Selezionare la finestra "Autoexec.bat". Cercare l'istruzione "PATH = …" (se non c'è, aggiungerla) e portarsi con il cursore alla fine dell'istruzione. Se non c'è già, aggiungere un ";". Aggiungere il percorso alla cartella bin del JDK, che dovrebbe essere simile a :
    C:\programmi\java\jdk1.6.0\bin
    Fare clic su "OK" e riavviare la macchina: l'installazione è terminata.

c. Per i sistemi Windows ME, eseguire i seguenti passi:
   Dal menu Start, scegliere “Programmi”, “Accessori”, “Strumenti di sistema” e “Informazioni di sistema”. Scegliere il menu “Strumenti” dalla finestra che viene presentata e fare clic su “Utilità di configurazione del sistema”. Fare clic sul tab “Ambiente”, selezionare PATH e premere “Modifica”. Portarsi con il cursore alla fine della riga. Se non c’è già, aggiungere un “;”. Aggiungere il percorso alla cartella “bin” del JDK, che dovrebbe essere simile a :
   C:\programmi\java\jdk1.6.0\bin
   Fare clic su “OK” e riavviare la macchina: l’installazione è terminata.

Appendice C

Documentazione di EJE (Everyone's Java Editor)

C.1 Requisiti di sistema

Per poter essere eseguito, EJE ha bisogno di alcuni requisiti di sistema. Per quanto riguarda l'hardware, la scelta minima deve essere la seguente:
Memoria RAM, minimo 16 Mb (raccomandati 32 Mb)
Spazio su disco fisso: 542 kb circa
Per quanto riguarda il software necessario ad eseguire EJE:
Piattaforma Java: Java Platform Standard Edition, v1.4.x (j2sdk1.4.x) o superiore
Sistema operativo: Microsoft Windows 9x/NT/ME/2000/XP (TM), Fedora Core 5, 4, 3, 2 & Red Hat Linux 9 (TM), Sun Solaris 2.8 (TM). Non testato su altri sistemi operativi…

C.2 Installazione ed esecuzione di EJE

EJE è un programma multipiattaforma, ma richiede due differenti script per essere eseguito sui sistemi operativi Windows o Linux. Sono quindi state create due distribuzioni, ma queste si differenziano solo per gli script di lancio…

C.2.1 Per utentiWindows (Windows 9x/NT/ME/2000/XP):

Una volta scaricato sul vostro computer il file eje.zip:

1. Scompattare tramite un utility di zip come WinRar o WinZip il file eje.zip
2. Lanciare il file eje.bat (eje_win9x.bat per windows 95/98) con un doppio clic.

C.2.2 Per utenti di sistemi operativi Unix-like (Linux, Solaris...):

Una volta scaricato sul computer il file eje.tar.gz (o eje.tar.bz2):

1. Dezippare tramite gzip (o bzip2) il file eje.tar.gz (o eje.tar.bz2). Verrà creato il file eje.tar.
2. Scompattare il file eje.tar. Verrà creata la directory EJE.
3. Cambiare i permessi del file eje.sh con il comando chmod. Da riga di comando digitare:
   chmod a+x eje.sh

4. Lanciare il file eje.sh nella directory EJE.

C.2.3 Per utenti che hanno problemi con questi script (Windows 9x/NT/ME/2000/XP & Linux, Solaris):

Da riga di comando (prompt DOS o shell Unix) digitare il seguente comando:
java -classpath . com.cdsc.eje.gui.EJE

> **Se il sistema operativo non riconosce il comando, allora non avete installato il Java Development Kit (1.4 o superiore), oppure non avete settato la variabile d'ambiente PATH alla directory bin del JDK (vedi "installation notes" del JDK).**

C.3 Manuale d'uso

EJE è un semplice e leggero editor per programmare in Java. Esistono tanti altri strumenti per scrivere codice Java, ma spesso si tratta di pesanti IDE che hanno bisogno a loro volta di un periodo di apprendimento piuttosto lungo per essere sfruttati con profitto. Inoltre tali strumenti, hanno bisogno di ampie risorse di sistema, che potrebbero o essere assenti, o non necessarie per lo scopo dello sviluppatore.
EJE si propone come editor Java, non per sostituire gli strumenti di cui sopra, bensì per scrivere codice in maniera veloce e personalizzata. E' stato creato pensando appositamente a chi si avvicina al linguaggio, ed è infatti scaricabile gratuitamente insieme al manuale "Object Oriented && Java 5", all'indirizzo http://www.claudiodesio.com/download/oo_&&_java_5.zip (per maggiori informazioni visitare http://www.claudiodesio.com).
Le principali caratteristiche di EJE (versione 2.7):

1. Possibilità di compilare ed eseguire file (anche con argomenti) direttamente dall'editor (anche appartenenti a package)
2. Supporto a file multipli (ma non a processi multipli) tramite tab
3. Colorazione delle parole significative per la sintassi Java
4. Veloce esplorazione del file system tramite un'alberazione delle directory, e possibilità di alberare cartelle di lavoro
5. Navigabilità completa di tutte le funzionalità tramite tastiera
6. Possibilità di annullare e riconfermare l'ultima azione per un numero infinito di volte
7. Utilità di ricerca e sostituisci espressioni nel testo

8. Inserimenti dinamici template di codice (è anche possibile selezionare del testo per poi circondarlo con template di codice), e di attributi incapsulati (proprietà JavaBean)
9. Personalizzazione dello stile di visualizzazione
10. Possibiltà di commentare testo selezionato
11. Possibilità di impostare messaggi da visualizzare dopo uno specificato periodo di tempo
12. Popup di introspezioni classi automatico per visualizzare i membri da utilizzare dopo aver definito un oggetto
13. Possibilità di aprire la documentazione della libreria standard del JDK in un browser java
14. Possibilità di generare documentazione automaticamente dei propri sorgenti mediante l'utility javadoc
15. Indentazione automatica del codice in stile C o Java
16. Navigazione veloce tra file aperti
17. E' possibile settare molte opzioni: tipo, stile e dimensione del font, abilitazione e disabilitazione del popup di introspezione, stile di indentazione, compilazione in base alla versione di java di destinazione, abilitare-disabilitare asserzioni, lingua, stile del look and feel, Java Development Kit, documentazione, classpath, directory di output etc...
18. E' possibile stampare i file sorgente
19. Supporto a Java versione 6

L'interfaccia che EJE mette a disposizione dello sviluppatore è molto semplice ed intuitiva. La figura 1) mostra EJE in azione (EJE si mostrerà nella versione inglese se lanciato su di un sistema operativo in lingua non italiana).

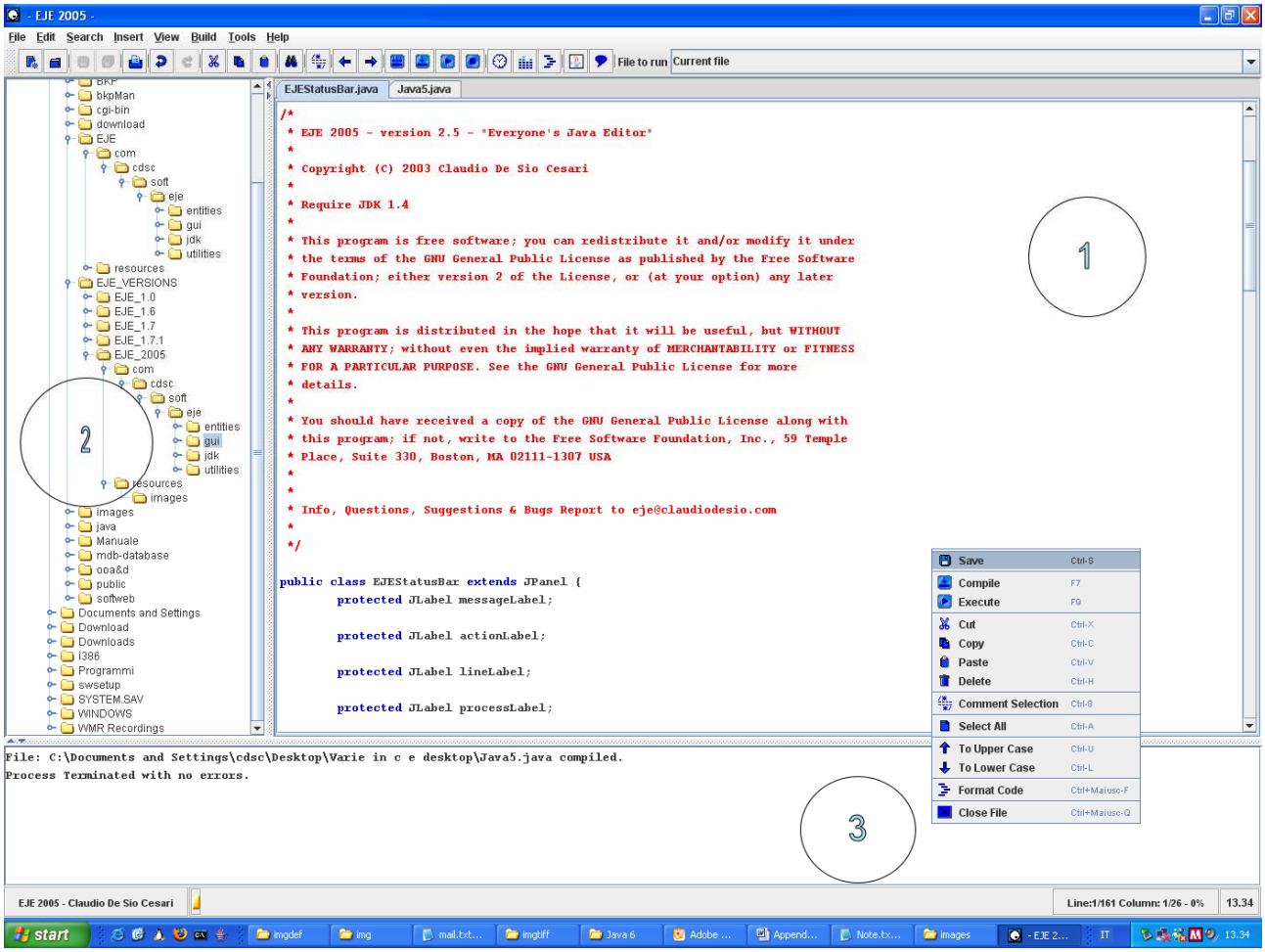


Figura C.1 – "Eje in azione"

Il pannello 1 mostra l'alberatura delle cartelle disponibili nel file system. Il contenuto delle cartelle non è visibile a meno che non si tratti di file sorgenti Java. E' possibile aprire nel pannello tali file facendo clic su essi.
Il pannello 2 mostrerà il contenuto dei file aperti.
Il pannello 3 invece mostrerà i messaggi relativi ai processi mandati in esecuzione dall'utente, come la compilazione e l'esecuzione dei file Java.

C.4 Tabella descrittiva dei principali comandi di EJE:

Comando	Icon	Dove si trova	Scorciatoi	Sinossi

	a		**a**	
Nuovo		Menù File, Barra degli strumenti	CTRL-N	Crea un nuovo file
Apri		Menù File, Barra degli strumenti	CTRL-O	Apre un file presente nel file-system
File recenti...		Menù File		Permette di aprire un file aperto recentemente
Salva		Menù File, Barra degli strumenti, Popup su area testo	CTRL-S	Salva il file corrente
Salva Tutto		Menù File, Barra degli strumenti	CTRL-SHIFT-S	Salva tutti i file aperti
Salva con nome		Menù File		Salva un file con un nome
Stampa...		Menù File, Barra degli strumenti	CTRL-P	Stampa il file corrente
Options		Menù File	F12	Apre la finestra delle opzioni
Chiudi File		Menù File, Popup su area testo	CTRL-SHIFT-Q	Chiude il file corrente
Esci		Menù File	CTRL-Q	Termina EJE
Annulla		Menù Modifica, Barra degli strumenti	CTRL-Z	Annulla l'ultima azione
Ripeti		Menù Modifica, Barra degli strumenti	CTRL-Y	Ripeti ultima azione
Taglia		Menù Modifica, Barra degli strumenti, Popup su area testo	CTRL-X	Sposta selezione negli appunti
Copia		Menù Modifica, Barra degli strumenti, Popup su area testo	CTRL-C	Copia selezione negli appunti
Incolla		Menù Modifica, Barra degli strumenti, Popup su area testo	CTRL-V	Incolla appunti
Cancella		Menù Modifica, Popup su area testo		Cancella selezione
Seleziona Tutto		Menù Modifica, Popup su area testo	CTRL-A	Seleziona tutto il testo

Rendi Maiuscolo		Menù Modifica, Popup su area testo	CTRL-U	Rende maiuscolo testo selezione
Rendi Minuscolo		Menù Modifica, Popup su area testo	CTRL-L	Rende minuscolo testo selezione
Trova		Menù Cerca, Barra degli strumenti	CTRL-F	Cerca espressioni nel testo
Trova Successivo		Menù Cerca	F3	Cerca la successiva espressione nel testo
Sostituisci		Menù Cerca	CTRL-H	Cerca e sostituisce espressioni nel testo
Vai alla riga		Menù Cerca	CTRL-G	Sposta il cursore alla riga
Template di classe		Menù Inserisci	CTRL-0	Inserisce (o circonda il testo selezionato con) un template di classe
Metodo main		Menù Inserisci	CTRL-1	Inserisce (o circonda il testo selezionato con) un template di metodo main
If		Menù Inserisci	CTRL-2	Inserisce (o circonda il testo selezionato con) un template di costrutto if
Switch		Menù Inserisci	CTRL-3	Inserisce (o circonda il testo selezionato con) un template di costrutto switch
For		Menù Inserisci	CTRL-4	Inserisce (o circonda il testo selezionato con) un template di costrutto for
While		Menù Inserisci	CTRL-5	Inserisce (o circonda il testo selezionato con) un template di costrutto while
Do While		Menù Inserisci	CTRL-6	Inserisce (o circonda il testo selezionato con) un template di costrutto do-while
Try/catch		Menù Inserisci	CTRL-7	Inserisce (o circonda il testo selezionato con) un template di blocco try-catch
Commenta		Menù Inserisci	CTRL-8	Inserisce (o circonda il testo

selezione				selezionato con) un template commento
Proprietà JavaBean		Menù Inserisci	CTRL-9	Apre wizard per creare proprietà JavaBean
Barra degli strumenti		Menù Visualizza		Nasconde-visualizza barra degli strumenti
Barra di stato		Menù Visualizza		Nasconde-visualizza barra di stato
Scagli cartella di lavoro		Menù Strumenti		Permette di scegliere una cartella di lavoro che verrà aperta nell'alberatura del pannello 1
Prossimo file		Menù Visualizza, Barra degli strumenti	F5	Seleziona il prossimo file
File precedente		Menù Visualizza, Barra degli strumenti	F4	Seleziona il file precedente
Compila tutto		Menù Sviluppo, Barra degli strumenti, Popup su area testo	SHIFT-F7	Compila tutti i file aperti
Compila		Menù Sviluppo, Barra degli strumenti, Popup su area testo	F7	Compila file corrente
Esegui		Menù Sviluppo, Barra degli strumenti, Popup su area testo	F9	Esegue file corrente
Esegui con argomenti		Menù Sviluppo	SHIFT-F9	Esegue file corrente sfruttando gli argomenti specificati
Interrompi processo		Menù Sviluppo, Barra degli strumenti		Interrompe processo corrente
Sveglia		Menù Strumenti		Permette di impostare un timeout per mostrare un messaggio
Monitor Risorse		Menù Strumenti		Mostra la memoria allocata e usata da EJE
Genera Documentazione		Menù Strumenti		Genera documentazione javadoc del file corrente
Indenta il		Menù Strumenti,	CTRL-	Indenta il codice

codice		Barra degli strumenti, Popup su area testo	SHIFT-F	
Commenta Selezione		Menù Strumenti	CTRL-SHIFT-C	Commenta il testo selezionato
Guida all'utilizzo		Menù Aiuto	F1	Mostra questo manuale utente
Documentazione Java		Menù Aiuto	F2	Mostra la documentazione della libreria standard Java
Informazioni su EJE		Menù Aiuto	CTRL-F1	Visualizza informazioni su EJE

Appendice D

Model View Controller Pattern (MVC)

D.1 Introduzione

Il pattern in questione è molto famoso ma è spesso utilizzato con superficialità dagli sviluppatori. Ciò è probabilmente dovuto alla sua complessità, dal momento che stiamo parlando di una vera e propria "composizione di pattern". Venne introdotto nel mondo del software per la costruzione di interfacce grafiche con Smalltalk-80, ma oggi deve gran parte della sua fama a Java. L'MVC è stato infatti utilizzato per la struttura di alcuni componenti Swing e, soprattutto, è coerente con l'architettura Java 2 Enterprise Edition (J2EE).
Questo documento è liberamente ispirato, per quanto riguarda la sua struttura, alla descrizione fornita proprio dal catalogo dei pattern J2EE (Java Blueprints).

D.2 Contesto

L'applicazione deve fornire una interfaccia grafica (GUI) costituita da più schermate, che mostrano vari dati all'utente. Inoltre le informazioni che devono essere visualizzate devono essere sempre quelle aggiornate.

Questo punto non è solitamente valido per architetture come J2EE, dove le GUI non sono costituite da applicazioni, ma da pagine HTML. Con l'avvento della tecnologia Ajax però, le cose stanno cambiando…

D.3 Problema

L'applicazione deve avere una natura modulare e basata sulle responsabilità, al fine di ottenere una vera e propria applicazione component-based. Questo è conveniente per poter più facilmente gestire la manutenzione dell'applicazione. Per esempio, ai nostri giorni, con la massiccia diffusione delle applicazioni enterprise, non è possibile

prevedere al momento dello sviluppo in che modo e con quale tecnologia gli utenti interagiranno con il sistema (WML?, XML?, Wi-Fi?, HTML?). Appare quindi chiaro il bisogno di un'architettura che permetta la separazione netta tra i componenti software che gestiscono il modo di presentare i dati e i componenti che gestiscono i dati stessi.

D.4 Forze

1. E' possibile accedere alla gestione dei dati con diverse tipologie di GUI (magari sviluppate con tecnologie diverse)
2. I dati dell'applicazione possono essere aggiornati tramite diverse interazioni da parte dei client (messaggi SOAP, richieste HTTP…)
3. Il supporto di varie GUI ed interazioni non influisce sulle funzionalità di base dell'applicazione

D.4.1 Soluzione e struttura

L'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business dai componenti che implementano la logica di presentazione e di controllo che utilizzano tali funzionalità. Vengono quindi definite tre tipologie di componenti che soddisfano tali requisiti:

1. il Model, che implementa le funzionalità di business
2. la View, che implementa la logica di presentazione
3. il Controller, che implementa la logica di controllo

La seguente figura D.1 rappresenta un diagramma di interazione, che evidenzia le responsabilità dei tre componenti.

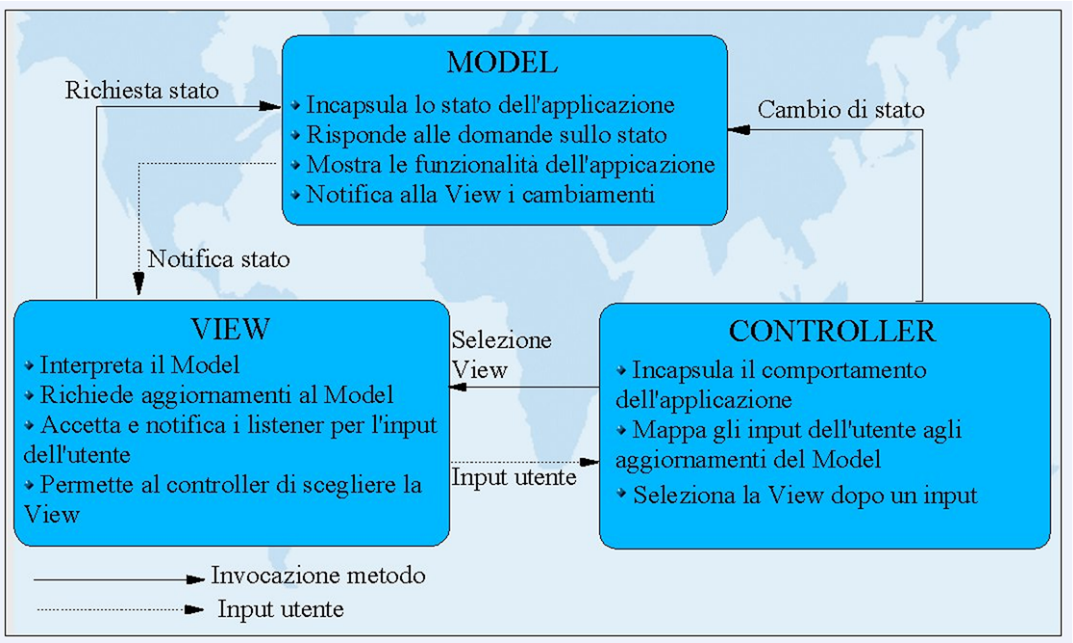


Figura D.1 – "MVC: diagramma di interazione"

D.6 Partecipanti e responsabilità

MODEL:
Analizzando la figura D.1, si evince che il core dell'applicazione viene implementato dal Model, il quale incapsulando lo stato dell'applicazione definisce i dati e le operazioni che possono essere eseguite su questi. Quindi definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento. Per lo sviluppo del Model quindi è vivamente consigliato utilizzare le tipiche tecniche di progettazione object-oriented, al fine di ottenere un componente software che astragga al meglio concetti importati dal mondo reale. Il Model può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View di presentare agli occhi degli utenti dati sempre aggiornati.

VIEW:
La logica di presentazione dei dati viene gestita solo e solamente dalla View. Ciò implica che questa deve fondamentalmente gestire la costruzione dell'interfaccia grafica (GUI), che rappresenta il mezzo mediante il quale gli utenti interagiranno con il sistema.

Ogni GUI può essere costituita da schermate diverse che presentano più modi di interagire con i dati dell'applicazione. Per far sì che i dati presentati siano sempre aggiornati è possibile adottare due strategie, note come "push model" e "pull model". Il push model adotta il pattern Observer, registrando le View come osservatori del Model. Le View possono quindi richiedere gli aggiornamenti al Model in tempo reale grazie alla notifica di quest'ultimo. Benché questa rappresenti la strategia ideale, non è sempre applicabile. Per esempio nell'architettura J2EE, se le View che vengono implementate con JSP restituiscono GUI costituite solo da contenuti statici (HTML) e quindi non in grado di eseguire operazioni sul Model. In tali casi è possibile utilizzare il "pull Model", dove la View richiede gli aggiornamenti quando "lo ritiene opportuno". Inoltre la View delega al Controller l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input e la scelta delle eventuali schermate da presentare.

CONTROLLER:
Questo componente ha la responsabilità di trasformare le interazioni dell'utente della View in azioni eseguite dal Model. Ma il Controller non rappresenta un semplice "ponte" tra View e Model. Realizzando la mappatura tra input dell'utente e processi eseguiti dal Model e selezionando la schermate della View richieste, il Controller implementa la logica di controllo dell'applicazione.

D.7 Collaborazioni

In figura D.2 viene evidenziata, mediante un diagramma delle classi, la vera natura del pattern MVC. In pratica, View e Model sono relazionati tramite il pattern Observer, dove la View "osserva" il Model. Il pattern Observer caratterizza anche il legame tra View e Controller, ma in questo caso è la View ad essere "osservata" dal Controller. Ciò supporta la registrazione dinamica al runtime dei componenti. Inoltre il pattern Strategy potrebbe semplificare la possibilità di cambiare la mappatura tra gli algoritmi che regolano i processi del Controller e le interazioni dell'utente con la View.

Che si tratti di un pattern che ne contiene altri, risulta abbastanza evidente. Ma nella sua natura originaria l'MVC comprendeva anche l'implementazione di altri pattern, come il Factory Method per specificare la classe Controller di default per una View, il Composite per costruire View ed il Decorator per aggiungere altre proprietà (per esempio lo scrolling).

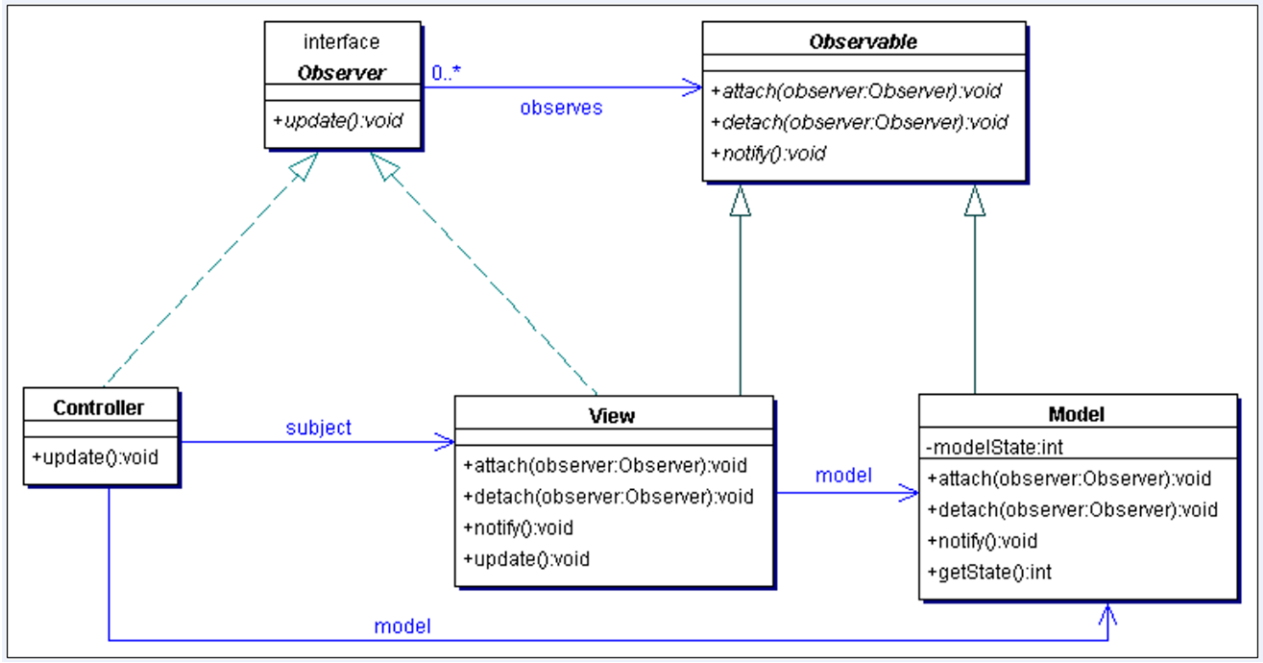


Figura D.2 – "MVC: diagramma delle classi"

Evidenziamo ora solo due dei possibili scenari che potrebbero presentarsi utilizzando un applicazione MVC-based: la richiesta dell'utente di aggiornare dati e la richiesta dell'utente di selezionare una schermata, rispettivamente illustrate tramite diagrammi di sequenze nelle figure D.3 e D.4.

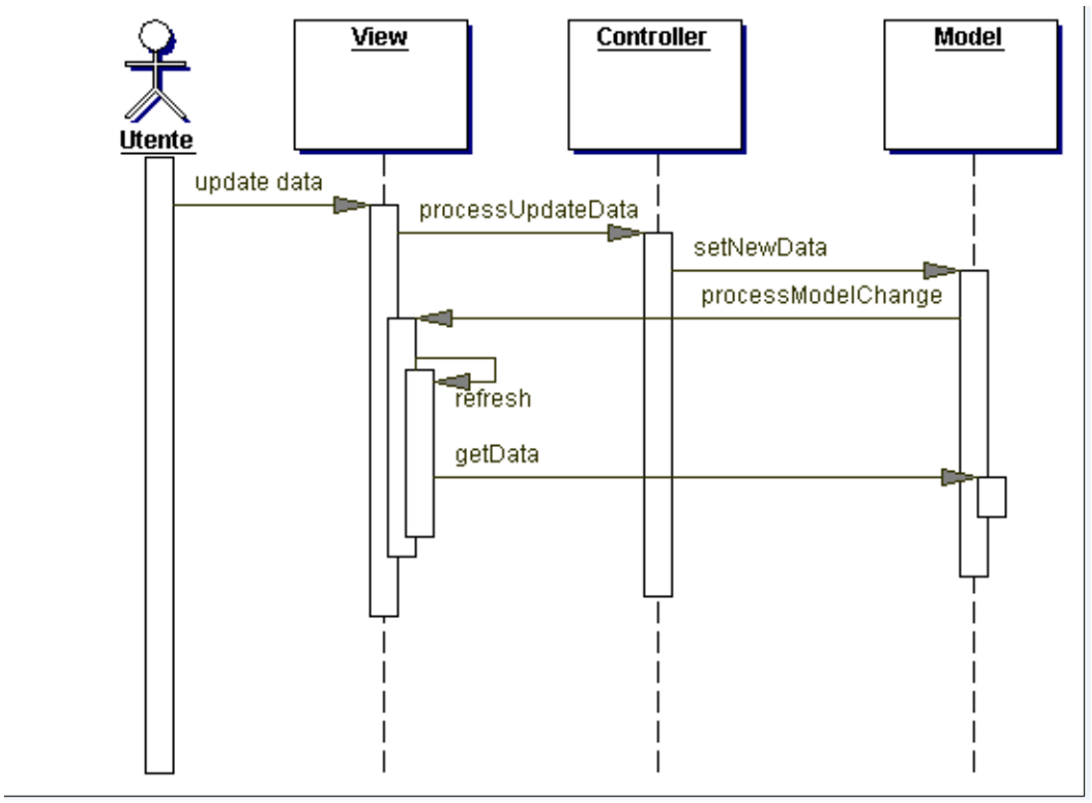


Figura D.3 – "Scenario aggiornamento dati"

Dalla figura D.3 evidenziamo il complesso scambio di messaggi tra i tre partecipanti al pattern il quale, benché possa sembrare inutile a prima vista, garantisce pagine sempre aggiornate in tempo reale all'utente.

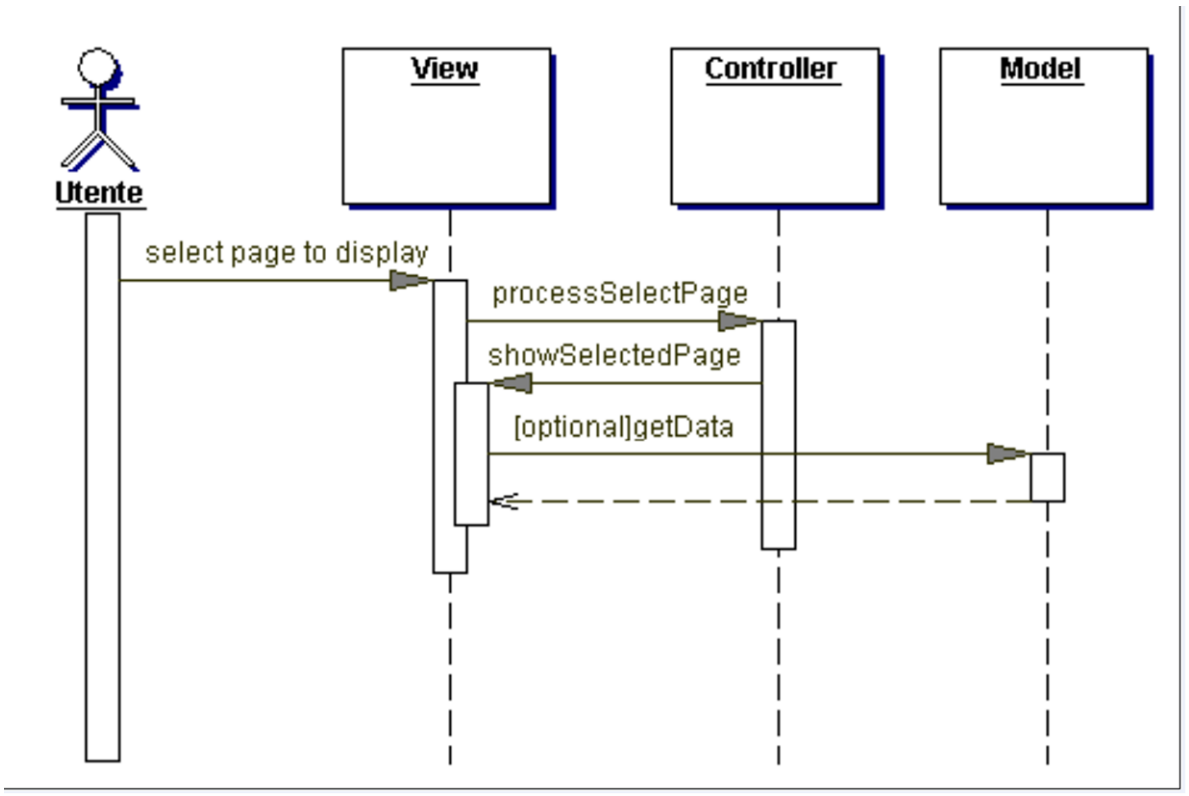


Figura D.4 – "Scenario selezione schermata"

Nella figura D.4 vengono enfatizzati i ruoli di View e Controller. La View, infatti non decide quale sarà la schermata richiesta, delegando ciò alla decisione del Controller (che grazie al pattern Strategy può anche essere anche cambiato dinamicamente al runtime); piuttosto si occupa della costruzione e della presentazione all'utente della schermata stessa.

D.8 Codice d'esempio

Nell'esempio presentato di seguito viene implementata una piccola applicazione che simula una chat ma funziona solo in locale, allo scopo di rappresentare un esempio semplice di utilizzo del pattern MVC. E' un esempio didattico e quindi si raccomanda al lettore di non perdersi nei dettagli inutili, ma si consiglia piuttosto, di concentrarsi sulle

caratteristiche del pattern. Nella figura D.5 è rappresentato il diagramma delle classi della chat dal punto di vista dell'implementazione.

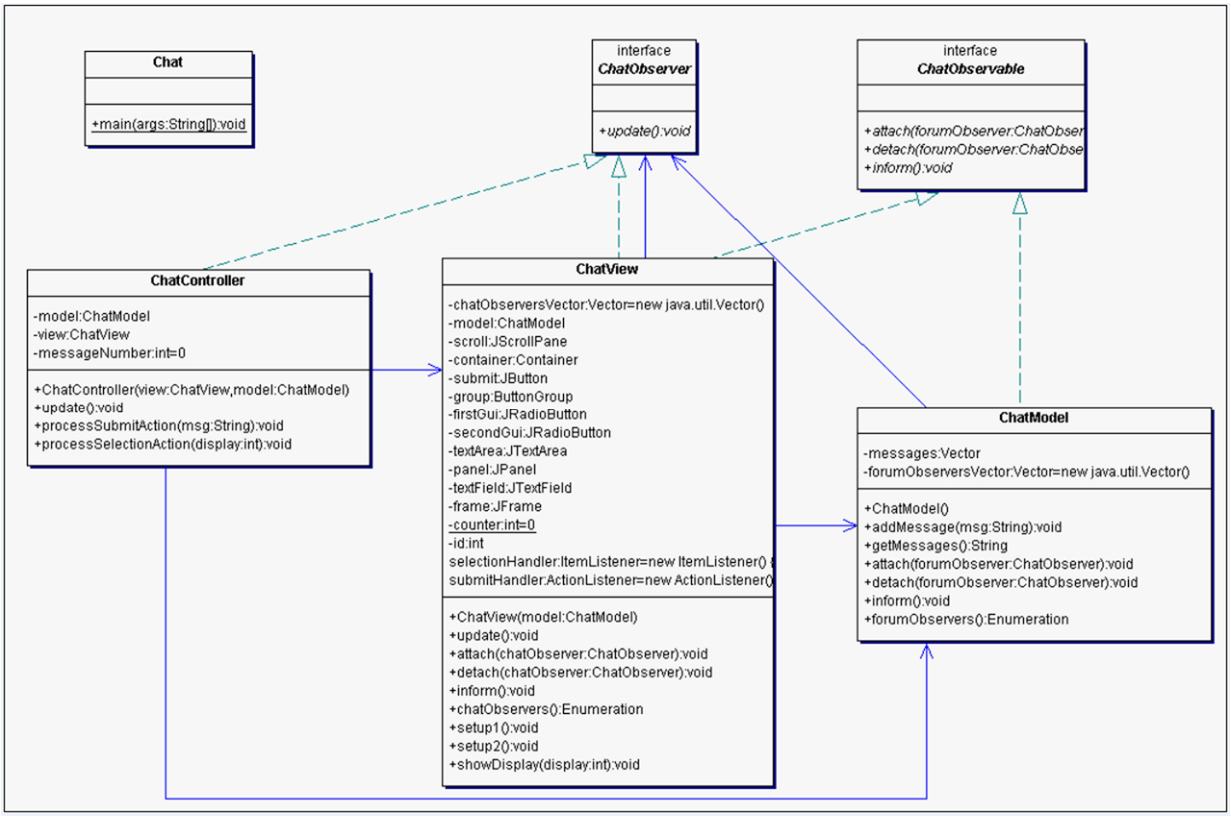


Figura D.5 – "Diagramma delle classi dal punto di vista dell'implementazione"

La classe `Chat` definisce il metodo `main()` che semplicemente realizza lo startup dell'applicazione, così come descritto dal diagramma di sequenza in figura D.6.

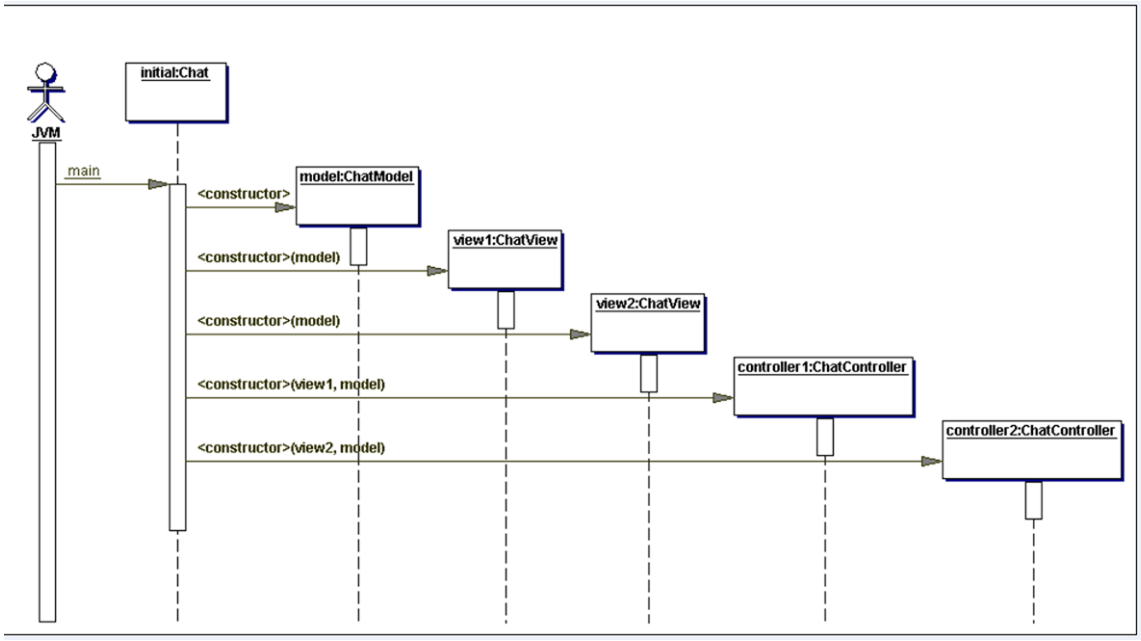


Figura D.6 – "Startup dell'applicazione"

La classe `ChatView` definisce due metodi, `setup1()` e `setup2()`, per realizzare due schermate (molto simili in realtà, ma è solo un esempio). Essa definisce anche due classi interne anonime che fungono da listener. I listener, una volta notificati, delegano all'oggetto (o agli oggetti) `ChatController` (registratisi in fase di startup come observer della View) tramite cicli di notifica come il seguente:

```
public void itemStateChanged(ItemEvent e) {
  ChatController con;
  for (int i = 0; i < chatObserversVector.size(); i++) {
    con = (ChatController)chatObserversVector.get(i);
    con.processSelectionAction(e.getStateChange());
  }
}
```

Di seguito è riportata l'intera classe `ChatController`:

```
public class ChatController implements ChatObserver {
  private ChatModel model;
  private ChatView view;
  private int messageNumber = 0;
```

```
  public ChatController(ChatView view, ChatModel model) {
    this.view = view;
    this.model = model;
    view.attach(this);
  }

  public void update() {
    messageNumber++;
    System.out.println(messageNumber);
  }

  public void processSubmitAction(String msg) {
    model.addMessage(msg);
  }

  public void processSelectionAction(int display) {
    view.showDisplay(display);
  }
}
```

Si può notare come in questo esempio il metodo `update()` si limiti a incrementare un contatore di messaggi, mentre la vera logica di controllo è implementata mediante i metodi (che potrebbero essere strategy, ma in questo caso abbiamo semplificato) `processSubmitAction()` e `processSelectionAction()`.
Per avere un quadro più completo, riportiamo la classe `ChatModel` per intero (per scaricare l'intero codice in file zip basta collegarsi a http://www.claudiodesio.com/download/mvc_code.zip):

```
import java.util.*;

public class ChatModel implements ChatObservable {
  private Vector messages;
  private Vector forumObserversVector = new Vector();

  public ChatModel() {
    messages = new Vector();
  }
```

```
  public void addMessage(String msg) {
    messages.add(msg);
    inform();
  }

  public String getMessages() {
    int length = messages.size();
    String allMessages = "";
    for (int i = 0; i < length; ++i) {
      allMessages += (String)messages.elementAt(i)+"\n";
    }
    return allMessages;
  }

  public void attach(ChatObserver forumObserver){
    forumObserversVector.addElement(forumObserver);
  }

  public void detach(ChatObserver forumObserver){
    forumObserversVector.removeElement(forumObserver);
  }

  public void inform(){
    java.util.Enumeration enumeration = forumObservers();
    while (enumeration.hasMoreElements()) {
      ((ChatObserver)enumeration.nextElement()).update();
    }
  }

  public Enumeration forumObservers(){
    return ((java.util.Vector)
forumObserversVector.clone()).elements();
  }
}
```

D.9 Conseguenze

1. Riuso dei componenti del Model; la separazione tra Model e View permette a diverse GUI di utilizzare lo stesso Model. Conseguentemente, i componenti del Model sono più semplici da implementare, testare e manutenere, giacché tutti gli accessi passano tramite questi componenti.
2. Supporto più semplice per nuovi tipi di client; infatti basterà creare View e Controller per interfacciarsi ad un Model già implementato.
3. Complessità notevole della progettazione; questo pattern introduce molte classi extra ed interazioni tutt'altro che scontate.

D.10 Conclusioni

Il pattern MVC introduce una notevole complessità all'applicazione, ed è sicuramente non di facile comprensione. Tuttavia il suo utilizzo si rende praticamente necessario in tantissime applicazioni moderne, dove le GUI sono insostituibili. Non conviene avventurarsi alla scoperta di nuove interazioni tra logica di business e di presentazione, se una soluzione certa esiste già.

Appendice E

Introduzione all'HTML

HTML è l'acronimo per HyperText Markup Language, ovvero linguaggio di marcatura per ipertesti. Un ipertesto è una tipologia di documento che include, oltre che a semplice testo, collegamenti ad altre pagine.
Non si tratta di un vero linguaggio di programmazione, ma solo di un linguaggio di formattazione di pagine. Non esistono per esempio strutture di controllo come if o for.
Le istruzioni dell'HTML si dicono tag. I tag, in italiano "etichette", sono semplici istruzioni con la seguente sintassi:

```
<NOME_TAG [LISTA DI ATTRIBUTI]>
```

Dove ogni attributo, opzionale, ha una sintassi del tipo:

```
CHIAVE=VALORE
```

Il valore di un attributo potrebbe e dovrebbe essere compreso tra due apicetti o due virgolette. Ciò è però necessario se e solo se il valore è costituito da più parole separate.
Inoltre, quasi Tutti i tag HTML, salvo rare eccezioni, vanno chiusi, con una istruzione del tipo:

```
</NOME_TAG>
```

Ad esempio, il tag `<HTML>` va chiuso con `</HTML>`; il tag `<applet code='StringApplet' width='100' height='100'>` va chiuso con `</applet>`.
Per scrivere una semplice pagina HTML non occorre altro che un semplice editor di testo come il blocco note. Un browser come Internet Explorer può poi farci visualizzare la pagina formattata.
Se salviamo il seguente file di testo con suffisso "htm" o "html" contentente i seguenti tag:

```
<HTML>
  <HEAD>
    <TITLE>Prima pagina HTML</TITLE>
  </HEAD>
  <BODY>
    <CENTER>
      Hello HTML
    </CENTER>
  </BODY>
</HTML>
```

possiamo poi visualizzare il risultato aprendo il nostro file con un browser come Internet Explorer.
L'HTML non è case-sensitive.
Non resta altro che procurarci un manuale HTML (da Internet se ne possono scaricare centinaia), o semplicemente imparare da pagine già fatte andando a spiarne il codice che è sempre disponibile. Esistono tantissimi tag da scoprire…

Appendice F

Introduzione allo Unified Modeling Language

F.1 Cos' è UML (What)?

Oggigiorno si sente parlare molto spesso di UML ma non tutte le persone che parlano di UML sanno di cosa realmente si tratti. Qualcuno pensa che sia un linguaggio di programmazione, e quest' equivoco è dovuto alla parola "language". Qualcun'altro pensa si tratti di una metodologia object oriented, e questo è probabilmente dovuto a cattiva interpretazione di letture non molto approfondite. Infatti si sente spesso parlare di UML congiuntamente a varie metodologie. Per definire quindi correttamente cos' è UML, è preferibile prima definire per grandi linee, cos' è una metodologia.
Una **metodologia** object oriented, nella sua definizione più generale, potrebbe intendersi come una coppia costituita da un processo e da un linguaggio di modellazione.

> **Quella riportata per precisione è la definizione di metodo. Una metodologia è tecnicamente definita come "la scienza che studia i metodi". Spesso però, questi termini sono considerati sinonimi.**

A sua volta un **processo** potrebbe essere definito come la serie di indicazioni riguardanti i passi da intraprendere per portare a termine con successo un progetto.
Un **linguaggio di modellazione** è invece lo strumento che le metodologie utilizzano per descrivere (possibilmente in maniera grafica) tutte le caratteristiche statiche e dinamiche di un progetto.
In realtà UML non è altro che un linguaggio di modellazione. Esso è costituito per linee generali da una serie di diagrammi grafici i cui elementi sono semplici linee, triangoli, rettangoli, omini stilizzati e così via. Questi diagrammi hanno il compito di descrivere in modo chiaro, tutto ciò che durante un progetto potrebbe risultare difficile o troppo lungo con documentazione testuale.

F.2 Quando e dove è nato (When & Where)

A partire dai primi anni ottanta la scena informatica mondiale iniziò ad essere invasa dai linguaggi orientati ad oggetti quali SmallTalk e soprattutto C++. Questo perché col crescere della complessità dei software, e la relativa filosofia di progettazione, la programmazione strutturata mostrò i suoi limiti e si rivelò insufficiente per soddisfare le sempre maggiori pretese tecnologiche. Ecco allora l' affermarsi della nuova mentalità ad

oggetti e la nascita di nuove teorie, che si ponevano come scopo finale, quello di fornire delle tecniche più o meno innovative per realizzare software, ovviamente, sfruttando i paradigmi degli oggetti. Nacquero una dopo l' altra le metodologie object oriented, in gran quantità e tutte più o meno valide. Inizialmente esse erano strettamente legate ad un ben determinato linguaggio di programmazione, ma la mentalità cambiò abbastanza presto. A partire dal 1988 iniziarono ad essere pubblicati i primi libri sull' analisi e la progettazione orientata agli oggetti. Nel ' 93 si era arrivati ad un punto in cui c' era gran confusione: analisti e progettisti esperti come James Rumbaugh, Jim Odell, Peter Coad, Ivar Jacobson, Grady Booch ed altri, proponevano tutti una propria metodologia ed ognuno di loro aveva una propria schiera di entusiasti seguaci. Quasi tutti gli autori più importanti erano americani ed inizialmente le idee che non provenivano dal nuovo continente, erano accolte con sufficienza, e qualche volta addirittura derise. In particolare Jacobson prima di rivoluzionare il mondo dell' ingegneria del software con il concetto di "Use Case", fu denigrato da qualche autore americano, che definì "infantile" il suo modo di utilizzare omini stilizzati come fondamentali elementi dei suoi diagrammi. Probabilmente fu solo un tentativo di sbarazzarsi di un antagonista scomodo, o semplicemente le critiche provennero da fonti non certo lungimiranti...
Questi episodi danno però l' idea del clima di competizione in quel periodo (si parla di "guerra delle metodologie"). UML ha visto ufficialmente la luce a partire dal 1997... i tempi dovevano ancora maturare...

F.3 Perché è nato UML(Why)

Il problema fondamentale era che diverse metodologie proponevano non solo diversi processi, il che può essere valutato positivamente, ma anche diverse notazioni. Era chiaro allora a tutti che non doveva e poteva esistere uno standard tra le metodologie. Infatti i vari processi esistenti erano proprietari di caratteristiche particolarmente adatte a risolvere alcune particolari problematiche. In pratica, quando si inizia un progetto, è giusto avere la possibilità di scegliere tra diversi stratagemmi risolutivi (processi). Il fatto che ogni processo sia strettamente legato ad un determinato linguaggio di modellazione ovviamente non rappresenta altro che un intralcio per i vari membri di un team di sviluppo. L' esigenza di un linguaggio standard per le metodologie era avvertita da tanti ma nessuno degli autori aveva intenzione di fare il primo passo.

F.4 Chi ha fatto nascere UML (Who)

Ci pensò allora la Rational Software Corporation (http://www.rational.com) che annoverava tra i suoi esperti Grady Booch, autore di una metodologia molto famosa all' epoca, nota come "Booch 93". Nel ' 94 infatti James Rumbaugh, creatore della Object

Modelling Tecnique (OMT), probabilmente la più utilizzata tra le metodologie orientate agli oggetti, si unisce a Booch alla Rational. Nell' ottobre del ' 95 fu rilasciata la versione 0.8 del cosiddetto "Unified Method". Ecco che allora lo svedese Ivar Jacobson nel giro di pochi giorni si unisce a Booch e Rumbaugh, iniziando una collaborazione che poi è divenuta storica. L' ultimo arrivato portava in eredità oltre che il fondamentale concetto di "Use Case", la famosa metodologia "Object Oriented Software Engineering" (OOSE) conosciuta anche come "Objectory", che in realtà era il nome della società di Jacobson, oramai inglobata da Rational. Ecco allora che i "tres amigos" (così vennero soprannominati) iniziarono a lavorare per realizzare lo "Unified Software Development Process" ("USDP") e soprattutto al progetto UML. L' Object Management Group (OMG, http://www.omg.org), un consorzio senza scopi di lucro che si occupa delle standardizzazioni, le manutenzioni e le creazioni di specifiche che possono risultare utili al mondo dell'information technology, nello stesso periodo inoltra a tutti i più importanti autori una "richiesta di proposta" ("Request for proposal", RFP) di un linguaggio di modellazione standard. Ecco che allora Rational insieme con altri grossi partner quali IBM e Microsoft, propose la versione 1.0 di UML nell' ottobre del 1997. OMG rispose istituendo una "revision task force" (RTF) capitanata attualmente da Cris Kobryn per apportare modifiche migliorative ad UML.
La versione attuale di UML è la 2.0 ma c'è molta confusione tra gli utenti di UML. Infatti, la difficoltà di "interpretare le specifiche", i punti di vista differenti da parte degli autori più importanti e l'ancoraggio di alcuni autori alle prime versioni del linguaggio (tres amigos in testa) rendono purtroppo il quadro poco chiaro. Effettivamente OMG ha come scopo finale quello di far diventare UML uno standard ISO, e per questo le specifiche sono destinate più che agli utenti di UML, ai creatori dei tool di sviluppo di UML. Ecco perchè le specifiche hanno la forma del "famigerato meta-modello UML". Ovvero UML viene descritto tramite se stesso. In particolare il meta modello UML è diviso in 4 sezioni e giusto per avere un'idea, viene definito un linguaggio (l' "object constraint language - OCL") solo allo scopo di definire impeccabilmente la sintassi. Tutto questo nella apprezzabilissima ipotesi futura di creare applicazioni solo trascinando elementi UML uno sull'altro tramite tool come Together (http://www.borland.com)…pratica che per chi vi scrive attualmente costituisce dal 50 al 70 % del ciclo di sviluppo del software.
Intanto bisogna orientarsi tra tante informazioni diverse…

Appendice G

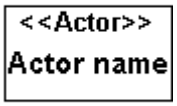
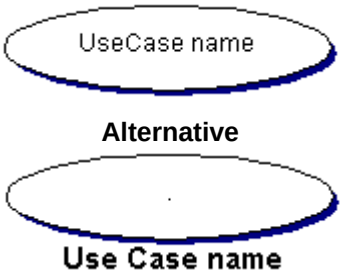
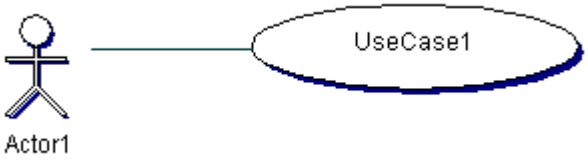
UML Syntax Reference

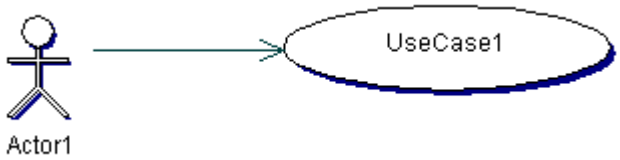
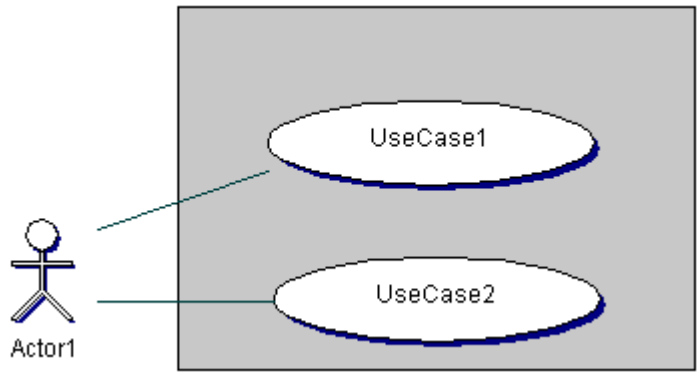
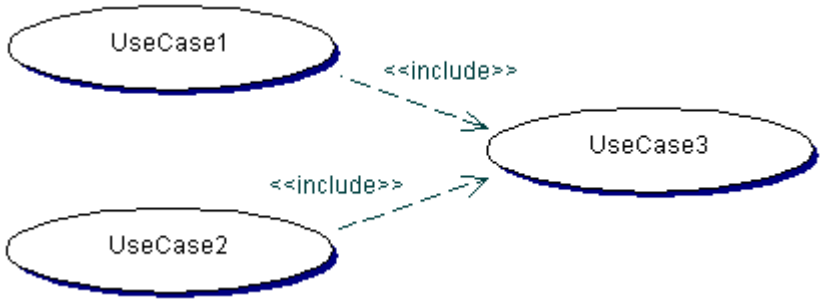
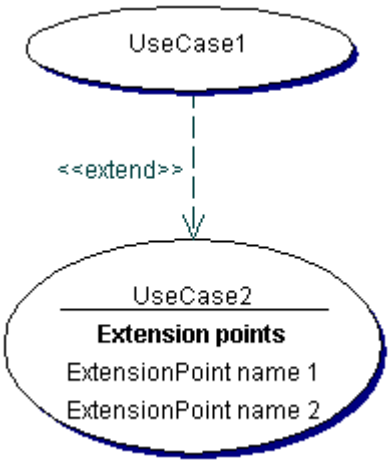
Unified Modeling Language Syntax Reference

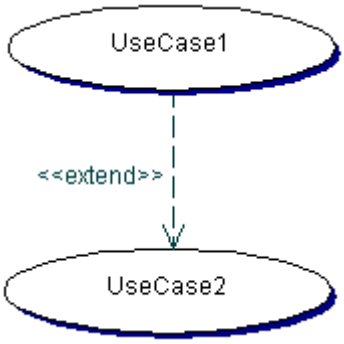
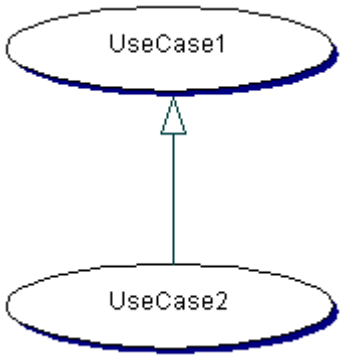
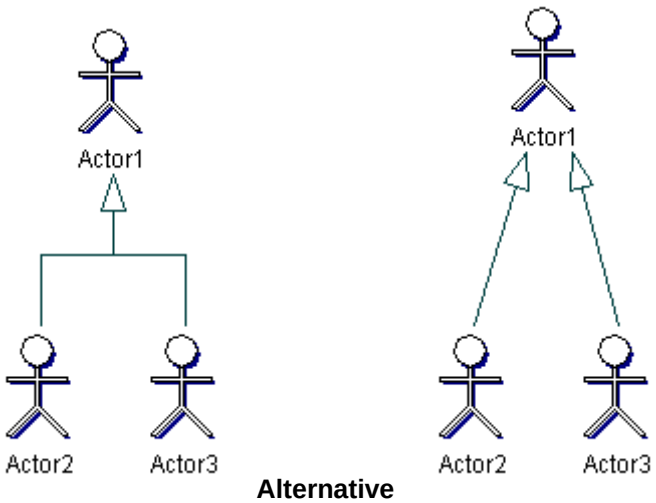
Diagram name	Element names			Synopsis
Use Case Diagram	Actor	Use Case		Diagramma dei casi d'uso: rappresentano le interazioni tra il sistema e gli utenti del sistema stesso.
	Relationship Link	System Boundary		
	Inclusion	Extension		
	Generalization	Actor Generalization		
Class Diagram	Class/Object	Association/link	Navigability	Diagramma delle classi: descrive le classi che compongono il sistema, e le relazioni statiche esistenti tra esse. NB: quando un diagramma mostra gli oggetti del sistema, spesso ci si riferisce ad esso come diagramma degli oggetti (Object Diagram).
	Attribute	Aggregation	Multiplicity	
	Operation	Composition	Qualified Association	
	Member Properties	Extension	Association Class	
	Abstract Class/Interface	Implementation	Roles Names	
Component Diagram	Component	Dependency		Diagramma dei componenti: descrive i componenti software e le loro dipendenze.
Deployment Diagram	Node	Link		Diagramma di installazione (di dispiegamento, schieramento): mostra il sistema fisico.
Interaction Diagrams:Sequence & Collaboration	Actor	Message		Diagrammi di interazione: mostrano come gruppi di oggetti collaborano in un determinato lasso temporale: Diagramma di sequenza: esalta la sequenza dei messaggi. Diagramma di collaborazione: esalta la struttura architetturale degli oggetti.
	Object	Asynchronous Message		
	Creation	Life Line		
	Destruction	Activity Line		
State Transition Diagram	State	Transition		Diagramma degli stati (di stato): descrive il comportamento di un oggetto
	Start	Action		

	End	**History**	**mostrando gli stati e gli eventi che causano i cambiamenti di stato (transizioni).**
Activity Diagram	**Activity**	**Flow**	**Diagramma delle attività: descrive i processi del sistema tramite sequenze di attività sia condizionali sia parallele.**
	Branch/Merge	**Fork/Join**	
	Swimlane	**Other elements**	
General Purpose Elements & Extension Mechanism	**Package**	**Iteration mark**	**Elementi generici e meccanismi d'estensione: elementi UML utilizzabili nella maggior parte dei diagrammi**

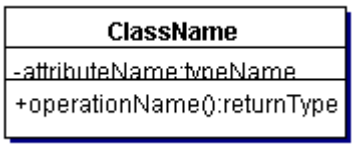
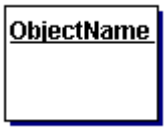
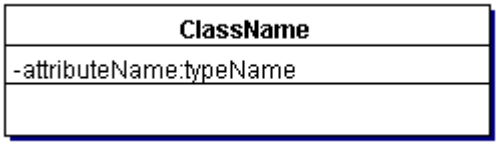
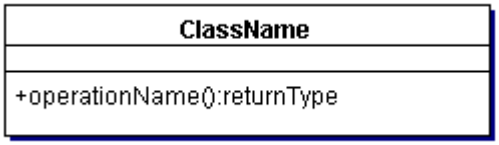
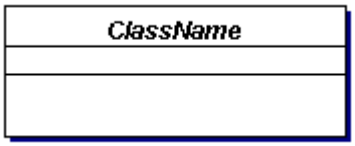
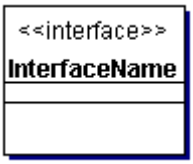
Use Case Diagram Syntax Reference

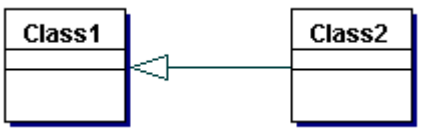
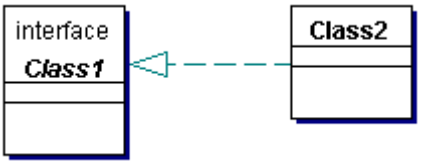
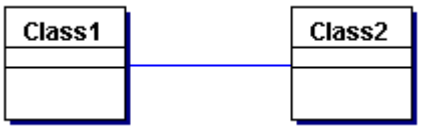
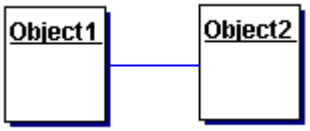
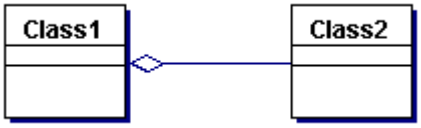
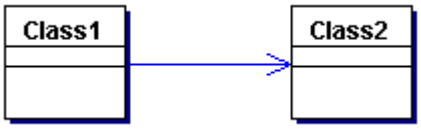
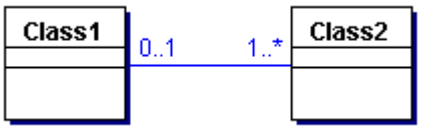
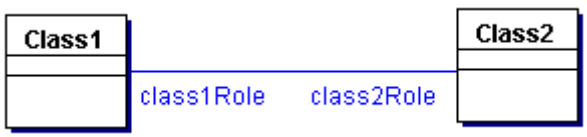
Element name	*Syntax*	*Synopsis*
Actor	 **Alternative** 	**Attore: ruolo interpretato dall'utente nei confronti del sistema. NB: un utente potrebbe non essere una persona fisica.**
Use Case		**Caso d'uso: insieme di scenari legati da un obiettivo comune per l'utente. Uno scenario è una sequenza di passi che descrivono l'interazione tra l'utenza ed il sistema. NB: È possibile descrivere scenari mediante diagrammi dinamici.**
Relationship link		**Associazione (o relazione): relazione che associa logicamente un attore ad uno caso d'uso.**

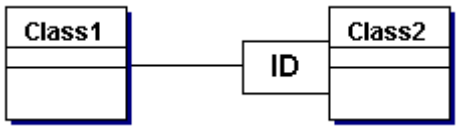
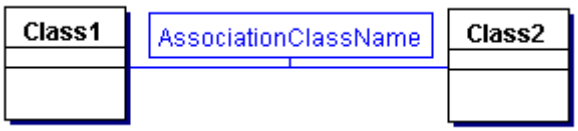
	Alternative 	
System Boundary		**Sistema (o delimitatore del sistema): delimitatore del dominio del sistema.**
Inclusion		**Inclusione: relazione logica tra casi d'uso, che estrae un comportamento comune a più casi d'uso.**
Extension	 **Alternative**	**Estensione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato, raggiunge lo scopo aggiungendo determinati punti d'estensione, che sono esplicitati nel caso d'uso base.** **Punto d'estensione: descrive un comportamento di un caso d'uso specializzato, non**

		**utilizzato dal caso d'uso base.**
Generalization		**Generalizzazione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato, raggiunge lo scopo aggiungendo nuovi comportamenti non utilizzati dal caso d'uso base, ma senza formalismi sintattici.**
Actor generalization	 **Alternative**	**Generalizzazione tra attori: relazione logica che lega attori. Un attore che specializza un attore base, può relazionarsi a qualsiasi caso d'uso relazionato al caso d'uso base. Inoltre può relazionarsi anche ad altri casi d'uso, non relazionati con il caso d'uso base.**

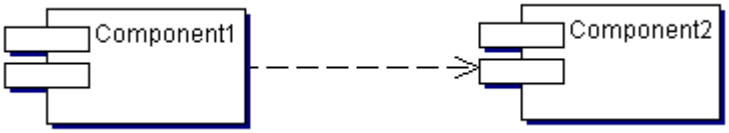
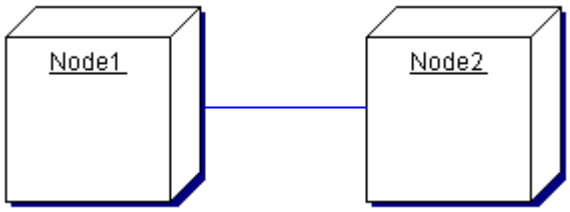
Class Diagram Syntax Reference

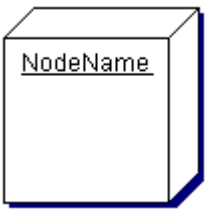
Element name	Syntax		Synopsis
Class & Object	 ***Class***	 ***Object***	**Classe: astrazione per un gruppo di oggetti che condividono stesse caratteristiche e funzionalità. Oggetto: creazione fisica (o istanza) di una classe**
Attribute			**Attributo (o variabile d'istanza o variabile membro o caratteristica di una classe): caratteristica di una classe/oggetto.**
Operation			**Operazione (o metodo o funzione membro): funzionalità di una classe/oggetto.**
Member Properties	**+ memberName *"public member"***		**Public, protected, private: modificatori di visibilità. Membro statico (o membro della classe): membro della classe. Operazione astratta: "signature" del metodo (metodo senza implementazione). Attributo derivato: attributo ricavato da altri.**
	# memberName *"protected member"*		
	- memberName *"private member"*		
	memberName *o* $memberName *"static member"*		
	operation o* operation {abstract} *"abstract operation"		
	/*attributeName *"derived attribute"		
Abstract Class & Interface	 ***Abstract Class***	 ***Interface***	**Classe astratta: classe che non si può istanziare e che può dichiarare metodi astratti. Interfaccia: struttura dati non istanziabile che può dichiarare solo metodi astratti (e costanti statiche pubbliche).**

Extension & Implementation	 ***Extension***	 ***Implementation***	**Estensione**: relazione di ereditarietà tra classi. **Implementazione**: estensione ai fini dell'implementazione dei metodi astratti di un interfaccia.
Association	 ***Association***	 ***Link***	**Associazione**: relazione tra classi dove una utilizza i servizi (le operazioni) dell'altra. **Link**: relazione tra associazione tra oggetti.
Aggregation			**Aggregazione**: associazione caratterizzata dal contenimento.
Composition			**Composizione**: aggregazione con cicli di vita coincidenti.
Navigability			**Navigabilità**: direzione di un'associazione.
Multiplicity			**Molteplicità**: corrispondenza tra la cardinalità del numero di oggetti delle classi coinvolte nell'associazione.
Role Names			**Ruoli**: descrizione del comportamento di una classe relativamente ad un'associazione.

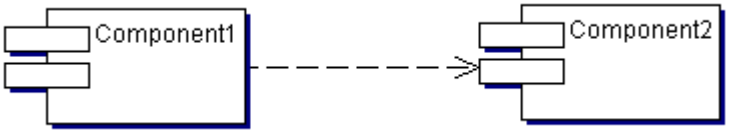
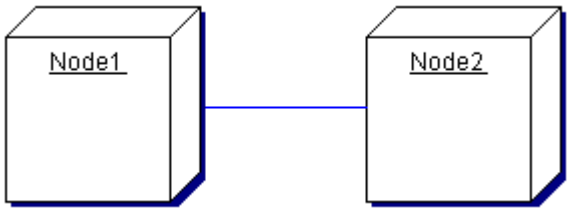
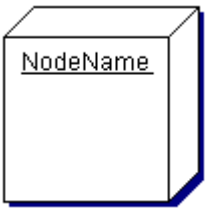
Qualified Association		**Associazione qualificata: associazione nella quale un oggetto di un classe è identificato dalla classe associata mediante una "chiave primaria".**
Association Class		**Class d'associazione: codifica in classe di un'associazione.**

Component & Deployment Diagram Syntax Reference

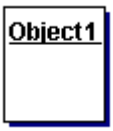
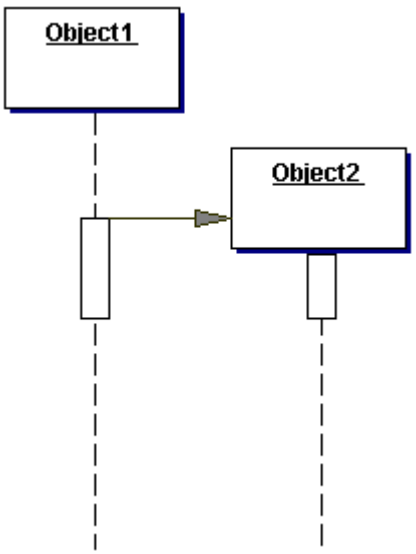
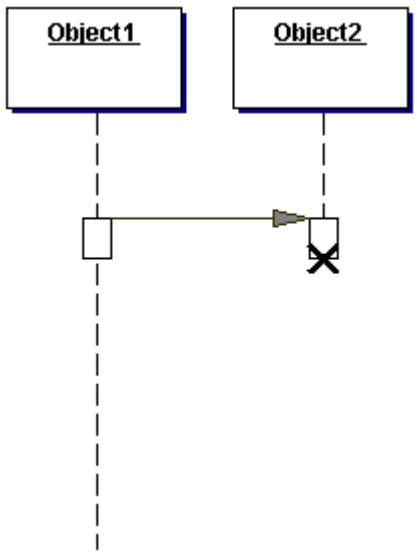
Element name	*Syntax*	*Synopsis*
Component		**Componente : modulo software eseguibile, dotato di identità e con un'interfaccia ben specificata, di cui di solito è possibile uno sviluppo indipendente.**
Dependency		**Dipendenza: relazione tra due elementi di modellazione, nella quale un cambiamento sull'elemento indipendente avrà impatto sull'elemento dipendente.**
Link		**Associazione (o relazione): relazione logica nella quale un partecipante (componente o nodo o classe) utilizza i servizi dell'altro partecipante.**

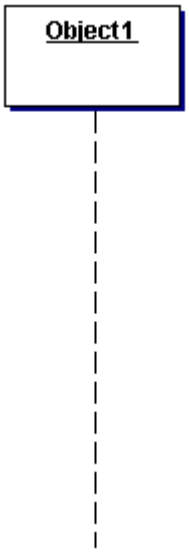
Node		**Nodo** : rappresentazione di una piattaforma hardware.

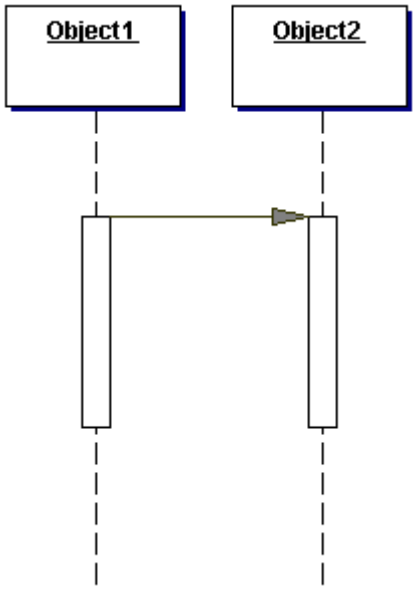
Component & Deployment Diagram Syntax Reference

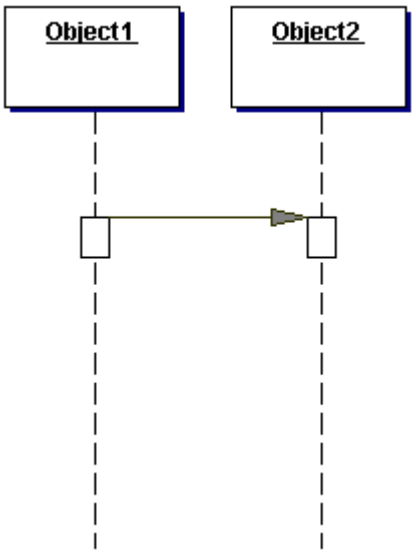
Element name	***Syntax***	***Synopsis***
Component		**Componente** : modulo software eseguibile, dotato di identità e con un'interfaccia ben specificata, di cui di solito è possibile uno sviluppo indipendente.
Dependency		**Dipendenza**: relazione tra due elementi di modellazione, nella quale un cambiamento sull'elemento indipendente avrà impatto sull'elemento dipendente.
Link		**Associazione** (o **relazione**): relazione logica nella quale un partecipante (componente o nodo o classe) utilizza i servizi dell'altro partecipante.
Node		**Nodo** : rappresentazione di una piattaforma hardware.

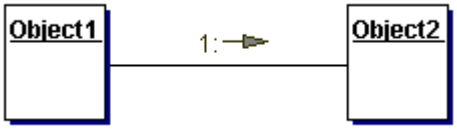
Interaction Diagram Syntax Reference

Element name	Syntax	Synopsis
Actor		**Attore**: ruolo interpretato dall'utente nei confronti del sistema. NB: un utente potrebbe non essere una persona fisica.
Object		**Oggetto**: creazione fisica (o istanza) di una classe
Creation & Destruction	 Creation  Destruction	**Creazione**: punto d'inizio del ciclo di vita di un oggetto (può coincidere con una chiamata al costruttore dell'oggetto creato). **Distruzione**: punto terminale del ciclo di vita di un oggetto (può coincidere con una chiamata al distruttore dell'oggetto creato).

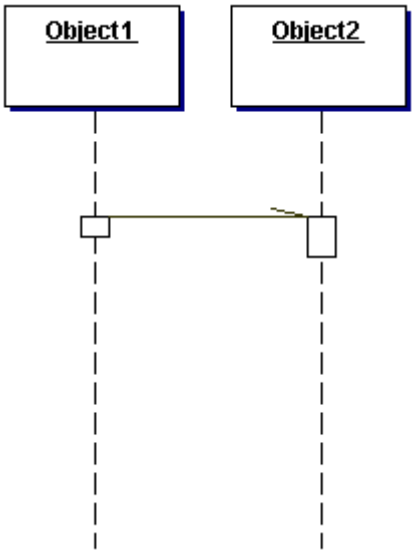
<table>
<tr>
<td>Life Line & Activity Line</td>
<td> 

Life Line</td>
<td> 

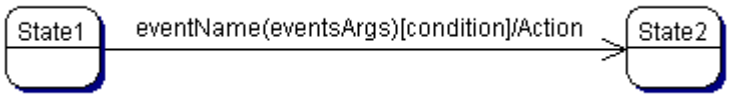
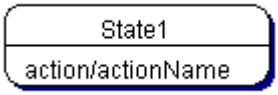
Activity Line</td>
<td>Linea di vita: linea di vita di un oggetto. Linea di attività: linea di attività di un oggetto (rappresenta il blocco d'esecuzione di un metodo).</td>
</tr>
<tr>
<td>Message</td>
<td> 

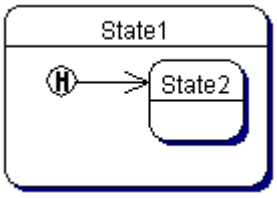
Sequence</td>
<td> 

Collaboration</td>
<td>Messaggio: messaggio di collaborazione tra oggetti (può coincidere con una chiamata al costruttore dell'oggetto di destinazione).</td>
</tr>
</table>

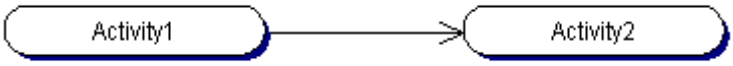
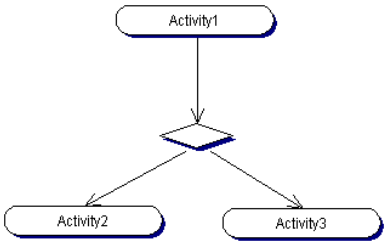
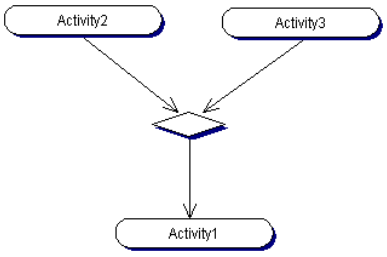
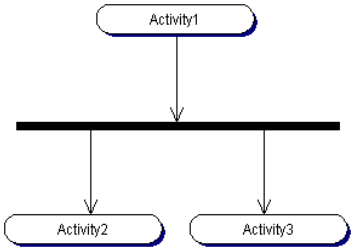
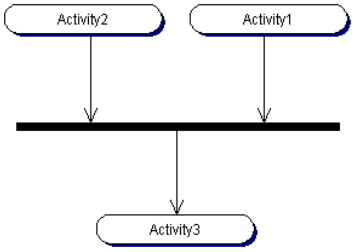
Asynchronous Message	 ***Asynchronuos Message***	**Messaggio asincrono: messaggio che può essere eseguito in maniera asincrona.**

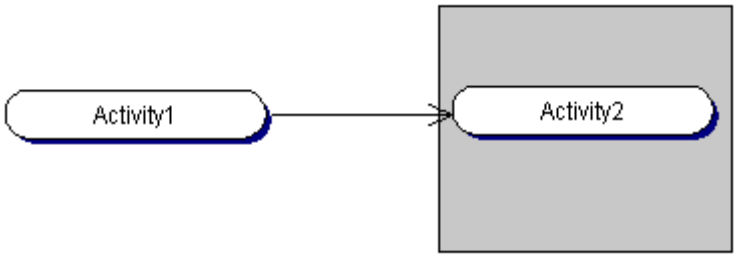
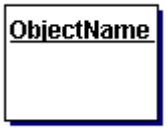
State Transition Diagram Syntax Reference

Element name	Syntax	Synopsis
State		**Stato: stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).**
Transition		**Transizione: attività che termina portando un oggetto in uno stato.**
Start & End	 ***Start***  ***End***	**Stato iniziale: punto iniziale di uno state transition diagram.** **Stato finale: punto terminale di uno state transition diagram.**
Action		**Azione: attività che caratterizza uno stato.**

History		**Stato con memoria: stato capace di ripristinare situazioni precedentemente attraversate.**

Activity Diagram Syntax Reference

Element name	***Syntax***	***Synopsis***
Activity		**Attività: processo assolto dal sistema.**
Flow		**Flusso (di attività): insieme di scenari legati da un obiettivo comune per l'utente.**
Branch & Merge	 ***Branch***  ***Merge***	**Ramificazione: rappresenta una scelta condizionale.** **Unione: rappresenta un punto di terminazione di blocchi condizionali.**
Fork & Join	 ***Fork***  ***Join***	**Biforcazione: rapresenta un punto di partenza per attività concorrenti.** **Riunione: rapresenta un punto di terminazione per attività concorrenti .**

SwimLane			**Corsia: aree di competenza di attività.**
Start & End	 ***Start***	 ***End***	**Stato iniziale: punto iniziale di uno state transition diagram. Stato finale: punto terminale di uno state transition diagram.**
Object & State	 ***Object***	 ***State***	**Oggetto: creazione fisica (o istanza) di una classe. Stato: Stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).**

General Purpose Elements Syntax Reference

Element name	Syntax		Synopsis
Package & Stereotype	 ***Package***	<<Stereotype>> ***Stereotype***	**Package: notazione che permette di ragruppare elementi UML. Stereotipo: meccanismo di estensione che permette di stereotipare costrutti non standard in UML.**
Constraint & Tagged value	{constraint} ***Constraint***	{key = value} ***Tagged Value***	**Vincolo: meccanismo di estensione che permette di esprimere vincoli. Valore etichettato: vincolo di proprietà.**
Iteration Mark & condition	* ***Iteration Mark***	[condition] ***Condition***	**Marcatore di iterazione: rappresenta un iterazione. Condition: rappresenta una condizione.**

Appendice H

Introduzione ai Design Patterns

L'applicazione dell'Object Orientation ai processi di sviluppo software complessi, comporta non poche difficoltà. Uno dei momenti maggiormente critici nel ciclo di sviluppo è quello in cui si passa dalla fase di analisi a quella di progettazione. In tali situazioni bisogna compiere scelte di design particolarmente delicate, dal momento che potrebbero pregiudicare il funzionamento e la data di rilascio del software. In tale contesto (ma non solo) si inserisce il concetto di design pattern, importato nell'ingegneria del software direttamente dall'architettura. Infatti la prima definizione di pattern fu data da Cristopher Alexander, un importante architetto austriaco (insegnante all'università di Berkeley - California) che iniziò a formalizzare tale concetto sin dagli anni '60. Nel suo libro "Pattern Language: Towns, Buildings, Construction" (Oxford University Press, 1977) Alexander definisce un pattern come una soluzione architetturale che può risolvere problemi in contesti eterogenei.
La formalizzazione del concetto di Design Pattern (pattern di progettazione) è ampiamente attribuita alla cosiddetta Gang of Four (brevemente GOF). La gang dei quattro è costituita da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides che, nonostante provenissero da tre continenti diversi, in quattro anni di confronti catalogarono la prima serie di 23 pattern che costituisce il nucleo fondamentale della tecnica. Nel 1994 vede la luce il libro considerato la guida di riferimento per la comunità dei pattern: "Design Patterns: elements of reusable object oriented software" (Addison-Wesley). Altri autori in seguito hanno pubblicato testi che estendono il numero dei pattern noti…

1.1. H.1 Definizione di Design Pattern

I Design Pattern rappresentano soluzioni di progettazione generiche applicabili a problemi ricorrenti all'interno di contesti eterogenei. Consapevoli che l'asserzione precedente potrebbe non risultare chiara al lettore che non ha familiarità con determinate situazioni, cercheremo di descrivere il concetto presentando, oltre che la teoria, anche alcuni esempi di pattern. In questo modo si potranno meglio apprezzare sia i concetti sia l'applicabilità.
L'idea di base, però, è piuttosto semplice. E' risaputo che la bontà della progettazione è direttamente proporzionale all'esperienza del progettista. Un progettista esperto risolve i problemi che si presentano utilizzando soluzioni che in passato hanno già dato buoni

risultati. La GOF non ha fatto altro che confrontare la (ampia) esperienza dei suoi membri nel trovare soluzioni progettuali, scoprendo così intersezioni abbastanza evidenti. Siccome queste intersezioni sono anche soluzioni che risolvono frequentemente problemi, in contesti eterogenei, possono essere dichiarate e formalizzate come Design Pattern. In questo modo si mettono a disposizione soluzioni a problemi comuni, anche ai progettisti che non hanno una esperienza ampia come quella della GOF. Quindi stiamo parlando di una vera e propria rivoluzione!

1.2. H.2 GOF Book: formalizzazione e classificazione

La GOF ha catalogato i pattern utilizzando un formalismo ben preciso. Ogni pattern, infatti, viene presentato tramite il nome, il problema a cui può essere applicato, la soluzione (non in un caso particolare) e le conseguenze. In particolare, ogni pattern viene descritto tramite l'elenco degli elementi a loro parere maggiormente caratterizzanti:

1. Nome e classificazione: importante per il vocabolario utilizzato nel progetto
2. Scopo: breve descrizione di cosa fa il pattern e suo fondamento logico
3. Nomi alternativi (se ve ne sono): molti pattern sono conosciuti con più nomi
4. Motivazione: descrizione di uno scenario che illustra un problema di progettazione e la soluzione offerta
5. Applicabilità: quando può essere applicato il pattern
6. Struttura (o modello): rappresentazione grafica delle classi del pattern mediante una notazione (in questa sede si utilizzerà UML, ma sul libro la cui nascita è antecedente alla nascita di UML vengono utilizzati OMT di Rumbaugh e i diagrammi di interazione di Booch)
7. Partecipanti: le classi/oggetti con le proprie responsabilità
8. Collaborazioni: come collaborano i partecipanti per poter assumersi le responsabilità
9. Conseguenze: pro e contro dell'applicazione del pattern
10. Implementazione: come si può implementare il pattern
11. Codice d'esempio: frammenti di codice che aiutano nella comprensione del pattern (in questa sede si utilizzerà Java, ma sul libro la cui nascita è antecedente alla nascita di Java, vengono utilizzati C++ e SmallTalk)
12. Pattern correlati: relazioni con altri pattern
13. Utilizzi noti: esempi di utilizzo reale del pattern in sistemi esistenti

I 23 pattern presentati nel libro della GOF sono classificati in tre categorie principali, basandosi sullo scopo del pattern:

1. **Pattern creazionali**: propongono soluzioni per creare oggetti

2. **Pattern strutturali**: propongono soluzioni per la composizione strutturale di classi e oggetti
3. **Pattern comportamentali**: propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti

Inoltre viene anche fatta una distinzione sulla base del raggio d'azione del pattern:

11. **Class pattern**: offrono soluzioni tramite classi e sottoclassi in relazioni statiche tra loro
12. **Object Pattern**: offrono soluzioni dinamiche basate sugli oggetti

Possiamo riassumere i 23 pattern del libro della GOF tramite questa tabella:

Scopo				
		Creazionale	**Strutturale**	**Comportamentale**
Raggio D'azione	**Class**	**Factory Method**	**Adapter (class)**	**Interpreter** **Template Method**
	Object	**Abstract Factory** **Builder** **Prototype** **Singleton**	**Adapter (object)** **Bridge** **Composite** **Decorator** **Facade** **Flyweight** **Proxy**	**Chain of responsability** **Command** **Iterator** **Mediator** **Memento** **Observer** **State** **Strategy**

Appendice I

Compilazione con Java 5

Supponiamo di avere a disposizione un Java Development Kit 1.5 o superiore.
Esistono due possibilità:

1. compilare un file che fa uso delle nuove feature di Tiger (per esempio il nuovo ciclo `foreach`)
2. compilare un file che non fa uso delle nuove feature di Tiger (per esempio utilizza una `Collection` senza sfruttare la parametrizzazione tramite generics)

Nel primo caso e la compilazione rimane sempre la stessa:

```
javac [opzioni] NomeFile.java
```

e non dovrebbe sorgere nessun problema.
Nel secondo caso, in cui ci troveremmo per esempio se volessimo compilare file scritte prima dell'avvento di Java 5, potremmo imbatterci in qualche warning. Addirittura potremmo anche andare incontro a qualche errore, per esempio utilizzando la parola `enum` come identificatore. Infatti, `enum` è ora una nuova parola chiave del linguaggio. Per esempio, non è raro trovare codice antecedente a Java 5 che dichiara i reference di tipo `Enumeration` proprio con l'identificatore `enum`.
Nel secondo caso, quindi, è opportuno specificare al momento della compilazione l'opzione:

```
-source 1.4
```

per la verità già incontrata nel Modulo 10 relativamente all'uso delle asserzioni. In questo modo avvertiremo il compilatore di compilare il file senza tener conto delle nuove caratteristiche di Java 5.
Il punto di forza di Java 5 è proprio il compilatore. Infatti, il bytecode prodotto sarà eseguibile anche da una Java Virtual Machine di versione precedente alla 5. Questo significa che, una volta compilato un file con Java 5, questo è eseguibile anche su JVM versione 1.4 o 1.3. Quindi le novità dal linguaggio sono gestite accuratamente dal compilatore. Per ottenere questo risultato è necessario specificare il flag "-source 1.4" (o

"1.3" etc…), e l'opzione "-target 1.4" (o "1.3" etc…). Infatti, l'opzione "-source" dichiara che il codice contenuto del file (o dei file) da compilare contiene una sintassi valida per la versione 1.4. Mentre l'opzione "-target" dichiara la versione di Java Virtual Machine su cui sarà possibile eseguire il bytecode compilato. Se non vengono specificate opzioni, il valore di default sia per "-source" che per "-target" sarà "1.5".

Quanto detto sopra può essere anche specificato con EJE. Basta lanciarlo con un JDK 1.5, o lanciarlo con un JDK 1.4.x, e specificare un JDK 1.5 dal menu File-Opzioni (la scorciatoia è F12). Dal tab "Java" potete, oltre che abilitare o meno le asserzioni, anche specificare la versione di Java da utilizzare (che di default dovrebbe rimanere 1.4, anche se avete lanciato EJE con Java 5). Questa operazione equivale a impostare il flag "-source" da riga di comando. Il combobox "target" permetterà di specificare il valore dell'opzione "-target".

Bibliografia

- Claudio De Sio Cesari: "Manuale di Java 6: programmazione orientata agli oggetti con Java Standard Edition 6" Hoepli (http://www.hoepli.it/editore/visbook.asp?id=88-203-3658-8&cat=vh)

- Patrick Naughton: "Il manuale Java 1.1" Mc.Graw-Hill

- Bruce Eckel: "Thinking in Java 2nd edition" http://www.bruceeckel.com

- AAVV: "The Java Tutorial" http://java.sun.com

- Calvin Austin, Monica Pawlan: "Wrinting Advanced Application for the Java Platform" http://java.sun.com

- AAVV: "Tha Java Language Specification" http://java.sun.com

- James Jaworsky: "Java 2 tutto e oltre" Apogeo

- Simon Roberts, Philip Heller, Michael Ernest: "Complete Java 2 certification study guide 5th edition"

- Kay Horstmann: "Java 2 i fondamenti" Mc.Graw-Hill, Sun Press

- Kay Horstmann: "Java 2 Tecniche avanzate" Mc.Graw-Hill, Sun Press

- Mark Wutka, et al: "Java Export Solutions" Macmillan Computer Publishing

- Martin Fowler, Kendall Scott: "UML Distilled" Addison-Wesley publishing

- Ivar Jacobson, Grady Booch, James Rumbaugh: "The Unified Software Development Process" Addison-Wesley publishing

- OMG: "UML version 1.3 specification" http://www.omg.org

- Ivar Jacobson, Grady Booch, James Rumbaugh: "The Unified Modelling Language Reference Manual" Addison-Wesley publishing

- Ivar Jacobson, Grady Booch, James Rumbaugh: "The Unified Modelling Language User Guide" Addison-Wesley publishing

- Simon Bennet, John Skelton, Ken Lunn: "Introduction to UML" McGraw-Hill (Schaum's)
- Gamma, Helm, Johnson, Vlissides: "Design Patterns" Addison-Wesley publishing

- Mark Grand: "Patterns in Java" Wiley & Sons

- Muler Pierre-Alain: "Instant UML" Addison-Wesley publishing

- Al Kelley-Ira Pohl: "C didattica a programmazione" Addison-Wesley publishing