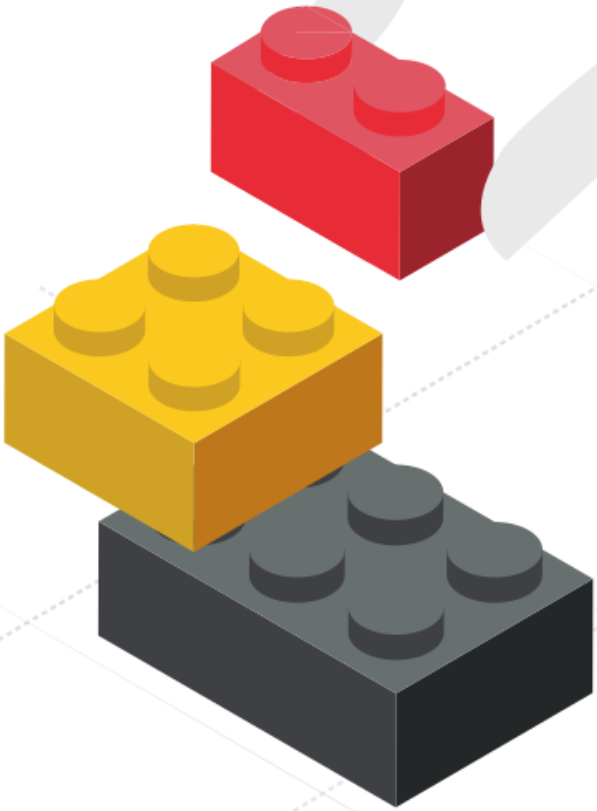
MASSIMILIANO TARQUINI

JAVA

MATTONE DOPO MATTONE



Un approccio moderno
alla programmazione

MATTONE
DOPO MATTONE

WWW.MATTONEDOPOMATTONE.IT

1. Premessa

Quando nel 2000 iniziai a scrivere la prima versione di 'Java Mattone dopo Mattone' non avrei mai immaginato che, di li a breve, sarebbe diventato uno dei libro più scaricati tanto che oggi, 2022, continuo a trovarne tracce sulle scrivanie di tanti programmatori. La prima versione fu rilasciata gratuitamente su internet diventando virale in pochissimi giorni.

Poi arrivò la pubblicazione con Hoepli. Era il 2002,il libro fu presentato allo SMAU lo stesso anno e fu un successo: in pochi giorni le copie disponibile presso lo stand della casa editrice andarono esaurite. Ancora ricordo l'emozione quando vidi il libro in vetrina alla libreria universitaria per la prima volta.

Immagine 1 Evoluzione della copertina nel tempo

Devo tantissimo a questo libro: professionalmente mi ha aperto tante porte ed offerto tante possibilità. Dopo due decenni, nonostante l'età, *Java Mattone dopo Mattone* continua a essere scaricato.

Nel frattempo Java è cambiato, si è evoluto, è diventato un linguaggio altamente espressivo, moderno. Rispetto alle prime versioni (ho iniziato ad usarlo per la prima volta nel 1996) sono stati introdotti migliaia di aggiornamenti in ottica di performance, stabilità e sicurezza. Ma non solo. Nel corso degli anni Java ha aperto le porte ad altri stili di programmazione facendo sue quanto di meglio paradigmi come il funzionale ed il dichiarativo hanno da offrire. Insieme ad una continua attenzione alle necessità dei programmatori, tutto questo ha contribuito a migliorare il linguaggio, e renderlo uno strumento che oggi conta una comunità di circa 30 milioni di programmatori.

Dopo 20 anni dalla prima uscita sento che è arrivato il momento di tornare a dedicare del tempo al libro. Ho quindi deciso di riscriverlo mantenendo però la struttura originale ed il metodo per l'apprendimento del linguaggio.

Dedico questa nuova versione del libro a tutti i programmatori: il libro affronterà gli aspetti del linguaggio partendo di concetti di base (per programmatori alle prime armi) fino ai concetti

avanzati per programmatori con esperienza. A tal fine, le sezioni del libro che riguardano concetti avanzati saranno chiaramente identificate: starà al lettore decidere su quali aspetti porre attenzione, e quali tralasciare per poi tornarci successivamente.

Il libro che mancava

Dopo tanti anni ricordo ancora la sensazione quando, alla ricerca del ... qualcosa di nuovo, mi scontravo con la difficoltà di reperire le informazioni specifiche senza aver ogni volta la sensazione di dover cercare un ago nel pagliaio. Non riesco a scordare il temo perso alla ricerca del libro che mi aiutasse a crescere sotto tutti i punti di vista (non basta essere esperti del linguaggio java per essere ottimi programmatori) con la possibilità magari di poter saltare intere sezioni stracolme di informazioni ormai note.

Quindi eccomi qui a scrivere il libro che ho sempre ricercato. A differenza della prima edizione, in questa nuova versione ho deciso di mettere in evidenza le evoluzioni del linguaggio in termini di capacità, sintassi, espressività partendo comunque dalle basi del linguaggio. In questo libro troverete quindi informazioni e tecniche per tutti i livelli di preparazione; i programmatori più giovani potranno imparare il linguaggio partendo dalle basi, e saltando a piedi pari le sezioni più complesse che richiedono una maggiore esperienza. I programmatori esperti potranno dedicarsi solo alle sezioni del libro da cui possono trarre informazioni relative a tecniche di programmazioni avanzate, informazioni sulle evoluzioni del linguaggio, idee, metodologie e spunti su strumenti per migliorare la qualità del codice prodotto.

Non avendo senso trattare tutte le versioni del linguaggio, e vista la distanza temporale che ci separa dalla prima versione di Java, ho deciso che il punto di partenza sarà quindi Java 8 (prima versione *Long Term Support* che prendo in considerazione): tutte le evoluzioni successive saranno chiaramente evidenziate facendo, di volta in volta, riferimento alla versione del linguaggio a partire dalla quale le innovazioni introdotte sono disponibili in forma non sperimentale. Tutte le nuove funzioni, attualmente in forma sperimentale o di *preview* non sono trattate da questo libro, ma saranno aggiunte in futuro qualora diventino effettive.

Ho deciso di dividere il libro in tre parti.

La **prima parte** riguarderà le caratteristiche del linguaggio ed è quindi dedicata a ciò che fa di Java ... il linguaggio Java. Ci concentreremo sul linguaggio, su aspetti specifici della modellazione ad oggetti ed infine sugli strumenti messi a disposizione da Java per modellare le applicazioni.

La **seconda parte** si concentrerà su aspetti specifici della programmazione. Analizzeremo le caratteristiche delle Java Core API e approfondiremo aspetti quali multithreading, programmazione concorrente, programmazione asincrona.

La **terza parte** del libro sarà quindi dedicata al codice sorgente: affronteremo aspetti della programmazione quali la qualità, la manutenibilità e la sicurezza del codice. Analizzeremo alcuni strumenti per effettuare la SAST del codice (analisi statica) e per una gestione efficiente, sicura e produttiva del codice Java.

Convenzioni

Le convenzioni in questo libro sono importanti perché non riguardano solamente lo stile di scrittura del codice o se una parola chiave va scritta maiuscoletto o corsivo oppure entrambi.

Utilizzando le convenzioni potrete decidere come leggere il libro, quali sezioni andare a toccare e quali sezioni saltare a piedi pari a seconda del vostro livello di preparazione.

Iniziamo dai simboli che utilizzerò per identificare: sezioni di codice a diversi livelli di difficoltà, note, convenzioni, una versione specifica del Java SDK.

Iniziamo da queste:

Ok lo ammetto ... Goku me lo vedevo e mi piaceva pure. Questa icona indica che la sezione è sicuramente dedicata ad un programmatore alle prime armi.

Al contrario questa indicherà che nella sezione sono riportate informazioni utili anche a programmatori esperti. In ogni caso, le sezioni contrassegnate con questa icona possono essere saltate per poi tornarci non appena si avrà maggiore dimestichezza con il linguaggio Java.

Come sono utilizzate nel libro?

Partiamo dal titolo dei capitoli. Ogni capitolo è identificato da un numero progressivo, un titolo significativo ed una serie di icone.

Il titolo sottostante si legge quindi: questo è il capitolo xx e contiene informazioni per programmatori con diversi livelli di preparazione.

xx Questo è un capitolo

Il lettore, dipendentemente dal suo livello di preparazione, sceglierà quindi quali sezioni leggere e quali rimandare per futuri approfondimenti quando l'esperienza lo consentirà, oppure quali saltare perché contenenti informazioni già note. Un programmatore esperto potrà invece dedicarsi solo alle sezioni del capitolo che contengono informazioni rilevanti per il suo livello di preparazione o che magari rappresentano una evoluzione del linguaggio ancora poco nota.

Per i paragrafi, la notazione è simile a quella di un capitolo tranne che per il numero mancante:

Questo è un paragrafo

Nota bene, per semplicità l'assenza della prima delle due icone identifica un paragrafo che tratta nozioni per neofiti. Quindi:

Questo è un paragrafo

è da considerarsi un paragrafo per tutti.

Ci sono poi le icone speciali. Poiché il libro affronta le basi del linguaggio java e le sua evoluzioni, le prossime icone indicano i vari dialetti del linguaggio e la versione della JVM a partire dalla quale un nuovo dialetto, una nuova sintassi oppure una nuova funzionalità sono diventate parte non sperimentale del linguaggio.

Ma non finisce qui. Ci sono poi alcuni *focus point* che richiamano l'attenzione del lettore su temi specifici e che, generalmente è consigliato leggere.

Questa sezione oserei chiamarla ... NOTA BENE! I nota bene sono accenti su alcune caratteristiche del linguaggio su cui è importante porre l'attenzione. Non ce ne sono molto ma, se ne trovate uno, consiglio di leggerlo con attenzione.

Questa è una nota. Le note sono utilizzate per evidenziare, spiegare, commentare sezioni del libro con informazioni che vale la pena mettere in evidenza. Contengono spesso curiosità interessanti.

Infine abbiamo la sezione dedicata a porre l'accento su convenzioni e best practices. Possono riguardare regole per la scrittura del codice, per l'assegnazione dei nomi o, più in generale suggerimenti per migliorare la qualità del codice prodotto e renderlo più leggibile.

A proposito ... le parole chiave saranno tutte, più o meno, minuscolo **bold** ed utilizzerò il *corsivo* per evidenziare esempi o porzioni di codice sorgente oppure per evidenziare nomi o parole speciali come nomi di variabili, nomi di funzioni o qualunque cosa che valga la pena fare emergere o differenziare all'interno del testo.

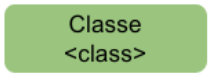
A differenza della prima versione del libro, ho cercato di introdurre maggiori formalismi. Pertanto la prossima è una definizione formale:

DEFINIZIONE: questa è una definizione

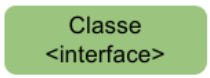
Relativamente alle convenzioni nelle immagini, utilizzeremo i seguenti simboli o notazioni:

Questa è una funzione oppure un metodo

Questa è una classe



Anche questa è una classe



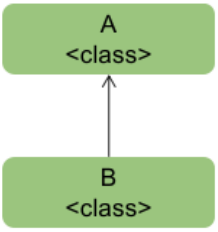
Questa è una interfaccia

Estende

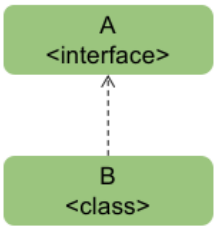
Implementa

Usa

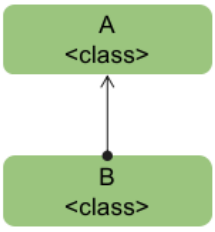
Diremo quindi che:



La classe B estende la classe A



La classe B implementa l'interfaccia A



La classe B usa la classe A

Da notare che i colori non hanno rilevanza, quindi possono variare a seconda dei casi.

Licenza D'uso

Questo è il libro aperto, strutturato per crescere nel tempo assorbendo nuove nozioni o funzionalità introdotte con le nuove versioni del linguaggio.

Questo libro è rilasciato con licenza Creative Commons:

"Attribuzione - Non commerciale 4.0 Internazionale (CC BY-NC 4.0)"

(https://creativecommons.org/licenses/by-nc/4.0/deed.it)

Pertanto sei libero di:

Condividere, riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato.

Modificare, remixare, trasformare il materiale e basarti su di esso per le tue opere.

Il licenziante non può revocare questi diritti fintanto che tu rispetti i termini della licenza, ovvero fino a che rispetterai le seguenti condizioni:

Attribuzione — Devi riconoscere una menzione di paternità adeguata, fornire un link alla licenza e indicare se sono state effettuate delle modifiche. Puoi fare ciò in qualsiasi maniera ragionevole possibile, ma non con modalità tali da suggerire che il licenziante avalli te o il tuo utilizzo del materiale.

Non Commerciale — Non puoi utilizzare il materiale per scopi commerciali.

Divieto di restrizioni aggiuntive — Non puoi applicare termini legali o misure tecnologiche che impongano ad altri soggetti dei vincoli giuridici su quanto la licenza consente loro di fare.

Non sei tenuto a rispettare i termini della licenza per quelle componenti del materiale che siano in pubblico dominio o nei casi in cui il tuo utilizzo sia consentito da una eccezione o limitazione prevista dalla legge.

Non sono fornite garanzie. La licenza può non conferirti tutte le autorizzazioni necessarie per l'utilizzo che ti prefiggi. Ad esempio, diritti di terzi come i diritti all'immagine, alla riservatezza e i diritti morali potrebbero restringere gli usi che ti prefiggi sul materiale.

Le fonti

Le fonti sono una cosa essenziale: nulla sarebbe stato possibile senza di esse.

Ovviamente, prima di tutto, 26 anni di esperienza da programmatore come professionista (senza contare gli anni passati a programmare il mio Commodore 64 in assembler o sperimentare c e Pascal sul mio caro vecchio Amiga 2000). Nella mia carriera ho lavorato con i principali linguaggi

di programmazione in ambiti disparati: dal militare, alle telecomunicazioni, ai sistemi enterprise. Ho iniziato a lavorare con Java nel lontano 1996. E' stato amore a prima vista.

Non posso non citare il vecchio Java Mattone dopo Mattone: una fonte vecchiotta, poco aggiornata, ma pur sempre uno strumento che negli anni ha dimostrato la validità del metodo didattico. Dalla vecchia versione ho ereditato l'impostazione del libro, intere sezioni che non sono mai invecchiate, la traccia principale.

Poi ovviamente, immancabile, la documentazione ufficiale della Oracle nelle varie versioni a partire da Java 8. Vale la pena citare,*Java Magazine* ricco di spunti di riflessione, tecniche di programmazione, nozioni base ed avanzate. Una fonte inesauribile di conoscenza.

Poi ci sono da ringraziare programmatori amici, conoscenti e non, da cui negli anni cui preso consigli, suggerimenti e frammenti di codice interessanti. Ricordatevi sempre che si impara da chiunque abbia qualcosa da dire... non scordatelo mai.

Vale la pena citare Baeldung (https://www.baeldung.com) fonte inesauribile ed attendibile. E' lo strumento che utilizzo da una vita quando si tratta di approfondire tematiche specifiche.

Infine, le citazioni:

1. ***Joshia Bloch*** *- You should favor composition over inheritance in Java. Here's why. 14 Luglio 2022 - https://blogs.oracle.com/javamagazine/post/java-inheritance-composition.*

2. ***Ben Evans*** *- Efficient JSON serialization with Jackson and Java. 02 Gennaio 2022 - https://blogs.oracle.com/javamagazine/post/java-json-serialization-jackson*

3. ***Christine H. Flood*** *- Understanding Garbage Collector: How the default garbage collectors work - 21 Novembre 2019 - https://blogs.oracle.com/javamagazine/post/understanding-garbage-collectors*

4. ***Thilina Ashen Gamage*** *- Understanding Java Memory Model - 22 Agosto 2018 - https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973*

Credits, ringraziamenti e scuse

Non posso non ringraziare Alessia Turturro per lo splendido lavoro fatto per definire la veste grafica del libro, ma sopratutto per l'illustrazione della copertina. Mi auguro possa aggiungere presto questo libro sul suo curriculum, tra i casi di successo.

Potete trovare i lavori si Alessia visitando la url https://dhero.it/ oppure il suo profilo linkedin https://www.linkedin.com/in/alessia-turturro/.

Un enorme grazie anche Catia Niccolai, compagna per la vita e collega nel lavoro, perché è stata lei più di altri a subire la lavorazione delle varie versioni del libro e per il lavoro di rilettura della bozza.

Infine Roberto Badessi: https://www.linkedin.com/in/robertobadessi/. Amico e collega di lavoro: lui ancora non lo sa, ma toccherà a lui fare la revisione tecnica della prima stesura del libro.

Infine voglio scusarmi preventivamente con tutti per gli esempi scelti in questo libro non sempre realistici dal punto di vista della problematica proposta. A mia discolpa voglio dire che ogni esempio è stato studiato per dimostrare o mostrare uno scenario rilevante solo al fine didattico e quindi non sempre aderente ad un problema reale.

2. La programmazione ad oggetti

Introduzione

Questo capitolo è dedicato al paradigma Object Oriented ed ha lo scopo di fornire ai neofiti della programmazione in Java i concetti di base necessari allo sviluppo di applicazioni orientate ad oggetti.

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

Il modello classico conosciuto come paradigma *Procedurale,* e successivamente quello *Funzionale*, possono essere riassunti in due parole: "Divide et Impera" ossia dividi e conquista. Difatti, in entrambe i casi, un problema complesso è suddiviso in problemi più semplici in modo che siano facilmente risolvibili mediante programmi procedurali o funzioni.

E' chiaro che in questo caso, l'attenzione del programmatore è accentrata al problema.

A differenza dei primi due, il paradigma Object Oriented accentra l'attenzione verso dati: l'applicazione è suddivisa in un insieme di oggetti in grado di interagire tra loro, e codificati in modo tale che la macchina sia in gradi di comprenderli.

Object Oriented non significa però solo un diverso approccio mentale alla modellazione delle applicazioni, ma una naturale evoluzione verso un modello di programmazione più moderno, che affronta ed offre una soluzione ai modelli predecessori.

Una naturale e necessaria evoluzione

Fu *Ada Lovelace* discepola e collaboratrice di *Charles Babbage*[1] a definire le specifiche del primo linguaggio di programmazione di tipo assemblativo nel 1837, e anche se questo non è rilevante per questo libro, ho voluto fare giusto omaggio a tutti i visionari che hanno saputo vedere avanti nel futuro.

[1] https://it.wikipedia.org/wiki/Charles_Babbage - Scienziato proto-informatico che per primo ebbe l'idea di un calcolatore programmabile.

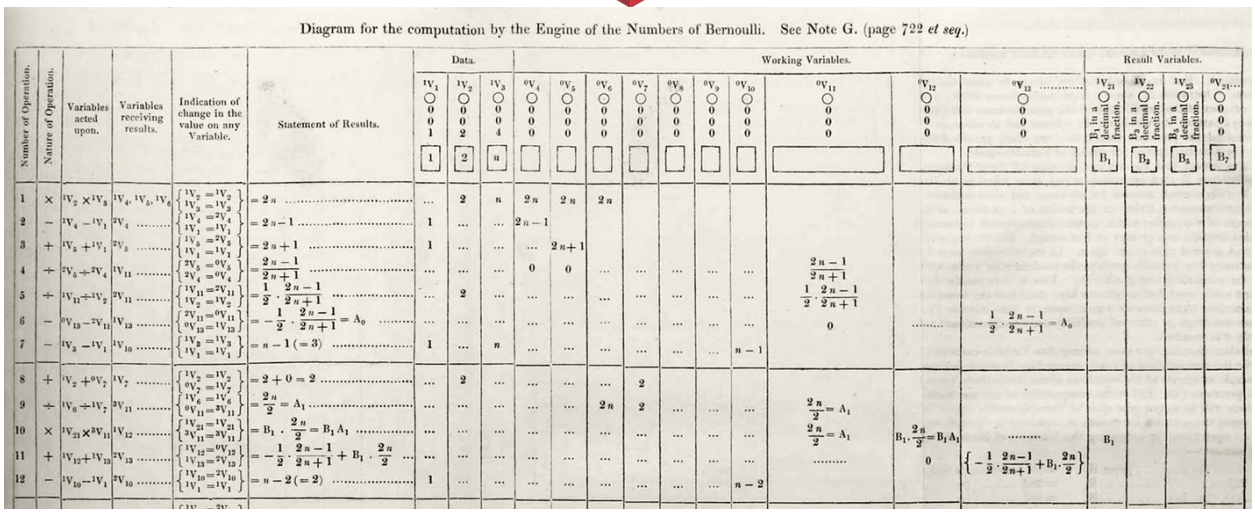


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Immagine 2 Diagramma di computazione dei numeri di Bernoulli

Poi fu il *Plankalkul (Konrad Zuse 1943-45)*, ma questa è un altra storia!

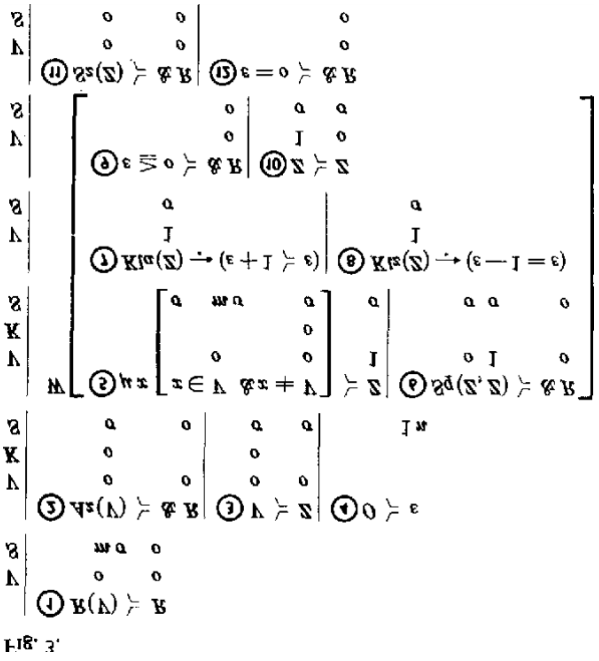


Immagine 3 Plankalkul - un precursore dei moderni linguaggi di programmazione

Dicevamo ... All'inizio ci fu il *Linguaggio Macchina*. All'epoca del linguaggio macchina le istruzioni corrispondevano con l'insieme delle istruzioni semplici eseguibili dal hardware, cosa che richiedeva un enorme fatica per codificare anche algoritmi molto semplici. I programmi scritti in linguaggio macchina risultavano illeggibili dall'uomo, ogni CPU ha il suo linguaggio macchina, e di conseguenza non erano trasportabili.

Fu quindi deciso che valeva fare uno sforzo per semplificare la vita al programmatore introducendo un seppur minimo livello di astrazione: nel *Linguaggio Assembler* ogni istruzione è identificata da una sigla, le variabili sono rappresentate da nomi piuttosto che da numeri.

Quando i programmi erano scritti in assembler (linguaggio molto simile al *Linguaggio Macchina*), ogni dato era globale, e le funzioni andavano disegnate a basso livello: ancora una volta l'attenzione del programmatore era orientata alla macchina anziché sul problema.

```
1  section .text
2      global _start       ;must be declared for using
           gcc
3  _start:                     ;tell linker entry point
4      mov edx, len    ;message length
5      mov ecx, msg    ;message to write
6      mov ebx, 1      ;file descriptor (stdout)
7      mov eax, 4      ;system call number (sys_write)
8      int 0x80        ;call kernel
9      mov eax, 1      ;system call number (sys_exit)
10     int 0x80        ;call kernel
11
12 section .data
13
14 msg db  'Hello, world!',0xa ;our dear string
15 len equ $ - msg         ;length of our dear string
```

Immagine 4 esempio linguaggio assembler

Il passo successivo fu quindi quello di rendere la codifica degli algoritmi il più possibile vicina al problema da risolvere anziché all'architettura della macchina ed avere linguaggi scritti comprensibili da un largo numero di programmatori. Tra gli anni 50 e 60 si passò ai linguaggi ad alto livello fino ad arrivare agli anni 70, anni che videro la nascita del linguaggio *C* (*Dennis Ritchie 1972 Bell Laboratories*).

Con l'avvento dei linguaggi procedurali come il linguaggio *C*, i programmi sono diventati più robusti e semplici da mantenere perché il linguaggio forniva regole sintattiche e semantiche che, supportate da un compilatore, consentivano un maggior livello di astrazione rispetto a quello fornito dall'assembler, un ottimo supporto alla scomposizione procedurale dell'applicazione, e come nel caso del *C* altissime prestazioni.

Programmazione Procedurale

Secondo il *paradigma procedurale*, il programmatore analizza il problema ponendosi dal punto di vista del computer che esegue solamente istruzioni semplici e di conseguenza adotta un approccio di tipo *divide et impera*[2]: il programmatore sa perfettamente che un'applicazione per quanto complessa può essere suddivisa in procedure e funzioni di piccola entità.

Quest'approccio è stato formalizzato in molti modi ed è ben supportato da un gran numero di linguaggi che forniscono al programmatore un ambiente in cui siano facilmente definibili procedure e funzioni.

---

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

Le procedure e le funzioni sono blocchi di codice riutilizzabile che possiedono un proprio insieme di dati, e realizzano specifiche funzionalità. Le procedure così definite possono essere richiamate ripetutamente in un programma, possono ricevere parametri che modificano il loro stato, possono tornare valori al codice chiamante.

Procedure e funzioni possono essere legate assieme a formare un'applicazione, ed è quindi necessario che i dati, all'interno di una applicazione, siano condivisi tra loro. Nella programmazione procedurale, questo meccanismo si risolve mediante l'uso di variabili globali, passaggio di parametri e ritorno di valori.

Nella prossima immagine è schematizzata una tipica applicazione procedurale ed il suo diagramma di flusso.

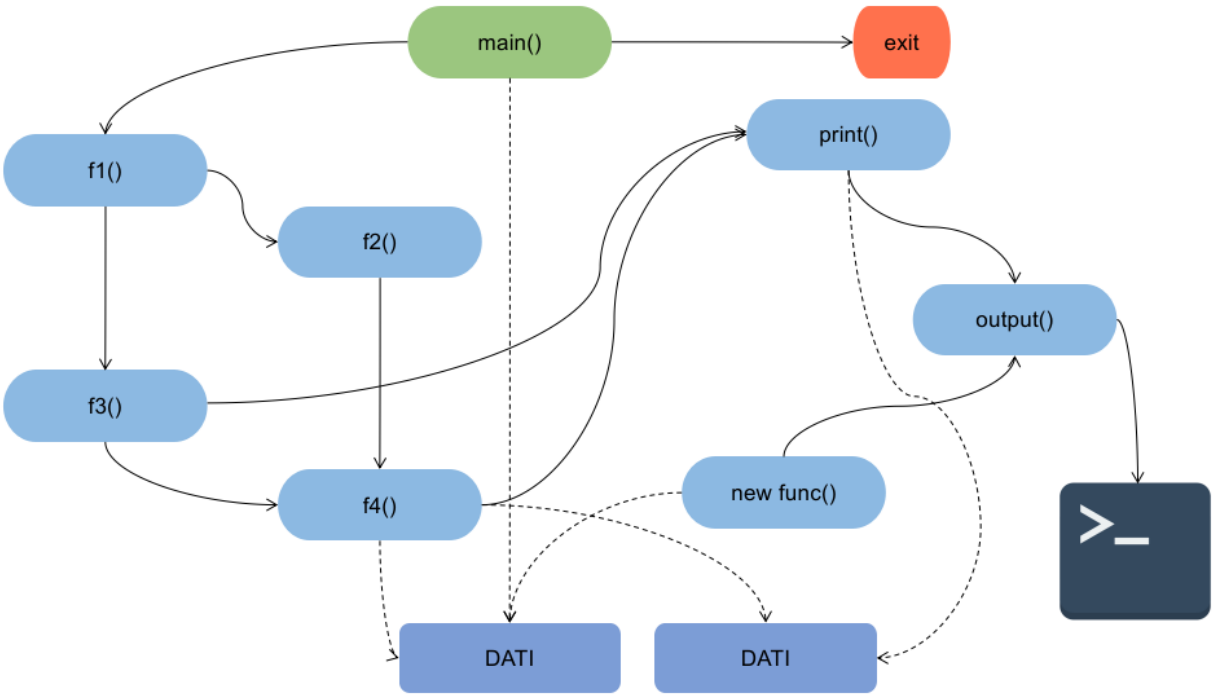


Immagine 5 diagramma di flusso di una applicazione procedurale

L'uso di variabili globali genera però problemi di protezione dei dati quando le procedure si richiamano tra loro. Per esempio, nell'applicazione mostrata nella figura precedente, la procedura *Output che* esegue una chiamata a basso livello verso il terminale dovrebbe essere chiamata soltanto dalla procedura *Print* la quale a sua volta modifica dati globali. A causa del fatto che le procedure non sono *auto-documentanti* (*self-documenting*), ossia non rappresentano entità ben definite, un programmatore dovendo modificare l'applicazione, e non conoscendone a fondo il codice, potrebbe creare una funzione *new func()* ed utilizzare la routine *Output* senza chiamare la procedura *Print* dimenticando quindi l'aggiornamento dei dati globali a carico di *Print* e producendo di conseguenza effetti indesiderati (*side-effects*) difficilmente gestibili.

Per questo motivo, le applicazioni basate sul modello procedurale erano difficili da aggiornare e controllare con meccanismi di *debug*. Gli errori derivanti da effetti indesiderati potevano presentarsi in qualunque punto del codice causando la propagazione incontrollata dell'errore.

Ad esempio, riprendendo ancora la nostra applicazione, una gestione errata dei dati globali dovuta ad una mancata chiamata a *Print* potrebbe avere effetto su *f4()* che a sua volta propagherebbe l'errore ad *f2()* ed *f3()* fino al metodo *main* del programma causando la terminazione anomala del processo.

Correggere gli errori procedurali

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

Il linguaggio *C*, ad esempio, fornisce strumenti mirati alla circoscrizione del problema come il modificatore **static** con il fine di delimitare sezioni di codice all'interno dell'applicazione in grado di accedere a dati globali, eseguire funzioni di basso livello, o evitare direttamente l'uso di variabili globali: se applicato ad una variabile locale il calcolatore alloca memoria permanente in modo molto simile a quanto avviene per le variabili globali. Questo meccanismo consente alla variabile dichiarata **static** di mantenere il proprio valore tra due chiamate successive ad una funzione. A differenza di una variabile locale non statica, il cui ciclo di vita (di conseguenza il valore) è limitato al tempo necessario per l'esecuzione della funzione, il valore di una variabile dichiarata **static** non andrà perduto tra chiamate successive.

La differenza sostanziale tra una variabile globale ed una variabile locale **static** è che la seconda è nota solamente al blocco in cui è dichiarata ossia è una variabile globale con scopo limitato, viene inizializzata solo una volta all'avvio del programma, e non ogni volta che si effettui una chiamata alla funzione in cui sono definite.

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

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

}
```

Usando una variabile **static**, la funzione è in grado di mantenere il valore della variabile tra chiamate successive evitando l'uso di variabili globali. Il modificatore **static** può essere utilizzato anche con variabili globali: se applicato ad un dato globale indica al compilatore che la variabile creata dovrà essere nota solamente alle funzioni dichiarate nello stesso file contenente la dichiarazione della variabile. Stesso risultato lo otterremmo applicando il modificatore ad una funzione o procedura.

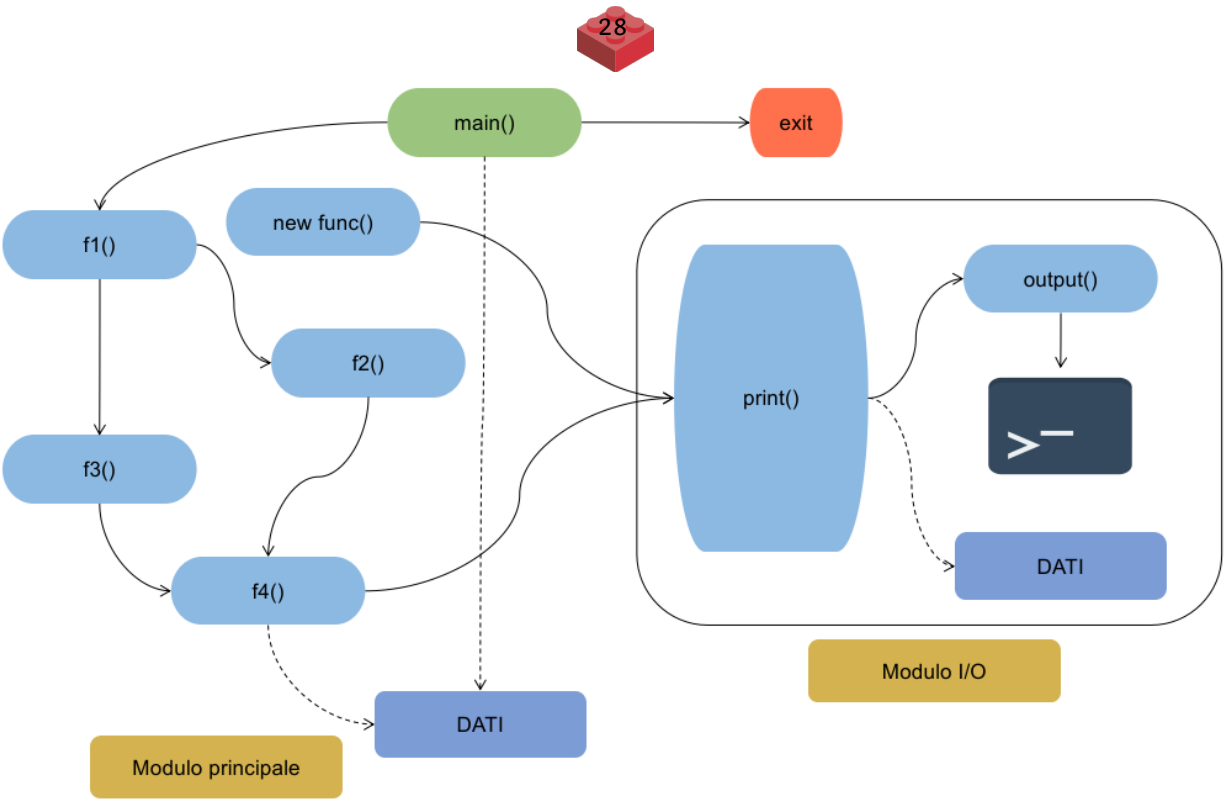


Immagine 6 Diagramma di una applicazione procedurale suddiviso per moduli

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

Un esempio tipico di applicazione suddivisa in moduli è schematizzato nella *Immagine 6 Diagramma di una applicazione procedurale suddiviso per moduli* nella quale è rappresentata una nuova versione del modello precedentemente proposto. Il modulo di I/O mette a disposizione degli altri moduli la funzione *Print* incapsulando la routine *Output* ed i dati sensibili.

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

Paradigma funzionale

Per rendere il codice più modulare e ridurre la possibilità di produrre effetti secondari dovuti alla gestione dei dati globali, il passo successivo fu quindi quello di iniziare a ragionare solo in termini di funzioni ed in particolar modo di quelle che sono chiamate *funzioni pure*, ovvero funzioni che non producono effetti secondari, e che tornano sempre lo stesso risultato a partire dal medesimo set di parametri di input.

Nonostante risolvesse la maggior parte dei problemi derivanti da un approccio procedurale, la programmazione funzionale ha comunque una serie di controindicazioni. Primo tra tutti è quello relativo alla performance ed alla duplicazione del codice: a causa della immutabilità delle variabili (le funzioni pure non modificano mai i valori dei parametri di input) è necessario

duplicare parti di codice e strutture dati anche solo per apportare piccoli cambiamenti al programma.

Inoltre, poiché ragioniamo solo in termini di funzioni pure, la ricorsione diventa uno strumento essenziale. Il risultato finale è un codice poco leggibile e assolutamente poco intuitivo.

In realtà l'approccio funzionale alla programmazione ha caratteristiche così interessanti da convincere la comunità, e quindi la Oracle, ad aggiungere in java il supporto per le funzioni a partire da Java 8. La programmazione funzionale e le sue caratteristiche saranno pertanto approfondite in una specifica sezione del libro.

Paradigma ad oggetti

Il paradigma Object Oriented rappresenta un salto avanti generazionale in quanto non solo formalizza la tecnica vista in precedenza di incapsulare e raggruppare parti di un programma ma divide le applicazioni in gruppi logici che rappresentano concetti sia a livello di utente sia applicativo Nella programmazione ad oggetti, il programmatore inizia con l'analizzare tutti i singoli aspetti concettuali che compongono un programma. Questi concetti sono chiamati oggetti ed hanno nomi legati a ciò che rappresentano: una volta che gli oggetti sono identificati, il programmatore decide di quali attributi (dati) e funzionalità (metodi) dotare le entità.

Il programmatore, in fase di analisi, dovrà includere le regole di interazione tra gli oggetti. Proprio grazie a queste interazioni sarà possibile riunire gli oggetti a formare un'applicazione.

A differenza di procedure e funzioni, gli oggetti sono *auto-documentanti* (*self-documenting*). Un'applicazione può essere scritta avendo solo poche informazioni ed il funzionamento interno di ogni oggetto è completamente nascosto al programmatore (Incapsulamento Object Oriented).

Classi di oggetti

Concentriamoci per qualche istante su alcuni concetti tralasciando l'aspetto tecnico del paradigma *Object Oriented,* e proviamo per un istante a pensare all'oggetto che state utilizzando: un libro.

Pensando ad un libro è naturale richiamare alla mente un insieme di oggetti aventi caratteristiche in comune: tutti i libri contengono delle pagine, ogni pagina contiene del testo e le note sono scritte sul fondo. Altra cosa che ci viene subito in mente riguarda le azioni che tipicamente compiamo quando utilizziamo un libro: voltare pagina, leggere il testo, guardare le figure etc.

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

Pensiamo ora ad un libro scritto in inglese. Ovviamente sarà comprensibile soltanto a persone in grado di comprendere questa lingua; d'altro canto possiamo comunque sfogliarne i contenuti (anche se privi di senso), guardarne le illustrazioni o scriverci dentro per prendere appunti.

Questo insieme generico di proprietà rende un libro utilizzabile da chiunque a prescindere dalle caratteristiche specifiche (nel nostro caso la lingua). Possiamo quindi affermare che un libro è, genericamente, un oggetto che contiene pagine e contenuti da guardare.

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

Ereditarietà

Con la definizione di una classe di oggetti, nel paragrafo precedente abbiamo stabilito che, in generale, un libro contiene pagine che possono essere sfogliate, sottolineate etc. Quindi, qualunque oggetto che corrisponda a questa definizione potrebbe essere a sua volta un libro ovvero, la definizione data di libro rappresenta una generalizzazione del concetto.

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

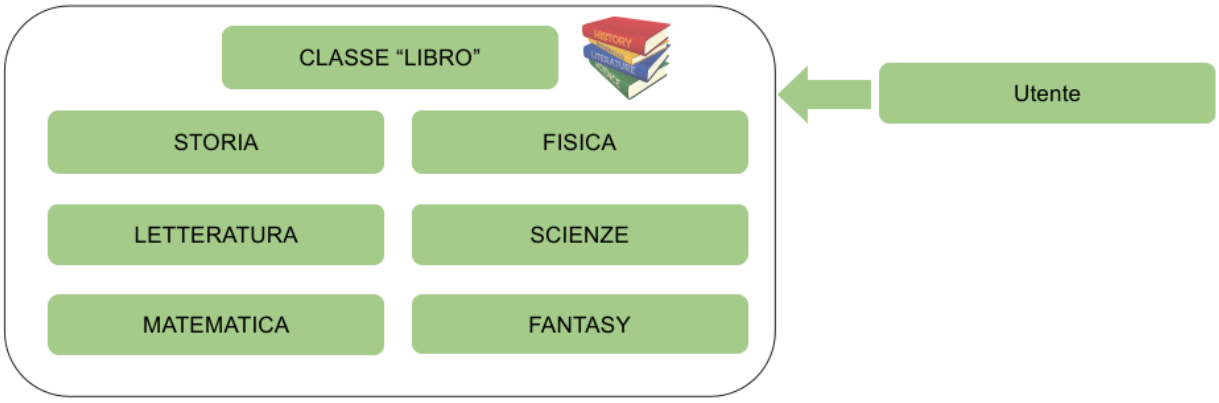


Immagine 7 La classe generica "Libro" e le sue specializzazioni

Il concetto di ereditarietà e polimorfismo nella programmazione ad oggetti

Estendendo i concetti illustrati alla programmazione iniziamo ad intravederne i reali vantaggi. Una volta stabilite le categorie di base, possiamo utilizzarle per creare tipi specifici di oggetti semplicemente ereditando e specializzando le regole di base (ovvero generiche).

Per definire questo tipo di relazioni, è utilizzata una forma a diagramma in cui la classe generica è riportata come nodo sorgente di un grafo orientato, i sotto nodi rappresentano categorie più specifiche, gli archi che uniscono i nodi sono orientati da specifico a generale (*Immagine 8 Diagramma di ereditarietà*).

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

DEFINIZIONE: L'ereditarietà è una relazione di generalizzazione/specializzazione: la superclasse definisce un concetto generale e la sottoclasse rappresenta una variante specifica di tale concetto generale.

Su questa interpretazione si basa tutta la teoria dell'ereditarietà nei linguaggi a oggetti. Oltre a essere un importante strumento di modellazione (e quindi significativo anche in contesti diversi dalla programmazione in senso stretto, per esempio in UML). L'ereditarietà, come vedremo, ha importantissime ripercussioni anche sulla riusabilità del software.

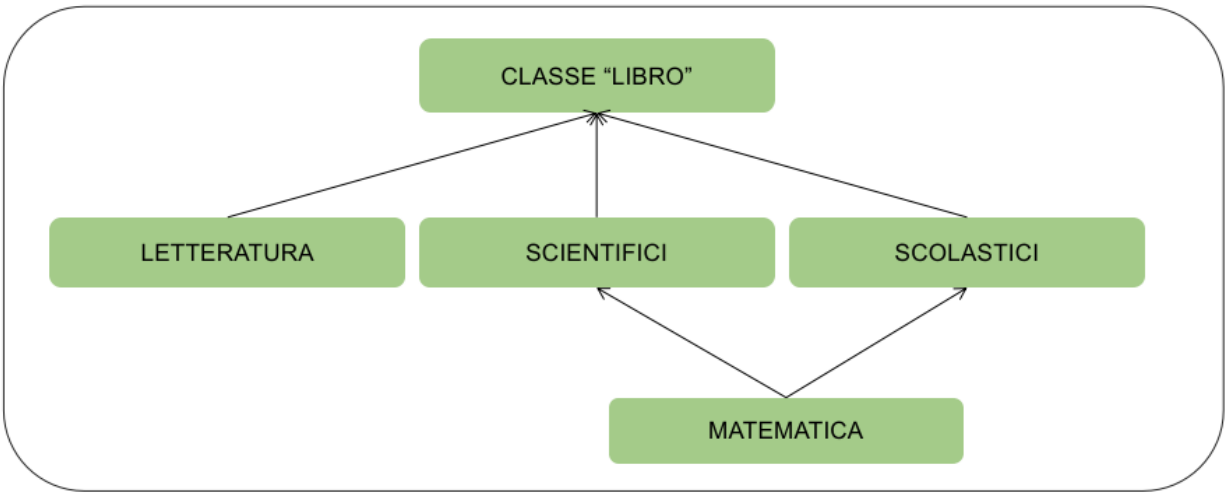


Immagine 8 Diagramma di ereditarietà

Come conseguenza dell'ereditarietà esiste un altra ed altrettanto utile proprietà nella programmazione ad oggetti: il *polimorfismo*.

Per polimorfismo si intende, letteralmente, la capacità di un oggetto di assumere più forme che può anche essere definito come l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.

Per provare a fare chiarezza sul concetto (cosa che faremo abbondantemente nei capitoli successivi) si pensi al diverso funzionamento di una macchina, un aereo ed un treno quando eseguono l'operazione di spostarsi. Macchina, aereo e treno possono essere sicuramente generalizzati nella classe dei mezzi di trasporto eppure la macchina si muoverà in maniera diversa rispetto al treno ed all'aereo.

Vantaggi nell'uso dell'ereditarietà

Come è facile intravedere, il vantaggio principale dell'ereditarietà è la capacità di fornire codice riutilizzabile. Creare una classe di oggetti per definire entità è molto di più che crearne una semplice rappresentazione: per la maggior parte delle classi, l'implementazione è spesso inclusa all'interno della descrizione.

Come conseguenza abbiamo un altro importantissimo vantaggio. L'organizzazione degli oggetti fornita dal meccanismo di ereditarietà rende semplici le operazioni di manutenzione di

un'applicazione: ogni volta che si renda necessaria una modifica, è in genere sufficiente crearne un nuovo all'interno di una classe di oggetti ed utilizzarlo per rimpiazzarne uno vecchio ed obsoleto. La modifica si propagherà (a meno di accezioni che vedremo successivamente) a tutti le classi lungo la catena di ereditarietà.

In Java ad esempio ogni volta che definiamo un concetto, esso è definito come una classe all'interno della quale è scritto il codice necessario ad implementare le funzionalità dell'oggetto per quanto generico esso sia. Se un nuovo oggetto è creato per mezzo del meccanismo della ereditarietà da un oggetto esistente, si dice che la nuova entità *deriva* dalla originale. Quando questo accade, tutte le caratteristiche dell'oggetto principale diventano parte della nuova classe. Poiché l'oggetto derivato eredita le funzionalità del suo predecessore, l'ammontare del codice da implementare è notevolmente ridotto. Il codice dell'oggetto d'origine è stato riutilizzato.

A questo punto è necessario iniziare a definire formalmente alcuni termini. La relazione di ereditarietà tra oggetti è espressa in termini di *superclasse* e *sottoclasse*.

DEFINIZIONE: Una superclasse è una classe più o meno generica utilizzata come punto di partenza per derivare nuove classi.

DEFINIZIONE: Una sottoclasse rappresenta la specializzazione di una superclasse.

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

Parleremo in realtà di *catena di ereditarietà*. Pertanto, una super-classe può essere a sua volta sotto-classe di una classe base, oppure una sotto-classe può essere classe base per nuove specializzazioni.

Programmazione Object Oriented ed incapsulamento

Come già ampiamente discusso, nella programmazione ad oggetti definiamo classi creando rappresentazioni di entità o nozioni da utilizzare come parte di un'applicazione. Per assicurarci che il programma lavori correttamente, ogni classe deve rappresentare in modo corretto il concetto di cui è modello senza che l'utente possa disgregarne l'integrità. Per fare questo è importante che ogni classe esponga solo la porzione di codice e dati che le componenti del programma devono utilizzare.

Ogni altro dato e codice devono essere nascosti affinché sia possibile mantenere l'oggetto in uno stato consistente.

Ad esempio, se un oggetto rappresenta uno stack[3] di dati (*Immagine 9 Stack di dati*), l'applicazione dovrà poter accedere solo al primo dato dello stack od inserirne uno nuovo sulla cima, ossia alle funzioni di *Push* e *Pop*.

---

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

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

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

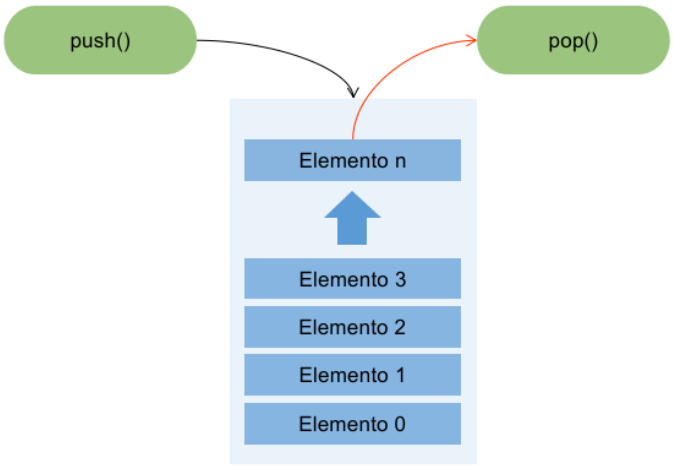


Immagine 9 Stack di dati

In generale quindi, quando parliamo di incapsulamento nella programmazione ad oggetti facciamo riferimento alla capacità di incapsulare stato e comportamento in un unica unità: la classe. Grazie all'incapsulamento un oggetto avrà un proprio stato privato a cui altri oggetti non possono accedere se non attraverso metodi specializzati pubblici (analogo di funzioni per i casi precedenti).

I vantaggi dell'incapsulamento

Una volta che un oggetto è stato incapsulato e testato, tutto il codice ed i dati associati sono protetti. Modifiche successive al programma non potranno causare rotture nelle dipendenze tra gli oggetti poiché non saranno in grado di vedere i legami tra dati ed entità. L'effetto principale sull'applicazione sarà quindi quello di localizzare gli errori evitandone la propagazione e dotando l'applicazione di grande stabilità; per sua natura, un oggetto incapsulato non produce effetti secondari.

Come abbiamo già visto, in un programma scomposto per blocchi le procedure e le funzioni tendono ad essere interdipendenti: ogni variazione al programma richiede spesso la modifica di funzioni condivise, cosa che può propagare un errore ad altri membri del programma che le utilizzano. In un programma *Object Oriented*, le dipendenze sono sempre strettamente sotto controllo e sono mascherate all'interno delle entità concettuali. Modifiche a programmi di questo tipo riguardano tipicamente l'aggiunta di nuovi oggetti, ed il meccanismo di ereditarietà ha l'effetto di preservare l'integrità dei membri dell'applicazione.

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

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

Un ultima precisazione: classi vs oggetti

Negli anni ho notato che si tende a fare una grave confusione tra classi ed oggetti. Da qui in poi, quando parleremo di classi ed oggetti faremo riferimento rispettivamente a:

DEFINIZIONE: Classe - un modello astratto, generico, per una famiglia di oggetti con caratteristiche comuni che definisce implicitamente un tipo di dato;

DEFINIZIONE: Oggetto - una istanza di una classe.

In definitiva, una classe esiste come entità unica, ma da una classe posso creare molti oggetti ognuno con un proprio stato.

Parlando di classi, faremo spesso riferimento ad una *classe* come ad un *tipo*.

Alcune buone regole per creare oggetti

1. *Un oggetto deve rappresentare un singolo concetto ben definito.*

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

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

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

3. Fare attenzione nell'utilizzo della ereditarietà.

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

E' tutto oro quello che luccica?

Quando parliamo di programmazione ad oggetti non dobbiamo fare riferimento ad uno specifico linguaggio di programmazione; programmare ad oggetti è più un approccio alla strutturazione della applicazione seguendo le regole descritte nei paragrafi precedenti. Anche una applicazione scritta in linguaggio C può essere una applicazione ad oggetti.

Nel prossimo esempio vediamo come sia possibile creare una classe in C utilizzando le strutture:

```
// Definizione di fake-class in C
    // MiaClasse.h

#ifndef _MIACLASSE_H
    #define _MIACLASSE_H

typedef struct _MiaClasse
    {
     int m_id;
     char * m_Name;
     short m_Param;
    } MiaClasse;

// Funzioni membro: attenzione in C non esiste il puntatore implicito all'istanza corrente (il
    "this" per intenderci), quindi ci occuperemo di passarlo manualmente

void MiaClasse_Init(MiaClasse *pThis, int id, char* name, short param); // Il costruttore
    void MiaClasse_Destroy(MiaClasse *pThis); // Il distruttore
    const char * MiaClasse_GetName();
    void MiaClasse_SetName(); // Metodi di incapsulamento

#endif



// MiaClasse.c

#include "MiaClasse.h"

void MiaClasse_Init(MiaClasse *pThis, int id, char* name, short param){
     if (pThis == NULL)  return;
     pThis->m_id = id;
    pThis->m_Name = name;
    pThis->m_Param = param;

}

// .... ecc....
```

Quindi, perché java? Perché Java, a differenza di C, è un linguaggio fortemente orientato agli oggetti e fornisce supporto nativo alla creazione di classi di oggetti. Sarà comunque sempre responsabilità del programmatore porre attenzione alla qualità del codice prodotto. Un programmatore inesperto potrebbe comunque creare codici poco intuitivi, e quindi poco mantenibili anche utilizzando un linguaggio come Java.

Storia breve del paradigma Object Oriented

L'idea di modellare applicazioni con oggetti piuttosto che funzioni, comparve per la prima volta nel 1967. Il termine *programmazione orientata ad oggetti* fu coniato da *Alan Kay* alla scuola di specializzazione nel 1966. La sua grande intuizione fu quella di pensare al software fatto di entità in grado di comunicare tramite passaggio di messaggi piuttosto che condividendo dati globali, e smettere di suddividere i programmi in *strutture dati* e *procedure* separate.

In altre parole, gli ingredienti essenziali per realizzare applicazioni *Object Oriented* erano:

1. Passaggio di messaggi;

2. Incapsulamento;

3. Binding dinamico (che vedremo successivamente).

Da notare che ereditarietà ed incapsulamento non erano considerati ingredienti essenziali.

Con il rilascio del linguaggio Simula, prodotto in Norvegia da *Ole-Johan Dahl* e *Kristen Nygaard*. Simula, linguaggio largamente utilizzato per sistemi di simulazione, adottava i concetti di classe ed ereditarietà.

Solo qualche anno dopo, lo stesso *Alan Kay* prese parte al progetto Xerox che lanciò sul mercato il linguaggio Smalltalk sviluppato nei laboratori di Palo Alto.

Se l'idea iniziale di introdurre un nuovo paradigma di programmazione non aveva riscosso grossi successi all'interno della comunità di programmatori, l'introduzione del termine *Object Oriented* stimolò la fantasia degli analisti, e negli anni immediatamente a seguire videro la luce un gran numero di linguaggi di programmazione ibridi come C++ e Objective-C, oppure ad oggetti puri come Eiffel.

Il decennio compreso tra il 1970 ed il 1980 fu decisivo per la metodologia *Object Oriented*. L'introduzione di calcolatori sempre più potenti e la successiva adozione di interfacce grafiche (GUI) prima da parte della Xerox e successivamente della Apple, spinse i programmatori ad utilizzare sempre di più il nuovo approccio in grado adattarsi con più semplicità alla complessità dei nuovi sistemi informativi.

Fu durante gli anni 70' che la metodologia *Object Oriented* guadagnò anche l'attenzione della comunità dei ricercatori poiché il concetto di ereditarietà sembrava potesse fornire un ottimo supporto alla ricerca sull'intelligenza artificiale.

Fu proprio questo incredibile aumento di interesse nei confronti della programmazione ad oggetti che, durante gli anni successivi, diede vita ad una moltitudine di linguaggi di programmazione e di metodologie di analisi spesso contrastanti tra loro.

Di fatto, gli anni 80' possono essere riassunti in una frase di *Tim Rentsch* del 1982:

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

"…la programmazione Object Oriented sarà negli anni 80 quello che fu nel 1970 la programmazione strutturata. Tutti ne erano a favore. Tutti i produttori di software promuovevano le loro soluzioni ed i loro prodotti a supporto. Ogni manager, a parole, ne era favorevole. Ogni programmatore la praticava (ognuno in modo differente). Nessuno sapeva di cosa si trattasse esattamente." [Rentsch82]

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

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

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

3. Il linguaggio Java

Introduzione

Java è un linguaggio di programmazione OO modellato dal linguaggio C e C++ di cui mantiene alcune caratteristiche.

Java è progettato per avere il minor numero possibile di dipendenze di implementazione per consentire agli sviluppatori di '*scrivere una volta, eseguire ovunque*'. Così infatti recita lo slogan di Java conosciuto con l'acronimo *WORA: write once, run anywhere*. Il codice è multi-piattaforma; una volta compilato sarà eseguibile su tutte le piattaforme per cui è previsto il supporto del linguaggio.

L'indipendenza dalla piattaforma è ottenuta grazie all'uso di uno strato software chiamato *Java Virtual Machine (JVM)* che traduce le istruzioni dei codici binari (*byte-code*), indipendenti dalla piattaforma e generati dal compilatore java, in istruzioni eseguibili dalla macchina che sta ospitando l'applicazione.

E' uno dei linguaggi di programmazione più popolari utilizzato, in particolare per le applicazioni web, le applicazioni client-server, le applicazioni basate su interfacce REST e micro-servizi.

E' stato presentato nel 1995: creato da *James Gosling* per l'azienda Sun Microsystems con il nome di *Oak*, oggi è di proprietà di Oracle, che ne cura la gestione. Si tratta di un linguaggio gratuito, orientato agli oggetti, basato su classi e tipizzato staticamente.

La sintassi orientata ad oggetti di Java supporta la creazione di classi, consentendo al programmatore di scrivere codice stabile e riutilizzabile per mezzo del paradigma *OO*.

Oltre ad essere il linguaggio di programmazione, Java fa anche riferimento all'intero ecosistema che gli gravita attorno, composto da tre componenti fondamentali:

1. Java Virtual Machine (JVM)

Un ambiente di esecuzione virtuale, indipendente dalla piattaforma, che converte il bytecode Java in linguaggio macchina e lo esegue.

2. *Java Runtime Environment (JRE)*

Un ambiente runtime necessario per eseguire programmi e applicazioni Java.

3. *Java Development Kit (JDK)*

Il componente principale dell'ambiente Java che contiene *JRE* insieme al compilatore Java, al debugger Java e ad altre classi.

La storia di Java: le origini fino ad oggi

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

strumento che fosse applicabile ad ogni tipo di apparato elettronico al fine di garantire un alto grado di interattività, diminuire i tempi di sviluppo, abbassare i costi di implementazione utilizzando un unico ambiente operativo.

Avendo subito focalizzato il problema sulla necessità di avere un linguaggio indipendente dalla piattaforma (il software non doveva essere legato ad un particolare processore) il gruppo iniziò i lavori nel tentativo di creare una tecnologia partendo dal linguaggio C++.

La prima versione del linguaggio fu chiamata Oak.

Attraverso una serie di eventi, quella che era la direzione originale del progetto subì vari cambiamenti, ed il target fu spostato dall'elettronica di consumo al world wide web.

Nel 1993 il "*National Center for Supercomputing Applications (NCSA)*" rilasciò *Mosaic*, una applicazione che consentiva agli utenti di accedere mediante interfaccia grafica ai contenuti di Internet allora ancora limitati al semplice testo. Nell'arco di un anno, da semplice strumento di ricerca, Internet è diventato il mezzo di diffusione che oggi tutti conosciamo in grado di trasportare testo ed ogni tipo di contenuto multimediale: fu così che, durante il 1994 la SUN decise di modificare il nome di Oak in Java.

La prima versione del linguaggio consentiva lo sviluppo applicazioni stand-alone e di piccole applicazioni chiamate applet in grado di essere eseguite attraverso la rete.

Il 23 Maggio del 1995 la SUN ha annunciato formalmente Java. Da quel momento in poi il linguaggio è stato adottato da tutti i maggiori produttori di software incluse IBM, Hewlett Packard e Microsoft.

Nel 2007 SUN Microsystem decide di modificare la licenza di java a passare alla GNU - General Public License.

Il 27 gennaio del 2010 la SUN Microsystem è stata acquisita dalla Oracle.

Nel 2018 Oracle decide di ritirare la *General Public Licence*: Java 8 sarà l'ultima versione rilasciata con licenza libera.

Al giorno d'oggi possiamo sicuramente affermare che, Java è uno dei linguaggi di programmazione più importanti al mondo la cui comunità è composta da programmatori ed utenti che utilizzano tale codice superando i 4 milioni di persone. Inoltre, dopo l'emergere di Android, Java ha aumentato sempre più la sua presenza anche nel settore mobile.

La roadmap del linguaggio Java

A partire da jdk 8 (Marzo 2014) Oracle ha deciso di cambiare strategia designando solo alcune versioni come versioni con supporto a lungo termine (LTS): 8,11,17 sono le versioni LTS correntemente rilasciate e supportate.

La prossima tabella (provenienza sito ufficiale Java/Oracle), riporta la roadmap a partire da Java 7 della tecnologia.

Oracle Java SE Support Roadmap*†				
Release	**GA Date**	**Premier Support Until**	**Extended Support Until**	**Sustaining Support**
7 (LTS)	July 2011	July 2019	July 2022	Indefinite
8 (LTS)	March 2014	March 2022	December 2030	Indefinite
9 (non-LTS)	September 2017	March 2018	Not Available	Indefinite
10 (non-LTS)	March 2018	September 2018	Not Available	Indefinite
11 (LTS)	September 2018	September 2023	September 2026	Indefinite
12 (non-LTS)	March 2019	September 2019	Not Available	Indefinite
13 (non-LTS)	September 2019	March 2020	Not Available	Indefinite
14 (non-LTS)	March 2020	September 2020	Not Available	Indefinite
15 (non-LTS)	September 2020	March 2021	Not Available	Indefinite
16 (non-LTS)	March 2021	September 2021	Not Available	Indefinite
17 (LTS)	September 2021	September 2026	September 2029	Indefinite
18 (non-LTS)	March 2022	September 2022	Not Available	Indefinite
19 (non-LTS)	September 2022	March 2023	Not Available	Indefinite
20 (non-LTS)	March 2023	September 2023	Not Available	Indefinite
21 (LTS)	September 2023	September 2028	September 2031	Indefinite

Secondo la nuova roadmap, Oracle intende realizzare future versioni LTS ogni due anni, il che significa che la prossima versione LTS pianificata sarà Java 21 a settembre 2023.

Le versioni non LTS sono considerate un insieme cumulativo di miglioramenti dell'implementazione della versione LTS più recente. Una volta resa disponibile una nuova versione di funzionalità, qualsiasi versione precedente non LTS verrà considerata sostituita. Ad esempio, Java SE 9 era una versione non LTS e subito sostituita da Java SE 10 (anche non LTS), Java SE 10 a sua volta è stata immediatamente sostituita da Java SE 11. Java SE 11 è comunque una versione LTS, e quindi riceverà supporto e rilasci periodici di aggiornamento, anche se Java SE 12 è stato rilasciato.

Licenza di distribuzione

Prima di JDK 8 e dell'acquisto di SUN Microsystem da parte di Oracle, Java veniva rilasciato con licenza d'uso gratuita che prevedeva il rilascio gratuito degli aggiornamenti pubblici.

Il mondo dello sviluppo Java fu scosso nel 2019 quando Oracle decise di cambiare la politica di licenza per JDK 8: poiché gli aggiornamenti pubblici gratuiti per Java 8 (ancora la versione più popolare tra gli sviluppatori) non venivano più forniti, le aziende dovevano pagare per il supporto commerciale o cercare una alternativa non commerciale. La tecnologia poteva comunque essere utilizzata gratuitamente per uso personale e non commerciale.

Con JDK 11 (settembre 2018) le cose non cambiarono, ma fu rilasciata una versione del JDK chiamata OpenJDK e rilasciata con licenza d'uso GPLv2+CPE per tutti gli utenti.

Da notare che non esiste una vera differenza tecnica tra Oracle JDK e OpenJDK: il processo di compilazione per Oracle JDK è basato su quello di OpenJDK.

Tuttavia, poiché Oracle JDK è orientato a clienti enterprise risulta essere migliore di altre per quanto riguarda la reattività e le prestazioni della JVM.

Esistono molte versioni della JVM basate tutte su OpenJDK e quindi rilasciate con stessa licenza. Amazon Corretto ad esempio, è una distribuzione gratuita e pronta per la produzione, basata su OpenJDK. Amazon Corretto offre supporto a lungo termine che include miglioramenti delle prestazioni e soluzioni per problemi relativi alla sicurezza.

Fino alla versione 17, la roadmap prevedeva una versione LTS di Java ogni 3 anni; nel 2021,Oracle introdusse una nuova roadmap di rilascio LTS una ogni due anni, e a partire da Java 17 ha introdotto la licenza d'uso "Oracle No-Fee Terms and Conditions" (NFTC)che rende JDK 17 e successivi gratuito per uso commerciale e di produzione con alcune restrizioni:

1. Sviluppo, test, prototipazione e dimostrazione delle applicazioni purché il software sia proprietario o sviluppato dalla propria azienda;

2. Esecuzione del software per uso personale o per operazioni interne all'azienda. Operazioni aziendali interne significano che si può implementare e utilizzare Oracle JDK all'interno della propria azienda;

3. Ridistribuzione del software secondo i termini NFTC senza addebitare alcuna tariffa ai licenziatari.

L'utilizzo di altre funzionalità commerciali (ad esempio, Java Advanced Management Console o GraalVM Enterprise Edition), e la ridistribuzione all'interno di programmi commerciali non è regolato dall'NTFC e rimane pertanto sotto licenza commerciale.

Caratteristiche. Ovvero ... Cosa fa di Java il linguaggio java?

Java è un linguaggio orientato agli oggetti progettato per massimizzare la portabilità del codice. In generale, un programma Java è un insieme di file di testo con estensione *.java* che viene compilato in uno o più file di *byte-code* con estensione *.class*.

Un file sorgente non dovrebbe essere più lungo di 2000 righe di codice. Nonostante sia una convenzione accettata ritengo che, in una applicazione ad oggetti ben modellata, un file sorgente non dovrebbe superare le 400-500 righe a meno di necessità particolari.

Ogni file sorgente contiene una sola classe o interfaccia pubbliche. il nome del file sarà identico, a meno dell'estensione, al nome della classe.

Java è un linguaggio case sensitive e, come tale, anche i nomi dei file rispettano i formato (minuscolo o maiuscolo) dei caratteri.

Il linguaggio java ha le seguenti caratteristiche:

1. *Semplicità.*

*T*utto in java è un oggetto; non sono consentiti puntatori, coercions automatiche (ovvero conversione implicita di tipo), ereditarietà multipla, ne tanto meno salti condizionati goto. La programmazione funzionale è stata introdotta in java dalla versione 7 e poi migliorata nella 8.

2. *Gestione della memoria.*

*L*a gestione della memoria è automatica ed è svolta dal *Garbage Collector*; il *Garbage Collector* è una parte dell'interprete che tiene traccia di tutti gli oggetti, e quando si accorge che un oggetto non può più essere utilizzato, automaticamente libera la memoria per rimetterla a disposizione del programma; per stabilire che un oggetto non può più essere utilizzato conta il numero dei riferimenti: quando un oggetto non ha più riferimenti la sua memoria può essere recuperata.

3. *Type safety (sicurezza dei tipi).*

Java è un linguaggio *type-safe*, ovvero esiste una relazione esplicita e controllata tra una variabile e il suo tipo (intero, virgola mobile, stringa, ecc..).
Quanti hanno programmato con linguaggi quali il C e il C++, oppure in Javascript, conoscono perfettamente quanti sono i problemi a cui si va incontro lavorando con linguaggi che non porgono attenzione a questa caratteristica: le conversioni implicite tra tipi, ad esempio interi a booleani o puntatori void ad altri puntatori, che sono caratteristiche di quei linguaggi divengono al tempo stesso la principale sorgente di errori di programmazione.

4. *Java è un linguaggio a tipizzazione statica.*

Un linguaggio è tale quando il tipo di dati di una variabile è noto al momento della compilazione. Ciò significa che è necessario specificare il tipo di variabile o dichiarare la variabile prima di poterla utilizzare.

Un *tipo* di dati definisce un insieme di possibili valori, e un insieme di operazioni che possono essere applicate su tali valori. I tipi ammessi in un programma Java sono i seguenti:

1. otto tipi primitivi definiti dal linguaggio (boolean, byte, char, short, int, long, float, double);

2. tipi reference (classi, interfacce e tipi array);

3. il null type;

4. *il tipo degenere void.*

5. *Gestione avanzata degli errori.*

Grazie alle eccezioni, anziché tentare di prevedere le situazioni di errore, Java esegue le operazione in blocchi di codice controllato eliminando la maggior parte dei problemi e delle ambiguità già in fase di scrittura del codice e di compilazione.

6. *Sicurezza.*

La sicurezza è un aspetto essenziale del codice moderno. Per raggiungere l'obiettivo, Java usa vari accorgimenti:

a) non permette l'uso dei puntatori (in realtà Java usa dei puntatori per fare riferimento agli oggetti ma i puntatori sono usati solo internamente e non sono disponibili ai programmatori);

b) Java è un linguaggio tipizzato staticamente, è in grado di svolgere la maggior parte dei controlli sul codice già al momento della compilazione.

c) l'interprete verifica l'integrità dei byte-code prima dell'esecuzione controllando che formino un programma valido, cioè che il codice rispetti i vincoli del linguaggio (quindi per esempio che non sia stato prodotto da un compilatore modificato che permetta di realizzare operazioni che non dovrebbero essere permesse);

d) la macchina virtuale assegna ad ogni applicazione il proprio ambiente di runtime che isola le applicazioni una dall'altra;

e) è stato sviluppato un modello di sicurezza che permette di stabilire misure specifiche per ogni applicazione: per esempio si può impedire ad una applicazione di leggere o scrivere sul disco o di effettuare connessioni in rete.

7. *Indipendente dalla piattaforma.*

Il codice Java può essere eseguito su qualsiasi piattaforma come Windows, Linux, iOS o Android senza riscriverlo. Ciò lo rende particolarmente efficiente nell'ambiente di oggi, dove si vogliono eseguire applicazioni su più dispositivi.

8. *Risorse di apprendimento di alta qualità.*

Java è presente sul mercato da parecchio tempo, pertanto sono disponibili molte risorse di apprendimento per i nuovi programmatori nonché una vasta comunità di esperti. Documentazione dettagliata, libri completi e corsi supportano gli sviluppatori nella curva di apprendimento.

9. Strumenti di sviluppo di alta qualità.

Java offre vari strumenti a supporto come l'editing automatizzato, il debug, i test.. Tali strumenti rendono la programmazione con Java efficiente in termini di tempo e di costi.

10. Funzioni e librerie integrate.

Con l'utilizzo di Java, gli sviluppatori non devono scrivere ogni nuova funzione da zero; Java fornisce un ricco ecosistema di funzioni e librerie integrate per sviluppare varie applicazioni.

11. Supporto alla programmazione concorrente.

Una delle potenti caratteristiche del linguaggio Java è il supporto per la programmazione concorrente o parallela. Tale caratteristica permette di organizzare il codice di una stessa applicazione in modo che possano esserne mandate in esecuzione contemporanea più parti in concorrenza fra loro.

12. Maven.

E' uno strumento completo per la gestione di progetti software Java, in termini di compilazione del codice, distribuzione, documentazione e collaborazione del team di sviluppo. E' in grado di scaricare automaticamente dipendenze del progetto dai repositories centralizzati, e occuparsi delle dipendenze transitive.

Indipendenza dalla piattaforma

Le istruzioni binarie di Java, indipendenti dalla piattaforma, sono comunemente conosciute come byte-code. Il byte-code è prodotto dal compilatore e necessita di uno strato di software per essere eseguito. Quest'applicazione, detta interprete (*Immagine 10 JVM e indipendenza della piattaform*), è nota come *Java Virtual Machine* (che per semplicità indicheremo con JVM). La JVM è un programma scritto mediante un qualunque linguaggio di programmazione, è dipendente dalla piattaforma su cui deve eseguire il byte-code, e traduce le istruzioni Java dalla forma di byte-code in istruzioni comprensibili dal processore della macchina che ospita l'applicazione.

La JVM è parte del *Java Runtime Environment* (JRE). Il JRE, è uno strato di software che viene eseguito in aggiunta al software del sistema operativo di un computer e fornisce le librerie di base ed altre risorse necessarie all'esecuzione di un programma java.

Non essendo il byte-code legato ad una particolare architettura hardware, questo fa sì che per trasferire un'applicazione Java da una piattaforma ad un'altra è necessario solamente che la nuova piattaforma sia dotata di un'apposita JVM. In presenza di un interprete un'applicazione Java potrà essere eseguita su qualunque piattaforma senza necessità di essere ulteriormente compilata.

Le versioni più recenti della JVM utilizzano strumenti chiamati "Just In Time Compilers", ovvero compilatori in grado di tradurre il byte-code in un formato eseguibile per una specifica piattaforma al momento dell'esecuzione del programma Java.

Un ultimo aspetto interessante di Java è quello legato agli sviluppi che la tecnologia sta avendo: negli ultimi anni infatti, molti produttori di sistemi elettronici hanno iniziato a rilasciare processori in grado di eseguire direttamente il byte-code a livello di istruzioni macchina senza l'uso di una virtual machine.

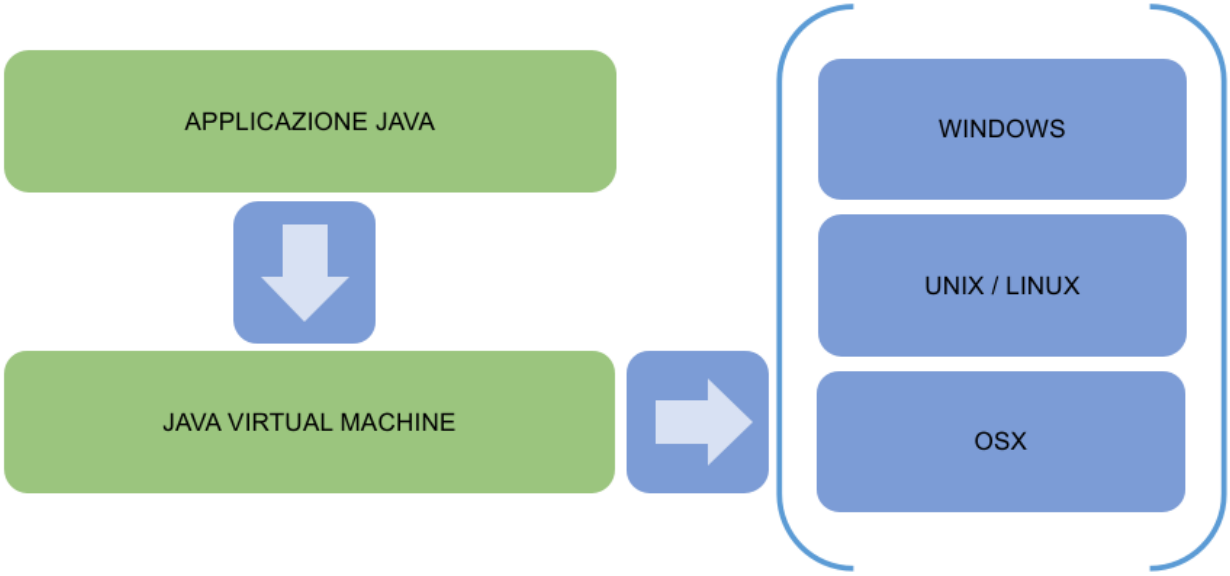


Immagine 10 JVM e indipendenza della piattaforma

Gestione della memoria: garbage collector

Un problema scottante quando si parla di programmazione è la gestione della memoria; quando si progetta un'applicazione, è molto complesso affrontare le problematiche inerenti al mantenimento degli spazi di memoria. Una gestione della memoria superficiale o mal pensata è spesso causa di un problema noto come *memory leak*: l'applicazione alloca risorse senza riuscire a rilasciarle completamente determinando la perdita di piccole porzioni di memoria che, se sommate, possono provocare l'interruzione anomala dell'applicazione o, nel caso peggiore, dell'intero sistema che la ospita. E' facile immaginare l'instabilità di sistemi informativi affetti da questo tipo di problema. Tali applicazioni richiedono spesso lo sviluppo di complesse procedure specializzate nella gestione, nel tracciamento e nel rilascio della memoria allocata.

Java risolve il problema alla radice sollevando il programmatore dall'onere della gestione della memoria grazie ad un meccanismo detto *Garbage Collector* . Il *Garbage Collector*, tiene traccia degli oggetti utilizzati da un'applicazione Java, e delle referenze a tali oggetti. Ogni volta che un oggetto non è più referenziato (ovvero utilizzato da altre classi) per tutta la durata di uno specifico intervallo temporale, è rimosso dalla memoria e la risorsa liberata è nuovamente messa a disposizione dell'applicazione che potrà continuare a farne uso.

Questo meccanismo è in grado di funzionare correttamente in quasi tutti i casi anche se molto complessi, ma non si può affermare che è completamente esente da problemi. Esistono, infatti, dei casi documentati, di fronte ai quali il *Garbage Collector* non è in grado di intervenire. Un caso tipico è quello della "referenza circolare" in cui un oggetto A referenzia un oggetto B e viceversa, ma l'applicazione non sta utilizzando nessuno dei due come schematizzato nella Immagine 11 Referenza circolar

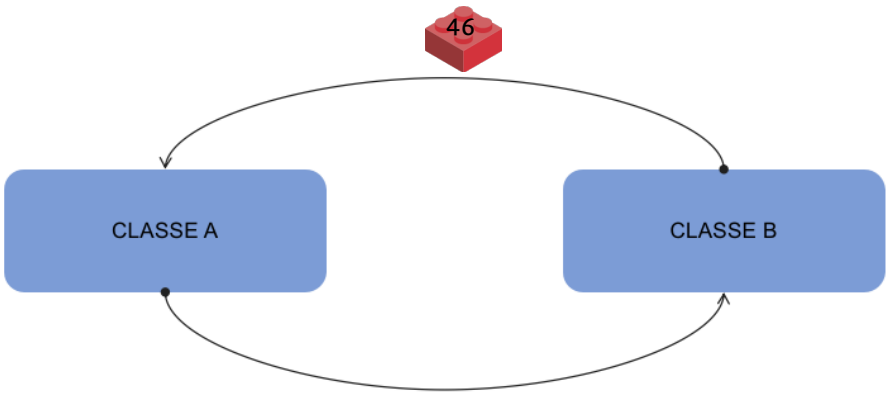


Immagine 11 Referenza circolare

Il classloader java

Quando la JVM è avviata la prima volta, il primo passo che deve compiere è quello di caricare le classi indispensabili all'applicazione affinché possa essere eseguita. Questa fase avviene secondo uno schema temporale ben preciso a carico di un modulo interno chiamato *launcher*.

Le prime ad essere caricate dal *launcher* sono le classi di base necessarie alla piattaforma Java per fornire lo strato di supporto alla applicazione a cui fornirà l'ambiente di *run-time* (la prima classe da caricare è quella che contiene il metodo *main* della applicazione).

Il secondo passo è caricare le classi Java appartenenti alle librerie di oggetti messi a disposizione con il JRE ed utilizzate all'interno della applicazione.

Infine vengono caricate le classi componenti l'applicazione e definite dal programmatore.

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

Responsabile del caricamento delle classi è un modulo della JVM chiamato *classloader*. Il *classloader* è utilizzato per caricare le classi dal file system locale. Di default il comportamento del *classloader* sarà quello di cercare un file *.class* relativo alla classe da caricare nel file system locale, nei path indicati nella variabile d'ambiente *CLASSPATH*.

Il funzionamento del Garbage Collector e del classloader sarà discusso, assieme al modello della memoria, in una sezione dedicata del libro.

Il Java Software Development Kit (SDK)

Java *Software Development Kit (SDK)* è un insieme di strumenti ed utilità ed è messo a disposizione gratuitamente da tanti produttori di software. Molte organizzazioni mantengono e distribuiscono la loro versione del Java SDK tra cui Amazon che distribuisce il proprio *Amazon Corretto* basato sul progetto open source *OpenJDK*. L'insieme base o standard delle funzionalità è comunque supportato direttamente da Oracle ed include un compilatore (*javac*), la JVM (*java*), un debugger e tanti altri strumenti necessari allo sviluppo di applicazioni Java.

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

1. *Windows 10,11;*

Installare il java SDK è un operazione semplice tanto quanto scaricare il pacchetto giusto dal sito di Oracle https://www.oracle.com/java/technologies/downloads/ ed eseguire il programma di installazione.

2. Linux Fedora;

Utilizziamo dnf per ottenere l'elenco delle versioni del Java SDK disponibili. Apriamo il nostro terminale e digitiamo:

dnf search openjdk

```
[test@localhost ~]$ dnf search openjdk
Last metadata expiration check: 0:01:31 ago on Wed 07 Apr 2021 02:43:05 PM CEST.
================================================================== Name & Summary Matched: openjdk
java-1.8.0-openjdk.x86_64 : OpenJDK 8 Runtime Environment
java-1.8.0-openjdk-accessibility.x86_64 : OpenJDK 8 accessibility connector
java-1.8.0-openjdk-accessibility-fastdebug.x86_64 : OpenJDK 8 accessibility connector for packages w
java-1.8.0-openjdk-accessibility-slowdebug.x86_64 : OpenJDK 8 accessibility connector for packages w
java-1.8.0-openjdk-demo.x86_64 : OpenJDK 8 Demos
java-1.8.0-openjdk-demo-fastdebug.x86_64 : OpenJDK 8 Demos optimised with full debugging on
java-1.8.0-openjdk-demo-slowdebug.x86_64 : OpenJDK 8 Demos unoptimised with full debugging on
java-1.8.0-openjdk-devel.x86_64 : OpenJDK 8 Development Environment
```

Immagine 12 Elenco dei java development kit disponibili per dnf

Copiamo la versione che vogliamo installare ed eseguiamo:

sudo dnf install [package name]

Ad esempio se vogliamo installare Java 11 sulla nostra macchina eseguiremo:

sudo dnf install java-11-openjdk.x86_64

dove .x86_64 rappresenta l'architettura x86 a 64 bit. A questo punto possiamo verificare l'installazione utilizzando il comando

java -version

```
[test@localhost ~]$ java --version
openjdk 11.0.10 2021-01-19
OpenJDK Runtime Environment 18.9 (build 11.0.10+9)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.10+9, mixed mode, sharing)
[test@localhost ~]$
```

Immagine 13 java --version output

3. Linux Ubuntu;

Un'opzione per l'installazione di Java consiste nell'utilizzare la versione inclusa in Ubuntu. Per impostazione predefinita, Ubuntu 22.04 include Open JDK 11. Per installare la versione OpenJDK di Java, prima è sempre bene aggiornare l'indice dei pacchetti apt:

sudo apt update

quindi possiamo digitare:

java -version

Se il pacchetto è installato otterremo lo stesso messaggio come mostrato in *Immagine 13 java --version output* oppure l'output mostrato in figura:*Immagine 14 java not found outpu*. Possiamo quindi procedere ad installare il JDK di default oppure una versione differente tra quelle disponibili.

```
Command 'java' not found, but can be installed with:

sudo apt install default-jre              # version 2:1.11-72build1, or
sudo apt install openjdk-11-jre-headless  # version 11.0.14+9-0ubuntu2
sudo apt install openjdk-17-jre-headless  # version 17.0.2+8-1
sudo apt install openjdk-18-jre-headless  # version 18~36ea-1
sudo apt install openjdk-8-jre-headless   # version 8u312-b07-0ubuntu1
```

Immagine 14 java not found output

Il compilatore Java - javac

Il compilatore Java (*javac*) fornito con il Java SDK, può essere trovato nella sotto cartella *bin* della cartella di installazione. Questo programma a linea di comando è un'applicazione che trasforma un file con estensione *.java* in un file con estensione *.class* contenente il byte-code relativo alla definizione della classe. Un file con estensione *.java* è un normale file in formato ASCII contenente del codice Java valido; il file generato dal compilatore Java potrà essere caricato ed eseguito dalla *Java Virtual Machine*.

Java è un linguaggio *case sensitive* ovvero sensibile al formato maiuscolo o minuscolo di un carattere: di conseguenza è necessario passare al compilatore java il parametro contenente il nome del file da compilare rispettandone il formato dei caratteri. Ad esempio, volendo compilare il file *MiaPrimaApplicazione.java* la corretta esecuzione del compilatore java è la seguente:

javac MiaPrimaApplicazione.java

Qualunque altro formato produrrà un errore di compilazione, ed il processo di generazione del byte-code sarà interrotto.

Un altro accorgimento da rispettare al momento della compilazione riguarda l'estensione del file da compilare: la regola vuole che sia sempre scritta esplicitamente su riga di comando. Ad

esempio la forma utilizzata di seguito per richiedere la compilazione di una file non è corretta, e produce l'interruzione immediata del processo di compilazione:

javac MiaPrimaApplicazione

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

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

Opzioni standard del compilatore

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

javac [opzioni] [filesorgenti] [@ElencoSorgenti]

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

1. -classpath classpath

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

2. -d cartella

Imposta la cartella di destinazione all'interno della quale verranno memorizzati i file contenenti il byte-code delle classi compilate. Se la classe fa parte di un package Java, il compilatore salverà i file con estensione *.class* in una sotto-cartella che rifletta la struttura del package e creata a partire dalla cartella specificata dall'opzione. Se questa opzione non viene specificata, il compilatore salverà i file contenenti il byte-code all'interno della cartella corrente. Se l'opzione *-sourcepath* non viene specificata, questa opzione viene utilizzata dal compilatore per trovare sia file *.class* che file *.java.*

3. -sourcepath sourcepath;

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

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

javac @ElencoSorgenti

Compiliamo una classe java

E' arrivato il momento di compilare la nostra prima applicazione java. Per completare questo piccolo esercizio è possibile utilizzare qualsiasi editor di testo come notepad o wordpad se utilizzate Windows, oppure vim se utilizzate Linux. Creiamo quindi la nostra definizione di classe copiando all'interno del vostro editor di testo il codice riportate di seguito.

Dopo anni che programmo penso di aver scritto il solito programmino per iniziare "Hello world" decine di volte, e con decine di linguaggi di programmazione differenti. Poiché inizia a starmi antipatico voglio spezzare questo cerchio e proporvi la mia versione della vostra prima applicazione. *LaMiaPrimaApplicazione*

Ed ecco il codice:

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

Salviamo il file nella cartella corrente facendo attenzione che il nome sia LaMiaPrimaApplicazione.java ed invochiamo il comando

javac LaMiaPrimaApplicazione.java

Al termine della compilazione troveremo nella cartella il file *LaMiaPrimaApplicazione.class*. Per eseguire la nostra applicazione Java basterà invocare la Java Virtual Machine nel modo seguente:

java LaMiaPrimaApplicazione

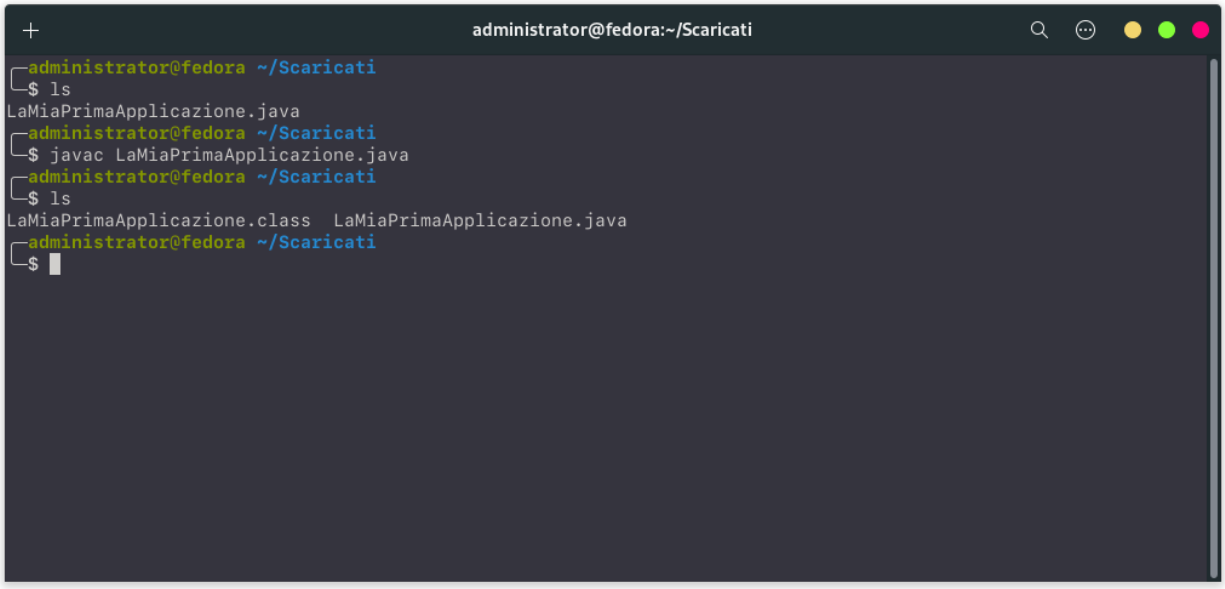


Immagine 15 Compilazione di una classe java

La sintassi della Java Virtual Machine

La sintassi generale del comando java è la seguente:

java [opzioni] class [argomenti ...]

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

1. -classpath classpath

Ridefinisce o sostituisce il valore della variabile d'ambiente CLASSPATH. Se nessuno dei due meccanismi viene utilizzato per comunicare al compilatore la posizione della classi necessarie alla esecuzione del bytecode, verrà utilizzata la cartella corrente;

2. -verbose:class

Visualizza le informazioni relative ad ogni classe caricata;

3. -verbose:gc

Visualizza informazioni relative agli eventi scatenati dal garbage collector.

4. -version

Visualizza le informazioni relative alla versione del prodotto ed esce;

5. -showversion

Visualizza le informazioni relative alla versione del prodotto e prosegue;

6. -? | -help

Visualizza le informazioni sull'uso della applicazione ed esce;

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

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

Salviamo in un file *TestArgomenti.java*. Dopo aver compilato la classe, eseguendo l'applicazione utilizzando la riga di comando seguente:

```
java TestArgomenti primo secondo terzo quarto
```

il risultato prodotto sarà il seguente:

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

Inserire commenti nel codice

Commentare correttamente il codice sorgente di un'applicazione è importante: primo perché un codice ben commentato è facilmente leggibile da chiunque; secondo, perché i commenti aiutano il programmatore ad evidenziare aspetti specifici di un algoritmo riducendo sensibilmente gli errori dovuti a distrazioni in fase di scrittura. Tuttavia i commenti sono spesso insufficienti e talvolta assenti.

Java consente di inserire commenti supportando entrambi i formati di C e C++: il primo include i commenti all'interno di blocchi di testo delineati dalle stringhe /* ed */, il secondo utilizza la stringa // per indicare una linea di documentazione. Nel prossimo esempio abbiamo modificato l'applicazione precedentemente scritta inserendo all'interno commenti utilizzando entrambe le forme descritte:

```
public class MiaPrimaApplicazione {
        /**
        * Il metodo main rappresenta il punto di ingresso
        * all'interno della applicazione MiaPrimaApplicazione
        */
```

```
    public static void main(String[] argv) {
        // Visualizza sullo schermo il messaggio
        // Finalmente la mia prima applicazione
        System.out.println("Finalmente la mia prima applicazione");
    }
}
```

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

Negli ultimi anni java è stato modificato fino a raggiungere oggi una capacità espressiva pari solo a pochi altri linguaggi. Una applicazione Java se scritta bene si legge facilmente ed intuitivamente. Questa caratteristica sta spostando l'attenzione dei programmatori alla qualità del codice tralasciando, spesso e volentieri, i commenti all'interno del codice stesso.

Tutto questo senza ridurre però la leggibilità del codice prodotto.

I 'doc comments' o javadoc

Un errore che spesso si commette durante lo sviluppo di un'applicazione è dimenticare o tralasciare volutamente la documentazione tecnica. Documentare il codice del prodotto che si sta sviluppando, richiede spesso giorni di lavoro e di conseguenza costi che pochi sono disposti a sostenere: il risultato è un prodotto poco o addirittura completamente non mantenibile.

Java fonde gli aspetti descrittivo e documentale, consentendo di utilizzare i commenti inseriti dal programmatore all'interno del codice sorgente dell'applicazione per produrre la documentazione tecnica necessaria ad una corretto rilascio di un prodotto.

Oltre ai formati descritti nel paragrafo precedente, Java prevede un terzo formato che utilizza le stringhe /** e */ per delimitare blocchi di commenti chiamati *doc comments*. I commenti che utilizzano questo formato sono utilizzati da uno strumento fornito con il Java SDK per generare automaticamente documentazione tecnica di una applicazione in formato ipertestuale. Questo strumento è chiamato *javadoc* ed è rilasciato con il Java SDK.

Javadoc esegue la scansione del codice sorgente, estrapola le dichiarazioni delle classi ed i *doc comments* e produce una serie di pagine HTML contenenti informazioni relative alla descrizione della classe, dei metodi e dei dati membri più una completa rappresentazione della gerarchia delle classi e le relazioni tra loro.

Oltre ai commenti nel formato descritto, questo strumento riconosce alcune etichette utili all'inserimento, all'interno della documentazione prodotta, di informazioni aggiuntive come l'autore di una classe o la versione. Le etichette sono precedute dal carattere '@': le più comuni sono elencate nella tabella seguente.

Etichette Javadoc		
Sintassi	**Descrizione**	**Applicabilità**
@see riferimento	Aggiunge una voce "see also" con un link definito da *riferimento*	classi, metodi, variabili
@author nome	Aggiunge una voce "Author" alla documentazione. Il parametro descrive il nome dell'autore della classe.	classi
@versione versione	Aggiunge una voce "Version" alla documentazione. Il parametro contiene il numero di versione della classe.	classi
@param parametro	Aggiunge la descrizione ad un parametro di un metodo.	metodi
@return descrizione	Aggiunge la descrizione relativa al valore di ritorno di un metodo.	metodi
@since testo	Aggiunge una voce "Since" alla documentazione. Il parametro identifica generalmente il numero di versione della classe a partire dalla quale il nuovo requisito è stato aggiunto.	classi, metodi, variabili
@deprecated testo	Imposta l'entità collegata come obsoleta.	classi, metodi, variabili
@throws classe descrizione	Aggiunge una voce "Throws" alla definizione di un metodo contenente il riferimento ad una eccezione generata e la sua descrizione.	metodi
@exception classe descrizione	Vedi @throws	metodi

Un esempio di doc comment è il seguente:

```
/**
 * La classe <STRONG>MiaPrimaApplicazione</STRONG> contiene solo il metodo
 * main che produce un messaggio a video.
 *
 * @version 0.1
 */
public class MiaPrimaApplicazione {
   /**
    * Il metodo main rappresenta il punto di ingresso
    * all'interno della applicazione MiaPrimaApplicazione
    *
    * @since 0.1
    *
    *      <PRE>
   * @param String[] : array contenente la lista dei parametri di input <br> passati per
```

```
     * riga di comando
     *      </PRE>
     *
     * @return void
     */
    public static void main(String[] argv) {
      // Visualizza sullo schermo il messaggio
      // Finalmente la mia prima applicazione
      System.out.println("Finalmente la mia prima applicazione");
    }
}
```

Java hotspot

Il maggior punto di forza di Java, la portabilità del codice tramite byte-code, è anche il suo tallone d'Achille. In altre parole, essendo Java un linguaggio tradotto (ovvero necessita di una virtual machine per essere eseguito), le applicazioni sviluppate con questa tecnologia hanno prestazioni inferiori rispetto ad altri linguaggi di programmazione come il C++.

Per colmare il gap, nell'aprile 1999 Sun ha introdotto uno dei più grandi cambiamenti in Java in termini di prestazione. La macchina virtuale HotSpot è una funzionalità chiave di Java che si è evoluta per consentire prestazioni paragonabili a (o meglio di) linguaggi come C e C++ utilizzando tecniche avanzate d'ottimizzazione del codice durante l'esecuzione, affiancate da una più efficiente gestione della memoria e dal supporto per i thread.

Questa macchina virtuale cerca di aumentare la velocità di esecuzione del bytecode convertendolo al volo in istruzioni assembly native della macchina fisica su cui si sta eseguendo l'applicazione. In genere per ottenere un aumento significativo delle prestazioni è sufficiente compilare in questo modo solo alcune parti critiche del programma, gli *hot spot* (*o punti critici*), appunto.

Non solo, il monitoraggio di questi punti critici dà la possibilità di cambiare dinamicamente e spostare l'attenzione su nuovi punti critici individuati durante il corso dell'esecuzione del programma.

Durante l'esecuzione vengono monitorate le parti di codice che sono eseguite più frequentemente: vengono tracciate informazioni relative al run-time della applicazione ed utilizzate dal compilatore *Just In Time (JIT)* per procedere all'ottimizzazione del codice. In questo modo *JIT* può procedere ad una ottimizzazione basata sulla conoscenza, e di conseguenza più consapevole della eventuale strategia di ottimizzazione da scegliere.

Questo approccio alla compilazione Just In Time ha un ulteriore vantaggio; poiché versioni successive della Java Virtual Machine migliorano di volta in volta le precedenti, ogni applicazione Java potrà beneficiare automaticamente di tutte le migliorie in termini di performance senza richiedere neanche che l'applicazione debba essere ricompilata.

Negli anni *hotspot* è stato migliorato ed ottimizzato. Sono state sviluppate nuove tecniche per la gestione della memoria; è stato sviluppato un nuovo *Garbage Collector* a bassa latenza, e importantissimo, è cambiata la gestione dei thread con un maggior supporto delle funzioni di sistema operativo su cui la JVM si appoggia. La possibilità di utilizzare *preemption* e *multi-tasking* a livello di sistema operativo si trasforma in una evidente maggiore velocità degli stessi che potranno avere a disposizione strumenti nativi ottimizzati per la memoria ed il processore su cui stanno girando nel dato istante.

4. Sintassi di java, dichiarazione di variabili ed operatori

Introduzione

La sintassi del linguaggio Java eredita in parte quella di C e C++ per la definizione di variabili e strutture complesse, ed introduce il concetto di variabili di dimensioni fisse ovvero non dipendenti dalla piattaforma.

Le espressioni in java rappresentano il meccanismo per effettuare calcoli all'interno della nostra applicazione; combinano variabili e operatori producendo un singolo valore di ritorno.

Le espressioni vengono utilizzate per generare valori da assegnare alle variabili o, come vedremo nel corso del capitolo, per modificare il corso della esecuzione di una applicazione: di conseguenza una espressione non rappresenta una unità di calcolo completa in quanto non produce assegnamenti o modifiche alle variabili della applicazione.

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

Nei prossimi paragrafi studieremo inoltre cosa sono le variabili, come utilizzarle e come implementare strutture dati più complesse mediante l'uso degli *array*. Infine, introdurremo le regole sintattiche e semantiche specifiche del linguaggio necessarie alla comprensione dei capitoli successivi.

Alcune regole sintattiche

Come C e C++, Java è un linguaggio indipendente dagli spazi, in altre parole, l'indentazione del codice di un programma ed eventualmente l'uso di più di una riga di testo sono opzionali. La sintassi del linguaggio Java può essere descritta da tre sole regole di espansione:

istruzione -> espressione

istruzione --> {
istruzione
[istruzione]

}

istruzione --> controllo_di_flusso
istruzione

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

Prendiamo in considerazione la terza regola:

istruzione --> controllo_di_flusso
istruzione

E sostituiamo il lato destro utilizzando la seconda espansione ottenendo:

istruzione --> controllo_di_flusso ***(2)*** *controllo_di_flusso*
istruzione *{*
istruzione
[istruzione]
}

Ove le parentesi quadre [] rappresentano blocchi opzionali. Applicando ora la terza regola di espansione otteniamo :

controllo_di_flusso ***(3)*** *controllo_di_flusso*
{ *{*
istruzione *controllo_di_flusso*
[istruzione] *istruzione*
} *}*

Prendiamo per buona che l'istruzione **if** sia una istruzione per il controllo del flusso della applicazione (controllo di flusso), e facciamo un ulteriore sforzo accettando sulla sua sintassi, ecco che l'ultima espansione assomiglierà al seguente blocco di codice:

```
if(i>=0)
{
  if(i==5)
        System.out.println(i);
  i++;
}
```

La seconda regola di espansione fornisce anche la definizione di *blocco di istruzioni* ovvero una sequenza di una o più istruzioni racchiuse all'interno di parentesi graffe.

Variabili

Per scrivere applicazioni Java, un programmatore deve poter creare oggetti. Per far questo, è necessario poterne rappresentare i dati. Il linguaggio Java mette a disposizione del

programmatore una serie di *tipi semplici* o *primitivi*, utili alla definizione di oggetti più complessi. I tipi numerici sono da considerarsi tutti con segno.

Per garantire la portabilità del byte-code da una piattaforma ad un'altra, Java fissa le dimensioni di ogni dato primitivo. Queste dimensioni sono quindi definite e non variano se passiamo da un ambiente ad un altro, cosa che non succede con gli altri linguaggi di programmazione.

La tabella a seguire schematizza i dati primitivi messi a disposizione da Java.

Variabili primitive Java			
tipo	**dimensione**	**valore minimo**	**valore massimo**
boolean	1 bit	-	-
char	16 bit	Unicode 0	Unicode 216 -1
byte	8 bit	-128	+127
short	16 bit	-215	+215 -1
int	32 bit	-231	+231 -1
long	64 bit	-263	+263 -1
float	32 bit	per numeri con la virgola, fino a 7 cifre dopo la virgola. Occupano 32 bit. In genere si aggiunge il suffisso f o F.	
double	64 bit	per numeri con la virgola, fino a 16 cifre dopo la virgola. Occupano 64 bit. In genere si aggiunge il suffisso d o D.	
void	-	-	-

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

tipo identificatore;

dove "tipo" è uno tra i tipi descritti nella prima colonna della tabella e definisce il dato che la variabile dovrà contenere, e l'identificatore rappresenta il nome della variabile. Il nome di una variabile può contenere caratteri alfanumerici, ma deve iniziare necessariamente con una lettera.

Il nome di una variabile segue solitamente la camelNotation: iniziano con una lettera minuscola, e qualora siano composte da più parole, ogni parola successiva alla prima ha l'iniziale maiuscola. Esempio: *nomeDiUnaVariabileCammellato*

E' possibile creare più di una variabile dello stesso tipo utilizzando una virgola per separare tra loro i nomi delle variabili:

tipo identificatore1, identificatore2, …. ;

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

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

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

Inizializzazione di variabili

Durante l'esecuzione del byte-code dell'applicazione, ogni volta che la JVM trova la dichiarazione di una variabile riserva spazio sulla memoria del computer per poterne rappresentare il valore, e ne associa l'indirizzo fisico all'identificatore per accedere al dato. Di conseguenza, ogni variabile Java richiede che al momento della dichiarazione le sia assegnato un valore iniziale. Ci sono due modi per inizializzare una variabile primitiva.

1. *Mediante inizializzazione esplicita nella dichiarazione.*

L'inizializzazione di una variabile Java può essere effettuata dal programmatore direttamente al momento della sua dichiarazione. La sintassi è la seguente:

tipo identificatore = valore;

dove valore rappresenta un valore legale per il tipo di variabile dichiarata, ed "=" rappresenta l'operatore di assegnamento.

2. *Mediante inizializzazione di default.*

Qualora il valore di una variabile non venga espressamente dichiarato, Java assegna ad ogni variabile un valore prestabilito. La tabella riassume il valore iniziale assegnato dalla JVM distinguendo secondo del tipo primitivo rappresentato.

Valori prestabiliti per le primitive	
tipo assegnato	**valore di default assegnato dalla JVM**
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0

Inferenza automatica del tipo

Negli ultimi anni, una parte importante dell'evoluzione del linguaggio Java è stata dedicata a rendere la sintassi più sintetica (nella programmazione solitamente si preferisce dire meno verbosa). Per fare ciò, ci si è concentrati sul come assegnare ulteriori compiti al compilatore. A partire dal Java 10, e poi definitivamente con Java 11 è stato introdotto il concetto di *inferenza automatica del tipo* per le variabili secondo cui il compilatore riesce a dedurre automaticamente il tipo della variabile locale che stiamo dichiarando permettendoci di utilizzare la parola **var** in luogo del tipo, e questo grazie alla sua natura di linguaggio tipizzato staticamente.

L'*inferenza automatica del tipo* in java funziona grazie alla capacità del compilatore di dedurre il tipo sulla base del risultato tornato dal lato destro di una istruzione var.

```
var bool = false; // dedotto il tipo boolean
var string = "Foqus";// dedotto il tipo String
var character= 'J'; // dedotto il tipo char
var integer = 8; // dedotto il tipo int
var byteInteger = (byte)8; // dedotto il tipo byte
var shortInteger = (short)8; // dedotto il tipo short
var longInteger = 8L; // dedotto il tipo long
var floatingPoint = 3.14F; // dedotto il tipo float
var doublePrecisionfloatingPoint = 3.14; // dedotto il tipo double
```

La deduzione del tipo funziona solo per variabili locali quindi non è applicabile per variabili d'istanza o dati membro della classe, per i tipi di ritorno o per il tipo di parametro dei metodi; inoltre non è possibile utilizzare la parola **var** per variabili locali che non siano inizializzate contestualmente alla dichiarazione, come nel seguente esempio:

```
var doublePrecisionFloatingPoint; //errore di compilazione
```

Variabili char e codifica del testo

Parlando della storia del linguaggio Java abbiamo evidenziato come la tecnologia proposta dalla SUN abbia avuto come campo principale di applicazione, Internet. Data la sua natura, era indispensabile dotarlo delle caratteristiche fondamentali a rispondere alle esigenze di internazionalizzazione proprie della più vasta rete del mondo.

La codifica ASCII, largamente utilizzata dalla maggior parte dei linguaggi di programmazione, utilizza una codifica ad otto bit idonea a rappresentare al massimo $2^8 = 256$ caratteri e quindi non adatta allo scopo.

Per superare i limiti imposti dal formato ASCII, fu adottato lo standard internazionale UNICODE. UNICODE è un sistema di codifica che assegna un numero univoco ad ogni carattere usato per la scrittura di testi in maniera indipendente dalla lingua, dalla piattaforma informatica e dal programma utilizzato.

Lo standard UNICODE utilizza una codifica a 16 bit studiata per rappresentare 2^{16}= 65536 caratteri di cui i primi 256 (UNICODE 0-255) corrispondono ai caratteri ASCII. Ognuna della 65536 possibili rappresentazioni è detta *code point*.

Per questo motivo, le variabili Java di tipo **char** utilizzano 16 bit per rappresentare il valore del dato, e le stringhe a loro volta sono rappresentate come sequenze di caratteri a 16 bit.

Questa caratteristica di Java rappresenta però una limitazione in quanto lo standard UNICODE comprende alcuni caratteri non usabile per la costruzione di stringhe di caratteri e letterali. Per risolvere il problema, Java utilizza la forma detta *sequenza di escape*

\uxxxx (xxxx=sequenza di massimo 4 cifre esadecimali)

Una sequenza di escape inizia con il carattere "\", contiene più di un carattere ima funziona come un singolo carattere perché non esiste una lettera che rappresenti testualmente il carattere di escape. Il compilatore converte la sequenza di caratteri in un singolo carattere di escape nel programma compilato. A seguire un elenco delle sequenze di escape con relativo carattere unico maggiormente utilizzate in programmazione Java:

\ ' Virgolette singole usate per i letterali dei caratteri
\ " Virgolette doppie usate per i letterali di stringa
\? Punto interrogativo
\ a Alert
\ f Feed form
\ n Nuova riga
\ t scheda orizzontale
\ v Scheda verticale
\ 0 Null

Variabili final: dichiarazione di costanti

A differenza di altri linguaggi di programmazione, Java non consente la definizione di costanti. Questo aspetto del linguaggio non è da considerarsi una limitazione perché è possibile simulare una costante utilizzando il modificatore **final**.

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

***final** tipo identificatore = valore;*

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

In Java è prassi definire le costanti con nomi composti da lettere tutte maiuscole, e se formati da più parole si dovrebbero separare con il carattere underscore '_'. Ad esempio nel caso di una variabile di tipo **final** utilizzeremo la forma: QUESTA_E_UNA_COSTANTE_FINAL

Scope di una variabile Java

Differentemente da linguaggi come il Pascal, in cui le variabili debbono essere dichiarate all'interno di un apposito blocco di codice, Java come il C e C++ lascia al programmatore la libertà di dichiarare le variabili in qualsiasi punto del codice del programma; altra caratteristica del linguaggio è che due o più variabili possono essere identificate dallo stesso nome.

Questa flessibilità richiede però che siano definite alcune regole per stabilire i limiti di una variabile evitando sovrapposizioni pericolose.

Lo *scope* di una variabile Java è la regione di codice all'interno della quale essa può essere referenziata utilizzando il suo identificatore, e ne determina il ciclo di vita individuando quando la variabile debba essere allocata o rimossa dalla memoria.

I blocchi di istruzioni ci forniscono il meccanismo necessario a determinare i confini dello scope di una variabile: di fatto, una variabile è visibile solo all'interno del blocco di istruzioni che contiene la sua dichiarazione ed ai sotto blocchi contenuti.

```
{
        int i;
        {
                //inizio scope della variabile somma
                int somma = somma+i;
                //finescope della variabile somma
        }
        //Questa riga di codice contiene un errore
        System.out.println("La somma vale: "+somma);
}
```

Il codice in esempio contiene un errore in quanto la variabile somma è visibile solo all'interno del blocco di istruzioni delimitato dalle parentesi graffe. La versione corretta del codice è la seguente:

```
{
        int i;
        int somma = somma+i;
        System.out.println("La somma vale: "+somma);
}
```

Nei capitoli successivi torneremo a parlare in dettaglio di questo tema.

Operatori

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

Operatori in ordine di precedenza	
operatore	**descrizione**
++ -- + -	Aritmetiche unarie e booleane
* / %	Aritmetiche
+ -	Addizione (o concatenazione), sottrazione
<< >> >>>	Shift di bit
(**tipo**)	Operatore di cast
< <= > >= instanceof	Operatori di comparazione
== !=	Uguaglianza e disuguaglianza
&	(bit a bit) AND
^	(bit a bit) XOR
|	(bit a bit) OR
&&	AND Logico
||	OR Logico
!	NOT Logico
expr ? expr :expr	Condizione a tre
= *= /+ %= += -= <<= >>>= n &= ^= |=	Assegnamento e di combinazione

Gran parte degli operatori Java appartengono all'insieme degli operatori del linguaggio C, a cui ne sono stati aggiunti nuovi a supporto delle caratteristiche proprie del linguaggio.

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

Come in C, gli operatori di uguaglianza e disuguaglianza sono == (uguale a) e != (diverso da): l'uso dell'operatore *digrafo*[4] == è necessario dal momento che il carattere = è utilizzato esclusivamente come operatore di assegnamento, e l'operatore != compare in questa forma per consistenza con la definizione dell'operatore logico ! (NOT).

Anche gli operatori *bit a bit* e quelli logici derivano dal linguaggio C. Nonostante le analogie in comune, sono completamente sconnessi tra loro. Ad esempio l'operatore & è utilizzato per combinare due interi operando bit per bit e l'operatore && è utilizzato per eseguire l'operazione

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

di AND logico tra due espressioni booleane: quindi, mentre (1011 & 1001) restituirà 1001, l'espressione (a == a && b != b) restituirà false.

Sempre dal linguaggio C, Java eredita gli operatori unari di incremento e decremento ++ e --:

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

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

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

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

Gli operatori logici && ed || in java sono di tipo *short-circuit* ovvero, se il lato sinistro di un'espressione fornisce informazioni sufficienti a completare l'intera operazione, il lato destro dell'espressione non sarà valutato. Per esempio, si consideri l'espressione booleana

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

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

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

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

E' sempre buona norma disegnare le porzioni di codice che contengono espressioni booleane in modo da sfruttare questa caratteristica degli operatori logici.

Operatori di assegnamento

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

tipo identificatore = espressione;

dove espressione rappresenta una qualsiasi espressione che produce un valore compatibile con il tipo definito da tipo, e identificatore rappresenta la variabile che conterrà il risultato. Tornando alla tabella degli operatori definita nei paragrafi precedenti, vediamo che l'operatore di assegnamento ha la priorità più bassa rispetto a tutti gli altri. La riga di codice Java produrrà

quindi la valutazione della espressione ed infine l'assegnamento del risultato alla variabile. Ad esempio:

int sommadiInteri= 5+10;

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

Oltre all'operatore = Java mette a disposizione del programmatore una serie di operatori di assegnamento di tipo *shortcut* (in italiano scorciatoia), definiti nella prossima tabella. Questi operatori combinano un operatore aritmetico o logico con l'operatore di assegnamento.

Operatori shortcut		
operatore	**utilizzo**	**forma equivalente**
+=	sx += dx	sx = sx + dx;
-=	sx -= dx	sx = sx - dx;
*=	sx *= dx	sx = sx * dx;
/=	sx /= dx	sx = sx / dx;
%=	sx %= dx	sx = sx % dx;
&=	sx &= dx	sx = sx & dx;
|=	sx |= dx	sx = sx | dx;
^=	sx ^= dx	sx = sx ^ dx;
<<=	sx <<= dx	sx = sx << dx;
>>=	sx >>= dx	sx = sx >> dx;
>>>=	sx >>>= dx	sx = sx <<< dx;

Operatore di cast

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

(nuovo tipo) identificatore|espressione

dove nuovo tipo rappresenta il tipo dopo la conversione, identificatore è una qualsiasi variabile numerica o carattere, espressione un'espressione che produca un valore numerico.

Prima di effettuare un'operazione di cast tra tipi primitivi, è necessario assicurarsi che non ci siano eventuali errori di conversione. Di fatto, il cast di un tipo numerico in un altro con minor precisione ha come effetto quello di modificare il valore del tipo con precisione maggiore affinché possa essere memorizzato in quello con precisione minore: ad esempio, il tipo long può rappresentare tutti i numeri interi compresi nell'intervallo (-2^{63} , $+2^{63}-1$), mentre il tipo int quelli compresi nell'intervallo (-2^{32} , $+2^{32}-1$).

La prossima applicazione di esempio effettua tre operazioni di cast: il primo tra una variabile di tipo **long** in una di tipo **int** evidenziando la perdita del valore del tipo promosso; il secondo di un tipo **int** in un tipo **long**; il terzo, di un tipo **char** in un tipo **int** memorizzando nella variabile intera il codice UNICODE del carattere 'A'.

```
public class Cast {
  public static void main(String[] argv) {
    long tipoLong;
    int tipoInt;
    char tipoChar;
    // Cast di un tipo long in un tipo int
    tipoLong = Long.MAX_VALUE;
    tipoInt = (int) tipoLong;
    System.out.println("La variabile di tipo long vale: " + tipoLong
        + ", La variabile di tipo int  vale:" + tipoInt);
    // Cast di un tipo int in un tipo long
    tipoInt = Integer.MAX_VALUE;
    tipoLong = (long) tipoInt;
    System.out.println("La variabile di tipo long vale: " + tipoLong + ", La variabile di tipo int
        vale:" + tipoInt);
    // Cast di un tipo char in un tipo int
    tipoChar = 'A';
    tipoInt = (int) tipoChar;
    System.out.println("La variabile di tipo char vale: " + tipoChar + ", La variabile di tipo int
        vale:"  + tipoInt);
  }
}
```

Il risultato della esecuzione del codice è riportato di seguito.

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

Operatori aritmetici

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

Gli operatori aritmetici sono suddivisi in due classi: operatori binari ed operatori unari. Gli operatori binari (ovvero operatori che necessitano di due operandi) sono cinque e sono schematizzati nella tabella seguente:

operatori aritmetici		
operatore	**utilizzo**	**descrizione**
+	res=sx + dx	res = somma algebrica di dx ed sx
-	res=sx - dx	res = sottrazione algebrica di dx da sx
*	res=sx * dx	res = moltiplicazione algebrica tra sx e dx
/	res=sx 7 dx	res = divisione algebrica di sx con dx
%	res=sx % dx	res = resto della divisione tra sx e dx

Esaminiamo ora le seguenti righe di codice:

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

Nasce il problema di rappresentare correttamente la variabile *res* affinché si possa assegnarle il risultato dell'operazione. Essendo 1.500.000.000.000 troppo grande perché sia assegnato ad una variabile di tipo int, sarà necessario utilizzarne una di un tipo in grado di contenere correttamente il valore prodotto. Il codice funzionerà perfettamente se riscritto nel modo seguente:

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

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

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

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

1. *+op : trasforma l'operando op in un tipo int se è dichiarato di tipo char, byte o short;*

2. *-op : restituisce la negazione aritmetica di op.*

Non resta che parlare degli operatori aritmetici di tipo *shortcut* (scorciatoia). Questo tipo di operatori consente l'incremento o il decremento di uno come mostrato nella tabella. A seconda della loro posizione producono un diverso risultato:

Operatori shortcut

operatore	forma estesa	risultato

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

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

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

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

In generale ++i, i++ --i, i-- incrementano o decrementano la variabile i di 1 ma in modi diversi a seconda della posizione. In particolare diremo che:

1. *Post incremento (i++): usiamo i++ se vogliamo usare il valore corrente di i e quindi incrementarlo di 1;*
2. *Pre incremento (++i): usiamo ++i se vogliamo incrementare i di 1 prima di utilizzare il valore di i;*
3. *Post decremento (i--): usiamo i-- se vogliamo usare il valore corrente di i e quindi decrementarlo di 1;*
4. *Pre decremento (--i): usiamo --i se vogliamo decrementare i di 1 e prima di utilizzare il valore di i;*

Operatori relazionali

Gli *operatori relazionali* sono detti tali perché si riferiscono alle possibili relazioni tra valori, producendo un risultato di verità o falsità come conseguenza del confronto.

A differenza dei linguaggi C e C++ in cui vero o falso corrispondono rispettivamente con i valori 0 e ≠0 restituiti da un'espressione, Java li identifica rispettivamente con i valori *true* e *false*, detti booleani e rappresentati da variabili di tipo **boolean**.

Nella tabella seguente sono riassunti gli operatori relazionali ed il loro significato.

operatore	sintassi	operatori shortcut descrizione
>	res=sx > dx	res = true se sx è maggiore di dx.
>=	res=sx >= dx	res = true se sx è maggiore o uguale di dx.
<	res=sx < dx	res = true se sx è minore di dx.
<=	res=sx <= dx	res = true se sx è minore o uguale di dx.
!=	res=sx != dx	res = true se sx è diverso da dx.
==	res=sx == dx	res = true se sx è uguale a dx.

Gli operatori relazionali < , >, <=, >= hanno tutti lo stesso livello di precedenza e associano da sinistra verso destra. Anche gli operatori di uguaglianza (== , !=) si associano da sinistra verso destra, hanno lo stesso livello di precedenza che tuttavia è più basso di quelli relazionali.

Operatori logici

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

operatore	sintassi	Operatori logici descrizione
&&	res=sx && dx	AND : res = true se sx e dx vagono entrambi true, false altrimenti.
||	res=sx || dx	OR : res = true se almeno uno tra sx e dx vale true, false altrimenti.
!	res = !sx	NOT : res = true se sx vale false, false altrimenti.
^	res= sx ^ dx	XOR : res = true se uno solo dei due operandi vale true, false altrimenti.

Gli operatori logici hanno tutti lo stesso livello di precedenza e associano da sinistra verso destra; sono tutti operatori binari a parte l'operatore ! (not), unario, che ritorna il complemento (negazione logica) dell'operando.

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

AND - &&		
sx	**dx**	**valore**
true	true	true
true	false	false
false	true	false
false	false	false

OR - ||		
sx	**dx**	**valore**
true	true	true
true	false	true
false	true	true
false	false	false

NOT - !	
sx	**valore**
true	false
false	true

XOR - ^		
sx	**dx**	**valore**
true	true	false
true	false	true
false	true	true
false	false	false

Ricapitolando:

1. L'operatore "&&" è un operatore binario e restituisce vero se entrambi gli operandi sono veri;

2. L'operatore "||" è un operatore binario e restituisce vero se almeno uno dei due operandi è vero;

3. L'operatore "!" è un operatore unario che afferma la negazione dell'operando;

4. L'operatore "^" è un operatore binario e restituisce vero se solo uno dei due operandi è vero;

Come vedremo in seguito, gli operatori relazionali, agendo assieme agli operatori logici, forniscono uno strumento di programmazione molto efficace in tutte le situazioni in cui il programma deve prendere delle decisioni.

Operatori logici e di shift bit a bit

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

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

Operatori di shift bia a bit		
operatore	**sintassi**	**descrizione**

Consideriamo quindi il seguente esempio:

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

L'esecuzione della applicazione produrrà quanto segue:

```
Il risultato di 100 >> 1 e': 50
Il risultato di 50 >> 1 e': 25
Il risultato di 25 >> 1 e': 12
Il risultato di 100 << 1 e': 200
Il risultato di 200 << 1 e': 400
Il risultato di 400 << 1 e': 800
```

Poiché la rappresentazione binaria del numero decimale 100 è 01100100, lo spostamento dei bit verso destra di una posizione produrrà come risultato il numero binario 00110010 che corrisponde al valore 50 decimale; viceversa, lo spostamento dei bit verso sinistra di una posizione, produrrà come risultato il numero binario 11001000 che corrisponde al valore 200 decimale. Appare evidente che le operazioni di shift verso destra o verso sinistra di 1 posizione

dei bit di un numero intero corrispondono rispettivamente alla divisione o moltiplicazione di un numero intero per 2.

Ciò che rende particolari questi operatori è la velocità con cui sono eseguiti rispetto alle normali operazioni di prodotto o divisione: di conseguenza, questa caratteristica li rende particolarmente appetibili per sviluppare applicazioni che necessitano di fare migliaia di queste operazioni in tempo reale.

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

Operatori logici bit a bit		
operatore	**sintassi**	**descrizione**
&	res=sx & dx	AND bit a bit
|	res=sx | dx	OR bit bit
~	res = ~sx	COMPLEMENTO A UNO bit a bit
^	res= sx ^ dx	XOR bit a bit

AND bit a bit		
sx	**dx**	**valore**
1	1	1
1	0	0
0	1	0
0	0	0

OR bit bit		
sx	**dx**	**valore**
1	1	1
1	0	1
0	1	1
0	0	0

COMPLEMENTO A UNO bit a	
sx	**valore**
1	0
0	1

XOR bit a bit		
sx	**dx**	**valore**
1	1	0
1	0	1
0	1	1
0	0	0

Nelle tabelle precedenti sono riportati tutti i possibi egli operatori nella tabella precedente. Tutte le combinaz un singolo bit degli operandi. Il prossimo è un esempio d bit a variabili di tipo **long**:

```
long sinistro = 100;
long destro = 125;
long risultato = sinistro & destro;
System.out.println("100 & 125 = " + risultato);
risultato = sinistro | destro;
System.out.println("100 | 125 = " + risultato);
risultato = sinistro ^ destro;
System.out.println("100 ^ 125 = " + risultato);
risultato = ~sinistro;
System.out.println("~ 125 = " + risultato);
```

Il risultato della esecuzione della applicazione sarà il seguente:

```
100 & 125 = 100
100 | 125 = 125
100 ^ 125 = 25
```

~125 = -101

Da momento che 100 in binario equivale a 01100100 e 125 a 01111101 allora:

01100100 & 01111101 = 01100100 (100)
01100100 & 01111101 = 01111101 (125)
01100100 ^ 01111101 = 00011001 (25)
~01100100 = 10011011 (155)

Nella prossima applicazione, utilizziamo gli operatori logici bit a bit per convertire un carattere minuscolo nel relativo carattere maiuscolo. I caratteri da convertire vengono trasmessi alla applicazione attraverso la riga di comando della Java Virtual Machine.

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

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

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

Per effettuare la conversione dei carattere nel relativo carattere minuscolo, abbiamo utilizzato l'operatore & per mettere a zero il sesto bit della variabile di tipo int, minuscolo , che contiene il codice UNICODE del carattere che vogliamo convertire.

Il carattere 'a' è rappresentato dal codice unicode 97, 'A' dal codice unicode 65 quindi, per convertire il carattere minuscolo nel rispettivo maiuscolo, è necessario sottrarre 32 al codice UNICODE del primo. La stessa regola vale per tutti i caratteri dell'alfabeto anglosassone:

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

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

Array

Java, come il C, consente l'aggregazione dei tipi base e degli oggetti mettendo a disposizione del programmatore gli array. In altre parole, mediante gli array è possibile creare collezioni di entità dette *tipi base dell'array*: i tipi base di un array possono essere oggetti o tipi primitivi ed il loro numero è chiamato *length* o lunghezza dell'array.

La sintassi per la dichiarazione di una variabile di tipo array è espressa dalla regola seguente:

tipo[] identificatore;

o, per analogia con il linguaggio C,

tipo identificatore[];

dove *tipo* rappresenta un tipo primitivo od un oggetto, ed *identificatore* è il nome che utilizzeremo per far riferimento ai dati contenuti all'interno dell'array.

Nel prossimo esempio, viene dichiarato un array il cui tipo base è rappresentato da un intero utilizzando entrambe le forme sintattiche riconosciute dal linguaggio:

```
int[] elencodiNumeriInteri;
int elencodiNumeriInteri[];
```

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

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

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

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

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

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

Lavorare con gli array

Creare un array vuole dire aver creato un contenitore vuoto, in grado di accogliere un numero massimo di elementi, definito dalla lunghezza, tutti della stessa tipologia definita dal tipo base dell'array *(Immagine 16 Array in jav*). Di fatto, creando un array abbiamo chiesto alla JVM di riservare spazio di memoria sufficiente a contenerne tutti gli elementi.

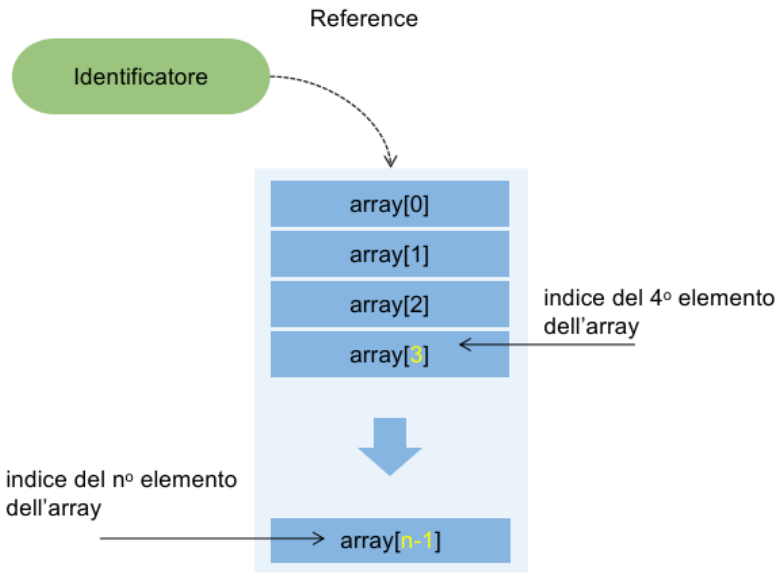


Immagine 16 Array in java

In java, come in C e C++, primo elemento in un array parte dalla posizione 0 e non 1, di conseguenza un array di n elementi sarà accessibile tramite l'operatore indice [0],[2],[3],..[n-1]. E' possibile ottenere la lunghezza dell'array tramite la proprietà *length*.

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

```
public class MaiuscoloMinuscolo {
  public static void main(String[] args) {
    // Dichiarazione e creazione di un array di caratteri
    // contenente i primi dieci caratteri minuscoli dell'alafabeto italiano
    char[] minuscolo = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l' };
    // Dichiarazione e creazione dell'array di caratteri
    // di lunghezza 10 che conterrà i caratteri maiscoli dopo
    // la copnversione
    char[] maiuscolo = new char[10];
    // Converto i caratteri in maiuscolo e li inserisco nel nuovo array
    // utilizzando due variabili int che conterranno il codice UNICODE dei
    // due caratteri
    int carattereMinuscolo;
    int carattereMaiuscolo;
    for (int i = 0; i < 10; i++) {
      carattereMinuscolo = minuscolo[i];
      // Eseguo la conversione in maiuscolo
      carattereMaiuscolo = carattereMinuscolo & 223;
      // Memorizzo il risultato nel nuovo array
      maiuscolo[i] = (char) carattereMaiuscolo;
      System.out.println("minuscolo[" + i + "]='" + minuscolo[i] + "', maiuscolo [" + i + "]='" +
          maiuscolo[i] + "'");
```

```
    }
  }

}
```

L'esecuzione del codice produrrà il seguente output:

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

Dall'esempio si nota che, come per il linguaggio C, Java accetta la notazione tra parentesi graffe per poter creare ed inizializzate l'array nello stesso momento.

```
char[] minuscolo = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l' };
```

In generale la sintassi è la seguente:

```
tipo identificatore[] = {valori separati da virgola};
```

Array multidimensionali

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

```
tipo[][] identificatore;
```

in cui *tipo* ed *identificatore* rappresentano rispettivamente il tipo base dell'array ed il nome che ci consentirà di accedere ai dati in esso contenuti.

Nel prossimo esempio in cui utilizzeremo un array bidimensionale per rappresentare la tavola per il gioco della dama.

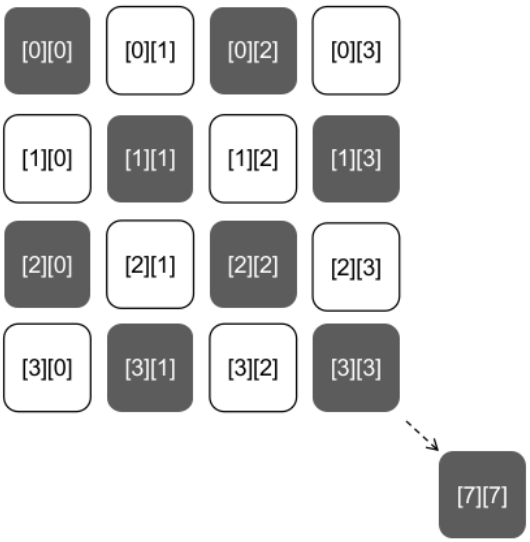


Immagine 17 tavola per il gioco della dama

Supponiamo che le nostre pedine, bianche o nere, siano rappresentate rispettivamente dalle stringhe *bianca*, *nera*. Nel prossimo esempio creiamo la nostra tavola per la dama e la inizializziamo il tavolo di gioco.

```
public class GiocoDellaDamaVersione1 {

  public static void main(String[] args) {

    final String BIANCA = "bianca";
    final String NERA = "nera";
    String dama[][] = new String[8][8];

    // Riga 1
    dama[0][1] = BIANCA;
    dama[0][3] = BIANCA;
    dama[0][5] = BIANCA;
    dama[0][7] = BIANCA;

    // Riga 2
    dama[1][0] = BIANCA;
    dama[1][2] = BIANCA;
    dama[1][3] = BIANCA;
    dama[1][6] = BIANCA;

    // Riga 3
    dama[2][1] = BIANCA;
    dama[2][3] = BIANCA;
    dama[2][5] = BIANCA;
    dama[2][7] = BIANCA;
```

```
    // Riga 6
    dama[5][0] = NERA;
    dama[5][2] = NERA;
    dama[5][3] = NERA;
    dama[5][6] = NERA;

    // Riga 7
    dama[6][1] = BIANCA;
    dama[6][3] = BIANCA;
    dama[6][5] = BIANCA;
    dama[6][7] = BIANCA;

    // Riga 8
    dama[7][0] = NERA;
    dama[7][2] = NERA;
    dama[7][3] = NERA;
    dama[7][6] = NERA;

  }
}
```

Nell'esempio abbiamo utilizzato l'operatore indice *[riga][colonna]* per inserire all'interno dell'array le pedine bianche e nere secondo la disposizione classica prevista per il gioco. Nella realtà, Java organizza gli array multidimensionali come *array di array*. Per questo motivo, non è necessario specificarne la lunghezza per ogni dimensione dichiarata, al momento della creazione: in altre parole, un array multidimensionale non deve essere necessariamente creato utilizzando una singola operazione **new**. Lo vediamo nel prossimo esempio in cui utilizzeremo un array di array per ottenere un risultato equivalente al precedente.

```
public class GiocoDellaDamaVersione2 {
  public static void main(String[] args) {

    final String BIANCA = "bianca";
    final String NERA = "nera";
    String dama[][] = new String[8][8];

    String[] riga1 = {
        BIANCA, null, BIANCA, null,  BIANCA, null, BIANCA, null };
    String[] riga2 = { null, BIANCA, null, BIANCA, null, BIANCA, null, BIANCA};
    String[] riga3 = { BIANCA, null, BIANCA, null,  BIANCA, null, BIANCA, null  };

    String[] riga6 = {  null, NERA, null, NERA, null, NERA, null, NERA };
    String[] riga7 = { NERA, null, NERA, null, NERA, null, NERA, null };
    String[] riga8 = {  null, NERA, null, NERA,  null, NERA, null, NERA };

    dama[0] = riga1;
    dama[1] = riga2;
```

```
        dama[2] = riga3;
        dama[5] = riga6;
        dama[6] = riga7;
        dama[7] = riga8;
    }
}
```

Utilizzare l'una o l'altra strada per creare la nostra tavola della dama è assolutamente indifferente, e i due esempi, sono assolutamente equivalenti. Sta quindi al programmatore scegliere come gestire al meglio situazioni analoghe che si possono presentare.

5. Istruzioni e controllo di flusso

Introduzione

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

Questo capitolo tratta le istruzioni condizionali, le istruzioni cicliche, quelle relative alla gestione dei package per l'organizzazione di classi e l'istruzione **import** per risolvere il problema della *posizione* delle definizioni di classi in altri file o package.

Istruzioni per il controllo di flusso

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

Le istruzioni per il controllo di flusso sono 6, e sono elencate a seguire. Hanno la sintassi definita dalle regole di espansione 2 e 3 viste nel paragrafo precedente.

Operatori per il controllo di flusso	
operatore	**descrizione**
if	Esegue o no un blocco di codice a seconda del valore restituito da una espressione booleana.
if-else	Determina quale tra due blocchi di codice sia quello da eseguire, a seconda del valore restituito da una espressione booleana.
switch	Utile in tutti quei casi in cui sia necessario decidere tra opzioni multiple prese in base al controllo di una sola variabile.
for	Esegue ripetutamente un blocco di codice.
for-in o for-each	Un metodo alternativo per l'attraversamento degli elementi di un array o, più in generale, di una collezione.
while	Esegue ripetutamente un blocco di codice controllando il valore di una espressione booleana prima della esecuzione del blocco.
do-while	Esegue ripetutamente un blocco di codice controllando il valore di una espressione booleana solo al termine della esecuzione del blocco.

Istruzione if

L'istruzione per il controllo di flusso **if** consente alla applicazione di decidere, in base ad una espressione booleana, se eseguire un blocco di codice.

Applicando le regole di espansione definite, la sintassi di questa istruzione è la seguente:

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

                                           istruzione;
                                           [istruzione]
```

Dove *condizione* rappresenta un'istruzione booleana valida. Di fatto, se l'espressione restituisce il valore *true* , sarà eseguito il blocco di istruzioni immediatamente successivo, in caso contrario il controllo passerà alla prima istruzione successiva al blocco **if**.

Un esempio di istruzione **if** è il seguente:

```
int x;
if(x>10){
        System.out.println("x è maggiore di 10");
        x=0;
}
x=1;
```

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

Istruzione if-else

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

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

                                           istruzione;
                                              [istruzione]
```

Se condizione restituisce il valore *true*, sarà eseguito il blocco di istruzioni di **if**, altrimenti il controllo sarà passato ad **else** e sarà eseguito il secondo blocco di istruzioni. Al termine, il controllo di flusso passa alla *istruzione3*. Il blocco **else** rappresenta quindi un alternativa al blocco **if** da eseguire solo se la condizione del blocco **if** non è verificata. Il prossimo esempio estende l'esempio precedente: se *x* sarà maggiore di 10 verrà eseguito il blocco **if** altrimenti il blocco **else**.

```
int x;
if(x>10){
        System.out.println("x è maggiore di 10");
        x=0;
}else{
        System.out.println("x è minore o uguale a 10");
}
x=1;
```

Istruzioni if, if-else annidate

Un'istruzione **if** *annidata*, rappresenta una forma particolare di controllo di flusso in cui un'istruzione **if** oppure **if-else** è controllata da un'altra istruzione **if** oppure **if-else** .

Utilizzando le regole di espansione, in particolare intrecciando ricorsivamente la terza regola con le definizioni di **if** e **if-else**, otteniamo la forma sintattica:

istruzione -> *controllo_di_flusso*
controllo_di_flusso
istruzione

controllo_di_flusso ->***if****(condizione)*
istruzione

oppure, nel caso di **if-else**

controllo_di_flusso ->***if****(condizione)*
istruzione

else
istruzione

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

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

```
}

istruzione4
```

Supponiamo ora di voler esprimere un giudizio sul voto di un esame universitario in base alla seguente classificazione:

1. se il voto è compreso tra 18 e 23 allora il giudizio sarà sufficiente;

2. se il voto è compreso tra 24 e 30 allora il giudizio sarà buono

Nel prossimo esempio una possibile soluzione utilizzando blocchi if con if-else annidati:

```
public class IfElseAnnidati {
  public static void main(String[] args) {
    int voto = Integer.parseInt(args[0]);
    if (voto >= 18 && voto <= 30)
      if (voto >= 24)
        System.out.println("Il risultato e' buono");
      else
        System.out.println("Il risultato e' sufficiente");
  }
}
```

Catene if-else-if

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

Usando lo stesso schema sintattico giù utilizzato in precedenza, otteniamo una possibile catena **if-else-if**:

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

istruzione5
```

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

```
public class InteroComeStringa {
  public static void main(String[] argv) {
    int intero = Integer.parseInt(argv[0]);
    if (intero == 0) {
      System.out.println("Zero");
    } else if (intero == 1) {
      System.out.println("Uno");
    } else if (intero == 2) {
      System.out.println("Due");
    } else if (intero == 3) {
      System.out.println("Tre");
    } else if (intero == 4) {
      System.out.println("Quattro");
    } else if (intero == 5) {
      System.out.println("Cinque");
    } else if (intero == 6) {
      System.out.println("Sei");
    } else if (intero == 7) {
      System.out.println("Sette");
    } else if (intero == 8) {
      System.out.println("Otto");
    } else if (intero == 9) {
      System.out.println("Nove");
    } else if (intero == 10) {
      System.out.println("Dieci");
    } else {
      System.out.println("Impossibile stanpare un valore maggiore di dieci");
    }
  }

}
```

Istruzione switch

L'esempio del paragrafo precedente rende evidente quanto sia complesso scrivere o leggere codice java utilizzando catene **if-else-if** arbitrariamente complesse. Per far fronte al problema, Java fornisce al programmatore un'istruzione di controllo di flusso che, specializzando la catena **if-else-if** rende più semplice la programmazione di un'applicazione.

L'istruzione **switch** è utile in tutti quei casi in cui sia necessario decidere tra scelte multiple, prese in base al controllo di una sola variabile. La sintassi dell'istruzione è la seguente:

```
switch (espressione){
case espressione_costante:
        istruzione
        [break]

case espressione_costante:
        istruzione
        [break]

.....

case espressione_costante:
        istruzione
        [break]

default:
        istruzione

}
istruzione
```

Dove: *espressione* rappresenta ogni espressione valida che produca un intero ed *espressione_costante* un'espressione che può essere valutata completamente al momento della compilazione. Quest'ultima, per funzionamento, può essere paragonata ad una costante. *Istruzione* è ogni istruzione Java come specificato dalle regole di espansione e [break] rappresenta l'inclusione opzionale della parola chiave **break** seguita da ";" .

In generale, dopo la valutazione di espressione, il controllo dell'applicazione salta al primo blocco **case** tale che *espressione == espressione_costante* ed esegue il relativo blocco di codice. Nel caso in cui il blocco termini con una istruzione **break**, l'applicazione abbandona l'esecuzione del blocco **switch** saltando alla prima istruzione successiva al blocco, altrimenti il controllo viene eseguito sui blocchi **case** successivi. Se nessun blocco *case* soddisfa la condizione, ossia *espressione != espressione_costante* la virtual machine controlla l'esistenza della label **default** ed esegue, se presente, solo il blocco di codice relativo ed esce da **switch**.

Nonostante la logica voglia che il metodo **switch** esegua uno ed un solo blocco **case**, in realtà **break** è necessario per uscire dal blocco **switch** appena il blocco **case** è stato eseguito. Se omettiamo **break**, i blocchi a seguire (specialmente il blocco default) saranno eseguiti.

L'unico blocco che non richiede un **break** è l'ultimo: il blocco **default**. Tuttavia è uso comune utilizzarlo comunque per rendere il codice più leggibile e meno soggetto ad interpretazioni errate o ad errori.

L’esempio del paragrafo precedente può essere riscritto nel modo seguente:

```
public class InteroComeStringa2 {
  public static void main(String[] argv) {
    int intero = Integer.parseInt(argv[0]);
    switch (intero) {
      case 0:
        System.out.println("Zero");
        break;
      case 1:
        System.out.println("Uno");
        break;
      case 2:
        System.out.println("Due");
        break;
      case 3:
        System.out.println("Tre");
        break;
      case 4:
        System.out.println("Quattro");
        break;
      case 5:
        System.out.println("Cinque");
        break;
      case 6:
        System.out.println("Sei");
        break;
      case 7:
        System.out.println("Sette");
        break;
      case 8:
        System.out.println("Otto");
        break;
      case 9:
        System.out.println("Nove");
        break;
      case 10:
        System.out.println("Dieci");
        break;
      default:
        System.out.println("Impossibile stanpare un valore maggiore di dieci");
break;
    }
  }
```

```
}
```

L'istruzione **switch** non è in grado di operare con tutti i possibili valori oppure oggetti, ma deve tornare valori compatibili con la lista a seguire:

1. *tipo primitivo **byte** (oppure il suo wrapper Byte);*
2. *tipo primitivo **short** (oppure il suo wrapper Short);*
3. *tipo primitivo **int** (oppure il suo wrapper Integer);*
4. *tipo primitivo **char** (oppure il suo wrapper Character);*
5. ***enum** (enumerazioni)*
6. *String (stringhe)*

L'istruzione **switch** non accetta *null* come valore. Nel caso, al momento della valutazione dell'espressione sarà generato un errore (*NullPointerException*).

Le variabili dichiarate all'interno del blocco **switch**, come da definizione di scope di una variabile, esistono e sono utilizzabile fino a che stiamo eseguendo il blocco stesso. Se vogliamo limitare lo scope delle variabili al blocco case, allora possiamo utilizzare le parentesi graffe per delimitarne la visibilità.

```
switch (codiceErrore) {
  case 101: {
    // questa variabile esiste solo in questo blocco
    int num = 200;
    break;
  }
  case 300: {
    // Questo codice viene compilato correttamente
    int num = 300;
    break;
}
}
```

Valgono in generale le regole definite per lo scope delle variabili java.

Poiché in java tutto è un oggetto, a partire da Java 5 sono state create le classi *wrapper*. Una classe wrapper è come un involucro (wrap) che ha lo scopo di contenere un valore primitivo, trasformandolo in un oggetto.

Torniamo per un attimo a considerare il blocco case ed in particolar modo poniamo l'attenzione alla *espressione_costante*. Come abbiamo detto, una espressione costante è una espressione il cui valore può essere valutato completamente al momento della compilazione. Consideriamo per un attimo il prossimo blocco di codice:

```
final String cane="CANE";
String gatto="GATTO";
switch (animale) {
        case cane: //compila correttamente
                        result = "animale domestico;
        case gatto: //non compila
                        result = "felino"
}
```

Il codice produrrà un errore in fase di compilazione fino a che non trasformeremo gatto in una costante utilizzando il modificatore **final**.

```
final String cane="CANE";
final  String gatto="GATTO";
switch (animale) {
        case cane: //compila correttamente
                        result = "animale domestico;
        case gatto: //compila correttamente
                        result = "felino"
}
```

Evoluzione dell'istruzione switch. Espressione switch

Quello che apprezzo del linguaggio Java è la sua capacità di evolvere nel tempo abbracciando le reali necessità dei programmatori. A partire da Java 12, definitivamente con Java 17, l'istruzione **switch** ha subito una serie di modifiche per renderla più efficace e vicina alle necessità della moderna programmazione. Prima di entrare nei dettagli, torniamo ancora per un attimo a considerare l'istruzione **switch** come la conosciamo.

```
switch (espressione) {
  case 1 :
    System.out.println("primo case");
    break;
  case 2 :
    System.out.println("secondo case");
    break;
  case 3 :
    System.out.println("terzo case");
```

```
    break;
  default :
    System.out.println("blocco di default");
    break;
}
```

Il codice fa esattamente quanto richiesto: valuta l'espressione, esegue il corrispondente blocco case, oppure il blocco di default e poi esce. Ma cosa succede se dimentichiamo un **break**?

```
switch (espressione) {
  case 1 :
    System.out.println("primo case");
    // break manca qui
  case 2 :
    System.out.println("secondo case");
    break;
  case 3 :
    System.out.println("terzo case");
    break;
  default :
    System.out.println("blocco di default");
    break;
}
```

Se passiamo come valore dell'espressione 1 il risultato sarà, chiaramente, quello errato causato da una banale dimenticanza di un **break**.

```
primo case
secondo case
```

Ci sono poi altri casi in cui per necessità di programmazione farebbe comodo poter dichiarare blocchi case utilizzando valori multipli come nel prossimo esempio in cui i primi due blocchi case sono assolutamente identici e quindi sarebbe logico raggrupparli in un unico blocco.

```
switch (razza) {
  case "jack russel":
    System.out.println("Questo è un cane");
    break;
  case "boxer" :
    System.out.println("Questo è un cane");
    break;
  case "soriano" :
    System.out.println("Questo è un gatto");
    break;
  default :
    System.out.println("questo non lo conosco");
```

```
    break;
}
```

Un blocco case può supportare espressioni costanti multiple

Questa è una prima importante modifica: potendo utilizzare costanti multiple, il codice dell'esempio precedente potrà essere riscritto in maniera più leggibile e compatta:

```
switch (razza) {
  case "jack russel", "boxer":
    System.out.println("Questo è un cane");
    break;
  case "soriano" :
    System.out.println("Questo è un gatto");
    break;
  default :
    System.out.println("questo non lo conosco");
    break;
}
```

*I blocchi **case** possono utilizzare l'operatore **instanceof** per fare il confronto di tipi. In questo caso il blocco **case** sarà eseguito solo se il tipo dell'espressione è uguale al tipo definito nel blocco **case**.*

A partire da Java 17, viene introdotta una nuova caratteristica chiamata *pattern matching* che consente di passare un oggetto come condizione per **switch** (prima di Java 17 non era possibile usare oggetti invece di espressioni), per poi fare il confronto di tipo (*pattern matching*) come etichette per i blocchi **case**.

```
switch (o) {
  case Integer i -> String.format("int %d", i);
  case Long l    -> String.format("long %d", l);
  case Double d  -> String.format("double %f", d);
  case String s  -> String.format("String %s", s);
  default        -> o.toString();
};
```

Nell'esempio abbiamo passato un oggetto come condizione per **switch** e abbiamo condizionato l'esecuzione dei blocchi **case** sulla base del tipo dell'oggetto. Non solo. Consideriamo ad esempio la riga di codice:

```
case Integer i -> String.format("int %d", i);
```

L'oggetto passato al blocco **switch non solo** viene controllato se di tipo *Integer*, ma in caso positivo viene assegnato alla variabile reference i di tipo *Integer*. Utilizzando l'operatore **instanceof** la stessa riga di codice dovrebbe essere riscritta come segue:

```
if(o instanceof Integer){
        Integer i = (Integer) o;
        String.format("int %d", i);
}
```

*I blocchi **case** accettano anche espressioni booleane chiamate guarded patterns. I guarded patterns sono condizioni che devono essere verificate come parte di una espressione. Sono precedute da &&*

Questa è forse una delle più importanti migliorie apportate al linguaggio java.

Supponiamo di voler verificare che un oggetto passato per parametro al blocco **switch** sia una stringa, e che vogliamo inserire nel blocco **case** un controllo aggiuntivo sulla lunghezza della stringa. Con i *guarded patterns* possiamo farlo nel modo seguente:

```
switch(value) {
 case String s && (s.length > 3) -> System.out.println("A short string");
 case String s && (s.length > 10 -> System.out.println("A medium string");
 default -> System.out.println("A long string");
}
```

***yeld** può essere utilizzato per tornare un valore*

L'istruzione switch adesso può essere utilizzata come una espressione in gradi di tornare un valore a seconda dell'input. La sintassi cambia leggermente, e poiché una espressione può tornare un solo valore, **yeld** sostituisce completamente **break**.

```
String genere = switch (razza) {
  case "jack russel", boxer":
    System.out.println("Questo è un cane");
   yeld "cane";
  case "soriano" :
    System.out.println("Questo è un gatto");
    yeld "gatto";
  default :
    throw new IllegalArgumentException("Il genere è sconosciuto");
}
```

E' importante ricordare che **switch**, se utilizzato come espressione, deve gestire tutti i possibili valori di input ovvero deve sempre poter tornare un valore. Come conseguenza valgono le seguenti regole:

*1. Una espressione **switch** deve tornare sempre un valore oppure una eccezione.*

Ne consegue che il prossimo esempio produrrà un errore in fase di compilazione:

```
String genere = switch (razza) {
  case "jack russel", boxer":
    System.out.println("Questo è un cane");
   yeld "cane";
  case "soriano" :
    System.out.println("Questo è un gatto");
    yeld "gatto";
}
```

2. Come conseguenza della distinzione tra istruzione ed espressione switch, utilizzare return (che analizzeremo in seguito) all'interno di un blocco case è consentito solo in una istruzione switch classica .

La sintassi di switch è stata aggiornata con l'aggiunta della notazione a freccia: -> . La notazione a freccia è valida sia nel caso di istruzione che nel caso di espressione switch.

In un blocco case, le istruzioni sul lato destro di -> vengono eseguite solo se il valore dell'espressione è uguale al suo lato sinistro. Le seguenti porzioni di codice sono tutte valide.

```
String genere = switch (razza) {
  case "jack russel", boxer" -> "cane";
  case "soriano" -> "gatto";
}
....
....

switch (razza) {
  case "jack russel","boxer" ->  System.out.println("Questo è un cane");
  case "soriano" -> {
        System.out.println("Questo è un gatto");
    }
  default -> System.out.println("questo non lo conosco");
}
```

Il vantaggio principale nell'utilizzare questa notazione è che, come si vede dall'esempio precedente, non abbiamo bisogno di usare **break** per uscire dal blocco **switch**: non appena sarà

eseguito un blocco case, l'espressione **switch** eseguirà il codice sulla destra ed uscirà immediatamente ritornando. Valgono in questo caso le seguenti regole e convenzioni:

*1. Se non abbiamo necessità di utilizzare opportunisticamente il meccanismo a cascata di **switch** ed eseguire comunque il metodo di **default** usiamo case -> altrimenti **case:***

2. All'interno di un blocco switch non possiamo usare alternatamente case -> oppure case: . L'uno escludo l'utilizzo dell'altro.

Come abbiamo anticipato, l'istruzione **switch** classica non consentiva l'utilizzo di espressioni

E' consentito usare null all'interno di una espressione

null. Java 17 ne consente l'utilizzo sia con le istruzioni che con le espressioni switch.

```
switch (razza) {
        case null -> System.out.println("OPS! Abbiamo una espressione null");
          case "jack russel", boxer" ->  System.out.println("Questo è un cane");
          case "soriano" -> {
                System.out.println("Questo è un gatto");
                }
          default -> System.out.println("questo non lo conosco");
}
```

Istruzione while

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

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

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

In dettaglio, un'istruzione **while** controlla il valore dell'espressione booleana: se il risultato restituito è *true* sarà eseguito il blocco di codice di **while**. Alla fine dell'esecuzione è nuovamente controllato il valore dell'espressione booleana, per decidere se ripetere l'esecuzione del blocco di codice o passare il controllo dell'esecuzione alla prima istruzione successiva al blocco **while**.

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

```
while (espressione)
        while (espressione)
                istruzione
```

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

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

Istruzione do-while

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

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

A differenza dell'istruzione **while**, ora il blocco di istruzioni sarà eseguito sicuramente almeno una volta.

```
public class DoWhile {
        public static void main(String[] args) {
          int i=1;
          do{
                  System.out.println("----- "+i);
                  i++;
          }while(i<=5);
        }
}
```

Il ciclo **do-while** è anche conosciuto come *exit controlled loop*. Un *exit controlled loop* è una categoria di loop in cui la condizione di test è controllata solo dopo l'esecuzione del blocco di codice. Pertanto, in un *exit controlled loop* il blocco di codice è eseguito almeno una volta anche quando la condizione di test fallisce subito.

Istruzione for

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

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

Nell'esempio, come prima operazione l'applicazione inizializza una variabile (*inizializzazione*) per il controllo del ciclo, quindi viene eseguita un'espressione condizionale (*condizione*) per decidere se eseguire o no il blocco di istruzioni dell'istruzione **while**, infine la variabile è aggiornata (*espressione*) in modo tale che possa determinare la fine del ciclo.

Tutte queste operazioni sono incluse nella stessa istruzione condizionale dell'istruzione **for**:

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

che, nella sua forma annidata assume la seguente forma:

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

      }

}
istruzione2
```

Quindi:

1. inizializzazione rappresenta la definizione della variabile per il controllo del ciclo. Possiamo quindi creare ed inizializzare una variabile oppure utilizzare una variabile già inizializzata;

2. condizione rappresenta l'espressione condizionale che controlla l'esecuzione del blocco di istruzioni. Deve tornare un valore booleano ed è eseguita ad ogni ciclo che terminerà nel momento in cui la condizione tornerà false;

3. espressione contiene le regole per l'aggiornamento, l'incremento o il decremento della variabile di controllo.

In una istruzione **for** la condizione viene sempre controllata all'inizio del ciclo. Nel caso in cui restituisca un valore *false* il blocco di istruzioni non verrà mai eseguito; nel caso restituisca il valore *true* il blocco di istruzioni viene invece eseguito. Successivamente viene aggiornato il valore della variabile di controllo e infine viene nuovamente valutata la condizione.

Il ciclo realizzato nel precedente esempio utilizzando il comando **while**, può essere riscritto utilizzando il comando **for** nel seguente modo:

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

L'istruzione **for**, in Java come in C e C++, è un'istruzione molto versatile poiché consente di scrivere cicli di esecuzione utilizzando molte varianti alla forma descritta. In pratica, il ciclo **for** consente di utilizzare zero o più variabili di controllo, zero o più istruzioni di assegnamento ed altrettanto vale per le espressioni booleane. Nella sua forma più semplice il ciclo **for** può essere utilizzato nella forma:

```
for( ; ; ){
    istruzione1

}
istruzione2
```

Questa forma realizza un ciclo infinito, non utilizza né variabili di controllo né istruzioni di assegnamento né tanto meno espressioni booleane. Anche le seguenti forme sono ammesse:

```
for( inizializzazione ; ;espressione ){
    istruzione1

}
istruzione2
```

```
for( inizializzazione ; condizione;){
    istruzione1

}
istruzione2
```

```
for( inizializzazione ; ;){
    istruzione1

}
istruzione2
```

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

Insomma, l'istruzione for è talmente adattabile che viene generalmente utilizzata in sostituzione di **while** e **do-while**.

Consideriamo ora il seguente esempio:

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

Il ciclo descritto utilizza due variabili di controllo con due operazioni di assegnamento distinte. Sia la dichiarazione ed inizializzazione delle variabili di controllo, che le operazioni di assegnamento utilizzando il carattere , *(virgola)* come separatore.

Istruzione for each (for in)

Consideriamo ora il prossimo blocco di codice:

```
public class CicloArrayConFor {
  public static void main(String[] args) {
    String mesiDellAnno[] = { "gennaio", "febbraio", "marzo",
        "aprile", "maggio", "giugno", "luglio", "agosto",
        "settembre", "ottobre", "novembre", "dicembre" };
    for(int i=0; i<mesiDellAnno.length; i++)
      System.out.println(mesiDellAnno[i]);
  }
}
```

Il codice implementa un esempio classico di ciclo su un array di stringhe. Ricordando quanto detto sugli array in java, una volta eseguito il codice stamperà a video tutti i mesi dell'anno.

```
gennaio
febbraio
.
.
.
novembre
dicembre
```

Il ciclo utilizza la proprietà *length* degli array all'interno della condizione per effettuare il controllo. Poiché abbiamo detto che in un array gli elementi sono indicizzati a partire da zero, il nostro ciclo terminerà quando la variabile di controllo *i* sarà minore di *array.length*.

Già a partire da Java 1.5 fu introdotta una forma alternativa al ciclo **for** utilizzabile per iterare su array e collezioni (qualcosa che approfondiremo nei capitoli successivi): l'istruzione **for-each** o anche nota come **for-in**.

La sintassi dell'istruzione **for-each** è la seguente:

```
for( tipo identificatore: <collezione|array>){
        istruzione1
}
```

utilizzando la nuova sintassi l'esempio precedente può quindi essere riscritto come segue:

```
public class CicloArrayConForEach {
  public static void main(String[] args) {
    String mesiDellAnno[] = { "gennaio", "febbraio", "marzo",
        "aprile", "maggio", "giugno", "luglio", "agosto",
        "settembre", "ottobre", "novembre", "dicembre" };
    for(String meseDellAnno : mesiDellAnno)
      System.out.println(meseDellAnno);
  }
}
```

La sintassi di **for-each** viene solitamente utilizzata in sostituzione dell'istruzione **for** standard tutte le volte che un contatore di ciclo non è realmente necessario, e ogni elemento di una raccolta deve essere elaborato. La sintassi inoltre è esternamente semplificata ed il codice risulta più leggibile. Piuttosto che dire:

fai questo per n volte fino a che

adesso potremmo leggerlo come:

per ogni elemento di tipo xxxx nell'array/collezione.

Esistono tuttavia alcune controindicazioni all'utilizzo dell'istruzione **for-each**, e ci sono cose che solo un semplice ciclo **for** ed un *iteratore*[5] possono fare. Durante un ciclo **for-each**:

1. *non possiamo rimuovere alcun elemento dalla raccolta o dall'array;*

2. *non possiamo modificare gli elementi della raccolta o dell'array;*

3. *non è possibile eseguire l'iterazione su più raccolte in parallelo.*

[5] In Java, un Iterator è un costrutto utilizzato per attraversare o scorrere la raccolta.

Quando possibile è consigliato utilizzare sempre il ciclo f**or-each** al posto del ciclo **for** classico: **for-each** non solo rende il codice più leggibile, ma in alcune situazioni offre prestazioni migliori.

Istruzioni di ramificazione

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

Istruzioni di ramificazione	
istruzione	**descrizione**
break	Interrompe l'esecuzione di un ciclo evitando ulteriori controlli sulla espressione condizionale, e ritorna il controllo alla istruzione successiva al blocco attuale.
continue	Salta un blocco di istruzioni all'interno di un ciclo e ritorna il controllo alla espressione booleana che ne governa l'esecuzione.
return	Indipendentemente dalla parte di codice in cui viene utilizzata, interrompe l'esecuzione del metodo attuale e ritorna il controllo al metodo chiamante.

Istruzione break

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

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

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

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

```
        break;
}
```

L'uso di quest'istruzione, tipicamente, è legato a casi in cui sia necessario poter terminare l'esecuzione di un ciclo a prescindere dai valori delle variabili di controllo utilizzate. Queste situazioni occorrono in quei casi in cui è impossibile utilizzare un parametro di ritorno come operando all'interno dell'espressione booleana che controlla l'esecuzione del ciclo, ed è pertanto necessario implementare all'interno del blocco meccanismi specializzati per la gestione dell'uscita controllata dal ciclo. Un esempio tipico è quello di chiamate a metodi che possono generare errori. Utilizzando il comando **break** è possibile interrompere l'esecuzione del ciclo non appena sia stato catturato l'errore.

L'istruzione **break** ha la seguente sintassi:

```
while|do-while|for|for-in|switch {
        istruzione
        break;
}
```

Istruzione continue

A differenza del caso precedente, l'istruzione **continue** non interrompe l'esecuzione del ciclo di istruzioni, ma al momento della chiamata produce un salto alla parentesi graffa che chiude il blocco restituendo il controllo all'espressione booleana che ne determina l'esecuzione.

Un esempio può aiutarci a chiarire le idee: il frammento di codice conta il numero delle occorrenze di numeri pari ci sono in una sequenza di interi compresa tra 1 e 20 memorizzando il risultato in una variabile di tipo **int** chiamata *pairs*.

```
int i=-1;
int pairs=0;
while(i<20){
        i++;
        if((i%2)!=0)
                continue; //esecuzione torna all'inizio del ciclo
        pairs ++;
}
```

Il ciclo **while** è controllato dal valore della variabile i inizializzata a –1. L'istruzione **if** effettua un controllo sul valore della variabile intera *i*: nel caso in cui i rappresenti un numero intero dispari, viene eseguito il comando **continue**, ed il flusso ritorna alla riga tre. In caso contrario viene aggiornato il valore di *pairs*.

Istruzione return

Return, rappresenta l'ultima istruzione di ramificazione ed è utilizzata per terminare l'esecuzione del metodo corrente, tornando il controllo al metodo chiamante. L'istruzione **return** può essere utilizzata in due forme:

return *espressione;*

return*;*

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

6. Package Java

Introduzione

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

I *package* rappresentano anche lo strumento messo a disposizione dal linguaggio per distribuire librerie di classi Java affinché possano essere riutilizzate all'interno di altre applicazioni; in questo caso sarà necessario utilizzare l'istruzione *import* per indicare quali classi utilizzare all'interno della nostra applicazioni.

In questo capitolo è mostrato in dettaglio meccanismo di raggruppamento e le caratteristiche. Tuttavia, altri aspetti saranno trattati nei capitoli successivi.

Package Java

I package Java organizzano le classi secondo una struttura gerarchica ed è generalmente utilizzato per raggruppare classi tra loro correlate. Immaginiamo ai package come una cartella (directory).

Di fatto, utilizzare i package comporta molti vantaggi:

1. Le classi possono essere mascherate all'interno dei package di appartenenza, facilitando l'incapsulamento anche a livello di file;

2. Le classi di un package possono condividere dati e metodi con classi di altri package;

3. I package forniscono un meccanismo efficace per distribuire oggetti;

4. I package consentono di evitare conflitti con i nomi delle classi;

5. Consentono di scrivere codice più leggibile e manutenibile;

6. I package utilizzati per la distribuzione di librerie java.

Sono divisi in due categorie:

DEFINIZIONE: Built-in packages ovvero quelli provenienti dalle core api java;

DEFINIZIONE: User-defined packages ovvero quelli creati dal programmatore o provenienti da librerie esterne.

Assegnare nomi ai packages

Nei capitoli precedenti abbiamo affermato che, la definizione della classe *MiaClasse* deve essere salvata nel file *MiaClasse.java* e ogni file con estensione *.java* deve contenere una sola classe.

Ogni definizione di classe è detta unità di compilazione.

I package combinano più unità di compilazione in un unico archivio, la cui struttura gerarchica rispetta quella del file system del computer.

Il nome completo di una unità di compilazione appartenente ad un package è determinato dal nome della classe, anteceduto dai nomi dei package appartenenti ad un ramo della gerarchia, separati tra loro dal carattere punto.

Ad esempio, se la classe CicloArrayConForEach appartiene alla gerarchia definita nella figura *Immagine 18 gerarchie di package*, il suo nome completo è:

javamattone.esercizi.capitolo5.CicloArrayConForEach

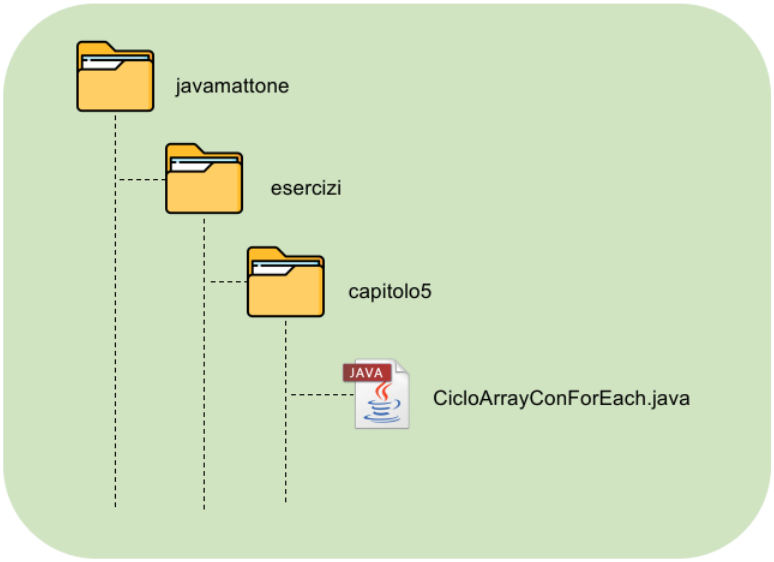


Immagine 18 gerarchie di packages

Mediante la variabile di ambiente *CLASSPATH* sarà possibile comunicare alla JVM come poter trovare il package definito inserendo il percorso fino alla cartella di primo livello nella gerarchia dei package.

Qualsiasi classe in java è identificata in maniera univoca dal suo nome completo ovvero il nome del package separato da punti seguito dal nome della classe (senza estensione .class alla fine).

Secondo le specifiche e le convenzioni, queste sono le regole per assegnare nomi ai packages:

1. I nomi dei package dovrebbero contenere solo caratteri minuscoli;

2. Tutti i package che iniziano con "java." Contengono le classi appartenenti alle Java Core API (Build-in packages);

3. *Il nome di un package dovrebbe essere composto usando una gerarchia simile a quella degli indirizzi Web ma in senso inverso, ovvero dal più generico al più specifico. Es: it.nomeazienda.*

Una volta definito il nome di un package, affinché una classe possa essere archiviata al suo interno, è necessario aggiungere la parola chiave **package** all'inizio del codice sorgente della definizione di classe contenente il nome completo del package che la conterrà come mostrato nel prossimo esempio:

```
package javamattone.esercizi.capitolo5;

public class CicloArrayConForEach {
  public static void main(String[] args) {
    String mesiDellAnno[] = { "gennaio", "febbraio", "marzo",
        "aprile", "maggio", "giugno", "luglio", "agosto",
        "settembre", "ottobre", "novembre", "dicembre" };
    for(String meseDellAnno : mesiDellAnno)
      System.out.println(meseDellAnno);
  }

}
```

La parola chiave **package** non deve assolutamente essere preceduta da nessuna linea di codice (è la prima riga all'interno di un file sorgente). Tutti le definizioni di classe java dovrebbero iniziare con la parola chiave **package**.

Per rispettare la regola, tutte le classi che non appartengono ad uno specifico package vengono assegnate automaticamente ad un package speciale con il nome vuoto.

Distribuzione di classi

Organizzare classi Java mediante package non ha come unico beneficio quello di organizzare definizioni di oggetti secondo una struttura logica, piuttosto i package rappresentano un meccanismo efficace per la distribuzione di un'applicazione Java.

Per capire le ragioni di quest'affermazione, dobbiamo fare un salto indietro nel tempo, a quando gli analisti del *Green Group* tracciarono le prime specifiche del linguaggio Java. Essendo Java un

linguaggio nato per la creazione di applicazioni internet, si rese necessario studiare un meccanismo in grado di ridurre al minimo i tempi di attesa dell'utente durante il trasferimento delle classi dal server al computer dell'utente. Le prime specifiche del linguaggio, stabilirono che i package Java potessero essere distribuiti in forma di archivi compressi in formato zip, riducendo drasticamente i tempi necessari al trasferimento dei file attraverso la rete.

Se all'inizio doveva essere uno stratagemma per ridurre le dimensioni dei file, con il passare del tempo la possibilità di distribuire applicazioni Java di tipo enterprise mediante archivi compressi si è rivelata una caratteristica alquanto vantaggiosa, tanto da stimolare la SUN nella definizione di un formato di compressione chiamato *jar* o *Java Archive* che, basandosi sull'algoritmo zip, inserisce all'interno dell'archivio compresso un file contente informazioni relative all'utilizzo delle definizioni di classe. Creare un archivio secondo questo formato, è possibile utilizzando l'applicazione *jar* che può essere trovata nella cartella *bin* all'interno della cartella di installazione dello Java SDK.

La applicazione *jar* può essere eseguita tramite la riga di comando ed ha la seguente sintassi:

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

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

c: *crea un nuovo archivio vuoto;*

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

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

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

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

CLASSPATH =/home/user/pro/mattone.jar

Manifest file

Quando creiamo un archivio jar, all'interno viene creato automaticamente un file.

META-INF/MANIFEST.MF

All' interno di un archivio jar può esistere solo un file manifest e contiene di default le seguenti righe:

Manifest-Version: 1.0
Created-By: 11.0.3 (AdoptOpenJDK)

Ogni riga del file manifest è formata da una coppia chiave valore; ogni chiave è definita *header*. Di seguito alcune delle chiavi più comuni che possono essere trovate all'interno del file manifest.

Manifest-Version: la versione della specifica del file manifest;

Created-By: la versione e le informazioni del tool che hanno creato il file manifest;

Multi-Release: se impostato a true, allora il jar file è di tipo Multi-Release (formato introdotto con java 9 per consentire di memorizzare all'interno di uno stesso jar file classi compilate con diverse versioni del compilatore java;

Built-By: il nome dell'utente o azienda che ha creato il package;

Build-Jdk: la versione del JDK con cui è stato compilato il package;

Se il nostro archivio jar contiene un programma eseguibile (classe java con metodo main), i seguenti due header possono essere utilizzati per specificare un *entry point*, il percorso della classe, la posizione di librerie esterne (*dipendenze*) o risorse necessarie:

Main-Class: nome completo della classe senza (non è richiesto .class);

Class-Path: una lista separata da spazi di path relativi ad altre librerie o risorse necessarie.

Qualora nel file manifest sia presente un header Main-Class, per seguire l'applicazione basterà chiamare la JVM con l'opzione -jar seguito dal nome dell'archivio:

java -jar miaApplicazioneJava.jar

File manifest e sicurezza

Grazie al file manifest possiamo firmare digitalmente un file jar. Il processo di firma digitale di una file jar è tuttavia fuori scopo di questo libro; mi limiterò quindi a dire che qualora sia richiesta il livello maggiore di sicurezza e verifica, potremmo utilizzare il tool fornito con il JDK *jarsigner*.

Sempre mediante manifest file è possibile aggiungere informazioni relative alla versione corrente della libreria/applicazione:

Name: il nome del package;

Implementation-Build-Date: la data di creazione;

Implementation-Title: il titolo della applicazione/libreria;

Implementation-Vendor: il nome del fornitore;

Implementation-Version: la vesione;

Specification-Title: il nome della specifica (se implementa una specifica);

Specification-Vendor: il fornitore/proprietario della specifica;

Specification-Version: la versione di riferimento per la specifica implementata;

Ad esempio, il file manifest, all'interno dell'archivio jar del connettore Mysql Server implementato dalla Oracle Corporation (proprietaria di MySQL Server) e basato sullo standard JDBC (Java DataBase Connectivity), contiene ad esempio le seguenti righe per il versionamento:

Specification-Title: JDBC
Specification-Version: 4.2
Specification-Vendor: Oracle Corporation
Implementation-Title: MySQL Connector/J
Implementation-Version: 8.0.16
Implementation-Vendor: Oracle

Istruzione import

Il runtime di Java fornisce un ambiente completamente dinamico in cui le classi non sono caricate fino a che non sono referenziate per la prima volta durante l'esecuzione dell'applicazione. Questo meccanismo consente di importare o modificare e ricompilare singole classi senza dover necessariamente ricaricare intere applicazioni.

Poiché il byte-code di ogni definizione di classe Java è memorizzato in un unico file avente lo stesso nome della classe, ma con estensione *.class*, la virtual machine può trovare i file binari appropriati cercando nelle cartelle specificate nella variabile di ambiente CLASSPATH. Inoltre, poiché le classi possono essere organizzate in package, è necessario specificare a quale package una classe appartenga pena l'incapacità della virtual machine di trovarla.

Un modo per indicare il package di appartenenza di una classe, è quello di specificarne il nome ad ogni chiamata alla classe. In questo caso diremo che stiamo utilizzando *nomi qualificati*.

Per meglio comprendere l'utilizzo dei nomi qualificati, prendiamo in esame una applicazione la cui struttura è schematizzata nella prossima immagine:

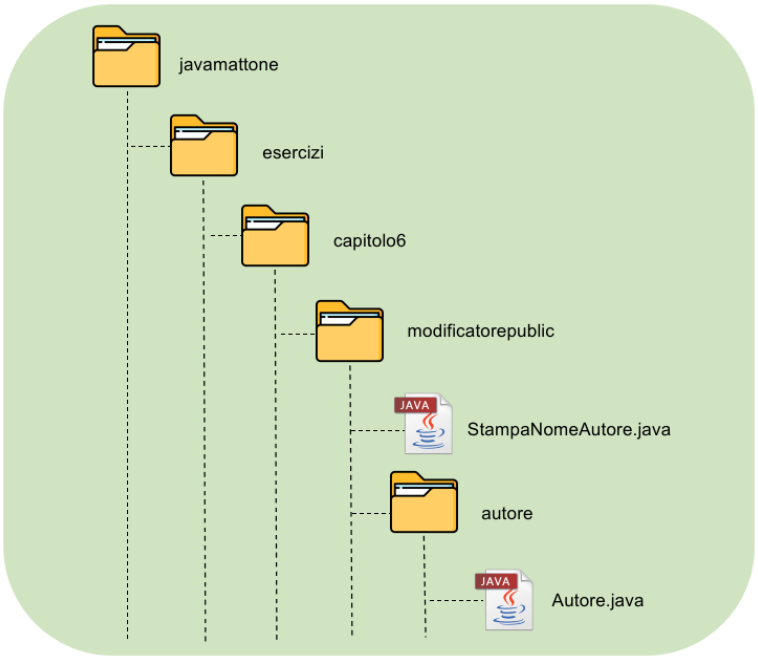


Immagine 19 - Nomi qualificati

La classe Autore ha la seguente definizione:

```
package javamattone.esercizi.capitolo6.modificatorepublic.autore;

public class Autore {
  static final String NOME = "Massimiliano";
  static final String COGNOME = "Tarquini";

  public String getNome() {
    return NOME;
  }
  public String getCognome() {
    return COGNOME;
  }
}
```

La classe *StampaNomeAutore*, la cui definizione è riportata a seguire, utilizza il nome qualificato per fare riferimento alla classe *Autore*:

```
package javamattone.esercizi.capitolo6.modificatorepublic;

public class StampaNomeAutore {
  public static void main(String[] args) {
    javamattone.esercizi.capitolo6.modificatorepublic.autore.Autore autore =
    new javamattone.esercizi.capitolo6.modificatorepublic.autore.Autore();
    System.out.println(autore.getNome()+","+autore.getCognome());
```

```
  }
}
```

L'uso di nomi qualificati non è sempre comodo, soprattutto quando i package sono organizzati con gerarchie a molti livelli come nel nostro esempio. Per venire incontro al programmatore, Java consente di specificare una volta per tutte il nome qualificato di una classe all'inizio del file contenente la definizione di classe, utilizzando la parola chiave **import**.

L'istruzione **import** ha come unico effetto quello di identificare univocamente una classe e quindi di consentire al compilatore di risolvere nomi di classe senza ricorrere ogni volta a nomi qualificati. Utilizzando questa istruzione, la classe *StampaNomeAutore* può essere riscritta affinché La JVM sia in grado di risolvere il nome di *Autore* ogni volta che sia necessario, semplicemente utilizzando il nome di classe.

La nuova definizione di classe è quindi la seguente:

```
package javamattone.esercizi.capitolo6.modificatorepublic;
import  javamattone.esercizi.capitolo6.modificatorepublic.autore.Autore;
public class StampaNomeAutore {
  public static void main(String[] args) {
    Autore autore = new Autore();
    System.out.println(autore.getNome()+","+autore.getCognome());
  }
}
```

Capita spesso di dover però utilizzare un gran numero di classi appartenenti ad un unico package. Per questi casi l'istruzione import supporta l'uso del carattere fantasma '*' che identifica tutte le classi pubbliche appartenenti ad un package. Il seguente codice è assolutamente equivalente al precedente:

```
package javamattone.esercizi.capitolo6.modificatorepublic;

import  javamattone.esercizi.capitolo6.modificatorepublic.autore.*;

public class StampaNomeAutore {
  public static void main(String[] args) {
    Autore autore = new Autore();
    System.out.println(autore.getNome()+","+autore.getCognome());
  }
}
```

Nonostante il carattere * abbia la funzione di wildcard, l'istruzione import non consente altre forme rispetto alle due riportate. Pertanto la forma

import *javamattone.esercizi.capitolo6.modificatorepublic.autore.Au**

non è consentita.

Tutte le definizioni di classe appartenenti al package java.lang non richiedono l'utilizzo dell'istruzione **import** per essere utilizzate.

Per concludere

In definitiva quindi, i package non sono solo contenitori di classi utili alla distribuzione di applicazioni Java, ma presentano una serie di vantaggi rilevanti che vale la pena evidenziare prima di concludere questo capitolo.

1. Un package dovrebbe rappresentare un gruppo di classi logicamente correlate tra loro.

Ogni package dovrebbe contenere gruppi di classi arbitrariamente grandi che costituiscono tra loro unità logiche. Le classi appartenenti ad un package dovrebbero pertanto implementare specifici sotto-contesti legati al dominio applicativo generale.

Il nome di un package contribuisce quindi alla comprensibilità del codice.

2. Definiscono uno spazio dei nomi (namespace).

Quando implementiamo operazioni complesse con un grande numero di classi o dipendenze, è molto facile che si vengano a creare omonimie. Grazie ai package è possibile risolvere il problema delle omonimie: il *nome qualificato* di una classe (che comprende il nome del package di appartenenza), consente di identificare univocamente la classe. Pertante il package contribuisce alla definizione dello spazio dei nomi.

3. Definiscono gli ambiti di visibilità delle classi;

Facendo riferimento all'esempio precedente, la classe autore è stata definita come segue:

```
public class StampaNomeAutore {
  .......
}
```

E' facile verificare che omettendo la parola chiave **public**, non sarà più possibile compilare la classe *StampaNomeAutore*, e questo perché , per default, una classe appartenente ad un package è visibile solo alle classi appartenenti allo stesso package a meno di modificarne la visibilità con gli opportuni modificatori. Il modificatore **public** rende la definizione di classe visibile a tutte le classi della applicazione.

Questa caratteristica dei package, che consente di attuare strategie per l'incapsulamento del codice basato, sarà comunque analizzata in dettaglio nei prossimi capitoli.

7. Definizione di classi ed oggetti

Introduzione

In questo capitolo saranno trattati gli aspetti specifici del linguaggio Java relativi alla definizione di classi ed alla creazione di oggetti: le regole sintattiche base per la creazione di classi, l'allocazione di oggetti e la determinazione del punto di ingresso (*entry point*) di un'applicazione.

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

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

Metodi

Un'istruzione rappresenta il mattone per creare le funzionalità di un oggetto. Nasce spontaneo chiedersi: come sono organizzate le istruzioni all'interno delle classi?

I metodi rappresentano il cemento che tiene assieme tutti i mattoni e raggruppano blocchi di istruzioni riuniti a fornire una singola funzionalità. Essi hanno una sintassi molto simile a quella della definizione di funzioni ANSI C e possono essere descritti con la seguente forma:

```
[modificatori] tipo_di_ritorno nome(lista_parametri_formali){
    istruzione
    [istruzione]
}

lista_parametri_formali = tipo identificatore [,lista_parametri_formali];
```

Dove *[modificatori]* identifica i modificatori che descrivono le proprietà del metodo in termini di visibilità (**private**, **protected**, **public**) e modificabilità (**final**), *tipo_di_ritorno* e tipo rappresentano ogni tipo di dato (primitivo o classe) e *nome* ed *identificatore* sono stringhe alfanumeriche, iniziano con una lettera e possono contenere caratteri numerici.

DEFINIZIONE: l'elenco tra parentesi (tipo identificatore [,tipo identificatore]) è anche chiamato elenco dei parametri formali.

La dichiarazione di una classe Java, deve sempre contenere la definizione di tipo di ritorno prodotto da un metodo: *tipo_di_ritorno*; se il metodo non ritorna valori, dovrà essere utilizzato il tipo speciale **void**.

Java prevede la possibilità di creare metodi con un numero variabile di argomenti, per farlo la sintassi è la seguente:

lista_parametri_formali = tipo ... identificatore | tipo identificatore [,lista_parametri_formali];

DEFINIZIONE: In logica, matematica, e informatica, ***l'arietà*** *di una funzione (metodo) o di un'operazione è il numero degli argomenti o operandi che richiede la funzione (metodo).*

Nel prossimo esempio mostra come creare un metodo con un numero variabile di parametri formali:

```
public class DemoElementiMultipli {

  void metodoConElementiMultipli(String... arg) {
    for (String argomento : arg) {
      System.out.println(argomento);

    }
  }
}
```

Il metodo *metodoConElementiMultipli* accetta come parametri un numero arbitrario di stringhe. Come appare chiaro dall'esempio, il parametro *arg* in questo caso dovrà essere trattato come un array del tipo specificato.

Definizione di una classe

Le istruzioni sono organizzate utilizzando metodi che contengono codice eseguibile che può essere invocato passandogli un numero limitato di valori come argomenti. D'altro canto, Java è un linguaggio orientato ad oggetti e come tale, richiede i metodi siano organizzati internamente alle classi.

Nel primo capitolo abbiamo associato il concetto di classe a quello di categoria; se trasportato nell'ambito del linguaggio di programmazione la definizione non cambia, ma è importante chiarire le implicazioni che ciò comporta. Una classe Java deve rappresentare un oggetto concettuale e per poterlo fare, deve raggruppare dati e metodi assegnando un nome comune.

La sintassi per la definizione di una classe è la seguente:

```
class nome{
        dichirazione_dei_dati
        dichirazione_dei_metodi
}

dichirazione_dei_dati =
        [modificatore] tipo identificatore;
        [dichiarazione_dei_dati]
```

I dati ed i metodi contenuti all'interno della definizione sono chiamati membri della classe e devono essere rigorosamente definiti all'interno del blocco di dichiarazione; non è possibile in nessun modo dichiarare variabili globali, funzioni o procedure. Questa restrizione del linguaggio Java scoraggia il programmatore ad effettuare una decomposizione procedurale, incoraggiando di conseguenza ad utilizzare l'approccio orientato agli oggetti. Valgono le seguenti definizione:

DEFINIZIONE: Le variabili dichiarate all'interno del blocco di dichiarazione di una classe sono dette variabili di istanza o dati membro della classe.

DEFINIZIONE: I metodi dichiarati all'interno del blocco di dichiarazione di una classe sono chiamati metodi membro della classe o metodi di istanza.

Ricordando la classe *Libro* descritta nel primo capitolo, avevamo stabilito che un libro è tale solo se contiene pagine da sfogliare, strappare ecc.. Utilizzando la sintassi di Java potremmo fornire una prima grossolana definizione della nostra classe nel modo seguente:

```
public class Libro {
    // dichiarazione delle variabili di istanza
    int numeroDiPagine = 100;
    int paginaCorrente = 0;
    String lingua = "Italiano";
    String tipologia = "Testo di letteratura Italiana";

    // dichiarazione dei metodi membro
    int getPaginaCorrente() {
            return paginaCorrente;
    }
    int getPaginaSuccessiva() {
                paginaCorrente++;
                return paginaCorrente;
    }
    int getPaginaPrecedente() {
                paginaCorrente--;
                return paginaCorrente;
    }
    String liguaggio() {
                return lingua;
    }
    String tipologia() {
                return tipologia;
    }
}
```

Una classe, nella programmazione orientata agli oggetti, è un costrutto di un linguaggio di programmazione usato come modello per creare oggetti. Il modello comprende attributi e metodi che saranno condivisi da tutti gli oggetti creati (*istanze*) a partire dalla classe.

Variabili reference

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

Per chiarire questo punto, esaminiamo la seguente dichiarazione:

int *contatore;*

La JVM, quando incontra questa riga di codice, crea il puntatore ad una variabile intera chiamata *contatore,* e contestualmente alloca quattro byte in memoria per l'immagazzinamento del dato inizializzandone il valore a 0, e questo perché la dimensione di un tipo primitivo è nota e di conseguenza la JVM è in grado di reagire di conseguenza.

Con le classi lo scenario cambia: in questo caso la JVM crea una variabile che conterrà il puntatore all'oggetto in memoria, ma non alloca risorse. Di fatto l'oggetto non viene creato fino a che non sarà fatto esplicitamente mediante l'operatore **new** (cosa che vedremo successivamente).

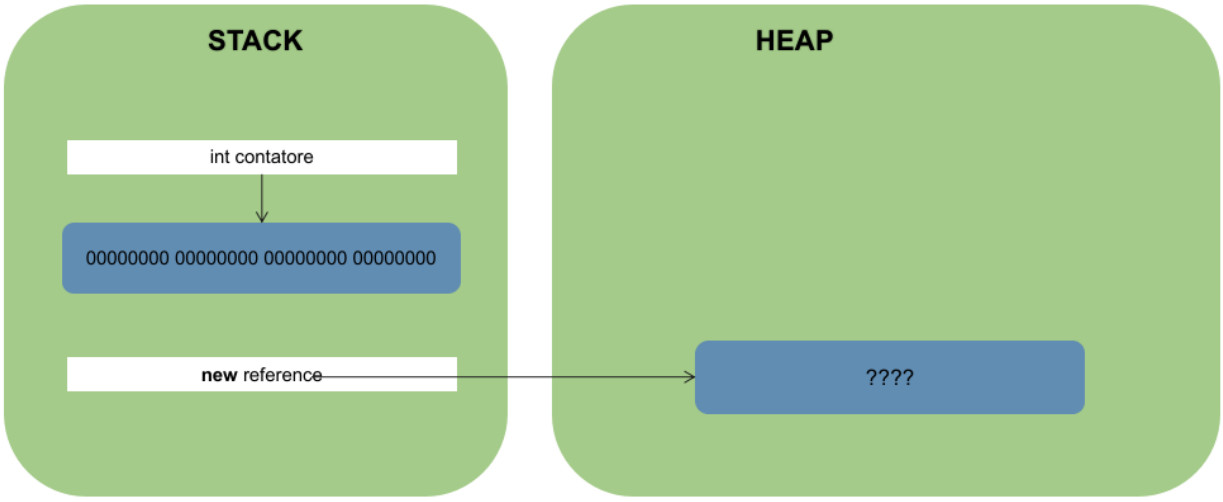


Immagine 20 differenza tra variabili primitive e variabili reference

Le variabili di questo tipo sono dette variabili *reference* ed hanno l'unica capacità di puntare ad oggetti del tipo compatibile: ad esempio una variabile *reference* di tipo *String* non può contenere puntatori oggetti di tipo *Integer*.

A differenza di oggetti che sono allocati nello *heap space*, le variabili primitive vengono allocate direttamente nello *stack space*. Nella seconda parte del libro il modello della memoria in Java è discusso approfonditamente.

DEFINIZIONE: Un reference è una variabile speciale che tiene traccia di istanze di tipi non primitivi.

Oltre che per gli oggetti, Java utilizza lo stesso meccanismo per gli array che non sono allocati al momento della dichiarazione, ma la JVM crea una variabile che conterrà il puntatore alla

corrispondente struttura dati in memoria. Di conseguenza, quando andremo a dichiarare un array mediante la sintassi nota:

tipo identificatore[];

non essendo ancora nota la dimensione dell'array la JVM creerà una variabile reference per allocare successivamente spazio in memoria non appena sarà nota la dimensione dell'array (*Immagine 21 Variabili reference ed arra*).

Le variabili reference sono concettualmente molto simili ai puntatori C e C++, ma non consentono la *conversione intero/indirizzo* o le *operazioni aritmetiche sugli indirizzi.*

Possono essere ugualmente utilizzate per la creazione di strutture dati complesse come liste, alberi binari e array multidimensionali. In questo modo eliminano gli svantaggi derivanti dall'uso di puntatori, mentre ne mantengono tutti i vantaggi.

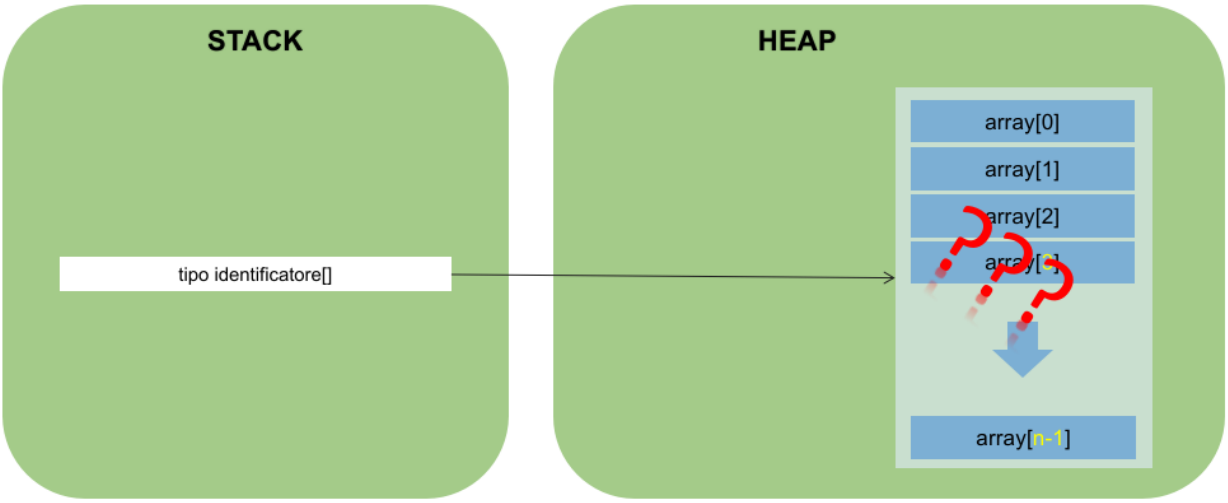


Immagine 21 Variabili reference ed array

La rappresentazione di un dato nel linguaggio C e C++ rispecchia il corrispondente dato macchina e questo comporta:

1. Il compilatore riserva solo la memoria sufficiente a rappresentare un tipo di dato (per una variabile di tipo byte saranno riservati 8 bit e per un variabile di tipo int 16 ecc.);

2. Il compilatore C e C++ non garantisce che la precisione di un dato sia quella stabilita dallo standard ANSI e di conseguenza, un tipo int potrebbe essere rappresentato con 16 o 32 bit secondo l'architettura di riferimento.

Java rappresenta i dati allocando sempre la stessa quantità di memoria indipendentemente dal tipo di dato da rappresentare: di fatto, le variabili si *comportano come se*: quello che cambia è che il programmatore vedrà una variabile **byte** comportarsi come tale ed altrettanto per le altre primitive. Questo garantisce la portabilità del byte-code su ogni architettura, assicurando che una variabile si comporterà sempre allo stesso modo.

Scope di una variabile Java

Come abbiamo già visto in precedenza, parlando di variabili, i blocchi di istruzioni ci forniscono il meccanismo necessario a determinare i confini dello scope di una variabile: di fatto, una variabile è referenziabile solo all'interno del blocco di istruzioni che contiene la sua dichiarazione ed ai sotto-blocchi contenuti.

Nel caso di classi le cose si complicano leggermente e richiedono l'introduzione di qualche nuova definizione.

DEFINIZIONE: Block scope: o variabili di blocco.

Ne abbiamo già parlato nei paragrafi precedenti. Una variabile definita all'interno di un blocco di codice sarà accessibile sollo all'interno del blocco stesso. Termina di esistere non appena l'esecuzione del codice esce dal blocco.

DEFINIZIONE: Method level scope: o variabili locali.

Ogni variabile dichiarata all'interno di un metodo, parametri formali compresi, non è accessibile al di fuori del metodo.

```
public class Macchina{
  public String colore;
  private int velocita;
  public void verniciatura(String nuovoColore, String vecchioColore) {
    // nuovoColore e vecchioColore sono accessibili solo in questo metodo
    // nuovoColore e vecchioColore sono distrutte appena il metodo verniciatura.
    // termina la sua esecuzione
  }
}
```

Nonostante la flessibilità messa a disposizione dal linguaggio Java, è buona regola definire tutte le variabili di istanza all'inizio del blocco di dichiarazione della classe e tutte le variabili locali al principio del blocco che ne delimiterà lo scope.

Questi accorgimenti non producono effetti durante l'esecuzione dell'applicazione, ma migliorano la leggibilità del codice sorgente consentendo di identificare facilmente quali saranno le variabili utilizzate per realizzare una determinata funzionalità.

DEFINIZIONE: Class level scope: o variabili di istanza.

Ogni variabile dichiarata all'interno di una classe è accessibile da ogni metodo nella classe. A seconda del modificatore di accesso (es: public, private) può essere acceduta da altri oggetti al di fuori della classe di definizione.

Non è possibile utilizzare la parola chiave var (inferenza automatica del tipo) per le variabili di istanza.

```
public class Macchina{
    private String colore;
    public String targa;
    private int velocita;

    public void verniciatura(String nuovoColore) {
                // colore è accessibile da tutti i metodi della classe ma non è
                // accessibile da altri oggetti
       colore = nuovoColore;
    }

    public class CompraMacchina{
            public static void main(String[] args){
                    Macchina miaNuovaMacchina = new Macchina();
                    //targa è accessibile dall'esterno e pertanto
                    //può essere modificato da altri oggetti
                    macchina.targa = "AB 123 CD";
            }
    }
```

Metodi differenti possono contenere dichiarazioni di variabili con identificatore uguale. Le variabili locali possono avere lo stesso identificatore delle variabili di istanza: in questo caso, sarà necessario specificare esplicitamente quale variabile si voglia referenziare utilizzando la parola speciale **this** che approfondiremo nei paragrafi seguenti.

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

```
public class Scope3 {
        //dichiarazione della variabile membro appoggio
        int appoggio = 5;
        void somma(int valore){
                //dichiarazione della variabile locale appoggio
                int appoggio = 3;
                //Sommo valore alla variabile locale
                appoggio = appoggio + valore;
                System.out.println("La variabile locale dopo la somma vale: "+appoggio);
                //sommo valore alla variabile membro
                this.appoggio = this.appoggio + valore;
                System.out.println("La variabile membro dopo la somma vale: "+this.appoggio);
                //sommo la variabile locale alla variabile membro
                this.appoggio = this.appoggio + appoggio;
                System.out.println("Dopo la somma delle due variabili,  vale:"+this.appoggio);
        }
```

```
}
```

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

L'esecuzione del codice darà il seguente risultato:

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

Metodi speciali: getter e setter

Abbiamo già parlato di incapsulamento e ne parleremo ancora in dettaglio nei prossimi paragrafi. Per il momento concedetemi di introdurre alcuni metodi speciali, i *getter* e i *setter*, che vengono usati per proteggere dati. Per ogni variabile di istanza, un metodo *getter* restituisce il suo valore, mentre un metodo *setter* ne imposta o ne aggiorna il valore. Per questo motivo, *getter* e *setter* sono anche conosciuti rispettivamente come *accessor* e *mutator*.

Riprendendo l'esempio del paragrafo precedente, possiamo riscrivere la classe *Macchina* nel modo seguente:

```
public class Macchina{
        private String colore;
         private String targa;
         private int velocita;
        public getTarga(){
                return targa;
        }
        public setTarga(String targa){
                //prima si modificare la targa potrebbe controllare se
                        //stiamo usando un numero di targa valido
                this.targa= targa;
        }
          public getColore(){
                return colore;
        }
        public setColore(String colore){
                this.colore = colore;
        }
```

```
}
```

```
public class CompraMacchina{
        public static void main(String[] args){
                Macchina miaNuovaMacchina = new Macchina();
                //la variabile di istanza targa non è più accessibile direttamente
                //ma possiamo utilizzare il metodo setter per impostarne il valore
                macchina.setTarga("AB 123 CD");
        }
}
```

Per convenzione, i metodi *getter* iniziano con il prefisso *get* ed i metodi *setter* iniziano con il prefisso *set*, seguiti dal nome della variabile. In entrambi i casi la prima lettera del nome della variabile deve essere maiuscola.

Normalmente i metodi *getter* e *setter* sono metodi pubblici.

In generale è buona norma utilizzare sempre i metodi *getter* e *setter* quando è necessario accedere in lettura/scrittura a variabili di istanza. Mediare l'accesso alle variabili di istanza mediante *setter* (incapsulamento) consente anche di controllare che il dato sia coerente con il contesto della classe ed eventualmente tornare un errore.

Nell'esempio precedente, potremmo controllare che la targa sia una numero di targa valido, prima di impostare la variabile membro targa.

Oggetto null

Come per i tipi primitivi, ogni variabile reference richiede che, al momento della dichiarazione, le sia assegnato un valore iniziale. Per questo tipo di variabili, il linguaggio Java prevede il valore speciale *null* che rappresenta un oggetto inesistente ed è utilizzato dalla JVM come valore prestabilito. Quando un'applicazione tenta di utilizzare una variabile reference contenente un riferimento a *null*, la JVM produrrà un messaggio di errore in forma di oggetto di tipo *NullPointerException*[6].

Oltre ad essere utilizzato come valore prestabilito per le variabili reference, l'oggetto *null* gioca un ruolo importante nell'ambito della programmazione per la gestione delle risorse: quando ad una variabile reference è assegnato il valore *null*, l'oggetto referenziato è rilasciato e se non utilizzato sarà inviato al *garbage collector* che si

[6] tratteremo le eccezioni nei capitoli successivi

occuperà di rilasciare la memoria allocata rendendola nuovamente disponibile.

Altro uso che può essere fatto dell'oggetto *null* riguarda le operazioni di comparazione come mostrato nel prossimo esempio in cui creeremo la definizione di classe per una struttura dati molto comune: la *Pila* o *Stack*.

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

La definizione di una p*ila* deve contenere i due metodi:

***void** push(**int**)*

***int** pop()*

Il metodo *push* ritorna un tipo **void** e prende come parametro un numero intero da inserire sulla cima della pila, il metodo *pop* non accetta parametri, ma restituisce l'elemento sulla cima della pila.

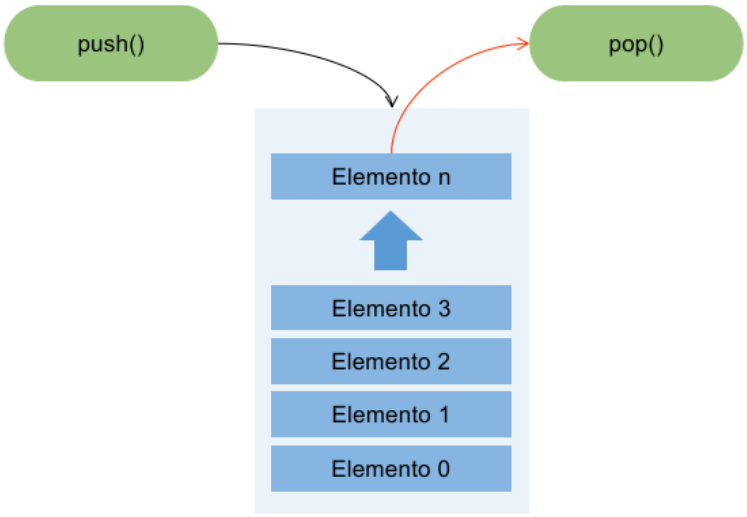


Immagine 22 - Struttura dati Pila

Una possibile definizione di classe è la seguete:

```
public class Pila {
   //dati è un array di interi. Al momento della dichiarazione
   // la variabile reference punta all'oggetto null
  int[] dati;
  int cima;

  void push(int dato) {
              if (dati == null) {
                cima = 0;
                dati = new int[20];
              }
              if (cima < 20) {
                dati[cima] = dato;
                cima++;
              }
  }

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

All'interno della definizione del metodo *void push(int i)* per prima cosa viene controllato se l'array è stato inizializzato utilizzando l'*operatore di uguaglianza* con il valore *null*, ed eventualmente è allocato un array di venti numeri interi. A seguire, l'algoritmo esegue un controllo per verificare se è possibile inserire elementi all'interno dell'array.

In particolare, essendo 20 il numero massimo di interi contenuti, mediante l'istruzione **if** il metodo accerta che la posizione puntata dalla variabile cima sia minore della lunghezza massima dell'array (ricordiamo che in un array le posizioni sono identificate a partire da 0): se l'espressione *cima<20* restituisce il valore *true*, il numero intero passato come parametro di input viene inserito nell'array nella posizione cima. La variabile cima viene quindi aggiornata in modo che punti alla prima posizione libera nell'array.

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

```
        cima ++;
    }
```

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

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

Possiamo quindi rispondere alla domanda rimasta disattesa nel paragrafo *Destinazione non trovata!*: l'oggetto *null* rappresenta il valore di default di una variabile reference dichiarata e mai stanziata.

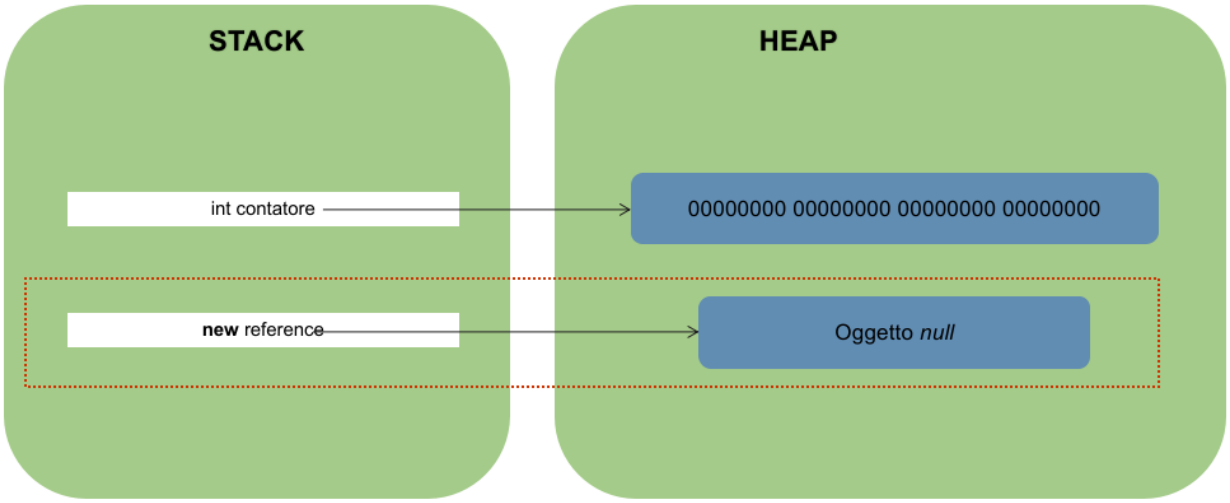


Immagine 23 Valore di default di una variabile reference

Per concludere, parlando di dichiarazioni di variabili, le due forme sono assolutamente equivalenti:

tipo identificatore

tipo identificatore = null

dove tipo rappresenta il nome di una classe. Non vale per i tipi primitivi.

Non è possibile utilizzare la parola chiave **var** per variabili locali inizializzate a *null*. Ad esempio:

```
var variabileNonIzializzata = null;
```

```
produrrà il seguente errore in fase di compilazione:
error: cannot infer type for local variable nullInitialized
var nullInitialized = null;
^
(variable initializer is 'null')
1 error
```

Creare istanze: operatore new

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

La sintassi dell'operatore **new** per creare il nuovo oggetto dalla sua definizione di classe è la seguente:

```
new NomeDellaClasse() ;
```

Le parentesi sono necessarie ed hanno un significato particolare che sveleremo presto, *NomeDellaClasse* è il nome di una definizione di classe appartenente alle API di Java oppure definita dal programmatore. Ad esempio, per creare un oggetto di tipo Pila utilizzando la definizione di classe del paragrafo precedente faremo uso dell'istruzione:

```
Pila unaIstanzaDiPila = new Pila();
```

La riga di codice utilizzata, dichiara una variabile *unaIstanzaDiPila* di tipo *Pila*, crea l'oggetto utilizzando la definizione di classe e memorizza il puntatore alla memoria riservata nella variabile reference.

Un risultato analogo può essere ottenuto anche nel modo seguente:

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

L'immagine *Immagine 24 Creazione di un oggetto: effetto sulla memori* mostra l'effetto dell'operatore **new** sulla memoria allocata dalla JVM per l'applicazione java: non appena viene invocato l'operatore, la JVM alloca spazio per contenere l'oggetto nello *heap-space* ed alloca spazio per rappresentare la variabile reference *unaIstanzaDiPila* nello *stack-space*;

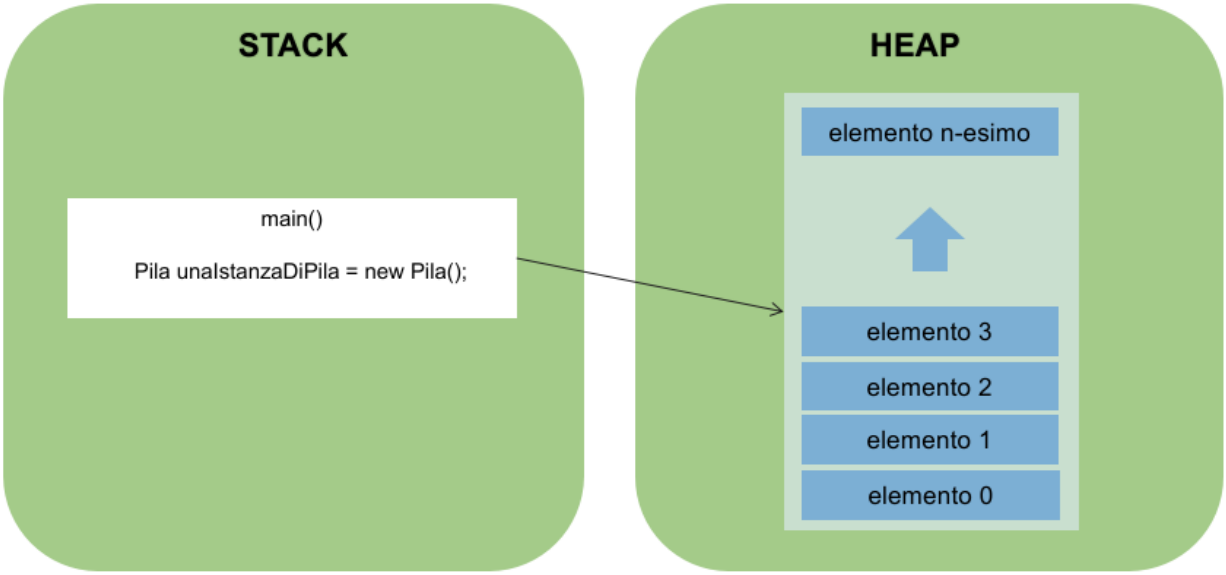


Immagine 24 Creazione di un oggetto: effetto sulla memoria

Ricordando quanto detto nei capitoli precedenti parla di array, notiamo come gli array vengano creati in maniera simile alle classi Java:

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

o, analogamente al caso precedente:

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

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

Oggetti ed array in forma anonima

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

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

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

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

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

In verità non è questo il modo più semplice per trasformare un intero in una stringa e, la classe *Integer* mette a disposizione metodi che non richiedono l'esistenza di una istanza della classe. Per completezza narrativa consentitemi comunque l'esempio.

Nel secondo caso, l'oggetto di tipo *Integer* è utilizzato senza essere associato a nessun identificatore e di conseguenza, è detto *oggetto anonimo*. Più in generale:

DEFINIZIONE: un oggetto è anonimo quando è creato utilizzando l'operatore ***new*** *omettendo la specifica del tipo dell'oggetto ed il nome dell'identificatore.*

Come gli oggetti, anche gli array possono essere utilizzati nella loro forma anonima. Anche in questo caso un array viene detto anonimo quando viene creato omettendo il nome dell'identificatore.

Utilizzare gli oggetti. Operatore punto "."

Una volta creato un oggetto, è molto probabile che vogliate anche utilizzarlo per farne qualcosa: potreste volerne modificare le variabili di istanza oppure chiamarne i metodi per svolgere azioni. Ci viene in aiuto l'*operatore . punto*.

L'operatore punto, se la visibilità lo consente, abbiamo visto fornire l'accesso alle variabili di istanza di una classe, tramite il suo identificatore. In questo caso la sintassi è semplice:

istanza_di_una_classe.identificatore

Vedremo, parlando del modificatore **static**, che esiste un tipo di variabile membro che non è da considerarsi variabile di istanza. E' quindi possibile, in questi casi, utilizzare l'operatore punto anche semplicemente utilizzando il nome della classe:

nome_della_classe.identificatore

Oltre ai dati membro, anche i metodi di una classe Java sono accessibili mediante lo stesso operatore. La sintassi è simile ma tiene conto del fatto che un metodo potrebbe richiedere il passaggio di parametri formali:

istanza_di_una_classe.nome_del_metodo([parametri_formali])

Come per le variabili, esistono metodi speciali che non richiedono una istanza della classe (oggetto) per essere eseguiti. In questo caso sarà ancora una volta possibile utilizzare il nome della classe piuttosto che un riferimento ad una sua istanza.

nome_della_classe.nome_del_metodo([parametri_formali])

Auto referenza esplicita

Consideriamo ora una possibile definizione della classe *Punto* che rappresenta un punto sul piano cartesiano:

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

        public int getX() {
                return x;
        }
        public void setX(int x2) {
                x = x2;
        }
        public int getY() {
                return y;
        }
        public void setY(int y2) {
                y = y2;
        }
}
```

Funzionante ma non è la migliore soluzione possibile perché il codice diventa meno leggibile: gli identificativi raddoppiano, e anche il rischio di sbagliarsi durante lo sviluppo. Sarebbe meglio se nel metodo *setter* si potesse utilizzare lo stesso identificativo della variabile di istanza.

Potremmo riscriverla in qualche modo come segue?

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

        public int getX() {
```

```
            return x;
        }
        public void setX(int x) {
            x = x;
        }
        public int getY() {
            return y;
        }
        public void setY(int y) {
            y = y;
        }
}
```

"La risposta è dentro di voi ma è ... SBAGLIATA!"

cit. Renato Guzzanti - Quelo

Perdonatemi la citazione, la risposta è ovviamente no a meno di andare contro alle regole definite parlando di *Class level scope* e *Method level scope*. Java prevede però un modo di auto referenza particolare identificabile con la variabile reference **this**. Il valore di **this** è modificato automaticamente dalla JVM affinché, ad ogni istante, sia sempre riferito all'oggetto attivo, intendendo per *oggetto attivo* l'istanza della classe in esecuzione durante la chiamata al metodo corrente. Questa modalità di accesso viene detta **autoreferenza esplicita** ed è applicabile ad ogni tipo di dato e metodo membro di una classe.

La nostra classe può essere quindi riscritta nel modo seguente:

```
public class Punto {
        int x;
        int y;
        public int getX() {
            return x;
        }
        public void setX(int x) {
            this.x = x;
        }
        public int getY() {
            return y;
        }
        public void setY(int y) {
            this.y = y;
        }
}
```

Per convenzione, il parametro del metodo *setter* ha lo stesso nome della variabile di istanza di cui è *mutator*.

La variabile reference **this** è una variabile reference a tutti gli effetti e come tale gode delle proprietà elencate di seguito.

*1. **this** può essere utilizzata per fare riferimento alle variabili di istanza della classe.*

*2. **this** può essere utilizzata per invocare i metodi della classe (vedi auto referenza implicita).*

```
public class EsempioThis {
        void metodo2() {
                System.out.println("sono nel metodo 2");
        }
        void metodo1() {
                System.out.println("Sono nel metodo 1");
                this.metodo2();
                // same as metodo2()
        }
}
```

*3. **This** può essere passata come argomento di un metodo*

```
public class EsempioThis {
        void metodo2() {
                System.out.println("sono nel metodo 2");
        }
        void metodo1() {
                System.out.println("Sono nel metodo 1");
                metodo2();// same as this.m()
                this.metodo2();
        }
        void metodo3(EsempioThis esempiothis)
        {
                esempiothis.metodo1();
        }

        void metodo4(){
                metodo3(this);
        }
}
```

*4. **This** può essere usata per tornare l'istanza corrente di una classe*

```
public class EsempioThis {
        .......

        EsempioThis getIstanza(){
                return this;
        }

}
```

Auto referenza implicita

Poiché, come abbiamo detto, ogni metodo deve essere definito all'interno di una definizione di classe, il meccanismo di *auto referenza esplicita* è molto comune in applicazioni Java; se però un riferimento non è ambiguo, Java consente di utilizzare un ulteriore meccanismo detto di *auto referenza implicita*, per mezzo del quale è possibile accede a dati membro o metodi di una classe senza necessariamente utilizzare esplicitamente la variabile reference **this**.

il meccanismo di *autoreferenza implicita* è valido sia per la chiamata ai metodi della stessa classe che per l'utilizzo di variabili di istanza.

```
public class EsempioThis {
        String tantoPerDirneUna;
        void metodo2() {
                System.out.println("sono nel metodo 2");
        }

        void metodo1() {
                System.out.println("Sono nel metodo 1");
                this.metodo2();
                // identico a this.metodo2()
                // utilizza this in maniera implicita
                metodo2();
                //possiamo fare riferimento alla variabile di istanza
                //senza usare this
                tantoPerDirneUna = "inizializziamo la variabile di istanza";
        }
}
```

Il meccanismo su cui si basa l'auto referenza implicita è legato alla visibilità di una variabile. Ricordando le regole che definiscono lo scope delle variabili Java, la JVM ricerca una variabile non qualificata risalendo a ritroso tra i diversi livelli dei blocchi di codice.

Inizialmente, Java ricerca la dichiarazione della variabile all'interno del blocco di istruzioni corrente:

1. se la variabile non è una variabile appartenente al blocco, risale tra i vari livelli del codice fino ad arrivare alla lista dei parametri formali del metodo corrente.

2. Se neanche la lista dei parametri formali del metodo soddisfa la ricerca, Java legge il blocco di dichiarazione dell'oggetto corrente utilizzando implicitamente la variabile reference ***this****.*

3 . Nel caso in cui la variabile non è neanche una variabile di istanza dell'oggetto, un codice di errore è generato al momento della produzione del byte-code dal compilatore.

Meno ambiguo è invece l'uso dell'*auto referenza implicita* se impiegata nella chiamata ai metodi della classe. In questo caso Java applicherà soltanto il terzo punto dell'algoritmo descritto per la determinazione dei riferimenti alle variabili, e questo perché di fatto un metodo non può essere definito all'interno di un altro, ne può essere utilizzato come argomento per il passaggio di parametri.

Stato di un oggetto Java

Gli oggetti Java rappresentano spesso tipi di dati molto complessi ed il cui stato, a differenza di un tipo primitivo, non può essere definito semplicemente dal valore della variabile reference che fanno riferimento all'oggetto stesso.

DEFINIZIONE: definiamo stato di un oggetto Java il valore in un certo istante delle variabili di istanza rilevanti della classe.

Di fatto, non tutte le variabili di istanza di una classe concorrono alla definizione dello *stato* di un oggetto bensì solo quelli sufficienti a fornire, ad un determinato istante, informazioni sufficienti a fotografarne la condizione esatta.

Come faccio quindi a determinare quali sono le variabili di istanza che concorrono allo stato di una classe? Ci viene in aiuto la definizione seguente:

DEFINIZIONE: Si definisce comportamento di un oggetto, le modalità di azione o di reazione di un oggetto, in termini di come il suo stato cambia e come i messaggi passano.

Per poter definire le variabili rilevanti, basterà quindi fare riferimento alle variabili che hanno influenza sul comportamento della classe.

Ad esempio, lo stato di un oggetto di tipo Punto sarà definito, ad un certo istante, dal valore delle due variabili di istanza x e y;

```
public class Punto {
        int x;
        int y;
        public int getX() {
                return x;
        }
        .....
}
```

Possiamo infine definire un criterio di uguaglianza tra due oggetti dello stesso tipo. Diremo quindi che:

DEFINIZIONE: due oggetti java dello stesso tipo (ovvero istanze di una stessa classe) sono uguali tra loro solo se si trovano nello stesso stato.

Comparazione di oggetti

La comparazione di oggetti Java è leggermente differente rispetto ad altri linguaggi di programmazione, e dipende dal modo in cui Java manipola gli oggetti stessi. Di fatto, un'applicazione Java non usa oggetti, ma *variabili reference come oggetti*.

Una normale comparazione effettuata utilizzando l'operatore di uguaglianza == metterebbe a confronto il riferimento agli oggetti in memoria e non il loro stato, producendo un risultato *true* solo se le due variabili reference puntano allo stesso oggetto e non se i due oggetti distinti di tipo uguale, sono nello stesso stato.

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

Molte volte però, ad un'applicazione Java, potrebbe tornare utile sapere se due istanze separate di una stessa classe sono uguali tra loro ovvero se due oggetti java dello stesso tipo si trovano nello stesso stato al momento del confronto. Java prevede un metodo speciale chiamato *equals()* che confronta lo stato di due oggetti: tutti gli oggetti appartenenti alle core API implementano questo metodo. Poiché ereditato da una classe base particolare che analizzeremo in seguito parlando di ereditarietà, anche gli oggetti istanze di classi definite dal programmatore ne forniscono una implementazione. Tuttavia, come capiremo meglio quando parliamo di ereditarietà, andrà specializzato in maniera specifica.

Il confronto delle due stringhe descritte nell'esempio precedente assume quindi la forma seguente:

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

Metodi e variabili di classe. Il qualificatore static

Finora abbiamo mostrato segmenti di codice dando per scontato che siano parte di un processo attivo: in tutto questo c'è una falla. Per sigillarla è necessario fare alcune considerazioni:

1) ogni metodo deve essere definito all'interno di una classe (questo incoraggia ad utilizzare il paradigma Object Oriented);

2) I metodi devono essere invocati utilizzando una variabile reference inizializzata in modo che faccia riferimento ad un oggetto in memoria.

Questo meccanismo rende possibile l'auto referenza poiché, se un metodo è invocato in assenza di un oggetto attivo, la variabile reference **this** non sarebbe inizializzata. Il problema è quindi che, in questo scenario, un metodo per essere eseguito richiede un oggetto attivo, ma fino a che non c'è qualcosa in esecuzione un oggetto non può essere caricato in memoria. L'unica possibile

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

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

```
class Euro {
        static double convertiInLire(double ammontare){
                return 1936.27*ammontare;
        }
}
```

Un metodo statico esiste sempre a prescindere dallo stato dell'oggetto (che potrebbe anche non essere inizializzato). Per accedere ad un metodo statico, un metodo consiste nell'utilizzare il nome della classe come se fosse una variabile reference:

```
double valoreInLire = Euro.convertiInLire(1,15);
```

Una alternativa è utilizzare una variabile reference:

```
Euro valuta= new Euro();
double valoreInLire = valuta.convertiInLire(1,15);
```

Proviamo ora o a modificare la nostra classe euro:

```
class Euro {
        final double CONVERSIONE_EURO_LIRA = 1936.27;
        static double convertinLire(double ammontare){
                // errore di compilazione. Non è possibile
                // la autoreferenza esplicita ne implicita.
                // this non è valorizzato
                return CONVERSIONE_EURO_LIRA*ammontare;
        }
}
```

Nella nuova versione della classe *Euro*, abbiamo espresso il tasso di conversione come una costante. Nonostante la cosa possa sembrare assolutamente lecita e ben scritta, nella realtà non sarà possibile compilare il codice.

Poiché abbiamo detto che un metodo statico esiste sempre, a prescindere che esista o no un oggetto in memoria, un metodo statico non inizializza la variabile reference **this**. Di conseguenza un oggetto statico non può utilizzare membri non statici della classe che, nel caso dell'esercizio precedente dovrà essere riscritta nel modo seguente:

```
class Euro {
        final static double CONVERSIONE_EURO_LIRA = 1936.27;
```

```
    static double convertinLire(double ammontare){
        return CONVERSIONE_EURO_LIRA*ammontare;
    }
}
```

A questo punto è necessario fare alcune precisazioni: se finora abbiamo parlato di variabili di istanza, variabili locali e metodi non statici è arrivato il momento di estendere le definizioni alle variabili e metodi di classe:

DEFINIZIONE: si definisce variabile di classe (ovvero variabili statiche) un dato comune a tutti gli oggetti di una classe. Non dipendono da nessuna istanza della classe stessa.

Mentre le variabili di istanza hanno generalmente valori diversi in ogni oggetto creato, per le variabili di classe viene memorizzato un unico valore che è comune a tutti gli oggetti della classe.

DEFINIZIONE: si definisce metodo di classe (ovvero metodi statici) un metodo comune le cui funzionalità appartengono all'intera classe, e non hanno nulla a che fare con una istanza della classe stessa.

I metodi statici non sono associati quindi ad una istanza ma solo ad una classe. Di conseguenza non potranno interagire con le variabili di istanza, ma solamente con variabili di classe.

Poiché le variabili di classe (statiche) sono comuni a tutti gli oggetti (dipendono dalla classe e non dall'istanza), e le costanti sono valori non modificabili, è consigliabile dichiarare tutte le costati come:

static final *tipo NOME_COSTANTE = valore;*

In questo modo non costringeremo la JVM ad allocare spazio in per ogni costante ogni qualvolta si utilizzi l'operatore **new** per ottenere una istanza della classe.

La convenzione vuole (e le convenzioni sono importanti) che nella dichiarazione di una costante si rispetti la seguente sintassi:

*[**public**|**private**|**protected**]**static final** tipo NOME_COSTANTE = valore;*

Anche l'ordine è importante e la parola chiave **static** deve comparire dopo il modificatore e prima della parola chiave **final**.

Il metodo main

Affinché la JVM possa eseguire un'applicazione, è necessario che abbia ben chiaro quale debba essere il primo metodo da eseguire. Questo metodo è detto *entry point* o punto di ingresso dell'applicazione.

Come per il linguaggio C, Java riserva allo scopo l'identificatore di membro *main*. Ogni classe può avere il suo metodo *main()*, ma solo quello della classe specificata alla JVM sarà eseguito all'avvio del processo. Questo significa che ogni classe di un'applicazione può rappresentare un potenziale punto di ingresso, che può quindi essere scelto all'avvio del processo scegliendo semplicemente la classe desiderata.

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

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

Infine, per il metodo *main* è necessario utilizzare il modificatore **public** che accorda alla virtual machine il permesso per eseguire il metodo. Tutto questo ci porta ad un'importante considerazione finale:

In java tutto è un oggetto, anche un'applicazione.

La classe System

Una delle classi predefinite in Java è la classe *System*. Questa classe ha una serie di metodi statici e rappresenta il sistema su cui la applicazione Java è in esecuzione.

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

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

Il metodo statico *System.exit(int number)* causa la terminazione della applicazione Java producendo il codice di errore passato come argomento al metodo.

L'oggetto *System* fornisce anche il meccanismo per ottenere informazioni relative al sistema ospite mediante il metodo statico *System.getProperty(String)*, che ritorna il valore della proprietà di sistema richiesta o, in caso di assenza, ritorna il valore **null**. Le proprietà, accessibili mediate questo metodo, possono variare a seconda del sistema su cui l'applicazione è in esecuzione; la tabella seguente elenca il nome delle proprietà la cui definizione è garantita indipendentemente dalla Java Virtual Machine a partire dalla versione 1.2

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

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

```
class TestSistema {
        public static void main(String[] argv) {
                System.out.println("file.separator= "+System.getProperty("file.separator") );
                System.out.println("java.class.path= "+System.getProperty("java.class.path") );
                System.out.println("java.class.version= "+System.getProperty("java.class.version") );
                System.out.println("java.home= "+System.getProperty("java.home") );
                System.out.println("java.version= "+System.getProperty("java.version") );
                System.out.println("java.vendor= "+System.getProperty("java.vendor") );
                System.out.println("java.vendor.url= "+System.getProperty("java.vendor.url") );
                System.out.println("os.name= "+System.getProperty("os.name") );
                System.out.println("os.arch= "+System.getProperty("os.arch") );
                System.out.println("os.version= "+System.getProperty("os.version") );
                System.out.println("path.separator= "+System.getProperty("path.separator") );
                System.out.println("user.dir= "+System.getProperty("user.dir") );
                System.out.println("user.home= "+System.getProperty("user.home") );
                System.out.println("user.name= "+System.getProperty("user.name") );
        }
}
```

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

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

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

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

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

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

8. Incapsulamento

Introduzione

L'incapsulamento di oggetti è il processo di mascheramento dei dettagli implementativi di un oggetto ad altri oggetti, con lo scopo di proteggere porzioni di codice o dati critici. I programmi scritti con questa tecnica, risultano molto più leggibili e limitano i danni dovuti alla propagazione di errori od anomalie all'interno dell'applicazione.

Una analogia con il mondo reale è rappresentata dal cambio dell'auto. Chiunque abbia la patente sa perfettamente come cambiare le marce utilizzando la leva del cambio.

Immagine 25 leva del cambio nasconde i dettagli tecnici degli organi del motore

La leva del cambio ci consente di cambiare le marce, mettere in folle oppure impedire all'autista di compiere azioni pericolose ad esempio inserendo la retromarcia quando la macchina è in movimento, e tutto questo nascondendo i dettagli degli organi del cambio.

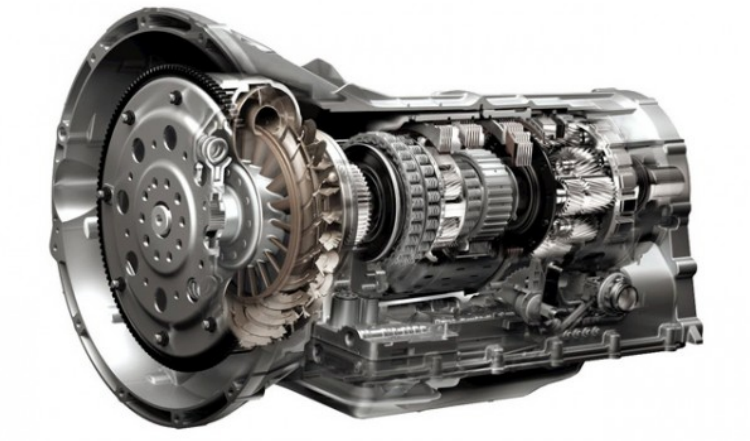


Immagine 26 Organi del cambio

Limitando l'uso della leva del cambio ad un insieme limitato di operazioni, si può: primo, proteggere macchina ed autista da azioni dannose; secondo, consentire all'autista di guidare tutte le macchine allo stesso modo senza doversi preoccupare delle differenze esistenti tra le varie realizzazioni degli organi del cambio.

Uno degli scopi primari di un disegno *Object Oriented*, dovrebbe essere proprio quello di fornire all'utente un insieme di dati e metodi che danno il senso dell'oggetto in questione. Questo è possibile farlo senza esporre le modalità con cui l'oggetto tiene traccia dei dati ed implementa il corpo (metodi) dell'oggetto. Nascondendo i dettagli, possiamo assicurare a chi utilizza l'oggetto che ciò che sta utilizzando è sempre in uno stato consistente a meno di errori di programmazione dell'oggetto stesso.

Uno stato consistente è uno stato permesso dal disegno di un oggetto. E' però importante notare che uno stato consistente non corrisponde sempre a quanto aspettato dall'utente dell'oggetto. Se infatti l'utente trasmette all'oggetto parametri errati, l'oggetto si troverà in uno stato consistente, ma non in quello desiderato.

Il concetto alla base del principio di incapsulamento è quella dell'*occultamento dell'informazione*, meglio noto con il termine di *information hiding*. Tale possibilità esprime l'abilità di nascondere al mondo esterno tutti i dettagli implementativi definiti all'interno di un oggetto.

Modificatori public, private e protected

Java fornisce supporto per l'incapsulamento a livello di linguaggio, tramite i modificatori **public**, **private** e **protected** da utilizzare al momento della dichiarazione di variabili e metodi.

*DEFINIZIONE: i membri di una classe, o l'intera classe, si definiscono **public** se sono liberamente accessibili da ogni oggetto componente l'applicazione.*

*DEFINIZIONE: i membri di una classe si definiscono **private** se possono essere utilizzati solo dai membri della stessa classe.*

*DEFINIZIONE: i membri di una classe si definiscono **protected se** possono essere utilizzati dai membri della stessa classe oppure da altre classi purché appartenenti allo stesso package.*

I membri privati mascherano i dettagli dell'implementazione di una classe.

Membri di una classe che non sono dichiarati **public** o **private** sono, per definizione, accessibili solo alle classi appartenenti allo stesso package. In letteratura, capita spesso che vi venga fatto riferimento come al *modificatore default* .

*DEFINIZIONE: i membri di una classe che non sono dichiarati **public** o **private** sono definiti package friendly.*

La prossima immagine schematizza quanto detto sopra:

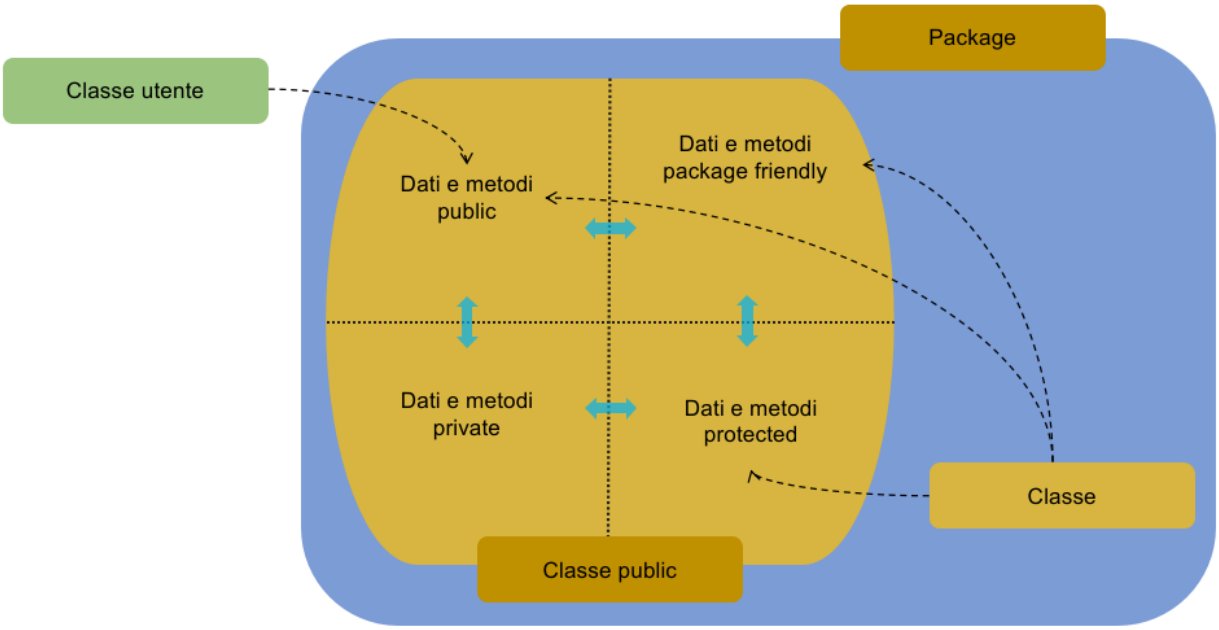


Immagine 27 Incapsulamento: modificatori public e private

Il modificatore private in dettaglio

Il modificatore **private** realizza incapsulamento a livello di definizione di classe e serve a definire membri che devono essere utilizzati solo da altri membri della stessa classe di definizione. E' il modificatore di accesso più restrittivo in assoluto il cui intento è quello di nascondere porzioni di codice della classe che non devono essere utilizzati da altre classi.

Un membro privato può essere utilizzato da un qualsiasi membro statico e non della stessa classe di definizione con l'accorgimento che i membri statici possono accedere solamente ad altri membri statici. E' consentito l'accesso anche ad altri oggetti di qualunque tipo purché esplicitamente passati per parametro.

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

private *tipo [identificatore];*

Oppure, nel caso di metodi:

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

E' buona regola dichiarare tutti le variabili di istanza di una classe **private** e mediare l'accesso a quelle per cui è necessario farlo utilizzando metodi *getter* e *setter*.

Il modificatore public in dettaglio

Il modificatore **public** consente di definire classi o membri di una classe visibili a qualsiasi oggetto definito all'interno dello stesso package e non. Questo modificatore deve essere utilizzato per definire l'interfaccia che l'oggetto mette a disposizione dell'utente.

Tipicamente metodi membro **public** utilizzano membri **private** per implementare le funzionalità dell'oggetto. Per dichiarare una classe od un membro pubblico si utilizza la parola chiave **public** anteposta alla dichiarazione :

public *tipo [identificatore];*

Oppure, nel caso di metodi:

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

Infine, nel caso di classi pubbliche

```
public class nome_classe{
        dichiarazione_dei_dati
        dichiarazione_dei_metodi
}
```

Il modificatore protected in dettaglio

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

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

protected *tipo [identificatore];*

Oppure, nel caso di metodi:

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

}
```

Nonostante possa sembrare ridondante, di questo modificatore torneremo a parlarne nei dettagli nel prossimo capitolo dove affronteremo il problema della ereditarietà.

Un esempio di incapsulamento

In questo paragrafo proviamo a mettere in pratica i principi di incapsulamento esposti nei paragrafi precedenti e lo facciamo realizzando un piccolo sistema di gestione dell'anagrafica clienti di un negozio.

Come abbiamo detto, il processo di incapsulamento riguarda la capacità di nascondere i dettagli implementativi di una classe al mondo esterno.

Il primo passo da compiere è quindi quello di dichiarare private tutte le variabili di istanza della classe.

```
public class Cliente {
        private String nome;
        private String cognome;
        private String numerodiTelefono;
}
```

Ovviamente, a questo punto, dobbiamo preoccuparci di creare un interfaccia pubblica affinché il mondo esterno possa accedere ai dati privati della nostra classe e lo facciamo utilizzando i metodi *getter* e *setter*.

```
public class Cliente {
        private String nome;
        private String cognome;
        private String numerodiTelefono;
        public String getNome() {
                return nome;
        }
        public void setNome(String nome) {
                this.nome = nome;
        }
        public String getCognome() {
                return cognome;
        }
        public void setCognome(String cognome) {
                this.cognome = cognome;
        }
        public String getNumerodiTelefono() {
                return numerodiTelefono;
        }
        public void setNumerodiTelefono(String numerodiTelefono) {
```

```
            this.numerodiTelefono = numerodiTelefono;
        }
    }
```

Abbiamo quindi creato la nostra interfaccia pubblica: tuttavia non è ancora chiarissimo quali siano i vantaggi dell'incapsulamento.

Notiamo che la classe *Anagrafica* ha un difetto perché consente di inserire valori *null* per entrambi *nome*, *cognome* del cliente. Possiamo però modificare la classe ed introdurre dei controlli a livello di interfaccia pubblica.

Poiché non abbiamo ancora affrontato la gestione degli errori in java, ci limiteremo a controllare se una variabile è *null* e stampare un messaggio video in caso positivo. La nuova versione della classe *Cliente* è quindi la seguente:

```
public class Cliente {
        private String nome;
        private String cognome;
        private String numerodiTelefono;
        public String getNome() {
            return nome;
        }
        public void setNome(String nome) {
                if(nome != null){
                  this.nome = nome;
                }else{
                  System.out.println("Impossibile accettare un nome vuoto");
                }

        }
        public String getCognome() {
             return cognome;
        }
        public void setCognome(String cognome) {
                if(nome != null){
                  this.cognome = cognome;
                }else{
                  System.out.println("Impossibile accettare un cognome vuoto");
                }
        }
        public String getNumerodiTelefono() {
            return numerodiTelefono;
        }
        public void setNumerodiTelefono(String numerodiTelefono) {
             this.numerodiTelefono = numerodiTelefono;
        }

}
```

E' finalmente evidente come, utilizzando l'incapsulamento delle variabili di istanza, abbiamo protetto i dati (nessun oggetto può accedervi direttamente *information hiding*) ed abbiamo utilizzato l'interfaccia pubblica per effettuare controlli utili a mantenere lo stato dell'oggetto consistente con il disegno della classe.

Notiamo ance un altra conseguenza importante dell'incapsulamento. Qualora volessimo introdurre controlli più complessi sui dati, ci basterà modificare l'implementazione dei metodi *getter* e *setter* senza modificare l'interfaccia pubblica. Da questo punto di vista l'incapsulamento offre quindi un altro vantaggio: potremmo modificare l'implementazione della nostra classe senza che oggetti provenienti dal mondo esterno ne risentano in quanto l'interfaccia è rimasta invariata.

Infine, tengo a fare presente che nell'esempio ci siamo limitati a realizzare incapsulamento a livello di dati e non ci siamo occupati dell'incapsulamento funzionale (metodi della classe). Tuttavia il principio rimane invariato: possiamo nascondere porzioni di codice proteggendo quei metodi i cui dettagli implementativi non devono interessare agli oggetti provenienti dal mondo esterno.

L'operatore new

Abbiamo già accennato che, per creare un oggetto dalla sua definizione di classe, Java mette a disposizione l'operatore **new** responsabile del caricamento dell'oggetto in memoria e della successiva creazione di un riferimento (*indirizzo di memoria*), che può essere memorizzato in una variabile di tipo reference ed utilizzato per accederne ai membri. Quest'operatore è paragonabile alla malloc in C, ed è identico al medesimo operatore in C++.

La responsabilità del rilascio della memoria allocata per l'oggetto non più in uso è del il Garbage Collector: per questo motivo, a differenza di C++ Java non prevede nessun meccanismo esplicito per distruggere un oggetto creato.

Quello che non abbiamo detto è che, questo operatore ha la responsabilità di consentire l'assegnamento dello stato iniziale dell'oggetto allocato. Di fatto, la sintassi dell'operatore **new** prevede un tipo seguito da un insieme di parentesi. Le parentesi indicano che, al momento della creazione in memoria dell'oggetto verrà chiamato un metodo speciale detto *costruttore*, responsabile proprio della inizializzazione del suo stato.

Le azioni compiute da questo operatore, schematizzate nella prossima figura, sono le seguenti:

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

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

*2 . Esegue il metodo costruttore dell'oggetto caricato per consentirne l'inizializzazione dei dati membro. Inizializza la variabile reference **this**.*

3 . Restituisce il riferimento alla locazione di memoria allocata per l'oggetto. Utilizzando l'operatore di assegnamento è possibile memorizzare il valore restituito in una variabile reference dello stesso tipo dell'oggetto caricato.

Immagine 28 Operatore **new**

Metodi costruttori

Tutti i programmatori, esperti e non, conoscono il pericolo costituito da una variabile non inizializzata. In un'applicazione Object Oriented, un oggetto è un'entità più complessa di un tipo primitivo e l'errata inizializzazione dello stato di un oggetto può essere causa della terminazione prematura dell'applicazione o della generazione di errori intermittenti difficilmente controllabili.

In molti altri linguaggi di programmazione, il responsabile dell'inizializzazione delle variabili è il programmatore. In Java, questo è impossibile poiché potrebbero essere membri privati di un oggetto, e quindi inaccessibili all'utente.

I costruttori sono metodi speciali chiamati in causa dall'operatore **new** al momento della creazione di un nuovo oggetto e servono ad impostarne lo stato iniziale.

I metodi costruttori utilizzano lo stesso nome della classe di cui sono membri e non restituiscono nessun tipo (void compreso).

Dal momento che Java garantisce l'esecuzione del metodo costruttore di un nuovo oggetto, un costruttore scritto intelligentemente garantisce che tutti i dati membro vengano inizializzati.

Sfogliando a ritroso tra gli esempi precedenti, dovrebbe sorgere spontaneo chiedersi come mai non abbiamo mai definito il metodo costruttore delle classi? Una classe di fatto non è mai sprovvista di costruttori. Di fatto:

Nel caso in cui il costruttore non sia definito dal programmatore, il compilatore Java crea automaticamente un costruttore senza parametri di default (*costruttore no-args*) .

I metodi costruttori sono responsabili della inizializzazione delle variabili di istanza di un oggetto al momento della sua creazione: garantiscono quindi che lo stato di un oggetto sia inizializzato correttamente rispettando il disegno della classe.

Per comprendere meglio i costruttori modifichiamo il nostro oggetto pila utilizzando il metodo costruttore per inizializzare le variabili di istanza. La pila creata conterrà 10 elementi al massimo e la variabile che punta alla cima della pila sarà inizializzata correttamente.

```
public class Pila {
        private int[] dati;
        private int cima;
        private int dimensioneMassima;

        /**
         * Il metodo costruttore della classe Pila, non ritorna nessun tipo
         * di dato, imposta la dimensione massima dell'array che
         * conterrà i dati della pila e crea l'array. Utilizza lo stesso
         * nome della classe e non ha tipo di ritorno
         */
        public Pila() {
                dimensioneMassima = 10;
                dati = new int[dimensioneMassima];
                cima = 0;
        }

        public void push(int dato) {
                if (cima < dimensioneMassima) {
                  dati[cima] = dato;
                  cima++;
                }
        }
        public int pop() {
                if (cima > 0) {
                  cima--;
                  return dati[cima];
                }
                return 0; // Bisogna tornare qualcosa
        }

}
```

Rispetto alla prima definizione della classe *Pila*, non è più necessario creare l'array di interi al momento della prima chiamata al metodo *push(int)* rendendo di conseguenza inutile il controllo sullo stato dell'array ad ogni sua chiamata:

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

Utilizzando il costruttore saremo quindi sempre sicuri che lo stato iniziale della classe è correttamente impostato.

Completiamo ricordando le principali differenze tra metodi membro e metodi costruttori di una classe:

1. Il costruttore deve avere lo stesso nome della classe. Non può averne un altro. I metodi possono avere qualsiasi nome.

2. Nel costruttore non si deve specificare il tipo del valore di ritorno e non è possibile nemmeno usare ***void****. Nei metodi è invece necessario indicare tipo,* ***void*** *o valore di ritorno.*

3. Il costruttore crea una nuova istanza (oggetto) della classe. I metodi non creano oggetti.

4. Il costruttore inizializza lo stato dell'oggetto appena creato. I metodi lo interrogano o lo modificano.

5. Un costruttore può essere invocato una sola volta per uno stesso oggetto mentre i metodi sono invocabili più volte per uno stesso oggetto.

Overloading dei costruttori

Java supporta molte caratteristiche per i costruttori, ed esistono molte regole per la loro creazione. In particolare, al programmatore è consentito scrivere più di un costruttore per una data classe, secondo le necessità di disegno dell'oggetto. Questa caratteristica permette di passare all'oggetto diversi insiemi di dati di inizializzazione, consentendo di adattarne lo stato ad una particolare situazione operativa.

Nell'esempio precedente, abbiamo ridefinito l'oggetto *Pila* affinché contenga un massimo di dieci elementi. Un modo per generalizzare l'oggetto è definire un costruttore che, prendendo come parametro un intero, inizializza la dimensione massima della *Pila* secondo le necessità dell'applicazione. La nuova definizione della classe potrebbe essere la seguente:

```
public class Pila {
        private int[] dati;
        private int cima;
        private int dimensioneMassima;

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

```
/**
* Questo metodo costruttore della classe Pila,
* consente di impostare la dimensione massima della Pila
* accettando un parametro di tipo int
*/
  public Pila(int capacitaMassimaDellaPila) {
            dimensioneMassima = capacitaMassimaDellaPila;
            dati = new int[dimensioneMassima];
            cima = 0;
  }

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

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

}
```

Avere aggiunto un secondo costruttore, ci consente di creare oggetti *Pila* di dimensioni fisse utilizzando il primo costruttore, o variabili utilizzando il costruttore che prende come parametro di input un tipo **int**. Ora, possiamo creare la nostra *Pila* liberi di poter decidere, al momento della creazione, quanti elementi potrà contenere. Per creare una istanza della classe *Pila* basterà utilizzare l'operatore new come segue:

```
Pila laMiaPila = new Pila(100);
Pila pilaDimensioneDiDefault = new Pila();
```

Le due chiamate, seppure con effetto differente, sono perfettamente funzionanti e valide. In definitiva quindi, la sintassi per definire un costruttore è la seguente:

```
[public|protected|private] nome_della_classe([lista_parametri_formali]) {
        istruzione
        [istruzione]
}
```

E la sintassi completa dell'operatore **new** di conseguenza, prende la forma seguente

```
new nome_della_classe([lista_parametri_formali]) ;
```

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

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

Chiamate incrociate tra costruttori

Un metodo costruttore ha la possibilità di effettuare chiamate ad altri costruttori appartenenti alla stessa definizione di classe. Questo meccanismo è utile perché i costruttori generalmente hanno funzionalità simili e un costruttore che assegna all'oggetto uno stato comune, potrebbe essere richiamato da un altro per sfruttare il codice definito nel primo.

Per chiamare un costruttore da un altro, è necessario utilizzare la sintassi speciale:

***this**(lista_dei_parametri);*

dove, lista_dei_parametri rappresenta la lista di parametri del costruttore che si intende chiamare.

Una chiamata incrociata tra costruttori, deve essere la prima riga di codice del costruttore chiamante. Qualsiasi altra cosa sia fatta prima, compresa la definizione di variabili, non consente di effettuare tale chiamata. Il costruttore corretto è determinato in base alla lista dei parametri: Java paragona lista_dei_parametri con la lista dei parametri di tutti i costruttori della classe fino a trovare i costruttore la cui lista dei parametri formali corrisponde alla chiamata.

Tornando alla definizione di *Pila*, notiamo che i due costruttori eseguono operazioni simili. Per ridurre la quantità di codice, possiamo chiamare un costruttore da un altro come mostrato nel prossimo esempio.

```
public class Pila {
        private int[] dati;
        private int cima;
        private int dimensioneMassima;

        /**
         * Il metodo costruttore della classe Pila, non ritorna nessun tipo
         * di dato, imposta la dimensione massima dell'array che
         * conterrà i dati della pila e crea l'array.
         */
        public Pila() {
                this(10);
        }

        /**
         * Questo metodo costruttore della classe Pila,
         * consente di impostare la dimensione massima della Pila
         * accettando un parametro di tipo int
         */
        public Pila(int capacitaMassimaDellaPila) {
                dimensioneMassima = capacitaMassimaDellaPila;
                dati = new int[dimensioneMassima];
                cima = 0;
        }

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

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

}
```

Costruttori private o protected: classi Singleton

Anche se l'esistenza dei modificatori di visibilità è strettamente legata al tema dell'incapsulamento, esistono situazioni in cui possono essere utilizzati per creare modelli

speciali di classi con proprietà del tutto particolari, ma che non hanno nulla a che fare con il tema trattato in questo capitolo.

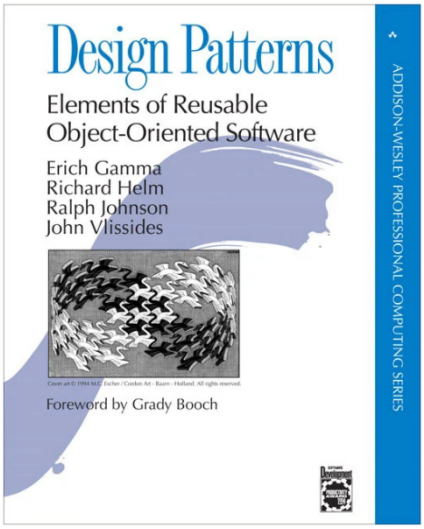
Torniamo per un attimo alla sintassi per la creazione di un costruttore. Qualcuno avrà certamente notato che i modificatori **private** e **protected** sono consentiti anche ne caso di questi metodi speciali.

*[**public**|**protected**|**private**] nome_della_classe([lista_parametri_formali]) {*
istruzione
[istruzione]
}

Possiamo certamente dire che, in quanto metodi, rispettano le regole di visibilità definite nei paragrafi precedenti: un costruttore **private** sarà visibile solo ad altri metodi della classe stessa, un costruttore **protected** potrà essere utilizzato solo da altri metodi della classe stessa oppure da oggetti le cui classi appartengono allo stesso package.

Ovviamente, poiché il costruttore può essere chiamato solo dall'operatore **new**, gli unici metodi di una classe che possono accedere ai costruttori privati sono i metodi statici (gli altri metodi hanno comunque bisogno di una istanza della classe, oggetto). Vale l'analogo per l'uso di **protected** con la differenza che un costruttore **protected** può essere comunque usato da una classe appartenente allo stesso package.

Per capire a cosa può servirci un costruttore definito private dobbiamo fare un salto indietro di 20 anni circa quando fu pubblicato per la prima volta un libro icona di quegli anni per gli addetti al settore: *"Design Patterns: Elements of Reusable Object-Oriented Software"*. I quattro autori *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides* anche noti come *GoF (Gand of Four)* grazie alla loro esperienza nella codifica di sistemi enterprise in C++ notarono che in programmazione esistono dei *pattern* di sviluppo (algoritmi che seguono schemi prestabiliti) che si ripetono. Questi pattern non sono relativi solo al C++ ma possono essere generalizzati a tutta la programmazione ad oggetti.



Il libro della Gand of Four è tutt'oggi un bestseller delle vendite su amazon.

Torniamo a noi. Tra i pattern descritti nel libro ne esiste uno chiamato *Singleton* di particolare interesse:

DEFINIZIONE: Singleton è un pattern creazionale che assicura che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale a questa istanza.

Il pattern singleton affronta quindi due aspetti della programmazione:

1. Assicurarsi che una classe possa avere una singola istanza.

Ad esempio classi che rappresentano accesso ad una risorsa condivisa oppure classi che rappresentano il contesto globale di una applicazione. Si capisce subito che è un comportamento

impossibile da implementare con un costruttore standard poiché, ad ogni chiamata, crea una nuova istanza della classe.

2. Fornire un punto di accesso globale all'unica istanza.

Similmente ad una variabile globale, un singleton consente di accedere ad alcuni oggetti o informazioni rilevanti da qualunque parte del codice. Rispetto ad una variabile globale, in quanto oggetto, protegge i dati mediante incapsulamento e rappresenta un unico punto di accesso ad una molteplicità di informazioni e funzionalità.

Passiamo quindi a creare il nostro primo *singleton*. In java esistono molto esempi di *singleton* forniti con le Java Core API; personalmente sono solito utilizzare questo pattern per rappresentare contesti applicativi globali, ovvero raccoglitori di oggetti id informazioni che devono essere creati ed inizializzati all'avvio della applicazione per essere condivisi con altri oggetti che concorrono alla applicazione. Di seguito una possibile implementazione di *singleton*:

```
public class Singleton{

        private static Singleton instance;
        public static Singleton getInstance() {
                instance = (instance == null) ? new Singleton () : instance;
                return instance;
        }
        private Singleton() {
        }
}
```

Per realizzare il singleton, la classe contiene un variabile statica chiamata *instance* di tipo "*il tipo stesso della classe*" anch'essa privata. In quanto variabile statica, ovvero variabile di classe, esiste unica, e non è accessibile da nessun altro oggetto in quanto dichiarata private. Dal momento che la classe ha solo un costruttore privato, qualsiasi tentativo di creare un istanza della classe con l'operatore **new** causerà un errore in fase di compilazione del codice.

il punto di accesso, che garantisce anche l'unicità della istanza della classe è il metodo:

```
public static Singleton getInstance() {
        return instance == null ? new Singleton() : instance;
}
```

anch'esso statico e dichiarato **public**. Il metodo controlla che la variabile *instance* sia inizializzata e, qualora la variabile sia *null* crea l'unica istanza possibile della classe e utilizza la variabile *instance* per memorizzarla.

Per accedere all'unica istanza della classe da qualsiasi punto del codice basterà chiamare:

```
GlobalContext.getInstance().getDataAvvioApplicazione();
```

Esistono diverse strategie per creare un singleton con java. Vederemo più avanti che quella utilizzata non è la strategia da preferire: in un regime di concorrenza più di un oggetto potrebbe accedere alla istanza della classe, e in alcuni casi si potrebbe verificare la creazione di una seconda istanza non voluta.

Blocchi di inizializzazione statici e di istanza

I metodi costruttori svolgono un ruolo importante in quanto incapsulano i dettagli implementativi della inizializzazione dello stato di un oggetto, tuttavia abbiamo visto che una classe Java non contiene solo membri di istanza, ma può contenere metodi e variabili statiche.

Per ovvi motivi, non è compito del metodo costruttore inizializzare le variabili di classe e questo perché:

*1. Il metodo costruttore è eseguito solo dal comando **new** quando viene creata una istanza dell'oggetto.*

2. Le variabili statiche sono variabili di classe e sono immediatamente disponibili non appena la classe viene caricata dal classloader.

Per risolvere il problema della corretta inizializzazione delle variabili statiche, a partire da Java 11, sono stati messi a disposizione i blocchi di ialinizzazione che, come per i metodi costruttori, incapsulano le logiche di inizializzazione dei membri statici di una classe. I blocchi di inizializzazione sono parti di codice anonime racchiuse tra parentesi graffe ed eseguite, a seconda del loro tipo, in fase di caricamento della classe (*blocchi statici*) o in fase di creazione di un oggetto della classe (*blocchi di istanza*). I blocchi di istanza possono essere utilizzati autonomamente o insieme a costruttori.

I criteri di esecuzione dei blocchi di inizializzazione sono i seguenti:

1. I blocchi statici vengono eseguiti quando la classe viene caricata in memoria dalla JVM. Ne consegue che i blocchi statici vengono eseguiti prima dei blocchi di istanza;

```
public class BlocchiStaticiEBlocchiDiIstanza {
        public BlocchiStaticiEBlocchiDiIstanza() {
                System.out.println("Chiamata a costruttore");
        }
        static {
                System.out.println("Blocco statico");
        }
        public static void main(String[] args) {
                BlocchiStaticiEBlocchiDiIstanza oca = new  BlocchiStaticiEBlocchiDiIstanza();
        }
        {
```

```
            System.out.println("Blocco di istanza");
        }
}
```

Una volta eseguita l'applicazione produrrà il seguente output:

```
Blocco statico
Blocco di istanza
Chiamata a costruttore
```

I due blocchi di inizializzazione sono eseguiti nell'rdine aspettato: quello statico, quello di istanza ed infine il metodo costruttore.

2. Una classe può avere più blocchi statici o di istanza e verranno eseguiti nella stessa sequenza in cui appaiono nel codice sorgente della classe;

```
public class BlocchiStaticiEBlocchiDiIstanza2 {

    public BlocchiStaticiEBlocchiDiIstanza2() {
        System.out.println("Chiamata a costruttore");
    }

    static {
        System.out.println("Blocco statico");
    }

    public static void main(String[] args) {
        BlocchiStaticiEBlocchiDiIstanza2 oca = new
            BlocchiStaticiEBlocchiDiIstanza2();
    }

    {
        System.out.println("Blocco di istanza");
    }

    static {
        System.out.println("Blocco statico 2");
    }

    {
        System.out.println("Blocco di istanza 2");
    }
}
```

```
Blocco statico
Blocco statico 2
```

```
Blocco di istanza
Blocco di istanza 2
Chiamata a costruttore
```

I blocchi statici sono eseguiti nell'ordine in cui compaiono: prima quelli statici e poi quelli di istanza.

3. I blocchi statici vengono eseguiti una sola volta al caricamento della classe, mentre i blocchi di istanza vengono eseguiti ogni volta che viene creato un oggetto della classe;

Grazie ai blocchi statici di inizializzazione possiamo riscrivere la classe *Singleton* nel modo seguente:

```
public class Singleton{
        private static Singleton instance;

        static{
                instance = new Singleton();
        }

        public static Singleton getInstance() {
                return instance;
        }
         private Singleton() {
        }
}
```

Classi interne nidificate

Le classi Java possiedono di un'importante caratteristica ereditata dai linguaggi strutturati come il Pascal in cui una funzione può essere dichiarata a qualsiasi livello del codice e la sua visibilità è limitata al blocco di codice che ne contiene la definizione. Java consente di dichiarare delle classi particolari, le classi interne, ovvero classi che possono essere dichiarate all'interno di altre classi.

Le classi interne rappresentano un ulteriore strumento per l'*information hiding*: possiamo infatti parlare di l'incapsulamento a livello di classe. In java esistono 4 tipi diversi di classi interne:

1. Classi interne non statiche o inner classes;

2. Classi interne statiche o static nested classes;

3. Classi locali;

4. *Classi anonime.*

Tuttavia, non è ancora il momento di parlare di classi anonime che saranno affrontate in seguito quando parleremo del concetto di interfaccia.

Classi inner

Una classe *inner* in Java è una classe definita in modo analogo ad ogni altra classe la cui definizione però è situata all'interno di un'altra classe chiamata *incapsulante* e raramente, *ospite*.

Nel prossimo esempio la classe *Esterna* è la classe incapsulante per la classe *Interna*.

```
public class Esterna{
        private int x;

        public class Interna{
                private int y;
                public void metodoInterno() {
                x=y;
                }
        }
}
```

Una classe *inner* è un membro, a tutti gli effetti, della classe *incapsulante*. Questa caratteristica non solo le consente di poter accedere ai metodi ed ai dati membro della classe incapsulante (vale la regola degli scope delle variabili), ma garantisce che ogni riferimento alla classe incapsulante sia relativo solo ad una determinata istanza. Formalmente,

DEFINIZIONE: Una classe interna è una classe definita all'interno della definizione della classe incapsulante ed esiste solo se associata in maniera univoca ad un'istanza della classe incapsulante.

In quanto membro della classe incapsulante, possiamo utilizzare l'operatore . (punto) per accedervi ed ottenere una sua istanza utilizzando l'istanza della sua classe ospite. Ecco alcuni modi equivalenti per ottenere un istanza di una classe *inner* **public**.

```
public static void main(String[] args){
        Esterna incapsulante = new Esterna();
        Esterna.Interna jk3 = incapsulante. new Interna();
        Interna jk4 = incapsulante.new Interna();
        Interna jk5 = new Incapsulante(). new Interna();

}
```

Le classi *inner* non possono avere metodi statici dal momento che, per definizione, sono implicitamente associate con un oggetto (istanza) della classe incapsulante e quindi non ha senso che contengano metodi statici.

A partire da Java 16, definitivamente con Java 17, anche le classi *inner* possono contenere membri statici.

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

```
class Esterna{
        class InnerLivello1{
                class InnerLivello2{
                ....
                }
        }
}
```

Riassumendo, le classi *inner* hanno le seguenti proprietà e restrizioni:

1. Non possono avere lo stesso nome della classe incapsulante;

2. In quanto membri di una classe, possono essere dichiarate private rendendole di fatto invisibili all'esterno;

3. Una classe inner può accedere a tutti i metodi e dati della classe ospitante;

4. Di contro, la classe ospitante può vedere solo la parte pubblica della classe inner;

5. Una istanza della classe inner non può esistere se non esiste una istanza della classe ospitante;

Classi inner ed autoreferenza

Parlando di autoreferenza, abbiamo definito il concetto di *istanza corrente* di una classe definendo *oggetto attivo* l'istanza della classe in esecuzione durante la chiamata ad un metodo. La natura delle classi interne estende necessariamente questo concetto; di fatto, nel caso di una classe interna, esisterà contemporaneamente più di un'istanza corrente. Nell'esempio mostrato nel paragrafo precedente, esisteranno contemporaneamente una istanza della classe *Esterna* ed una istanza della classe *Interna*.

Supponiamo ora di avere tre classi A,B,C strutturate come mostrato nell'immagine che segue; diremo allora che:

1. Durante l'esecuzione del codice della classe interna C, esistono le istanze correnti di A,B,C;

2. Durante l'esecuzione del codice della classe interna B, esistono le istanze correnti di A,B;

3. Durante l'esecuzione del codice della classe A, non esistono altre istanze correnti differenti da quella dell'oggetto attivo;

4. Nel caso di esecuzione di un metodo statico della classe A, non esistono istanze correnti di nessun tipo.

Immagine 29 Istanze correnti di una classe interna

In generale, durante l'esecuzione di un metodo qualsiasi appartenente ad una classe interna C esistono: l'istanza corrente della di C e tutte le istanze correnti delle classi che incapsulano C sino ad arrivare alla classe di primo livello.

Detto questo, sorge spontaneo domandarsi come le classi interne influiscono sull'utilizzo della variabile reference **this**.

Come tutte le classi, anche le classi interne utilizzano il meccanismo di autoreferenza implicita per determinare quale metodo o variabile utilizzare in caso di chiamata, ma se fosse necessario fare riferimento ad un particolare tipo di istanza corrente, deve essere utilizzata la variabile reference **this** preceduta dal nome della classe da riferire.

Nella figura precedente:

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

2 . Durante l'esecuzione del codice della classe interna B, this, A.this rappresentano i riferimenti alle istanze correnti delle classi B ed A;

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

Classi nested static

La classe *inner* può anche essere dichiarata statica (ovvero può essere annidata staticamente). In questo caso parleremo di classi *nested-static*. Questae classi annidate possono essere considerate equivalenti ad una classe normale tranne che, per motivi stilistici o di comodità, sono definite all'interno di un'altra classe.

Nel caso di classi nested-static il modificatore **static** si comporta quindi diversamente da come lo conosciamo indicando semplicemente che l'istanza della classe inner non dipende dall'istanza della classe incapsulante. Per questo non si parla di classi statiche, ma di classi annidate in maniera statica.

Come per le classi inner, anche per le classi nested-static valgono alcune regole o restrizioni:

1. *Sono accessibili dall'esterno usando la sintassi*

   new *nome_classe_ospite.nome_classe_nested_static(lista_parametri_formali)*

2. *Non valgono le regole di autoreferenza: non dipendono dall'istanza della classe incapsulante;*

3. *Non avendo nessun riferimento alle classi ospite, non possono accedere ai membri non statici della classe incapsulante;*

Ecco quindi un esempio completo di riepilogo:

```
public class Esterna {
        private int x;
        public class Interna {
                private int y;
                public void metodoInterno(){
                        Esterna.this.x = 40;
                }
        }
        public static class InternaStatica {
                private int y;
                public void metodoInterno(){
                //...
                }
                public static void metodoInternoDue(){
                //...
                }
        }
        public static void main(String[] args) {
                Esterna a = new Esterna();
                Interna l = new Esterna().new Interna();
                InternaStatica r = new Esterna.InternaStatica();
                r.metodoInterno();
                Esterna.InternaStatica.metodoInternoDue();
        }
}
```

Classi locali

Come abbiamo anticipato, le classi interne, possono essere definite anche nei blocchi di definizione dei metodi di una classe incapsulante e sono dette *classi locali*.

```
public class EsempioClasseLocale {
        public static void main(String[] args){
                class Punto{
                        private int x;
                        private int y;

                        public Punto(int x, int y){
                                this.x = x;
                                this.y = y;
                        }

                        public int getX() {
                                return x;
                        }
                         public void setX(int x) {
                                this.x = x;
                        }
                        public int getY() {
                                return y;
                         }
                        public void setY(int y) {
                                this.y = y;
                        }
                }
                Punto classeLocale = new Punto(1,2);
                System.out.println(classeLocale.getX()+","+classeLocale.getY());
        }
}
```

La classe locale Punto è quindi incapsulata all'interno del blocco del metodo main della applicazione *EsempioClasseLocale*. Le classi locali rispettano la regola dello scope di blocco pertanto sono unità che esistono solo all'interno del blocco in cui sono dichiarate.

Le classi locali possono accedere a tutte le variabili definite nello stesso blocco, ma con un unica accortezza: poiché Java non consente l'utilizzo di variabili globali, l'utilizzo del modificatore **final** rappresenta l'unico mezzo per prevenire eventuali errori nell'uso condiviso di una variabile locale da parte di due o più classi innestate.

L'uso del modificatore **final** è mostrato nel prossimo esempio:

```
public class EsempioClasseLocale2 {

        public static void main(String[] args){
                final int valore = 5;
                class Locale{

                        public Locale(){

                        }

                        public void stampaValore(){
                                //La prossima istruzione è consentita perchè valore
                                //è dichiarata final
                                System.out.println(valore);
                        }
                }
                 Locale locale = new Locale();
                locale.stampaValore();
        }
}
```

Se rimuoviamo **final**, il compilatore java produrrebbe un errore e la applicazione non verrebbe compilata.

Alcuni buoni motivi per utilizzare le classi nidificate

Ecco alcuni buoni motivi per cui vale la pena utilizzare questo tipo di classi:

1. E' un ottimo modo per raggruppare classi che sono solo di supporto per un altra classe (helper classes).

Se una classe è di supporto per una sola altra classe, è logico incapsularla nella classe che dovrà farne uso. Annidare le classi di supporto rende la struttura del package più leggera.

2. Rafforzare l'incapsulamento di dati membro sensibili

Consideriamo due classi A e B ed immaginiamo che B debba accedere ai membri di A che, altrimenti, sarebbero stati dichiarati **private**. Nidificare la classe B all'interno della classe A, B potrà accedere a tutti i membri della classe A, e di conseguenza i membri di A potranno essere dichiarati **private**. Sia i membri sensibili di A che la classe B saranno quindi nascoste ad altre classi rafforzando l'incapsulamento.

3. Rendono il codice più compatto e leggibile

Nidificare piccoli classi all'interno dell'unica classe che la utilizzerà renderà il codice più compatto e comprensibile

Singleton: una nuova strategia

Anche se i singleton sembrano oggetti tutto sommato semplici da creare e manipolare, come abbiamo anticipato, le loro implementazioni possono soffrire di vari problemi: di fatto potremmo finire per avere più di una sola istanza della classe e questo perché la strategia già proposta mediante variabile statica e metodo *getInstance()* statico non è *thread-safe*: funziona bene in una applicazione *single-threaded*, ma in una applicazione *multi-threading* non è in grado di garantire l'atomicità della operazione di creazione dell'istanza della classe.

Una operazione atomica consiste in un'operazione di esecuzione indivisibile dal punto di vista logico. Il concetto di thread safety ed atomicità sarà affrontato in dettaglio nel capitolo dedicato alle applicazioni multithreading in Java.

Abbiamo già proposto una alternativa thread-safe che fa uso di blocchi statici di inizializzazione; ne proponiamo ora una terza che utilizza le classi nested-static insieme ai blocchi statici di inizializzazione.

```
import java.time.Instant;

public class Singleton{

        protected static class ResourceHolder {
                public static final Singleton instance;
                static {
                        instance = new Singleton();
                }
        }
        public static Singleton getInstance() {
                return ResourceHolder.instance;
         }
        private Singleton () {
        }
}
```

Poiché il blocco statico viene eseguito non appena la classe è caricata in memoria dal classloader, e la classe statica si comporta come un membro di classe, abbiamo reso il nostro singleton thread-safe evitando il rischio di creare più istanze della stessa classe.

9. Ereditarietà

Introduzione

L'ereditarietà è la caratteristica dei linguaggi *Object Oriented* che consente di utilizzare definizioni di classe come base per la definizione di nuove che ne specializzano il concetto. L'ereditarietà offre inoltre un ottimo meccanismo per aggiungere funzionalità ad un programma con rischi minimi nei confronti di quelle già esistenti, nonché un modello concettuale che rende un programma *Object Oriented* auto-documentante rispetto ad un analogo scritto con linguaggi procedurali.

Per utilizzare correttamente l'ereditarietà, il programmatore deve conoscere a fondo gli strumenti forniti in supporto dal linguaggio. Questo capitolo introduce al concetto di ereditarietà in Java, alla sintassi per estendere classi, all'*overloading* e *overriding* di metodi.

Infine, in questo capitolo introdurremo ad una particolarità rilevante del linguaggio Java che, include sempre la classe *Object* nella gerarchia delle classi definite dal programmatore.

Disegnare una classe base

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

Facendo uso del meccanismo della ereditarietà, è necessario tener sempre ben presente alcuni concetti.

1. L'ereditarietà consente di utilizzare una classe come punto di partenza per la scrittura di nuove classi.

Questa caratteristica può essere vista come una forma di riciclaggio del codice: i membri della classe base sono concettualmente copiati nella nuova classe.

2. Come conseguenza diretta, l'ereditarietà consente alla classe derivata di modificare la superclasse.

In altre parole, ogni aggiunta o modifica ai metodi della superclasse, sarà applicata solo alla classe derivata. La classe base sarà quindi protetta dalla generazione di nuovi eventuali errori, che rimarranno circoscritti alla classe derivata. La classe derivata per ereditarietà, supporterà tutte le caratteristiche della classe base.

In definitiva, tramite questa tecnica è possibile creare nuove varietà di entità utilizzando entità già definite mantenendone tutte le caratteristiche e le funzionalità. Questo significa che se una applicazione è in grado di utilizzare una classe base, sarà in grado di utilizzarne la derivata allo

stesso modo. Per questi motivi, è importante che una classe base rappresenti le funzionalità generiche delle varie specializzazioni che andremo a definire.

Il modello di ereditarietà proposto da Java è un detto di ereditarietà singola. A differenza da linguaggi come il C++ in cui una classe derivata può ereditare da molte classi base (ereditarietà multipla), Java consente di poter ereditare da una sola classe base. Nelle due prossime figure sono illustrati rispettivamente i due modelli di ereditarietà multipla e singola.

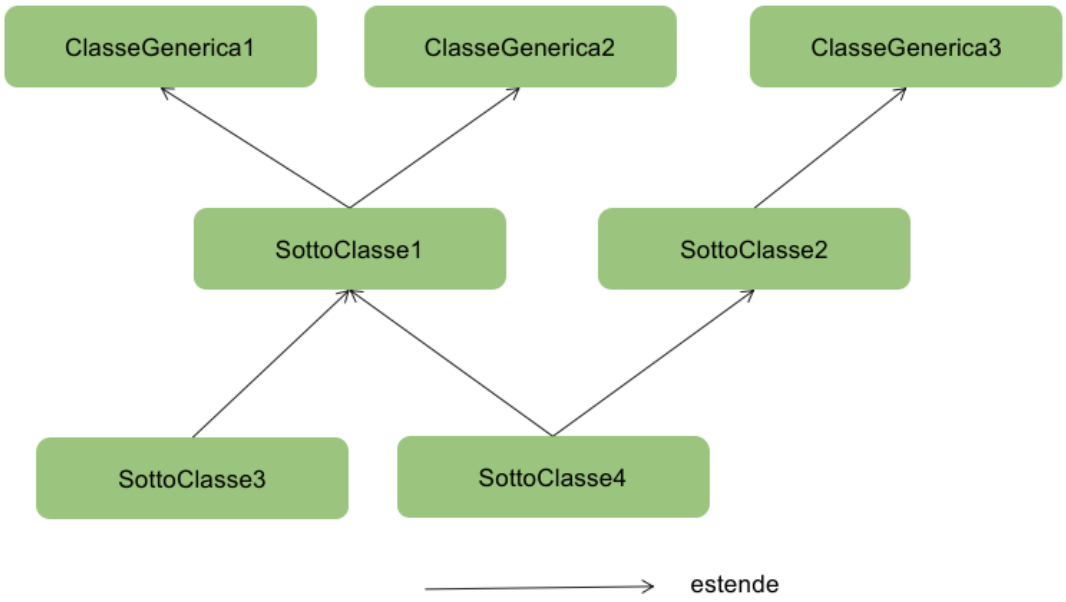


Immagine 30 Modello ad ereditarietà multipla

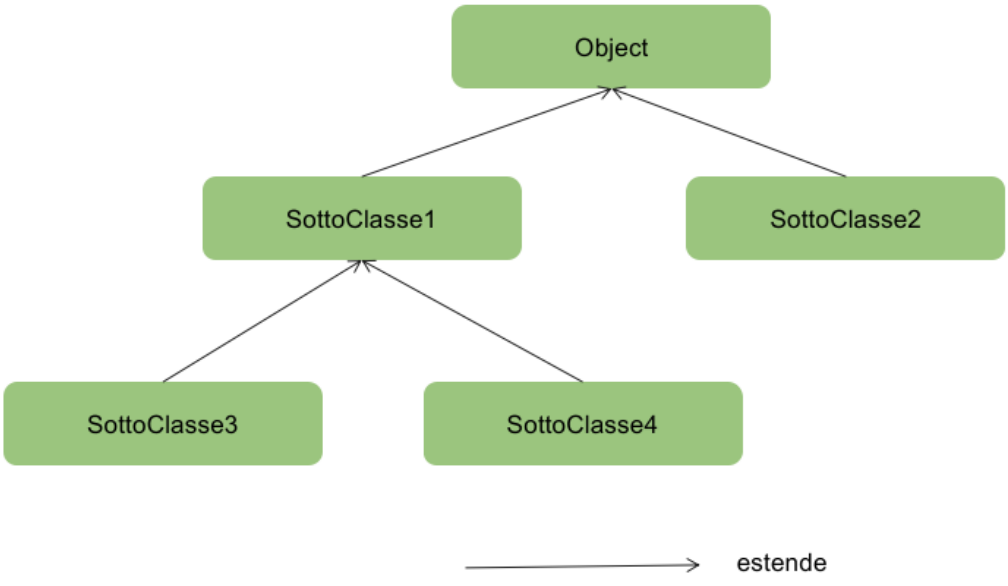


Immagine 31 Immagine 27 Modello ad ereditarietà singola

La relazione tra classe derivata e classe base è convenzionalmente denotata con una freccia piena la cui direzione va dalla classe derivata alla classe base.

Diremo quindi che:

1. Una superclasse e una sua sottoclasse sono legate da un meccanismo di ereditarietà.

2. La sottoclasse eredita da una classe base (superclasse) lo stato e il comportamento. Questo significa che la sottoclasse possiede tutti i campi e metodi della superclasse.

3. Nella sottoclasse si possono aggiungere o modificare campi e metodi per specializzare la superclasse.

Proviamo a disegnare una classe base, e per farlo pensiamo ad un veicolo generico: questo potrà muoversi, svoltare a sinistra o a destra o fermarsi. Di seguito sono riportate le definizioni della classe base *Veicolo*, e della classe contenete il metodo *main* dell'applicazione *Autista*.

```
public class Veicolo {
 public static final int DRITTO = 0;
 public static final int SINISTRA = -1;
 public static final int DESTRA = 1;

 public String tipo;
 public int velocita;
 public int direzione;

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

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

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

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

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

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

A seguire la applicazione Autista:

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

L'output della applicazione è il seguente:

```
Veicolo generico si sta movendo a: 1 Kmh
Veicolo generico ha sterzato a sinistra
Veicolo generico sta procedendo in linea retta
Veicolo generico si è fermato
```

Overload di metodi

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

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

```
public class Veicolo {
 public static final int DRITTO = 0;
 public static final int SINISTRA = -1;
 public static final int DESTRA = 1;
 public String tipo;
 public int velocita;
 public int direzione;

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

 /**
  * Simula la messa in marcia del veicolo alla velocità di 1 km/h
  */
 public void muovi() {
  muovi(1);
 }

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

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

 /**
```

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

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

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

Avendo a disposizione anche il metodo *muovi(int velocita)*, possiamo migliorare la nostra simulazione facendo in modo che il veicolo possa accelerare o decelerare ad una determinata velocità. Inoltre, abbiamo utilizzato il nuovo metodo generico per implementare il precedente attraverso una chiamata a metodo della stessa classe. Di seguito la nostra nuova applicazione Autista ed il suo output:

```
public class Autista {

 public static void main(String args[]) {
  Veicolo v = new Veicolo();
  v.muovi();
  v.muovi(10);
  v.muovi(20);
  v.svoltaSinistra();
  v.procediDiritto();
  v.ferma();
 }
}
```

```
Veicolo generico si sta movendo a: 1 Kmh
Veicolo generico si sta movendo a: 10 Kmh
Veicolo generico si sta movendo a: 20 Kmh
Veicolo generico ha sterzato a sinistra
```

Veicolo generico sta procedendo in linea retta
Veicolo generico si è fermato

L'*overloading* di metodi è possibile poiché, il nome di un metodo non definisce in maniera univoca un membro di una classe. Ciò che consente di determinare in maniera univoca quale sia il metodo correntemente chiamato di una classe Java è la sua firma (*signature*). Vale la seguente definizione:

DEFINIZIONE: si definisce firma di un metodo, il suo nome assieme alla lista dei suoi parametri formali.

Per concludere, ecco alcune linee guida per utilizzare correttamente l'*overloading* di metodi. Come conseguenza diretta della definizione precedente:

1. *non possono esistere due metodi aventi nomi e lista dei parametri formali contemporaneamente uguali.*

2. *2. i metodi di cui si è fatto l'overloading devono implementare vari aspetti di una medesima funzionalità.*

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

Overloading di costruttori

Parlando dei metodi costruttori di una classe, abbiamo detto che i metodi costruttori altro non sono che dei metodi speciali che vengono richiamati ogni qualvolta viene creata l'istanza di una classe. In quanto metodi, per i costruttori valgono le stesse regole per effettuare l'*overloading* dei metodi.

Una classe java può possedere più di un costruttore, ognuno con una firma specifica.

Se l'*overloading* di metodi serve per implementare diverse versioni di una stessa funzionalità, l'*overloading* di costruttori è utilizzato per consentire la creazione di oggetti differenziandone le proprietà a seconda della lista dei parametri formali del costruttore. Da notare che anche questa è una forma di *polimorfismo*, tema che tratteremo successivamente nel libro.

Estendere una classe base

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

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

[modificatori] ***class*** *nome* ***extends*** *nome_super_classe*

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

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

Grazie al meccanismo della ereditarietà la classe *Macchina* eredita *comportamento* ed *interfaccia* della superclasse *Veicolo*. Come appare chiaro dal codice a seguire, *Autista* è in grado di utilizzare *Macchina* esattamente come utilizzava nel caso precedente *Veicolo;* l'unico cambiamento che abbiamo dovuto apportare è quello di creare un costruttore specializzato per la nuova classe. Il nuovo costruttore semplicemente modifica il contenuto della variabile tipo affinché l'applicazione stampi i messaggi corretti.

```
public class Autista {

 public static void main(String args[]) {
  Macchina v = new Macchina();
  v.muovi();
  v.muovi(10);
  v.muovi(20);
  v.svoltaSinistra();
  v.procediDiritto();
  v.ferma();
 }
}
```

Ereditarietà ed incapsulamento

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

Modificatori ed ereditarietà	
modificatore	**accessibilità**
public	accessibile
private	non accessibile
protected	accessibile solo in alcuni casi
final	la classe non può essere usata come classe base indipendentemente dalla sua visibilità
 **sealed**	ovvero tipi sigillati: una classe **sealed** deve dichiarare quali sono le sue uniche sottoclassi

Le direttive che regolano il rapporto tra i tre modificatori e l'ereditarietà sono le seguenti:

*1. il modificatore **public** consente di dichiarare dati e metodi membro visibili e quindi utilizzabili da un'eventuale sottoclasse;*

*2. il modificatore **private** nasconde completamente dati e metodi membro dichiarati tali.*

*3. il modificatore **protected** consente l'accesso ad un metodo o dato membro di una classe:*

a) a tutte le sue sottoclassi, definite o no all'interno dello stesso package;

b) alle sole classi appartenenti allo stesso package, se riferite tramite una variabile reference.

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

*4. il modificatore **final** fa si che tutte le classi qualificate come tali non possono essere utilizzate come classe base, ovvero, nessuna nuova classe può ereditare da esse.*

Il modificatore **sealed**, introdotto in via definitiva solo a partire da Java 17 sarà trattato in maniera approfondita successivamente in questo capitolo.

Prima di concludere questo paragrafo, soffermiamoci un attimo sul modificatore **protected** al quale è dedicato il prossimo esempio in cui sono definite le due classi *Produttore* e *Consumatore* entrambe appartenenti allo stesso package: *javamattone.esercizi.capitolo9.esempio2*.

```
package javamattone.esercizi.ereditarieta.esempio2;

public class Produttore {
    protected String datoProtetto = "Questo dato e' di tipo protected";

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

```
package javamattone.esercizi.ereditarieta.esempio2;

public class Consumatore {
    private Produttore producer = new Produttore();

    void metodoConsumatore() {
        producer.stampaDatoProtetto();
    }
}
```

La definizione della classe *Produttore* contiene due membri di tipo **protected**: il primo, è una variabile di istanza tipo *String* chiamata *datoProtetto*, il secondo, un metodo, chiamato *stampaDatoProtetto* che, torna un tipo **void** e produce in output la stampa a terminale del valore del primo membro.

La classe *Consumatore*, ha un riferimento alla classe *Produttore*, tramite la variabile di istanza *producer* che utilizza per chiamare il metodo *stampaDatoProtetto*.

Poiché le due classi appartengono allo stesso package, alla classe *Consumatore* è consentito l'utilizzo del metodo **protected** *stampaDatoProtetto()* di Produttore senza che il compilatore Java segnali errori.

Analogamente, i membri **protected** della classe *Produttore* saranno visibili alla classe *Produttore2*, definita per mezzo della ereditarietà ed appartenente allo stesso package:

```
package javamattone.esercizi.ereditarieta.esempio2;

public class Produttore2 extends Produttore {
  public SottoProduttore () {
    datoProtetto = "Il valore di datoProtetto viene modificato";
  }

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

Consideriamo ora la classe *Produttore3*, definita anche questa per ereditarietà a partire dalla classe base classe base *Produttore*, ma appartenente ad un sotto-package del package corrente:

```
package javamattone.esercizi.ereditarieta.esempio2.sottopackage;

import javamattone.esercizi.capitolo9.esempio2.Produttore;

public class Produttore3 extends Produttore {
  public TerzoProduttore() {
    datoProtetto = "Il valore di datoProtetto viene ulteriormente modificato";
  }

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

In questo caso, la nuova classe potrà utilizzare il metodo *stampaDatoProtetto()* ereditato dalla superclasse, anche se non appartenente al medesimo package, ma non potrà utilizzare lo stesso metodo se chiamato direttamente come metodo membro dell'oggetto di tipo Produttore referenziato dalla variabile *producer* passata come parametro al metodo *stampaDato*.

Se provassimo a compilare la classe precedente, il compilatore produrrebbe il seguente messaggio di errore:

```
[ERROR]
/home/administrator/WORK/Progetti/javamattone/esercizi/esempi/src/main/java/javamattone/esercizi/capitolo9/esempio2/sottopackage/TerzoProduttore.java:[12,17]
stampaDatoProtetto() has protected access in
javamattone.esercizi.capitolo9.esempio2.Produttore
```

Conseguenze dell'incapsulamento nella ereditarietà

A questo punto non possiamo non soffermarci sulle conseguenze dell'incapsulamento dei dati di una superclasse nei confronti delle sue sottoclassi. Ad esempio, cosa succederebbe se sbagliassimo qualche cosa nella definizione del costruttore della classe derivata? Che impatto avrebbe sul comportamento della superclasse? Lo stato della superclasse sarebbe inizializzato correttamente?

E ancora, potremmo chiederci cosa succederebbe se nella classe base ci fossero dei dati **private** che il costruttore della classe derivata non può inizializzare od aggiornare?

Come possiamo assicurarci che lo stato di una sottoclasse, che in qualche modo dipende da quello della superclasse, sia correttamente inizializzato? Inoltre, se torniamo per un istante ad occuparci della classe base *Veicolo*, guardando attentamente la definizione del metodo costruttore vediamo immediatamente che è molto simile a quella del costruttore della classe *Macchina*. Poiché ereditarietà significa anche riuso del codice, potrebbe tornare utile riusare il costruttore della classe base per effettuare almeno una parte delle operazioni di inizializzazione.

Consideriamo ora l'esempio seguente:

```
public class Circonferenza {
  private double piGreco;
  private double raggio;

  public Circonferenza() {
    piGreco= 3.14;
    }

  public double getRaggio() {
    return raggio;
  }

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

  public double circonferenza() {
    return (2 * piGreco) * getRaggio();
  }

  public double area() {
    return (getRaggio() * getRaggio()) * piGreco;
  }

}
```

La classe base *Circonferenza*, contiene la definizione di un dato membro privato di tipo **double** che rappresenta la costante matematica Π (pi greco) e mette a disposizione due metodi che consentono di calcolare rispettivamente: la lunghezza della circonferenza e l'area del cerchio il cui raggio può essere impostato utilizzando il metodo *setter* della proprietà raggio. Cosa importante, il costruttore della classe, ha la responsabilità di inizializzare al valore corretto il dato membro p*iGreco*.

Definiamo ora la classe *Ruota* per mezzo del meccanismo di ereditarietà a partire dalla classe Circonferenza.

```
public class Ruota extends Circonferenza {
  public Ruota(double raggio) {
    setRaggio(raggio);
    //La prossima riga produce un errore di compilazione
    piGreco = 3.14;
    }

}
```

Il metodo costruttore della nuova classe, oltre ad impostare il raggio della ruota utilizzando il metodo *setRaggio* di *Circonferenza* avendolo ereditato dalla classe base, tenta di impostare il valore del dato privato *piGreco* causando un errore durante la compilazione. Per eliminare il problema possiamo riscrivere la classe *Ruota* nel modo seguente:

```
public class Ruota extends Circonferenza {
  public Ruota(double raggio) {
    setRaggio(raggio);
    }
}
```

Abbiamo omesso inizializzazione della variabile *piGreco:* l codice sicuramente non produce errori in fase di compilazione, ma potrebbe causare un effetto collateriale (*side-effect*): la variabile *piGreco* non è inizializzata correttamente.

In informatica si dice che una funzione produce un effetto collaterale quando modifica un valore o uno stato al di fuori del proprio scoping locale. Per esempio, una funzione ha un effetto collaterale quando modifica una variabile globale o statica, quando modifica uno dei suoi argomenti, quando scrive dati su di un display o su di un file o quando invoca altre funzioni con effetti collaterali. (def. Wikipedia)

In altre parole, Java applica l'incapsulamento anche a livello di ereditarietà.

Una classe base gode di tutti i benefici derivanti da un uso corretto dell'incapsulamento; d'altra parte può accadere che, proprio a causa di questo alcuni lo stato della classe base potrebbe non essere inizializzato correttamente: poiché l'interfaccia di *Circonferenza* non prevede metodi pubblici in grado di modificare il valore di p*iGreco*, l'unico modo per assegnarle il valore corretto è *utilizzare il costruttore della classe base*.

Java deve quindi poter garantire chiamate a costruttori risalendo nella catena delle classi di una gerarchia.

A dirla tutta, la JVM ci assicura che almeno un costruttore della super classe venga invocato (tipicamente il costruttore vuoto). Quindi, in questo caso nessun effetto indesiderato. Ma solo in questo caso.

Nella prossima sezione ci occuperemo in dettaglio dei meccanismi utilizzati dalla JVM per assicurare che le classi di una catena di ereditarietà siano correttamente inizializzate.

Ereditarietà e costruttori

Il meccanismo utilizzato da Java per assicurare la chiamata di un costruttore per ogni classe di una gerarchia, si basa su alcune regole di base.

1. Ogni classe deve avere un costruttore.

2. Se una definizione di classe non contiene nessun costruttore, e solo in questo caso, Java per definizione, assegnerà alla classe un costruttore vuoto e senza lista di parametri che chiameremo per comodità costruttore nullo. Il costruttore nullo viene attribuito automaticamente dal compilatore.

La due definizioni di classe si equivalgono:

```
public class ClasseVuota {
}
```

```
public class ClasseVuota {
    public ClasseVuota(){
    }
}
```

3. Se una classe è derivata da un'altra, allora può effettuare una chiamata al costruttore della classe base immediatamente precedente nella gerarchia, utilizzando la sintassi:

***super**(lista_degli_argomenti);*

dove *lista_degli_argomenti* rappresenta la lista dei parametri formali del costruttore da chiamare.

4. *Una chiamata esplicita al costruttore della classe base deve essere effettuata prima di ogni altra operazione, incluso la dichiarazione di variabili.*

Come conseguenza delle prime due regole, allora vale anche quanto segue:

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

E questo perché il costruttore vuoto viene aggiunto dalla JVM solo se nessun altro costruttore è stato definito. In caso contrario, la JVM utilizzerà i costruttori implementati senza preoccuparsi di aggiungerne uno nuovo.

Il problema della corretta inizializzazione dello stato della superclasse *Circonferenza* può essere quindi risolto nel modo seguente:

```
public class Circonferenza {
  private double piGreco;
  private double raggio;

  public Circonferenza(double raggio) {
    piGreco = 3.14;
    setRaggio(raggio);
  }

  public double getRaggio() {
    return raggio;
  }

  .....

}
```

Facendo *overriding* abbiamo aggiunto un nuovo costruttore, differente dal costruttore senza parametri formali, che prende *raggio* come parametro formale ed inizializza i due membri della superclasse *raggio* e *piGreco* garantendo la corretta inizializzazione dello stato della superclasse.

Poiché un costruttore è già presente, il compilatore Java non aggiungerà il costruttore nullo quindi, la chiamata *super()* non è consentita.

La omissione del costruttore vuoto è generalmente una strategia di programmazione per costringere altri programmatori a preoccuparsi del corretto passaggio di valori ad un oggetto già in fase di creazione dello stesso.

Basterà quindi riscrivere la classe Ruota nel modo seguente per garantirci che tutto funzioni correntemente.

```
public class Ruota extends Circonferenza {
  public Ruota(double raggio) {
    super(raggio);
    }
  public static void main(String[] args){
    Ruota ruota = new Ruota(5);
                    System.out.println("Questa ruota ha una circonferenza di:
"+ruota.circonferenza());
```

```
    }

}
```

Aggiungere nuovi metodi

Quando estendiamo una classe, lo facciamo perché stiamo specializzando una classe generica in qualcosa di più specifico. Questo processo di specializzazione della classe base prevede che si possano aggiungere nuovi metodi alla classe derivata.

Per esempio, alla classe *Macchina* potremmo aggiungere il metodo *segnala()* aggiungendo alla definizione di classe la possibilità, solo per la *Macchina*, di utilizzare un segnalatore acustico. Continueremo a mantenere tutte le vecchie funzionalità, ma ora la macchina è in grado di emettere segnali acustici.

```
public class Macchina extends Veicolo {
        public Macchina() {
          velocita = 0;
          direzione = DRITTO;
          tipo = "Macchina";
        }
        public void segnala() {
                System.out.println(tipo + "ha attivato il segnalatore acustivo ");
       }
}
```

Definire nuovi metodi all'interno di una classe derivata ci consente di definire quelle caratteristiche particolari non previste nella definizione generica del concetto, e necessarie a specializzare le nuove classi.

```
public class Autista {

    public static void main(String args[]) {
            Macchina fiat = new Macchina();
            // i prossimi metodi fanno parte dell'interfaccia ereditata da Veicolo
            fiat.muovi();
            fiat.muovi(10);
            fiat.muovi(20);
            fiat.svoltaSinistra();
            fiat.procediDiritto();
            fiat.ferma();
             // fiat.segnala è un metodo specializzato appartenente solo alla definizione
             // di Macchina
            fiat.segnala();
    }
}
```

La versione aggiornata della classe *Autista* mostra una caratteristica interessante come conseguenza dell'ereditarietà. Nell'applicazione abbiamo utilizzato il nostro oggetto *fiat* di tipo *Macchina* sia come *Veicolo* che come *Macchina*. Questo ci da lo spunto per iniziare a parlare di polimorfismo.

Polimorfismo per metodi o per dati

Nel paragrafo precedente, utilizzando il meccanismo di ereditarietà abbiamo creato la classe *Macchina* ereditando da *Veicolo* e aggiungendo successivamente un metodo per specializzarne il comportamento; quindi nell'esempio abbiamo utilizzato l'oggetto *fiat* di tipo *Macchina* nelle sue due forme: *Veicolo* prima e *Macchina* poi. Abbiamo utilizzato due diverse forme, *Macchina* e *Veicolo*, dello stesso oggetto *fiat*.

La possibilità per una classe java di assumere forme diverse a seconda del contesto è chiamata polimorfismo. Il polimorfismo, concetto cardine della programmazione Object Oriented, in java è realizzato in molteplici modi. Per il momento ci limitiamo a dire che:

DEFINIZIONE: Il polimorfismo per metodi e per dati ci consente di assegnare ad una classe diverse forme mediante ereditarietà, ovvero ereditando la forma ed il comportamento della superclasse, quindi aggiungendo metodi o modificando quelli esistenti (overriding) per specializzare forma e comportamento al nuovo contesto.

La trattazione del polimorfismo in Java non si esaurisce a questo, ma richiede un esame approfondito. Cosa che faremo nei prossimi capitoli.

Il meccanismo di *overriding* è analizzato a fondo nel prossimo paragrafo.

Overriding di metodi

Capita spesso che un metodo ereditato da una classe base non sia adeguato rispetto alla specializzazione della classe. Per ovviare al problema, Java ci consente di ridefinire il metodo originale semplicemente riscrivendo il metodo in questione, nella definizione della classe derivata.

Anche in questo caso, definendo nuovamente il metodo nella classe derivata, non si corre il pericolo di manomettere la superclasse. Il nuovo metodo sarà eseguito al posto del vecchio, anche se la chiamata fosse effettuata da un metodo ereditato dalla classe base.

Ad esempio, la definizione della classe *Macchina*, potrebbe ridefinire il metodo *muovi(int)* affinché controlli che la velocità, passata come parametro, non superi la velocità massima di 130 Km/h consentiti al tipo di mezzo.

Di seguito la nuova definizione della classe macchina:

```
public class Macchina extends Veicolo {
  public Macchina() {
    setVelocita(0);
    setDirezione(Veicolo.DRITTO);
    setTipo("Macchina");
  }

  public void segnala() {
    System.out.println(getTipo() + "ha attivato il segnalatore acustivo ");
  }

  /**
   * Simula la messa in marcia del veicolo consentendo di specificare
   * la velocità in km/h.
   * Questo metodo ridefinisce il metodo analogo
   * definito nella classse base Veicolo.
   * Per adattare il metodo al nuovo oggetto, il metodo
   * verifica che la velocità passata come argomento
   * sia al massimo pari alla velocità massima della macchina.
   @Override
    public void muovi(int velocita) {
     setVelocita(velocita<130?velocita:130);
     System.out.println(getTipo() + "si sta movendo alla velocità consentita di: "
           + getVelocita() + " Kmh");
  }

}
```

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

```
public static void main(String args[]) {
 Macchina v = new Macchina();
 v.muovi();
```

```
 v.muovi(10);
 v.muovi(20);
 v.svoltaSinistra();
 v.procediDiritto();
 v.ferma();
}
Macchinasi sta movendo alla velocit consentita di: 1 Kmh
Macchinasi sta movendo alla velocit consentita di: 10 Kmh
Macchinasi sta movendo alla velocit consentita di: 20 Kmh
Macchina ha sterzato a sinistra
Macchina sta procedendo in linea retta
Macchina si è fermato
```

DEFINIZIONE: L'override dei metodi è il meccanismo che ci permette di ridefinire un metodo di una classe più generalista adattandolo così alla classe più specializzata.

Poiché overriding significa adattare un metodo esistente ad una classe specializzata, dovremmo mantenere comunque coerenza per quanto riguarda la semantica del metodo: anche il comportamento sarà analogo a quello del metodo modificato anche se adattato ad un contesto diverso (macchina e veicolo si muovono entrambi, ma lo fanno in modo diverso).

Quando si esegue l'override di un metodo, è possibile utilizzare l'annotazione *@Override* che indica al compilatore che si intende eseguire l'override di un metodo della superclasse. Se, per qualche motivo, il compilatore rileva che il metodo non esiste in una delle super classi appartenente alla catena di ereditarietà, genererà un errore.

E se non volessimo consentire l'override di un metodo di una superclasse? Il modificatore *final* ci viene in aiuto! La regola è la seguente:

*1. se il metodo di una super classe è dichiarato **final**, non può essere modificato mediante override.*

Se il metodo muovi della classe Veicolo fosse stato dichiarato final,

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

l'override nella classe Macchina avrebbe causato un errore di compilazione.

Chiamare metodi della classe base

La variabile reference **super**, può essere utilizzata anche nel caso in cui sia necessario eseguire un metodo della superclasse, evitando che la JVM esegua il rispettivo metodo ridefinito nella classe derivata con il meccanismo di overriding. Grazie alla variabile reference **super**, il programmatore non deve necessariamente riscrivere un metodo completamente, ma è libero di utilizzare le funzionalità definite all'interno del metodo della classe base per poi passare la palla alle funzionalità definite mediante overriding.

Esaminiamo ancora la definizione della classe *Macchina* concentrando l'attenzione sulla definizione del metodo muovi(int):

```
@Override
public void muovi(int velocita) {
   setVelocita(velocita<130?velocita:130);
   System.out.println(getTipo() + "si sta movendo alla velocità consentita di: "
         + getVelocita() + " Kmh");
}
```

Se adesso lo paragoniamo al metodo definito nella sua superclasse Veicolo:

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

E' evidente che possiamo riscriverlo nel modo seguente:

```
@Override
  public void muovi(int velocita) {
     super.muovi(velocita<130?velocita:130);
     System.out.println(getTipo() + "si sta movendo alla velocità consentita di: "
     + getVelocita() + " Kmh");
  }
```

E' importante notare che, a differenza della chiamata ai metodi costruttori della superclasse effettuata facendo uso solo della parola chiave super ed eventualmente la lista dei parametri formali, ora è necessario utilizzare l'operatore ".", specificando il nome del metodo da chiamare.

In questo caso, **super** ha lo stesso significato di una variabile reference, come **this** è creato dalla JVM al momento della creazione dell'oggetto, e con la differenza che **super** fa sempre riferimento alla superclasse della classe attiva ad un determinato istante.

Compatibilità tra variabili reference

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

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

La ragione alla base di questa funzionalità è che, gli oggetti derivati hanno sicuramente almeno tutti i metodi della classe base (li hanno ereditati), e quindi non ci dovrebbero essere problemi nell'utilizzarli. Nel caso in cui un metodo sia stato ridefinito mediante overriding, questo tipo di riferimento effettuerà una chiamata al nuovo metodo. Non sarà comunque possibile invocare metodi aggiunti alla classe derivata per *overloading* (sono evidentemente sconosciuti alla classe base).

Quanto detto ci consente di definire il concetto di *compatibilità* e *compatibilità stretta* tra variabili reference.

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

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

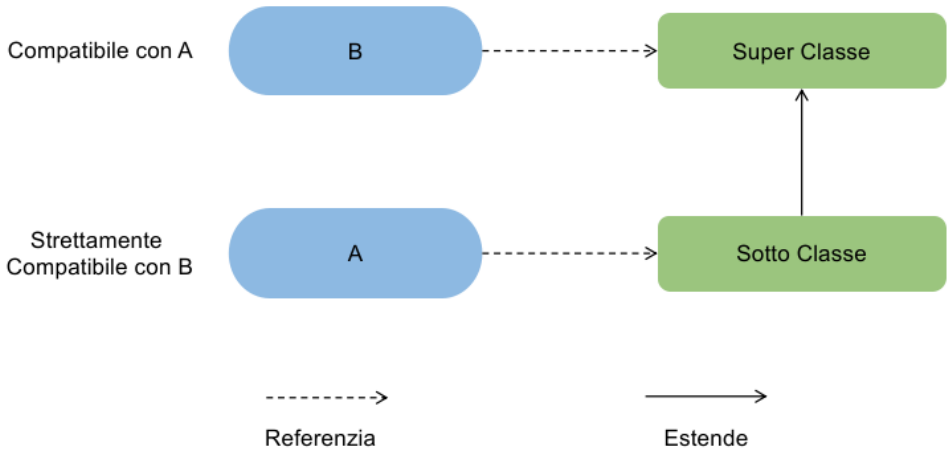


Immagine 32 compatibilità tra variabili reference

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

Viceversa, la variabile reference *veicoloGenerico* di tipo *Veicolo* è compatibile con la variabile reference *ferrari* di tipo *Macchina*.

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

Veicolo veicoloGenerico = new Veicolo();

Run-time e compile-time

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

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

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

Come conseguenza delle definizioni, il tipo a *compile-time* è sempre costante, quello a *run-time* variabile.

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

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

Veicolo veicolo = new Macchina();

Volendo fornire una definizione generale, diremo che:

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

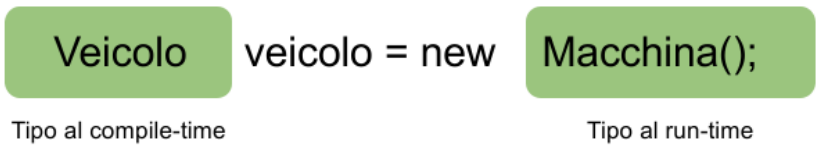


Immagine 33 tipi al run.time e compile-time

Per riassumere, ecco alcuni esempi:

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

Veicolo v = new Macchina();

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

v = new Veicolo();

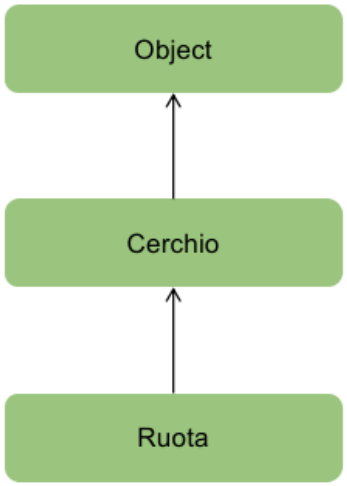
3. intero rappresenta un tipo int sia a run-time che a compile-time

```
int intero=0;
```

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

In sostanza, la creazione di un oggetto, usando l'operatore **new**, genera un'istanza di una classe che, in base al principio di ereditarietà, è anche un'istanza di tutte le sue *superclassi*. Esprimendo il concetto in modo più formale si ottiene la seguente regola:

Avendo un oggetto d e due classi B, C, se (B è sottoclasse di C) e (d è un'istanza di B) ALLORA (d è anche un'istanza di C)



Se ad esempio consideriamo la gerarchia nell'immagine, per quanto detto in precedenza potremmo dire che:

Una istanza di Ruota è anche una istanza di Cerchio e di Obect

e questo perché, grazie al meccanismo di ereditarietà, *Ruota* eredita forma e comportamento (metodi ed attributi) dalle sue superclassi. Vale anche la seguente affermazione:

Una istanza di Cerchio è anche una istanza di Object.

Al contrario:

Un'istanza della classe Cerchio non può essere considerata anche istanza della classe Ruota, perché gli oggetti della classe Ruota hanno attributi e metodi aggiuntivi rispetto alla classe Cerchio.

Accesso a metodi attraverso variabili reference

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

```
public class CompatibilitaReference {
  public static void main(String[] args) {
    Macchina fiat = new Macchina();
    Veicolo veicolo = fiat;
    fiat.segnala();
    // Questa è una chiamata illegale
    veicolo.segnala();
  }
}
```

Se proviamo a compilare il codice, il compilatore produrrà un messaggio di errore. Dal momento che, il tipo rappresentato da una variabile al *run-time* può cambiare, il compilatore assumerà che la variabile reference veicolo sta facendo riferimento all'oggetto del tipo rappresentato al compile-time (*Veicolo*).

In altre parole, anche se una classe *Macchina* possiede un metodo *segnala()*, questo non sarà utilizzabile tramite una variabile reference di tipo *Veicolo*.

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

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

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

Cast dei tipi

Java, fornisce un modo per aggirare alle limitazioni imposte dalle differenze tra tipo al *run-time* e tipo al *compile-time*, permettendo di risolvere il problema verificatosi con la applicazione *CompatibilitaReference*.

Come abbiamo già visto per i tipi primitivi, il casting è l'operazione che consente la conversione tra diversi tipi di dato predefiniti. Java permette di applicare il casting anche ad oggetti: il casting tra oggetti è l'operazione con la quale un oggetto può cambiare tipo se rispetta certe condizioni.

DEFINIZIONE: Il cast tra tipi è quindi la tecnica che consente di dichiarare alla JVM che una variabile reference temporaneamente rappresenterà un tipo differente da quello rappresentato al compile-time.

La sintassi necessaria a realizzare un'operazione di cast di tipo è la seguente:

(nuovo_tipo) identificatore

Dove *nuovo_tipo* è il tipo desiderato, e *identificatore* è una variabile reference che vogliamo convertire temporaneamente.

Riscrivendo l'esempio precedente utilizzando il meccanismo di cast, il codice verrà compilato ed eseguito correttamente :

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

L'operazione di cast è possibile su tutti i tipi purché il tipo della variabile reference ed il nuovo tipo siano almeno compatibili. Le regole sono le seguenti:

1. Il casting da una sottoclasse a una superclasse è un'operazione consentita sempre, per esempio è possibile trasformare una Ruota in un Cerchio.

2. Il casting da una superclasse a una sottoclasse è possibile solo se l'oggetto è veramente un esemplare della sottoclasse, altrimenti non è permesso.

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

Nel secondo caso, il controllo della correttezza del casting può essere fatto solo durante l'esecuzione (run-time). Se il casting non è permesso, viene segnalata l'eccezione *ClassCastException*.

Per concludere, il cast del tipo di una variabile reference, ha effetto solo sul tipo rappresentato al compile-time e non sull'oggetto in se stesso.

L'operatore instanceof

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

La sintassi formale è la seguente:

identificatore ***instanceof*** *tipo_referenziabile*

Dove *identificatore* rappresenta una variabile reference, e *tipo_referenziabile* è un tipo referenziabile. Il tipo rappresentato al run-time dalla variabile reference identificatore sarà confrontato con il tipo definito da *tipo_referenziabile*. L'operatore tornerà uno tra i due possibili valori *true* o *false*. Nel primo caso (*true*) saremo sicuri che il tipo rappresentato da identificatore consente di rappresentare il tipo rappresentato *da tipo_referenziabile*; al contrario *false* indica che identificatore fa riferimento all'oggetto **null** oppure, che non rappresenta il tipo definito da *tipo_referenziabile*.

In poche parole, se è possibile effettuare il cast di *identificatore* in *tipo_referenziabile*, **instanceof** restituirà *true*. Poiché come conseguenza della ereditarietà esiste compatibilità tra variabili reference appartenenti ad una catena gerarchica allora potremo dire che:

```
Macchina alfaRomeo= new Macchina();
// alfaRomeo instanceof Macchina = true
// alfaRomeo instanceof Veicolo = true
```

```
Veicolo veicolo = new Veicolo();
//veicolo instanceof Macchina = false
```

Prima di effettuare il cast di un oggetto sconosciuto, sarebbe prudente effettuare un controllo con **instanceof** attenzione che ci eviterà di incorrere in una *ClassCastException* in fase di esecuzione. Questo approccio è anche noto come *instanceof-and-cast*.

```
if (obj instanceof String) {
  String s = (String) obj;
  // posso usare s
}
```

A partire da Java 14 (definitivamente da java 17) l'operatore instanceof è stato aggiornato con l'aggiunta di una nuova funzione di *pattern matching* che possiamo tradurre come *corrispondenza di modello*.

Nella direzione di eliminare codice inutile e rendere il sorgente più leggibile, è stato notato che l'idioma *instanceof-and-cast* è molto comune in programmazione, di conseguenza a partire da Java 14 e definitivamente da java 17 possiamo riscrivere la porzione di codice

```
if (obj instanceof String) {
  String s = (String) obj;
  // posso usare s
}
```

nel modo seguente:

```
if (obj instanceof String s) {
  // posso usare s
}
```

Questa versione dell' operatore **instanceof** funziona come segue: se *obj* è un'istanza di *String*, viene eseguito il cast su *String* e assegnato alla variabile reference *s*. Lo scope della variabile *s* è limitato al blocco *true* dell'istruzione **if**, ma non nel blocco **else** (*false)* della stessa. Quindi:

```
if (obj instanceof String s) {
  // posso usare s
} else{
  //qui non posso usare s
}
```

Il motivo è abbastanza chiaro: se il tipo non è noto non sarò in grado di stabilire che tipo reference utilizzare per renderlo disponibile al blocco **else**.

Da notare che lo scope della variabile reference s dipende dalla semantica dell'espressione condizionale. Se infatti riscriviamo l'espressione come nel prossimo frammento di codice, le cose cambiano di conseguenza:

```
if (!(obj instanceof String s)) {
    //qui non posso usare s
} else{
    // posso usare s
}
```

Poiché lo scope della variabile reference creata da **instanceof** dipende dalla semantica del confronto, più è complicata l'istruzione **if** più complicato sarà definirne lo scope.

Un' occhio alla qualità del codice

Dal momento che lo scopo di questo libo è anche insegnare a scrivere del buon codice, prima di proseguire con questa sezione vorrei soffermarmi sul codice prodotto fino a qui: l'intento è iniziare a fornire alcuni consigli utili per scrivere un codice come lo farebbe un programmatore esperto.

In particolare prendiamo in esame la classe Veicolo.

Prima cosa concentriamoci sulla parte relativa ai dati ed in particola modo alle variabili di istanza:

```
                    ......
 * Nome del veicolo
 */
 public String tipo;
 /**
 * velocità del veicolo espressa in Km/h
 */
 public int velocita;
 /**
 * Direzione di marcia del veicolo
 */
 public int direzione;
.........
```

Essendo tutte dichiarate **public** ci siamo potuti permettere di definire il costruttore della sottoclasse macchina nel modo seguente:

```
 .......
public Macchina() {
    velocita = 0;
    direzione = DRITTO;
    tipo = "Macchina";
```

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

Inutile a dire che questo è il modo migliore per sbagliare! Una inizializzazione incontrollata dei membri di una classe che ne definiscono lo stato non andrebbe mai permessa. Un paio di consigli:

In una superclasse, ed in generale sempre, le variabili di istanza che definiscono lo stato dell'oggetto dovrebbero sempre essere dichiarate private. L'accesso e la modifica andranno gestiti attraverso opportuni metodi *getter* e *setter*.

Tutti i metodi *getter* e *setter* di una classe andrebbero scritti alla fine della definizione di classe. Alcune classi contengono decine di questi metodi che non aggiungono nulla alla semantica della classe anzi, gonfiano il codice con quello che ormai abbiamo imparato a chiamare *boiled plate code*. Per il momento quindi limitiamoci a sbatterli in fondo alla definizione della classe. Più in la vedremo come risolvere il problema in modo drastico e definitivo.

Concentriamoci adesso sul prossimo frammento di codice:

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

SINISTRA è una costante statica definita pubblica. Anche se formalmente l'accesso al suo valore è scritto correttamente, è consigliabile sempre usare la seguente regola:

L'accesso alle costanti statiche dovrebbe sempre essere effettuato in maniera statica attraverso il nome della classe, e questo perché il nome della classe aiuta a definire meglio il contesto in cui la variabile è stata definita. Nel nostro caso

SINISTRA

risulterà meno leggibile e contestualizzato di:

Veicolo.SINISTRA

Nonostante Java consenta di dichiarare variabili e costanti in qualunque punto della definizione di una classe, è buona regola dichiarare tutte le costanti all'inizio della definizione, e prima della definizione dei membri della classe.

Sulla base dei consigli dati, la classe *Veicolo* andrebbe riscritta in questo modo:

```
public class Veicolo {
  //dichiarazione delle costanti
 public static final int DRITTO = 0;
 public static final int SINISTRA = -1;
 public static final int DESTRA = 1;

//dichiarazione delle variabili di istanza
 private String tipo;
 private int velocita;
 private int direzione;

//Dichiarazione dei costruttori
 public Veicolo() {
  velocita = 0;
  direzione = Veicolo.DRITTO;
  tipo = "Veicolo generico";
 }

//Dichiarazione dei metodi membro

public void muovi() {
  muovi(1);
 }

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

 public void ferma() {
  velocita = 0;
```

```
  System.out.println(tipo + " si è fermato");
 }

 public void svoltaSinistra() {
  direzione = Veicolo.SINISTRA;
  System.out.println(tipo + " ha sterzato a sinistra");
 }

 public void svoltaDestra() {
  direzione = Veicolo.DESTRA;
  System.out.println(tipo + " ha sterzato a destra");
 }


 public void procediDiritto() {
  direzione = Veicolo.DRITTO;
  System.out.println(tipo + " sta procedendo in linea retta");
 }
//Dichiarazione dei metodi getter e setter

 public String getTipo() {
  return tipo;
 }
 public int getVelocita() {
  return velocita;
 }
 public int getDirezione() {
  return direzione;
 }
 public void setVelocita(int velocita) {
  this.velocita = velocita;
 }
 public void setTipo(String tipo) {
  this.tipo = tipo;
 }
 public void setDirezione(int direzione){
  this.direzione = direzione;
 }

}
```

Di conseguenza Macchina diventa così:

```
package javamattone.esercizi.capitolo9.esempio1;

public class Macchina extends Veicolo {
   public Macchina() {
      setVelocita(0);
      setDirezione(Veicolo.DRITTO);
      setTipo("Macchina");
   }
```

```
    public void segnala() {
      System.out.println(getTipo() + "ha attivato il segnalatore acustivo ");
    }

}
```

Classi sealed (classi sigillate)

A partire da Java 15, e quindi in via definitiva con Java 17, è stata introdotta la possibilità di definire classi **sealed** o *sigillate*. Poiché (come vedremo nei capitoli successivi) la possibilità è estesa anche alle *interfacce*, piuttosto che fare riferimento alle *classi sealed* è più corretto parlare di *tipi sealed*.

A differenza delle classi **final** che non possono essere estese, una classe **sealed** è caratterizzata dal fatto che nella sua dichiarazione ha la possibilità di specificare da quali sottoclassi deve essere estesa direttamente tramite il nuovo modificatore **sealed**, e la clausola definita dalla nuova parola contestuale **permits**.

Di seguito un esempio di classi *sealed*:

```
public sealed class SealedClass permits SealedClass1, SealedClass2{
  ....
}

public non-sealed class SealedClass1 extends SealedClass{
  ....
}

public final class SealedClass2 extends SealedClass {
  ...
}
```

Valgono le seguenti restrizioni:

*1. Tutte le classi che estendono una classe **sealed**, devono essere dichiarate **final**, **sealed** o **non-sealed**.*

L'utilizzo del modificatore **final**, come già sappiamo, implica che che la sottoclasse non potrà essere più essere estesa. Se invece anche la sottoclasse fosse dichiarata **sealed**, allora anche tale sottoclasse dovrebbe dichiarare la clausola **permits**, specificando le uniche classi che devono estenderla. Infine, se non vogliamo più utilizzare vincoli, sarà obbligatorio utilizzare il modificatore **non-sealed:** una classe **non-sealed** si comporta come una classe ordinaria (ovvero che possono quindi essere estese senza particolari vincoli);

*2. Le sottoclassi di una classe **sealed** devono essere dichiarate all'interno dello stesso package della superclasse. Nel caso stessimo scrivendo un'applicazione modulare, le sottoclassi potrebbero anche risiedere in package diversi, che però devono essere definiti all'interno dello stesso modulo;*

*3. In una classe **sealed**, bisogna sempre dichiarare la clausola **permits**, a meno che le sottoclassi vengano dichiarate nello stesso file della superclasse. In tal caso il compilatore automaticamente eleggerà le sottoclassi contenute nello stesso file, come uniche sottoclassi della classe **sealed**.*

```
// file ClassiSealedNoPermits.java


public sealed class ClassiSealedNoPermits {
  ...
}

final class ClassiSealedNoPermitsSub1 extends ClassiSealedNoPermits {
  ....
}

non-sealed class ClassiSealedNoPermitsSub2 extends ClassiSealedNoPermits {
 ...
}
```

Il modificatore **sealed** serve a limitare l'ereditarietà e di conseguenza pone un limite al polimorfismo.

Limitare l'ereditarietà, è un ottimo strumento visto che l'estensione di una classe implica una forte relazione di dipendenza tra la superclasse e la sottoclasse, e tale relazione a sua volta implica che l'evoluzione di una gerarchia di classi deve essere sempre gestita globalmente: modificare un metodo di una superclasse potrebbe modificare anche il comportamento delle sottoclassi.

Inoltre, le nostre classi potrebbero essere estese in maniera inappropriata.

Possiamo quindi utilizzare i tipi **sealed** per progettare gerarchie semplici, robuste e in contesti ben noti. La sintassi inoltre garantisce maggiori informazioni sul tipo, aggiungendo maggiore leggibilità al nostro codice e quindi migliorandone la qualità totale.

Classi immutabili

Esiste un tipo particolare di classi in java chiamate dette *immutabili*.

DEFINIZIONE: nella programmazione ad oggetti, una classe immutabile è una classe che, una volta creata, non cambia mai il suo stato.

Un esempio di classi immutabili in Java è la classe *String*. Un oggetto di tipo *String* è immutabile in quanto una volta creato non è più possibile modificarlo.

Prendiamo ad esempio il metodo *toUpperCase()* che trasforma la stringa nella forma equivalente a caratteri maiuscoli; questo metodo, come anche gli altri disponibili nella classe String non

modificano l'oggetto a cui vengono applicati ma agiscono su una copia dell'oggetto originario, restituendo appunto un nuovo oggetto con il contenuto modificato.

Un oggetto è considerato immutabile anche se gli attributi utilizzati internamente cambiano ma lo stato dell'oggetto appare invariato da un punto di vista esterno.

Le classi immutabili comportano una serie di benefici interessanti:

1. Assenza di stati non validi

Poiché l'oggetto non è modificabile, gli utenti sanno esattamente cosa aspettarsi da esso. Il codice non può essere modificato, il che significa che non è possibile introdurre incongruenze che potrebbero portare a errori di run-time. Una volta che l'oggetto è stato creato mediante il suo costruttore (che verificherà lo stato iniziale dell'oggetto) non sarà più possibile modificarlo.

2. Thread safety

Può essere condiviso tra thread senza problemi di mutua esclusione o problemi di mutazione dei dati. Non è necessaria la sincronizzazione in quanto l'oggetto non può essere modificato.

3. Codice più leggibile: è più facile progettarli ed implementarli

In generale è più conveniente utilizzare un metodo costruttore per impostare lo stato di un oggetto che i suoi metodi *setter* dopo la sua creazione, e questo perché il costruttore può essere imposto e controllato al momento della compilazione. Di conseguenza, un oggetto immutabile è anche più semplice da testare. Inoltre, poiché il codice non può essere modificato per mezzo della ereditarietà, non è possibile introdurre incongruenze che potrebbero portare a errori di run-time.

4. Sono più sicuri

Una volta creato e verificato un oggetto immutabile, nessun altro thread o processo in background sarà in grado di modificare l'oggetto senza la conoscenza diretta dell'utente. Ciò è utile per i programmi che richiedono un'elevata sicurezza.

Per creare una classe immutabile le regole sono le seguenti:

*1. Dichiarare la classe come **final** in modo che non possa essere estesa oppure rendere private il costruttore ed usare un factory method (metodo statico) per generare le istanze;*

2. Rendere private tutti le variabili di istanza in modo che l'accesso diretto non sia consentito;

3. Non fornire metodi setter per le variabili di istanza;

*4. Rendere tutti i campi **final** in modo che il valore possa essere assegnato solo una volta;*

5. Inizializzare tutti i campi tramite un costruttore;

6. Nei metodi getter di campi di tipi mutabili restituire una copia piuttosto che restituire il riferimento all'oggetto reale (java passa tutti gli oggetti per riferimento quindi è bene tornare sempre una copia dell'oggetto piuttosto che un riferimento all'oggetto originale).

Tipi record

Una critica che spesso viene mossa nei confronti di Java è che spesso questo linguaggio obbliga il programmatore a scrivere molto codice inutilmente o, meglio, a scrivere del codice che potrebbe essere evitato se si disponesse di costrutti più snelli. A partire da Java 17 sono stati introdotti i *tipi record* con cui possiamo rappresentare contenitori di dati immutabili senza creare una classe appositamente. La sintassi dei *record* aiuta gli sviluppatori a concentrarsi sulla progettazione di tali dati, senza perdersi nei dettagli implementativi.

Nel prossimo esempio utilizziamo il tipo record per creare un contenitore che rappresenta una persona fisica:

```
record PersonaFisica(
    String nome,
    String cognome,
    int eta,
    double altezza,
    double peso) {
}

public class EsempioTipoRecord {

  public static void main(String[] args) {
            // TODO Auto-generated method stub
            PersonaFisica autore = new PersonaFisica("Massimiliano",
                "Tarquini", 52, 1.82, 80.0);
            PersonaFisica autore2 = new PersonaFisica("Massimiliano",
                "Tarquini", 52, 1.82, 80.0);
            System.out.println("Il nome dell'autore è: " + autore.nome());
            System.out.println(autore);
            if(autore.equals(autore2)){
              System.out.println("I due oggetti hanno lo stesso stato");
    }

  }
}
```

Il risultato dell'esecuzione della nostra applicazione è il seguente:

```
Il nome dell'autore è: Massimiliano
PersonaFisica[nome=Massimiliano, cognome=Tarquini, eta=52, altezza=1.82, peso=80.0]
I due oggetti hanno lo stesso stato
```

PersonaFisica è stato trasformato dal compilatore in una classe che estende la classe base *java.lang.Record* e i metodi *getter* si chiameranno esattamente come i nomi delle variabili d'istanza (in questo caso nome(), cognome(), ...).

Con poche righe di codice, il compilatore definirà per noi l'equivalente classe immutabile generando automaticamente oltre ai metodi *getter*, i metodi *toString()*, *equals()*, *hashcode()* ed il costruttore (detto costruttore canonico).

Blocchi di inizializzazione ed ereditarietà

Prima di completare la sezione torniamo a parlare per un attimo dei blocchi di istanza e dei blocchi statici di inizializzazione già trattati nel paragrafo *Destinazione non trovata!.* Abbiamo detto che i blocchi statici vengono eseguiti una sola volta al caricamento della classe, mentre i blocchi di istanza vengono eseguiti ogni volta che viene creato un oggetto della classe.

Cosa succede nel caso di ereditarietà? Le regole di funzionamento sono elencate a seguire:

1. I blocchi di istanza vengono eseguiti nel costruttore della classe subito dopo la chiamata ***super*** *al costruttore della classe padre;*

```
public class ClasseBase {
  public ClasseBase() {
    System.out.println("Questo è il costruttore della classe base!");
  }
}

public class SottoClasse extends ClasseBase {
  public SottoClasse() {
    System.out.println("Costruttore della sottoclasse");
  }
  static {
    System.out.println("Blocco statico");
  }

  public static void main(String[] args) {
    SottoClasse sottoClasse= new SottoClasse();
  }
  {
    System.out.println("Blocco di istanza");
  }
}
```

Blocco statico
Questo è il costruttore della classe base!
Blocco di idtanza
Costruttore della sottoclasse

2. I blocchi statici della classe padre vengono eseguiti prima dei blocchi statici delle sottoclassi.

```
public class ClasseBase {
  public ClasseBase() {
    System.out.println("Questo è il costruttore della classe base!");
  }
   static {
    System.out.println("Blocco statico della classe base");
   }
}
```

```
public class SottoClasse extends ClasseBase{
  public SottoClasse() {
    System.out.println("Costruttore della sottoclasse");
  }
  static {
    System.out.println("Blocco statico");
  }
  public static void main(String[] args) {
    SottoClasse sottoClasse = new SottoClasse();
  }
  {
    System.out.println("Blocco di istanza");
  }
}
```

Blocco statico della classe base
Blocco statico
Questo è il costruttore della classe base!
Blocco di istanza
Costruttore della sottoclasse

10.Tipi di base

Introduzione

E' arrivato il momento di soffermarci per un attimo e parlare di alcuni tipi a cui vale la pena dedicare una sezione apposita. In questo capitolo tratteremo quindi di alcune classi speciali di utilizzo comune ed altre che sono state aggiunte nel tempo per svolgere compiti specifici, e che giocano un ruolo importante nella programmazione con Java.

La classe Object è la madre di tutte le classi in Java. Poiché Java è un linguaggio ad oggetti basato sulla ereditarietà singola, è logico pensare che in cima alla piramide di ereditarietà ci sia una classe da cui derivano tutte le classi Java. Attraverso la classe Object, tutte le classi Java ereditano alcuni metodi essenziali per garantire il funzionamento corretto del linguaggio.

Parleremo anche delle classi wrapper, ovvero quelle classi che fungono da semplici contenitori per tipi primitivi. Ne analizzeremo il funzionamento e le proprietà. Vedremo come sono evolute nel tempo per adattarsi sempre meglio alla cantieristiche di Java.

Infine tratteremo i *tipi enumerati*, tipi che possono assumere sono un numero limitato di valori costanti.

La classe Object

Come abbiamo già accennato, la gerarchia delle classi Java ha origine da una unica classe: la classe *Object*. La classe *Object* del package *java.lang* è alla radice della gerarchia di tutte le classi.

1. *Tutte le classi che non estendono esplicitamente nessuna classe sono sottoclassi dirette della classe predefinita Object.*

2. *Dato che tutte le classi derivano da Object, tutte le classi in Java ne ereditano i metodi.*

I metodi definiti nella classe *Object* sono metodi di uso comune, utilizzati spesso in autonomia dalla JVM per implementare funzionalità essenziali. Questa caratteristica degli oggetti Java, nasce principalmente allo scopo di garantire alcune funzionalità base comuni a tutte le classi, includendo la possibilità di esprimere lo stato di un oggetto in forma di stringa, la possibilità di comparare due oggetti o clonarne uno tramite il metodo ereditato *clone()*.

Il metodo *toString()*, ad esempio, è utilizzato implicitamente tutte le volte che utilizziamo la reference di un oggetto per accedere al suo stato in forma di stringa:

```
String esempio = "esempio di stringa";
System.out.println(esempio);
// equivale a
System.out.println(esempio.toString());
```

Grazie ai metodi ereditati dalla dalla classe *Object* non incorreremo mai nel rischio di vederci recapitare una eccezione a causa del fatto che un metodo potrebbe non essere stato definito.

Poiché la classe Object è la superclasse diretta o indiretta di tutte le classi, è compatibile con tutte le classi Java e quindi, una variabile di tipo Object può contenere riferimenti a oggetti di qualsiasi classe.

Comparare due oggetti: il metodo equals()

Abbiamo detto che due istanze (oggetti) di uno stesso tipo sono uguali se è uguale il loro stato al momento del confronto.

Abbiamo anche anticipato che l'operatore di booleano di uguaglianza == non è sufficiente a confrontare due oggetti poiché opera a livello di variabili reference, di conseguenza confronta due puntatori a locazioni di memoria producendo il valore *true* se e solo se le due variabili reference puntano allo stesso oggetto senza preoccuparsi di confrontare lo stato dei due oggetti.

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

Il metodo *equals()* che viene ereditato dalla classe *Object* usa semplicemente l'operatore == per confrontare due oggetti, cioè restituisce *true* solo se si tratta di due riferimenti allo stesso oggetto.

E' quindi necessario ridefinirlo in modo da effettuare un confronto più appropriato.

Nel prossimo esempio torniamo a considerare la classe *Punto.* Nella nuova versione abbiamo semplicemente aggiunto un metodo costruttore che ci consente di inizializzare lo stato dell'oggetto. Stato rappresentato ovviamente dai dati membro *x,y*.

Per leggibilità, eliminiamo i metodi *setter* inquanto poco importanti ai fini del nostro esempio.

```
public class Punto {
  int x;
  int y;
  public Punto(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }
```

```
  public static void main(String[] args) {
    Punto punto = new Punto(10,10);
    Punto punto1 = new Punto(10,10);
    System.out.println("punto è uguale a punto1? "+punto.equals(punto1));
  }
}
```

Poiché la classe non estende esplicitamente nessuna superclasse, come abbiamo detto lo erediterà implicitamente dalla classe *Object*. Il metodo ereditato tuttavia non darà i risultati attesi inquanto non tiene conto dello stato dell'oggetto. L'esecuzione del metodo *main* fornisce infatti il seguente risultato:

```
punto è uguale a punto1? false
```

Nonostante i due punti siano effettivamente lo stesso punto (x e y sono identici) il risultato darà comunque *false* in quanto le due variabili reference sono comunque distinte tra loro.

Utilizzando il meccanismo di *override* dei metodi, andremo a riscrivere il metodo *equals(Object o)* creandone una versione specializzata alla classe considerata.

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

  public Punto(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    return (o instanceof Punto && x==((Punto)o).getX() && y==((Punto)o).getY());
  }

  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public static void main(String[] args) {
    Punto punto = new Punto(10,10);
    Punto punto1 = new Punto(10,10);

    System.out.println("punto è uguale a punto1? "+punto.equals(punto1));
  }
```

}

Il risultato adesso sarà quello atteso:

punto è uguale a punto1? ***true***

Il metodo hashcode()

Come per il metodo *equals()*, il metodo *hashcode()* è ereditato nella sua implementazione di base dalla classe *Object*. Entrambi sono metodi particolari in quanto richiedono di essere ridefiniti specializzandone il comportamento alla definizione di classe corrente.

Il metodo *hashcode()* restituisce un valore *hash* che identifica l'oggetto corrente ed è utilizzato da java per fornire supporto a particolari strutture dati (@see Destinazione non trovata!) che utilizzano il codice hash per garantire un accesso immediato ad un oggetto in una collezione.

DEFINIZIONE: Una funzione hash in Java può essere definita come una funzione che restituisce un valore intero corrispondente che identifica univocamente un oggetto. La funzione hash restituisce sempre lo stesso valore intero per lo stesso oggetto.

DEFINIZIONE: Il valore intero restituito dalla funzione hash è chiamato valore hash.

Nella sua implementazione di *default*, *hashcode()* utilizza l'indirizzo dell'area di memoria dove l'oggetto è allocato mappandolo con un numero intero (univoco).

Nonostante ci si possa accontentare del funzionamento standard del metodo, esistono regole che ne definiscono il comportamento in funzione del metodo *equals()*; pertanto si è soliti ridefinire entrambi i metodi. Tali regole sono elencate a seguire:

1. Se invocato sullo stesso oggetto più di una volta durante un'esecuzione di un'applicazione, il metodo hashcode() deve restituire sempre lo stesso valore intero, a condizione che non venga modificato lo stato dell'oggetto utilizzato anche dal metodo equals().

2. Se due oggetti sono uguali secondo il metodo equals() invocare il metodo hashcode() su ciascuno dei due oggetti deve produrre lo stesso risultato intero.

3. Non è necessario che il metodo hashcode() produca risultati distinti quando invocato su oggetti che risultino non uguali secondo il metodo equals(). Tuttavia restituire hash code distinti per oggetti non uguali può migliorare le prestazioni e la affidabilità delle strutture dati basate su tali codici.

A seguire, una possibile implementazione del metodo *hashcode()* per la classe punto:

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

  public Punto(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    return (o instanceof Punto && x == ((Punto) o).getX() && y == ((Punto) o).getY());
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + x;
    result = prime * result + y;
    return result;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }
}
```

Il metodo toString()

E' l'ultimo dei metodi che analizzeremo per la classe Object. Il *toString()* restituisce la rappresentazione di stringa di un oggetto. È ampiamente utilizzato per il debug, il logging etc. etc. L'implementazione predefinita del metodo restituisce una stringa composta dal nome completo della classe, il carattere @, seguito dalla rappresentazione esadecimale senza segno del codice *hash* dell'oggetto.

Per la classe *Punto*, il metodo *toString()* ereditato da *Object* tornerà la stringa:

javamattone.esercizi.capitolo9.metodoequals.Punto@501

Ovviamente, una rappresentazione in formato stringa di un oggetto dovrebbe riportare informazioni relative al nome della classe ed allo stato dell'oggetto. Nel prossimo esempio una possibile implementazione del metodo *toString* per la classe *Punto*:

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

  public Punto(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    return (o instanceof Punto && x == ((Punto) o).getX() && y == ((Punto) o).getY());
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + x;
    result = prime * result + y;
    return result;
  }

  @Override
  public String toString() {
    return "Punto [x=" + x + ", y=" + y + "]";
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public static void main(String[] args) {
    Punto punto = new Punto(10, 10);
    System.out.println("oggetto punto corrente: " + punto);
  }
}
```

Il risultato dell'esecuzione del metodo *main* è la seguente:

```
oggetto punto corrente:  Punto [x=10, y=10]
```

Il metodo getClass()

Il metodo *getClass()* restuisce una istanza della classe *Class* che contiene informazioni sulla classe da cui è stato chiamato il metodo *getClass()* come mostrato nel prossimo esempio:

```
public class ClassTest {
  public static void main(String[] args) {
    Punto punto = new Punto(10,20);
    Class puntoAsClass = punto.getClass();

    System.out.println("simpleName: " + puntoAsClass.getSimpleName());
    System.out.println("name: " + puntoAsClass.getName());
    System.out.println("nome canonico: "+puntoAsClass.getCanonicalName());
    System.out.println("package name: "+puntoAsClass.getPackageName());
    System.out.println("Metodi della classe");

    Method[] metodiDellaClasse = puntoAsClass.getMethods();
    for(Method method : metodiDellaClasse){
      System.out.println("- " + method.getName());
      System.out.println("  ritorna: " + method.getReturnType());
    }
  }
}

name: javamattone.esercizi.tipibase.getclass.Punto
nome canonico: javamattone.esercizi.tipibase.getclass.Punto
package name: javamattone.esercizi.tipibase.getclass
Metodi della classe
- main
  ritorna: void
- equals
  ritorna: boolean
- toString
  ritorna: class java.lang.String
- hashCode
  ritorna: int
- getX
  ritorna: int
- getY
  ritorna: int
....
```

Quanto mostrato nell'esempio rappresenta solamente la punta dell'iceberg: l'oggetto *Class* consente di accedere in maniera dinamica a tutta la struttura oggetto rappresentato fino a consentire di modificare al run-time il comportamento dell'oggetto con una tecnica chiamata *reflection*.

DEFINIZIONE: La reflection è la capacità di un programma di esaminare e/o modificare a run-time il proprio comportamento, e in particolare la struttura del proprio codice sorgente.

Avete capito bene … mediante *reflection* siamo in grado di modificare al run-time il codice stesso di un oggetto istanziato iniettando all'interno nuovo codice da eseguire oppure modificandone l'interfaccia pubblica.

Tutto questo mediante le Java Reflection API a cui è dedicata una sezione di questo libro. Per il momento ci basti sapere che Le API Reflection sono raggruppata all'interno del package *java.lang.reflect*, e nessuna delle classi in esso contenute ha un costruttore pubblico (tranne la classe *java.lang.reflect.ReflectPermission*): per poter utilizzare queste classi bisogna infattu invocare degli appropriati metodi della classe *java.lang.Class* che stiamo accennando.

Class rappresenta quindi l'*entry point* per tutte le operazioni di r*eflection.*

Classi wrapper

Per ogni tipo primitivo, in java, esiste una classe corrispondente (detta *Wrapper*), il cui nome può ottenere capitalizzando il nome, tranne nel caso di *Integer* e *Character* che oltre alla prima lettera cambiano anche il nome, come mostrato nella tabella seguente:

Classi wrapper	
tipo primitivo	**classe wrapper**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

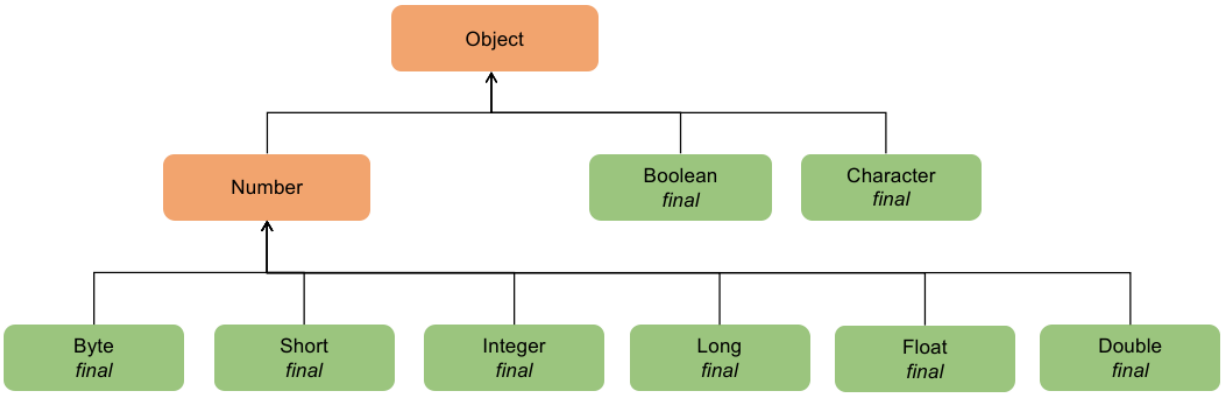


Immagine 34 Gerarchia delle classi wrapper

I wrapper sono utilizzate per rappresentare tipi primitivi come oggetti: ad esempio, come vedremo, le *Collection* java non supportano tipi primitivi ma solo tipi referenziabili. Letteralmente, sono involucri per il corrispondente tipo primitivo.

Sono disegnate per essere immutabili quindi, hanno un costruttore che consente di assegnare il valore del tipo che rappresentano e, dopo la creazione, non consentono più di modificarlo.

```
Integer wrapperIntero = new Integer(1);
Long wrapperLong = new long(1234567890);
Double wrapperDouble = new Double(1.2);
```

Come tutte le classi immutabili, hanno un metodo getter per accedere al valore rappresentato in modo protetto; inoltre, mettono a disposizione un metodo per estrarre il valore della variabile primitiva incapsulata:

1. *Byte fornisce il metodo byteValue();*
2. *Short fornisce il metodo shortValue();*
3. *Integer fornisce il metodo intValue();*
4. *Long fornisce il metodo longValue();*
5. *Double fornisce il metodo doubleValue();*
6. *Float fornisce il metodo floatValue();*
7. *Boolean fornisce il metodo booleanValue();*
8. *Character fornisce il metodo charValue();*

Già dalla versione 5, java mette a disposizione per le classi *wrapper* le funzioni di *autoboxing* e *unboxing*.

DEFINIZIONE: L'autoboxing è la conversione automatica che il compilatore Java effettua tra i tipi primitivi e le corrispondenti classi di wrapper di oggetti.

Ad esempio, convertire un *int* in un *Integer* , un *double* in un *Double* e così via.

Di seguito alcuni esempi di autoboxing:

```
Character charWrapper = 'a';
Short shortWrapper = 1;
Boolean booleanWrapper = false;
...
public void stampaWrapper(Integer a){
        System.out.println(a);
}
....
```

```
stampaWrapper(1);
```

In definitiva, il compilatore Java applica l'*autoboxing* quando un valore primitivo è:

1. Passato come parametro a un metodo che prevede un oggetto della classe wrapper corrispondente;

2. Assegnato a una variabile della classe wrapper corrispondente;

DEFINIZIONE: Al contrario, la conversione di un oggetto di un tipo wrapper (Integer) nel suo valore primitivo (int) corrispondente è chiamata unboxing.

A seguire alcuni esempi di unboxing:

```
char primitivoChar = charWrapper ;
long primitvoLong = longWrapper;
booleam primitivoBoolean = booleanWrapper;
....
public void stampaPrimitivo(short a){
System.out.println(a);
}
....
stampaPrimitivo(shortWrapper );
```

Il compilatore Java applica l'*unboxing* quando un oggetto di una classe *wrapper* è:

1. Passato come parametro a un metodo che prevede un valore del tipo primitivo corrispondente;

2. Assegnato a una variabile del tipo primitivo corrispondente;

Autoboxing e unboxing consentono agli sviluppatori di scrivere codice più pulito, facilitandone la lettura.

Enumerazioni o tipi enumerativi

Immaginiamo di voler scrivere un programma che gestisce l'agenda degli appuntamenti. E presumibile che dovremo gestire un elenco che identifica i mesi dell'anno. Un metodo sarebbe ovviamente quello di creare una classe con una serie di costanti che rappresentano i 12 mesi nel modo seguente:

```
public class MesiDellAnno {
  public static final int GENNAIO = 1;
  public static final int FEBBRAIO = 2;
  public static final int MARZO = 3;
  //....
  public static final int DICEMBRE = 12;
```

```
}
```

Da qualche parte del codice potremmo creare ad esempio la variabile intera *meseCorrente* che potrebbe rappresentare il mese correntemente gestito:

```
int meseCorrente = MesiDellAnno.GENNAIO:
```

Funzionerebbe perfettamente anche se, in quanto variabile intera, non potremmo ad esempio verificare che il valore è un valore valido (qualunque intero sarebbe accettabile), ne tanto meno aspettarci che il compilatore segnali un errore qualora il valore non sia nel range aspettato (0..12).

Le enumerazioni rispondono all'esigenza di disporre di classi che possono assumere un insieme limitato di valori costanti al compile-time. Introdotte a partire da Java 5, ricordano il tipo *enum* del linguaggio C e C++ con la differenza che non sono gestite come costanti intere, ma sono vere e proprie classi. Per questo motivo parliamo di *tipo enum.*

Grazie ad i tipi enumerativi, possiamo facilmente creare liste di costanti e riscrivere la classe *MesiDellAnno* nel modo seguente:

```
public enum MesiDellAnno {
   GENNAIO, FEBBRAIO,
   MARZO, APRILE,
   MAGGIO, GIUGNO,
   LUGLIO, AGOSTO,
   SETTEMBRE, OTTOBRE,
   NOVEMBRE, DICEMBRE

}
```

Le costanti definite in questo modo rendono il codice più leggibile, consentono il controllo in fase di compilazione ed evitano comportamenti imprevisti dovuti al passaggio di valori non validi. Come vedremo, documentano l'elenco dei valori accettati introducendo, per le costanti definite una sorta di *namespace* rappresentato dal nome della classe. Nel caso dell'esempio precedente *MesiDellAnno*.

Non solo, una variabile di un tipo enumerativo potrà prendere solamente uno dei valori previsti dalla implementazione della classe, e poiché le enumerazioni sono costanti al compile-time, il compilatore potrà controllare che il suo valore sia esattamente uno tra quelli previsti per quel tipo:

```
MesiDellAnno  meseCorrente = MesiDellAnno.GENNAIO
```

Le classi di tipo enumerazione sono create a partire della parola chiave **enum**: la sintassi è la seguente:

```
enum nome_della_enumerazione {
   valore_costante
  ,[valore_costante]

}
```

Per creare variabili di tipo enum invece la sintassi è la seguente:

```
tipo_enumerativo identificatore = tipo_enumerativo.valore
```

Al momento della compilazione, il tipo **enum** sarà trasformato dal compilatore Java in una classe che estende implicitamente dalla classe *java.lang.Enum* con una serie di metodi e membri interessanti aggiunti automaticamente dal compilatore, e che in caso di classe con costanti avremmo dovuto scrivere a mano.

Vedremo nel capitolo dedicato al polimorfismo che alle enumerazioni è comunque consentito una forma di ereditarietà attraverso le interfacce.

Per le enumerazioni valgono le seguenti proprietà e restrizioni:

1. In una classe enumerata, le prime righe devono contenere l'elenco dei valori possibili.

*2. Le classi enumerate sono automaticamente **final**.*

3. Le classi enumerate non possono estendere altre classi.

Come conseguenza diretta del fatto che sono classi che già derivano dalla classe *java.lang.Enum* e Java non consente ereditarietà multipla.

4. Sono tipi type-safe

In quanto costanti al compile-time.

*5. Tutte le costanti definite in un enum sono **public static final**.*

6. Cme tutte le classi, possono essere dichiarate all'interno della definizione di un altra classe.

I valori di una enumerazione sono costanti in fase di compilazione (compile-time), il che significa che non è possibile modificarne i valori. Qualsiasi assegnazione comporterà un errore in fase di compilazione. Poiché sono dichiarate **static,** le costanti di una enumerazione possono essere accedute in forma statica utilizzando il nome della classe:

```
MesiDellAnno  meseCorrente = MesiDellAnno.GENNAIO
```

*7. Non è possibile creare una istanza di un tipo **enum.***

I tipi **enum** assicurano che solo una istanza delle variabili possa esistere al run-time nella JVM

Utilizzare una enumerazione

Adesso che abbiamo capito come creare una enumerazione, vediamo come utilizzarle. Per farlo modifichiamo l'esempio visto nel capitolo precedente:

```
public class Agenda {
  private MesiDellAnno currentMonth;

  public enum MesiDellAnno {
    GENNAIO, FEBBRAIO, MARZO, APRILE,  MAGGIO, GIUGNO,
    LUGLIO, AGOSTO,  SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBR
  }
  public boolean isGennaio(){
    return currentMonth == MesiDellAnno.GENNAIO;
  }
}
```

Prima cosa che notiamo è che possiamo utilizzare l'operatore == per comparare due tipi enumerativi. L'operatore di uguaglianza garantisce la sicurezza rispetto ai tipi (type safety) sia al run-time che al compile-time. Proviamo a comprendere meglio il significato di questa affermazione concentrandoci, prima di tutto, sulla sicurezza di tipi al run-time.

Il metodo *isGennaio* proposto nell'esempio precedente potrebbe essere scritto in due modi distinti entrambi validi dal momento che un tipo **enum** comunque è una sottoclasse di *Object*.

```
public boolean isGennaio(){
    return currentMonth == MesiDellAnno.GENNAIO;
  }


public boolean isGennaio(){
    return currentMonth.equals( MesiDellAnno.GENNAIO);
  }
```

La seconda versione del metodo però potrebbe causare una eccezione di tipo *NullPointerException* dal momento che la variabile *currentMonth* potrebbe valere **null** (). Al contrario, la prima versione del metodo non genererà mai nessuna eccezione.

Parlando di sicurezza dei tipi al compile-time consideriamo il prossimo frammento di codice in cui in cui *Colori* è un tipo **enum** diverso da *MesiDellAnno.*

```
private MesiDellAnno currentMonth;
//produce un errore al compile time
currentMonth == Colori.VERDE;
```

```
//non produce errori al compile time tornando semplicemente false
currentMonth.equals(Colori.VERDE);
```

Nel primo caso il compilatore segnalerà un errore di comparabilità di tipo garantendo sicurezza dei tipi al contrario, nel secondo caso compilerà correttamente senza segnalare l'eventuale anomalia.

Per ottenere l'elenco di tutti i i possibili valori di un tipo **enum**, possiamo utilizzare il metodo *value()*, che restituisce un array di tutte le costanti contenute nella enumerazione. Questo metodo è utile quando, ad esempio, si vogliono scorrere tutte le costanti all'interno di un ciclo *for*.

```
for(MesiDellAnno mese : MesiDellAnno.values()){
    if(mese==meseDellAnno)
       System.out.println(mese);
}
```

Infine, possono essere utilizzare insieme al costrutto **switch**:

```
public class StampaMesiDellAnno {
  public enum MesiDellAnno {
  .......
  public static void main(String[] args) {
    MesiDellAnno meseDellAnno = MesiDellAnno.GENNAIO;
    switch (meseDellAnno) {
      case GENNAIO:
        System.out.println("Gennaio");
        break;
      case FEBBRAIO:
        System.out.println("Febbraio");
        break;
      case MARZO:
        System.out.println("Marzo");
        break;
       .......
      default:
        System.out.println(meseDellAnno);
    }
  }
}
```

Aggiungere costruttori, metodi e campi ad una enumerazione java

Le enumerazioni java sono tipi estremamente flessibili. Una classe **enum** può contenere campi, metodi e costruttori come una classe normale, ma con alcune restrizioni sui costruttori:

1. tutti i costruttori devono avere visibilità privata o default (package friendly);

*2. non è possibile invocarli esplicitamente con **new**, neanche all'interno della classe stessa.*

Adesso, supponiamo di voler distinguere i mesi tra mesi invernali e non. Possiamo modificare la nostra enumerazione come segue:

```
public enum MesiDellAnno {
  GENNAIO(true), FEBBRAIO(true), MARZO(true), APRILE(false),
  MAGGIO(false), GIUGNO(false),
  LUGLIO(false), AGOSTO(false),  SETTEMBRE(false), OTTOBRE(false),
  NOVEMBRE(true), DICEMBRE(true);

  private final boolean invernale;

  private MesiDellAnno(boolean invernale){
    this.invernale = invernale;
  }
  public boolean isIvernale(){
    return invernale;
  }
}
```

Potremmo ad esempio creare una applicazione per stampare solo i mesi invernali:

```
public class StampaMesiDellAnnoCiclo {
  public static void main(String[] args) {
    MesiDellAnno meseDellAnno = MesiDellAnno.GENNAIO;
    for(MesiDellAnno mese : MesiDellAnno.values()){
      if(mese.isIvernale())
        System.out.println(mese);
    }
  }
}
```

Dal momento che i tipi enumerati sono delle costanti **static final**, la convenzione vuole che i nome dei tipi enumerati siano scritti come le costanti: maiuscolo eventualmente separati dal carattere '_' underscore.

Oppure potremmo ottenere lo stesso risultato nel modo seguente:

```
public enum MesiDellAnno {
  GENNAIO{
    @Override
    public boolean isIvernale() {
      return true;
    }
  },
  FEBBRAIO{
    @Override
    public boolean isIvernale() {
      return true;
    }
  },
...
...
  GIUGNO{
    @Override
    public boolean isIvernale() {
      return false;
    }
  };
...
public boolean isIvernale() {
    return true;
  }

}
```

11. Stringhe

Introduzione

A differenza del linguaggio C, in cui una stringa è gestita mediante array di caratteri terminati dal valore *null*, Java rappresenta le stringhe come oggetti, e come tali dotati di tutte le caratteristiche previste.

String è sicuramente la classe più utilizzata in Java pertanto negli anni ha attirato a se l'attenzione della comunità più di una volta. Rispetto alla prima versione di questo libro, oggi le cose sono cambiate tanto da valere la pena dedicare a questa classe un capitolo intero. Con l'evoluzione del linguaggio java, si sta cercando di rendere le stringhe più efficienti e meno verbose.

E' una classe dalle molteplici proprietà: oltre ad essere una classe immutabile, possiede una sintassi semplificata per la creazione di una istanza. La classe *String* possiede due cugini mutabili: *java.lang.StringBuilder* e *java.lang.StringBuffer*.

In quanto oggetti immutabili, le stringhe godono inoltre di particolari proprietà per la gestione della memoria. Infatti, la gestione della memoria di questi oggetti *String* è caratterizzata dal riuso delle istanze già create mediante un'apposita *pool di stringhe*, uno spazio di memoria appartenente allo *heap-space*.

La classe *String* fornisce tutti i metodi necessari a manipolare stringhe: un oggetto stringa può essere alterato, concatenato, possiamo ricercare all'interno di una stringa oppure sostituirne una porzione.

Infine, in questa sezione introdurremo i *text block*. I *text block*, introdotti in via sperimentale a partire da Java 13, consentono di manipolare testi multi linea in maniera semplificata rispetto alla sintassi classica delle stringhe.

String Literals

Prima di procedere, fermiamoci per un secondo per definire gli *string literals*. In Java gli *string literals* sono sequenze di 0 o più caratteri delimitati dal carattere " (doppio apice).

Alcuni esempi di string literals sono i seguenti:

```
"questo è uno string literal valido"
"" //questo è uno string literal vuoto
"ciao" //questo è uno string literal composto da 4 caratteri
```

Gli *string literal* possono essere concatenati tra loro mediante l'operatore + (concatenazione di stringhe)

```
"questo è uno string "+"literal valido"
```

e possono essere rappresentati su più righe soltanto mediante l'operatore di concatenazione.

```
"questo è uno string "+"literal valido. Utilizzando "
+"l'operatore concatenazione posso "
+"dividerlo su più righe."
```

Creare una stringa

La classe *String* mette a disposizione diversi costruttori per creare stringhe utilizzando l'operatore **new**. In quanto oggetti immutabili, una volta creati non possono essere modificati direttamente, e manterranno sempre il valore assegnatogli inizialmente. Il più semplice è il costruttore vuoto:

```
String prima = new String(); //questa è una stringa vuota
```

Per creare una stringa possiamo anche utilizzare *string literals*, oppure passare come parametro al costruttore apposito un'altro oggetto *String*:

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

In alternativa, Java consente al programmatore di assegnare ad una variabile reference di tipo *String* uno *string literal* direttamente

```
String prima = "Questa è una stringa creata da uno string literal";
String prima = "Questa è una stringa "+"creata da uno string literal";
```

Così come per gli *string literals*, è possibile concatenare due stringhe mediante le loro variabili reference con l'operatore di concatenazione +:

```
String stringa1 = "Hello";
String stringa2 = "World"
String helloWorld = stringa1+" "+stringa2;
```

Gli string literals in Java sono essi stessi oggetti di tipo *String*. Ogni volta che il compilatore Java ne incontra uno, lo sostituisce con una variabile reference. Se consideriamo ad esempio il seguente frammento di codice:

```
String stringa1 = "Hello";
```

Il compilatore, al momento della compilazione del codice sostituisce la stringa "Hello" con una variabile reference ad oggetto di tipo *String*. Quindi, uno string literal può essere utilizzato come una variabile reference ad una stringa avente come contenuto lo string literal stesso. Il prossimo frammento di codice utilizza il metodo length() della classe *String* per calcolare la lunghezza di string literals.

```
int lunghezzaStringaVuota= "".length();    // lunghezzaStringaVuota = 0
```

```
int helloString=  "Hello".length(); // helloString = 5
```

La natura di oggetti immutabili causa a volte confusione nei programmatori di primo pelo. Consideriamo il seguente frammento di codice:

```
String stringa;
stringa = "prima stringa";
stringa = "seconda stringa"
```

e' evidente che la stringa nel passaggio è mutata, tuttavia anche se la variabile reference può mutare, il contenuto in memoria collegata alla variabile reference rimane immutabile. Utilizzando il modificatore **final** possiamo rendere immutabile anche la variabile reference:

```
final String stringa = "prima stringa";
stringa = "seconda stringa" //questa istruzione non è consentita
```

La lunghezza di una stringa non è altro che il numero di caratteri che sono contenuti nella stringa. La classe *String* ha un metodo integrato, *length()*, che torna il numero di caratteri di qualsiasi stringa. Poiché anche uno string literal è un oggetto di tipo *String* possiamo tranquillamente utilizzare il metodo *length()* nel modo seguente:

```
"La lunghezza di questa stringa è ".length();
```

Pool di stringhe

Grazie al fatto di essere oggetti immutabili, le stringhe in Java godono inoltre di particolari proprietà per la gestione della memoria: la JVM è difatti in grado di ottimizzare la quantità di memoria allocata utilizzando un pool di stringhe in cui memorizzare solo una copia di ogni string literal pronto per essere riutilizzato più volte. Questo processo è chiamati *interning*. Andiamo adesso ad analizzare i diversi casi per meglio comprenderne il funzionamento del meccanismo di *interning*. Iniziamo dal primo: una string creata mediante operatore **new.**

```
String prima = new String("str");
```

In questo caso, la JVM si comporterà come segue:

1. Cerca nel pool di stringhe se esiste già uno string literal uguale;

2. Se non lo trova, crea un nuovo oggetto String con valore "str" e lo aggiunge allo string pool creando una nuova variabile reference per lo string literal;

3. Lo string literal "str" nel codice sarà rimpiazzato dalla variabile reference appena creata;

*4. Poiché stiamo usando l'operatore **new**, la JVM creerà una nuovo oggetto nello heap-space*

L'algoritmo è schematizzato nella prossima immagine

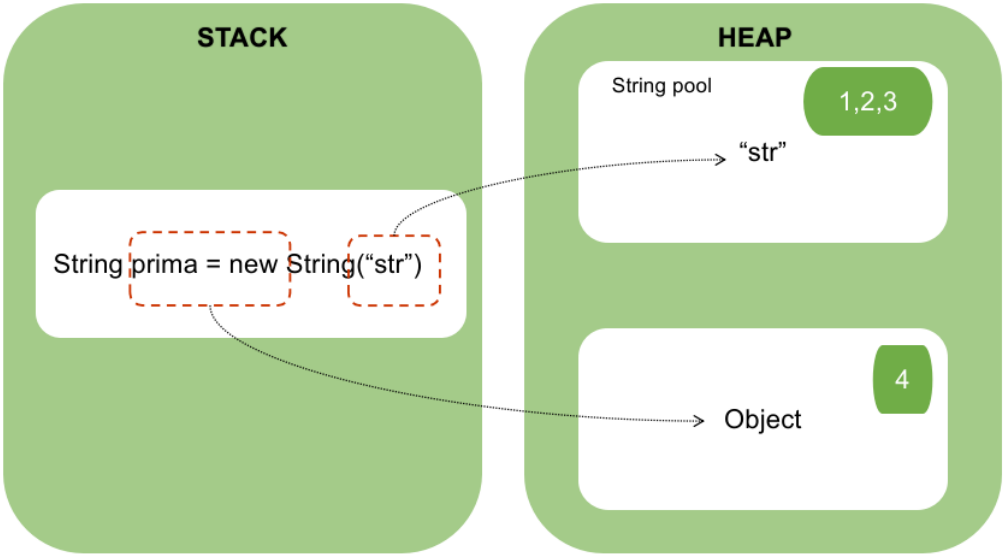


Immagine 35 - gestione dello string pool caso 1

Consideriamo ora il nuovo scenario:

String prima = new String("str");
String seconda= new String("str");

Rispetto al caso precedente, per creare la seconda stringa riutilizzerà lo string literal già presente nello string pool. Dovrà comunque allocare un nuovo oggetto *String* nello heap-space.

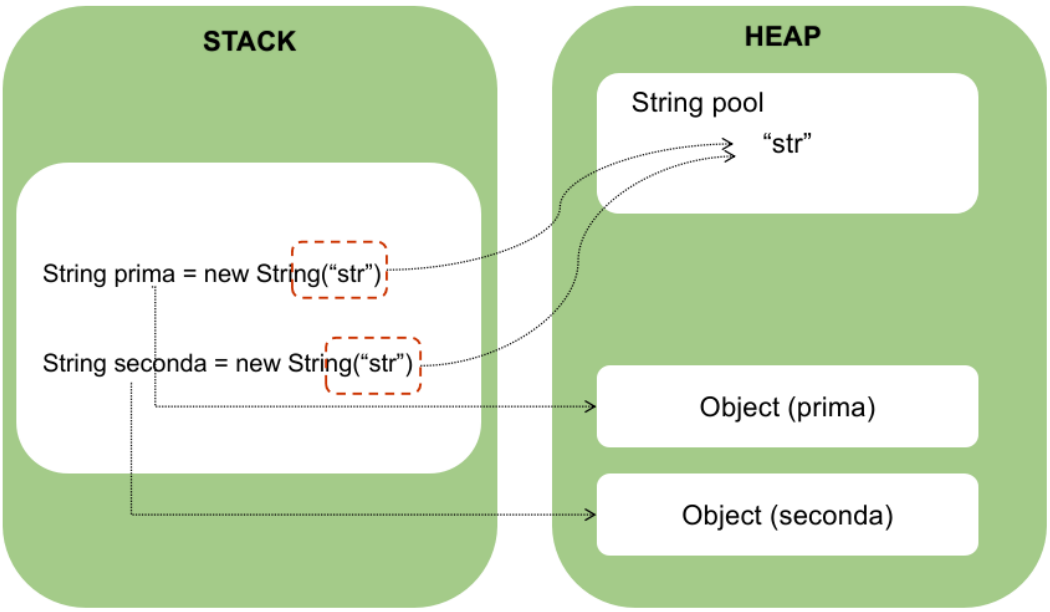


Immagine 36 Gestione dello string pool caso 2

Nel primo caso la JVM ha dovuto creare due variabili reference, nel secondo caso tre. Se inoltre volessimo confrontare le due variabili reference prima e seconda con l'operatore di confronto == come aspettato il risultato sarebbe false: ho due oggetti distinti nello *heap-space* che stanno utilizzando lo stesso contenuto in memoria nello *string pool*.

```
public class Esempio1 {
  public static void main(String[] args) {
    String prima = new String("str");
    String seconda = new String("str");
    System.out.println("Il risultato della comparazione è:"+(prima==seconda));
  }
}

Il risultato della comparazione è:false
```

Cosa succede nel caso in cui volessimo creare stringhe utilizzando direttamente *string literal* come un oggetto? Consideriamo la seguente applicazione di esempio:

```
public class Esempio2 {
  public static void main(String[] args) {

    String prima = "Hello world";
    String seconda = "Hello world";
    String terza = "Hello"+" world";
    final String quarta = "Hello"+" world";
    String quinta = new String(prima);

    System.out.println("Il risultato della comparazione tra prima e seconda è:"+(prima==seconda));
    System.out.println("Il risultato della comparazione tra prima e terza è:"+(prima==terza));
    System.out.println("Il risultato della comparazione tra prima e quarta è:"+(prima==quarta));
    System.out.println("Il risultato della comparazione tra prima e quinta è:"+(prima==quinta));
  }

}

Il risultato della comparazione tra prima e seconda è:true
Il risultato della comparazione tra prima e terza è:true
Il risultato della comparazione tra prima e quarta è:true
Il risultato della comparazione tra prima e quinta è:false
```

1. Non appena la JVM incontra il primo string literal memorizza il contenuto nel pool di stringhe creando una variabile reference associandone a ***prima*** *il valore. Non utilizzando l'operatore* ***new*** *null'altro verrà creato nello heap.*

*2. Assegna la stessa variabile reference a **seconda**. Non utilizzando l'operatore **new** null'altro verrà creato nello heap;*

*3. Esegue il lato destro dell'espressione (concatenazione di stringhe). Poiché il risultato è già presente nel string pool associa la stessa reference anche in **terza**. Non utilizzando l'operatore **new** null'altro verrà creato nello heap-space;*

4. Stessa cosa per quarta.

*5. Lo string literal già esiste: la JVM passa la variabile reference al costruttore dell'oggetto String e, poiché stiamo utilizzando l'operatore **new** un nuovo oggetto sarà creato di conseguenza nello heap-space.*

Ricordando quanto detto all'inizio del libro riguardo all'operatore == se utilizzato con variabili reference, notiamo subito che prima, seconda, terza, quarta hanno memorizzato tutte lo stesso puntatore alla locazione in memoria. Il confronto restituirà quindi il valore *true*. La variabile reference quinta, come nel caso precedente, rappresenta un nuovo oggetto come conseguenza dell'uso di **new**. Lo avreste mai detto?

Generalmente l'uso di *string literals* è più efficiente che l'utilizzo dell'operatore **new** in quanto riduciamo il numero di operazioni che la JVM deve compiere quando vengono create nuove stringhe. Inoltre, poiché uno *string literal* è una costante al tempo della compilazione (inclusi quelli come risultato della concatenazione di altri *string literals*) di conseguenza viene immediatamente aggiunto allo *string pool* rendendo ancora più efficiente la gestione delle stringhe.

Nei prossimi paragrafi elenchiamo alcune delle funzioni che la classe *String* ci mette a disposizione per modificare le stringhe. E' bene ricordare che in quanto immutabile, tutti i metodi della classe String non modificano mai il valore rappresentato dalla stringa, bensì ne tornano una nuova.

Concatenazione di stringhe

Concatenare due stringhe significa unire diverse sequenze di caratteri in un'unica entità. Abbiamo già accennato più volte all'operatore di concatenazione, tuttavia non tutto è stato detto in quanto questo operatore possiede alcune caratteristiche interessanti come è possibile vedere subito nel prossimo frammento di codice:

```
public class Concatenazione {

  public static void main(String[] args) {
    String stringaConInt = ""+5;
    String stringConDouble = ""+1.2;
    String stringconBoolean = ""+true;

    Integer integer = 1;
    String stringaConInteger = ""+5;
  }
```

```
}
```

L'operatore di concatenazione + infatti non solo concatena due stringhe, ma trasforma implicitamente un tipo in una stringa prima di effettuarne la concatenazione.

Oltre all'operatore di concatenazione +, la classe *String* mette a disposizione il metodo *concat* che prende come parametro formale una stringa e restituisce una nuova stringa ottenuta dalla concatenazione delle due stringhe: quella rappresentata dalla variabile reference e quella ottenuta in input. Nel prossimo frammento di codice un esempio di utilizzo del metodo *concat*.

```
public class DemoConcatenazione {
  public static void main(String[] args) {
    String stringaIniziale = "String iniziale";
    String stringaConcatenata = stringaIniziale.concat(" + un altro pezzetto");

    System.out.println("La string iniziale è: "+stringaIniziale);
    System.out.println("La string concatenata è: "+stringaConcatenata);

  }
}

La string iniziale è: String iniziale
La string concatenata è: String iniziale + un altro pezzetto
```

Come aspettato, la stringa iniziale non è stata modificata dalla operazione di concatenazione.

Trasformazione di stringhe

Esistono diverse trasformazioni possibili su una stringa: conversione MAIUSCOLO/minuscolo, eliminazione degli spazi iniziali, sostituzione di sotto stringhe e tenti altri. E' bene ribadire che il risultato dei metodi a seguire devono essere memorizzati con nuove variabili reference a meno di non poterli utilizzare per via del fatto che la stringa iniziale non viene mai modificata. Per una descrizione completa dei metodi disponibili per le stringhe in Java è bene comunque consultare la documentazione ufficiale.

I metodi *toLowerCase()* e *toUpperCase()* ritornano rispettivamente una nuova stringa i cui caratteri sono convertiti tutti in forma minuscola oppure maiuscola.

```
String stringaIniziale = "Questa è Una stringa.";
String stringaMinuscola = stringaIniziale.toLowerCase();
String stringaMaiuscola = stringaIniziale.toUpperCase();
```

Il metodo metodo *trim()* elimina gli spazi bianchi all'inizio ed alla fine della stringa.

```
String stringaIniziale = " Questa è Una stringa. ";
String stringaTrimmed = stringaIniziale.trim();
```

Per la sostituzione di sotto stringhe esistono svariati metodi che ben si adattano a diversi possibili casi d'uso.

Il metodo *replaceAll(String regex, String replacement)* utilizza una espressione regolare per identificare tutte le possibili sotto stringhe e quindi sostituirle con la stringa identificata da *replacement*. Ad esempio, volendo rimuove tutti i caratteri ' ' (spazio) da una stringa possiamo utilizzare il seguente frammento di codice:

```
String stringaConSpazi = " a b c d  e f g h i"
String stringaSenzaSpazi = str.replaceAll(" ",""));
System.out.println(stringaSenzaSpazi );
...
Output: abcdefghi
```

Analogamente, i metodi *replaceFirst(String regex, String replacement)* e *replaceLast(String regex, String replacement)* sostituisco, rispettivamente, la prima e l'ultima occorrenza della stringa identificata dalla espressione regolare.

Un'espressione regolare è un modello, scritto in un opportuno linguaggio, attraverso il quale una parola, o frase, può essere cercata all'interno di un testo, oppure validata per conformarsi ad un certo formato.

A titolo di esempio, l'espressione regolare ^[a-zA-Z0-9]+$ rappresenta una qualsiasi stringa alfanumerica contenente caratteri dell'alfabeto maiuscoli e minuscoli ed i numeri da 0 a 9.

Text block

Java 17 implementa in via definitiva nuova caratteristica chiamata *text block* che permette di utilizzare la classe *String* in maniera più proficua e più semplice. Tale caratteristica permette alle stringhe di essere definite su più linee utilizzando una nuova sintassi.

La sintassi multi-linea *dei text block* è più naturale rispetto al passato in cui era necessario ricorrere continuamente a concatenazioni di stringhe, a caratteri di escape come \n, e ad una complessa gestione delle virgolette e delle spaziature. Con la nuova sintassi il codice è meno verboso, più leggibile e, cosa non da poco, più semplice da scrivere. Consideriamo infatti il prossimo esempio:

```
String htmlFile = "<HTML>\n" +
                  "  <BODY>\n" +
                  "    <H1 style=\"color: blue;\">Hello World!</H1>\n" +
                  "  </BODY>\n" +
                  "</HTML>";
```

e proviamo a riscriverlo utilizzando la sintassi *text blocks*:

```
String htmlFile = """
        <HTML>
         <BODY>
          <H1 style="color: blue;">Hello World!</H1>
         </BODY>
        </HTML>""";
```

Appare immediatamente evidente quanto possa risultare vantaggioso utilizzare i *text blocks* piuttosto che la sintassi classica di Java per le stringhe.

Come si vede nell'esempio, il *text block* è delimitato da un *delimitatore di apertura* ed un *delimitatore di chiusura* entrambi rappresentati da: **"""** (*sequenza di tre doppi apici*).

In particolare valgono le regole seguenti:

1. Il delimitatore di apertura (in inglese opening delimiter) è definito da tre virgolette, seguita da zero o più spazi ed un terminatore di linea (ovvero, l'andare da capo). Il contenuto del text block parte dal primo carattere dopo la terminazione di linea. Gli eventuali spazi bianchi compresi tra le tre virgolette e la terminazione di linea non sono presi in considerazione.

2. Il delimitatore di chiusura (in inglese closing delimiter) invece, è definito solo da tre virgolette. Il contenuto del text block, termina con il carattere che precede la prima virgoletta del delimitatore di chiusura.

Una volta compilato, un *text block* diventa quindi uno *string literal* a tutti gli effetti, ed al *run-time* verrà immagazzinato nella pool di stringhe come descritto nei paragrafi precedenti.

12.Eccezioni

Introduzione

Le eccezioni Java sono utilizzate in quelle situazioni in cui sia necessario gestire condizioni anomale, ed i normali meccanismi si dimostrano insufficienti ad indicare completamente una condizione di errore o, un'eventuale anomalia.

Formalmente, un'eccezione è un evento che si scatena durante l'esecuzione di un programma, causando l'interruzione del normale flusso delle operazioni.

Queste condizioni di errore possono svilupparsi in seguito ad una gran varietà di situazioni anomale: il malfunzionamento fisico di un dispositivo di sistema, la mancata inizializzazione di oggetti particolari quali ad esempio connessioni verso basi dati, o semplicemente errori di programmazione come la divisione per zero di un intero.

Tutti questi eventi hanno la caratteristica comune di causare l'interruzione dell'esecuzione del metodo corrente.

Il linguaggio Java, cerca di risolvere alcuni di questi problemi al momento della compilazione del codice sorgente, tentando di prevenire ambiguità che potrebbero essere possibili cause di errori, ma non è in grado di gestire situazioni complesse o indipendenti da eventuali errori di scrittura del codice. Queste situazioni sono molto frequenti e spesso sono legate ai costruttori di classe.

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

Oltre a quanto menzionato, esistono casi particolari in cui le eccezioni facilitano la vita al programmatore fornendo un meccanismo flessibile per descrivere eventi che, in mancanza delle quali, risulterebbero difficilmente gestibili.

Torniamo ancora una volta a prendere in considerazione la classe Pila. Dal momento che il metodo *push()* non prevede parametri di ritorno, è necessario un meccanismo alternativo per gestire un eventuale errore causato da un trabocco dell'array.

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

```
public class Pila {
  private int[] dati;
  private int cima;
  private int dimensioneMassima;

  public Pila() {
    this(10);
  }

  public Pila(int capacitaMassimaDellaPila) {
    dimensioneMassima = 10;
    dati = new int[dimensioneMassima];
    cima = 0;
  }
  public void push(int dato) {
    if (cima < dimensioneMassima) {
      dati[cima] = dato;
      cima++;
    }
  }
  public int pop() {
    if (cima > 0) {
      cima--;
      return dati[cima];
    }
    return 0; // Bisogna tornare qualcosa
  }
}
```

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

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

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

Propagazione di oggetti

Il punto di forza delle eccezioni consiste nel permettere la propagazione di un oggetto a ritroso, attraverso la sequenza corrente di chiamate tra metodi. Opzionalmente, ogni metodo può:

1. Catturare l'oggetto per gestire la condizione di errore utilizzando le informazioni trasportate, terminandone la propagazione;

2. Prolungare la propagazione ai metodi subito adiacenti nella sequenza di chiamate.

Ogni metodo, che non sia in grado di gestire l'eccezione, è interrotto nel punto in cui aveva chiamato il metodo che sta propagando l'errore. Se la propagazione di un'eccezione raggiunge il metodo *main* dell'applicazione senza essere arrestata, l'applicazione termina in maniera incontrollata. Consideriamo l'esempio seguente:

```
class Eccezioni{
        double metodo1(){
                double d;
                d=4.0 / metodo2();
                System.out.println(d) ;
        }
        float metodo2(){
                float f ;
                f = metodo3();
                //Le prossime righe di codice non vengono eseguite
                //se il metodo 3 fallisce
                f = f*f;
                return f;
        }
                int metodo3(){
                if(condizione)
                return espressione ;
                else
                // genera una eccezione e propaga l'oggetto a ritroso
                // al metodo2()
        }
}
```

Questo pseudo codice Java, rappresenta una classe formata da tre metodi: *metodo1()* che restituisce un tipo **double** il cui valore è determinato sulla base di quello restituito da *metodo2()* di tipo **float**. A sua volta, *metodo2()* restituisce un valore **float** calcolato in base a quello di ritorno del *metodo3()* che, sotto determinate condizioni, genera un'eccezione.

L'esecuzione del metodo1() genera quindi la sequenza di chiamate schematizzato nella prossima figura. Se si verificano le condizioni a seguito delle quali *metodo3()* genera l'eccezione, l'esecuzione del metodo corrente si blocca e l'eccezione viene propagata a ritroso verso *metodo2()* e *metodo1()*.

Una volta propagata, l'eccezione deve essere intercettata e gestita. In caso contrario si propagherà sino al metodo *main* dell'applicazione causando la terminazione dell'applicazione.

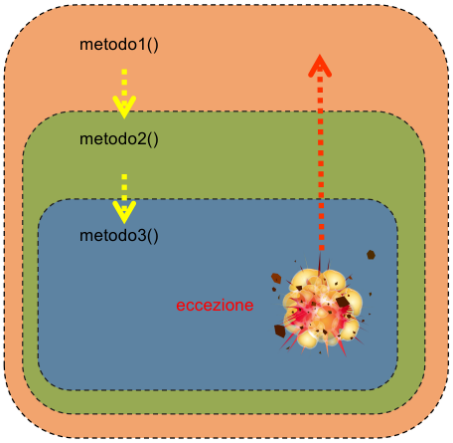


Immagine 37 propagazione a ritroso di una eccezione

DEFINIZIONE: Propagare un oggetto è detto exception throwing, e fermarne la propagazione exception catching.

In generale, gli oggetti da propagare come eccezioni devono derivare dalla classe base *java.lang.Exception* appartenente alle *Java Core API*. A partire da questa, è possibile creare nuovi tipi di eccezioni per mezzo del meccanismo dell'ereditarietà, specializzando il codice secondo il caso da gestire.

Stack Trace

Prima di proseguire, soffermiamoci un attimo per definire il concetto di *stack trace*.

DEFINIZIONE: Lo stack trace consente di tracciare la sequenza delle invocazioni tra funzioni annidate in uno specifico istante della applicazione.

Uno stack trace è quindi un elenco di *fotogrammi* dello stack che rappresentano un momento specifico durante l'esecuzione di un'applicazione. Questi fotogrammi possono contenere informazioni su un metodo oppure una funzione chiamata dal codice, generalmente iniziano dal metodo corrente e si estendono fino al punto di avvio della applicazione (metodo *main*).

In Java questo è particolarmente utile in quanto, grazie allo stack trace, è possibile generare una immagine dello stack della JVM e stamparlo a video per ottenere informazioni relative alla sequenza di chiamate che hanno generato un errore.

Nel prossimo esempio utilizziamo un metodo della classe Thread (che analizzeremo successivamente nel libro) per stampare a video lo stack trace di una applicazione Java.

```
public class StackTrace {
  public static void main(String[] args) {
    a();
  }
  static void a() {
    b();
  }
  static void b() {
    c();
  }
```

```
    static void c() {
        d();
    }
    static void d() {
        Thread.dumpStack();
    }
}

java.lang.Exception: Stack trace
    at java.base/java.lang.Thread.dumpStack(Unknown Source)
    at javamattone.esercizi.eccezioni.StackTrace.d(StackTrace.java:22)
    at javamattone.esercizi.eccezioni.StackTrace.c(StackTrace.java:18)
    at javamattone.esercizi.eccezioni.StackTrace.b(StackTrace.java:14)
    at javamattone.esercizi.eccezioni.StackTrace.a(StackTrace.java:10)
    at javamattone.esercizi.eccezioni.StackTrace.main(StackTrace.java:6)
```

Oggetti throwable

Come abbiamo detto, Java consente di propagare solo alcuni tipi di oggetti. Di fatto, Java richiede che tutti gli oggetti da propagare siano derivati dalla classe base *java.lang.Throwable*.

Nonostante questo sembri smentire quanto affermato nel paragrafo precedente, in cui affermavamo che devono derivare dalla classe base *java.lang.Exception*, in realtà entrambe le affermazioni sono vere. Vediamo perché.

Tecnicamente, tutte le eccezioni generate dal linguaggio Java derivano dalla classe *java.lang.Throwable*, sottoclasse di *Object*. La classe *Throwable* contiene i metodi necessari a gestire lo *stack tracing*, ovvero la sequenza delle chiamate tra metodi, per la propagazione dell'oggetto a ritroso lungo la sequenza corrente delle chiamate. I costruttori di *Throwable*, hanno la responsabilità di mettere in moto il meccanismo di propagazione e consentono di aggiungere dettaglio all'errore in varie forme.

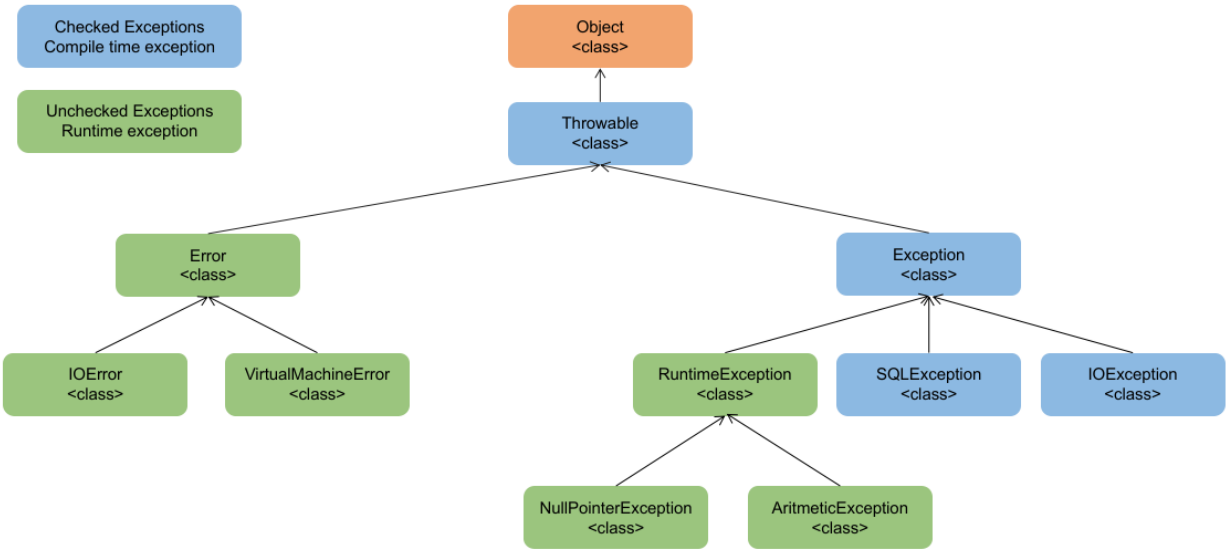


Immagine 38 Eccezioni: gerarchia

Nella immagine è schematizzata la gerarchia delle eccezioni in java. Nella realtà le classi sono molto di più di quelle mostrate nella figura in cui sono rappresentate solo alcune tra le eccezioni più comuni.

Per convenzione, ogni eccezione definita dal programmatore deve derivare da *java.lang.Exception* che a sua volta deriva da *Throwable*.

I costruttori della classe *Throwable* sono i seguenti:

Metodi costruttori di *Throwable*
Throwable()
Crea un oggetto Throwable con messaggio di errore nullo. Nota: cause non è inizializzata e può essere successivamente inizializzata da una chiamata a *initCause(java.lang.Throwable)*.
Throwable(String message)
Crea un oggetto *Throwable* con il messaggio di errore. Il messaggio di errore trasportato da un oggetto *Throwable*, è accessibile grazie al metodo *getMessage()*. Nota: *cause* non è inizializzata e può essere successivamente inizializzata da una chiamata a *initCause(java.lang.Throwable)*.
Throwable(String message, Throwable cause)
Crea un oggetto *Throwable* con il messaggio di errore e la causa. Il messaggio di errore trasportato da un oggetto *Throwable* e la causa, sono accessibili rispettivamente grazie al metodo *getMessage()* e *getCause()*. Nota: *cause* può essere **null**. In questo caso significa che la causa del problema non è nota. Nota: il messaggio di dettaglio associato alla causa non viene automaticamente incorporato nel messaggio di dettaglio di questo oggetto *throwable*.
protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)
Crea un oggetto *Throwable* con il messaggio di errore e la causa. Il messaggio di errore trasportato da un oggetto *Throwable,* e la causa, sono accessibili rispettivamente grazie al metodo *getMessage()* e *getCause()*. Nota: *cause* può essere **null**. In questo caso significa che la causa del problema non è nota. Nota: gli altri costruttori di *Throwable* considerano la *soppressione* come abilitata e il tracciamento dello *stack* come scrivibile.
Throwable(Throwable cause)
Costruisce un nuovo oggetto throwable con la causa specificata e un messaggio di dettaglio di errore (cause==null ? null : cause.toString()). Nota: La causa può essere null. In questo caso significa che la causa del problema non è nota. Nota: in genere cause contiene la classe e il messaggio di dettaglio della causa).

Per poter gestire la propagazione dell'oggetto lungo la sequenza delle chiamate, i costruttori effettuano una chiamata al metodo ***public*** *Throwable fillInStackTrace()*, il quale registra lo stato dello stack di sistema. Se *writableStackTrace* è disabiliato (*false*) la chiamata a *fillInStackTrace()* non produce nessun effetto.

Il metodo ***public*** *void printStackTrace()* consente invece di stampare sullo standard error lo stack trace della JVM.

Eccezioni controllate ed eccezioni incontrollate (checked e unchecked)

Nella *Immagine 38 Eccezioni: gerarchia* viene messo in evidenza che le eccezioni Java sono suddivise in due categorie distinte: le *eccezioni controllate* o *checked exceptions* e quelle *incontrollate* o *unchecked exceptions*.

Le prime, *checked exceptions* , rappresentano la maggior parte delle eccezioni a livello applicativo (comprese quelle definite dall'utente) e hanno bisogno di essere gestite esplicitamente (da qui la dicitura *controllate*). Le eccezioni di questo tipo:

1. Possono essere definite dal programmatore;

2. Devono essere allocate mediante operatore ***new****;*

3. Hanno la necessità di essere esplicitamente propagate;

4. Richiedono di essere esplicitamente gestite dal programmatore.

Una eccezione controllata potrebbe ad esempio rappresentare un errore durante il tentativo di apertura del file, a causa di un inserimento errato del nome (*IOException*).

Le eccezioni derivate dalla classe *java.lang.RuntimeException Immagine 38 Eccezioni: gerarchia*, appartengono invece alla seconda categoria di eccezioni. Le *eccezioni incontrollate*, sono generate automaticamente dalla JVM e sono relative a tutti quegli errori di programmazione che, tipicamente, non sono controllati dal programmatore a livello applicativo: riferimento ad oggetti nulli, accesso errato al contenuto di un array ecc. ecc.

Ad esempio, un'eccezione incontrollata di tipo *java.lang.ArrayIndexOutOfBoundException* sarà automaticamente generata se, tentassimo di inserire un elemento all'interno di un array, utilizzando un indice maggiore di quelli consentiti dalle dimensioni dell'oggetto.

```
byte[] elenco = new byte[20];
//Indici consentiti 0..19
elenco[20]=5;
```

il cui output è il seguente:

```
java.lang.ArrayIndexOutOfBoundsException
at src.esercizi.eccezioni.ArrayOutOfBounds.main(ArrayOutOfBounds.java:8)
Exception in thread "main"
```

Le eccezioni appartenenti alla seconda categoria, per loro natura sono difficilmente gestibili, non sempre possono essere catturate e gestite, causano spesso la terminazione anomala dell'applicazione. A differenza delle prime, che devono essere obbligatoriamente controllate e per le quali la mancanza di controllo produce un errore in fase di compilazione, le secondo non richiedono un controllo esplicito (anche se possono essere comunque catturate e gestite a differenza degli errori che vedremo successivamente), e non possono essere controllate in fase di compilazione.

Java dispone di un gran numero di eccezioni predefinite di tipo *incontrollato*, in grado di descrivere le principali condizioni di errore a livello di codice sorgente. Esaminiamo le più comuni.

NullPointerException

L'eccezione *java.lang.NullPointerException* è sicuramente la più comune tra queste, ed è generata tutte le volte che l'applicazione tenti di fare uso di un oggetto nullo. In particolare sono cinque le condizioni che possono causare la propagazione di quest'oggetto:

1. Effettuare una chiamata ad un metodo di un oggetto nullo;

2. Accedere o modificare un dato membro pubblico di un oggetto nullo;

3. Richiedere la lunghezza di un array nullo;

4. Accedere o modificare i campi di un array nullo;

*5. Propagare un'eccezione nulla (ovvero non allocata mediante operatore **new**).*

Di fatto, questo tipo di eccezione viene generata automaticamente ogni volta si tenti di effettuare un accesso illegale ad un oggetto **null**.

IndexOutOfBoundException

L'eccezione *IndexOutOfBoundException* è generata tutte le volte che si tenta di accedere ad un indice errato in una collezione di elementi: collezioni, array, stringhe (array di caratteri). E' estesa da due sottoclassi che rappresentano due diverse specializzazioni dello stesso errore.

L'eccezione *java.lang.ArrayIndexOutOfBoundException*, già introdotta in questo paragrafo, rappresenta la prima delle due ed è utilizzata per controllare tutte le situazioni in cui si tenti di utilizzare indici errati per accedere a dati contenuti in strutture dati ordinate come ad esempio gli array.

La seconda sottoclasse di *java.lang.IndexOutOfBoundException* è la classe *java.lang.StringIndexOutOfBoundException*, generata da alcuni metodi dell'oggetto *String* , per indicare che un indice è minore di zero oppure maggiore o uguale alla dimensione della stringa

rappresentata (una stringa in Java è rappresentata mediante un array di byte i cui elementi sono hanno indici a partire da 0).

```
String nome = "massimiliano";
System.out.println("La lunghezza di nome e': "+nome.length());
//La prossima operazione genera una eccezione
System.out.println("Il carattere "+nome.length()+"è: "
```

```
La lunghezza di nome e': 12
java.lang.StringIndexOutOfBoundsException: String index out of range: 12
at java.lang.String.charAt(String.java:516)
at src.esercizi.eccezioni.StringOutOfBounds.main(StringOutOfBounds.java:10)
Exception in thread "main"
```

ArithmeticException

La classe java.lang.ArithmeticException, è generata tutte le volte che si tenti di effettuare una operazione aritmetica non consentita, come ad esempio la divisione di un numero per zer

```
int i = 100;
int j = 0;
int risultato = i/j;
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at javamattone.esercizi.capitolo10.esempio1.ArithmeticErrorCondition.main(ArithmeticErrorCondition.java:7)
```

Errori

Oltre alle eccezioni di primo e secondo tipo, il *package java.lang* contiene la definizione di un altro tipo particolare di classi chiamate *errori*. Queste classi, definite anche loro per ereditarietà dalla superclasse *java.lang.Throwable*, rappresentano errori gravi della JVM e causano l'interruzione anomala dell'applicazione. A causa della loro natura, questo tipo di eccezioni non sono mai catturate per essere gestite: se un errore accade, la JVM stampa un messaggio di errore su terminale ed esce.

ClassNotFoundException

Questa eccezione, è generata dal classloader della Java Virtual Machine quando l'applicazione richiede di caricare un oggetto, la cui definizione non è indicata all'interno della variabile d'ambiente *CLASSPATH*.

OutOfMemoryError

E' l'errore che mai vorremmo vedere in quanto generalmente associato ad un memory leak. Questo errore viene infatti generato quando non c'è spazio sufficiente per allocare un nuovo

oggetto nello heap-space di Java a causa del fatto che il Garbage Collector non è in grado di rendere disponibile lo spazio per accogliere un nuovo oggetto.

I motivi possono essere diversi:

1. La Java Virtual Machine ha esaurito la quantità di memoria prestabilita

La JVM alloca per lo *heap-space* una quantità di memoria prestabilita: il valore di default per lo heap-space è 512M. Qualora si superi lo spazio a disposizione, e la JVM non fosse più in grado di espandere lo heap-space per contenere nuovi oggetti, la JVM ritornerà un errore di tipo *OutOfMemoryError* .

In questi casi è possibile agire sulla JVM al momento della sua esecuzione, al fine di modificare lo spazio allocabile per lo *heap-space*. Per questo la JVM mette a disposizione i seguenti parametri:

-Xmx<size>: per impostare la dimensione massima che può avere l'Heap;

-Xms<size>: per impostare la dimensione iniziale che deve avere l'Heap;

-XX:MaxHeapFreeRatio<ratio>: per impostare la percentuale massima da garantire di spazio libero all'interno dello Heap;

-XX:MinHeapFreeRatio<ratio>: per impostare la percentuale minima da garantire di spazio libero all'interno dello Heap;

Dove -Xms deve essere minore o uguale a -Xmx (alcune guide riportano che un valore identico dei due parametri sia ottimale per evitare che lo heap-space sia sottoposto a operazioni di ridimensionamento o crescita da parte del Garbage Collector).

2. La memoria nativa è insufficente per supportare l'espansione dello heap-space;

3. A causa di un memory leak;

E' la condizione per cui l'applicazione Java continua ad allocare spazio nello heap-space senza mai rilasciare le risorse a causa della impossibilità per i Garbage Collector di rimuove oggetti non più utilizzati.

Stack trace ed eccezioni

In Java, stack trace ed eccezioni sono spesso associate tra loro: quando un'applicazione Java lancia un'eccezione, è facile vedere la stampa a video dell'immagine dello stack della JVM.

Ciò è dovuto al modo in cui funzionano le eccezioni.

Quando il codice Java genera un'eccezione, il run-time cerca nello stack un metodo con un gestore in grado di catturare e gestire l'eccezione: se ne trova uno, gli passa l'eccezione; in caso contrario l'eccezione prosegue la sua corsa verso il metodo main fino a causare la terminazione del programma. Quindi, le eccezioni e lo stack di chiamate sono collegati direttamente.

Consideriamo il prossimo esempio ed il suo output:

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

```
    metodo1(null);
  }
  static void metodo1(int[] a) {
    metodo2(a);
  }
  static void metodo2(int[] b) {
    System.out.println(b[0]);
  }
}
```

Il cui output è il seguente:

```
Exception in thread "main" java.lang.NullPointerException: Cannot load from int array because "b" is null
    at javamattone.esercizi.capitolo10.esempio1.StackTrace.metodo2(StackTrace.java:13)
    at javamattone.esercizi.capitolo10.esempio1.StackTrace.metodo1(StackTrace.java:9)
    at javamattone.esercizi.capitolo10.esempio1.StackTrace.main(StackTrace.java:5)
```

Leggendo dal basso verso l'alto, il metodo *main()* della applicazione esegue una chiamata a *metodo1()* passandogli come argomento un array nullo, array che viene passato a sua volta al *metodo2()* che tenta di visualizzarne sul terminale il valore del primo elemento. Essendo nullo l'array, nel momento in cui *metodo2()* tenta di leggere l'elemento sulla cima, l'applicazione produce una eccezione di tipo *java.lang.NullPointerException*.

Le righe 2,3,4 del messaggio identificano la sequenza delle chiamate attive e la riga del codice sorgente, mentre la prima riga restituisce un messaggio come definito nella stringa passata al costruttore della classe.

Definire eccezioni personalizzate

Quando definiamo nuovi oggetti, è spesso desiderabile disegnare nuovi tipi di eccezioni che li accompagnino.

Come specificato nei paragrafi precedenti, un nuovo tipo di eccezione deve essere derivata da *java.lang.Exception* ed appartenere quindi alla prima categoria di eccezioni: quelle controllate. Il funzionamento interno della nuova eccezione non è ristretto da nessuna limitazione.

Ad esempio, potremmo creare una eccezione di tipo *StackOutOfBoundException*, per segnalare che la *Pila* definita in precedenza, ha raggiunto la capienza massima. Di seguito la definizione della classe:

```
public class StackIndexOutOfBoundsException extends Exception {
  public StackIndexOutOfBoundsException() {
  }
  public StackIndexOutOfBoundsException(String message) {
    super(message);
  }
  public StackIndexOutOfBoundsException(Throwable cause) {
    super(cause);
  }
  public StackIndexOutOfBoundsException(String message, Throwable cause) {
```

```
    super(message, cause);
  }
  public StackIndexOutOfBoundsException(String message, Throwable cause,
                                        boolean enableSuppression,
                                        boolean writableStackTrace) {
    super(message, cause, enableSuppression, writableStackTrace);
  }

}
```

Poiché una eccezione è un oggetto come altri, potremmo includere all'interno qualsiasi altro tipo di informazione che ritenessimo necessario trasportare. Ad esempio, se volessimo trasportare la dimensione massima della Pila all'interno della eccezione, potremmo modificare la classe nel modo seguente:

```
public class StackIndexOutOfBoundsException extends Exception {

  private int capienzaMassima;

  public StackIndexOutOfBoundsException(int capienzaMassima) {
    this.capienzaMassima = capienzaMassima;
  }

  public StackIndexOutOfBoundsException(String message, int capienzaMassima) {
    super(message);
    this.capienzaMassima = capienzaMassima;
  }

  public StackIndexOutOfBoundsException(Throwable cause, int capienzaMassima) {
    super(cause);
    this.capienzaMassima = capienzaMassima;
  }

  public StackIndexOutOfBoundsException(String message, Throwable cause,
int capienzaMassima)
  {
    super(message, cause);
    this.capienzaMassima = capienzaMassima;
  }

  public StackIndexOutOfBoundsException(String message, Throwable cause,
                                        boolean enableSuppression,
                                        boolean writableStackTrace,
                                        int capienzaMassima) {
    super(message, cause, enableSuppression, writableStackTrace);
    this.capienzaMassima = capienzaMassima;
  }
```

```
    public int getCapienzaMassima() {
      return capienzaMassima;
    }

}
```

L'istruzione throw

La definizione di un oggetto di tipo *Throwable*, non è sufficiente a completare il meccanismo di propagazione dell'oggetto. Nei paragrafi precedenti abbiamo affermato che, la propagazione di un'eccezione controllata deve essere gestita esplicitamente dall'origine della propagazione dell'oggetto, fino alla cattura e successiva gestione del medesimo.

Le eccezioni, vengono propagate a ritroso attraverso la sequenza dei metodi chiamanti tramite l'istruzione **throw** che ha sintassi:

throw *oggetto_throwable;*

dove *oggetto_throwable* è una istanza valida dell'oggetto *Throwable*. E' importante tener presente che *oggetto_throwable* rappresenta un oggetto valido creato mediante l'operatore **new**, e non semplicemente un tipo di dato.

L'istruzione **throw**, una volta chiamata, causa la terminazione immediata del metodo corrente ed invia l'oggetto specificato al metodo chiamante. A differenza di **return** però, non consente di ritornare un parametro di ritorno anche se specificato nella definizione del metodo.
Non c'è modo da parte del chiamante di riprendere il metodo terminato senza effettuare una nuova chiamata. Anche in questo caso, il metodo non riprenderà dal punto in cui è stato interrotto.

La clausola throws

Le eccezioni possono essere propagate solo dai metodi che ne dichiarano la possibilità. Tentare di generare una eccezione all'interno di un metodo che, non ha precedentemente dichiarato di avere la capacità di propagare tali oggetti, causerà un errore in fase di compilazione.

Per dichiarare che un metodo ha la capacità di causare eccezioni, è necessario utilizzare la clausola **throws** che, indica al metodo chiamante che un oggetto eccezione potrebbe essere generato o propagato dal metodo chiamato. La clausola **throws** ha sintassi:

[modificatori] tipo nome (parametri_formali) ***throws*** *tipo_throwable{[,tipo_throwable]}{*
    istruzione
    [istruzione]
}

Un metodo con una clausola **throws**, può generare solo eccezioni del tipo dichiarato da *tipo_throwable* oppure ogni tipo derivato da esso.

Se un metodo contenente una clausola throws viene ridefinito (overrided) attraverso l'ereditarietà, il nuovo metodo può scegliere se contenere o no la clausola throws. Nel caso in cui scelga di contenerla, sarà costretto a dichiarare lo stesso tipo del metodo originale o, al massimo, un tipo derivato.

Analizziamo nuovamente nei dettagli il metodo membro push(int) della classe Stack.

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

Quando viene chiamato il metodo *push*, l'applicazione procede correttamente fino a che non si tenti di inserire un elemento all'interno della pila piena. In questo caso non è possibile venire a conoscenza del fatto che l'elemento è andato perduto.

Utilizzando il meccanismo delle eccezioni è possibile risolvere il problema, modificando leggermente il metodo affinché generi un'eccezione quando si tenti di inserire un elemento nella pila piena. Per far questo, utilizziamo l'eccezione definita nei paragrafi precedenti:

```
public void push(int dato) throws StackOutOfBoundsException{
    if (cima < dimensioneMassima) {
      dati[cima] = dato;
      cima++;
    } else {
        throw new StackOutOfBoundsException(
        "La pila ha raggiunto la dimensione massima.
        Il valore inserito è andato perduto +[" + dato + "]", dati.length);
     }
}
```

La nuova versione del metodo genererà una condizione di errore segnalando alla applicazione l'anomalia ed evitando che dati importanti vadano perduti.

Istruzioni try / catch

A questo punto siamo in grado di generare e propagare un'eccezione. Consideriamo quindi la prossima applicazione in cui usiamo la classe *Pila* modificata e proviamo a compilare:

```
public class ExceptionCatching {
  public static void main(String[] argv) {
    int capienza = 10;
    Pila stack = new Pila(capienza);
    for (int i = 0; i <= capienza; i++) {
      System.out.println("Inserisco " + i + " nella pila");
      stack.push(i);
      System.out.println("Il dato è stato inserito");
```

```
    }
  }
}
```

Al momento della compilazione otterremo il seguente messaggio di errore:

```
StackOutOfBound.java:12: unreported exception
src.esercizi.eccezioni.StackOutOfBoundException; must be caught or declared to be thrown
stack.push(i);
^
1 error
*** Compiler reported errors
```

Il compilatore allerta il programmatore avvertendo che l'eccezione eventualmente propagata dal metodo *push* della classe *Pila* deve essere obbligatoriamente catturata e gestita. Una volta generata un'eccezione, l'applicazione è destinata alla terminazione salvo che l'oggetto propagato sia catturato prima di raggiungere il metodo *main* del programma o, direttamente al suo interno.

Di fatto, Java obbliga il programmatore a catturare un'eccezione al più all'interno del metodo *main* di un'applicazione. Questo compito spetta all'istruzione **catch**.

Questa istruzione fa parte di un insieme di istruzioni dette *guardiane*, deputate alla gestione delle eccezioni ed utilizzate per racchiudere e gestire le chiamate a metodi che le generano. L'istruzione catch non può gestire da sola un'eccezione, ma deve essere sempre accompagnata da un *blocco try*. Il *blocco try* è utilizzato come guardiano per il controllo di un blocco di istruzioni, potenziali sorgenti di eccezioni ed ha la sintassi seguente:

```
try {
     istruzione;
    [istruzione]
}

catch (tipo_throwable nome) {
   istruzione;
   [istruzione]
   }

catch (tipo_throwable nome) {
   istruzione;
   [istruzione]
   }
```

Valgono le seguenti regole:

*1. L'istruzione **catch**, cattura tutte le eccezioni di tipo compatibile con il suo argomento e solamente quelle generate dalle chiamate a metodi racchiuse all'interno del blocco **try**.*

*2. Se un'istruzione nel blocco **try** genera un'eccezione, le rimanenti istruzioni nel blocco non sono eseguite.*

*3. L'esecuzione di un blocco **catch** esclude automaticamente tutti gli altri.*

Per comprendere meglio il funzionamento del blocco di guardia, immaginiamo di avere definito una classe con tre metodi: *f1()*, *f2()* e *f3()*, e supponiamo che i primi due metodi generano rispettivamente un'eccezione di tipo *IOException* ed un'eccezione di tipo *NullPointerException*. Lo pseudo codice seguente, rappresenta un esempio di blocco di guardia, responsabile di intercettare e poi gestire le eccezioni propagate dai due metodi.

```
try{
        f1(); //Una eccezione in questo punto fa saltare f2() e f3()
        f2(); //Una eccezione in questo punto fa saltare f3()
        f3();
}
catch (IOException e) {
        System.out.println(e.toString())
}
catch (NullPointerException e) {
        System.out.println(e.toString())
}
```

Nel caso in cui sia il metodo *f1()* a generare e propagare l'eccezione, l'esecuzione del blocco di guardia, **try**, passerebbe il controllo della esecuzione blocco **catch** che dichiara di gestire l'eccezione generata da *f1()*:

```
catch (IOException e) {
        System.out.println(e.toString())
}
```

Se invece fosse il metodo *f2()* a propagare l'eccezione, *f1()* terminerebbe correttamente, *f3()* non verrebbe eseguito ed il controllo passerebbe al blocco catch

```
catch (NullPointerException e) {
        System.out.println(e.toString())
}
```

Infine, se nessuna eccezione viene generata, i tre metodi vengono eseguiti correttamente ed il controllo passa alla prima istruzione immediatamente successiva al blocco try/catch.

Per concludere, una versione corretta della applicazione *ExceptionCatching* potrebbe essere la seguente:

```
public class ExceptionCatching {
  public static void main(String[] argv) {
    int capienza = 10;
    Pila stack = new Pila(capienza);
    for (int i = 0; i <= capienza; i++) {
      System.out.println("Inserisco " + i + " nella pila");
      try {
        stack.push(i);
        System.out.println("Il dato è stato inserito");
      } catch (StackIndexOutOfBoundsException e) {
        System.out.println("Il dato è andato perduto");
        e.printStackTrace();
      }
    }
  }
}
```

```
Inserisco 0 nella pila
Il dato è stato inserito
.....
Inserisco 9 nella pila
Il dato è stato inserito
Inserisco 10 nella pila
Il dato è andato perduto

javamattone.esercizi.capitolo10.esempio2.StackIndexOutOfBoundsException: La pila ha raggiunto la dimensione massima. Il valore inserito è andato perduto +[10]
    at javamattone.esercizi.capitolo10.esempio2.Pila.push(Pila.java:40)
    at javamattone.esercizi.capitolo10.esempio2.ExceptionCatching.main(ExceptionCatching.java:10)
```

Singoli catch per eccezioni multiple

Differenziare i blocchi catch affinché gestiscano ognuno particolari condizioni di errore identificate dal tipo dell'eccezione propagata, consente di poter specializzare il codice affinché possa prendere decisioni adeguate per la soluzione del problema verificatosi. Esistono però dei casi in cui è possibile trattare più eccezioni utilizzando lo stesso codice: in questi casi è necessario un meccanismo affinché il programmatore non debba replicare inutilmente linee di codice.

La soluzione è a portata di mano: abbiamo detto nel paragrafo precedente che ogni istruzione **catch** cattura solo le eccezioni compatibili con il tipo definito dal suo argomento. Ricordando quanto detto parlando di oggetti compatibili, questo significa che un'istruzione **catch** cattura ogni eccezione dello stesso tipo definito dal suo argomento o derivata dal tipo dichiarato. D'altra parte sappiamo che tutte le eccezioni sono definite per ereditarietà a partire dalla classe base *Exception*.

Nel prossimo esempio, il blocco catch catturerà tutte le eccezioni possibili:

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class EccezioniMultiple1 {
  public static void main(String[] args) {
    System.out.println("Inserisci due numeri: ");
    Scanner sc = new Scanner(System.in);
    try {
      System.out.println("primo numero: ");
      int num1 = sc.nextInt();
      System.out.println("secondo numero: ");
      int num2 = sc.nextInt();
      int dividedNum = num1 / num2;
      System.out.println("After division result: " + dividedNum);
    } catch (Exception e) {
     e.printStackTrace();
    }
  }
}
```

Modifichiamo leggermente l'esempio:

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class EccezioniMultiple1 {
  public static void main(String[] args) {
    System.out.println("Inserisci due numeri: ");
    Scanner sc = new Scanner(System.in);
    try {
      System.out.println("primo numero: ");
      int num1 = sc.nextInt();
      System.out.println("secondo numero: ");
      int num2 = sc.nextInt();
      int dividedNum = num1 / num2;
      System.out.println("After division result: " + dividedNum);
    } catch (InputMismatchException e) {
     System.out.println("ERRORE: hai inserito un dato non valido");
    }
    catch (Exception e) {
     e.printStackTrace();
    }
  }
```

```
}
```

In questo caso, il primo blocco catch è specializzato nel catturare e gestire un solo tipo di eccezione. Tutte le altre saranno catturate nel secondo blocco. Da notare che invertire i blocchi non ha senso:

```
.....
catch (Exception e) {
      e.printStackTrace();
    }catch (InputMismatchException e) {
     System.out.println("ERRORE: hai inserito un dato non valido");
    }
    -------
```

Il primo dei due, quello generico, catturerebbe tutte le eccezioni rendendo inutilizzabile il secondo blocco, quello specializzato. In ogni caso il compilatore produrrebbe il seguente messaggio di errore.

EccezioniMultiple1.java:[30,11] exception java.util.InputMismatchException has already been caught

A partire da Java 7 è possibile utilizzare l'operatore | *(OR bit a bit)* per specificare un elenco di eccezioni da catturare in un blocco catch.

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class EccezioniMultiple1 {
  public static void main(String[] args) {
    System.out.println("Inserisci due numeri: ");
    Scanner sc = new Scanner(System.in);
    try {
      System.out.println("primo numero: ");
      int num1 = sc.nextInt();
      System.out.println("secondo numero: ");
      int num2 = sc.nextInt();
      int dividedNum = num1 / num2;
      System.out.println("After division result: " + dividedNum);
    } catch (InputMismatchException | ArithmeticException e) {
     System.out.println("ERRORE: hai inserito un dato non valido oppure hai diviso un
numero per zero");
    }
    catch (Exception e) {
     e.printStackTrace();
    }
  }
}
```

In questi casi l'operatore **instanceof** potrebbe essere utile a determinare la natura dell'eccezione catturata:

```
if (e instanceof InputMismatchException) {
    System.out.println("Error! - E' necessario nserire un numero");
} else if (e instanceof ArithmeticException) {
    System.out.println("Error! - Non è possibile dividere un numero per 0");
} else {
    System.out.println("OPS! C'è stato un errore. Riprova");
}
```

Le altre istruzioni guardiane. Finally

Di seguito ad ogni blocco **catch**, può essere utilizzato opzionalmente un blocco **finally** che sarà *sempre* eseguito prima di uscire dal blocco guardiano, e come ultimo blocco eseguito.

Questo blocco di istruzioni, offre la possibilità di eseguire sempre un certo insieme di istruzioni a prescindere da come i blocchi guardiani catturano e gestiscono le condizioni di errore.

I blocchi finally non possono essere evitati dal controllo di flusso della applicazione.

Le istruzioni **break**, **continue** o **return** all'interno del blocco **try** o all'interno di un qualunque blocco **catch** verranno eseguito solo dopo l'esecuzione delle istruzioni contenute nel blocco **finally**. Solo una chiamata del tipo *System.exit()* ha la capacità di evitare l'esecuzione del blocco di istruzioni in questione.

La sintassi è la seguente:

```
try {
  istruzione;
  [istruzione]
}catch (tipo_throwable nome) {
  istruzione;
  [istruzione]

}finally {
  istruzione;
  [istruzione]
}
```

Utilizzare un blocco **finally** è utile in una molteplicità di situazioni ed aiuta a scrivere codice più compatto e comprensibile. Solo per citare alcuni casi in cui il blocco **finally** può rivelarsi estremamente vantaggioso:

1. Abbiamo del codice che deve essere eseguito indipendentemente dal fatto che venga generata o meno un'eccezione.

2. Ci sono risorse che devono essere chiuse indipendentemente dal fatto che venga generata o meno un'eccezione: chiudere le connessioni ad un database o chiudere risorse che lo richiedono è un buon esempio.

3. Eseguire blocchi di codice che, in alternativa, sarebbero stati letteralmente duplicati tra i vari blocchi catch.

4. *Eseguire blocchi di codice che devono essere obbligatoriamente eseguiti prima che un metodo ritorno a seguito della chiamata e* ***return****.*

Modifichiamo l'esempio precedente:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class EccezioniMultiple1 {
  public static void main(String[] args) {
    System.out.println("Inserisci due numeri: ");
    Scanner sc = new Scanner(System.in);
    try {

      System.out.println("primo numero: ");
      int num1 = sc.nextInt();

      System.out.println("secondo numero: ");
      int num2 = sc.nextInt();

      int dividedNum = num1 / num2;

      System.out.println("After division result: " + dividedNum);

    }catch(ArithmeticException | InputMismatchException e) {
      System.out.println("Hai inserito una stringa oppure hai diviso un numero per zero");
    }
    catch (Exception e) {
      System.out.println("OPS! C'è stato un errore. Riprova");
    }finally{
      System.out.println("Siamo nel blocco finally");
    }
  }
}
```

Eseguendo l'applicazione notiamo subito che il blocco **finally** viene eseguito anche nel caso in cui non venga generata nessuna eccezione.

```
Inserisci due numeri:
primo numero:
1
secondo numero:
```

```
2
After division result: 0
Siamo nel blocco finally
```

Blocchi try-with-resources

Abbiamo accennato al fatto che il blocco **finally** può essere utilizzato in cui esistono risorse che richiedono di essere chiuse prima della terminazione del codice. Nel prossimo frammento di codice è riportato un caso tipico di utilizzo del blocco **finally** per garantirci che le risorse allocate vengano sempre e correttamente chiuse.

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {

  FileReader fr = new FileReader(path);
  BufferedReader br = new BufferedReader(fr);
  try {
    return br.readLine();
  } finally {
    br.close();
    fr.close();
  }
}
```

Tuttavia, questo esempio potrebbe ancora causare un *leak* con le risorse rilasciate. Nella realtà, un programma deve fare di più che fare affidamento sul Garbage Collector per recuperare la memoria di una risorsa: deve sopratutto rilasciare la risorsa al sistema operativo chiamando, in genere, il metodo *close()* della risorsa. Tuttavia, se un programma non riesce a farlo prima che il GC recuperi la risorsa le informazioni necessarie per rilasciare la risorsa andranno perse con l'effetto finale che la risorsa è persa in quanto considerata ancora in uso dal sistema operativo.

In altre situazioni, se la chiamata al metodo *close()* di *BufferedReader* generasse un eccezione, allora la risorsa *FileReader* sarebbe persa.

A partire da java 7, è possibile riscrivere lo stesso blocco di codice nel modo seguente:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    return br.readLine();
  }
}
```

Questo tipo di blocco, come è facile intuire, non è semplicemente una semplificazione sintattica, ma sottintende alla chiusura assistita delle risorse. La sintassi è la seguente:

```
try (creazione_risorse_autoclosable){
        istruzione;
        [istruzione]
}

[catch (tipo_throwable nome) {
        istruzione;
        [istruzione]

}]

[finally {
        istruzione;
        [istruzione]
}]
```

dove *creazione_risorse_autoclosable* è il blocco che rappresenta la creazione mediante operatore **new** di risorse di tipo *Autoclosable* (gli oggetti *AutoCloseable* e *Closeable* sono delle classi che implementano rispettivamente le interfaccie *java.lang.AutoCloseable* e *java.io.Closeable*).

Le regole sono le seguenti:

1. *try-with-resources chiude le risorse nell'ordine inverso rispetto a come sono dichiarate.*

Nell'esempio precedente sarà chiuso prima BufferedReader e successivamente FileReader.

*2. try-with-resources può avere i blocchi **catch** e **finally** come il classico try.*

E' comunque importante sapere che nel try-with-resources sia il **catch** che il **finally** sono eseguiti solo dopo che la risorsa è stata chiusa.

13.Polimorfismo di forma ed ereditarietà avanzata: interfacce

Introduzione

L'ereditarietà rappresenta uno strumento di programmazione molto potente; d'altra parte il semplice modello di ereditarietà presentato non risolve alcuni problemi molto comuni, e se non bastasse crea alcuni problemi potenziali che possono essere risolti solo scrivendo codice aggiuntivo.

Uno dei limiti più comuni di un modello di ereditarietà singola è che, non prevede l'utilizzo di una classe base come modello puramente concettuale, ossia priva dell'implementazione delle funzioni base. Se facciamo un passo indietro, ricordiamo che abbiamo definito una *Pila* (pila) come un contenitore all'interno del quale inserire dati da recuperare secondo il criterio *primo ad entrare, ultimo ad uscire*. Potrebbe esserci però un'applicazione che richiede vari tipi differenti di *Pila*: uno utilizzato per contenere valori interi ed un altro utilizzato per contenere valori reali a virgola mobile. In questo caso, le regole da utilizzare per manipolare la *Pila* sarebbero le stesse, quello che cambia sono i tipi di dato contenuti.

Anche se utilizzassimo la classe *Pila* riportata di seguito, sarebbe impossibile per mezzo della semplice ereditarietà creare specializzazioni dell'entità rappresentata, a meno di riscrivere una parte sostanziale del codice del nostro modello in grado di contenere solo valori interi.

```
public class Pila {
  private int[] dati;
  private int cima;
  private int dimensioneMassima;
  public Pila() {
    this(10);
  }
  public Pila(int capacitaMassimaDellaPila) {
    dati = new int[capacitaMassimaDellaPila];
    cima = 0;
  }
  public void push(int dato) throws StackIndexOutOfBoundsException {
    if (cima < dimensioneMassima) {
      dati[cima] = dato;
      cima++;
    } else {
      throw new StackIndexOutOfBoundsException(
          "La pila ha raggiunto la dimensione massima. Il valore inserito è andato perduto +["
          + dato + "]",
          dati.length);
    }
  }
```

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

Un altro problema che non è risolto dal modello ad ereditarietà è quello di non consentire ereditarietà multipla, ossia la possibilità di derivare una classe da due o più classi base; la parola chiave **extends** prevede solamente un singolo argomento.

Java risolve tutti questi problemi con due variazioni al modello di ereditarietà definito: *interfacce* e *classi astratte*. Le *interfacce*, sono entità simili a classi, ma non contengono implementazioni delle funzionalità descritte. Le classi astratte, anch'esse simili a classi normali, consentono di implementare solo parte delle caratteristiche dell'oggetto rappresentato.

Interfacce e *classi astratte* assieme, permettono di definire un concetto senza dover conoscere i dettagli di una classe, posponendone l'implementazione attraverso il meccanismo dell'ereditarietà.

Polimorfismo : "un'interfaccia, molti metodi"

Polimorfismo è la terza parola chiave del paradigma ad oggetti. Derivato dal greco, significa *pluralità di forme* ed è la caratteristica che ci consente di utilizzare un'unica interfaccia per una moltitudine di azioni. Quale sia la particolare azione eseguita, dipende solamente dalla situazione in cui ci si trova.

In realtà abbiamo già parlato di polimorfismo identificando due tipologie di polimorfismo quello per classe (ereditarietà) e quello per metodi e dati:

1. polimorfismo ad hoc: ovvero method overloading.

Questa forma di polimorfismo è nota sin dagli anni '60 quando venne citata anche nel compendio "*Fundamental Concepts in Programming Languages*" di *Christopher Strachey* (1967), e consiste concretamente nella possibilità di ridefinire un medesimo metodo usando set di parametri diversi.

2. polimorfismo per inclusione: ovvero method overriding.

Strettamente legata all'ereditarietà prevede che una sottoclasse possa ridefinire metodi ereditati da una delle sue superclassi.

In questo capitolo parleremo di una ulteriore forma di polimorfismo che chiameremo *polimorfismo di forma*. Questo tipo di polimorfismo prevede la possibilità di definire interfacce generiche da specializzare solo successivamente al caso specifico.

Esiste infine un quarto tipo di polimorfismo detto polimorfismo parametrico, ma sarà approfondito in dettaglio nel prossimo capitolo.

Interfacce

Formalmente, un'interfaccia Java rappresenta un *prototipo* che consente al programmatore di definire lo scheletro di una classe: nomi dei metodi, tipi ritornati, lista degli argomenti. Al suo interno il programmatore può definire dati membro purché di tipo primitivo con un'unica restrizione: Java considererà implicitamente questi dati come **static** e **final** (costanti).

Le interfacce, sono molto utili per definire gruppi di classi aventi un insieme minimo di metodi in comune, senza fornirne però una implementazione comune.

Ad esempio, se volessimo generalizzare la definizione di *Pila*, potremmo definire un'interfaccia contenente i prototipi dei metodi *push* e *pop* comuni a tutti gli oggetti di questo tipo. Poiché, il tipo di dato gestito da questi oggetti dipende dall'implementazione della pila, ogni classe costruita da quest'interfaccia, avrà la responsabilità di gestire in modo appropriato i propri dati. Quello che un'interfaccia non consente è proprio l'implementazione del corpo dei metodi.

DEFINIZIONE: Un'interfaccia stabilisce il protocollo di una classe, senza preoccuparsi dei dettagli di implementazione. Ha una struttura simile a una classe, ma può contenere solo costanti e metodi d'istanza astratti (quindi non può contenere costruttori, variabili di istanza, definizione di metodi statici).

La scopo è quello di definire un *protocollo* per il comportamento che deve essere implementato da una classe, per cui vale la seguente regola:

1. Una qualsiasi classe che implementa una data interfaccia è quindi obbligata a fornire l'implementazione di tutti i metodi elencati nell'interfaccia.

Definizione di un'interfaccia

Per consentire al programmatore la dichiarazione di interfacce, Java dispone della parola chiave **interface**. La sintassi necessaria a definire una interfaccia è la seguente:

```
[public | private]interface identificatore [extends tipo[,tipo]]{
    definizione_di_costanti;
    definizione_dei_metodi_astratti;
}
```

Gli elementi necessari a dichiarare questo tipo particolare di classi, sono quindi elencati di seguito:

*1. I modificatori opzionali **public** o **private** per definire la visibilità della classe;*

*2. La parola chiave **interface**;*

3. Un nome che identifica la classe;

*4. Opzionalmente, la clausola **extends** se l'interfaccia è definita a partire da una interfaccia base;*

5 . Le dichiarazioni dei dati membro della classe (costanti), e dei prototipi dei metodi (metodi astratti).

Proviamo, ed esempio, a definire un'interfaccia che ci consenta di generalizzare la definizione della classe *Pila*. Per poterlo fare dobbiamo immaginare come generalizzare il tipo dell'oggetto che sarà gestito dalla *Pila*. Ricordiamo che l'oggetto *Object*, in quanto padre di tutte le classi Java, è compatibile con qualsiasi tipo definito o definibile, comprese le classi wrapper con cui possiamo rappresentare in forma di oggetti tutti i tipi primitivi. Questa caratteristica è alla base di ogni generalizzazione in Java in quanto ogni oggetto può essere ricondotto ad *Object*, e viceversa, da Object possono essere derivati tutti gli altri oggetti.

L'interfaccia generalizzata di un oggetto generico Pila è quindi la seguente:

```
import javamattone.esercizi.capitolo10.esempio2.StackIndexOutOfBoundsException;
public interface Pila {
  public void push(Object dato) throws StackIndexOutOfBoundsException;
  public Object pop();
}
```

Si noti infine che, nella definizione:

*1. Le variabili devono essere inizializzate e non possono essere modificate successivamente: anche se non sono dichiarate **final** di fatto sono delle costanti;*

2. I metodi sono tutti astratti: infatti al posto del corpo c'è solo un punto e virgola;

*3. I metodi dichiarati in una interfaccia sono sempre **public**. Di conseguenza, i corrispondenti metodi di una classe che implementa l'interfaccia devono essere **public**.*

*4. Una interfaccia può estendere una o più interfacce base (non classi), indicate dopo la parola chiave **extends**. Per le interfacce non vale la restrizione di ereditarietà singola che vale per le classi.*

Implementare una interfaccia

Poiché un'interfaccia rappresenta solo il prototipo di una classe, affinché possa essere utilizzata è necessario che ne esista un'implementazione che rappresenti una classe allocabile.

Per implementare un'interfaccia, Java mette a disposizione la parola chiave **implements** che nella forma più semplice ha la seguente sintassi:

```
class nome implements interfaccia{
      corpo_della_classe
}
```

La nostra classe pila di interi potrà quindi essere definita a partire dall'interfaccia Pila nel modo seguente:

```
public class PilaDiInteri implements Pila {

  private Integer[] dati;
  private int cima;
  private int dimensioneMassima;

  public PilaDiInteri() {
    this(10);
  }

  public PilaDiInteri(int capacitaMassimaDellaPila) {
    dati = new Integer[capacitaMassimaDellaPila];
    cima = 0;
  }

  @Override
  public void push(Object dato) throws StackIndexOutOfBoundsException {
    if (cima < dimensioneMassima) {
      dati[cima] = (Integer) dato;
      cima++;
    } else {
      throw new StackIndexOutOfBoundsException(
          "La pila ha raggiunto la dimensione massima. Il valore inserito è andato perduto
          +[" + dato + "]",
           dati.length);
    }
  }

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

}
```

Come per l'override di metodi, quando si implementa un metodo da interfaccia è consigliabile utilizzare l'annotazione **@Override** per indicare al compilatore che si intende implementare un metodo definito in una interfaccia.

Quando una classe implementa un'interfaccia è obbligata a fornirne un'implementazione di tutti i prototipi dei metodi. In caso contrario il compilatore genererà un messaggio di errore.

Di fatto, possiamo pensare ad un'interfaccia come ad una specie di contratto che l'ambiente di Java stipula con una classe. Implementando un'interfaccia la classe non si limita a definire un concetto da un modello logico (molto utile al momento del disegno dell'applicazione), ma assicurerà l'implementazione di almeno i metodi definiti nell'interfaccia.

La relazione che intercorre tra un'interfaccia ed una classe Java, è anch'essa una forma di ereditarietà (da qui la similitudine della annotazione *@Override*). Se l'interfaccia dovesse contenere definizioni di dati membro (anche se costanti), le stesse saranno ereditate dalla classe costruita da essa mediante la clausola **implements**.

Conseguenza diretta di quest'affermazione, è la possibilità di utilizzare le interfacce come *tipi* per definire variabili reference in grado di far riferimento ad oggetti definiti mediante implementazione di un'interfaccia.

```
Pila pila = new PilaDiInteri();
```

Valgono in questo caso tutte le regole già discusse parlando di compatibilità di tipi.

Interfacce sealed

Il modificatore **sealed** può essere utilizzato anche per le interfacce. I vincoli e la sintassi sono gli stessi che abbiamo presentato per le classi.

Rispetto ad una classe **sealed** però, un'interfaccia sealed può specificare sia da quali sottoclassi può essere implementata, sia da quali interfacce può essere estesa. In particolare le interfacce **sealed** possono essere estese anche da tipi **record**: infatti questi ultimi non possono estendere classi ma possono **implementare** interfacce. I tipi record hanno il vantaggio di essere implicitamente dichiarati **final**, quindi non dovremo preoccuparci di utilizzare un altro modificatore quando li dichiariamo (vedi le regole per la definizione di classi sealed e record).

Ereditarietà multipla

Se l'operatore **extends** limitava la derivazione di una classe da una sola classe base, l'operatore **implements** ci consente di implementare una classe da quante interfacce desideriamo, semplicemente elencando le interfacce da implementare, separate tra loro con una virgola.

```
class nome implements interfaccia{[,interfaccia]}{
        corpo_della_classe
}
```

Questa caratteristica permette al programmatore di creare gerarchie di classi molto complesse in cui una classe eredita la natura concettuale di molte entità. Se una classe implementa interfacce multiple dovrà fornire tutte le funzionalità per i metodi definiti in tutte le interfacce.

Classi astratte

Capitano casi in cui l'astrazione offerta dalle interfacce eccede rispetto alle necessità del programmatore (nelle interfacce non si possono implementare funzionalità alcune). Per risolvere questo problema, Java fornisce un metodo per creare classi base *astratte,* ossia classi che possono essere parzialmente implementate. Le classi *astratte* possono essere utilizzate come normali classi base e rispettano le definizioni fornite per la compatibilità tra classi; tuttavia, queste classi non sono complete, e come le interfacce non possono essere allocate direttamente.

Per estendere le classi astratte, si utilizza la clausola **extends**, di conseguenza può essere utilizzata solo una classe astratta per creare nuove definizioni di classe.

Quando una classe astratta implementa una interfaccia, non è obbligata a fornire una implementazione di tutti i metodi definiti. Questo meccanismo fornisce la scappatoia alla costrizione imposta dalle interfacce di doverne implementare tutte le funzionalità all'interno di un'eventuale nuova definizione di classe aumentando ulteriormente la flessibilità del linguaggio nella creazione delle gerarchie di derivazione.

Per definire una classe astratta Java mette a disposizione la parola chiave **abstract**. Questa clausola informa il compilatore che, alcuni metodi della classe potrebbero essere semplicemente prototipi o astratti.

```
abstract class nome
{
   attributi
   metodi_astratti
   metodi_non_astratti
}
```

Ogni metodo che rappresenta semplicemente un prototipo deve essere dichiarato **abstract**, utilizzando la sintassi seguente:

```
[private|public] abstract tipo_di_ritorno nome(argomento [,argomento] );
```

Quando una classe deriva da una classe base astratta il compilatore richiede che tutti i metodi astratti siano definiti. Se la necessità del momento costringe a non definire questi metodi, la nuova classe dovrà a sua volta essere definita **abstract**.

Immaginiamo ora di avere le due classi Sfera e Cubo:

```
public class Sfera {
  private double raggio;
  private double pesoSpecifico;

  public Sfera(double raggio, double ps) {
    this.raggio = raggio;
    pesoSpecifico = ps;
  }

  public double volume() {
    return 4 / 3 *
        Math.PI * Math.pow(raggio, 3);
  }

  public double superficie() {
    return 4 * Math.PI * raggio * raggio;
  }

  public double peso() {
    return pesoSpecifico * volume();
  }

}
```

```
public class Cubo {
  private double lato;
  private double pesoSpecifico;

  public Cubo(double lato, double ps) {
    this.lato = lato;
    pesoSpecifico = ps;
  }

  public double volume() {
    return Math.pow(lato, 3);
  }

  public double superficie() {
    return 6 * lato * lato;
  }

  public double peso() {
    return pesoSpecifico * volume();
  }

}
```

Le due classi hanno molte cose in comune quindi, potremmo pensare di generalizzarle a partire da una classe *Solido* che dovrebbe contenere:

1. *un pesoSpecifico;*
2. *un metodo volume();*
3. *un metodo superficie();*
4. *un metodo peso();*

Utilizzare una classe base potrebbe essere più che sufficiente, basterebbe utilizzare l'*overriding* di metodi per specializzarli successivamente.

Come si calcola però la superficie di un solido generico?

Ovviamente non esiste una formula generica quindi, in alternativa, dovrei rimuovere il metodo *superficie()* dalla classe base *Solido* per poi demandare alle sotto classi l'implementazione del metodo mancante mediante *overloading* con il rischio di introdurre inconsistenze nel disegno delle classi creando gerarchie deboli. Cosa analoga per il metodo *volume()*.

Utilizzando le classi astratte posso dichiarare il metodo *superficie()* astratto e demandare alle sottoclassi l'implementazione obbligatoria del codice.

La classe astratta *Solido* è definita come segue:

```
public abstract class Solido {
  private double pesoSpecifico;

  protected Solido(double ps) {
    pesoSpecifico = ps;
  }

  public double peso() {
    return volume() * pesoSpecifico;
  }

  public abstract double volume(); // metodo astratto

  public abstract double superficie(); // metodo astratto

}
```

A seguire le classi Sfera e Cubo ottenute a partire dalla classe base astratta:

```
public class Sfera extends Solido {
  private double raggio;

  public Sfera(double raggio, double
                pesoSpecifico) {
    super(pesoSpecifico);
    this.raggio = raggio;
  }

  public double volume() {
    return 4 / 3 * Math.PI
* Math.pow(raggio, 3);
  }
```

```
public class Cubo extends Solido {
  private double lato;

  public Cubo(double lato,
     double pesoSpecifico) {
    super(pesoSpecifico);
    this.lato = lato;
  }

  public double volume() {
    return Math.pow(lato, 3);
  }

  public double superficie() {
```

```
  public double superficie() {
    return 4 * Math.PI * raggio * raggio;
  }

}
```

```
    return 6 * lato * lato;
  }

}
```

Metodi di default

Come abbiamo detto, i normali metodi di interfaccia sono dichiarati come astratti e devono essere definiti obbligatoriamente in ogni classe che implementerà l'interfaccia dando l'onere al programmatore di implementare tutti i metodi. Ancora più importante, proprio per questo motivo non è più possibile implementare nuovi metodi nell'interfaccia dopo che è stata pubblicata; in caso contrario tutti i programmatori dovrebbero adattare la loro implementazione alle nuove interfacce interropendo la retro-compatibilità sia in termini di sorgenti che di binari.

Per questi motivi, durante la progettazione di un'applicazione, la maggior parte dei framework fornisce una classe di implementazione di base da estendere il più delle volte **astratta**. Una volta estesa possiamo sovrascrivere i metodi applicabili alla nostra applicazione.

Per risolvere questo problema, Java 8 ha introdotto il concetto di *metodi predefiniti* (**default**) che consentono alle interfacce di avere metodi con implementazione che non influenzano le classi che implementano l'interfaccia, e consentono di non modificare la retro-compatibilità binaria delle applicazioni esistenti.

Consideriamo quindi la prossima interfaccia e la sua implementazione: l'interfaccia *Autore* come parte di una *API (Application Prgramming Interface)* che abbiamo utilizzato per sviluppare una applicazione che contiene la sua implementazione *AutoreDelLibro*.

```
public interface Autore {
  public String getNomeECognome();
  public String getDataDiNascita();
}
```

```
public class AutoreDelLibro implements Autore{
  @Override
  public String getNomeECognome() {
    return "Massimiliano Tarquini";
  }
  @Override
  public String getDataDiNascita() {
    return "14 Aprile 1800";
  }
}
```

La prima versione della applicazione è stata appena rilasciata, ma ci accorgiamo che dobbiamo aggiungere dei metodi all'interfaccia. Questo ovviamente non sarebbe possibile senza dover modificare anche la classe *AutoreDelLibro* ... a meno di aggiungere nuovi metodi con una implementazione di *default*:

```
public interface Autore {
  public String getNomeECognome();
```

```
public class AutoreDelLibro implements Autore{
  @Override
```

```
    public String getDataDiNascita();
    default double getPeso(){
      return 100.0;
    }
  }
```

```
    public String getNomeECognome() {
      return "Massimiliano Tarquini";
    }
    @Override
    public String getDataDiNascita() {
      return "14 Aprile 1800";
    }
  }
```

Poiché il metodo di default contiene una implementazione di base, non saremo costretti a modificare la classe *AutoreDelLibro* che continuerà a funzionare allo stesso modo mantenendo la sua retro-compatibilità.

La sintassi per implementare i metodi predefiniti utilizza la parola chiave **default**:

```
[modificatori] default tipo_di_ritorno nome(lista_parametri_formali ){
    istruzione
    [istruzione]
}
```

Valgono le seguenti regole:

1. *Un programmatore può decidere di non implementare un metodo predefinito nella classe che implementa l'interfaccia;*

2. *Un programmatore può decidere di sovrascrivere i metodi predefiniti mediante overriding dei metodi non **final**;*

3. *I metodi predefiniti possono essere ri-dichiarati astratti all'interno di una classe astratta (ri-astrazione) costringendo la sottoclasse a re-implementare il metodo.*

Metodi statici

I metodi statici nelle interfacce sono simili ai metodi predefiniti, con l'unica differenza è che non è possibile sovrascriverli. Sono quindi utili in tutti quei casi in cui vogliamo implementare metodi che non vogliamo vengano riscritti nelle classi che implementano l'interfaccia.

Inoltre, i metodi statici nelle interfacce consentono di raggruppare metodi di utilità correlati tra loro, senza dover creare classi di utilità artificiali che sono semplicemente contenitori per metodi statici.

La definizione di un metodo statico all'interno di un'interfaccia è identica alla definizione di un metodo statico in una classe.

Interfacce Inner

Come per le classi, anche le interfacce possono essere definite all'interno della definizione di una classe. In questo caso, per analogia, parleremo di *interfacce inner*.

```
public class ClasseEsterna{

  interface interfacciaAnnidata{
    public void metodo1();
    .....
   }
}
```

Classi anonime

E' arrivato il momento di definire l'ultimo tipo di classe nidificata: le *classi anonime*. Le classi anonime sono un meccanismo di java che consente di rendere il codice più conciso e leggibile consentendo di *dichiarare ed instanziare* una classe allo stesso tempo, oltre che a fornire un ulteriore meccanismo per incapsulare codice a livello di classe. Sono simili alle classi locali con la differenza che non hanno nome e non possono quindi essere riutilizzate.

La loro definizione è strettamente dipendente alla definizione delle interfacce: da qui il motivo per cui abbiamo rimandato la trattazione a questa sezione.

Nei capitoli precedenti abbiamo già utilizzato alcuni classi nella loro forma anonima definendo un oggetto anonimo nel modo seguente:

DEFINIZIONE: un oggetto è anonimo quando è creato utilizzando l'operatore ***new*** *omettendo la specifica del tipo dell'oggetto ed il nome dell'identificatore.*

```
String interoComeStringa = (new Integer(10).toString());
```

Da non confondere assolutamente con i concetti che vedremo in questo paragrafo.

A differenza delle classi locali, che hanno bisogno della parola chiave **class** per essere costruite, le classi anonime vengono create come parte di una espressione grazie alla sintassi estesa dell'operatore **new**.

Le classi anonime possono essere create a partire mediante ereditarietà da classi esistenti oppure implementando interfacce. La sintassi per creare una classe anonima estendendo una classe esistente è la seguente:

new *nome_della_classe ([lista_argomenti]) {corpo-della-classe }*

dove nome_della_classe rappresenta la classe da estendere e *lista_argomenti* la lista dei parametri formali di uno dei costruttori della classe.

Nel caso di classi anonime implementando interfacce la sintassi cambia leggermente: Poiché le interfacce non hanno metodo costruttore, non sarà possibile specificare la lista dei parametri formali, ci si limiterà pertanto ad utilizzare il costruttore nullo.

new *nome_della_interfaccia ( ) {corpo-della-classe }*

Una volta creata una classe anonima, sarà possibile utilizzare una variabile reference per poterla utilizzare.

Le classi anonime si comportano quindi esattamente come le classi *locali*, dalle quale si distinguono solamente per la sintassi utilizzata per la loro definizione e creazione, e come le classi locali, hanno accesso alle variabili locali ed i parametri formali del metodo in cui sono contenute a patto che essi siano dichiarati **final**.

Nel prossimo esempio, l'applicazione *ClasseAnonima* crea una classe anonima per ereditarietà a partire dalla classe inner *HelloWorldAnonimo*.

```
public class ClasseAnonima {
  class HelloWorldAnonimo {
    public void display() {
      System.out.println("hello world!");
    }
  }

  public void eseguiTest(){
    HelloWorldAnonimo test = new HelloWorldAnonimo(){
      @Override
      public void display(){
        System.out.println("hello world dalla classe anonima!");
      }
    };
    test.display();
  }

}
```

La versione basata su interfacce è la seguente:

```
public class ClasseAnonima {
  interface HelloWorldAnonimo {
    public void display();
  }

  public void eseguiTest(){
    HelloWorldAnonimo test = new HelloWorldAnonimo(){
      @Override
      public void display(){
        System.out.println("hello world dalla classe anonima!");
      }
```

```
        };
        test.display();
    }
}
```

Alcune regole per creare classi anonime:

1. Non possono avere membri statici eccetto per le costanti ***static final.***

Se fatta all'interno di una classe statica, la dichiarazione

```
static int y = 0;
```

produrrà un errore in fase di compilazione.

2. La sintassi delle classi anonime non ci consente di implementare interfacce multiple.

3. Poiché non hanno un nome, non possono essere classi dichiarate ***abstract****.*

4. Possono accedere alle variabili nello stesso scope di appartenenza purché siano dichiarate ***final****.*

Non esistono linee guida specifiche per l'utilizzo delle classi anonime. Le uniche linee guida disponibili sono relative allo stile di scrittura:

La parentesi graffe di apertura della definizione della classe non dovrebbe essere isolate su una riga, ma invece seguire la parentesi tonda di chiusura nell'operatore ***new****. Analogamente l'operatore* ***new*** *deve, quando possibile, apparire sulla stessa riga dell'espressione di assegnazione di cui fa parte.*

Il corpo della classe anonima deve essere indentato rispetto al punto di inizio della linea che contiene l'operatore ***new****.*

La parentesi graffa che chiude una classe anonima non dovrebbe essere isolata su una riga, ma dovrebbe essere seguito dal resto dell'espressione che la contiene. Spesso si tratta di un punto e virgola o una parentesi chiusa seguita da un punto e virgola. Questo punteggiatura indica al lettore che questo non un blocco di codice ordinaria e rende più facile identificare classi anonime.

Quando usare classi anonime o classi locali?

La decisione è generalmente una questione di stile del programmatore. In generale la scelta dovrebbe ricadere sulla soluzione che rende il codice più leggibile o favorisce il disegno delle classi.

Final vs Effectively final

Abbiamo appena affermato che classi anonime possono accedere alle variabili nello stesso scope di appartenenza purché siano dichiarate **final**. Eppure, consideriamo il prossimo esempio:

```
public class EffectivelyFinal {
  interface Operatore{
    public default int eseguiQualcosa();
  }

  public static void main(String[] args) {
    int x = 10; //non è dichiarata final
    int y = 20;  //non è dichiarata final

    Operatore somma = new Operatore(){
      @Override
      public int eseguiQualcosa(){
        return x+y;
      }
    };
  }
}
```

Nonostante le variabili x e y non siano state dichiarate **final**, il codice viene compilato correttamente e l'esecuzione si comporta come atteso.

Non appena modifichiamo il codice nel modo seguente:

```
public class EffectivelyFinal {
  interface Operatore{
    public default int eseguiQualcosa();
  }

  public static void main(String[] args) {
    int x = 10; //non è dichiarata final
    int y = 20;  //non è dichiarata final

    Operatore somma = new Operatore(){
      @Override
      public int eseguiQualcosa(){
        x=y; //il compilatore torna un errore
```

```
            return x+y;
        }
      };
    }
}
```

Il compilatore tonerà un errore del tipo:

Local variable x defined in an enclosing scope must be final or effectively final

Il concetto di *effectively final* è stato introdotto solo recentemente e dipende dalla capacità del compilatore di dedurre, qualora una variabile non sia **final**, se il valore assegnato al momento della inizializzazione non venga modificato ovvero, di capire se la variabile si comporta, a tutti gli effetti, come una variabile **final**.

Nonostante funzioni, se una variabile può essere di tipo **final** meglio dichiararlo piuttosto che lasciare al compilatore l'onere di ritenerla *effectively final*. Questo perché il compilatore Java utilizza le variabili dichiarate **final** per effettuare ottimizzazioni del codice che, altrimenti, non saranno mai prese in considerazione.

Enumerazioni ed interfacce

Completiamo questa sezione aggiungendo qualche dettaglio al funzionamento dei tipi enumerativi.

Abbiamo già visto come aggiungere costruttori, dati e metodi ad una classe di tipo **enum**; quando dobbiamo aggiungere alcuni metodi attinenti al significato della enumerazione, e questi metodi richiedono una implementazione specifica per ogni tipo costante, possiamo utilizzare le interfacce come mostrato nel prossimo esempio:

```
public interface PrototipoOperatori {
  public double esegui(double operando1, double operando2);
}


public enum OperazioniSuTipiDouble implements PrototipoOperatori {
  SOMMA {
    @Override
    public double esegui(double operando1, double operando2) {
      return operando1 + operando2;
    }

  },
  SOTTRAI{
    @Override
    public double esegui(double operando1, double operando2) {
```

```
        return operando1 - operando2;
      }
    },
    MOLTIPLICA{
      @Override
      public double esegui(double operando1, double operando2) {
        return operando1 * operando2;
      }
    };

}
```

14.Programmazione dichiarativa: annotazioni

Introduzione

La programmazione dichiarativa è un paradigma di programmazione in cui il programmatore definisce ciò che deve essere realizzato dal programma senza preoccuparsi di come deve essere implementato. In altre parole l'approccio si concentra sull'obiettivo deve essere raggiunto invece di cercare di istruire su come raggiungerlo.

Il linguaggio SQL è un ottimo esempio di linguaggio dichiarativo: mediante una query sql il programmatore dichiara le sua necessità demandando al motore l'onere di preoccuparsi come fare:

*select * from tabella where campo in {} ...*

Nonostante Java sia un linguaggio imperativo, nel suo lungo percorso per arricchire lo spazio concettuale del linguaggio ed aumentarne il potere espressivo abbracciando alcuni principi propri della programmazione dichiarativa (vedremo nei capitoli successivi che la ricerca di espressività non si ferma alla programmazione dichiarativa).

Esiste una classe speciale di interfacce, chiamate annotazioni, che introdotte a partire da Java 5, sono diventate nel tempo uno strumento essenziale in tanti ambiti della programmazione con Java.

Mediante le annotazioni, Java offre al programmatore la possibilità di specificare informazioni relative a determinate entità senza dover ricorrere a descrittori esterni oppure inutili commenti, offre un maggior controllo di errori a compile-time, riduce la quantità di codice aumentando nel contempo la semplicità di utilizzo.

In questa sezione vedremo in dettaglio cosa sono le annotazioni, come dichiararle e come utilizzarle. Parleremo inoltre di meta-annotazioni e vedremo quali sono quelle più comunemente utilizzate.

Cosa sono le annotazioni

Per definire le annotazioni utilizziamo la vecchia classe base *Veicolo*:

```
public class Veicolo {

 private String tipo;
 private int velocita;
 private int direzione;
 public static final int DRITTO = 0;
 public static final int SINISTRA = -1;
 public static final int DESTRA = 1;

 public Veicolo() {
  velocita = 0;
```

```
 direzione = Veicolo.DRITTO;
 tipo = "Veicolo generico";
}


public void muovi() {
 muovi(1);
}
public void muovi(int velocita) {
 this.velocita = velocita;
 System.out.println(tipo + " si sta movendo a: " + velocita + " Kmh");
}
public void ferma() {
 velocita = 0;
 System.out.println(tipo + " si è fermato");
}
public void svoltaSinistra() {
 direzione = Veicolo.SINISTRA;
 System.out.println(tipo + " ha sterzato a sinistra");
}
public void svoltaDestra() {
 direzione = Veicolo.DESTRA;
 System.out.println(tipo + " ha sterzato a destra");
}
public void procediDiritto() {
 direzione = Veicolo.DRITTO;
 System.out.println(tipo + " sta procedendo in linea retta");
}
public String getTipo() {
 return tipo;
}
public int getVelocita() {
 return velocita;
}
public int getDirezione() {
 return direzione;
}

public void setVelocita(int velocita) {
 this.velocita = velocita;
}

public void setTipo(String tipo) {
 this.tipo = tipo;
}

public void setDirezione(int direzione){
 this.direzione = direzione;
}
```

```
}
```

Definiamo quindi la classe *Macchina* come sottoclasse di *Veicolo*. Poiché la macchina sarà leggermente diversa da *Veicolo* (una specializzazione), decidiamo di modificare il metodo *muovi(int velocita)* della superclasse mediante il meccanismo di *overriding*:

```
public class Macchina extends Veicolo {
  public Macchina() {
    setVelocita(0);
    setDirezione(Veicolo.DRITTO);
    setTipo("Macchina");
  }

  public void segnala() {
    System.out.println(getTipo() + "ha attivato il segnalatore acustivo ");
  }

  public void muovi(double velocita) {
    System.out.println(getTipo() + "si sta movendo alla velocit consentita di:"
+ getVelocita() + " Kmh");
  }
}
```

La compilazione andrà a buon fine e non ci accorgiamo che abbiamo introdotto un errore in quanto, nonostante l'intenzione fosse quella di modificare il metodo muovi mediante l'overriding, la firma del metodo modificato *muovi(double velocita)* non corrisponde a quella del metodo della classe base *muovi(int velocita),* in definitiva quindi il metodo della superclasse non è stato modificato senza che il compilatore ci segnalasse l'errore commesso.

Se avessimo segnalato al compilatore l'intenzione di effettuare l'overriding di un metodo della classe base le cose sarebbero andate diversamente ed il compilatore avrebbe potuto segnalare l'anomalia.

L'annotazione *@Override* serve proprio a questo. Se infatti modifichiamo il metodo della sottoclasse annotandolo nel modo giusto:

```
@Override
 public void muovi(double velocita) {
    System.out.println(getTipo() + "si sta movendo alla velocit consentita di:"
    + getVelocita() + " Kmh");
  }
```

il compilatore Java segnalerà un errore:

```
error: method does not override or
       implement a method from a supertype
    @Override
```

Abbiamo usato la annotazione *@Override* per indicare l'intenzione del programmatore di modificare il metodo della classe base: il compilatore compresa l'intenzione del programmatore, ha controllato che il nuovo metodo modificasse effettivamente il metodo della superclasse segnalando di conseguenza un errore.

In sostanza, abbiamo usato una annotazione per documentare il codice, e allo stesso tempo abbiamo consentito al compilatore di rinforzare i controlli su alcuni metodi risparmiando tempo anche per il debugging del codice.

Le annotazioni sono quindi degli strumenti che servono per aggiungere *metadati* a sezioni della nostra applicazione: packages, moduli, classi ma anche metodi o variabili membro di una classe. Sono *interfacce* con informazioni aggiuntive che possono essere utili per definire valori, comportamenti, ma non solo. Non modificano l'applicazione, si limitano a documentare l'elemento a cui sono agganciate.

A differenza della documentazione standard, scritta in un linguaggio comprensibile solo all'uomo, le annotazioni sono comprensibili sia all'uomo che al compilatore e possono essere utilizzate quindi non solo per documentare il codice ma anche per consentire al compilatore di prendere decisioni sulla base delle stesse.

Le annotazioni, a differenza dei commenti, sono disponibili al run-time e quindi possono essere utilizzate per scopi specifici o per una moltitudine di usi: il codice assomiglia sempre più a quello di un linguaggio dichiarativo piuttosto che imperativo.

Per concludere quindi, le annotazioni sono interfacce speciali che hanno una moltitudine di funzioni: documentazione, verifica e rinforzo da parte del compilatore, validazione a run-time, generazione di codice da parte di frameworks , controllo del run-time.

Definire annotazioni

Dal momento che le annotazioni sono qualcosa di più che semplici commenti, la loro creazione deve necessariamente sottostare ad alcune regole. Di fatto sono delle interfacce particolari, e come tale rispettano le regole di sintassi e semantica già viste per classi ed interfacce a meno di alcune distinzioni.

La sintassi generale per la definizione delle annotazioni è la seguente:

*[**public**|**protected**|package friendly] **@interface** nome_annotazione {*
[elemento_della_annotazione]
}

elemento_della_annotazione = <tipo> <nome_elemento>();
[elemento_della_annotazione]

Come le interfacce possono essere dichiarate pubbliche o visibili a livello di package; *nome_annotazione* rappresenta il nome della annotazione.

La definizione di una annotazione è soggetta alle seguenti restrizioni:

1. Una annotazione non può ereditare da un altra annotazione. Ogni annotazione implementa implicitamente l'interfaccia java.lang.annotation.Annotation.

2. I metodi di una annotazione non possono specificare dati membro, non possiedono parametri ne clausole throws dal momento che non sono chiamati ad effettuare nessuna operazione.

I metodi delle annotazioni si comportano banalmente come variabili membro e servono solo ad associare alla istanza della annotazione un valore di un tipo.

3. Il tipo del valore di ritorno dei metodi di una annotazione è compreso tra i seguenti:

a) *tipi primitivi (byte, short, int, long, char, boolean, float, double);*

b) *String;*

c) *Class (parleremo dei tipi .class successivamente);*

d) *enum;*

e) *array dei tipi sopra riportati;*

f) *un tipo annotazione;*

A differenza delle interfacce in cui possiamo dichiarare metodi di default o metodi statici, le annotazioni non lo consentono, e questo perché un metodo, per definizione, è un qualcosa che contiene alcune logiche mentre le annotazioni sono qualcosa che rappresentano semplicemente i valori degli elementi (metadati) a corredo della annotazione.

Costruiamo la nostra prima annotazione; potremmo per esempio decidere di annotare classi e metodi della applicazione sviluppata dal nostro team con informazioni sul programmatore, la versione, la data. Possiamo per esempio creare una annotazione *RilasciataDa* come segue:

```
public @interface RilasciataDa {
   String nome();
   String cognome();
   String version();
   String data();
}
```

che possiamo utilizzare per annotare la classe *ApplicazioneAnnotata*:

```
@RilasciataDa(nome = "Massimiliano", cognome = "Tarquini", version = "0.0.1", data = "15/06/2022")
public class ApplicazioneAnnotata {
   public static void main(String[] args) {
      System.out.println("Questa è una applicazione java annotata");
   }
}
```

Il processo di annotazione è semplice: una volta definita l'annotazione in un file *RilasciataDa.java* (per la nomenclatura dei file vale la solita regola Java) può essere immediatamente utilizzata per annotare il codice. La sintassi è la seguente:

@nome_annotazione({[nome_elemento = valore_elemento]})

Qualche giorno dopo aver rilasciato la *ApplicazioneAnnotata* viene richiesto al team di fare una modifica evolutiva e, questa volta, è il vostro collega a modificare la classe. Ancora una volta possiamo utilizzare la annotazione già creata per documentare le attività svolte annotando questa volta porzioni di codice, metodi, e non solo la classe principale:

```
@RilasciataDa(nome = "Massimiliano", cognome = "Tarquini", version = "0.0.1", data = "15/6/ 2022")
public class ApplicazioneAnnotata {

  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", version = "0.0.1", data = "20 giugno 2022")
  private int parametro1;

  public static void main(String[] args) {
    System.out.println("Questa è una applicazione java annotata");
  }

  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", version = "0.0.1", data = "20 giugno 2022")
  public int getParametro1() {
    return parametro1;
  }

  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", version = "0.0.1", data = "20 giugno 2022")
  public void setParametro1(int parametro1) {
    this.parametro1 = parametro1;
  }
}
```

Esistono diversi tipi di annotazioni. Nei prossimi paragrafi le analizzeremo in dettaglio.

A differenza delle interfacce, le annotazioni posso essere istanziate e persistere a run-time. Tuttavia, per le annotazioni non è previsto l'utilizzo dell'operatore **new**. La dichiarazione

@nome_annotazione({[nome_elemento = valore_elemento]})

equivale a chiamare il costruttore e creare una istanza della annotazione.

Ad esempio, la prossima riga crea una istanza del tipo *RilasciataDa*.

```
@RilasciataDa(nome = "Andrea", cognome = "Di Paolo", version = "0.0.1", data = "20/06/22")
```

Aggiungere valori di default alle annotazioni

A differenza della dichiarazione delle interfacce, la sintassi generale per la definizione di annotazioni prevede anche la possibilità di aggiungere agli elementi di una annotazione dei valori predefiniti. A seguire la sintassi completa:

```
[public|protected|package friendly] @interface nome_annotazione {
        [elemento_della_annotazione]
        }

elemento_della_annotazione = <tipo> <nome_elemento>() [default valore_elemento];
        [elemento_della_annotazione]
```

Potremmo quindi modificare la nostra annotazione aggiungendo un valore **default** all'elemento version:

```
public @interface RilasciataDa {
  String nome();
  String cognome();
  String version() default "0.0.1";
  String data();
}
```

A meno di voler specificare valori differenti, avendo dichiarato un valore di default per l'elemento *version*, possiamo modificare la applicazione omettendo l'elemento dalla annotazione:

```
@RilasciataDa(nome = "Massimiliano", cognome = "Tarquini", data = "15 giugno 2022")
public class ApplicazioneAnnotata {
  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", data = "20 giugno 2022")
  private int parametro1;
  public static void main(String[] args) {
    System.out.println("Questa è una applicazione java annotata");
  }
  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", data = "20 giugno 2022")
  public int getParametro1() {
    return parametro1;
  }
  @RilasciataDa(nome = "Andrea", cognome = "Di Paolo", data = "20 giugno 2022")
  public void setParametro1(int parametro1) {
    this.parametro1 = parametro1;
  }
}
```

A seguire, il frammento di codice mostra come aggiungere valori predefiniti ad una annotazione per i diversi tipi di elementi ammessi.

```
public @interface DefaultTest {
    double doubleElement() default 12.89;
    int intElement() default 12;
    int[] arrayOfIntElement() default { 1, 2 };
    String stringElement() default "Hello";
    String[] arrayOfStringElement() default { "abc", "xyz" };
    Class classElement() default Exception.class;
    Class[] arrayOfClassElement() default { Exception.class, java.io.IOException.class };
    MesiDellAnno enumElement() default MesiDellAnno.GENNAIO;
    MesiDellAnno[] arrayOfEnumElement() default {MesiDellAnno.GENNAIO, MesiDellAnno.FEBBRAIO};
}
```

Utilizzare le annotazioni

Utilizzare le annotazioni equivale a porre l'accento sui dati: tipi e modalità di trattamento. Nei paragrafi precedenti abbiamo accennato che gli elementi di una annotazione possono essere solamente di uno dei seguenti tipi:

a) *tipi primitivi (byte, short, int, long, char, boolean, float, double);*

b) *String;*

c) *Class*

d) *un tipo enum;*

e) *array dei tipi sopra riportati;*

f) *un tipo annotazione;*

Per i tipi Class abbiamo poco da dire. E' generalmente raro vederli nelle definizioni di annotazioni ed in genere vengono utilizzati da *framework* per svolgere compiti specifici (tratteremo alcuni di questi framework nella seconda e terza sezione del libro, ci limiteremo comunque a qualche accenno ad uno dei framework più largamente utilizzati alla fine di questo capitolo).

Tutti gli altri tipi saranno discussi a breve.

Valgono le regole seguenti:

*1. Non è possibile utilizzare il valore **null** all'interno delle annotazioni;*

2. I valori da assegnare agli elementi di una annotazione devono essere costanti al compile time quindi

*3. Non è possibile utilizzare l'operatore **new** per definire un valore di **default** o assegnare un valore ad un elemento di una annotazione;*

*4. Non è possibile utilizzare variabili reference per definire un valore di **default** o assegnare un valore ad un elemento di una annotazione;*

Per costanti compile time si intendono variabili **final** valorizzate al momento della compilazione. Parliamo quindi di variabili di classe dichiarate **final** e valorizzate.

Tipi primitivi

Usare i *tipi primitivi* per definire una annotazione è abbastanza intuitivo e lo abbiamo già accennato nei paragrafi precedenti.

```
public @interface Primitivi {
  byte tipoByte() default 1+2;
  short tipoShort() default Short.MIN_VALUE;
  int tipoInt() default Integer.MAX_VALUE;
  long tipoLong() default 12345678;
  float tipoFloat() default 7.8F;
  double tipoDouble()  default 7.8;
  char tipoChar() default 'y';
  boolean tipoBoolean() default true;
}
```

Posso ottenere una istanza della annotazione utilizzando i valori di default (se disponibili) oppure impostare nuovi valori: sono ammessi operatori aritmetici, costanti, l'operatore di cast.

```
@Primitivi(tipoByte=Byte.MIN_VALUE, tipoShort=(short)12.5, tipoInt = 4+6, tipoChar='A')
```

Stringhe

Per le *stringhe* valgono le stesse regole, ma possiamo utilizzare l'operatore di concatenazione.

```
public @interface Stringhe {
  String stringa2() default "test";
  String string3() default "nome"+"cognome";
}
```

```
@Stringhe(stringa2="questa è una string", stringa3="questa"+" è una stringa concatenata")
```

Il prossimo frammento di codice produrrà un errore in fase di compilazione dovuta al fatto che stiamo tentando di utilizzare una variabile reference e l'operatore **new** per assegnare un valore di default ad un elemento della annotazione.

```
public @interface Stringhe {
// la prossima riga produce un errore in fase di compilazione
String stringa1() default  new String("errore");
  String stringa2() default "test";
  String string3() default "nome"+"cognome";
}
```

Array

Una annotazione può contenere elementi di tipo array purché dei tipi ammessi per una annotazione:

1. *tipi primitivi (byte, short, int, long, char, boolean, float, double);*
2. *String;*
3. *Class*
4. *un tipo enum;*
5. *un tipo annotazione;*

Gli array devono essere specificati come costanti, e quindi utilizzando la notazione tra parentesi graffe. Di conseguenza, se provassimo ad utilizzare il metodo *values()* di una enumerazione il compilatore produrrebbe un errore interropendo la compilazione.

```
public @interface Arrays {
  String[] arrayDiStringhe() default {"stringa", "stringa"};
  MesiDellAnno[] arrayDiTipoEnum() default {MesiDellAnno.GENNAIO, MesiDellAnno.OTTOBRE};
  Class[] arrayDitipoClass() default {MesiDellAnno.class,String.class};
  int[] arraydiTipoInt();
}
```

Per ottenere un istanza della annotazione:

```
@Arrays(arraydiTipoInt={1,2,3,4,5}, arrayDiTipoEnum = {MesiDellAnno.MARZO, MesiDellAnno.GIUGNO});
```

Annotazioni comuni in Java

@Overrides: si applica ai metodi di una classe per indicare che un determinato metodo ridefinisce il corrispondente metodo ereditato da superclasse. Indica che il compilatore deve segnalare un errore se il metodo contiene errori nel nome o nei parametri formali. Senza annotazione un metodo errato sarebbe normalmente riconosciuto come un nuovo metodo della sottoclasse.

@Deprecated: si applica ai metodi ed indica al programmatore che il metodo non dovrebbe essere utilizzato in quanto non più supportato. A partire dalla versione 9 di Java introduce due elementi aggiuntivi: l'attributo *since* è una stringa che indica a partire da quale versione del prodotto o delle API il metodo è stato deprecato mentre, l'attributo *forRemoval* specifica se l'elemento sarà rimosso nel prossimo rilascio.

@SuppressWarning: si applica ad un *tipo* (classe) oppure ad un *metodo*. Disattiva la segnalazione di warning del compilatore.

Tipi di annotazioni

Esistono diversi tipi di annotazioni in Java e differiscono per tipologia, caratteristiche, utilizzo. Le annotazioni java sono suddivise in:

1. Annotazioni tipo marker

Questo tipo di annotazioni non hanno elementi, il loro contenuto informativo è definito dal nome stesso della annotazione. Sono generalmente utilizzate da strumenti, uno tra i tanti il framework *Lombok*, che utilizzano le annotazioni per generare automaticamente codice boilerplate;

2. Annotazioni single value (a valore singolo).

Un'annotazione a valore singolo contiene un solo elemento chiamato *value*.

```
@interface AnnotazioneAValoreSingolo{
    int value();
}
```

Questa classe di annotazioni consente di utilizzare una sintassi abbreviata per la specifica del valore del membro. Questo tipo di annotazioni consentono infatti di passare il valore del membro senza doverne includere il nome:

```
@AnnotazioneAValoreSingolo(10)
```

3. Annotazioni multi value (multi valore).

Sono le più comuni in java. Sono annotazioni di uso generale: possono essere utilizzate da framework, dal compilatore Java, oppure possono servire semplicemente ad aggiungere contenuto informativo al codice.

4. Meta annotazioni.

Sono annotazioni che possono essere utilizzate solo per annotare altre annotazioni. Sono parte della *Java Core API* e sono definite nel package *java.lang.annotation*. A seguire alcune delle più comuni meta-annotazioni: *@Target, @Retention, @Inherited, @Documented, @Repeatable*

Per boilerplate code si intende una sezione di codice che viene ripetuta molte volte all'interno del programma senza aggiungere nulla di concettuale alla classe.

In italiano *boilerplate code* può essere tradotto come codice verboso.

Un esempio di codice boilerplate sono i metodi *getter* e *setter*, metodi che non aggiungono nulla alla logica di funzionamento di un oggetto ma che sono comunque verbosi e ripetitivi.

@Retention

Le annotazioni possono essere utilizzati in diversi modi. Ad esempio, come abbiamo già accennato, le *Java Reflection API*, appartenenti alle Core API, consentono al programmatore di eseguire vari controlli e operazioni su classi, variabili di istanza, metodi e annotazioni durante l'esecuzione della applicazione.

In generale, la strategia di memorizzazione di una annotazione, *retention*, può essere una delle seguenti:

1. Solo codice sorgente.

L'annotazione è disponibile in fase di compilazione, ma non aggiunta ai file .class. Può essere utilizzata, ad esempio, da un processore di annotazione;

2. Solo definizioni di classe (predefinito).

La loro definizione è mantenuta nel file .class, ma la loro istanza non è disponibile al run-time;

3. Definizione di classe e run-time.

La loro definizione è mantenuta nel file della classe e la loro istanza è disponibile al run-time per essere utilizzata tramite le *Java Reflection API*.

Per specificare la *retention* di una annotazione possiamo utilizzare l'annotazione *single value @Retention*. *@Retention* contiene un unico membro di nome *value* di tipo *enum* i cui valori sono definiti in *java.lang.annotation.RetentionPolicy*.

RetentionPolicy definisce tre costanti: *SOURCE*, *CLASS* e *RUNTIME* che possono essere utilizzate per specificare una condotta (policy o retention) di tipo 1,2 oppure 3.

Se *@Retention* non viene utilizzata, una annotazione avrà uno scope limitato alle definizioni di classe (tipo 2).

Dal momenti che *RetentionPolicy.CLASS* rappresenta la strategia di memorizzazione predefinita, per utilizzare una annotazione con le *Java Refelction API* è necessario specificare esplicitamente *RetentionPolicy.RUNTIME*.

A titolo di esempio, definiamo tre annotazioni ognuna con un diversi livello di *retention*:

```
@Retention(RetentionPolicy.SOURCE)
public @interface SourceRetention {
   String value() default "Retention di tipo SOURCE";
}

@Retention(RetentionPolicy.CLASS)
public @interface ClassRetention {
   String value() default "Retention di tipo CLASS";
}


@Retention(RetentionPolicy.RUNTIME)
public @interface RuntimeRetention {
   String value() default "Retention di tipo RUNTIME";
}
```

Definiamo quindi tre classi annotate, rispettivamente, con una delle precedenti annotazioni. Rispettivamente:

```
@SourceRetention
public class SourceRetentionAnnotated {
}

@ClassRetention
public class ClassRetentionAnnotated {
}


@RuntimeRetention
public class RuntimeRetentionAnnotated {
}
```

Infine la nostra applicazione:

```
public class TestRetention {

  public static void main(String[] args) {

    Annotation souceAnnotations[] = new
          SourceRetentionAnnotated().getClass().getAnnotations();
    Annotation classAnnotations[] = new
          ClassRetentionAnnotated().getClass().getAnnotations();
    Annotation rutnimeAnnotations[] = new
          RuntimeRetentionAnnotated().getClass().getAnnotations();

    System.out.println(
        "La classe  SourceRetentionAnnotated è stata annotata: "
            + souceAnnotations.length + " volte");

    System.out.println(
        "La classe  ClassRetentionAnnotated è stata annotata: "
            + classAnnotations.length + " volte");

    System.out.println(
        "La classe  RuntimeRetentionAnnotated è stata annotata: "
            + rutnimeAnnotations.length + " volte");

  }
}
```

```
La classe  SourceRetentionAnnotated è stata annotata: 0 volte
La classe  ClassRetentionAnnotated è stata annotata: 0 volte
```

Come aspettato, al run-time le classi annotate rispettivamente con *@SourceRetention* e *@ClassRetention* non risultano avere annotazioni collegate.

@Target

Target è utilizzata per definire i contesto in cui una annotazione può essere utilizzata. E' una annotazione a singolo valore con un unico membro (*value*) di tipo *array di enumerazioni java.lang.annotation.ElementType*. Nella prossima tabella sono elencate le costanti definite nella enumerazione:

Costanti definite in java.lang.annotation.ElementType	
costante	**descrizione**
ANNOTATION_TYPE	Indica che la annotazione può essere utilizzata per annotare altre annotazioni. Rende l'annotazione una meta-annotazione.
CONSTRUCTOR	Annotazione di metodi costruttori
FIELD	annotazione di variabili di istanza
LOCAL_VARIABLE	Utilizzata per variabili locali (metodi). Non è disponibile al runtime ed è quindi utilizzata solo per direttive compile-time.
METHOD	Annotazione di metodi
MODULE	Annotazione di moduli.
PACKAGE	Annotazioni di package
PARAMETER	Annotazione di parametri formali di un metodo
TYPE	Annotazione di classi, interfacce (incluse altre annotazioni), tipi enum.
TYPE_PARAMETER	Utilizzata per annotare i tipi Java Generics[7]
TYPE_USE	Uso di un tipo

Ad esempio, la prossima annotazione può essere applicata solo a metodi:

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
        // questa annotazione può essere applicata solo a metodi
}
```

Posso anche utilizzare un array per aggiungere più valori:

7 Java generics saranno trattati in dettaglio successivamente

@Target({ElementType.FIELD, ElementType.TYPE})

A seguire alcuni esempi di utilizzo:

Esempi di utilizzo	
Tipo	**Frammento di codice con esempio**
ANNOTATION_TYPE	*@Retention(RetentionPolicy.RUNTIME)* *@interface Annotazione*
CONSTRUCTOR	*@MyAnnotation* *public CostruttoreDellaMiaClasse() {}*
FIELD	*@Attributo* *private int variabileDiIstanza;*
LOCAL_VARIABLE	*for (@VariabileDiControllo int i = 0; i < 100; i++) {* *@RisultatoAtteso* *String risultato;* *}*
METHOD	*@AnnotazionePerIl Metodo* *public void metodoAnnotato(Integer parametroIntero){* *}*
PACKAGE	*@PackageConEsempi* *package javamattone.esercizi.capitolo12.retention;*
PARAMETER	*public void metodoAnnotato(@ParametroAnnotato Integer parametroIntero){* *}*
TYPE_USE	*Object oggettoGenerico = "Questa è una stringa";* *String s = (@AnnotazioneTypeUSe String) oggettoGenerico ;*

Una annotazione applicata d una variabile locale non è mai disponibile al run-time indipendentemente dalla strategia di memorizzazione (*retention*) della annotazione, e questo perché le variabili locali non sono mai accessibile tramite *reflection* al run-time.

@Inherited

Inherited è una *meta annotazione* di tipo *marker*. Se una annotazione è annotata con *@Inherited*, la sua istanza viene ereditata nel caso in cui la classe annotata venga derivata per mezzo della ereditarietà. Non ha alcun effetto in tutti gli altri casi.

Sarà più chiaro il significato dopo aver visto il prossimo esempio. Delle due prossime annotazioni, *@Annotazione2* è annotata con *@Inherited*:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Annotazione1 {
   int value();
}
```

```
import java.lang.annotation.Inherited;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Annotazione2 {
   int value();
}
```

Consideriamo adesso la superclasse annotata con entrambe, *@Annotazione1* ed *@Annotazione2*:

```
@Annotazione1(1024)
@Annotazione2(2048)
public class SuperClasse {

}
```

Poiché *@Annotazione2* è annotata *@Inherited*, qualsiasi sottoclasse costituita a partire da questa superclasse erediterà una istanza di *@Annotazione2* con valore 2048 assegnato alla superclasse.

```
//Eredita @Annotazione2(2048) dalla superclasse
public class SottoClasse extends SuperClasse{
}
```

Verifichiamo quanto affermato utilizzando le Java Reflection API come già fatto nel casi di *@Retention*. L'esecuzione della applicazione *TestInherited* (il cui codice segue immediatamente dopo)

```
La classe  SuperClasse è stata annotata: 2 volte
La classe  SottoClasse è stata annotata: 1 volte
        interface javamattone.esercizi.annotazioni.inherited.Annotazione2
```

Dimostra che *SottoClasse* eredita da *SuoperClasse* una sola annotazione di tipo *Annotazione2*.

```
public class TestInherited {

  public static void main(String[] args) {
    Annotation[] annotazioniSuperClasse = (new SuperClasse()).getClass().getAnnotations();
    Annotation[] annotazioniSottoClasse = (new SottoClasse()).getClass().getAnnotations();
    System.out.println(
        "La classe  SuperClasse è stata annotata: "
            + annotazioniSuperClasse.length + " volte");

    System.out.println(
        "La classe  SottoClasse è stata annotata: "
            + annotazioniSottoClasse.length + " volte");

    for(Annotation annotation: annotazioniSottoClasse){
      System.out.println(annotation.annotationType());
    }
  }

}
```

@Repeatable

A volte accadono situazioni in cui vorremmo poter utilizzare la stessa annotazione per decorare una classe più di una volta. Supponiamo di voler creare un servizio che invia una mail agli utenti, e supponiamo di voler creare una applicazione che utilizza una annotazione *@Scheduler* per pianificare l'invio delle mail. Di seguito una possibile implementazione:

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Scheduler {
  int ora();
  int minuti();
  int secondi();
}
```

Potremmo utilizzare questa annotazione per decorare la classe di servizio come segue:

```
@Scheduler(ora=12, minuti=00, secondi=30)
public class Mailer {
  ....
}
```

Cosa succede però se volessimo pianificare l'invio della mail a diversi orari? E' possibile utilizzare la stessa annotazione per annotare la classe più di una volta e schedulare l'invio di mail più volte nel tempo? Ovvero, possiamo modificare il codice nel modo seguente?

```
@Scheduler(ora=12, minuti=00, secondi=30)
@Scheduler(ora=13, minuti=00, secondi=30)
public class Mailer {

}
```

Per il momento la risposta è no: in fase di compilazione il compilatore Java produrrebbe il seguente errore:

> *Duplicate annotation of non-repeatable type @Scheduler. Only annotation types marked @Repeatable can be used multiple times at one target.Java(16778113)*

Tuttavia ci fornisce un aindicazione interessante ..

> *Only annotation types marked* ***@Repeatable*** *can be used multiple times at one target.Java(16778113)*

Grazie alla annotazione @Repeatable possiamo fare in modo che il compilatore non ci dia più errori se riscrivendo la definizione della annotazione *@Scheduler* nel modo seguente:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Schedulers.class)
public @interface Scheduler {
  int ora();
  int minuti();
  int secondi();
}
```

Dove, *Schedulers.class* è il contenitore che sarà utilizzato dalla Java Virtual Machine per memorizzare tutte le ripetizioni della classe. Dovrà essere definito come una annotazione single *value*, con un unico membro di tipo Array di tipo eguale alla annotazione che andiamo a ripetere. A seguire la definizione della nuova annotazione:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Schedulers {
  Scheduler[] value();
}
```

Adesso, la prossima applicazione sarà compilata correttamente:

```
@Scheduler(ora=12, minuti=00, secondi=30)
@Scheduler(ora=13, minuti=00, secondi=30)
@Scheduler(ora=14, minuti=00, secondi=30)
public class Mailer {
  public static void main(String[] args) {
    // Getting annotation by type into an array
    Scheduler[] schedulers = Mailer.class.getAnnotationsByType(Scheduler.class);
    for (Scheduler scheduler : schedulers) {   // Iterating values
      System.out.println("Ora:"+scheduler.ora()+" Minuti: "+scheduler.minuti()
+" Secondi: "+scheduler.secondi());
    }
  }

}
```

```
Ora:12 Minuti: 0 Secondi: 30
Ora:13 Minuti: 0 Secondi: 30
Ora:14 Minuti: 0 Secondi: 30
```

Preprocessori di annotazioni

Consideriamo ad esempio la seguente classe POJO (Plain Old Java Object):

```
public class BoilerPlatePersona {
  private String nome;
  private String cognome;
  private int eta;
  private double peso;
  private String indirizzo;
  private String cap;

 //Inizio codice boilerplate
  public BoilerPlatePersona (){
  }

   public BoilerPlatePersona (String nome, String cognome, int eta, double peso, String
indirizzo, String cap) {
    this.nome = nome;
    this.cognome = cognome;
    this.eta = eta;
    this.peso = peso;
    this.indirizzo = indirizzo;
    this.cap = cap;
  }
  public String getNome() {
    return nome;
  }
```

```
    public void setNome(String nome) {
      this.nome = nome;
    }
    public String getCognome() {
      return cognome;
    }
    public void setCognome(String cognome) {
      this.cognome = cognome;
    }
    public int getEta() {
      return eta;
    }
    public void setEta(int eta) {
      this.eta = eta;
    }
    public double getPeso() {
      return peso;
    }
    public void setPeso(double peso) {
      this.peso = peso;
    }
    public String getIndirizzo() {
      return indirizzo;
    }
    public void setIndirizzo(String indirizzo) {
      this.indirizzo = indirizzo;
    }
    public String getCap() {
      return cap;
    }
    public void setCap(String cap) {
      this.cap = cap;
    }
  }
```

Un POJO (Plain Old Java Object) è una classe contenente solo variabili membro private. Oltre alle variabili membro private, contiene solo metodi *getter* e *setter* usati da queste variabili membro. Non ha un suo comportamento, tuttavia potrebbe sovrascrivere alcuni metodi come equals() o hashcode().

Nella programmazione moderna è ance identificata come DTO o Data Transfer Object.

La classe contiene la definizione dei dati membro, un elenco di metodi *getter* e *setter* per potervi accedere o modificare, due costruttori: il costruttori di default ed il costruttore che prende in input tanti parametri formali quanti sono i dati membro della classe.

Data la natura della classe, i metodi costruttori ed i metodi *getter* e *setter* non aggiungono nulla alla funzionalità della classe, forniscono un contributo minimo, ma rendono il codice decisamente poco leggibile.

Immaginate se potessimo riscrivere la stessa classe nel modo seguente:

```
@NoArgsConstructor
@AllArgsConstructor
public class Persona {
  @Getter @Setter
  private String nome;
  @Getter @Setter
  private String cognome;
  @Getter @Setter
  private int eta;
  @Getter @Setter
  private double peso;
  @Getter @Setter
  private String indirizzo;
  @Getter @Setter
  private String cap;
}
```

Abbiamo già anticipato che le annotazioni di tipo marker possono essere utilizzate da processori di annotazioni per generare in automatico codice boilerplate. Grazie alle annotazioni *@Getter, @Setter, @NoArgsConstructor e @AllArgsConstructor* potremmo per esempio utilizzare un processore di annotazioni per generare in automatico, solo al momento della compilazione, tutto il codice boilerplate, e riscrivere la definizione di classe in forma più compatta e leggibile.

I preprocessori di annotazioni sono classi Java che possono essere agganciate al compilatore, vengono chiamati in causa in fase di compilazione e tipicamente generano codice che sarà successivamente compilato in forma di byte code pronto per essere eseguito dalla Java Virtual Machine.

Nonostante i preprocessori di annotazioni non sono nello scopo di questo testo, mi sembra interessante accennare al fatto che tutti i preprocessori di annotazioni sono definiti a partire dalla classe base astratta *javax.annotation.processing.AbstractProcessor*. Il prossimo frammento di codice mostra, solo a titolo di esempio, la definizione base di un preprocessore di annotazioni.

```
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.lang.model.element.TypeElement;
```

```
public class EsempioPreprocessore extends AbstractProcessor {
  @Override
  public boolean process(Set<? extends TypeElement> arg0, RoundEnvironment arg1) {
    return false;
  }

}
```

La classe base astratta *javax.annotation.processing.AbstractProcessor* contiene tutti i metodi necessari alla gestione del ciclo di vita del preprocessore, nonché della chiamata principale all'oggetto che eseguirà fisicamente il lavoro sul sorgente.

Un processore, per poter essere agganciato al compilatore, deve essere pacchettizzato in un jar e registrato nel file incluso nel pacchetto:

META-INF/services/ javax.annotation.processing.Processor

Lombok

Tra i preprocessori di annotazioni il più famoso e caso agli sviluppatori è sicuramente *Project Lombok*. Come tutti i preprocessori di codice è uno strumento che utilizza annotazioni per generare codice, evitando di creare codice ripetitivo e difficile da manutenere.

Torniamo a considerare la classe POJO appena vista implementando tutti i metodi compresi quelli che ereditati di Object *equals(), toString(), hashCode()* :

```
public class BoilerPlatePersona {
  private String nome;
  private String cognome;

  public BoilerPlatePersona (){

  }

  public BoilerPlatePersona (String nome, String cognome) {
    this.nome = nome;
    this.cognome = cognome;

  }
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  @Override
  public String toString() {
    return "BoiledPlatePersona [nome=" + nome + ", cognome=" + cognome + "]";
```

```
    }

    public String getCognome() {
      return cognome;
    }
    public void setCognome(String cognome) {
      this.cognome = cognome;
    }

    @Override
    public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + ((nome == null) ? 0 : nome.hashCode());
      result = prime * result + ((cognome == null) ? 0 : cognome.hashCode());
      return result;
    }

    @Override
    public boolean equals(Object obj) {
      if (this == obj)
        return true;
      if (obj == null)
        return false;
      if (getClass() != obj.getClass())
        return false;
      BoiledPlatePersona other = (BoiledPlatePersona) obj;
      if (nome == null) {
        if (other.nome != null)
          return false;
      } else if (!nome.equals(other.nome))
        return false;
      if (cognome == null) {
        if (other.cognome != null)
          return false;
      } else if (!cognome.equals(other.cognome))
        return false;
      return true;
    }

  }
```

La definizione di classe contiene solo due dati membro privati, e nonostante questo abbiamo dovuto scrivere oltre 60 righe di codice solo per implementare i metodi di base: getter, setter, costruttori, metodi equals, toString ed hashcode. Grazie a Lombok possiamo ottenere il medesimo risultato in poco meno di 20 righe di codice riscrivendo la classe nel modo seguente:

```
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
```

```
@ToString
public class Persona {
  @Getter @Setter
  private String nome;
  @Getter @Setter
  private String cognome;
}
```

E' immediatamente evidente come, utilizzando uno strumento come Lombok siamo in grado di scrivere codice comprensibile, pulito e manutenibile; per non parlare del fatto che ci siamo risparmiati un sacco di tempo nel dover scrivere tutto quel codice ripetitivo: è vero che oggi lo stesso codice può essere generato automaticamente dal vostro ide, ma è altrettanto vero che la leggibilità del codice è comunque un aspetto importante della programmazione.

Scendendo un po più in dettaglio, ecco alcune delle principali annotazioni messe a disposizione dal preprocessore di annotazioni:

1. @Getter e @Setter:

inutile a dirlo, sono le annotazioni che consentono di omettere i metodi *getter* e *setter* nella definizione della nostra classe. Entrambe le annotazioni possono essere utilizzate a livello di dato membro o a livello di classe: nel secondo caso, i metodi get e set verranno generati per tutti i campi non-statici.

2. @EqualsAndHashCode:

genera automaticamente i metodi *equals* e *hashcode* per la nostra classe. È possibile configurare quali campi verranno utilizzati per l'implementazione utilizzando gli elementi *exclude* e *of* dell'annotazione oppure le annotazioni marker *lombok.EqualsAndHashCode.Exclude*.

3. @ToString:

è ovviamente la annotazione da utilizzare per aggiungere il metodo *toString* ed implementare la rappresentazione in forma di stringa del nostro oggetto. Anche in questo caso è possibile utilizzare gli elementi *exclude* e *of* dell'annotazione oppure l'annotazione *lombok.ToString.Exclude* per escludere un dato membro dalla rappresentazione stringa dell'oggetto.

Nel prossimo esempio, i dati membro vengono rimossi, rispettivamente, dal metodo *toString* e *hashCode*:

```
@Getter @Setter @lombok.ToString.Exclude
private String nome;
@Getter @Setter @lombok.EqualsAndHashCode.Exclude
private String cognome;
```

4. @NoArgsConstructor:

consente di generare il costruttore senza argomenti

5. @AllArgsConstructor

produce un costruttore sulla base di tutti i campi dichiarati nella classe.

Lombok però è molto di più. Per la documentazione completa e per conoscere le altre annotazioni disponibili è consigliato consultare pagina ufficiale del progetto.

A proposito, l'applicazione a seguire utilizza la classe *Persona* ed i suoi metodi generati dal preprocessore Lombok:

```
public static void main(String[] args) {
   Persona persona = new Persona("Massimiliano", "Tarquini");
   Persona persona2 = new Persona("Massimo", "Tarquini");
   System.out.println("Persona: "+persona.toString());
   System.out.println("Persona2: "+persona2.toString());
   System.out.println("Persona e persona2 sono uguali? "+persona.equals(persona2));
   System.out.println("HashCode di persona è: "+persona.hashCode());
   persona2.setNome("Mauro");
   persona2.setCognome("Rossi");
   System.out.println("Persona2: "+persona2.toString());
}
```

```
Persona: Persona(nome=Massimiliano, cognome=Tarquini)
Persona2: Persona(nome=Massimo, cognome=Tarquini)
Persona e persona2 sono uguali? false
HashCode di persona è: -1481671440
Persona2: Persona(nome=Mauro, cognome=Rossi)
```

15.Generics Java

Introduzione

Parlando di polimorfismo, ne abbiamo identificato diverse forme ed abbiamo esaminato come, a seconda del caso d'utilizzo, ognuna delle forme definite potesse rappresentare un vantaggio strategico per meglio rappresentare le entità che compongono la nostra applicazione ad oggetti.

Facendo un breve passo indietro ricordiamo le possibili forme di polimorfismo incontrate:

1. polimorfismo ad hoc ovvero overloading dei metodi;

2. polimorfismo per inclusione ovvero overriding dei metodi;

3. polimorfismo di forma ovvero specializzazione a partire da interfacce generiche.

Tuttavia, tutti queste forme di polimorfismo si concentrano sulla specializzazione di *classi di oggetti* intervenendo sulla forma e sulla sostanza, ma non tengono conto di alcune necessità che richiedono una differente forma di specializzazione: quella di *tipo*.

Il polimorfismo di tipo, che si aggiunge all'elenco di cui sopra, risponde alla domanda: come posso rendere implementazione di una classe indipendente dal tipo.

Nonostante Java già disponga di un oggetto universale, Object, compatibili con tutti i tipi per via della ereditarietà, vedremo presto che non è sufficiente e che, anche in situazioni banali, può diventare motivo di malfunzionamenti ed effetti secondari.

In questa sezione affronteremo quindi il polimorfismo di *tipo* ed i *generics*, che rappresentano lo strumento per implementarlo.

Il problema

Per meglio comprendere il motivo che ha spinto la SUN ad introdurre i *generics* già a partire dalla versione 1.5, torniamo a considerare la nostra solita classe *Pila*. Utilizzando la classe *Object*, possiamo creare una pila generica in gradi di gestire ogni possibile tipo.

Eccone una possibile implementazione:

```
public class Pila {
  private Object[] dati;
  private int cima;
  private int dimensioneMassima;

  public Pila() {
    this(10);
  }

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

  public void push(Object elementoDellaPila) throws StackIndexOutOfBoundsException {
    if (cima < dimensioneMassima) {
      dati[cima] = elementoDellaPila;
      cima++;
    } else {
      throw new StackIndexOutOfBoundsException(
          "La pila ha raggiunto la dimensione massima. Il valore inserito è andato perduto
           +[" + elementoDellaPila + "]",
          dati.length);
    }
  }

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

public int length(){
return dati.length;
}
}
```

Abbiamo definito una Pila generica, e,come mostrato nella applicazione di test possiamo usarla con una varietà di tipi:

```
public class TestPila {
  public static void main(String[] args) {
    Pila pilaDiStringhe = new Pila();
    Pila pilaDiInteri = new Pila();
    try {
      pilaDiStringhe.push("elemento della pila");
      pilaDiInteri.push(1);
      System.out.println("Il primo elemento della pila di stringhe è: "+pilaDiStringhe.pop());
      System.out.println("Il primo elemento della pila di interi è: "+pilaDiInteri.pop());
    } catch (StackIndexOutOfBoundsException e) {

      e.printStackTrace();
    }

  }
```

Tutto bene quel che finisce bene? Non è detto, e se proviamo ad utilizzare gli elementi della pila ce ne accorgiamo immediatamente. Modifichiamo leggermente l'applicazione di test.

```
public class TestPila2 {
  public static void main(String[] args) {
    Pila pilaDiStringhe = new Pila();
    try {
      pilaDiStringhe.push("elemento della pila");
      String primoElementoDellaPila = pilaDiStringhe.pop();
      System.out.println("Il primo elemento della pila di stringhe è: "+primoElementoDellaPila);
    } catch (StackIndexOutOfBoundsException e) {

      e.printStackTrace();
    }

  }
}
```

Ciò che otterremo è un errore in fase di compilazione:

incompatible types: java.lang.Object cannot be converted to java.lang.String

e questo perché sarà necessario effettuare un cast esplicito da Object a String.

```
....
 pilaDiStringhe.push("elemento della pila");
 String primoElementoDellaPila = (String)pilaDiStringhe.pop();
 System.out.println("Il primo elemento della pila di stringhe è: "+primoElementoDellaPila);
 ....
```

Ok! Ogni volta che devo usare un elemento della pila devo fare il cast di tip: cosa sarà mai? I problemi purtroppo non si fermano qui perché, se complichiamo la nostra applicazione ci rendiamo subito conto del reale rischio che si corre:

```
public class TestPila3 {
  public static void main(String[] args) {
    Pila pilaDiStringhe = new Pila();
    try {
      pilaDiStringhe.push("elemento della pila");
      pilaDiStringhe.push(1);
      pilaDiStringhe.push(2.5);

      for(int i=0; i<=pilaDiStringhe.length(); i++){
        String primoElementoDellaPila = (String)pilaDiStringhe.pop();
        System.out.println("Il primo elemento della pila di stringhe è: "+primoElementoDellaPila);
      }
    } catch (StackIndexOutOfBoundsException e) {

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

Avendo utilizzato Object per generalizzare la pila, abbiamo potuto inserire qualsiasi tipo di dato senza che ci fosse un controllo formale e senza errori in fase di compilazione. Non appena l'applicazione entra nel ciclo **for** e tenta di convertire il tipo numerico in una stringa, l'applicazione si interrompe con una eccezione:

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Double cannot be cast to class java.lang.String (java.lang.Double and java.lang.String are in module java.base of loader 'bootstrap')
    at javamattone.esercizi.generics.TestPila3.main(TestPila3.java:14)
```

Non potendo contare sul supporto del compilatore, per mantenere la *type-safety* al run-time l'unico modo è utilizzare l'operatore **instanceof** e controllare il tipo prima di effettuare il cast:

```
for(int i=0; i<=pilaDiStringhe.length(); i++){
  Object elementoInCima = pilaDiStringhe.pop();
  if(elementoInCima instanceof String){
    String primoElementoDellaPila = (String)pilaDiStringhe.pop();
  }else if(elementoInCima instanceof Integer){
        Integer primoElementoDellaPila = (Integer )pilaDiStringhe.pop();
      }else if(elementoInCima instanceof Double){
        DoubleprimoElementoDellaPila = (Double)pilaDiStringhe.pop();
      }
      .....
}
```

E questo per tutti i possibili tipi! E' immediatamente evidente quanta complessità bisogna gestire, e quanto *Object* sia inadatto a rappresentare situazioni anche apparentemente semplici come definire una Pila di oggetti generici.

L'unica soluzione sembra quella di poter, ad un certo punto del codice, definire il tipo che rappresenterà la *Pila* e lasciare al compilatore Java l'onere di fare per noi tutti i controlli; ovvero, costruire un oggetto indipendente dal tipo (compreso Object).

Cosa sono i java generics

Generics sono stati aggiunti a partire da Java 5 per risolvere i problemi evidenziati nel paragrafo precedente e garantire il controllo del tipo in fase di compilazione con la conseguente rimozione del rischio di *ClassCastException*.

Generics significa anche *tipo come un parametro*: l'idea è far si che i tipi in Java possano essere parametri per classi metodi ed interfacce.

DEFINIZIONE: generics consentono di creare classi che funzionano con differenti tipi di dati: un tipo che utilizza i generics è chiamato anche tipo generico.

Come conseguenza dell'introduzione dei *generics* in Java, tutte le *Java core API* della versione 1.5 furono riscritte in logica *generics* con lo scopo di fornirne un implementazione indipendente dal tipo (*type safe*).

Furono quindi introdotte le *Collection API*, ovvero un insieme di oggetti basati su *generics* che consentono di gestire collezioni di tipi, e che analizzeremo in maniera approfondita in un capitolo appositamente dedicato.

Prendiamo ora in esame la seguente definizione di classe:

```
public class GenericoVecchioStile {
  private Object tipoDaGestire;
  public static void main(String[] args) {
    GenericoVecchioStile type = new GenericoVecchioStile();
        type.setTipoDaGestire("Questa è una stringa");
          //type casting, soggetto a errori può causare ClassCastExceptio
        String stringa = (String) type.getTipoDaGestire();
  }
  public Object getTipoDaGestire() {
    return tipoDaGestire;
  }
  public void setTipoDaGestire(Object tipoDaGestire) {
    this.tipoDaGestire = tipoDaGestire;
  }
}
```

Come nel caso della Pila, utilizziamo un oggetto per cercare di rappresentare una classe generica che tuttavia, è soggetta a tutti gli errori già visti. Usando *generics* possiamo riscrivere la classe nella sua forma di tipo generico come segue:

```
public class TipoGenerico<T> {
  private T tipoDaGestire;

  public static void main(String[] args) {
        TipoGenerico<String> tipoGenerico = new TipoGenerico<>();
          tipoGenerico.setTipoDaGestire("Questa è una stringa");
          String stringa = tipoGenerico.getTipoDaGestire();

  }

  public T getTipoDaGestire() {
    return tipoDaGestire;
  }

  public void setTipoDaGestire(T tipoDaGestire) {
    this.tipoDaGestire = tipoDaGestire;
  }
}
```

Analizzando il metodo *main* si notano subito alcune differenze dal caso precedente:

1. Dobbiamo dichiarare il tipo al momento della creazione dell'oggetto:
TipoGenerico<String> tipoGenerico = new TipoGenerico<>();

2. Non abbiamo bisogno del casting di tipo;

3. Non esiste il rischio di causare una ClassCastException a runtime;

4. Se non dichiariamo il tipo al momento della creazione dell'oggetto, il compilatore produce un errore del tipo "Incorrect number of arguments for type TipoGenerico<T>; it cannot be parameterized with arguments <>".

La sintassi per la dichiarazione di un tipo generico è quindi la seguente:

```
[modificatori] class nome<T>{
    dichirazione_dei_dati
    dichirazione_dei_metodi
}
```

dove *T* rappresenta il *parametro formale di tipo* (formal type parameter).

Valgono le seguenti regole:

1. Il parametro formale di tipo T può essere usato per la dichiarazione dell'elenco dei parametri formali di un metodo nonché per la definizione del tipo tornato da un metodo.

```
public T getTipoDaGestire() {
    return tipoDaGestire;
}

public void setTipoDaGestire(T tipoDaGestire) {
    this.tipoDaGestire = tipoDaGestire;
}
```

Avendo definito i metodi *getter* e *setter* della classe, è evidente che vale anche la prossima regola:

1. Il parametro formale di tipo T può essere utilizzato per dichiarare la lista dei dati membro del tipo generico:

```
private T tipoDaGestire;
```

Nel caso in cui la lista dei parametri formali di tipo di un tipo generico debba contenere più tipi distinti, la sintassi sarà la seguente:

```
[modificatori] class nome<T,U,V, ...>{
        dichirazione_dei_dati
        dichirazione_dei_metodi
}
```

Dal momento che in Java le convenzioni sono importanti, anche per i Java *generics* sono state introdotte convenzioni sui nomi.

1. I parametri formali di tipo sono formati tutti da una sola lettera maiuscola;

2. I nomi più comunemente utilizzati per i parametri formali di tipo sono i seguenti:

a) *E - Elemento: utilizzato normalmente dalle Collections API;*

b) *K - Key (Chiave): utilizzato generalmente dai tipi Map delle java Collections API per indicare la chiave della coppia <chiave,valore>;*

c) *N - Numerico*

d) *T - Tipo*

e) *V - Valore: utilizzato generalmente dai tipi Map delle java Collections API per indicare il valore della coppia <chiave,valore>;*

f) *S,U,V etc. - rappresentano il secondo, terzo, etc. etc, tipo nella lista dei parametri formali di tipo*

Istanze di tipi generici: operatore diamond <>

Per creare una istanza di un tipo generico, possiamo utilizzare l'operatore **new** con la differenza che adesso dovremo specificare obbligatoriamente il parametro formale di tipo, ovvero il tipo referenziabile che sarà utilizzato in vece del tipo generico. La prima versione di *generics* prevedeva la seguente sintassi:

*tipo_generico identificatore<tipo[,tipo]> = **new** tipo_generico<tipo[,tipo]>();*

Succedeva purtroppo che utilizzando questa sintassi con le mappe appartenenti alle *Collection* java, creare una istanza di un tipo generico assomigliava a qualcosa di simile:

Map<String, List<Map<String, Map<String, Integer>>>> cars = new HashMap<String, List<Map<String, Map<String, Integer>>>>();

Il motivo di questa sintassi ridondante e verbosa era dovuta al fatto che all'epoca della introduzione di generics coesistevano ancora in Java definizioni di classe con tipi generici e non, e di conseguenza il compilatore doveva poter distinguere tra i due tipi.

A partire da Java 7, fu introdotto l'*operatore diamond* <>. L'operatore *diamond* consente l'inferenza di tipo al momento della compilazione. Di conseguenza la sintassi fu stata semplificata nella seguente forma:

*tipo_generico identificatore<tipo[,tipo]> = **new** tipo_generico**<>**();*

consentendo di utilizzare una forma più leggibile e compatta per la creazione di oggetti da tipi generici:

*Map<String, List<Map<String, Map<String, Integer>>>> cars = **new HashMap<>();***

Per utilizzare l'operatore diamond, l'unica regola è la seguente:

1. Quando creiamo una istanza di un tipo generico, i parametri formali di tipo possono essere solo di un tipo referenziabile. Non possono essere utilizzati tipi primitivi.

Metodi Generici

Esistono situazioni in cui non vogliamo che una intera classe sia un tipo generico: a volte è sufficiente limitarci ad alcuni metodi come nel caso di metodi statici. La sintassi per dichiarare un metodo generico è la seguente:

```
[public|private|protected|static|package-friendly] <T,U,V, . . .> tipo nome(parametri_fromali){
        corpo_del_metodo
}
```

Dove *tipo* è il tipo di ritorno del metodo, *<T,U,V, . . .>* è la lista dei *parametri formali di tipo*, *nome* è il nome del metodo, *parametri_formali* è l'elenco dei parametri formali.

Nel prossimo esempio, la classe *MetodiGenericiDemo* contiene la definizione del metodo generico *isEqualTo* che effettua la comparazione di due tipi generici passati come parametri formale.

```
public class MetodiGenericiDemo<T> {
  private T membroGenerico;

  public static <T> boolean isEqualTo(T tipoGenerico1, T tipoGenerico2) {
    return tipoGenerico1.equals(tipoGenerico2);
  }

  public MetodiGenericiDemo(T membroGenerico) {
    this.membroGenerico = membroGenerico;
  }

  public static void main(String[] args) {

    MetodiGenericiDemo<String> metodiGenericiDemo =
          new MetodiGenericiDemo<>("sono un metodo generico");

     //primo metodo di invocazione del metodo statico isEqualTo
     boolean isEquals = MetodiGenericiDemo.<String>isEqualTo("stringa1", "stringa2");
     //questo secondo metodo è equivalente al primo
     isEquals = MetodiGenericiDemo.isEqualTo("stringa1", "stringa2");
  }
}
```

Nell'esempio, le due invocazioni del metodo generico *isEqualto* sono equivalenti:

```
boolean isEquals = MetodiGenericiDemo.<String>isEqualTo("stringa1", stringa2");
isEquals = MetodiGenericiDemo.isEqualTo("stringa1", "stringa2");
```

cosa possibile grazie al fatto che il compilatore è in grado di determinare il tipo per inferenza.

Come nel caso delle classi generiche, non è possibile usare tipi primitivi per tipi parametrici.

Poiché i metodi costruttori sono metodi speciali, anche i metodi costruttori possono essere dichiarati generici.

Interfacce generiche

Un'interfaccia generica è molto simile a qualsiasi altra interfaccia. Può essere utilizzata per dichiarare variabili, può essere restituita da un metodo come prototipo di un tipo, può essere passata come argomento. Anche per le interfacce generiche vale la regola dell'ereditarietà multipla.

Ciò che differenzia interfacce generiche dalle altre interfacce è che, come facile immaginare, consentono di utilizzare dati e metodi astratti.

La sintassi per la dichiarazione di una interfaccia generica è la seguente:

```
[modificatori] interface nome < T,U,V,W...> {
    dichirazione_dei_dati
    dichirazione_dei_metodi_astratti;

}
```

Consideriamo l'interfaccia generica definita di seguito:

```
public interface InterfacciaGenerica<T> {
  public void setDato(T dato);
  public T getDato();
}
```

Esistono tre modi per implementare un interfaccia:

1. Creando una classe generica;

```
public class ClasseGenerica<T> implements InterfacciaGenerica<T>{
  private T dato;
  @Override
   public void setDato(T dato) {
      this.dato = dato;
  }

  @Override
    public T getDato() {
    return dato;
  }
}
```

2. Creando una classe che utilizza tipi non generici (ovvero specializzata);

```
public class ClasseGenerica implements InterfacciaGenerica<String>{
  private String dato;
  @Override
   public void setDato(String dato) {
      this.dato = dato;
  }

  @Override
    public String getDato() {
    return dato;
  }
```

```
}
```

3. Rimuovendo o ignorando i parametri formali di tipo (fortemente sconsigliato)

```
public class RimossiParametriFormalidiTipo implements InterfacciaGenerica {
  private Object dato;
  @Override
  public void setDato(Object dato) {
    this.dato = dato;
  }
  @Override
  public Object getDato() {
    return dato;
  }
}
```

La classe RimossiParametriFormalidiTipo elimina completamente i parametri formali di tipo: in questo caso Java accetta che il tipo generico venga sostituito con l'oggetto *Object* cosa fortemente sconsigliata per via dei problemi descritti all'inizio di questo capitolo.

Le regole formali per definire le interfacce generiche sono quindi le seguenti:

1. La classe che implementa un interfaccia generica deve essere anch'essa generica. Sono ammesse le due varianti già descritte. Qualsiasi altro tentativo causerà un errore in fase di compilazione;

2. Posso utilizzare una classe non generica solo dichiarando un tipo formale non generico per l'interfaccia generica;

3. La classe generica che implementa un interfaccia generica può contenere, a sua volta, altri parametri formali di tipo.

Nel prossimo esempio, la classe generica utilizza parametri formali di tipo aggiuntivi rispetto a quanto dichiarato per l'interfaccia generica:

```
public class MoltiParametriFormaliDiTipo<T,U>
        implements InterfacciaGenerica<T>{

  private T dato;
  private U altroDato;

        .......
}
```

Tipi parametrici vincolati (bounded type parameters)

Esistono delle situazioni in cui, per necessità legate al disegno applicativo, dobbiamo creare classi generiche specializzate nel trattare solo alcuni tipi. Supponiamo ad esempio di voler creare

una classe generica che manipoli solo tipi numerici. Sorge il problema di come limitare i tipi che possono essere utilizzati come argomento nell'elenco dei parametri formali di tipo.

Java consente di specificare i vincoli per una classe generica ad un tipo specifico ed alle sua sottoclassi. La sintassi è la seguente:

```
[modificatori] class nome<T extends tipo>{
        dichirazione_dei_dati
        dichirazione_dei_metodi
}
```

o, nel caso di un interfaccia generica:

```
[modificatori] interface nome< T extends tipo> {
        dichirazione_dei_dati
        dichirazione_dei_metodi_astratti;

}
```

In alternativa, qualora si volessero specificare più vincoli, sarà possibile farlo separandoli con il carattere & con la seguente sintassi:

```
[modificatori] class nome<T extends tipo & tipo [& tipo]>{
        dichirazione_dei_dati
        dichirazione_dei_metodi
}
```

o, nel caso di un interfaccia generica:

```
[modificatori] interface nome< T extends tipo & tipo [& tipo]> {
        dichirazione_dei_dati
        dichirazione_dei_metodi_astratti;

}
```

Le regole sono le seguenti:

1. i vincoli possono essere classi o interfacce: i tipi parametrici possono essere sostituiti con il tipo effettivo di una classe o di un'interfaccia.

```
//Questa è una dichiarazione valida
public class ClasseGenericaConVincoli<T extends Integer>
```

2. Nel caso di vincoli multipli si possono utilizzare classi ed interfacce limitando la classe ad un solo tipo.

```
//Questa dichiarazione genera un errore in quanto sto utilizzando due classi
public class ClasseGenericaConVincoli<T extends Integer & Double>

//La prossima è una dichiarazione valida
public class ClasseGenericaConVincoli<T extends Integer & Interfaccia1 & Interfaccia2>
```

3. Se tra i vincoli compare una classe, dovrà comparire prima delle interfacce.

Nel prossimo esempio, utilizziamo il vincolo sui parametri formali di tipo per specificare che il nostro tipo generico è imitato ai tipi numerici:

```
public class BoundedTypes<T extends Number> {
  T dato;

  BoundedTypes(T dato) {
    this.dato = dato;
  }

  public void stampa() {
    System.out.println("Il valore è: " + this.dato);
  }

  public static void main(String[] args) {
    BoundedTypes<Integer> sample1 = new BoundedTypes<Integer>(20);
    sample1.stampa();
    BoundedTypes<Double> sample2 = new BoundedTypes<Double>(20.22d);
    sample2.stampa();
    BoundedTypes<Float> sample3 = new BoundedTypes<Float>(125.332f);
    sample3.stampa();
  }
}
```

L'applicazione compila correttamente è fornisce il seguente output:

```
Il valore è: 20
Il valore è: 20.22
Il valore è: 125.332
```

Se però tentassimo di utilizzare un tipo non numerico, come ad esempio un tipo *String*, il compilatore tornerebbe subito un errore di questo tipo:

```
BoundedTypes<String> sample4 = new BoundedTypes<String>("125.332f");
```

type argument java.lang.String is not within bounds of type-variable T

Wildcard: Upper Bounded e Lower Bounded parameters

Immaginiamo per un attimo di aver definito un tipo generico *Pila<T>* e di trovarci nella situazione particolare di doverla utilizzare come parametro formale di un metodo senza conoscere a priori il tipo che dovrà rappresentare. Supponiamo di risolvere il problema nel modo seguente:

```
public class UnboundedExample {
  public static void stampaOggetto(Pila<Object> tipoSconosciuto){
    System.out.println(tipoSconosciuto);
  }
}
```

La dichiarazione è valida e l'utilizzo di *Object* non comporta problemi in quanto i metodi *push* e *pop* della pila non utilizzano metodi specifici dipendenti dal tipo (diremo in questo caso che sono indipendenti dal tipo).

In tutti questo scenario Java prevede l'utilizzo del *carattere wildcard ?* che sta ad indicare un tipo qualunque (al pari di Object). La definizione di classe nell'esempio diventa quindi:

```
public class UnboundedExample {
  public static void stampaOggetto(Pila<?> tipoSconosciuto){
    System.out.println(tipoSconosciuto);
  }
}
```

In particolare diremo quindi che:

1. Il carattere jolly (wildcard) ? può essere utilizzato per rappresentare un tipo sconosciuto (un-bounded).

Nel caso del nostro esempio diremo quindi che stiamo utilizzando una Pila di tipo sconosciuto.

L'utilizzo del *carattere jolly ? un-bounded* è utile sopratutto nei seguenti scenari:

Stiamo implementando una porzione di codice in cui utilizziamo solo metodi forniti dalla classe *Object*;

Il codice utilizza metodi della classe generica che non dipendono dal parametro firmale di tipo (es: pila.push(elemento) oppure pila.pop()).

Oltre al carattere wildcard un-bounded, Java prevede altri due caratteri jolly: *upper-bounded* e *lower-bounded*:

DEFINIZIONE: Upper-bound wildcard consente di utilizzare tutti i sotto-tipi di una classe data come parametro formale di tipo. La sintassi è la seguente:

*tipo_generico<? **extends** tipo>*

DEFINIZIONE: Lower-bounded wildcard consente di utilizzare tutti i super-tipi di una classe data come parametro formale di tipo. In questo secondo caso la sintassi è la seguente:

*tipo_generico<? **super** tipo>*

Ad esempio, se volessimo utilizzare una Pila di tutti i tipi numerici, il nostro esempio diventerà:

```
public class UnboundedExample {
  public static void stampaOggetto(Pila<? extends Number> tipoSconosciuto){
    System.out.println(tipoSconosciuto);
  }
}
```

E se invece volessimo utilizzare una Pila di soli tipo Integer, Number, Object, il codice sarà il seguente:

```
public class UnboundedExample {
  public static void stampaOggetto(Pila<? super Integer> tipoSconosciuto){
    System.out.println(tipoSconosciuto);
  }
}
```

Ereditarietà e tipi generici

Come si comportano i tipi generici nel casi di ereditarietà? Ovvero, cosa succede alla compatibilità di tipo quando utilizziamo tipi generici? Per meglio comprendere il problema facciamo un piccolo salto indietro e consideriamo il seguente frammento di codice:

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

Poiché esiste una relazione padre-figlia tra le classi A e B, grazie alla compatibilità dei tipi il prossimo frammento di codice è valido:

```
B b = new B();
A a = b;
```

Cosa succede nel caso di tipi generici? Consideriamo il nuovo frammento di codice:

```
Pila<B> pilaB = new Pila<>();
Pila<A> pilaA =pilaB;
```

In questo caso il compilatore produrrebbe un errore al momento della generazione del byte-code. In definitiva, nonostante B sia un sotto-tipo di A, Pila<B> non è un sotto-tipo di Pila<A>.

Esiste una relazione di parentela tra tipi generici?

La risposta ce la fornisce proprio il carattere *wildcard* *?*. Nella prossima immagine è schematizzato come, attraverso ? sia possibile stabilire una relazione di parentela tra tipi i tipi generici *Pila<A>* e *Pila<B>*:

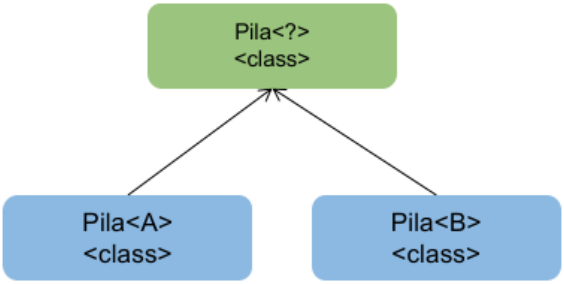


Immagine 39 - Ereditarietà e tipi generici

Pila<A> e Pila<B> hanno quindi un parente in comune che è proprio Pila<?> quindi è corretto scrivere:

```
Pila<B> pilaB = new Pila<>();
Pila<?> pilaA =pilaB;
```

Il carattere *wildcard* consente quindi di creare delle relazioni di parentela tra tipi generici sulla base del tipo dichiarato come parametro formale. Nella prossima immagine viene mostrato, data la catena do ereditarietà dei tipi numerici in Java, le relazioni di parentela determinate dall'uso del carattere *wildcard*.

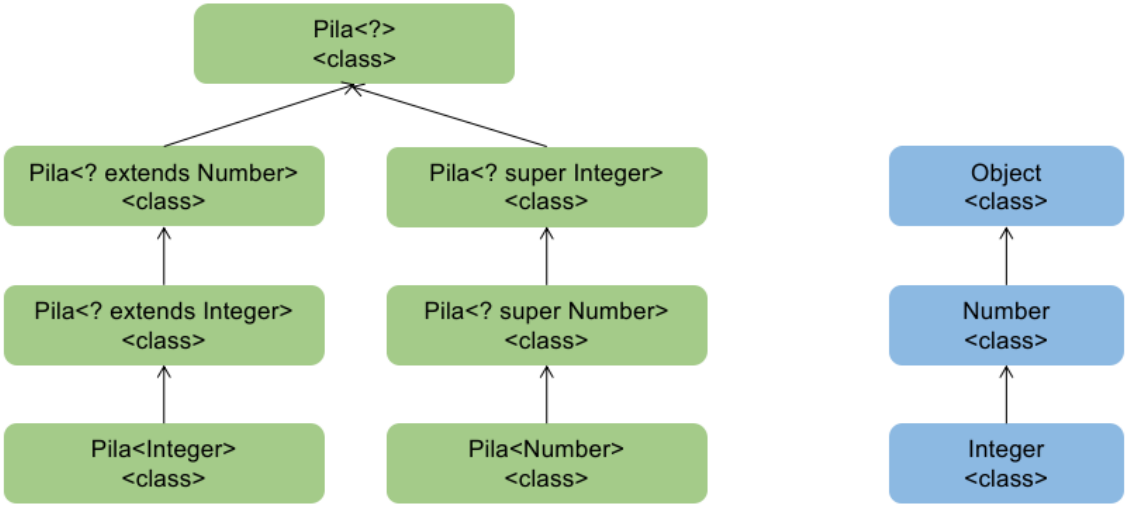


Immagine 40 Relazioni di parentela tra tipi generici tramite wildcard

Cancellazione dei tipi (type erasure)

Quando furono introdotti i generics, per supportare la programmazione generica fu modificato il compilatore Java per fornire controlli di tipo più rigorosi in fase di creazione del byte-code.

Per capire meglio come il compilatore Java tratta le i tipi generici, consideriamo la prossima definizione di classe:

```
class GenericType<T> {
  T classParam;

  GenericType(T classParam)
  {
    this.classParam= classParam;
  }
  T getClassParam()
  {
    return classParam;
  }
}
```

Quando la classe viene compilata, il compilatore sostituisce tutti i parametri di tipo generici con i relativi tipi non generici. Il processo di sostituzione dei tipi generici con tipi ordinari è detto *cancellazione dei tipi* o *type erasure*.

La classe precedente, una volta compilata sarà equivalente a:

```
class GenericType{
  Object classParam;

  GenericType(Object classParam)
  {
    this.classParam= classParam;
  }
  Object getClassParam()
  {
    return classParam;
  }
}
```

Durante il processo di cancellazione dei tipi, il compilatore provvede quindi a:

1. Sostituire tutti i parametri di tipo nei tipi generici sostituendo ogni tipo con il relativo limite (nel caso di tipi limitati) o Object nel caso di tipi illimitati. Il byte-code prodotto, quindi, contiene solo classi, interfacce e metodi ordinari;

2. Inserire appositi cast di tipo espliciti per preservare la type-safety;

3. Generare metodi ponte (bridge methods) per preservare il polimorfismo nei tipi generici con ereditarietà.

Per meglio comprendere l

a funzione dei *metodi ponte*, consideriamo ad esempio le due classi:

```
public  class  ClasseGenericaBase<T>
{
   public T data;
   public Node(T data) {
      this.data = data;
    }

   public void setData(T data) {
      this.data = data;
   }
}
```

```
public class ClasseGenericaEstesa extends ClasseGenerica<Integer>
{
public MyNode(Integer data) {
   super(data);
 }


public void setData(Integer data) {
     super.setData(data);
}
}
```

La classe *ClasseGenericaEstesa* estende la superclasse *ClasseGenericaBase* ed effettua l'override del metodo *setData*. Quando le due classi vengono compilate, per via della cancellazione dei tipi diventeranno:

```
public class ClasseGenericaBase {
   public Object data;
   public Node(Object data) {
      this.data = data;
  }

   public void setData(Object data) {
      this.data = data;
   }
}
```

```
public class ClasseGenericaEstesa extends ClasseGenerica
   public MyNode(Integer data) {
       super(data);
   }


   public void setData(Integer data) {
      super.setData(data);
   }
}
```

E' subito evidente che il metodo *setData(Integer)* non rappresenta l'override del metodo *setData(Object)* in quanto le firme dei due metodi non coincidono: il polimorfismo per i tipi generici non potrebbe quindi essere rispettato. Per risolvere il problema, e rispettare il polimorfismo anche con i tipi generici il compilatore aggiunge alla classe *ClasseGenericaEstesa* un metodo ponte che, in delega al metodo *setData(Integer)*, garantisce il corretto funzionamento della classe *ClasseGenericaEstesa* che diventa quindi:

```
public class ClasseGenericaEstesa extends ClasseGenerica
        public MyNode(Integer data) {
            super(data);
        }
        public void setData(Object data) {
            setData((Integer)data);
        }

        public void setData(Integer data) {
            super.setData(data);
        }
}
```

16.Programmazione funzionale

Introduzione

Così come nel capitolo precedente ci siamo occupati di come Java supporta la programmazione dichiarativa mediante annotazioni, in questo capitolo parleremo di programmazione funzionale. Abbiamo già accennato al paradigma funzionale ed alle funzioni pure. In questa sezione analizzeremo in dettaglio i principi base del paradigma funzionale e parleremo delle tecniche di programmazione.

Analizzeremo come sono supportate dal linguaggio, e sopratutto cercheremo di rispondere alla domanda che adesso vi sta balenando per la testa: che vantaggi può portare la programmazione funzionale ad un linguaggio puro ad Oggetti?

Per chi non ha mai programmato in funzionale alcuni concetti saranno complessi ed alcuni passaggi non esattamente immediati, ma credetemi, ne varrà la pena.

A partire da Java 8 sono state introdotte alcune caratteristiche del linguaggio che, in qualche maniera, introducono il supporto in Java alla programmazione mediante funzioni. Poiché tutto in Java è un oggetto, la realizzazione di funzioni passa attraverso le interfacce funzionali: un tipo particolare di interfacce con un solo metodo che rappresentano quanto di più vicino a funzioni il linguaggio possa offrire.

Insieme alle interfacce funzionali, è stato introdotto il concetto di interfaccia anonima che, come le classi anonime, *è una entità che ha un corpo ma non ha un nome*. Per gestire le interfacce anonime (l'operatore **new** non può essere usato per le interfacce) sono state introdotte le espressioni lambda. La sintassi delle espressioni lambda prende spunto dal lambda calcolo, un sistema formale definito nel 1936 dal matematico *Alonzo Church*, sviluppato per analizzare formalmente le funzioni e il loro calcolo.

Un tipo particolare di espressione lambda è detta *method reference,* consente di creare espressioni lambda semplici referenziando metodi esistenti. Esistono quattro tipi di *method references*:

Le interfacce funzionali Sono definite usando l'annotazione *@FunctionalInterface*, per indicare il carattere particolare dell'interfaccia, ma soprattutto per permettere al compilatore di generare errori se l'interfaccia non soddisfa i requisiti funzionali, ad esempio se contiene più di un metodo astratto. Java mette a disposizione un gran numero di interfacce funzionali predefinite.

Infine, In Java 8 è stata introdotta la nuova classe *java.util.Optional*, che è l'implementazione del tipo *Option* o *Maybe*, comune nella programmazione funzionale. Si tratta di un contenitore con un singolo valore, che può essere presente oppure no, e fornisce metodi per operare su di esso. Il valore *null* corrisponde all'assenza di valore e può essere sostituito con *Optional.empty()*, che crea un'istanza Optional vuota.

Caratteristiche e principi base della programmazione funzionale

In questo paragrafo analizzeremo alcuni principi base ed alcune caratteristiche della programmazione funzionale. Successivamente identificheremo le più interessanti, e capiremo come sono supportate nel linguaggio Java.

First-Class Function e Higher-Order Functions

Iniziamo subito con un paio di definizioni:

DEFINIZIONE: un linguaggio di programmazione si dice di tipo first-class functions se le funzioni sono trattate come variabili.

In un linguaggio di questo tipo le funzioni possono essere assegnate ad una variabile oppure passate come parametro formale ed infine tornate come valore di ritorno di una altra funzione.

Come conseguenza della definizione di *first-class function*, allora:

DEFINIZIONE: si definisce una funzione higher-order function una funzione che accetta un altra funzione come parametro formale oppure ritorna una funzione.

Funzioni pure

DEFINIZIONE: Una funzione si dice pura se, dato un insieme di parametri di input, ritornerà sempre lo stesso valore di ritorno senza produrre effetti secondari (side-effects).

L'idea è quella che ogni cosa nel codice accade all'interno di funzioni che prendono in input parametri e non utilizzano dati globali; suddividendo le funzionalità in funzioni ordinate, riutilizzabili, e con responsabilità ben definite il codice diventerà una serie di chiamate a funzioni a cui passiamo dei parametri: ogni funzione elabora i propri dati e restituisce i valori richiesti che possono essere utilizzati dal programma come parametri per altre funzioni.

Come conseguenza della propria definizione, le funzioni pure hanno alcune proprietà che vale la pena evidenziare.

1. *Le funzioni pure ritornano sempre lo stesso valore per un dato insieme di parametri di input.*

Può sembrare banale, ma non lo è. Molte funzioni potrebbero utilizzare variabili globali in grado di modificare il risultato finale sulla base del loro valore. Ad esempio potrebbero condizionare il risultato al valore della variabile globale rendendo il codice vulnerabile alla possibilità di ritornare valori differenti.

2. *Le funzioni pure non producono effetti secondar*i.

Questo significa che le funzioni pure non modificano i valori di input ricevuti ne lo stato globale della applicazione. Le funzioni pure operano quindi solo all'interno del loro contesto, non invadono territori sconosciuti e di conseguenza non creano effetti secondari inaspettati.

3. *Le funzioni pure utilizzano solo i parametri che vengono passati.*

Allo stesso modo, le funzioni pure operano solo sulle variabili che vengono loro passate come argomenti. Ciò rende le loro dipendenze più esplicite e quindi le cose più chiare sulle operazioni svolte da queste funzioni.

4. *I dati sono immutabili.*

Le funzioni pure non modificano mai i valori dei parametri di input.

La funzione mostrata nel prossimo esempio è una funzione non pura. La prima volta che la funzione viene chiamata con parametro di input 1, *sum(1)*, produrrà come valore di ritorno 1. Ma se facessimo la stessa chiamata per la seconda volta, *sum(1)* produrrebbe come valore di ritorno 2 dal momento che il valore di ritorno non dipende solamente dal parametro di input ma è influenzato dal valore della variabile globale *value*.

```
var value = 0;

function sum(n) {
      value = value + n;
    return value;

}
```

Immutabilità

E' uno dei concetti base della programmazione funzionale fa riferimento alla impossibilità di modificare una proprietà od una struttura dati senza doverne produrre una copia.

In realtà conosciamo benissimo il concetto di immutabilità, e abbiamo già analizzato in maniera approfondita quali strumenti offre Java per creare classi immutabili. Pertanto null'altro aggiungeremo in questo capitolo riguardante questo concetto.

Trasparenza Referenziale

E' forse il concetto più difficile da capire parlando di programmazione funzionale. In generale diremo che per trasparenza referenziale intendiamo quella proprietà delle funzioni pure secondo la quale una funzione può essere sostituita con il suo valore e il comportamento risultante è lo stesso di prima del cambiamento. In termini pratici, possiamo sostituire la funzione pura con il valore prodotto o con un funzione pura analoga senza introdurre effetti secondari.

Per meglio comprendere il concetto consideriamo la funzione pura somma:

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

consideriamo quindi la seguente espressione:

```
var sommaDiInteri = somma(5,5)*2;
```

Possiamo tranquillamente sostituire la funzione con il suo valore senza modificare il comportamento dell'espressione:

```
var sommaDiInteri = 10*2;
```

Sembra banale, ma non lo è. Vedremo quali sono le implicazioni della trasparenza referenziale nel linguaggi ad oggetti ed in particolare in Java.

Closures (chiusure)

Le chiusure sono un altro concetto cardine della programmazione funzionale la cui definizione non è sufficiente a comprenderne il vero significato.

DEFINIZIONE: Nei linguaggi di programmazione, una chiusura è una ***astrazione*** *che combina una funzione con le variabili libere presenti nell'ambiente in cui è definita secondo le regole di scope del linguaggio. Le variabili libere dell'ambiente rimangono accessibili per tutta la durata di vita (extent) della chiusura e pertanto persistono nel corso di invocazioni successive della chiusura (WIKIPEDIA).*

In definitiva, una *closure* è un oggetto particolare che combina la funzione e l'ambiente all'interno del quale la funzione viene definita (contesto della funzione) e può contenere, oltre la funzione, variabili appartenenti allo scope in cui la funzione è stata creata.

Tutto chiaro? Immagino di no! Cerchiamo di spiegare un po meglio il concetto di chiusura nei linguaggi funzionali, e per farlo utilizziamo il linguaggio *javascript*. Consideriamo il prossimo frammento di codice:

```
var G = 'G';
// La definizione di funzione_A crea una closure
function function_A() {
        var A = 'A'

        // La definizione di funzione_B crea una closure
        function function_B() {
                var B = 'B'
                console.log(A, B, G);
        }
        //lo scope di function_B viene creato
        functionB(); // stampa A, B, G
        //lo scope di function_B viene cancellato
        //rimane invariata la closure di function_B
        A = 42;
        //lo scope di function_B viene creato
        function_B(); // stampa 42, B, G
        //lo scope di function_B viene cancellato
        //rimane invariata la closure di function_B
}
//lo scope di function_A viene creato
function_A();
//lo scope di function_A viene cancellato
```

```
//rimane invariata la closure di function_A
```

In Javascript come in Java, possiamo dichiarare funzioni all'interno di altre funzioni: per le variabili vale la regola degli scope sintattici: una funzione può accedere alle variabili globali ed a quelle locali. Quando una funzione viene eseguita verrà prima di tutto creato il suo *scope*, tutte le variabili definite all'interno della funzione avranno vita fino a che la funzione non completerà la sua esecuzione e lo scopo della funzione distrutto.

Quindi, al di là della definizione sintattica di scope, lo scope di una funzione è qualcosa che viene creato al momento dell'esecuzione e distrutto quando la funzione ritorna.

La *chiusura*, al contrario dello scope, viene creata al momento della *definizione* della funzione (non ha nulla a vedere con la sua esecuzione) e comprende tutto ciò a cui la funzione può accedere una volta eseguita.

Nell'esempio, la definizione di *function_B* crea automaticamente una *chiusura* che consente alla funzione di accedere allo scope di *funcion_B*, quello di *function_A* ed allo scope *globale*. Ogni volta che eseguiamo *function_B* possiamo accedere alle variabili *C,A* e *G* attraverso la *chiusura* che era stata precedentemente creata, e grazie alla quale *function_B* può accedere al valore di *A* che nel frattempo è cambiato.

Tecniche di programmazione funzionale

Tutti i principi e le definizioni discusse nel paragrafo precedente hanno un impatto nelle tecniche di programmazione quando si utilizza il paradigma funzionale. Vediamo le principali:

Composizione di funzioni

Come conseguenza della definizione di *first-class functions ed higher-orderfunctions*, la composizione di funzioni fa riferimento a quelle tecnica di programmazione che consente di raggruppare funzioni semplici per creare funzioni complesse.

Ad esempio, potremmo ottenere la popolazione totale delle regioni del centro Italia combinando le due funzioni *regioniDelCentroItalia()* e *popolazionePerRegione(Regione)* come mostrato nel frammento di codice a seguire:

```
int totalePopolazioneRegioniCentroItalia = calcolaTotalePopolazione( regioniDelCentroItalia,
                                                                     popolazionePerRegione );
```

Monadi

Derivante dalla parola greca *monade*, unità indivisibile, le monadi possono essere immaginate come una sorta di wrapper che inseriscono i valori che dobbiamo trattare in un contesto, e restituiscono il valore racchiuso nel contesto stesso affinché le operazioni possano essere concatenate ..

In informatica una monade può essere definita nel modo seguente:

DEFINIZIONE: si definisce monade una classe di funzioni che hanno un solo parametro di input, e possono essere concatenate tra loro.

In sostanza, è una classe di funzioni che consente di realizzare funzioni complesse concatenando una serie di funzioni semplici in cui l'output di una diventa l'input di un'altra. Ad esempio, una struttura tipica realizzabile mediante monadi sono le *pipeline*.

DEFINIZIONE: in informatica, il concetto di pipeline (in inglese, tubatura — composta da più elementi collegati — o condotto) viene utilizzato per indicare un insieme di componenti software collegati tra loro in cascata, in modo che il risultato prodotto da uno degli elementi (output) sia l'ingresso di quello immediatamente successivo (input).

.

Immagine 41 Schema di una pipeline

Nell'immagine è schematizzata la struttura di una pipeline: i dati entrano nella pipeline e vengono processati dalla funzione F_1 il cui output diventa input della funzione successiva. Le implicazioni sono tante, ad esempio potremmo fare in modo che ogni componente di una pipeline possa procedere parallelamente alle altre: appena terminato di processare un set di dati potrebbe passare il risultato alla funzione successiva e riprendere a lavorare immediatamente un nuovo set di dati.

La pipeline dati, è una tecnologia utilizzata anche nell'architettura hardware dei microprocessori dei computer per incrementare il *throughput*, ovvero la quantità di istruzioni eseguite in una data quantità di tempo, parallelizzando i flussi di elaborazione di più istruzioni.

(WIKIPEDIA)

Currying

DEFINIZIONE: nella programmazione funzionale, il currying è la trasformazione di una funzione con più argomenti in una funzione con un argomento che restituisce una funzione sul resto degli argomenti.

In parole semplici, il *currying* è una tecnica di programmazione che consente di scomporre una funzione con n argomenti nel modo seguente:.

f(a,b,c,d) -> f(a)(b)(c)(d)

Consideriamo il prossimo esempio in *Javascript*, in particolare prendiamo a riferimento la somma di due elementi:

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

var risultato= somma(3,5);
```

Mediante la tecnica del currying possiamo riscrivere la stessa funzione nel modo seguente:

```
function somma_curried(x) {
  return function(y) {
    return x + y;
  };
}
var risultato = somma_curried(3)(5);
```

Uno dei maggiori vantaggi di questa tecnica, sopratutto nella programmazione funzionale, è che consente di creare codice sicuramente più compatto e leggibile favorendo la riusabilità come è facile vedere nel prossimo esempio:

```
const somma_parziale= sommaCurried(3);
var risultato1 = somma_parziale(10); // 13
var risultato2 = somma_parziale(15); // 18
```

Ricorsione

DEFINIZIONE: In informatica viene detto algoritmo ricorsivo un algoritmo espresso in termini di se stesso, ovvero in cui l'esecuzione dell'algoritmo su un insieme di dati comporta la semplificazione o suddivisione dell'insieme di dati e l'applicazione dello stesso algoritmo agli insiemi di dati semplificati.

Per capire meglio cosa si intende per ricorsione possiamo ricorrere al prossimo esempio. Definiamo *numero triangolare n-esimo* l'area del n-triangolo intesa come la somma delle aree dei delle sezioni quadrate.

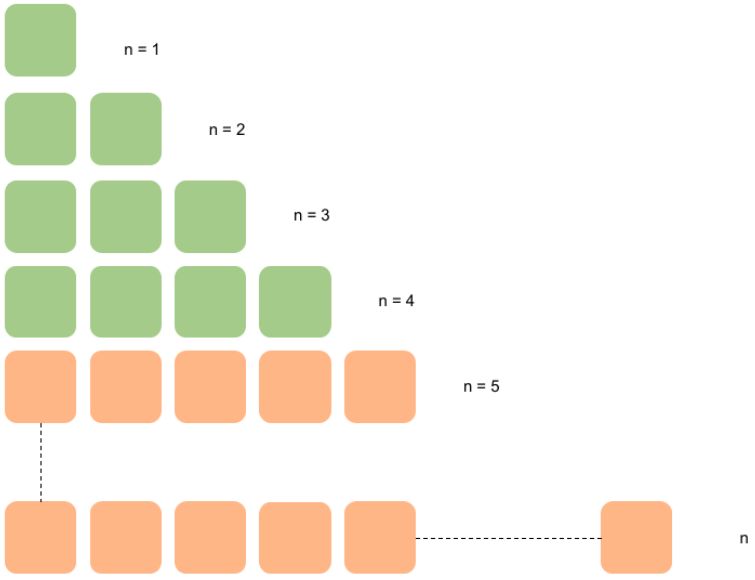


Immagine 42Ricorsione - area di una forma triangolare

Nell'ipotesi in cui ogni sezione quadrata abbia area unitaria, il numero triangolare n-esimo , per n da 1 a 4 sarà il seguente:

*n = 1 -> **1***
*n = 2 -> 1 + 2 = **3***
*n = 3 -> **3** + 3 = **6***
*n = 4 -> **6** + 4 = 10*

Ovvero, possiamo fornire una formula generale nel modo seguente:

numero triangolare n-esimo = numero triangolare (n-1) esimo + n

Andiamo a trovare una algoritmo che consenta di calcolare il n*umero triangolare n-esimo*.

Una possibile implementazione utilizzando un ciclo **for** è la seguente:

```
function numero_triangolare(int n){
        int numero_triangolare_parziale = 0;
        if(n==1) numero_triangolare_parziale = 1;
        for(int i=2; i<=n; i++){
                numero_triangolare_parziale= numero_triangolare_parziale+ i;
        }
        return numero_triangolare_parziale;
}
```

Che nella forma ricorsiva può essere riscritta nel modo seguente:

```
function numero_triangolare(int n){
        int risultato = 0;
        if(n==1)  return 1
        else  return  risultato = n+ numero_triangolare(n-1);
}
```

Quello che abbiamo fatto è:

1. Risolvere i casi base e semplificando l'insieme dei dati;

2. *combinare la soluzione con il risultato ottenuto applicando la stessa soluzione su problemi analoghi ma di dimensione inferiore (ovvero applicato ad un insieme di dati semplificato).*

La ricorsione ha un vantaggio fondamentale: permette di scrivere poche linee di codice per risolvere un problema anche molto complesso. Tuttavia, essa ha anche un enorme svantaggio: le prestazioni.

Infatti, la ricorsione genera una quantità enorme di overhead, occupando lo stack per un numero di istanze pari alle chiamate della funzione che è necessario effettuare per risolvere il problema. Funzioni che occupano una grossa quantità di spazio in memoria, pur potendo essere implementate ricorsivamente, potrebbero dare problemi a tempo di esecuzione. Inoltre, la ricorsione impegna comunque il processore in maniera maggiore per popolare e distruggere gli stack.

Rappresentazione di funzioni in Java

Prima di Java 8, per poter creare una funzione primitiva eravamo costretti a creare una classe con un unico metodo; tutto questo al costo di scrivere un sacco di codice boilerplate.

Un passo avanti in termini di semplificazione fu fatto introducendo classe ed interfacce inner, ma sopratutto le classi anonime, che come sappiamo bene potevano essere create a partire da interfacce ed avevano la seguente sintassi:

new *nome_della_interfaccia ( ) {corpo-della-classe }*

Erano la cosa più simile ad una funzione che esistesse in Java. Nel prossimo esempio utilizziamo l'interfaccia *OperatoreBinario* per definire due classi anonime *somma* e *sottrazione* che rappresentano due funzioni primitive rispettivamente per sommare e sottrarre due interi.

```
public class ClassiAnonimeComeFunzioni {
  interface OperatoreBinario{
    public int esegui(int x, int y);
  }
  public static void main(String[] args) {
    int x = 10;
    int y = 20;
    OperatoreBinario somma = new OperatoreBinario(){
      @Override
      public int esegui(int x, int y){
        return x+y;
      }
    };
    OperatoreBinario sottrazione = new OperatoreBinario(){
      @Override
      public int esegui(int x, int y){
        return x-y;
      }
    };
    System.out.println("Somma vale: "+ somma.esegui(x, y));
    System.out.println("Sottrazione vale: "+ sottrazione.esegui(x, y));
  }
}
```

Questo approccio alla rappresentazione di funzioni in java ha sicuramente una serie di vantaggi:

1. Consente di rappresentare funzioni first-class.

Come evidente dall'esempio, mediante questa sintassi possiamo trattare le funzioni così definite come delle variabili reference del tipo definito dall'interfaccia. inoltre, possiamo passare la funzione come parametro formale ad un metodo oppure creare metodi che ritornano funzioni come parametri formali come mostrato nel prossimo esempio:

```
public class FirstClassFunctions {

  interface OperatoreBinario {
    public int esegui(int x, int y);
  }

  public static OperatoreBinario operatoreSomma() {
    return new OperatoreBinario() {
      @Override
      public int esegui(int x, int y) {
        return x + y;
      }
    };
  }

  public static int eseguiOperatore(OperatoreBinario operatore, int x, int y) {
    return operatore.esegui(x,y);
  }

  public static void main(String[] args) {
    int x = 10;
    int y = 20;

    System.out.println("Somma vale: " +
                       FirstClassFunctions.operatoreSomma().esegui(x, y));
    System.out.println("Sottrazione vale: " +

    FirstClassFunctions.eseguiOperatore(new OperatoreBinario() {
      @Override
      public int esegui(int x, int y) {
        return x - y;
      }
    }
    , x, y) );
  }

}
```

2. Consentono di realizzare funzioni pure;

*3. Accedono solo ai dati locali dichiarati **final**, e di conseguenza non possono modificarli quindi*

4. Non producono effetti secondari;

5. Possono essere usate per la composizione di funzioni;

6. Rispettano la regola della trasparenza referenziale.

7. Realizzano delle chiusure

Le classi anonime operano come *closure*: catturano variabili definite nello scope sintattico e eseguono operazioni su di esse.

Interfacce funzionali

Nella direzione di introdurre il concetto di funzione in un linguaggio fortemente orientato agli oggetti (in Java tutto è un oggetto), a partire da Java 8 sono state introdotte nel Java SDK un gruppo di interfacce chiamate *interfacce funzionali:*

DEFINIZIONE: Le interfacce funzionali sono interfacce che possiedono un solo metodo astratto detto anche functional method sul quale vengono mappati i valori di input e quello di output. Le interfacce funzionali sono annotate con l'annotazione @FunctionalInterface.

Le interfacce funzionali consentono di specificare metodi di *default* statici e metodi non statici dotati di implementazione all'interno dell'interfaccia chiamati *metodi di estensione*.

@FunctionalInterface, come *@Override,* non solo denota un tipo particolare di interfaccia, ma è utilizzata dal compilatore per verificare che la struttura dell'interfaccia rispetti la definizione.

In Java esistono diverse interfacce funzionali, tuttavia il package *java.util.function* contiene la definizione di un gruppo speciale di interfacce costruite tutte a partire dal concetto matematico di funzione.

DEFINZIONE: una funzione matematica è una relazione tra due insiemi, A e B, chiamati anche dominio e codominio, che associa a ogni elemento del dominio A, uno e un solo elemento del codominio B.

Quindi, una funzione ha un suo dominio, l'insieme dei possibili valori (parametri) su cui operare ed un codominio ovvero l'insieme dei possibili valori di output prodotti.

Java supporta solo interfacce funzionali semplici (unarie e binarie) e questo perché interfacce funzionali con più argomenti sono rare da utilizzare, ma sopratutto è buona regola, qualora sia necessario, ridurre le funzioni complesse e funzioni semplici mediante il *currying*.

Eccone alcune.

Interfaccia Function<T, R>

Definisce una funzione che accetta un argomento di tipo *T* e restituisce un risultato di tipo *R*.

```
public interface Function<T, R> {
        R apply(T t);
}
```

Il suo metodo astratto è *apply*, che applica la funzione definita all'oggetto *t* passato come argomento. Possiede tre metodi di estensione; due di questi forniscono supporto alla *composizione* di funzioni

1. default <V> Function<V,R> ***compose****(Function<? super V,? extends T> before):*

restituisce una funzione composta che prima applica la funzione *before* al suo input poi applica se stessa, *apply*, al risultato ottenuto. Generalmente utile per modificare l'input prima che venga applicata la funzione *apply*.

2. default <V> Function<T,V> ***andThen****(Function<? super R,? extends V> after):*

restituisce una funzione composta che prima applica se stessa (*apply*) suo input, poi applica la funzione *after* al risultato ottenuto. In questo caso possiamo utilizzarla per modificare il *tipo* restituito da una funzione.

3. static <T> Function<T,T> ***identity****():*

Ritorna la funzione identità che restituisce sempre il su argomento.

Nell'esempio utilizziamo interfaccia *Function* per realizzare la funzione unaria *sottraiUno* che sottrae 1 all'argomento di tipo intero.

```
import java.util.function.Function;

public class FunctionExample {

  public static void main(String[] args) {

    Function<Integer,Integer> sottraiUno = new Function<>(){
      @Override
      public Integer apply(Integer arg){
        return arg-1;
      }
    };
  }
}
```

Interfaccia BiFunction<T,U, R>

Specializzazione di Function, definisce una funzione che accetta 2 argomenti, di tipo T e U, e restituisce un risultato di tipo R. Come la precedente il suo metodo astratto è *apply*.

```
public interface BiFunction<T, U, R> {
        R apply(T t, U u);
}
```

Ha un solo metodo di estensione per il supporto alla *composizione* di funzioni:

1. default <V> BiFunction<T,U,V> ***andThen****(Function<? super R,? extends V> after):*

restituisce una funzione composta che prima applica se stessa (*apply*) suo input, poi applica la funzione *after* al risultato ottenuto.

Utilizzando BiFunction, possiamo riscrivere la classe *ClassiAnonimeComeFunzioni* come segue:

```
public class ClassiAnonimeComeFunzioni {

  public static void main(String[] args) {
    int x = 10;
    int y = 20;

    BiFunction<Integer,Integer,Integer> somma = new BiFunction<>(){
      @Override
      public Integer apply(Integer a, Integer b){
        return a+b;
      }
    };

    BiFunction<Integer,Integer,Integer> sottrazione = new BiFunction<>(){
      @Override
      public Integer apply(Integer a, Integer b){
        return a-b;
      }
    };

    System.out.println("Somma vale: "+ somma.apply(x, y));
    System.out.println("Sottrazione vale: "+ sottrazione.apply(x, y));
  }

}
```

Interface BinaryOperator<T>

Derivata da *BiFunction* definisce un'operazione su due operandi dello stesso tipo T e produce un risultato sempre dello stesso tipo. Oltre al metodo ereditato da *BiFunction* definisce due metodi di estensione:

1. static <T> BinaryOperator<T> ***maxBy****(Comparator<? super T> comparator):*

ritorna un *BinaryOperator*, che restituisce l'elemento maggiore tra i due argomenti, in base al termine di confronto specificato da *comparator.*

2. static <T> BinaryOperator<T> ***minBy****(Comparator<? super T> comparator):*

ritorna un *BinaryOperator*, che restituisce l'elemento minore tra i due argomenti, in base al termine di confronto specificato da *comparator.*

Nell'esempio precedente, abbiamo utilizzato l'interfaccia *BiFunction* per definire due operatori binari somma e sottrazione che accettano tipi interi e ritornano entrambi un intero. Possiamo utilizzare l'interfaccia *BinaryOperator* per ottenere lo stesso risultato in forma più compatta:

```
public class ClassiAnonimeComeFunzioni {

  public static void main(String[] args) {
    int x = 10;
    int y = 20;

    BinaryOperator<Integer> somma = new BinaryOperator<>(){
      @Override
      public Integer apply(Integer a, Integer b){
        return a+b;
      }
    };

    BinaryOperator<Integer> sottrazione = new BinaryOperator<>(){
      @Override
      public Integer apply(Integer a, Integer b){
        return a-b;
      }
    };

    System.out.println("Somma vale: "+ somma.apply(x, y));
    System.out.println("Sottrazione vale: "+ sottrazione.apply(x, y));
  }

}
```

Interfacce Consumer<T> e Supplier<T>

Sono due interfacce funzionali complementari tra loro: la prima rappresenta una funzione che non prende dati di input ma ritorna un restituisce un risultato di tipo *T*, la seconda prende un parametro di tipo T di input ma non ritorna nessun *output*.

La definizione dell'interfaccia *Supplier* è la seguente:

```
public interface Supplier<T> {
        T get();
}
```

Non ha metodi di estensione. Al contrario, l'interfaccia *Consumer* ha la seguente definizione:

```
public interface Consumer<T> {
        void accept(T t);
}
```

ha un solo metodo di estensione per il supporto alla composizione di funzioni:

1. default Consumer<T> ***andThen****(Consumer<? super T> after):*

restituisce una funzione composta che prima applica se stessa (*accept*) suo input, poi applica la funzione *after* al risultato ottenuto.

Anche se possono sembrare strane, queste due semplici funzioni hanno un utilizzo frequente in Java in quanto su di esse si basano gli *stream* Java (a cui dedicheremo una intera sezione di questo libro).

L'interfaccia Supplier inoltre è utilizzata per supportare alcuni tipi primitivi (non referenziabili) attraverso le interfacce specializzate *LongSupplier*, *IntSupplier*, *DoubleSupplier* and *BooleanSupplier* appartenenti allo stesso package.

Una possibile implementazione di *Supplier*:

```
class RandomDigitSupplier implements Supplier<Integer> {
        @Override
        public Integer get() {
                Integer i = new Random().nextInt(10);
                return i;
        }
}
```

Interfaccia Biconsumer<T,V>

Specializzazione di *Consumer*, definisce un'operazione che accetta due argomenti di tipo *T* e *U* e non restituisce risultati. Come *Consumer*, il suo metodo astratto è *accept:*

```
public interface BiConsumer<T,U> {
        void accept(T t, U u);
}
```

Come *Consumer* ha un solo metodo di estensione:

2. default BiConsumer<T> ***andThen****(BiConsumer<? super T> after):*

restituisce una funzione composta che prima applica se stessa (*accept*) suo input, poi applica la funzione *after* al risultato ottenuto.

Interfaccia Predicate<T>

Rappresenta un predicato, cioè una funzione booleana, che riceve un solo argomento; il suo metodo astratto è test, che valuta il predicato definito sull'argomento passato:

```
public interface Predicate<T> {
        boolean test(T t);
}
```

Ha i seguenti metodi di estensione:

1. default Predicate<T> ***and****(Predicate<? super T> other):*

restituisce un predicato composto, che rappresenta il risultato dell'operatore logico *AND* su due predicati, sfruttando il *short-circuiting*, se il predicato restituisce *false*, *other* non viene valutato

2. static <T> Predicate<T> ***isEqual****(Object targetRef):*

restituisce un predicato che verifica se due argomenti sono uguali, usando *Objects.equals(Object, Object).*

3. default Predicate<T> negate():

restituisce un predicato che rappresenta la negazione logica del suo predicato

4. default Predicate<T> ***or****(Predicate<? super T> other):*

restituisce un predicato composto, che rappresenta il risultato dell'operatore logico *OR* su due predicati, sfruttando il *short-circuiting,* se il predicato restituisce *true*, *other* non viene valutato.

Il metodo statico *Objects.equals(Object a, Object b)* ritorna true solo se i due oggetti sono uguali (hanno lo stesso stato). A differenza dal metodo *equals* di Object, *Objects.equals* è un metodo *null-safe:*

1. Se entrambi i parametri sono ***null*** *ritorna true;*

2. Se il primo parametro è ***null*** *ritorna false;*

3. altrimenti ritorna il risultato dell'esecuzione a.equals(b).

Interfaccia BiPredicate<T,U>

Specializzazione di *Predicate*, rappresenta un predicato, cioè una funzione booleana, che riceve due argomenti; il suo metodo astratto è *test*, che valuta il predicato definito sui due argomenti passati:

```
public interface BiPredicate<T, U> {
        boolean test(T t, U u);
}
```

Come il tipo *Predicate* ha i metodi di estensione:

1. default BiPredicate<T,U> ***and****(BiPredicate<? super T,? super U> other):*

restituisce un predicato composto, che rappresenta il risultato dell'operatore logico *AND* su due predicati, sfruttando il *short-circuiting*, se il predicato restituisce *false*, *other* non viene valutato

2. default BiPredicate<T,U> ***negate****():*

restituisce un predicato che rappresenta la negazione logica del suo predicato

3. default BiPredicate<T,U> ***or****(BiPredicate<? super T,? super U> other)*

restituisce un predicato composto, che rappresenta il risultato dell'operatore logico *OR* su due predicati, sfruttando il *short-circuiting,* se il predicato restituisce *true*, *other* non viene valutato.

Espressioni lambda

Per rendere meno verbosa la sintassi prevista per le classi anonime, e per sfruttare appieno il potenziale offerto dalle *interfacce funzionali*, la versione 8 di Java introduce una nuova sintassi semplice e coincisa per implementare le interfacce funzionali : le *espressioni lambda.*

A differenza delle classi anonime, mediante le *espressioni lambda* non dobbiamo utilizzare l'operatore **new**, e poiché sono utilizzabili solo con interfacce aventi un solo metodo astratto, non è necessario ridefinire l'intero metodo: basterà fornirne l'implementazione.

La sintassi delle espressioni lambda è la seguente:

(lista_degli_argomenti) **->** *corpo_dell_espressione*

corpo_dell_espressione = istruzione;

oppure

```
corpo_dell_espressione = { istruzione;
                           [istruzione;]
                         }
```

dove *lista_degli_argomenti* è la lista degli argomenti da passare alla funzione e può contenere 0 o n argomenti, **->** è detto operatore *arrow*, *corpo_dell_espressione* rappresenta il corpo dell'espressione lambda.

Nel caso di interfacce funzionali con metodo astratto senza parametri utilizzeremo quindi la notazione *zero-parameters*:

() -> corpo_della_funzione;

Nel caso di metodi astratti con un parametro la sintassi diventerà:

(nome_parametro) ->corpo_della_funzione;

Nel caso di metodi astratti con due parametro la sintassi sarà (come scontato):

(nome_parametro_1, nome_parametro_2) -> corpo_della_funzione;

dove *nome_parametro* in generale rappresenta un identificatore che potremmo utilizzare nel corpo della espressione come valore di input alla funzione.

Il concetto di espressioni lambda e la sua sintassi deriva dalla branca della matematica chiamata *lambda calculus*, che si occupa di definire formalmente cosa può o non può essere calcolato. Secondo il lambda calculus, l'applicazione di una funzione viene considerata come azione primaria del calcolo e le funzioni sono viste in senso astratto, senza considerazioni su ciò che effettivamente rappresentano, ma concentrandosi sui valori. Portando questi concetti nei linguaggi di programmazione, una funzione viene definita come ciò che può essere trattato come un valore. Di conseguenza un'espressione lambda può essere assegnata ad una variabile, passata come argomento ad un metodo o manipolata come qualsiasi altro valore.

Usando la nuova sintassi, possiamo riscrivere l'esempio del paragrafo precedente nel modo seguente:

```
public class ClassiAnonimeComeFunzioni {
  public static void main(String[] args) {
    int x = 10;
    int y = 20;
    BinaryOperator<Integer> somma = (a,b)->{
        return a+b;
     };

    BinaryOperator<Integer> sottrazione= (a,b)->{
        return a+b;
     };
    System.out.println("Somma vale: "+ somma.apply(x, y));
    System.out.println("Sottrazione vale: "+ sottrazione.apply(x, y));
  }
}
```

Per le funzioni lambda valgono le seguenti proprietà:

1. Un espressione lambda definisce l'unico metodo astratto di una interfaccia funzionale;

2. Un espressione lambda viene trattata come una funzione: il compilatore non produce un file .class;

3. Un'espressione lambda può essere assegnata ad una variabile o usata come parametro formale di un metodo (first-class functions);

Poichè in java tutto è un oggetto, una espressione lambda è un oggetto. Quando viene definita, una espressione lambda crea una variabile reference ad un tipo *interfaccia funzionale,* e ne fornisce l'implementazione del metodo astratto. Ne consegue che:

4. Il tipo tornato da una espressione lambda è compatibile con l'interfaccia funzionale da cui la funzione è stata creata;

5. Un espressione lambda ha lo stesso numero di parametri richiesti dal metodo astratto definito nell'interfaccia funzionale. I parametri hanno lo stesso tipo.

Consideriamo adesso il prossimo esempio:

```
public class LambdaExample {
  public static boolean executePredicate(Predicate<String> predicato, String argomento) {
    return predicato.test(argomento);
  }

  public static void main(String[] args) {
    BinaryOperator<Integer> sottrazione = (a, b) -> {
      return a + b;
    };

    executePredicate(stringa -> stringa != null && stringa.length() > 0, "stringa non vuota");
  }

}
```

Con le espressioni lambda, il tipo dell'espressione può essere dedotto dal compilatore utilizzando il codice circostante. Ad esempio, il tipo di interfaccia funzionale può essere dedotta dalla dichiarazione del metodo (inferenza di tipo):

```
static boolean executePredicate(Predicate<String> predicato, String argomento) {
```

di conseguenza non sarà necessario specificare il tipo dell'espressione

```
executePredicate(stringa -> stringa != null && stringa.length() > 0, "stringa non vuota");
```

Nelle 'espressioni lambda è spesso possibile dedurre anche i tipi di parametro. Nell'esempio precedente, il compilatore può dedurre il loro tipo dalla dichiarazione del metodo

executePredicate. Pertanto, il tipo del parametro *stringa* viene dedotto direttamente dalla dichiarazione del metodo.

Se però assegniamo una espressione lambda ad una variabile reference, è necessario specificare il tipo dell'espressione in quanto il compilatore non è in grado di utilizzare la type inference per determinarlo automaticamente.

```
BinaryOperator<Integer> sottrazione = (a, b) -> {
    return a + b;
};
```

Tornare valori da una espressione lambda: return statement

Una espressione lambda è qualcosa che per definizione torna valori (eventualmente il tipo **void**). come per i metodi Java, una espressione lambda può utilizzare il metodo **return** come mostrato nel prossimo frammento di codice.

```
UnaryOperator<Integer> esempio = (x) -> {
    return x++;
};
```

che equivale a scrivere:

```
UnaryOperator<Integer> esempio = (x) -> x++;
```

In questo caso il compilatore può dedurre che il risultato dell'espressione x++è il valore che deve essere tornato ed accetta quindi l'omissione dell'istruzione **return**.

Se però provassimo a modificare la seconda definizione nel modo seguente, il compilatore segnalerebbe un errore e non sarebbe più possibile eseguire l'applicazione:

```
UnaryOperator<Integer> esempio2 = (x) -> {
    x++;
};
```

In definitiva quindi il compilatore non è sempre in grado di dedurre l'espressione da ritornare, e pertanto non è possibile omettere sempre l'istruzione **return**.

Le regole empiriche sono le seguenti:

*1. Se la definizione di espressione lambda contiene una sola riga di codice, allora è possibile omettere l'istruzione **return**.*

*2. Se la definizione di una espressione lambda contiene più righe di codice allora sarà necessario includere le istruzioni tra parentesi {} ed utilizzare l'istruzione **return** con un tipo compatibile con il tipo di ritorno del metodo astratto dichiarato nell'interfaccia funzionale.*

Vale anche il contrario

3. Se la definizione di una espressione lambda è inclusa tra parentesi {}, allora sarà comunque necessario utilizzare l'istruzione ***return*** *anche se la definizione contiene una sola istruzione.*

L'istruzione **return** non può essere utilizzata all'interno di una espressione lambda per terminare la normale esecuzione del codice.

Variable capture e Closure delle espressioni lambda in Java

Una espressione lambda è in gradi di acquisire un riferimento oppure il valore di variabili dichiarate al di fuori del suo blocco di definizione. Sono tre i tipi di variabili che una espressione lambda può acquisire: variabili locali, di istanza, variabili di classe (statiche).

Acquisire variabili locali

Una espressione lambda può acquisire il valore di una variabile locale dichiarata all'esterno del corpo della funzione come mostrato nel prossimo esempio:

```
String variabileLocale= "Test";

Interfacciafunzionale esempio = (chars) -> {
  return variabileLocale+ ":" + new String(chars);
};
```

Come si può vedere dall'esempio, il corpo della funzione può fare riferimento alla variabile locale *variabileLocale* che è dichiarata al di fuori del corpo dell'espressione lambda. Vale la regola seguente:

1. una espressione lambda può accedere ad una variabile locale se e solo se la variabile è final oppure effectively final.

Questo significa che il prossimo frammento di codice produrrà un errore in fase di compilazione:

```
String variabileLocale= "Test";
Interfacciafunzionale esempio = (chars) -> {
  return variabileLocale+ ":" + new String(chars);
};
variabileLocale = "nuovo valore";
```

Acquisire variabili di istanza

Un'espressione lambda può anche acquisire una variabile di istanza dell'oggetto che la ha creata.

```
public class EsempioAcquisizioneIstanza{
  private String variabileDiIstanza= "variabile di istanza";
  public void stampa(){
      Display display = (arg) ->{
            System.out.println(this.variabileDiIstanza);
            this.variabileDiIstanza = "nuovo valore";
            System.out.println(this.variabileDiIstanza);
          }
      display.stampa();
   }
}
```

L'interfaccia funzionale è definita come segue:

```
@FunctionalInterface
  private interface Display{
    public void test(String arg);
  }
```

Notare il riferimento a *this.variabileDiIstanza* all'interno del corpo della funzione: l'espressione lambda acquisisce la variabile di istanza del nome dell'oggetto *EsempioAcquisizioneIstanza* che la racchiude. A differenza delle variabili locali, è anche possibile modificare il valore della variabile di istanza dopo la sua acquisizione.

La semantica di **this** è in realtà una delle principali differenze tra espressioni lambda ed interfacce anonime. Un'implementazione di un'interfaccia anonima può avere le proprie variabili di istanza a cui fare riferimento attraverso la variabile reference **this**. Al contrario, una espressione lambda non può avere le proprie variabili di istanza, quindi **this** conterrà sempre un riferimento all'oggetto che la racchiude.

Acquisire variabili statiche

Un'espressione lambda può anche acquisire variabili statiche. Ciò non dovrebbe sorprendere, poiché le variabili statiche sono raggiungibili da qualsiasi punto di un'applicazione Java a condizione che la variabile statica sia accessibile (pubblica oppure definita nello stesso scope della funzione).

```
public class EsempioAcquisizioneStatica{
  private static String variabileStatica= "variabile statica";
  public void stampa(){
      Display display = (arg) ->{
            System.out.println(variabileStatica);
            variabileStatica= "nuovo valore";
            System.out.println(variabileStatica);
          }
      display.stampa();
```

```
    }
}
```

Come per le variabili di istanza,anche le variabili statiche possono essere modificate.

Closure

Definire una espressione lambda equivale a creare delle *closure* a run-time. Come abbiamo detto le *closure* sono degli *oggetti speciali* che possono contenere valori o reference a variabili presenti nel contesto della funzione.

Il meccanismo di creazione della *closure* è legato alla possibilità che ha la funzione di accedere alle variabili secondo le regole definite per gli scope sintattici (*variable capture*) .

Nel prossimo esempio, quando viene eseguito, il metodo *main* crea una istanza della classe, e di conseguenza viene creato lo *scope* per l'oggetto *example* di tipo *ClosureExample*.

Non appena viene chiamato il metodo *closureExample.eseguiEsempio()* viene creato lo scope del metodo.

Poichè il metodo *eseguiEsempio* contiene la definizione di una espressione lambda, viene creata la *closure* per consentire alla funzione di ottenere le informazioni sul contesto ed acquisire *variabilelocale* ed ai dati membro della classe *variabileDiIstanza*.

```
public class ClosureExample {
  public Integer variabileDiIstanza = 15;

  public void eseguiEsempio() {

    final int variabileLocale = 10;
    System.out.println("variabileLocale appartiene allo scope corrente e vale: "+variabileLocale);
    System.out.println("La closure della funzione viene creata");

    BinaryOperator<Integer> sottrazione = (a, b) -> {
      int variabileLocaleAllaFunzione = 200;
      System.out.println("Lo scope della funzione è stato creato: a e b valgono:" + a + ", " + b);
      System.out.println("variabileLocale  vale: " + variabileLocale);
      System.out.println("variabileLocaleAllaFunzione  vale: " + variabileLocaleAllaFunzione);
      System.out.println("variabileDiIstanza vale: " + variabileDiIstanza);
      variabileDiIstanza++;
      variabileLocaleAllaFunzione = variabileLocaleAllaFunzione*2;
      System.out.println("Adesso variabileDiIstanza vale: " + variabileDiIstanza);
      System.out.println("Adesso variabileLocaleAllaFunzione vale: " + variabileLocaleAllaFunzione);
      return a + b + variabileLocale;
    };

    sottrazione.apply(10, 10);
    System.out.println("Lo scope della funzione è stato distrutto");
    sottrazione.apply(20, 20);
    System.out.println("Lo scope della funzione è stato distrutto");
```

```
        System.out.println("La closure della funzione viene distrutta");
    }
}
```

Il metodo main è il seguente:

```
    public static void main(String[] args) {
        System.out.println("Lo scope per l'oggetto closureExample viene creato ");
        ClosureExample closureExample = new ClosureExample();
        System.out.println("Lo scope per il metodo closureExample.eseguiEsempio viene creato ");
        closureExample.eseguiEsempio();
        System.out.println("Lo scope per il metodo closureExample.eseguiEsempio viene eliminato ");
        System.out.println("Lo scope per l'oggetto closureExample viene eliminato ");
    }
```

Se eseguito il risultato è quello che segue. L'output è stato volutamente indentato per evidenziare ciclo di vita della closure, e degli scope man mano che vengono creati e distrutti.

```
Lo scope per l'oggetto closureExample viene creato
Lo scope per il metodo closureExample.eseguiEsempio viene creato
        variabileLocale appartiene allo scope corrente e vale: 10
        La closure della funzione viene creata
        Lo scope della funzione è stato creato: a e b valgono:10, 10
                variabileLocale  vale: 10
                variabileLocaleAllaFunzione  vale: 200
                variabileDiIstanza vale: 15
                Adesso variabileDiIstanza vale: 16
                Adesso variabileLocaleAllaFunzione vale: 400
        Lo scope della funzione è stato distrutto
        Lo scope della funzione è stato creato: a e b valgono:20, 20
                variabileLocale  vale: 10
                variabileLocaleAllaFunzione  vale: 200
                variabileDiIstanza vale: 16
                Adesso variabileDiIstanza vale: 17
                Adesso variabileLocaleAllaFunzione vale: 400
        Lo scope della funzione è stato distrutto
        La closure della funzione viene distrutta
Lo scope per il metodo closureExample.eseguiEsempio viene eliminato
Lo scope per l'oggetto closureExample viene eliminato
```

Anche se il concetto di *closure* può sembrare ovvio o addirittura banale, per meglio comprenderne l'importanza consideriamo ora il prossimo esempio in cui vedremo una *closure* in azione

```
public class ClosureExample2 {
    @FunctionalInterface
```

```
    private interface TestClosure{
      public void test(String arg);
    }

    public static void eseguiTestDellaclosure(TestClosure closureTester){
      String nomeDelMetodo = "eseguiTestDellaclosure";
      closureTester.test(nomeDelMetodo);
    }
    public static void main(String[] args) {
      final String variabileLocaleAlMetodoMain = "variabile che appartiene allo scope di main ";
      eseguiTestDellaclosure((arg)->{
        System.out.println("Sto eseguendo il metodo: "+arg);
        System.out.println("La variabile locale del metodo main vale: "+variabileLocaleAlMetodoMain);
      });
    }

  }
```

Nell'esempio, la funzione definita come argomento del metodo statico *eseguiTestDellaclosure* utilizza la variabile locale *variabileLocaleAlMetodoMain* in quanto nel suo scope.

Poiché l'esecuzione della funzione è demandata ad un metodo statico esterno al metodo *main* in cui è definita, la funzione sarà eseguita fuori dallo scope in cui è stata creata, ma sopratutto all'interno di un metodo statico che per natura non accede all'istanza della classe in cui è defnito. Tuttavia quando eseguiamo l'applicazione il risultato è il seguente:

```
Sto eseguendo il metodo: eseguiTestDellaclosure
La variabile locale del metodo main vale: variabile che appartiene allo scope di main
```

In definitiva, nonostante la funzione lambda sia stata eseguita in uno scope diverso da quello di definizione, continuerà a poter utilizzare il valore di *variabileLocaleAlMetodoMain* in quanto la sua *closure* ha tenuto memoria dei riferimenti al contesto iniziale in cui la funzione è stata creata.

L'esempio dimostra anche perché le variabili locali possono essere utilizzate solo se **final** o **effectively final**. Se utilizzata impropriamente, una funzione lambda potrebbe bypassare l'incapsulamento delle variabili locali rischiando di causare effetti indesiderati.

Currying

Abbiamo già detto che il *currying* è una tecnica che consente di trasformare una funzione con tanti argomenti in una funzione con un solo argomento. Vediamo la tecnica in azione, e per farlo faremo riferimento ad un esempio creato ad hoc per mostrare la tecnica, ma tutto sommato

lontano dal mondo reale per formulazione ed implementazione (ma per questo mi sono già scusato nel capitolo introduttivo del libro).

```
@FunctionalInterface
private interface Sommatore{
        public Integer somma(Integer a, Integer b);
}

@FunctionalInterface
private interface Incrementatore{
        public Integer incrementa(Integer a);
}
```

Per lo scopo utilizziamo le le due interfacce funzionali *Sommatore* e *Incrementatore*: la prima restituirà la somma dei due parametri passati per argomento, la seconda incrementa l'argomento. L'esempio completo è il seguente:

```
public class EsempioCurrying {

  @FunctionalInterface
  private interface Sommatore{
    public Integer somma(Integer a, Integer b);
  }

  @FunctionalInterface
  private interface Incrementatore{
    public Integer incrementa(Integer a);
  }


  public static Incrementatore curryingDiSommatore(Sommatore sommatore, int valoreDaIncrementare){
    return a -> sommatore.somma(a,valoreDaIncrementare);
  }

  public static void main(String[] args) {
    int valoreDaIncrementare = 1;
    Sommatore sommatore = (a,b) -> a+b;
    Incrementatore incrementatore = curryingDiSommatore(sommatore, valoreDaIncrementare);
    System.out.println(incrementatore.incrementa(1));
  }

}
```

Il metodo *curryingDiSommatore* prende un Sommatore ed un intero che rappresenta la quantità che utilizzerà Incrementatore per incrementare l'input, e restituisce un tipo *Incrementatore* ottenuto mediante *currying* della funzione sommatore.

Nell'esempio vengono definite due closure: quella di *sommatore* nel metodo *main* e quella di *incrementatore* nel metodo statico *curryingDiSommatore*. Se non fosse possibile creare delle *closure*, l'applicazione proposta non potrebbe funzionare in quanto la funzione sommatore utilizza la variabile *valoreDaIncrementare* definita nello scope del metodo *main*.

Espressioni lambda vs classi anonime

Anche se le classi anonime possono essere utilizzate per rappresentare funzioni ed hanno molte caratteristiche comuni con le espressioni lambda, la verità è che sono entità profondamente diverse diverse tra loro. Di fatto, le espressioni lambda non sono una semplice semplificazione sintattica per scrivere le classi anonime in maniera più concisa.

Una classe anonima è una classe inner senza un nome, per la quale viene creata una sola istanza. Può essere creata a partire da una interfaccia oppure estendendo una classe, può contenere variabili di istanza e definizioni di metodi. Utilizzano l'operatore **new,** e possono avere metodi costruttori.

Una espressione lambda al contrario è paragonabile alla definizione di un metodo anonimo pertanto non può contenere definizioni di metodi o estendere classi. La sua sintassi consente solo di implementare interfacce speciali (interfacce funzionali) che hanno solo un metodo astratto, di cui restituiscono una istanza. A differenza delle classi anonime, devono essere inserite in un contesto in cui il compilatore possa determinare i tipi mediante *type inference*.

Le differenze però non finiscono qui. Nel prossimo esempio utilizziamo l'interfaccia funzionale

```
@FunctionalInterface
private interface Display {
   public void stampaAVideo();
}
```

per definire una classe anonima ed una espressione lambda. Entrambe utilizzano la variabile reference **this** per accedere alla *variabile* definita all'interno del loro scope.

Il codice completo è il seguente:

```
public class LambdaVsAnonime {
   private final String variabile = "Questa è la variabile di istanza della classe esterna";
   @FunctionalInterface
   private interface Display {
      public void stampaAVideo();
   }
   public void eseguiEsempio() {
```

```
        Display displayAnonimo = new Display(){
          private final String variabile = "Questa è la variabile di istanza della classe anonima";

          @Override
          public void stampaAVideo(){
            System.out.println("this.variabile fa riferimento a: "+this.variabile);
          }
        };
                Display displayComeLambda = () -> System.out.println("this.variabile fa riferimento a: "+this.variabile);
        System.out.println("Eseguo la funzione definita mediante classe anonima");
        displayAnonimo.stampaAVideo();
        System.out.println("Eseguo la funzione definita mediante classe anonima");
        displayComeLambda.stampaAVideo();
      }

      public static void main(String[] args) {
        LambdaVsAnonime lambdaVsAnonime = new LambdaVsAnonime();
        lambdaVsAnonime.eseguiEsempio();
      }
    }
```

Nonostante il corpo delle due funzioni sia il medesimo, il risultato contiene una sorpresa. Vediamola:

```
Eseguo la funzione dfinita mediante classe anonima
this.variabile fa riferimento a: Questa è la variabile di istanza della classe anonima
Eseguo la funzione definita mediante classe anonima
this.variabile fa riferimento a: Questa è la variabile di istanza della classe esterna
```

Quello che si nota immediatamente è che cambia il modo in cui il compilatore tratta la variabile reference **this**: all'interno dell'espressione lambda **this** fa infatti riferimento all'istanza della classe in cui la funzione è definita mentre, nel caso della classe anonima **this** fa riferimento all'istanza stessa della classe.

In generale diremo che:

*DEFINIZIONE: In una espressione lambda, le variabili reference **this** e **super** sono dette lexically scoped.*

All'interno del corpo della funzione, **this** farà riferimento alla istanza corrente della classe che racchiude la funzione, **super** alla istanza corrente della superclasse.

Nella prossima tabella sono, le due forme sono messe a confronto:

Espressioni Lambda	Classi Anonime
Rappresenta una funzione anonima (metodo senza nome).	E' una classe senza nome.

Non può estendere classi astratte ne classi concrete.	Può estendere classi astratte a classi concrete.
Può implementare solo interfacce con un singolo metodo astratto.	Può implementare interfacce che contengono un numero arbitrario di metodi astratti.
Non può contendere variabili di istanza.	Può contenere variabili diistanza.
this e **super** sono lexically scoped.	**this** punta sempre alla istanza della classe anonima, **super** all'istanza della sua superclasse.
Risiedono nella memoria permanente della Java Virtual Machine.	Sono allocate in maniera dinamica.
Hanno generalmente prestazioni migliori delle classi anonime in quanto non necessita di costi extra per la gestione della memoria al run-time.	Sono soggette ai meccanismi di caricamento dinamico degli oggetti al run-time.
Son preferibili quando dobbiamo implementare interfacce con un solo metodo.	Sono preferibili alle funzioni quando dobbiamo implementare interfacce con molti metodi.

Method Reference

Come abbiamo detto, le funzioni lambda consentono di creare metodi anonimi, e nonostante questo ci sono spesso dei casi in cui le espressioni lambda contengono solo la chiamata ad un altro metodo.

Per tutti questi casi, Java fornisce una sintassi semplificata per eseguire il metodo chiamata *method reference*.

Vediamo un esempio:

```
@FunctionalInterface
  private interface Display {
     public void stampaAVideo(String s);
  }



Display stampante= (s) ->{
                                System.out.println(s);
                            }
```

Poiché il corpo della funzione contiene un solo metodo possiamo rimuovere le parentesi {}, e allo stesso tempo rimuovere le parentesi () che delimitano la definizione dell'unico attributo *s*. La definizione diventa quindi:

```
Display stampante= s -> System.out.println(s);
```

Dal momento che l'espressione lambda non fa altro che girare l'attributo s al metodo invocato, possiamo sostituire la chiamata al metodo con la sua reference:

```
Display stampante= s -> System.out::println
```

Da notare l'operatore **::** . Questo operatore indica al compilatore che l'istruzione contiene una *method reference.* La method reference è rappresentata quindi da tutto ciò che compare sulla parte destra dopo l'operatore **::** che chiameremo per semplicità *operatore method reference***.**

L'operatore method reference consente di referenziare diversi tipi di metodi:

1. Metodi statici

2. Metodo di istanza di un parametro

3. Metodi di istanza

4. Costruttori

Analizziamo i singoli casi in dettaglio.

Method reference con metodi statici

E' il caso più semplice di method reference. Consideriamo la seguente interfaccia con un solo metodo astratto:

```
interface Sommatore{
    public Integer somma(Integer a, Integer b);
 }
```

Una interfaccia con un singolo metodo astratto è anche detta Single Abstract Method ovvero SAM.

A seguire, la classe con il metodo statico su cui vogliamo effettuare la *method reference*:

```
public class MetodiStatici {
  public static Integer faiLaSomma(Integer a, Integer b){
    return a+b;
  }
}
```

Dal momento che la firma del metodo statico *faiLaSomma* corrisponde alla firma del metodo astratto dell'interfaccia *Sommatore*, allora possiamo creare una espressione lambda che implementa *Sommatore* e referenzia *MetodiStatici.faiLaSomma*

```
Sommatore sommatore2 = MetodiStatici::faiLaSomma;
```

che, ovviamente, equivale a scrivere:

```
Sommatore sommatore = (a,b) ->  MetodiStatici.faiLaSomma(a, b);
```

Da notare che, poiché il metodo è statico utilizzeremo il nome della classe per fare riferimento al metodo. La sintassi generale è la seguente:

```
nome_della_classe::methodo_statico
```

In generale, gli attributi dell'espressione lambda vengono passati al metodo nell'ordine esatto in cui compaiono.

Method reference con metodi di istanza

Analogamente al caso precedente, possiamo referenziare un metodo di istanza come mostrato nel prossimo esempio:

```
public class MetodiDiIstanza {

  public boolean contiene(String a, String b){
    return a.contains(b);
  }
  interface Predicato{
    public boolean contiene(String a, String b);
  }

  public void esegui(){
    Predicato predicato = this::contiene;
  }
}
```

Il metodo main:

```
  public static void main(String[] args) {
    MetodiDiIstanza istanza = new MetodiDiIstanza();
    Predicato predicato = istanza::contiene;
  }
```

Poiché la firma del metodo di istanza *contiene* corrisponde alla firma del metodo astratto dell'interfaccia *Predicato*, allora possiamo creare una espressione lambda che implementa *Predicato* e referenzia *contiene.*

A differenza del caso precedente in cui abbiamo utilizzato il nome della classe per effettuare la referenza:

*Sommatore sommatore2 = **MetodiStatici**::faiLaSomma;*

nel nostro caso, essendo un metodo di istanza dovremo utilizzare una istanza della classe e quindi una variabile reference:

```
MetodiDiIstanza istanza = new MetodiDiIstanza();
    Predicato predicato = istanza::contiene;
```

E' evidente che, se chiamata da un altro metodo di istanza potremo utilizzare la variabile reference this:

*Predicato predicato = **this**::contiene;*

In questo caso la sintassi è la seguente:

variabile_reference::methodo_di_istanza

dove

*variabile_reference = identificatore | **this** | **super***

Method reference con un metodo di istanza di un parametro

Adesso le cose si complicano un pochino perché la method reference può essere effettuata anche usando un metodo di istanza di uno degli attributi della funzione.

Procediamo per gradi partendo, come al solito, dalla definizione di una interfaccia SAM:

```
interface Modificatore{
  public String rimpiazza(String s1, String s2, String s3);
}
```

Questa volta la nostra improbabile interfaccia rappresenta un modificatore di stringhe che rimpiazza in *s1* tutte le occorrenze di *s2* con *s3*. L'espressione lambda è la seguente:

Modificatore modificatore = (s1,s2,s3) -> s1.replaceAll(s2,s3);

Utilizziamo quindi il metodo *replaceAll* dell'oggetto *String* per effettuare la modifica. La sintassi per il method reference in questo caso ci consente di sostituire la funzione con:

Modificatore modificatore = String::replaceAll;

In questo caso, il compilatore farà il match utilizzando il tipo del primo parametro, s1, ed utilizzando i restanti parametri come parametri formali della chiamata al metodo.

In questo caso la sintassi è la seguente:

parametro_formale::methodo_di_istanza

Method reference su costruttori

Poiché anche i costruttori sono metodi, dobbiamo considerare anche questo ultimo caso. In generale, possiamo effettuare una method reference al metodo costruttore di una classe utilizzando il nome della classe seguito da **::new** .

```
interface StringFactory{
  public String creaStringa(String s1);
}
```

La firma del metodo *creaStringa(String)* dell'interfaccia *StringFactory* corrisponde al costruttore *String(String)* dell'oggetto stringa. Possiamo quindi scrivere la nostra espressione lambda nel modo seguente:

```
StringFactory factory = String::new;
String prova = factory.creaStringa("prova");
```

La sintassi è la seguente:

*nome_della_classe::**new***

Consigli e best practices

Completiamo il capitolo con alcuni consigli utili.

Il contenuto del package java.util.function soddisfa la maggior parte delle necessità dei programmatori quindi, prima di riscrivere una interfaccia funzionale è consigliabile valutare di utilizzare una di quelle standard.

Se le interfacce funzionali in *java.util.function* non fossero adeguate, valutiamo se utilizzare la tecnica del currying delle funzioni riducendo la nostra funzione all'esecuzione di funzioni parziali con meno parametri come mostrato in questo capitolo.

Solo alla fine, se proprio fosse necessario creare una nuova interfaccia funzionale:

Utilizziamo sempre l'annotazione *@FunctionalInterface*.

Come abbiamo anticipato, l'annotazione *@FunctionalInterface* suggerisce al compilatore che intendiamo creare una interfaccia funzionale e quindi con un solo metodo astratto. Se per errore dovessimo aggiunger altri metodi astratti, il compilatore segnalerà l'errore preservando l'integrità del codice.

Un errore tipico è il seguente:

```
@FunctionalInterface
interface A{
        public void funzione();
}

@FunctionalInterface
interface C extends A{
}
```

Nel frammento di codice, C estende A ed eredita un unico metodo astratto. Al momento della compilazione il compilatore Java non produrrà nessun errore.

Immaginiamo ora di essere costretti a cambiare il disegno per ragioni legate alla evoluzione del software che stiamo sviluppando nel modo seguente:

```
@FunctionalInterface
interface A{
        public void funzione();
}

@FunctionalInterface
interface B{
        public void altraFunzione();
}

@FunctionalInterface
interface C extends A,B{
}
```

Poiché C estende da A e B il compilatore segnalerà che C contiene ora due metodi astratti, *funzione()* ereditato da A e *altraFunzione()* ereditato da B e quindi C non può essere considerata interfaccia funzionale. potremmo quindi risolvere il problema preservando il disegno modificando B nel modo seguente:

```
@FunctionalInterface
interface A{
        public void funzione();
}
```

```
@FunctionalInterface
interface B{
        public void funzione();
}

@FunctionalInterface
interface C extends A,B{
}
```

Dal momento che un interfaccia funzionale è una interfaccia SAM, è cattiva pratica comune superare il limite del singolo metodo astratto aggiungendo metodi **default**.

Un interfaccia funzionale rappresenta una funzione, se aggiungere metodi di default diventa una necessità allora abbiamo sbagliato qualcosa nel disegno della applicazione.

Le buone scelte architetturali dovrebbero sempre essere preferibili alle scelte di convenienza. Aggiungere metodi **default** dovrebbe essere fatto solo per ragioni di compatibilità tra versioni differenti di codice e mai per ragioni di disegno.

Nelle interfacce funzionali, ed in generale nelle interfacce, non abusare mai dei metodi di default.

E se proprio non ci interessa porre attenzione alle scelte architetturali, ricordiamo sempre che implementare oppure estendere interfacce con metodi **default** uguali può rappresentare un problema come nel prossimo esempio. Riprendendo il frammento di codice dell'esempio precedente,

```
@FunctionalInterface
interface A{
        public void funzione();
        default void metodoInComune(){
        }
}

@FunctionalInterface
interface B{
        public void funzione();
        default void metodoInComune(){
        }

}

@FunctionalInterface
interface C extends A,B{
}
```

Produrrebbe l'errore in fase di compilazione:

Duplicate default methods named metodoInComune with the parameters () and () are inherited from the types DefaultMethodAbuse.B and DefaultMethodAbuse.AJava(67109917)

Proseguendo nella narrazione, consideriamo ora il codice seguente:

```
public class Overloading {
  public interface OverloadDiMetodi {
    public void metodo(Callable<String> c) throws Exception;
    public void metodo(Supplier<String> c);
  }
  public class OverloadDiMetodiImpl implements OverloadDiMetodi {
    @Override
    public void metodo(Callable<String> c) throws Exception {
      c.call();
    }
    @Override
    public void metodo(Supplier<String> c) {
      System.out.println(c.get());
    }
  }
}
  public static void main(String[] args) {
    Overloading overloading = new Overloading();
    OverloadDiMetodiImpl demo = overloading.new OverloadDiMetodiImpl();
    String s = "stringa di esempio";
    demo.metodo(() -> s);

  }
```

Così come sono stati costruiti, il compilatore non è in grado di dedurre quale dei due metodi stiamo chiamando dalla firma. I compilatori moderni sono in grado di intervenire restituendo un messaggio di errore in fase di compilazione:

The method metodo(Callable<String>) is ambiguous for the type Overloading.OverloadDiMetodiImpl

Diversamente, un vecchio compilatore avrebbe prodotto byte-code pertanto ci saremmo accorti del problema chissà quando durante l'esecuzione del programma.

```
reference to process is ambiguous
both method process(java.util.concurrent.Callable<java.lang.String>)
in com.baeldung.java8.lambda.tips.ProcessorImpl
and method process(java.util.function.Supplier<java.lang.String>)
in com.baeldung.java8.lambda.tips.ProcessorImpl match
```

Unico soluzione al problema è un inutile ma necessario cast esplicito al momento della chiamata:

```
demo.metodo((Supplier<String>) () -> s);
```

oppure

```
demo.metodo((Callable<String>) () -> s);
```

Oppure rinominare i metodi evitando l'overloading.

Meglio evitare di eseguire l'overloading di metodi che utilizzano interfacce funzionali come parametro.

Prima di concludere non dovrebbero mancare alcuni consigli stilistici.

Il codice di una espressione lambda dovrebbe essere sempre breve ed auto-documentante.

Le espressioni lambda non sono oggetti, non definiscono concetti, rappresentano funzioni concentrandosi sull'algoritmo e non sulle forme, di conseguenza dovremmo favorire funzioni brevi (preferibilmente su una riga) a costrutti complessi e con molte righe di codice; le espressioni lambda dovrebbero descrivere in modo coinciso la funzione che rappresentano.

Sempre meglio evitare blocchi di codice all'interno di espressioni lambda utilizzando, preferibilmente, metodi *wrapper*.

Ad esempio, potremmo riscrivere il prossimo frammento di codice:

```
InterfacciaFunzionale funzione =String  parametro -> {
        String result = "qualcosa" + parametro ;
         //tante linee di codice che modificano il parametro
         return result;
};
```

In maniera più coincisa sfruttando la *closure* dell'espressione:

```
InterfacciaFunzionale funzione = String parametro -> {
        this.modificaParametro(parametro)
};
public String modificaParametro(String parametro){
```

```
        String result = "qualcosa" + parametro ;
         //tante linee di codice che modificano il parametro
        return result;
}
```

Quando possibile utilizzare sempre la *method reference*.

L'esempio precedente diventerebbe:

```
InterfacciaFunzionale funzione = String parametro -> {
        return this::modificaParametro
};
```

Quando possibile evitiamo di utilizzare il comando **return**.

Sappiamo che l'espressione lambda non sempre richiede che debba essere utilizzato return in maniera esplicita. Sarebbe più compatto scrivere il codice precedente come:

```
InterfacciaFunzionale funzione = String parametro -> {
        this::modificaParametro
};
```

Una espressione lambda di una sola riga andrebbe sempre scritta senza utilizzare parentesi graffe.

```
InterfacciaFunzionale funzione = String parametro -> this::modificaParametro;
```

Infine:

Lasciamo che sia il compilatore mediante inferenza di tipo a definire il tipo dei parametri dell'espressione lambda

```
InterfacciaFunzionale funzione = parametro -> this::modificaParametro;
```

17.Java Collections Framework

Introduzione

Molto spesso, aggiungerei praticamente sempre, si devono utilizzare collezioni di oggetti più o meno complessi, per rappresentare problemi di programmazione anche semplici. I casi possono essere molteplici e le collezioni non sempre hanno comportamenti omogeneo: la dimensione delle collezioni potrebbe non essere nota a priori, gli elementi potrebbero dover essere ordinati oppure la collezione potrebbe non dover contenere duplicati. Poi, potrebbero esserci diverse modalità di accesso: indicizzato, *LIFO (Last In First Out)* oppure *FIFO (First In First Out)* oppure attraverso una chiave.

Tutto questo ha portato allo sviluppo di tipologie diverse di collezioni, sia per quanto riguarda le funzioni offerte sia per quanto riguarda le prestazioni che possono privilegiare alcuni aspetti (ad esempio la velocità di ricerca per collezioni che cambiano poco oppure la velocità di inserimento per collezioni i cui elementi cambiano spesso) a discapito di altri.

Nel corso degli anni, il *framework* è stato arricchito fino a comprendere una moltitudine di implementazioni ed algoritmi tale da richiedere un libro ad hoc per essere trattate tutte. In questa sezione ci occuperemo quindi delle implementazioni più comuni. Gli aspetti legati al supporto alla programmazione concorrente saranno affrontati, in maniera specifica, nel capitolo dedicato al multithreading in Java.

Strutture dati più comuni

In letteratura esistono diversi tipi di strutture dati, ognuna delle quali ha un significato preciso in termini di elementi memorizzati e modalità di accesso ai dati. Tuttavia, ne esiste un gruppo che rappresenta un insieme significativo per quello che riguarda la programmazione e che consente di affrontare la maggior parte dei problemi di programmazione.

Questo insieme, che rappresenta anche la base per la creazione di strutture più complesse, è descritto a seguire:

1. set (insieme)

Insieme non ordinato di dati. E' caratterizzato dal fatto che non può contenere duplicati.

2. lista ordinata

E' una sequenza di dati organizzata secondo un generico ordine (ad esempio di grandezza, di inserimento, di importanza...). E' realizzata concatenando un dato con il successivo.

3. pila o stack

E' una lista ordinata secondo l'ordine di inserimento, in cui le operazioni di inserimento ed estrazione si basano sul metodo *LIFO*: *Last In FIrst Out*. Si tratta di una struttura dati molto utilizzata nei sistemi a microprocessore sulla quale si basano, ad esempio, le operazioni di chiamata di funzioni e la ricorsione (da qui il nome di stack-space o stack-trace in Java).

4. coda o queue.

E' una lista ordinata secondo l'ordine di inserimento, in cui le operazioni di inserimento ed estrazione si basano sul metodo *FIFO: First In FIrst Out*. Sono molto utilizzate in informatica in tutti i casi in cui sia necessario collegare due o più entità che lavorano a velocità diverse (ad esempio si pensi alla tastiera: le digitazioni dell'utente vengono memorizzate in una coda mentre il computer e' occupato; lo stesso accade per la coda di stampa e per l'invio e la ricezione di dati da internet);

5. coda doppia o deque

E' una lista ordinata secondo l'ordine di arrivo in cui i dati sono inseriti ed estratti da entrambe le estremità da cui il nome: *deque (pronunciato come deck)* contrazione di *double ended queue*;

6. mappa o dizionario

E' una collezione in cui i dati sono individuati da una *chiave*; gli elementi della collezione sono quindi delle coppie <*chiave,valore*>. Sono molto utilizzati nei database e in tutte quelle applicazioni dove si vuole un accesso veloce alle informazioni;

Prima della versione 1.2 di Java esistevano solo tre tipi per raggruppare oggetti in Java: una classe *Array*, la classe *Vector* e la classe *Hashtable*. Utilizzavano tutte la classe Object per generalizzare, non offrivano nessun metodo ottimizzato per gestire i dati secondo le modalità richieste per implementare le varie strutture dati. A partire dalla versione 1.2 furono arricchite aggiungendo strutture dati mancanti ed offrire un insieme più completo di possibili scelte. A partire da Java 5 furono modificate diventando tipi generici.

Il framework

Java Collections Framework - JFC è una libreria formata da interfacce e classi generiche nate per gestire gruppi di oggetti (collezioni); in continua evoluzione mette a disposizione del programmatore:

1. Interfacce

Tipi di dato astratto che rappresentano le collezioni. Permettono di manipolare i dati senza entrare nei dettagli della singola implementazione, così come è possibile fare con una qualsiasi interfaccia presente nel mondo Java;

2. Implementazioni

Classi di uso comune che implementano le interfacce suddette e che evolvono negli anni per assecondare le necessità di una programmazione sempre più evoluta (negli ultimi anni si sono aggiunte classe sincronizzate a supporto della programmazione concorrente);

3. Algoritmi

Funzioni ottimizzate per compiere in maniera efficiente le operazioni più comuni sulle Collezioni, quali ad esempio operazioni di ordinamento e di ricerca. Grazie al polimorfismo, tali algoritmi sono a loro volta specializzati a seconda del tipo di collezione che andremo a gestire.

Nelle prossime due immagini sono rappresentate le gerarchie delle interfacce e delle classi più comuni che compongono il *framework*.

La caratteristica più importante però è stata la generalizzazione di questi concetti che ha portato a proporre, a partire da java 5, un insieme di classi ed interfacce generiche che rendono ancora più flessibile lo strumento.

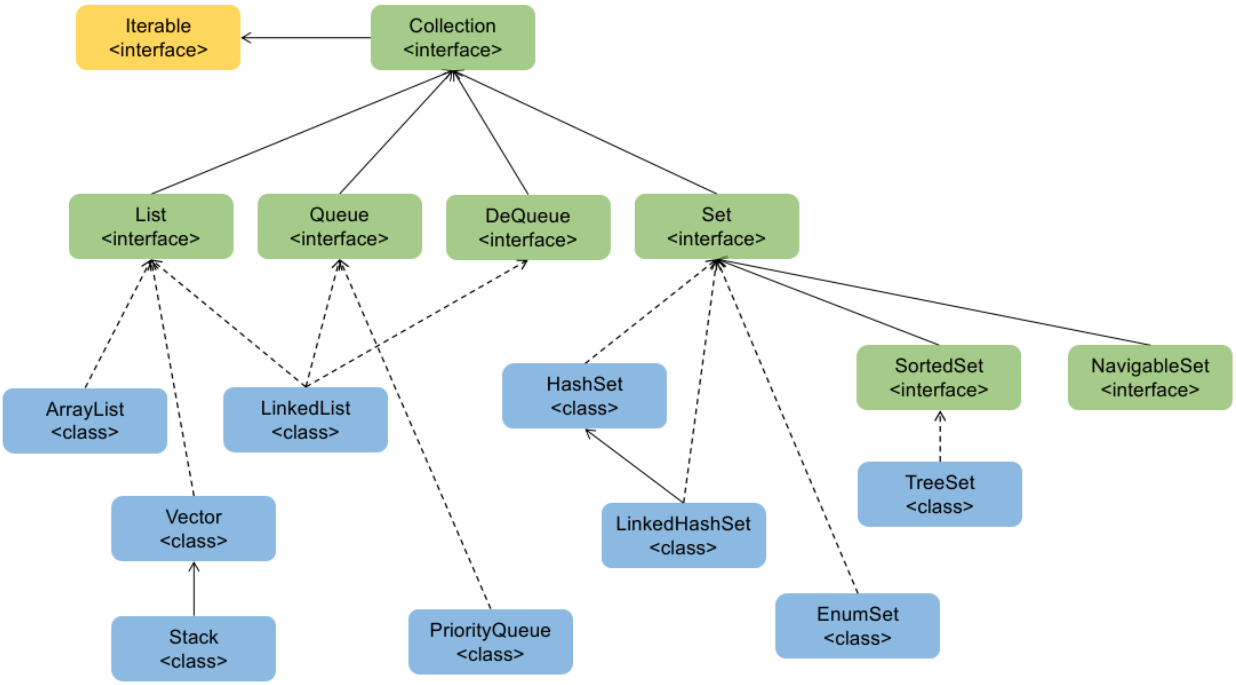


Immagine 43 Java Collection Framework - Gerarchia Collection

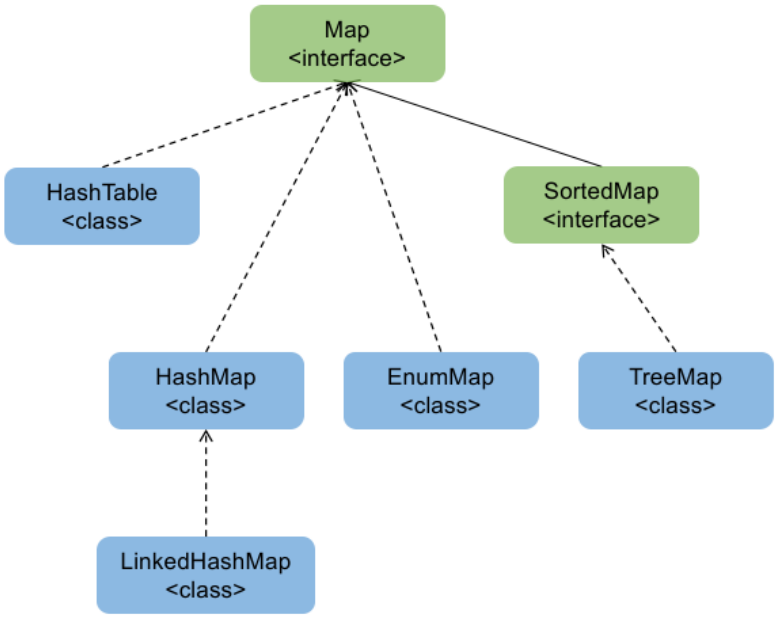


Immagine 44 java CollectionFramework - Gerarchia Map

Il *Java Collection Framework* contiene decine di definizioni di classe, tuttavia non è scopo di questo libro esaminarle tutte (per cui consiglio di fare riferimento alla documentazione ufficiale), piuttosto prenderemo in considerazione quelle più comunemente utilizzate: in questo capitolo ci occuperemo delle collezioni che implementano le interfacce *Map* e *Collection*; nei prossimi capitoli, parlando di programmazione concorrente, parleremo delle collezioni appartenenti al package *java.util.concurrent*.

Iniziamo con le due interfacce principali: l'interfaccia *Collection* a sua volta derivata da *Iterable* e l'interfaccia *Map*. Ciascuna delle interfacce è implementata da almeno una classe predefinita che è realizzata utilizzando le strutture dati più efficienti e gli algoritmi ottimizzati per il determinato tipo di collezione che la classe dovrà rappresentare.

Le classi appartenenti alle Java Collection API tipicamente realizzano tutti e soli i metodi richiesti dall'interfaccia che implementano; sono dotate di costruttori senza argomenti e ridefiniscono opportunamente i metodi *equals()*, *hashcode()* e *clone()*.

Il framework definisce due classi di oggetti: le *liste iterabili o collezioni* e le *mappe*.

Le *collezioni* rappresentano gruppi di elementi e possono essere ordinate (liste) oppure sparse (insiemi). Esistono diversi tipi di implementazioni a rappresentare gruppi di dati con diversi comportamenti:

1. modificabili oppure non modificabili

2. con ripetizioni oppure senza ripetizioni (come gli insiemi)

3. struttura lineare oppure ad albero

4. elementi ordinati oppure non ordinati

Le *mappe* sono collezioni di elementi rappresentati da coppie *<chiave, valore>* e consentono di accedere in maniera veloce ad ogni valore della collezione attraverso la sua chiave.

Accedere agli elementi di una collezione

Ogni tipo appartenente al JFC implementa metodi specifici per accedere ai dati presenti nella collezione ovviamente dipendenti dalla natura della collezione stessa. Tuttavia esistono dei metodi standard che consentono scorrere in maniera veloce gli elementi di una collezione attraverso gli iteratori.

DEFINIZIONE: un iteratore è un tipo che consente di utilizzare gli elementi di una collezione all'interno di un ciclo.

In Java esistono due diversi tipi di iteratori: *fail-fast e fail-safe.*

DEFINIZIONE: Gli iteratori fail-fast si arrestano immediatamente dopo la segnalazione di un errore. Tutte le operazioni vengono interrotte all'istante.

Sono utilizzati in programmazione concorrente e ritornano immediatamente una eccezione di tipo *ConcurrentModificationException* non appena viene modificato un elemento della collezione da parte di un processo concorrente.

DEFINIZIONE: Gli iteratori fail-safe non si arrestano in caso di errore privilegiando l'esecuzione alla gestione del problema.

Al contrario, non generano eccezioni se una collezione viene modificata strutturalmente durante l'iterazione. Questo perché gli iteratori di questo tipo operano su un clone della raccolta originale.

Il tipo di iteratore, *fail-safe* oppure *fail-fast* in Java dipende dal tipo di collezione e non dalla specifica implementazione. Tipicamente, gli iteratori *fail-fast* sono utilizzati per tutte le collezioni che richiedono particolare attenzione alla *thread-safety* (programmazione concorrente).

Esistono alcune differenze sostanziali tra i due tipi che vale evidenziare, e che sono discusse nella prossima tabella:

Fail-Save vs Fail-Fast		
ambito	**fail-fast**	**fail-safe**
eccezioni	Generano una eccezione di tipo *ConcurrentModificationException.*	Non generano nessuna eccezione.
memoria	Richiede poca memoria durante l'iterazione sugli elementi della collezione.	Richiede più memoria durante l'iterazione sugli elementi della collezione.
modifica	Non consente modifiche durante l'iterazione.	Consente modifiche durante l'iterazione.
performance	E' generalmente molto veloce.	E' significativamente più lenta dell'altro caso.
applicabilità	HashMap, ArrayList, Vector, HashSet, etc.	CopyOnWriteArrayList, ConcurrentHashMap ed in generale tutte le collezioni definite in java.util.concurrent

Interfaccia Enumeration

Il Java Collection Framework consente di utilizzare tipi enumerabili attraverso l'interfaccia generica *java.util.Enumeration*. Enumeration contiene la definizione delle funzioni per poter iterare attraverso gli elementi di una collezione. La definizione è la seguente:

```
public interface Enumeration<E>{
  default Iterator<E> asIterator() ...
  E nextElement();
  boolean hasMoreElements();
}
```

Nonostante sia un interfaccia considerata vecchia ed obsoleta, alcuni tipi appartenenti al JFC come la classe Vector, Stack e HashTable la utilizzano ancora.

Tra le principali caratteristiche dei tipi Enumeration:

1. Non supporta aggiunta, cancellazione o modifica degli elementi diuna collezione;

2. Consente di iterare sugli elementi di una collezione solo in una direzione;

Definisce 2 metodi astratti ed un metodi di default:

Metodi di Enumeration
default Iterator<E> asIterator()
Restituisce un *Iterator* che attraversa gli elementi rimanenti coperti da questa enumerazione.
E nextElement()
Ritorna il prossimo elemento nella collezione aggiornando l'indice di iterazione della collezione;
boolean hasMoreElements()
Ritorna *true* se e solo se nella collezione c'è ancora almeno un elemento da scandire;

Il pattern Iterator e l'interfaccia Iterable

Il pattern *Iterator* ha come obiettivo quello di definire un metodo standard per scorrere gli elementi di una collezione. Java fornisce una implementazione di questo pattern tramite l'interfaccia generica *java.util.Iterator* la cui definizione è la seguente:

```
public interface Iterator<E> {
   default void forEachRemaining(Consumer<? super E> action){
    .....
    }
   boolean hasNext();
   E next();
   void remove();   // Optional
}
```

L'interfaccia *Itarator* ha quattro metodi:

Metodi di Iterator
default void forEachRemaining (Consumer<? super E> action)
Esegue l'azione specificata per ogni elemento rimanente finché tutti gli elementi non sono stati elaborati o l'azione genera un'eccezione.
E next()
Ritorna il prossimo elemento nella collezione aggiornando l'indice di iterazione della collezione;

boolean hasNext()

Ritorna true se e solo se nella collezione c'è ancora almeno un elemento da scandire;

remove()

Rimuove dalla collezione l'ultimo elemento ritornato. Questo metodo può essere chiamato solo successivamente ad una chiamata al metodo next()

Come il pattern Singleton, anche il pattern *Iterator* è un pattern *creazionale,* e fa parte dei patterns proposti dalla *Gang Of Four* negli anni 90.

Il metodo *remove* non sempre è implementato concretamente. In generale è sempre consigliabile fare riferimento alla documentazione della classe che si sta utilizzando.

Il metodo forEachRemaining sarà più chiaro successivamente quando parleremo di interfacce funzionali ed espressioni Lambda.

Il prossimo frammento di pseudo-codice mostra come utilizzare il tipo *Iterator* per stampare tutti gli elementi di una collezione:

```
static void stampaElementi(Collection<?> collection) {
  Iterator<?> iterator = collection.iterator();
  while ( iterator.hasNext() )
    System.out.println(iterator.next());
}
```

Il pattern *Iterator* consente la scansione di una Collezione solo in una direzione; non è quindi possibile muoversi a ritroso mentre si accede agli elementi.

L'interfaccia *java.lang.Iterable*, a sua volta è estesa dall'interfaccia *Collection*, che contiene la definizione di un solo metodo che ritorna un tipo Iterator:

```
public interface Iterable<T> {
   default void forEachRemaining(Consumer<? super E> action){
   .....
   }
   default Spliterator<T> spliterator();
  Iterator iterator(); // crea un nuovo iteratore


}
```

Scandire una lista con ListIterator

Un tipo *ListIterator*, come evidente dalla definizione dell'interfaccia, oltre ad includere le funzionalità di Iterator di cui è sottoclasse, supporta funzioni aggiuntive che consentono di modificare l'elemento corrente nella lista oppure aggiungere un elemento nella posizione precedente a quella corrente:

```
public interface ListIterator<E> extends Iterator<E> {
// boolean hasNext();
// E next();
boolean hasPrevious();
E previous();
int nextIndex();
int previousIndex();
// void remove(); // Optional
void set(E e); // Optional
void add(E e); // Optional
}
```

In dettaglio:

Metodi di ListIterator
previous()
Restituisce l'elemento precedente della lista, e contemporaneamente sposta il cursore all'indietro.
hasPrevious()
Verifica se il cursore ha ancora un predecessore o si `e raggiunto l'inizio della lista.
nextIndex(), previousIndex()
Restituiscono l'indice dell'elemento che sarebbe restituito da next() e previous() rispettivamente (ma

non spostano il cursore)

All'inizio della lista (quando hasPrevious()==*false*), previousIndex() restituisce -1, mente alla fine della lista (quando hasNext()==*false*), nextIndex() retituisce list.size() (gli elementi sono indicizzati come al solito a partire da 0 fino a list.size()-1).

set(o)

Modifica l'elemento nella posizione corrente (ovvero la posizione che sarebbe restituita da una chiamata a next()).

add(o)

Aggiunge un elemento alla lista nella posizione precedente a quella corrente (ovvero la posizione che sarebbe restituita da una chiamata a previous()).

Interfaccia Collection

L'interfaccia *Collection* definisce l'aspetto di una generica collezione i cui metodi permettono di svolgere operazioni quali:

1. Operazioni di base quali inserimento, cancellazione, ricerca di un elemento nella collezione;

2. Operazioni che lavorano su intere collezioni quali l'inserimento, la cancellazione la ricerca di collezioni di elementi;

3. Metodi che ci consentono di ottenere un oggetto Iterable;

3. Operazioni per trasformare il contenuto della collezione in un array.

4. Operazioni opzionali che potrebbero non essere implementate, e che in quel caso, propagano un eccezione di tipo UnsupportedOperationException.

Di seguito la definizione dell' l'interfaccia Collection:

```
public interface Collection extends Iterable {

  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(Object element);   // Optional
  boolean remove(Object element); // Optional
  Iterator iterator();

  boolean containsAll(Collection c);
  boolean addAll(Collection c);   // Optional
  boolean removeAll(Collection c); // Optional
  boolean retainAll(Collection c); // Optional
  void clear();              // Optional
```

```
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

A partire da questa sono definite altre quattro interfacce: *List*, *Set*, *Queue*, *DeQueue* che rappresentano rispettivamente liste ordinate, insiemi, code e code doppie.

Interfaccia List e le sue implementazioni

L'interfaccia *List*, definita a partire da *Collection* e riportata a seguire, è utilizzata per rappresentare lista ordinate che possono contenere dati duplicati.

Oltre ai m

Il criterio di ordinamento dei dati in una lista è normalmente di tipo posizionale ovvero basato sulla posizione dell'elemento nella lista. Tuttavia, esistono strumenti che consentono di modificare l'ordine sulla base di criteri specifici legati al problema che dobbiamo rappresentare.

per:

1. *Accedere agli elementi della lista in base alla posizione;*

2. *Trovare la posizione di un oggetto;*

3. *Estrarre di sotto-sequenza della collezione;*

4. *Scandire gli elementi in maniera bidirezionale mediante ListIterator: oltre alla possibilità di scandire la lista con Iterator, List fornisce una forma più avanzata di scansione che consente di muoversi in tutte e due le direzioni.*

```
public interface List<E> extends Collection<E> {
        // Positional access
        E get(int index);
        E set(int index, E element); //optional
        boolean add(E element); //optional
        void add(int index, E element); //optional
        E remove(int index); //optional
        boolean addAll(int index,
        Collection<? extends E> c); //optional

        // Search
        int indexOf(Object o);
        int lastIndexOf(Object o);

        // Iteration
        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);

        // Range-view
        List<E> subList(int from, int to);
}
```

Oltre alla definizione dei metodi astratti, l'interfaccia *List* mette a disposizione un serie di metodi di default alcuni dei quali tratteremo successivamente nel libro.

ArrayList e LinkedList

Due delle classi che forniscono una implementazione di questa interfaccia sono:

1. ArrayList

Rappresenta una lista ordinata secondo l'ordine di inserimento: utilizza un array dinamico in grado di aumentare o ridurre la propria capacità. Ha complessità O(1) per la ricerca e O(n) per inserimento e cancellazione. Ritorna un Iterator di tipo *fail-fast*.

2. LinkedList

Rappresenta una coda doppia o dequeue. Utilizza una lista doppia con riferimenti al successore ed al predecessore. Ha una complessità O(n) nel caso di ricerca, inserimento e cancellazione, O(1) per inserimento o cancellazione in testa/coda e scansione della collezione pertanto fornisce prestazioni migliori rispetto all' ArrayList se gli elementi vengono spesso inseriti o eliminati all'interno dell'elenco. Ritorna un Iterator di tipo *fail-fast*.

La classe *Vector* offre un ulteriore implementazione dell'interfaccia *List*. Come *ArrayList*, è un array dinamico in grado di modificare la propria dimensione. E' una classe *thread-safe* pertanto non è consigliabile utilizzarla se non in quelle applicazioni che richiedono di gestire collezioni di oggetti in concorrenza. Ritorna un *Iterator* di tipo *fail-fast*;

Non sarà comunque trattata in questa sezione.

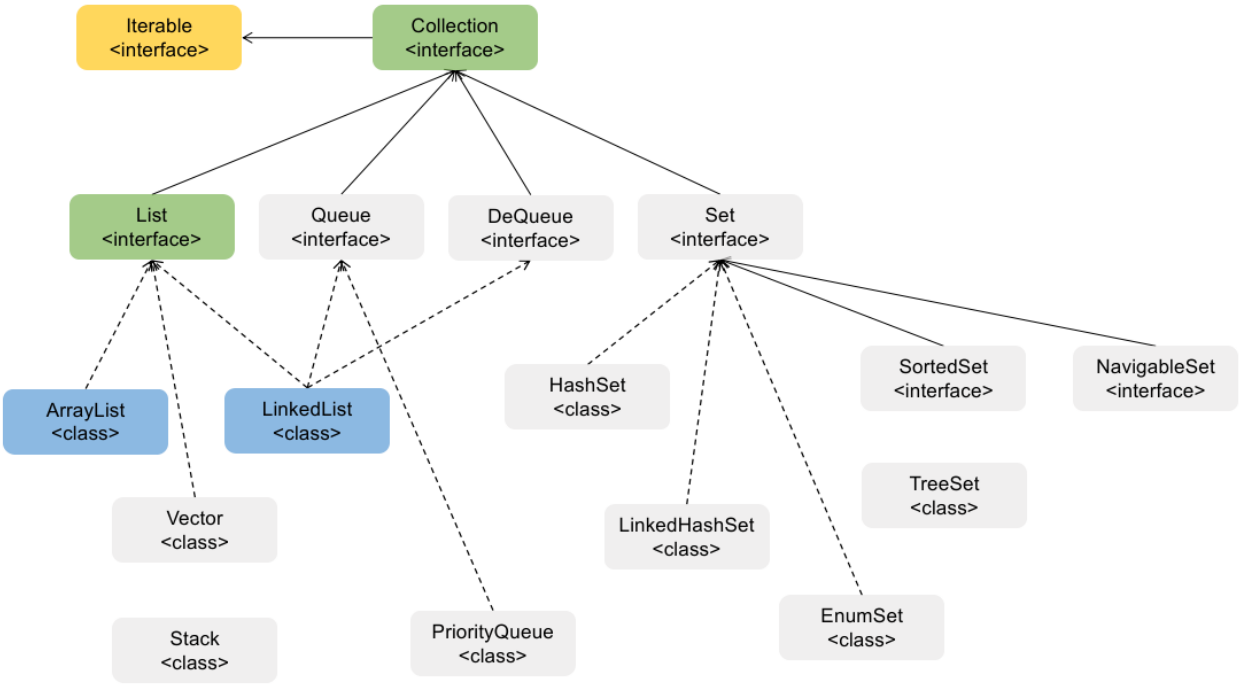


Immagine 45 - ArrayList e LinkedList

Nell'immagine sono messe in evidenza le due implementazioni e la loro gerarchia.

A seguire un esempio di utilizzo di liste. Poiché sono tipi generici, è necessario specificare il tipo al momento della creazione dell'oggetto:

```
public class ListDemo {

  public static void main(String[] args) {
    List<String> arrayList = new ArrayList<String>();/
    arrayList.add("elemento-1");
    arrayList.add("elemento-2");
    List<String> linkedList = new LinkedList<String>();
    linkedList.add("elemento-1");
    linkedList.add("elemento-2");
  }
}
```

Tutte le liste, implementazione dell'interfaccia *List*, oltre all'iteratore standard mettono a disposizione un tipo *ListIterator* per lo scorrimento bidirezionale delle liste. La prossima applicazione utilizza *ListIterator* per scandire una lista nelle due direzioni:

```
public class ListIteratorDemo {
  public static void main(String[] args) {
    List<String> arrayList = new ArrayList<String>();
    arrayList.add("elemento-1");
    arrayList.add("elemento-2");
    ListIterator<String> forwardListIterator = arrayList.listIterator();
    while (forwardListIterator.hasNext()) {
      System.out.println(forwardListIterator.next());
    }
    System.out.println();
    ListIterator<String> backWardListIterator = arrayList.listIterator(arrayList.size());
    while (backWardListIterator.hasPrevious()) {
      System.out.println(backWardListIterator.previous());
    }
  }
}
```

La prossima tabella mette a confronto le due implementazioni:

ArrayList	LinkedList
Utilizza un array dinamico, di conseguenza la manipolazione di elementi è inefficiente in quanto richiede lo spostamento di porzioni della memoria quando vengono aggiunti o rimossi elementi;	Utilizza una lista di elementi con riferimenti al predecessore/successore. Risulta più performante della *ArrayList* dal momento che non richiede spostamenti di porzioni di memoria qualora vengano aggiunti o rimossi elementi.
Rappresenta solo una lista in quanto implementa solo l'interfaccia *List*.	Implementa entrambe le interfacce *List* e *Deque*. Può comportarsi come una lista o come una coda.
ArrayList è la scelta migliore quando si tratta di memorizzare ed accedere in modo sparso a grosse quantità di dati.	*LinkedList* rappresenta la scelta migliore quando si tratta di manipolare dati.
Dal momento che utilizza un array dinamico, quando viene inizializzata ha una capienza iniziale di 10 elementi.	Al momento della creazione è una lista vuota dal momento che non necessità di array per essere implementata.

Interfaccia Set e le sue implementazioni

L'interfaccia *Set*, che estende *Collection*, rappresenta un insieme non ordinato di dati e come tale, non consente dati duplicati. Nella prossima immagine sono identificate le implementazioni dell'interfaccia *Set* prese in considerazione in questa sezione.

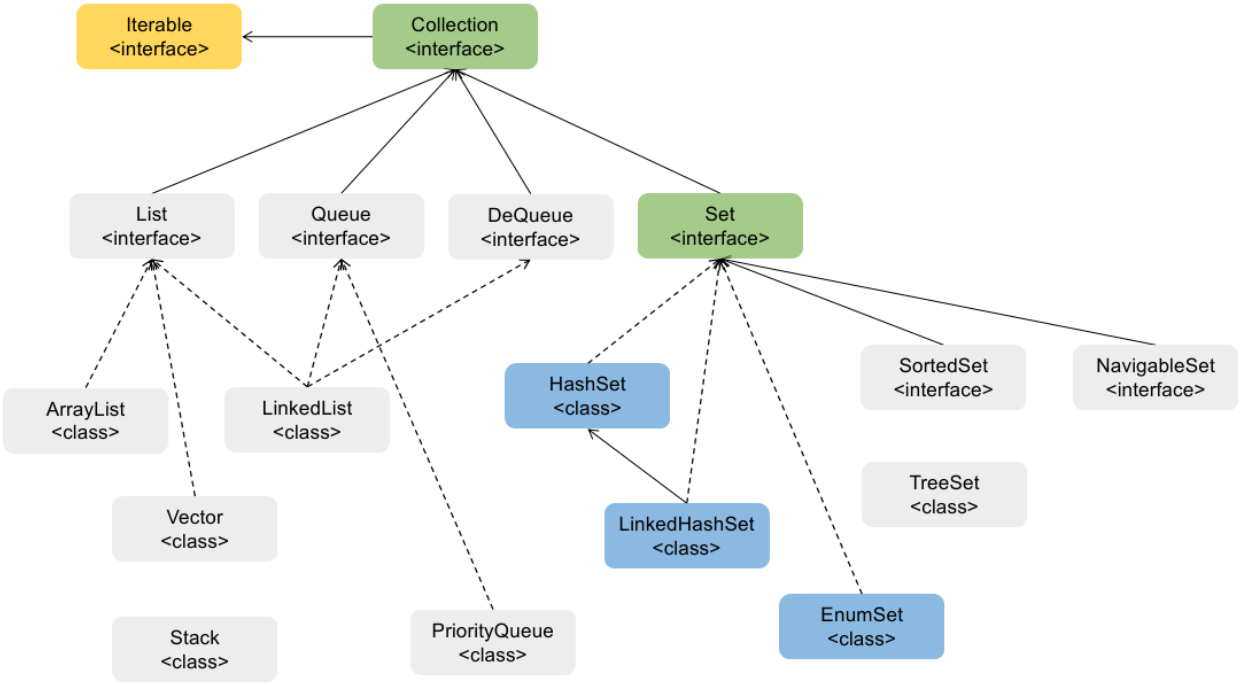


Immagine 46 - Interfaccia set e sue implementazioni

Set non aggiunge nessun nuovo metodo alla superclasse ed è definita come segue:

```
public interface Set<E> extends Collection<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element); // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);// Optional
  boolean removeAll(Collection<?> c); // Optional
  boolean retainAll(Collection<?> c); // Optional

  void clear(); // Optional
  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

Oltre alla definizione dei metodi astratti, l'interfaccia *Set* mette a disposizione un serie di metodi di default alcuni dei quali tratteremo successivamente nel libro.

Ciò che caratterizza i tipi Set in Java sono le seguenti proprietà:

1. A differenza di List, Set NON consente di aggiungere elementi duplicati.

2. Set consente di aggiungere al massimo un solo elemento nullo.

3. *Set implementa tutte le più comuni funzioni segli insiemi*

Di fatto, le funzioni bulk rappresentano le seguenti operazioni sugli insiemi:

$$set1.containsAll(set2) \Rightarrow set1 \subseteq set2$$
$$set1.addAll(set2) \Rightarrow set1 \cup set2$$
$$set1.removeAll(set2) \Rightarrow set1 \backslash set2$$
$$set1.retainAll(set2) \Rightarrow set1 \cap set2$$

A seguire alcune delle più comuni implementazioni dell'interfaccia Set.

HashSet e LinkedHashSet

HashSet utilizza una tabella hash memorizzare la raccolta di elementi senza tener conto dell'ordine di inserimento. *LinkedHashSet* è simile ad *HashSet*, ma al contrario della prima mantiene l'ordine di inserimento. Entrambe non consentono duplicati.

LinkedHashSet eredita da *HashSet*, entrambe implementano l'interfaccia Set.

1. HashSet

Come anticipato, utilizza il meccanismo di *hashing* per memorizzare gli elementi all'interno di una hash-table, di conseguenza non consente oggetti duplicati (ricordiamo che due oggetti uguali devono avere anche lo stesso codice hash), e non fornisce nessun meccanismo per mantenere l'ordine di inserimento ordinando invece in base al valore dei codici hash.

Consente di memorizzare anche valori *null*.

Non è thread-safe, pertanto non può essere utilizzata in una applicazione multi-threaded senza implementare le dovute strategie per la sincronizzazione degli accessi.

2. LinkedHashSet

Simile alla classe precedente, da cui di fatto eredita, utilizza il meccanismo di *hashing* per memorizzare gli elementi. Differisce da *HashSet* in quanto è in grado di mantenere l'ordine di inserimento, per farlo utilizza entrambe una hash-table ed una *LinkedList.*

Non è thread-safe, pertanto non può essere utilizzata in una applicazione multi-threaded senza implementare le dovute strategie per la sincronizzazione degli accessi.

Nei prossimi due esempi, le due implementazioni a confronto. Nel primo esempio utilizzeremo un tipo *HashSet* mentre nel secondo il tipo *LinkedHashSet* : l'esecuzione delle due applicazioni darà risultati diversi dovuti al diverso meccanismo di memorizzazione dei dati all'interni dei due insiemi.

La prima applicazione è la seguente:

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetDemo {
  public static void main(String[] args) {
```

```
    HashSet<String> hashSet = new HashSet<>();
    hashSet.add("Mela");
    hashSet.add("Pera");
    hashSet.add("Banana");
    hashSet.add("Pesca");
    Iterator<String> iterator = hashSet.iterator();
    while(iterator.hasNext()){
      System.out.println(iterator.next());
    }

  }
}
```

L'esecuzione di *HashSetDemo* ha come risultato:

```
Pera
Mela
Pesca
Banana
```

Come aspettato, non tiene conto dell'ordine di inserimento. Se invece utilizziamo *LinkedHashSet* le cose cambiano:

```
import java.util.LinkedHashSet;
import java.util.Iterator;
public class LinkedHashSetDemo {
  public static void main(String[] args) {
    LinkedHashSet<String> hashSet = new LinkedHashSet<>();
    hashSet.add("Mela");
    hashSet.add("Pera");
    hashSet.add("Banana");
    hashSet.add("Pesca");
    Iterator<String> iterator = hashSet.iterator();
    while(iterator.hasNext()){
      System.out.println(iterator.next());
    }

  }
}
```

Come atteso, in questo caso, il risultato terrà conto dell'ordine di inserimento degli elementi:

```
Mela
Pera
Banana
Pesca
```

Nella prossima tabella, i due tipi a confronto:

HashSet	LinkedHashSet
Memorizza i dati mediante una HashTable.	Memorizza i dati mediante una HashTable ed una LinkList per mantenere l'ordine.
Non tiene traccia dell'ordine di inserimento, di conseguenza non possiamo predire un ordine per gli elementi.	Tiene traccia dell'ordine di inserimento, di conseguenza possiamo predire un ordine per gli elementi.
Richiede meno memoria di *LinkedHashSet.*	Richiede più memoria a causa delle strutture dati utilizzate per mantenere l'ordine di inserimento.
Non è thread-safe.	Non è thread-safe.
E' sensibilmente più performante di *LinkedHashSet.*	E' sensibilmente meno performante di *HashSet.*

In definitiva quindi, se mantenere l'ordine di inserimento è una priorità, allora sarà il caso di utilizzare *LinkedHashSet*, viceversa sarà sempre preferibile utilizzare *HashSet*.

EnumSet

EnumSet è una classe specializzata per implementare insiemi a partire da costanti enumerabili di tipo *Enum*. Oltre ad utilizzare tecniche di gestione della collezione ottimizzata per l'utilizzo del tipo memorizzato, la specializzazione consiste in una serie id metodi statici aggiuntivi per la creazione (*factory*) della collezione a partire da un tipo *Enum*.

Un *factory method* è un metodo responsabile della creazione di un oggetto a partire da un oggetto generico.

Nella prossima tabella sono elencati i metodi statici specializzati per questa classe:

Metodi factory di EnumSet
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)
Specializzato nella creazione di un EnumSet contenente tutti gli elementi definiti nel tipo passato per parametro.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
Specializzato nella creazione di un EnumSet a partire da una collezione.

static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)
Specializzato nella creazione di un EnumSet vuoto del tipo specificato da elementType.
static <E extends Enum<E>> EnumSet<E> of(E e)
Specializzato nella creazione di un EnumSet contenente l'elemento specificato.
static <E extends Enum<E>> EnumSet<E> of(E ... arg1)
Specializzato nella creazione di un EnumSet contenente solo gli elementi specificati.
static <E extends Enum<E>> EnumSet<E> range(E from, E to)
Specializzato nella creazione di un EnumSet contenente il range di elementi specificati.

L'utilizzo dei metodi factory è mostrato nel prossimo frammento di codice:

```
import java.util.EnumSet;
import java.util.Set;

enum MesiDellAnno {
   GENNAIO, FEBBRAIO,
   MARZO, APRILE,
   MAGGIO, GIUGNO,
   LUGLIO, AGOSTO,
   SETTEMBRE, OTTOBRE,
   NOVEMBRE, DICEMBRE
}

public class EnumSetDemo {

   public static void main(String[] args) {
      Set<MesiDellAnno> mesi1 = EnumSet.allOf(MesiDellAnno.class);
      Set<MesiDellAnno> mesi2 = EnumSet.noneOf(MesiDellAnno.class);
      Set<MesiDellAnno> mesi3 = EnumSet.range(MesiDellAnno.GENNAIO, MesiDellAnno.APRILE);
      Set<MesiDellAnno> mesi4 = EnumSet.of(MesiDellAnno.GENNAIO);
      Set<MesiDellAnno> mesi5 = EnumSet.of(MesiDellAnno.GENNAIO,
                                           MesiDellAnno.FEBBRAIO, MesiDellAnno.MARZO);
      }

}
```

Interfaccia Map e le sue implementazioni

A differenza delle *Collection*, le mappe rappresentano gruppi di elementi memorizzati in forma di <*chiave, valore*>; ogni coppia <*chiave, valore*> è anche definita *entry* che in italiano può essere tradotta come *voce*.

Come per le collezioni, l'interfaccia Map definisce

1. Operazioni di base quali inserimento, cancellazione, ricerca;

2. Metodi che ci consentono di ottenere oggetti Iterable;

3. Metodi che ci consentono di ottenere gli elenchi delle chiavi e dei valori come collezioni;

4. La sotto-interfaccia Entry<K,V> che rappresenta una generica entry (voce) della mappa.

La definizione dell'interfaccia *Map* è la seguente:

```
public interface Map<K,V> {

  public int size();
   public boolean isEmpty();
   public boolean containsKey(Object arg0);
   public boolean containsValue(Object arg0);
   public V get(Object arg0);
   public V put(K arg0, V arg1);
   public V remove(jObject arg0);
   public void putAll(Map<? extends K,? extends V> arg0);
   public void clear();
   public java.util.Set<K> keySet();
   public java.util.Collection<V> values();
   public java.util.Set<Map.Entry<K,V>> entrySet();
   public boolean equals(Object arg0);
   public int hashCode();

  //METODI DI DEFAULT
}
```

Oltre alla definizione dei metodi astratti, l'interfaccia *Map* mette a disposizione un serie di metodi di default alcuni dei quali tratteremo successivamente nel libro.

A differenza dei tipi che implementano *Collection*, le mappe non possono essere traversate ovvero non è possibile iterare sugli elementi della mappa attraverso un tipo *Iterator*, e questo proprio perché le voci di una Mappa sono coppie <*chiave, valore*>. Tuttavia, l'interfaccia *Map* mette a disposizione i due metodi:

```
public java.util.Set<K> keySet();
public java.util.Collection<V> values();
```

che, rispettivamente, ritornano l'insieme delle chiavi e l'elenco dei valori che possiamo utilizzare quando dobbiamo iterare sulle voci.

Le mappe consentono di memorizzare valori uguali, ma non consentono di duplicare le chiavi; questo il motivo percui il metodo *keySet()* restituisce un tipo *Set* invece di un topo *Collection*. Di conseguenza, la cardinalità dell'insieme delle chiavi potrebbe essere minore della dimensione dell'elenco dei valori.

Nelle prossime sotto-sezioni andremo ad analizzare tre diverse implementazioni dell'interfaccia *Map* che, come vedremo, hanno alcune cose in comune con i tipi già discussi per liste ed insiemi.

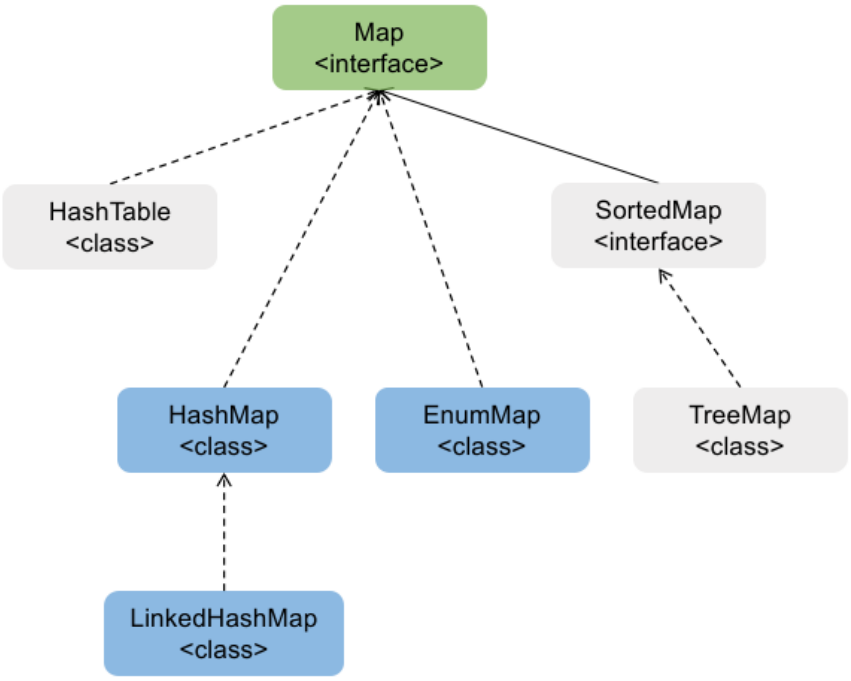


Immagine 47 Implementazioni dell'interfaccia Map

HashMap e LinkedHashMap

Come per le altre Hash... e LinkedHash..., *HashMap* e *LinkedHashMap* utilizzano entrambe il meccanismo di hashing per memorizzare le voci con la differenza che la prima non tiene conto dell'ordine di inserimento mentre la seconda si. Con qualche piccola differenza rispetto alle collezioni.

Grazie al meccanismo di *hashing*, possiamo accedere in maniera molto efficiente ai valori O(1) una volta nota la chiave.

1. HashMap

Utilizza il meccanismo di *hashing* sulle *chiavi* per memorizzare le voci, pertanto due voci con chiavi uguali non possono coesistere. Non è una classe thread-safe, e come aspettato non tiene traccia dell'ordine di inserimento.

Ammette una sola chiave *null*, ma possono coesistere diverse voci con valore nullo.

2. LinkedHashMap

E' sostanzialmente simile alla precedente con la differenza che tiene conto dell'ordine di inserimento, e pertanto utilizzerà ulteriori strutture dati per tenerne traccia.

Anche se può sembrare ripetitiva, ecco la tabella di confronto delle due classi:

HashMap	LinkedHashMap
Memorizza i dati mediante una HashTable.	Memorizza i dati mediante una HashTable ed una LinkList per mantenere l'ordine.
Non tiene traccia dell'ordine di inserimento, di conseguenza non possiamo predire un ordine per gli elementi.	Tiene traccia dell'ordine di inserimento, di conseguenza possiamo predire un ordine per gli elementi.
Richiede meno memoria di *LinkedHashMap.*	Richiede più memoria a causa delle strutture dati utilizzate per mantenere l'ordine di inserimento.
Non è thread-safe.	Non è thread-safe.
E' sensibilmente più performante di *LinkedHashMap.*	E' sensibilmente meno performante di *HashMap*.

Ed ora qualche esempio di utilizzo delle mappe.

```
public class HashMapDemo {

  public static void main(String[] args) {
    Map<Integer, String> mappa = new HashMap<>();
    mappa.put(1, "Mela");
    mappa.put(3, "Pera");
    mappa.put(2, "Banana");
```

```
    mappa.put(4, "Pesca");
    //sovrascrive il valore associato alla stessa chiave: pesca
    mappa.put(4, "Ananas");

    for(Integer chiave : mappa.keySet()){
      System.out.println("<"+chiave+","+mappa.get(chiave)+">");
    }

  }
}
```

Poiché due chiavi uguali non possono esistere, l'inserimento di *<4,Ananas>* sovrascriverà *<4,Pesca>*. Se infatti eseguiamo il metodo *main* il risultato sarà il seguente:

```
<1,Mela>
<2,Banana>
<3,Pera>
<4,Ananas>
```

Notiamo anche che, come aspettato l'output non tiene conto dell'ordine di inserimento, ma sopratutto le voci son ordinate sulla base del tipo/valore della *chiave*.

Se utilizziamo invece una *LinkedHashMap* il risultato dovrebbe essere ormai prevedibile:

```
public class LinkedHashMapDemo {
  public static void main(String[] args) {
    Map<Integer, String> mappa = new LinkedHashMap<>();
    mappa.put(1, "Mela");
    mappa.put(3, "Pera");
    mappa.put(2, "Banana");
    mappa.put(4, "Pesca");
    //sovrascrive il valore associato alla stessa chiave: pesca
    mappa.put(4, "Ananas");

    for(Integer chiave : mappa.keySet()){
      System.out.println("<"+chiave+","+mappa.get(chiave)+">");
    }
  }
}
```

Anche in questo caso perderemo una voce perché sovra scritta, tuttavia la mappa adesso ricorda l'ordine di inserimento:

```
<1,Mela>
<3,Pera>
<2,Banana>
<4,Ananas>
```

EnumMap

E' la classe specializzata nella gestione delle mappe in cui la chiave di ogni voce è rappresentata da un tipo enumerabile.

Hanno le seguenti caratteristiche:

1. E' una collezione ordinata basata sull'ordine naturali delle chiavi;

2. E' un tipo estremamente performante, molto più delle HashMap;

3. Le chiavi di una EnumMap devono essere tutte dello stesso tipo derivato da Enum;

4. Non consente l'utilizzo di chiavi null generando, eventualmente, NullPointerException;

5. Gli iteratori sono tutti di tipo fail-safe;

6. Utilizzano array per rappresentare le strutture dati, e per questo sono estremamenti efficenti e compatte.

I costruttori sono i seguenti:

Metodi factory di EnumMap
EnumMap(Class<K> keyType) Utilizzato per creare una EnumMap vuota con le chiavi del tipo specificato.
EnumMap(EnumMap<K,? extends V> m) Utilizzato per creare una mappa vuota con il tipo della chiave uguale a quello della mappa specificata per parametro.
EnumMap(Map<K,? extends V> m) Utilizzato per creare una EnumMap a partire dalla mappa specificata.

L'uso di un tipo EnumMap è mostrato nel prossimo esempio:

```
enum MesiDellAnno {
  GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO,
  LUGLIO, AGOSTO, SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE
}

  public static void main(String[] args) {
    EnumMap<MesiDellAnno, String> mappa1 = new EnumMap<>(MesiDellAnno.class);
    // Per aggiungere elementi possiamo usare il metodo put
    mappa1.put(MesiDellAnno.GENNAIO, "Gennaio");
    mappa1.put(MesiDellAnno.FEBBRAIO, "Febbraio");
    mappa1.put(MesiDellAnno.MARZO, "Marzo");
```

```
        System.out.println("Mappa 1 contiene: "+mappa1.size()+ " elementi");
        // oppure utilizzare il metodo remove per rimuovere un elemento
        mappa1.remove(MesiDellAnno.MARZO);
        System.out.println("Dopo remove mappa 1 contiene: "+mappa1.size()+ " elementi");
        System.out.println("Creo mappa 2 usando mappa 1");
        EnumMap<MesiDellAnno, String> mappa2 = new EnumMap<>(mappa1);
        System.out.println("Mappa 2 contiene: "+mappa2.size()+ " elementi");
    }
```

Il risultato dell'esecuzione è il seguente:

```
Mappa 1 contiene: 3 elementi
Dopo remove mappa 1 contiene: 2 elementi
Creo mappa 2 usando mappa 1
Mappa 2 contiene: 2 elementi
```

Una Pila Generica

Siamo giunti alla fine di questa sezione; non mi rimane che fare una sola cosa: è arrivato il momento di dire addio alla classe Pila che ci ha accompagnato dall'inizio del libro.

Il JFC mette infatti a disposizione una propria implementazione di una pila con la classe *java.util.Stack*. nell'immagine seguente quella piccolina laggiù in fondo che deriva dalla classe *Vector*.

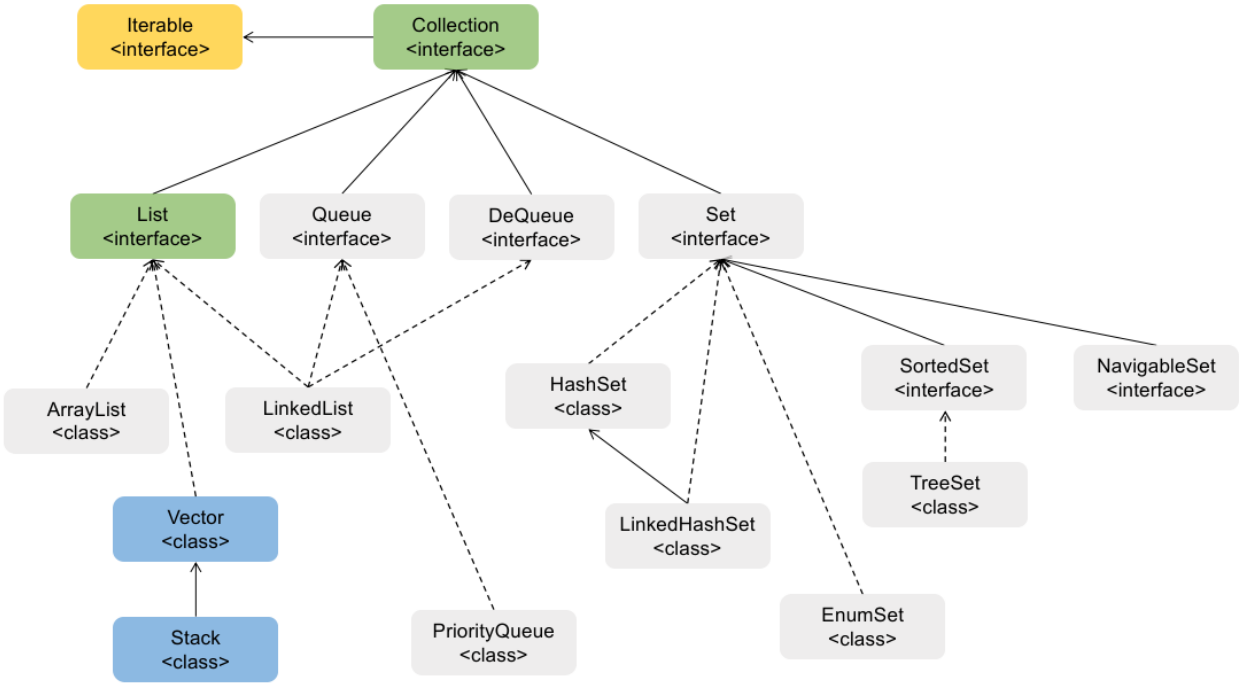


Immagine 48 La classe Stack

Poiché eredita dalla classe *Vector*, la classe Stack è una classe thread-safe che consente duplicazioni di elementi. Data la natura dell'oggetto è banale far notare che mantiene l'ordine di inserimento degli elementi utilizzando però un approccio LIFO (Last In First Out).

Oltre ai metodi *push* e *pop* che conosciamo benissimo mette a disposizione altri metodi come mostrato nella prossima tabella:

Metodi di Stack
public E push(E item)   Aggiunge un oggetto in cima allo stack e lo ritorna come valore della funzione.
public E pop()   Rimuove l'oggetto in cima allo stack e lo ritorna come valore della funzione.
public E peek()   Letteralmente consente di dare un occhiata al primo oggetto dello stack senza rimuoverlo dalla pila.
public boolean empty()   Torna true se la pila è vuota, false altrimenti.
public int search(Object o)   Restituisce la posizione dell'oggetto come espressione della distanza dalla cima dalla pila a partire da 1. Se esiste più di una copia dell'oggetto, torna la distanza dell'oggetto più vicino alla cima.

Iterare su una collezione: forEach e forEachRemaining

Tradizionalmente, l'operazione di iterazione sulle collezioni viene realizzata tramite i comandi **for** e **for in** come mostrato nel prossimo esempio in cui vengono mostrate le due alternative:

```
List<String> citta= Arrays.asList("Roma", "Milano", "Verona");

for(int i=0; i<citta.size(); i++){
   System.out.println(citta.get(i));
}

for(String nome: citta){
   System.out.println(nome);
}
```

In Java 8, l'operazione di iterazione è stata resa più efficiente, grazie all'introduzione dei metodi *forEach* e *forEachRemaining* che consentono di utilizzare l'espressività e la versatilità offerta dalla espressioni lambda. Di fatto rappresentano entrambi la versione funzionale del classico ciclo **for**.

1. forEach(Consumer<? super T> action)

E' un metodo definito nell'interfaccia *Iterable*: riceve come attributo un tipo *Consumer* che rappresenta l'operazione da eseguire sull'elemento corrente della collezione.

2. *forEachRemaining(Consumer<? super E> action)*

Molto simile a *forEach*, possiamo utilizzarlo qualora dovessimo decidere di iterare la collezione mediante un tipo *Iterator*.

Utilizzare il metodo forEach

Possiamo utilizzare *forEach* per iterare una raccolta ed eseguire una determinata azione su ogni elemento. L'azione da eseguire è rappresentata da una interfaccia di tipo *Consumer e* viene passata a forEach come argomento. Come abbiamo già visto, l'interfaccia *Consumer* è un'interfaccia funzionale (un'interfaccia con un singolo metodo astratto): accetta un input e non restituisce alcun risultato.

```
@FunctionalInterfacepublic interface Consumer {
  void accept(T t);
}
```

il maggior beneficio di utilizzare le interfacce funzionali come tipo per l'argomento è la possibilità di utilizzare le *espressioni lambda*, e di conseguenza la *method reference*. Pertanto possiamo riscrivere il frammento di codice già proposto nel modo seguente:

```
List<String> citta= Arrays.asList("Roma", "Milano", "Verona");
citta.forEach(nome-> System.out.println(nome));
```

oppure, in alternativa:

```
citta.forEach(System.out::println);
```

Volendo paragonare *forEach* con il costrutto **for-in**, è evidente che entrambi consentono di iterare sugli elementi di una collezione. Tuttavia una differenza esiste ed è un po più sottile.

Un ciclo **for-in** è un meccanismo esterno alla classe (così come lo sono i tipi *Iterator* e le enumerazioni), e pertanto ad ogni ciclo deve '*portare fuori*' un elemento dalla collezione.

Nonostante il ciclo:

```
for(String nome: citta){
```

Utilizzare il metodo forEachRemaining

Molto simile a *forEach*, consente di sfruttare il meccanismo delle espressioni lambda attraverso l'interfaccia funzionale *Consumer*. Poiché questo metodo è definito nell'interfaccia *Iterator*, l'esempio precedente dovrà essere adattato nel modo seguente:

```
List<String> citta= Arrays.asList("Roma", "Milano", "Verona");
citta.iterator().forEach(nome-> System.out.println(nome));
citta.iterator().forEach(System.out::println);
```

forEachRemaining vs forEach

Per capire le differenza tra i due metodi di iterazione, bisogna capire meglio le differenze tra *Iterable* ed *Iterator*. Un *Iterator*, come abbiamo già visto, è qualcosa che non contiene elementi, possiede un '*next element*' ed una fine. Al contrario, *Iterable* contiene un elenco finito o infinito di elementi che possono essere iterati ottenendo di volta in volta il prossimo elemento.

In pratica, un tipo *Iterable* può essere iterato mediante un tipo *Iterator*.

Ricordiamo che l'interfaccia *Collection* è anche un tipo *Iterable* da cui deriva, e restituisce un tipo *Iterator* per scorrere gli elementi della lista in una direzione.

Data una collezione quindi, chiamando il metodo *forEach* di *Iterable* scorriamo tutti gli elementi dal primo all'ultimo, al contrario se chiamiamo il metodo *next* di *Iterable* prima di iniziare il ciclo, aggiorniamo l'indice dell'elemento corrente e pertanto *forEachRemaining* consentirà di scorrere gli elementi restanti a partire dall'indice corrente.

```
List<String> citta = Arrays.asList("Roma", "Milano", "Verona", "Udine");
citta.forEach(System.out::println);
```

produrrà quindi come output:

```
Roma
```

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

nasconda tutta la complessità al programmatore, dovrà comunque utilizzare implicitamente i metodi *hasNext* e *next* per accedere agli elementi.

Al contrario, *forEach* è un iteratore interno e richiede solo che venga dichiarato cosa deve fare con gli elementi della lista attraverso una espressione lambda.

```
Milano
Verona
Udine
```

Il codice:

```
Iterator<String> iterator = citta.iterator();
iterator.next();
iterator.forEachRemaining(System.out::println);
```

stamperà a video:

```
Milano
Verona
Udine
```

Iterare su una mappa: forEach

Le mappe per loro natura non sono oggetti iterabili in quanto elenchi di elementi rappresentati da coppie <chiave, valore>. Nonostante questo l'interfaccia *Map* fornisce una variante di *forEach* mediante il metodo *default*:

```
default void forEach(BiConsumer<? super K,? super V> action)
```

che prende come attributo un tipo *BiConsumer<T,U>* che, guarda caso, ricorda vagamente la coppia <chiave, valore>. L'interfaccia funzionale *BiConsumer* è definita come segue:

```
public interface BiConsumer<T,U> {
        void accept(T t, U u);
}
```

Come per i casi precedenti, *forEach* consente di traversare tutti gli elementi della mappa come mostrato nel prossimo frammento di codice:

```
Map<String,String> cittaPerRegione = new HashMap<>();
cittaPerRegione.put("Roma", "Lazio");
cittaPerRegione.put("Milano", "Lombardia");
```

```
cittaPerRegione.put("Napoli", "Campania");
cittaPerRegione.forEach((citta,regione)->System.out.println("Città: "+citta+" Regione: "+regione));
```

che produrrà il seguente output a video:

```
Città: Roma Regione: Lazio
Città: Napoli Regione: Campania
Città: Milano Regione: Lombardia
```

Iterare su una mappa: keySet, values ed entrySet

Una alternativa all'utilizzo di *forEach* con una mappa è offerta dai metodi *values()*, *keySet()* ed *entrySet()* che restituiscono rispettivamente: una collezione di tutti i valori memorizzati all'interno della mappa, un insieme delle chiavi, un insieme contenente tutte le entries <chiave, valore> memorizzate nella mappa.

```
Map<String,String> cittaPerRegione = new HashMap<>();
cittaPerRegione.put("Roma", "Lazio");
cittaPerRegione.put("Milano", "Lombardia");
cittaPerRegione.put("Napoli", "Campania");

Set<String> insiemeDelleChiavi = cittaPerRegione.keySet();

Set<Entry<String,String>> insiemeDelleEntry = cittaPerRegione.entrySet();

Collection<String> listaValori = cittaPerRegione.values();
```

Una volta ottenute le tre collezioni possiamo utilizzare gli strumenti già visti nei paragrafi precedenti. Nel prossimo frammento di codice utilizziamo il metodo *forEach* di *Iterable*:

```
insiemeDelleChiavi.forEach(chiave -> {
        String valore = cittaPerRegione.get(chiave);
        cittaPerRegione.forEach((citta, regione) -> System.out.println("Città: "
                + chiave + " Regione: " + valore));
});

insiemeDelleEntry.forEach(entry -> {
        System.out.println("Città: " + entry.getKey() + " Regione: " + entry.getValue());
});

listaValori.forEach(valore -> {
        System.out.println("valore");
});
```

ricordando che le chiavi di una mappa non possono essere duplicate mentre i valori si, ecco il motivo per cui una mappa torna un insieme delle chiavi ed una lista di valori che può contenere duplicati.

Espressioni lambda e gestione delle eccezioni

Le espressioni lambda hanno rappresentato da subito un salto in avanti enorme incoraggiando l'utilizzo della programmazione funzionale grazie ad una sintassi efficace che ha semplificato la creazione di funzioni; sia dal punto di vista del codice che diventa leggero e conciso, sia dal punto di vista della rappresentazione del contesto della funzione mediante le *closure*.. Tuttavia, per loro natura, le espressioni lambda rendono complicato gestire le eccezioni, ma non solo: per gestire le eccezioni il codice torna ad essere verboso, ingombrante e poco maneggevole.

Iniziamo con l'analizzare il caso delle eccezioni di tipo *unchecked*. E per farlo consideriamo il prossimo esempio:

```
List<Integer> divisori = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
divisori.forEach(divisore -> System.out.println(1 / divisore));
```

E' una sintassi che ormai dovrebbe esserci familiare: l'espressione lambda divide il numero 1 per i dividendi estratti dalla lista. Tutto funziona perfettamente finché la lista non contiene lo 0; in questo caso l'operazione *1/0* genererà una eccezione di tipo *AritmeticException.* Per evitare che la applicazione venga interrotta dall'eccezione (ricordiamo che non siamo costretti a gestire questo tipo di eccezioni), usiamo un blocco di guardia, stampiamo a video un messaggio di errore e continuiamo a iterare sulla lista di interi.

```
List<Integer> divisori = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
divisori.forEach(divisore -> {
        try {
                System.out.println(1 / divisore);
        } catch (ArithmeticException e) {
                System.err.println(e.getMessage());
        }
});
```

Ed ecco che improvvisamente il codice torna ad essere fastidiosamente ma inevitabilmente verboso, ma in assenza di un blocco di guardia, il codice:

```
public static void main(String[] args) {
        List<Integer> divisori = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,0);
        divisori.forEach(divisori.forEach(divisore -> System.out.println(1 / divisore));
```

```
            System.out.println("Questa riga non verrà mai eseguita se l'array contiene uno 0 ");
        }
```

se eseguito provocherà una eccezione che risalirà lungo la catena delle chiamate fino al metodo *main* provocando la terminazione immediata dell'eccezione.

```
...
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at javamattone.esercizi.collezioni.eccezioni.EccezioniLambda.lambda$0(EccezioniLambda.java:13)
    at java.base/java.util.Arrays$ArrayList.forEach(Unknown Source)
    at javamattone.esercizi.collezioni.eccezioni.EccezioniLambda.main(EccezioniLambda.java:12)
```

Un modo di evitare il problema mantenendo allo stesso tempo il codice pulito e conciso può essere quella di costruire un metodo wrapper come mostrato nel codice seguente:

```
private static Consumer<Integer> wrapper(Consumer<Integer> consumer) {
  return i -> {
    try {
      consumer.accept(i);
    } catch (ArithmeticException e) {
      System.err.println(
          "Ops ... una eccezione di tipo ArithmeticException  :" + e.getMessage());
    }
  };
}

public static void main(String[] args) {
  List<Integer> divisori = Arrays.asList(0,1, 2, 3, 4, 5, 6, 7, 8, 9);
  divisori.forEach(wrapper(divisore -> System.out.println(1 / divisore)));
}
```

funzionere perfettamente, ma potreste tranquillamente domandarvi se ha senso un approccio che, banalmente, prende un blocco di codice e lo sposta da un altra parte. Se poco ci interessa l'aspetto stilistico del codice, il vantaggio c'è ... eccome!

La tecnica dei metodi *wrapper* è molto utile se affiancata a tipi generici grazie ai quali possiamo creare metodi *wrapper* da utilizzare in una miriade di casi d'uso diversi da quello specifico. Utilizzando i generics potremmo generalizzare il metodo precedente nel modo seguente:

```
static <T, E extends Exception> Consumer<T> wrapper(Consumer<T> consumer, Class<E> clazz) {

  return i -> {
    try {
      consumer.accept(i);
    } catch (Exception ex) {
```

```
        try {
          E exCast = clazz.cast(ex);
          System.err.println(
               "Exception occured : " + exCast.getMessage());
        } catch (ClassCastException ccEx) {
          throw ex;
        }
      }
    };
}

public static void main(String[] args) {
  List<Integer> divisori = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 0);
  divisori.forEach(wrapper(divisore -> System.out.println(1 / divisore), ArithmeticException.class));
}
```

Il nuovo metodo wrapper prende due parametri come argomento: un tipo *T* ed un tipo *Exception*. In questo modo possiamo generalizzare il nostro metodo *wrapper*, e favorirne il riuso del codice in diversi contesti di una applicazione.

Nel caso di eccezioni *checked* le cose cambiano un pochino perché, diversamente dalle prime, siamo costretti a gestirle (le espressioni lambda non fanno sconti in questo caso).

Prendiamo in considerazione il prossimo codice:

```
public static void apriIlFile(String i) throws FileNotFoundException{
  ...
}
public static void main(String[] args) {
  List<String> fileNames= Arrays.asList("file1", "file2");
  listaInteri.forEach(s -> apriIlFile(s));
}
```

Ovviamente, poiché *IOException* è di tipo *checked* il compilatore tornerà un errore in fase di compilazione:

Unhandled exception type IOExceptionJava(16777384)

La tecnica del metodo wrapper è valida anche per le eccezioni di tipo *checked*. Ma cosa succede se vogliamo propagare una eccezione piuttosto che gestirla? Modifichiamo leggermente il codice:

```
public static void trasmetti(Integer i) throws IOException{

}
```

```
public static void main(String[] args) {
   List<String> fileNames= Arrays.asList("file1", "file2");
   listaInteri.forEach(s -> {
     try {
       trasmetti(s);
     } catch (FileNotFoundExceptione e) {
       throw e;
     }
   });
}
```

Anche questa volta otterremo un errore in fase di compilazione in quanto **throw** non è ammesso. ricordando che una espressione funzionale rappresenta l'implementazione di un metodo astratta di una interfaccia funzionale, come tutti i metodi per poter propagare una eccezione *checked* dovrà dichiaralo mediante clausola **throws**. Nel caso di *forEach*, il tipo atteso *Consumer* è definito come segue:

```
public interface Supplier<T> {
        T get();
}
```

L'unico metodo astratto, *get*, non contiene nessuna clausola **throws**. In questo caso, la soluzione sarà catturare l'eccezione come visto in precedenza e fare il *throw* di una eccezione *unchecked* tipo *RuntimeException*.

```
public static void main(String[] args) {
   List<Integer> listaInteri = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 0);

public static void main(String[] args) {
   List<String> fileNames= Arrays.asList("file1", "file2");

   listaInteri.forEach(s -> {
     try {
       trasmetti(s);
     } catch (FileNotFoundExceptione e) {
       throw new RuntimeException(e);
     }
   });
}
```

Il che ci riporterebbe al caso precedente. Per rendere più conciso il codice dell'espressione lambda questa volta possiamo procedere utilizzando una interfaccia funzionale ed un metodo wrapper come segue:

```
@FunctionalInterface
public interface wrappedConsumer<T, E extends Exception> {
  void accept(T t) throws E;
}

static <T> Consumer<T> wrapper(
  wrappedConsumer<T, Exception> wrappedConsumer) {

  return i -> {
    try {
      wrappedConsumer.accept(i);
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  };
}
public static void apriIlFile(String s) throws FileNotFoundException{
}
public static void main(String[] args) {
  List<String> nomiDeiFiles= Arrays.asList("file1", "file2");
  nomiDeiFiles.forEach(wrapper(s->apriIlFile(s)));
}
```

18.La monade Stream - Java Stream API

Introduzione

Nella programmazione le monadi sono strutture che mettono un valore all'interno di un contesto e consentono di operare sul valore mediante concatenazione di funzioni che tornano il contesto stesso: l'output di un diventa l'input di un'altra .

Le monadi hanno molti vantaggi:

1. Riducono la duplicazione di codice;

Sono per definizione funzioni semplici, first-class ed higher-order riusabili nel loro contesto. Come tali:

2. Migliorano la manutenibilità e la leggibilità del codice;

3. Eliminano i side effect;

4. Nascondono la complessità incapsulando i dettagli implementativi;

5. Sono fortemente orientate alla composizione di funzioni.

La possibilità di concatenare funzioni di una monade come pipeline di operazioni è uno degli aspetti più rilevanti. Le pipeline sono strutture con caratteristiche uniche ed interessanti: prima di ogni cosa consentono di parallelizzare il flusso delle operazioni incrementano il *throughput* (ovvero la quantità di istruzioni eseguite in una data quantità di tempo); secondo, poiché le operazioni sono eseguite in maniera asincrona, migliorano il rendimento generale di ogni funzione, e di conseguenza le prestazioni dell'intera applicazione. In generale, più lunga è la pipeline, più saranno le istruzioni eseguite simultaneamente. Tuttavia, esistono anche alcuni svantaggi: il blocco di una delle funzioni potrebbe causare lo svuotamento dell'intera struttura.

L'approccio tramite pipeline è detto *pipelining*.

A partire da Java 8 sono, le monadi sono diventate parte del linguaggio che ne fornisce diverse implementazioni. Grazie alle espressioni lambda, il pipelining è diventata una tecnica apprezzata e largamente utilizzata nella moderna programmazione Java.

In questa sezione ci occuperemo degli Stream, una monade che rappresenta una sequenza di elementi come un oggetto che contiene una serie di metodi che possono essere concatenati tra loro a formare una pipeline di funzioni esprimibili in termini di espressioni lambda.

Gli Stream, a differenza delle collezioni, non sono strutture dati create per contenere gruppi di oggetti, sono flussi di dati che utilizzano collezioni come sorgenti di dati, come anche array oppure canali di input come ad esempio files. Utilizzano strumenti interni per iterare sugli elementi: usando gli stream il programmatore non deve preoccuparsi della gestione del pipelining potendosi invece concentrare sulle operazioni che le singole fasi di pipelining devono eseguire sui dati.

In questa sezione impareremo ad utilizzare gli stream Java: capiremo come crearli ed analizzeremo le classi e le interfacce che compongono le Java Stream API.

Approfondiremo i diversi tipi di operazioni che possiamo applicare ai dati in uno stream, ed infine paleremo di come Java supporta il parallelismo mediante strutture particolari chiamate *parallel stream*.

Dagli iteratori agli stream

Immaginiamo di avere una lista di interi arbitrariamente grande e di voler creare una nuova lista che contenga tutti gli elementi della prima che sono diversi da 3. Una possibile soluzione è quella riportata nel prossimo frammento di codice ed è basata su un *Iterator* ed un ciclo **while**.

```
List<Integer> elencoDiInteri = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 3);
List<Integer> elencoDiInteriModificato = new ArrayList<>();
Iterator<Integer> iterator = elencoDiInteri.iterator();
while (iterator.hasNext()) {
  Integer interoCorrente = iterator.next();
  if (interoCorrente != 3)
    elencoDiInteriModificato.add(interoCorrente);
}
```

In alternativa potremmo utilizzare *forEach* per ottenere un codice più compatto e leggibile:

```
List<Integer> elencoDiInteri = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 3);
List<Integer> elencoDiInteriModificato = new ArrayList<>();
 elencoDiInteri.forEach(interoCorrente -> {
        if (interoCorrente != 3)
              elencoDiInteriModificato.add(interoCorrente);
 });
```

Se invece utilizzassimo gli stream il codice diventerebbe il seguente:

```
List<Integer> elencoDaStream = elencoDiInteri.stream().filter(i -> i != 3).toList();
```

Con gli *stream* non è più necessario preoccuparsi della gestione del ciclo, del confronto e della costruzione della nuova lista. I nomi delle operazioni descrivono perfettamente cosa andrà a realizzare la pipeline, ma sopratutto lo stream sarà in autonomo nello schedulare la sequenza delle operazioni nella pipeline garantendo però risultato finale sarà quello atteso.

```
List<Integer> elencoDaStream = elencoDiInteri.parallelStream().filter(i -> i != 3).toList();
```

In alternativa possiamo utilizzare *parallelStream*, che a differenza del precedente, tenderà a favorire l'esecuzione parallela delle operazioni.

A differenza degli stream, in un ciclo classico è necessario specificare esattamente come andrà processato il dato senza lasciare spazio a possibili ottimizzazioni. Gli stream abbracciano il

principio *"what not how"*: come è evidente nel nostro esempio, il programmatore si dovrà concentrare su cosa fare senza doversi preoccupare di come verrà fatto: ordine del ciclo e lo scheduling delle funzioni.

Gli stream sono quindi monadi che fatte per concentrarsi sul cosa e mail sul come. Dall'esempio notiamo che:

1. Gli stream non contengono i dati che processano, ma sono creati a partire da una sorgete di dati.

2. Le operazioni sugli stream non modificano la sorgente dati.

Nell'esempio, *filter* non modifica la lista di partenza.

Operazioni intermedie ed operazioni terminali

Gli stream sono definiti a partire dell'interfaccia generica *java.util.stream.Stream,* la cui gerarchia è schematizzata nella prossima immagine.

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {
....
}
```

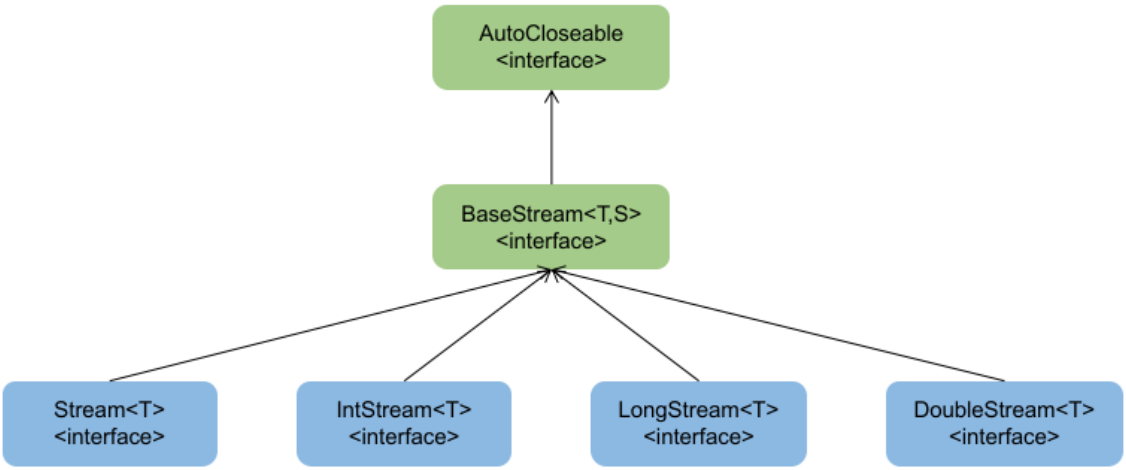


Immagine 49 Gerarchia java.util.stream

java.util.stream.Stream contiene la definizione di molti metodi alcuni dei quali tornano a loro volta un tipo *Stream.* I metodi che ritornano un tipo *Stream* possono essere concatenati tra loro come nel prossimo esempio in cui iteriamo su un elenco di nomi per ritornare l'elenco dei primi 10 che iniziano per la lettera A:

```
List <String> nomiPerA =
  nomi.stream()
    .filter(name -> name.startsWith("A"))
    .limit(10)
    .collect(Collectors.toList());
```

In generale, il flusso di lavoro di uno stream è sempre lo stesso:

1. Lo stream viene creato;

2. Concatena le operazioni intermedie come filter, map, limit, reduce, find, match, e tante altre che trasformano la sorgente in qualcosa di diverso per passi successivi.

3. *Applica l'operazione terminale per produrre il risultato.*

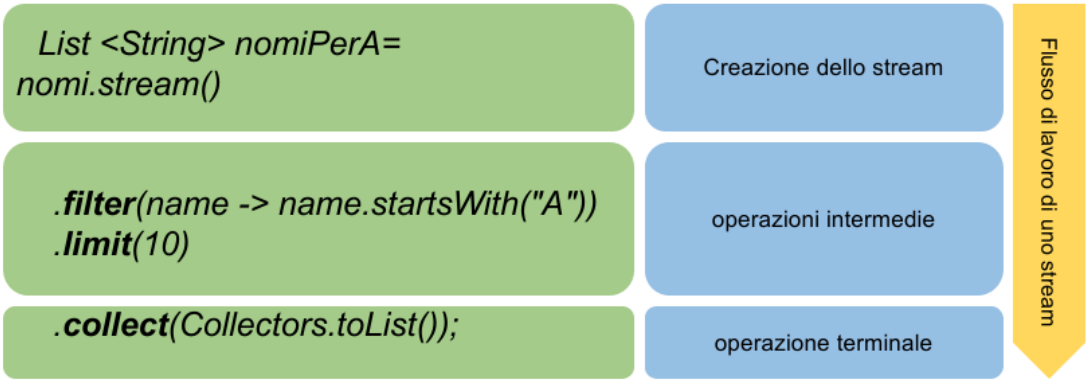


Immagine 50 Flusso di lavoro di uno stream

Valgono quindi le seguenti definizioni:

DEFINIZIONE: definiamo operazioni intermedie, i metodi che restituiscono un tipo Stream e quindi possono essere concatenati a formare una pipeline di operazioni.

DEFINIZIONE: definiamo operazioni terminali le operazioni che possono essere utilizzate per chiudere la pipeline.

Le operazioni terminali sono quindi le funzioni che operano sullo stream modificato e chiudono la pipeline in qualche modo. Nell'esempio, *collect* è l'*operazione terminale* che trasforma lo stream modificato in una lista di stringhe.

Se ad esempio volessimo solo stampare i nomi che iniziano per A, l'operazione terminale sarebbe *forEach*:

```
nomi.stream()
  .filter(name -> name.startsWith("A"))
  .limit(10)
  .forEach(s -> System.out.println(s));
}
```

Più in generale, i metodi definiti nell'interfaccia *Stream* sono suddivisibili in *intermedi* e *terminali*. I metodi intermedi sono quelli che generano uno stream, gli

altri sono da considerare metodi terminali a meno di alcuni altri che sottintendono alla creazione di uno stream e che vedremo a seguire.

Streams - Lazy evaluation

La cosa più interessante sugli è che gli stream sono pigri: le *operazioni intermedie* sono di tipo lazy: vengono applicate solo quando è necessario ottenere un risultato ovvero quando viene invocata l'*operazione terminale*.

Questo approccio, chiamato *lazy-evaluation*, è una caratteristica fondamentale con un impatto rilevante sulle prestazioni sopratutto quando dobbiamo manipolare stream di grandi dimensioni. Cerchiamo di capire meglio di cosa si tratta.

Concettualmente, l'idea è quella di posticipare l'elaborazione dei dati fino a che non si è sicuri di quali dati debbano essere realmente utilizzati, ed eventualmente concentrarsi solo su un sottoinsieme di essi. Qualora dovessimo trattare flussi di dati di grandi dimensioni è evidente che questa caratteristica rappresenterebbe un evidente vantaggio.

Il meccanismo usato dagli stream java per implementare *lazy evaluation* è schematizzato nella prossima figura.

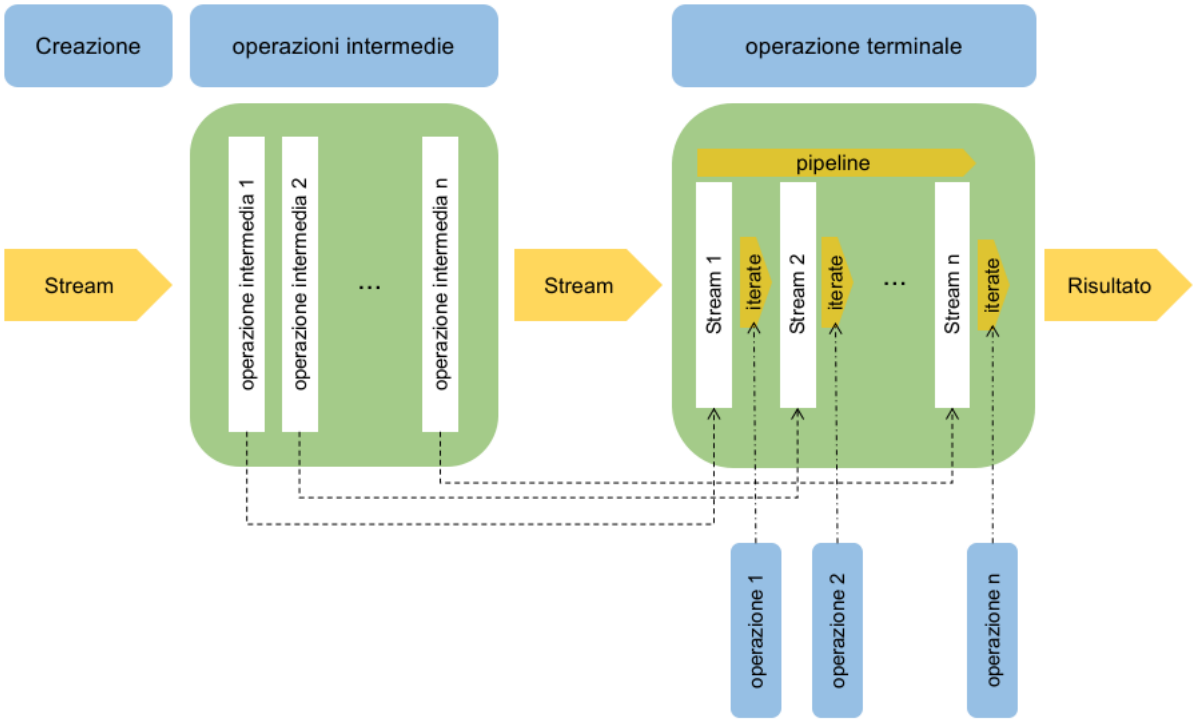


Immagine 51 lazy evaluation

Le operazioni intermedie non sono eseguite fino a che non viene invocata l'operazione terminale. Nella fase preparatoria ogni operazione intermedia crea uno stream a cui associa la funzione passata come attributo al metodo chiamato. Gli stream vengono quindi accumulati in una pipeline, e tuttavia nessuna delle operazioni terminali viene eseguita.

Non appena viene invocata l'operazione terminale, inizia la processazione della pipeline e viene prodotto il risultato. Gli algoritmi interni sono ottimizzati per eseguire le operazioni in maniera estremamente efficiente.

Per meglio comprendere il meccanismo di *lazy-evaluation* vediamo alcuni esempi. Nel prossimo vedremo come, in effetti, la processazione degli stream inizia solo al momento dell'invocazione della funzione terminale.

```
List<String> nomi = Arrays.asList("Antonio", "Andrea", "Giuseppe");

Stream<String> streamDiNomi = nomi.stream()
    .filter(name -> {
      System.out.println("Nome da filtrare: "+name);
      return name.startsWith("A");
    });
System.out.println("Mi fermo per 3 secondi");
Thread.sleep(3000);
System.out.println("Sono passati 3 secondi");
streamDiNomi.forEach(s -> System.out.println(s));
```

Nell'esempio l'*operazione terminale* viene eseguita solo dopo che siano passati 3 secondi dalla applicazione delle *operazioni intermedie*. L'output dimostra che tutte le operazioni intermedie sono eseguite solo quando viene chiamato il metodo *forEach*.

```
Mi fermo per 3 secondi
Sono passati 3 secondi
Nome da filtrare: Antonio
Antonio
Nome da filtrare: Andrea
Andrea
Nome da filtrare: Giuseppe
```

Da notare che, sostituendo lo *stream* con un *parallelStream*, il risultato rimane invariato, ma le pipeline vengono processate in parallelo:

```
Mi fermo per 3 secondi
Sono passati 3 secondi
Nome da filtrare: Andrea
Nome da filtrare: Antonio
Nome da filtrare: Giuseppe
Andrea
Antonio
```

Notiamo inoltre che l'operazione terminale, in effetti, si troverà ad operare solo sui dati che effettivamente vogliamo processare.

Operazioni short-circuit

Le cartucce a disposizione per ottimizzare gli stream non finiscono qui in quanto gli stream implementano il concetto di operazioni *short-circuit*.

DEFINIZIONE: sono definiti short-circuit i metodi dell'interfaccia Stream che interrompono l'esecuzione la processazione della pipeline non appena la loro condizione viene soddisfatta.

I seguenti metodi sono da considerarsi short-circuit

```
boolean anyMatch(Predicate<? super T> predicate);
boolean allMatch(Predicate<? super T> predicate);
 boolean noneMatch(Predicate<? super T> predicate);
Stream<T> limit(long maxSize);
Optional<T> findFirst();
Optional<T> findAny();
```

Basta modificare l'esempio precedente come segue:

```
List<String> nomi = Arrays.asList("Antonio", "Andrea", "Giuseppe");
Stream<String> streamDiNomi = nomi.stream()
    .filter(name -> {
      System.out.println("Nome da filtrare: "+name);
      return name.startsWith("A");
    }).limit(1);
System.out.println("Mi fermo per 3 secondi");
Thread.sleep(3000);
System.out.println("Sono passati 3 secondi");
streamDiNomi.forEach(s -> System.out.println(s));
```

L'output dimostra che non appena stampato il primo nome, l'esecuzione delle restanti operazioni nella pipeline vengono immediatamente interrotte:

```
Mi fermo per 3 secondi
Sono passati 3 secondi
Nome da filtrare: Andrea
Nome da filtrare: Antonio
Antonio
```

I metodi dell'interfaccia Stream

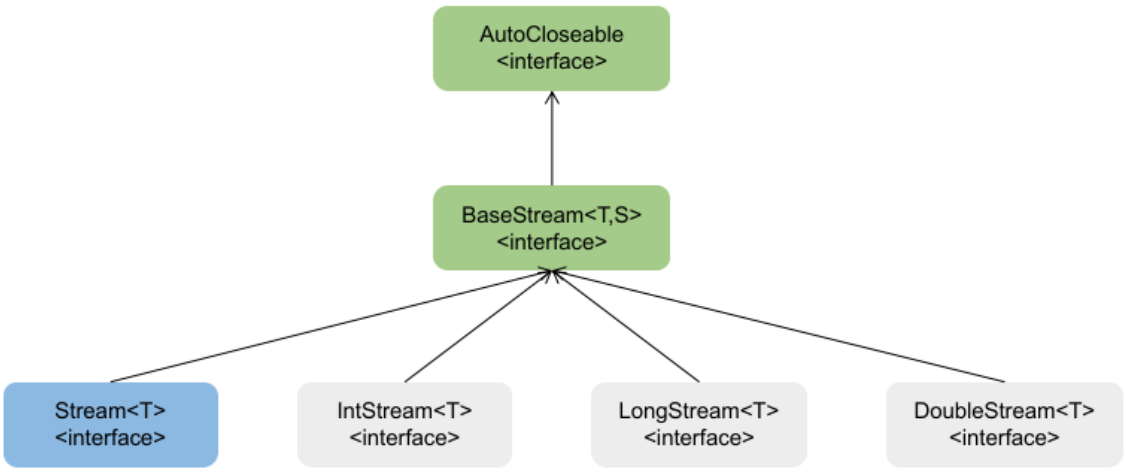


Immagine 52 interfaccia Stream

Nella prossima tabella sono elencati i metodi principali dell'interfaccia Stream focalizzando sui metodi *creazionali* (ovvero che sottintendono alla creazione di uno stream), *intermedi e terminali*. Per la lista completa fare riferimento alla documentazione ufficiale a corredo del vostro JDK.

I metodi riportati nella tabella fanno riferimento all'ultima versione LTS di Java, Java 17, potrebbero pertanto esserci alcune differenti rispetto alle versioni precedenti del linguaggio.

Metodi dell'interfaccia Stream
Operazioni creazionali
static <T> Stream.Builder<T> builder(); Ritorna uno stream builder.
static <T> Stream<T> empty() Ritorna uno stream vuoto.
static <T> Stream<T> generate(Supplier<? extends T> s) Restituisce un flusso non ordinato, sequenziale, infinito in cui ogni elemento è generato dal Supplier fornito. Questo metodo è adatto per generare stream costanti, stream di elementi casuali, ecc.
static <T> Stream<T> of(T t) Restituisce uno stream con un solo elemento.
static <T> Stream<T> of(T... values) Restituisce uno stream i cui elementi sono quelli specificati come argomenti
static <T> Stream<T> ofNullable(T t) Se T è non nullo, allora restituisce uno stream con un solo elemento. Altrimenti restituisce uno stream vuoto.

static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)

Restituisce uno Stream ordinato, sequenziale, infinito prodotto dall'applicazione iterativa di una funzione *f* a un elemento *seed* iniziale, producendo uno Stream costituito da *seed*, *f(seed)*, *f(f(seed))*, *ecc*.

Il primo elemento (posizione 0) nello stream sarà il seme fornito. Per n > 0, l'elemento in posizione n sarà il risultato dell'applicazione della funzione f all'elemento in posizione n - 1.

Operazioni terminali

boolean anyMatch(Predicate<? super T> predicate);

Restituisce *true* se gli elementi di questo stream corrispondono al predicato fornito.

E' una operazione terminale short-circuit.

boolean allMatch(Predicate<? super T> predicate);

Restituisce *true* se tutti gli elementi di questo stream corrispondono al predicato fornito.

E' una operazione terminale short-circuit.

<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)

Esegue un'operazione di riduzione mutabile sugli elementi di questo stream. Una riduzione mutabile è quella in cui il valore prodotto dalla riduzione è un contenitore di risultati mutabili, ad esempio un *ArrayList*, in cui gli elementi vengono incorporati.

<R, A> R collect(Collector<? super T,A,R> collector)

Esegue un'operazione di riduzione mutabile sugli elementi di questo stream usando un tipo *Collector*.

long count()

Conta in numero degli elementi nello stream.

Optional<T> findAny()

Ritorna un Optional vuoto se lo stream è vuoto, altrimenti un Optional che descrive uno qualunque degli elementi dello stream corrente. Da notare che il risultato di questo metodo è totalmente non deterministico.

E' una operazione terminale short-circuit.

Optional<T> findFirst()

Ritorna un Optional vuoto se lo stream è vuoto, altrimenti un Optional che descrive il primo elemento dello stream corrente. Da notare che se lo stream non è ordinato, il risultato di questo metodo è totalmente non deterministico.

E' una operazione terminale short-circuit.

void forEachConsumer<? super T> action)

Esegue una operazione rappresentata da *Consumer* ad ogni elemento dello stream corrente.

void forEachOrdered(Consumer<? super T> action)

Esegue una operazione rappresentata da *Consumer* ad ogni elemento dello stream corrente. Se per lo stream corrente è definito un ordine, allora le operazioni vengono effettuato nell'ordine definito.

Optional<T> max(Comparator<? super T> comparator)

Restituisce l'elemento maggiore dello stream corrente in base al Comparator fornito. Questo è un caso particolare di riduzione.

Optional<T> min(Comparator<? super T> comparator)

Restituisce l'elemento minore dello stream corrente in base al Comparator fornito. Questo è un caso particolare di riduzione.

boolean noneMatch(Predicate<? super T> predicate)

Restituisce true se nessun elemento di questo flusso corrisponde al predicato fornito. Se possibile, non applica il predicato a tutti gli elementi. Se il flusso è vuoto, viene restituito true e il predicato non viene valutato.

E' una operazione terminale short-circuit.

Optional<T> reduce(BinaryOperator<T> accumulator)

Esegue una riduzione sugli elementi del flusso corrente, utilizzando una funzione di accumulazione associativa, e restituisce un Optional che descrive il valore ridotto, se presente.

Object[] toArray()

Ritorna un array contenente tutti gli elementi dello stream corrente.

<A> A[] toArray(IntFunction<A[]> generator)

Restituisce un array contenente gli elementi dello stream corrente, utilizzando la funzione *generator* per allocare l'array restituito, nonché eventuali array aggiuntivi che potrebbero essere necessari per un'esecuzione partizionata o per il ridimensionamento.

default List<T> toList()

Accumula gli elementi dello stream corrente in un tipo List. Gli elementi nell'elenco saranno inseriti nell'ordine in cui compaiono nello stream corrente. La Lista restituita è immutabile; le chiamate a qualsiasi metodo *mutator* provocheranno sempre la generazione di *UnsupportedOperationException*.

Non ci sono garanzie sul tipo di implementazione dell'elenco restituito.

Operazioni intermedie

Stream<T> distinct()

Ritorna uno stream contenente gli elementi *distinti seconso il metodo equals(o)* dello stream di partenza.

Stream<T> filter(Predicate<? super T> predicate)

Restituisce uno stream costituito dagli elementi di dello stream originario che corrispondono al

predicato specificato.

<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)

Restituisce uno stream costituito dai risultati della sostituzione di ogni elemento del flusso corrente con i contenuti dello stream mappato prodotto applicando la funzione di mappatura fornita a ogni elemento. Ogni flusso mappato viene chiuso dopo che i relativi contenuti sono stati inseriti nel flusso corrente.

Stream<T> limit(long maxSize)

Restituisce uno stream costituito dagli elementi del flusso corrente, troncati in modo da non essere più lunghi di *maxSize* in lunghezza.

E' una operazione intermedia short-circuit.

<R> Stream<R> map(Function<? super T,? extends R> mapper)

Restituisce uno stream costituito dai risultati dell'applicazione della funzione mapper agli elementi dello stream corrente.

default <R> Stream<R> mapMulti(BiConsumer<? super T,? super Consumer<R>> mapper)

Restituisce uno stream costituito dai risultati della sostituzione di ogni elemento del flusso corrente con più elementi, in particolare zero o più elementi. La sostituzione viene eseguita applicando la funzione di mappatura, *mapper*, a ciascun elemento insieme a un argomento C*onsumer* che accetta elementi per la sostituzione. La funzione di mappatura chiama C*onsumer* zero o più volte per fornire gli elementi sostitutivi.

Stream<T> peek(Consumer<? super T> action)

Restituisce un flusso costituito dagli elementi del flusso corrente. L'azione fornita viene eseguita su ciascun elemento solo man mano che gli elementi vengono consumati dallo stream risultante.

Stream<T> skip(long n)

Restituisce uno stream costituito dagli elementi rimanenti del flusso corrente dopo aver scartato i primi n elementi. Se lo stream corrente contiene meno di n elementi, verrà restituito uno stream vuoto.

Stream<T> sorted()

Restituisce uno stream costituito dagli elementi dello stream corrente, ordinati secondo l'ordine naturale. Se gli elementi di questo flusso non sono di tipo *Comparable*, potrebbe essere generata una *java.lang.ClassCastException* quando viene eseguita l'operazione terminale.

Stream<T> sorted(Comparator<? super T> comparator)

Restituisce uno stream costituito dagli elementi dello streamcorrente, ordinati in base al *Comparator* fornito.

default Stream<T> takeWhile(Predicate<? super T> predicate)

Se lo stream corrente è ordinato, restituisce un flusso costituito dal prefisso più lungo di elementi presi da dallo stream corrente che corrispondono al predicato specificato. Altrimenti restituiscuno stream costituito da un sottoinsieme di elementi presi da questo flusso che corrispondono al predicato dato.

Se il flusso corrente è ordinato, il prefisso più lungo è una sequenza contigua di elementi dello stream corrente che corrispondono al predicato dato. Il primo elemento della sequenza è il primo elemento dello stream corrente, e l'elemento immediatamente successivo all'ultimo elemento della sequenza non corrisponde al predicato specificato.

Se questo flusso non è ordinato e alcuni (ma non tutti) elementi di questo flusso corrispondono al predicato dato, allora il comportamento di questa operazione è non deterministico; è libero di prendere qualsiasi sottoinsieme di elementi corrispondenti (che include l'insieme vuoto).

Indipendentemente dal fatto che lo stream sia ordinato o non ordinato, se tutti gli elementi dello stream corrente corrispondono al predicato dato, questa operazione prende tutti gli elementi (il risultato è lo stesso dell'input), o se nessun elemento dello stream corrisponde al predicato dato, allora nessun elemento verrà preso ed il risultato sarà uno stream vuoto.

E' una operazione intermedia short-circuit.

Altre operazioni

static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)

Crea un flusso concatenato i cui elementi sono tutti gli elementi del primo flusso seguiti da tutti gli elementi del secondo flusso. Il flusso risultante è ordinato se entrambi i flussi di input sono ordinati e parallelo se uno dei flussi di input è parallelo.

Questo metodo è di tipo lazy. Per una completa definizione di flusso lazy fare riferimento ai prossimi paragrafi di questo capitolo.

Creare uno stream

Come abbiamo già anticipato, nel caso di collezioni possiamo utilizzare i metodi *stream()* e *parallelStream()* per creare rispettivamente uno stream ed uno stream parallelo. Esistono tuttavia molti modi di creare stream; analizziamo quelli più comunemente utilizzati.

Il metodo empty()

Partiamo quindi dalla creazione di uno stream vuoto:

```
Stream<String> streamVuoto = Stream.empty();
```

Uno stream vuoto è generalmente utilizzato come valore di ritorno di metodi in alternativa a **null**.

Il metodo of ...

Consente di creare uno stream a partire dall'elenco dei valori passati come attributi al metodo *of* che abbiamo visto avere due varianti: la prima che accetta un solo valore e torna uno stream con un elemento, la seconda che accetta un numero variabile di valori.

```
Stream.of("bianco");
Stream.of("bianco", "rosso", "giallo", "blu", "verde");
```

Purtroppo il metodo *of* non consente di utilizzare attributi nulli quindi

```
Stream.of(null);
```

produrrebbe una eccezione di tipo *NullPointerException*. In questi casi possiamo utilizzare l'alternativa *null-safe:*

```
Stream.ofNullable(null);
```

Il metodo builder()

In alternativa al metodo *of*, possiamo utilizzare il metodo *builder* che consente di creare uno stream utilizzando il pattern builder. In questo caso, l'esempio del caso precedente potrà essere riscritto come segue:

```
Stream<String> streamDaBuilder = Stream.<String>builder()
.add("bianco")
.add("rosso")
.add("giallo")
.add("blu")
.add("verde").build();
```

Quando lo stato di una classe Java ha molti molti campi, la creazione di un oggetto tramite costruttore diventa un qualcosa di infernale: molti campi potrebbero essere nulli e potrebbe essere necessario implementare anche molti costruttori diversi per diversi set di variabili di stato. Il pattern builder.

Nella programmazione ad oggetti tale pattern è molto di voga poiché separa la costruzione di un oggetto complesso dalla sua rappresentazione, cosicché il processo di costruzione stesso possa creare diverse rappresentazioni.

Ciò ha l'effetto immediato di rendere più semplice la classe, permettendo a una classe *builder* separata di focalizzarsi sulla corretta costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti.

Il metodo generate

Il metodo generate() accetta un *Supplier<T>* per la generazione degli elementi. Quando si utilizza questo metodo, va ricordato che potrebbe essere generato uno Stream di grandezza infinita quindi sarà responsabilità del programmatore impostare un limite raggiunto il quale la generazione verrà interrotta. Per far questo possiamo combinare il metodo *generate* con il metodo *limit*. Poiché *limit* è un metodo di tipo short-circuit, la generazione dello stream sarà interrotta non appena sarà verificato il predicato passato come attributo a limit.

```
Stream<Double> streamCasuale= Stream.generate(new java.util.Random()::nextDouble).limit(10);
```

Il metodo iterate

Un altra alternativa è il metodo iterate

```
Stream<Integer> streamIterated = Stream.iterate(2, n -> n + 2).limit(20);
```

Il primo elemento del flusso risultante è il primo parametro del metodo *iterate: 2*. Per ogni elemento successivo, la funzione specificata viene applicata all'elemento precedente. Nell'esempio sopra il secondo elemento sarà di conseguenza: 4.

Come per generate, è necessario impostare un limite alla creazione dello stream per evitare che sia generato uno stream infinito.

Stream di array

Come anticipato, un *array* può essere utilizzato come sorgente per uno stream. La classe *java.util.Arrays* mette a disposizione il metodo stream per costruire uno stream a partire da un array.

```
String[] arrayDiStringhe = { "bianco", "rosso", "giallo", "blu", "verde" };
Stream<String> atreamDiStringhe = Arrays.stream(arrayDiStringhe);
```

String come sorgente di uno stream

Il metodo *splitAsStream* della classe *java.util.regex.Pattern*, consente si utilizzare una espressione regolare e spezzare la stringa iniziale in token per poi restituire lo stream relativo:

```
Stream<String> stringAsStream = Pattern.compile(",").splitAsStream("bianco,rosso,giallo,blu,verde");
```

Operazioni sugli stream: operazioni intermedie

L'interfaccia *Stream* contiene un grande numero di operatori intermedi per operare con gli stream; gli operatori intermedi sono quelli che restituiscono un altro stream, e possono essere concatenati tra loro a formare pipelines complesse. Prima di rimboccarci le maniche ed analizzarne alcuni prepariamo la seguente lista di stringhe su cui andremo a costruire tutti gli esempi a seguire:

```
List<String> cittaItaliane = Arrays.asList(
        "Ancona", "Milano", "Roma", "Padova",
        "Napoli", "Rovigo", "Ravenna",
        "Varese", "Lecce", "Bari");
```

Le operazioni intermedie applicate ad una pipeline possono avere un peso diverso a seconda dell'ordine in cui vengono eseguite. Esiste una regola, più che altro dettata dal buonsenso, che generalmente andrebbe rispettata per ottimizzare le prestazioni della pipeline:

Le operazioni intermedie che riducono la grandezza di uno Stream (es: *filter* oppure *distinct*) vanno messe sempre prima delle operazioni che prendono in considerazione tutti gli elementi dello stream (es: *map*)).

Dal memento che gli stream operano con una logica *lazy*, ovvero eseguono le operazioni intermedie solo al momento dell'esecuzione di una operazione terminale, il prossimo frammento di codice non sarà mai eseguito:

```
cittaItaliane.stream().filter(citta->citta.startsWith("R"))
```

Il metodo filter

Il metodo filter() accetta un predicato che utilizza per filtrare tutti gli elementi dello Stream. Questa operazione intermedia crea un nuovo Stream contenente tutti gli elementi che soddisfano il predicato e può essere utilizzato chiamare un'altra operazione intermedia o terminale.

```
cittaItaliane.stream().filter(citta->citta.startsWith("R")).forEach(System.out::println);

Roma
Rovigo
Ravenna
```

Essendo una operazione intermedia, può essere utilizzata più volte per eseguire filtri in serie.

```
cittaItaliane.stream().filter(citta->citta.startsWith("R"))
.filter(citta->citta.endsWith("a")).forEach(System.out::println);

Roma
Ravenna
```

Il metodo map

Map può essere utilizzato per trasformare gli elementi dello stream: produce un altro stream applicando agli elementi dello stream corrente la funzione mapper passata per argomento.Nel prossimo esempio utilizziamo map per trasformare gli elementi dello stream in stringhe contenenti tutti caratteri minuscoli:

```
cittaItaliane.stream().map(String::toLowerCase).forEach(System.out::println);

ancona
milano
```

roma
padova
napoli ...

Può essere concatenato con altri operatori intermedi:

```
cittaItaliane.stream().map(String::toLowerCase)
.filter(citta->citta.startsWith("a")).forEach(System.out::println);
```

ancona

Può essere utilizzato anche per cambiare il *tipo* degli elementi nello stream:

```
cittaItaliane.stream().map(String::hashCode).forEach(System.out::println);
```

1965541612
-1990238798
2553009
...

Il metodo skip

Il metodo skip produce uno stream ottenuto sorvolando il primi n elementi dello stream corrente:

```
cittaItaliane.stream().skip(5).map(String::toUpperCase).forEach(System.out::println);
```

che una volta eseguito produrrà il seguente output a terminale:

ROVIGO
RAVENNA
VARESE
LECCE
BARI

Da notare che nell'esempio *map* è stato utilizzato dopo *skip* per ridurre il numero di elementi a cuo applicare la trasformazione *String::toUpperCase*.

Il metodo sorted

Modificando il codice dell'esempio precedente come segue:

```
cittaItaliane.stream().map(citta->{
        System.out.println(citta);
        return citta.toUpperCase();
}).skip(5).forEach(System.out::println);
```

Il metodo sorted() è un'operazione intermedia che restituisce una *vista ordinata* dello stream. Gli elementi nello stream sono ordinati in ordine naturale a meno che non passiamo un Comparator personalizzato come argomento.

Dal risultato dell'esecuzione possiamo notare una cosa interessante riguardo al pipelining delle funzioni intermedie:

Stream di map	*Stream di skip -> forEach*
Ancona	*Rovigo*
Milano	*ROVIGO*
Roma	*Ravenna*
Padova	*RAVENNA*
Napoli	*Varese*
	VARESE
	Lecce
	LECCE
	Bari
	BARI

Anche se non stiamo utilizzando un *parallelStrem*, in maniera del tutto indipendente, lo scheduler ha deciso di eseguire le due pipeline in cascata eseguendo *skip* contemporaneamente a *map* man mano che i dati sono disponibili.

Questo comportamento è alla base dell'ottimizzazione del throughput degli stream rispetto rispetto agli iteratori classici.

```
cittaItaliane.stream().map(String::toUpperCase).sorted().forEach(System.out::println);
```

```
ANCONA
BARI
LECCE
MILANO
NAPOLI
PADOVA
....
```

Il metodo *sorted* crea semplicemente una vista ordinata dello stream senza manipolare la collezione sorgente.

Operazioni sugli stream: operazioni terminali

Le operazioni terminali sono utilizzate per restituire un risultato di un qualche tipo dopo l'elaborazione di tutti gli elementi dello stream.

In logica *lazy evaluation*, una volta invocata provocherà l'esecuzione degli stream nella pipeline producendo il risultato finale.

Il metodo forEach

Lo abbiamo già incontrato nel paragrafo precedente, consente di iterare sugli elementi dello stream eseguendo le operazioni definite dall'espressione lambda passata per argomento.

```
cittaItaliane.stream().map(String::toLowerCase)
.filter(citta->citta.startsWith("a")).forEach(System.out::println);

ancona
```

Il metodo collect

Il metodo *collect* viene utilizzato per raccogliere gli elementi ed effettuare varie operazioni di riduzione come accumulare gli elementi in collezioni, raggruppare gli elementi dello stream corrente secondi vari criteri. Il metodo prende come argomento un tipo *Collectors* che contiene moltissimi metodi utili che coprono la maggior parte dei casi utili.

Il caso più semplice è utilizzare collect per collezionare gli elementi dello stream in una collezione:

```
// Accumula le citta in una lista
List<String> list = cittaItaliane.stream().collect(Collectors.toList());

// Accumula le citta in un TreeSet
Collection<String> set = cittaItaliane.stream().collect(Collectors.toCollection(TreeSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = cittaItaliane.stream().collect(Collectors.joining(", "));
```

Consideriamo adesso la classe seguente:

```
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Dipendente{
  private String nome;
  private String cognome;
  private String dipartimento;
  private double stipendio;
}
```

Nel prossimo esempio, alcune delle possibilità offerte dalla classe *Collectors*:

```
List<Dipendente> dipendenti = new ArrayList<>();
dipendenti.add(Dipendente.builder().nome("Massimiliano").cognome("Tarquini")
    .eta(30).stipendio(2500.00)
    .dipartimento("Software Factory").build());
dipendenti.add(Dipendente.builder().nome("Giuseppe").cognome("Marascia")
    .eta(25).stipendio(2350.00)
    .dipartimento("Personale").build());
dipendenti.add(Dipendente.builder().nome("Massimo").cognome("Morucci")
    .eta(25).stipendio(2350.00)
    .dipartimento("Software Factory").build());

// ritorna la somma dei salari dei dipendenti
Double sommaDeiSalari = dipendenti.stream()
     .collect(Collectors.summingDouble(Dipendente::getStipendio));
System.out.println("La somma dei salari è: " + sommaDeiSalari);

// estrae la media dei salari
Double mediaDeiSalari = dipendenti.stream()
     .collect(Collectors.averagingDouble(Dipendente::getStipendio));
System.out.println("La media dei salari è: " + mediaDeiSalari);

// raggruppa gli elementi per dipartimento
Map<String, List<Dipendente>> dipendentiPerDipartimento = dipendenti.stream()
    .collect(Collectors.groupingBy(Dipendente::getDipartimento));
```

I metodi allMatch, noneMatch, anyMatch

Uno stream mette a disposizione una serie di metodi per il matching che possono essere usati per verificare se un predicato, passato come argomento, soddisfa gli elementi dello stream. Tutte le operazioni di matching restituiscono un risultato booleano.

I metodi *allMatch*, *noneMatch*, *anyMatch* ritornano true rispettivamente se: tutti gli elementi soddisfano il predicato, nessun elemento soddisfa il praticato, almeno un elemento soddisfa il predicato. Pertanto, l'output del prossimo esempio sarà:

```
boolean matchedResult = cittaItaliane.stream().anyMatch((s) -> s.startsWith("A"));
System.out.print(matchedResult);
matchedResult = cittaItaliane.stream().allMatch((s) -> s.startsWith("A"));
System.out.print(matchedResult);
matchedResult = cittaItaliane.stream().noneMatch((s) -> s.startsWith("A"));
System.out.print(matchedResult);
```

true, false, false

Il metodo *anyMatch* è un metodo di tipo *short-circuit*: non appena un elemento dello stream corrente soddisfa il predicato argomento del metodo, l'esecuzione delle operazioni verrà interrotta immediatamente. Come anyMatch anche il prossimo metodo è di tipo *short-circuit*.

Il metodo findFirst

Il metodo findFirst ritorna un *Optional* che descrive il primo elemento dello stream ed interrompe immediatamente il flusso delle operazioni. Ritorna un *Optional* vuoto altrimenti.

```
String primaCittaTrovata = cittaItaliane.stream().filter(s -> s.startsWith("A")).findFirst()
.orElse("sconosciuta");
```

Il metodo count

Il metodo *count* è un'operazione terminale che restituisce il numero di elementi nello stream corrente come valore **long**.

```
long numeroDiElementi = cittaItaliane.stream().count();
```

Il metodo reduce

Il metodo *reduce* esegue una riduzione sugli elementi dello stream corrente con la funzione passata come argomento. Il risultato è un tipo Optional che descrive il valore ridotto.

Nell'esempio riduciamo tutte le stringhe concatenandole usando il separatore #.

```
Optional<String> riduzione = cittaItaliane.stream()
.reduce((s1, s2) -> s1 + "#" + s2);
riduzione.ifPresent(System.out::println);
```

```
Ancona#Milano#Roma#Padova#Napoli#Rovigo#Ravenna#Varese#Lecce#Bari
```

Esecuzione parallela

Grazie agli stream, per sfruttare il calcolo parallelo tutto ciò che dobbiamo fare è creare uno *stream parallelo* invece di uno *stream sequenziale.* In definitiva, per parallelizzare ognuno degli esempi sopra citati basterà sostituire la chiamata *stream()* con la chiamata *parallelStream()*.

Sebbene la piattaforma Java fornisce già supporto per il calcolo parallelo e la concorrenza (di cui parleremo ampiamente nei prossimi capitoli), uno dei fattori chiave delle stream API è la possibilità di rendere il calcolo parallelo più facilmente accessibile agli sviluppatori.

Uno dei problemi maggiori nel passaggio da un algoritmo sequenziale ad uno parallelo è rappresentato proprio dalla complessità intrinseca nel passaggio da un codice che affronta i problemi nel modo classico, sequenzialmente, ad un codice che sfrutta la concorrenza dei thread per sfruttare le moderne architetture multi-core.

Per questo motivo, gli stream nascono anche per incoraggiare un idioma che consenta il passaggio da sequenziale a parallelo in maniera semplice spostando l'attenzione del programmatore su quello che fa fatto, e non sul come farlo.

Stream con tipi primitivi

Gli stream sono studiati per funzionare principalmente con raccolte di oggetti, e non con tipi primitivi. Fortunatamente,la libreria standard include tre implementazioni specializzate per fornire un modo per lavorare con i tre tipi primitivi più utilizzati: int, long e double: *IntStream*, *LongStream* e *DoubleStream*.

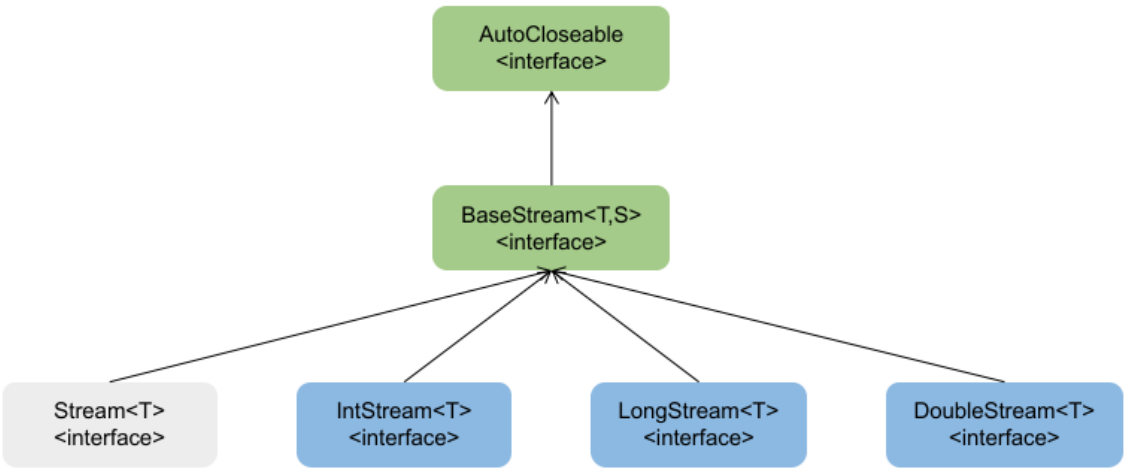


Immagine 53 Stream di tipi primitivi

Java non fornisce supporto per altri tipi primitivi, ma si limita a supportare i tipi più utilizzati in grado di coprire la maggior parte dei casi d'uso. Non è stato ritenuto utile supportare altri tipi.

Il limite principale degli stream di tipi primitivi è quello relativo alla necessità di trasformare i tipi primitivi nei corrispettivi oggetti wrapper quando di ha necessità di trasformare le collezioni.

Creazione di stream con tipi primitivi

Esistono diversi modi per creare stream di tipi primitivi. L'interfaccia Stream ad esempio dispone di tre operazioni intermedie che modificano lo stream corrente e restituiscono uno dei tre tipi *IntStream*, *LongStream* e *DoubleStream*.

Operazioni intermedie di Stream che generano stream di tipi primitivi
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)
Restituisce un DoubleStream costituito dai risultati della sostituzione di ogni elemento del flusso corrente con i contenuti del DoubleStream mappato prodotto applicando la funzione di mappatura fornita a ogni elemento.
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)
Restituisce un DoubleStream costituito dai risultati della sostituzione di ogni elemento del flusso corrente con i contenuti dello stream mappato prodotto applicando la funzione di mappatura fornita a ogni elemento.

LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)
Restituisce un LongStream costituito dai risultati della sostituzione di ogni elemento del flusso corrente con i contenuti dello stream mappato prodotto applicando la funzione di mappatura fornita a ogni elemento.
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
Restituisce un DoubleStream costituito dai risultati dell'applicazione della funzione mapper agli elementi di questo flusso.
DoubleStream mapToInt(ToIntFunction<? super T> mapper)
Restituisce un DoubleStream costituito dai risultati dell'applicazione della funzione mapper agli elementi di questo flusso.
LongStream mapToLong(ToLongFunction<? super T> mapper)
Restituisce un DoubleStream costituito dai risultati dell'applicazione della funzione mapper agli elementi di questo flusso.

Creare stream da elenchi noti

Se dobbiamo creare stream a partire da un elenco di interi o double, possaimo utilizzare il metodo *of*:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5);
LongStream stream = LongStream.of(1, 2, 3, 4, 5);
DoubleStream stream = DoubleStream.of(1.0, 2.0, 3.0, 4.0, 5.0);
```

Il metodo factory: range

Il metodo factory, range, restituisce un *IntStream* oppure *LongStream* che rappresentano stream di sequenze di numeri interi generati automaticamente a partire da *startInclusive* (incluso) fino a *endExclusive* (escluso) con passo 1:

```
IntStream stream = IntStream.range(1, 10); //1,2,3,4,5,6,7,8,9
LongStream stream = LongStream.range(10, 100);
```

Simile a range è il metodo *rangeClosed* che restituisce lo stesso stream includendo però l'ultimo numero:

```
IntStream stream = IntStream.rangeClosed(1, 10); //1,2,3,4,5,6,7,8,9,10
```

Arrays.stream()

Possiamo chiamare il metodo statico *Arrays.stream()* direttamente su un array. La chiamata restituirà un'istanza della classe stream corrispondente al tipo di array. Ad esempio, se chiamiamo *Arrays.stream()* su un array *int[]*, restituirà un'istanza di *IntStream*.

```
// int[] -> Stream
int[] array = new int[]{1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(array);

// long[] -> Stream
long[] array = new long[]{1, 2, 3, 4, 5};
LongStream stream = Arrays.stream(array);

// double[] -> Stream
double[] array = new double[]{1.0, 2.0, 3.0, 4.0, 5.0};
DoubleStream stream = Arrays.stream(array);
```

Operazioni sugli stream primitivi

Tutte e tre le classi, *IntStream, LongStream e DoubleStream*, sono costituite da valori numerici, ha quindi senso fornire direttamente supporto a tutte le principali operazioni per lavorare nello specifico con tipi primitivi.

Operazioni aritmetiche

Le operazioni aritmetiche di somma, media, massimo e minino sono supportate da tutte e tre le classi *IntStream, LongStream e DoubleStream,* e sono le seguenti: *sum(), average(), max() e min()*. Insieme a *count()* che restituisce il numero di elementi nello stream corrente, rappresentano operazioni terminali.

I metodi a seguire provengono dalla classe *IntStream.*

Metodi aritmetici provenienti dalla classe IntStream
int sum() Restituisce la somma degli interi all'interno dello stream.
OptionalDouble average() Restituisce un OptionalDouble che descrive la media aritmetica degli elementi dello stream corrente. Nel caso di stream vuoto restituisce un OptionalDouble vuoto.
OptionalInt max() Restituisce un OptionalInt che descrive l'elemento massimo stream corrente. Nel caso di stream vuoto restituisce un OptionalInt vuoto.

OptionalInt min()
Restituisce un OptionalInt che descrive l'elemento minimo stream corrente. Nel caso di stream vuoto restituisce un OptionalInt vuoto.
int count()
Restituisce il numero degli elementi nello stream.

Nel prossimo esempio viene costruito lo stream utilizzando uno dei metodi analizzato in precedenza, viene utilizzato il metodo max per estrarre un *OptionalInt* che viene convertito in un tipo **int** tramite la chiamata al metodo *getAsInt:*

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
IntStream stream = Arrays.stream(array);
System.out.println(stream.max().getAsInt());
```

Analogamente, possiamo scrivere:

```
System.out.println(stream.min().getAsInt());
System.out.println(stream.average().getAsDouble());
System.out.println(stream.count());
System.out.println(stream.sum());
```

Il metodo summaryStatistics

Consideriamo adesso il prossimi esempio:

```
public static void main(String[] args) {
        int[] array = new int[] { 1, 2, 3, 4, 5 };
        IntStream stream = Arrays.stream(array);

        System.out.println(stream.max().getAsInt());
        //la prossima riga di codice genera una eccezione
        System.out.println(stream.min().getAsInt());
        System.out.println(stream.average().getAsDouble());
        System.out.println(stream.count());
        System.out.println(stream.sum());
}
```

Il codice viene compilato correttamente, ma non appena viene eseguita la chiamata al metodo *min*, la applicazione Java va in eccezione. L'output della applicazione infatti è il seguente:

```
5
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.base/java.util.stream.AbstractPipeline.evaluate(Unknown Source)
```

```
at java.base/java.util.stream.IntPipeline.reduce(Unknown Source)
at java.base/java.util.stream.IntPipeline.min(Unknown Source)
at javamattone.esercizi.stream.primitives.Aritmeticoperations.main(Aritmeticoperations.java:13)
```

Il motivo è che uno stream deve essere utilizzato (richiamando un'operazione di flusso intermedio o terminale) solo una volta. L'unica alternava apparentemente sembrerebbe quella di creare lo stream ogni volta che dobbiamo accedere ai dati riepilogativi riscrivendo il metodo *main* come segue:

```
public static void main(String[] args) {
        int[] array = new int[] { 1, 2, 3, 4, 5 };
        IntStream stream = Arrays.stream(array);

        System.out.println(stream.max().getAsInt());
        stream = Arrays.stream(array);
        System.out.println(stream.min().getAsInt());
        stream = Arrays.stream(array);
        System.out.println(stream.average().getAsDouble());
        stream = Arrays.stream(array);
        System.out.println(stream.count());
        stream = Arrays.stream(array);
        System.out.println(stream.sum());
}
```

Ma questo è contraddittorio rispetto all'idea degli stream di mantenere il codice conciso e compatto. Per risolvere il problema, un altro modo per trovare i dati riepilogativi visti nel paragrafo precedente è utilizzare il metodo *summaryStatistics()* che restituisce una delle seguenti classi: *IntSummaryStatistics, LongSummaryStatistics e DoubleSummaryStatistics* rispettivamente per *Intstrem*, *LongStream*, *DoubleStrem*.

L'esempio precedente può essere riscritto utilizzando di *summaryStatistics()* come fatto nel prossimo esempio:

```
public static void main(String[] args) {
        int[] array = new int[] { 1, 2, 3, 4, 5 };
        IntStream stream = Arrays.stream(array);
        IntSummaryStatistics summaryStatistics = stream.summaryStatistics();

        System.out.println(summaryStatistics.getMax());
        System.out.println(summaryStatistics.getMin());
        System.out.println(summaryStatistics.getCount());
        System.out.println(summaryStatistics.getAverage());
        System.out.println(summaryStatistics.getSum());
}
```

Boxing e Unboxing

Ci sono volte in cui abbiamo bisogno di convertire i valori primitivi nei loro equivalenti wrapped. In questi casi, possiamo usare il metodo *boxed*:

```
public static void main(String[] args) {
List<Integer> tipiWrapped = IntStream.rangeClosed(1, 10)
        .filter(i -> i % 2 == 0)
        .boxed()
        .collect(Collectors.toList());
}
```

Gestire “Stream has already been operated upon or closed” Exception in Java

Per concludere questa sezione, analizziamo velocemente l’eccezione *IllegalStateException*, già incontrata nei paragrafi preceenti, molto comune quando si lavora con gli stream.

Partiamo dal prossimo esempio:

```
public static void main(String[] args) {
        Stream<String> stringStream = Stream.of("A", "B", "C", "D");
        Optional<String> result1 = stringStream.findAny();
        System.out.println(result1.get());
        Optional<String> result2 = stringStream.findFirst();
}
```

Non appena verrà eseguito il metodo terminale *findAny*, lo stream verrà chiuso e qualsiasi altro accesso ad esse provocherà un eccezzione *IllegalStateException.*

Il motivo è che In Java 8, ogni classe *Stream* rappresenta una sequenza di dati monouso e supporta diverse operazioni di I/O. Di conseguenza, ogni volta che viene invocata una operazione terminale (*lazy evaluation*) lo stream viene consumato e chiuso.

L’unica soluzion, data la natura stessa degli stream. è quella di ricreare lo stream ogni volta che dobbiamo fare uso. In alternativa viene in aiuto l’interfaccia funzionale *Supplier* che ci consente di riscrivere l’esempio precedente come segue:

```
public static void main(String[] args) {
        Supplier<Stream<String>> streamSupplier = () -> Stream.of("A", "B", "C", "D");
        Optional<String> result1 = streamSupplier.get().findAny();
        System.out.println(result1.get());
        Optional<String> result2 = streamSupplier.get().findFirst();
        System.out.println(result2.get());
}
```

In questo modo, invocando il metodo funzionale get() sul *Supplier*, ci viene restituito un oggetto *Stream* appena creato, sul quale possiamo tranquillamente eseguire un'altra operazione.

19.La monade Optional

Introduzione

Supponiamo di avere diverse funzioni che gestiscono un numero intero sia come parametro d'ingresso che di uscita. Supponiamo anche che alcune di queste siano *funzioni parziali* vale a dire che per alcuni valori di input non definiscono un corrispondente valore di output, ma restituiscono *null; o*gni volta che combiniamo due funzioni, dobbiamo pertanto gestire il caso in cui il valore sia *null.*

Ciò che facciamo solitamente è inserire molti **if** nella sequenza di funzioni, magari annidati tra loro.

La monade *Optional* ha proprio lo scopo di gestire la presenza o l'assenza di un valore. Trasformando le nostre funzioni di un qualsiasi tipo in funzioni che accettano e restituiscono un tipo *Optional*, e così facendo possiamo combinare le funzioni disinteressandoci del caso null, perché sarà la monade stessa a prendersi in carico la responsabilità di gestirlo.

Potendo ignorare il caso **null** il risultato sarà una sequenza semplice e lineare di combinazioni di funzioni, facile da leggere e comprendere. I problemi connessi al dato ed alla sua gestione è demandata ad *Optional*.

Optional definisce una serie di metodi che consentono di operare sui dati mediante pipelining.

Per motivi evidenti, dal momento che questa classe di oggetti gestisce dati di tipo disparato, hanno molto a che vedere con i tipi generici cosa che in effetti vedremo nei prossimi paragrafi.

Il tipo Optional

Un tipo generico *Optional<T>* è un wrapper per entrambi: un oggetto di tipo T oppure un oggetto nullo. Nel primo caso diremo che il valore è *presente (present)*, nel secondo diremo che il contenitore è *vuoto (empty)*.

Si presti attenzione al fatto che un oggetto Optional empty non è equivalente ad un oggetto nullo.

Optional deve esser inteso come una alternativa valida al semplice uso di una reference di tipo T che potrebbe essere nulla costringendoci ad implementare blocchi di codice che potrebbero tranquillamente essere demandati ad *Optional*.

E' ottima pratica utilizzare *Optional* come tipo ritornato. Un metodo che torna un tipo *Optional* sta ad indicare che quel metodo potrebbe ritornare un valore nullo.

Optional fa parte del package *java.util* è mette a disposizione diversi metodi: ci sono metodi che servono per creare oggetti Optional, metodi che ci consentono di operare con il valore rappresentato senza necessariamente doverlo rimuovere dal contesto definito dalla monade, metodi per il pipelining delle operazioni per la trasformazione dl valore.

Esistono specializzazioni della classe utilizzate per rappresentare alcuni tipi primitivi, quali: *OptionalDouble*, *OptionalInt* e *OptionalLong* che consentono di estrarre il valore convertendolo nel corrispondente tipo primitivo; tuttavia poiché questi casi specifici non aggiungono nulla alla descrizione generale per questo tipo, rimando alla documentazione ufficiale l'onere del dettaglio.

Nella prossima tabella sono elencati i metodi messi a disposizione dal tipo *java.util.Optional:*

Metodi costruttori di *java.util.Optional*
static <T> Optional<T> empty()
Ritorna una istanza vuota (empty) di optional.
Optional<T>filter(Predicate<?super T> predicate)
Se il valore è presente e il valore corrisponde al predicato ritorna un Optional che descrive il valore, un Optional vuoto altrimenti.
<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
Se un valore è presente applica la funzione al valore e ritorna un Optional con il valore trasformato. Un Optional vuoto altrimenti.
T get()
Se un valore è presente, ritorna il valore NoSuchElementException altrimenti.
void ifPresent(Consumer<? super T> consumer)
Se un valore è presente applica il Consumer con il valore, altrimenti non fa nulla.
boolean isPresent()
Ritorna true se il valore è presente, false altrimenti.
<U> Optional<U> map(Function<? super T,? extends U> mapper)
Se un valore è presente applica la funzione di mappatura al valore e ritorna un Optional con il valore trasformato. Un Optional vuoto altrimenti.
static <T> Optional<T> of(T value)
Ritorna un tipo Optional present con il valore non nullo specificato. Produce NullPointerException se value è nullo.
static <T> Optional<T> ofNullable(T value)
Ritorna un Oprional che descrive il valore: empty se il valore è nullo, present altrimenti.
T orElse(T other)
Ritorna il valore se presente, other altrimenti.
T orElseGet(Supplier<? extends T> other)
Ritorna il valore se presente, altrimenti invoca other e ritorna il valore prodotto dalla funzione.

orElseThrow(Supplier<? extends X> exceptionSupplier) throws T <X extends Throwable>
Ritorna il valore se presente, altrimenti propaga una eccezione creata da exceptionSupplier
Metodi inseriti a partire da Java 9
Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)
Se esiste un valore allora torna un Optional che descrive il valore, altrimenti un Optional prodotto dalla funzione supplier.
void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
Se il valore è presente esegue l'azione action altrimenti emptyAction.
Stream<T> stream()
Se esiste un valore ritorna uno stream contenente solo il valore. Uno stream vuoto altrimenti.
Metodi inseriti a partire da Java 10
orElseThrow() throws NoSuchelementException
Presente a partire da Java 10, se il valore è nullo torna una eccezione di tipo NoSuchElementException
Metodi inseriti a partire da Java 11
boolean isempty()
Presente solo a partire da Java 11. Ritorna true se Optional è vuoto, false altrimenti.

Creare un oggetto Optional

Ci sono diversi modi di creare un tipo Optional. Partiamo pertanto dal caso più semplice: un optional vuoto.

Per creare un oggetto optional vuoto ci basterà invocare il metodo statico *empty()*:

```
Optional<String> empty = Optional.empty();
```

In questo caso, poiché abbiamo creato un Optional vuoto, il metodo *empty.isPresent()* tornerà false. Se vogliamo creare un oggetto *Optional* con un valore non nullo possiamo utilizzare il metodo:

```
Optional<String> present= Optional.of("stringa non nulla");
```

Tuttavia, se il valore in questo caso è nullo *of* produrrà una eccezione del tipo *NullPointerException*. Qualora non siamo sicuri che il valore non sia nullo, possiamo utilizzare il metodo *ofNullable*:

```
Optional<String> optional= Optional.ofNullable(null);
Optional<String> optional= Optional.ofNullable(“stringa non nulla”);
```

A seguire un esempio:

```
Optional<String> empty = Optional.ofNullable(null);
Optional<String> present = Optional.ofNullable("Stringa non vuota");

System.out.println("empty è vuoto: "+empty.isEmpty());
System.out.println("present è vuoto: "+present.isEmpty());
```

Dall’esecuzione vediamo subito che, nel primo caso otteniamo un Optional vuoto, mentre nel secondo caso Optional conterrà il valore *“stringa non nulla”*.

```
empty è vuoto: true
present è vuoto: false
```

Controllare la presenza di un valore

Come mostrato nell’esempio precedente, *Optional* mette a disposizione due metodi per controllare se un valore è presente: *isPresent()*, e a partire da Java 11 *isEmpty()* che restituiscono true rispettivamente se: Optional contiene un valore, Optional è vuoto.

Per completare l’esempio:

```
Optional<String> empty = Optional.ofNullable(null);
Optional<String> present = Optional.ofNullable("Stringa non vuota");

System.out.println("empty è vuoto: "+empty.isEmpty());
System.out.println("present è vuoto: "+present.isEmpty());

System.out.println("empty contiene un valore: "+empty.isPresent());
System.out.println("present contiene un valore: "+present.isPresent());
```

Se eseguito produrrà il seguente output a terminale:

```
empty è vuoto: true
present è vuoto: false

empty contiene un valore: false
present contiene un valore: true
```

Utilizzare i valori Optional

Il metodo più diretto per ottenere un valore da *Optional* è il metodo *get* che ritorna una eccezione di tipo NoSuchElementException nel caso in cui *Optional* sia vuoto come ne prossimo esempio:

```
Optional<String> empty = Optional.ofNullable(null);
Optional<String> present = Optional.ofNullable("java mattone dopo mattone");

String stringFromPresent = present.get();
System.out.println("La lunghezza di stringFromPresent è: "+stringFromPresent.length());

String stringFromEmpty = empty.get();
System.out.println("La lunghezza di stringFromEmpty è: "+stringFromEmpty.length());
```

Se eseguito produrrà in output

```
La lunghezza di stringFromPresent è: 25
Exception in thread "main" java.util.NoSuchElementException: No value present
    at java.base/java.util.Optional.get(Unknown Source)
    at javamattone.esercizi.optional.OttenereValori.main(OttenereValori.java:13)
```

Poiché *NoSuchElementException*, alla pari di *NullPointerException*, è una eccezione *unchecked*, le due sono paragonabili. Dal momento che lo scopo di Optional è quello di evitare di doversi ritrovare a gestire casi di questo tipo in futuro il metodo *get* sarà probabilmente deprecato.

Per il momento il consiglio è utilizzare gli altri metodi messi a disposizione da *Optional* che consentono di gestire anticipatamente il caso del valore nullo.

Il metodi orElse ed orElseGet

Un primo approccio è quello del valore di *default*: qualora *Optional* sia vuoto, potremmo decidere di ritornare un valore di default utilizzando il metodo *orElse*:

```
Optional<String> optional = Optional.ofNullable(null);
String nome = optional.orElse("Mario Rossi");
```

Da notare che *erElse* può utilizzare consente l'oggetto **null** come tipo di default quindi, nonstante sia sconsigliato, il prossimo frammento di codice è assolutamente consentito:

```
String nome = optional.orElse(null);
```

E molto simile al metodo precedente con la differenza che il valore ritornato è prodotto da un *Supplier* preso come argomento. anche in questo caso può tornare **null**.

```
nome = optional.orElseGet(()->"Mario rossi");
```

```
nome = optional.orElseGet(()->null);
```

Nonostante possa sembra che il primo metodo, *orElse*, sia ridondante rispetto ad *orElseGet*, esiste una differenza tanto sostanziale quanto subdola tra i due metodi. Cerchiamo di capire quale e per farlo creiamo un metodo *getDefault()* che torna una stringa che rappresenta, guarda caso , il nome di default. Quando viene chiamato il metodo lascia traccia di se stampando a video la stringa: *"ritorno il valore di default"*.

```
public String getDefault(){
    System.out.println("ritorno il valore di default");
    return "Mario Rossi";
  }
```

Nell'esempio utilizziamo i due metodi *orElse* ed *orElseGet* a partire da due Optional vuoti:

```
 public void eseguiTest(){

    String nomediDefault = Optional.ofNullable(null).orElseGet(this::getDefault);
    System.out.println("Valore di default: "+ nomediDefault);

    nomediDefault = Optional.ofNullable(null).orElse(getDefault());
    System.out.println("Valore di default:"+ nomediDefault);

  }

  public static void main(String[] args) {
    OrElseGetvsOrElse orElseGetvsOrElse = new OrElseGetvsOrElse();
    orElseGetvsOrElse.eseguiTest();
  }
```

Eseguendo il metodo *main* quello che otteniamo è esattamente quello che ci attendiamo:

```
ritorno il valore di default
Valore di default: Mario Rossi

ritorno il valore di default
Valore di default:Mario Rossi
```

in entrambi i casi il metodo viene chiamato il metodo *getDefault* che, come aspettato, ha segnalato la sua chiamata.

Modifichiamo adesso l'esempio facendo in modo che i metodi *orElse* e *orElseGet* siano eseguiti sue due *Optional* che, questa volta, contengono un valore non nullo.

```
public void eseguiTest() {

    String text = "Giuseppe Marascia";
    System.out.println("Utilizzo orElseGet:");
    String nomediDefault = Optional.ofNullable(text).orElseGet(this::getDefault);
    System.out.println("Valore di default:" + nomediDefault);
    System.out.println("Using orElse:");
    nomediDefault = Optional.ofNullable(text).orElse(getDefault());
    System.out.println("Valore di default:" + nomediDefault);

  }
```

Basta eseguire il codice per notare subito uno strano *side-effect*:

```
Utilizzo orElseGet:
Valore di default:Giuseppe Marascia

Using orElse:
ritorno il valore di default
Valore di default:Giuseppe Marascia
```

Nonostante stiamo invocando *orElse* su un tipo Optional presente, comunque viene invocato il metodo *getDefault*. Alla fine comunque il risultato è quello atteso. Nonostante la chiamata sia a costo zero, cosa succederebbe però se il metodo contenesse, ad esempio, una chiamata ad un servizio REST?

Il metodo orElseThrow

Il metodo *orElseThrow* cambia completamente strategia, e invece di tornare un valore di default ritorna una eccezione personalizzabile. Nel prossimo esempio creiamo una eccezione personalizzata di tipo *checked:*

```
public class EccezionePersonalizzata extends Exception{
}
```

Poiché l'eccezione è di tipo *checked*, questa volta quanto meno saremo costretti a gestirla in maniera esplicita inserendo la chiamata in un blocco di guardia:

```
public static void main(String[] args) {
    String nullName = null;
    try {
      String name = Optional.ofNullable(nullName).orElseThrow(
        EccezionePersonalizzata::new);
    } catch (EccezionePersonalizzata e) {
      System.out.println("Si è verificato un errore");
    }
  }
```

Il metodo funziona allo stesso modo con le eccezioni di tipo *unchecked*. A partire da Java 10 ne esiste una versione che non accetta attributi, e che nel casi di *Optional vuoto* torna una eccezione *unchecked* di tipo *NoSuchElementException*.

Il metodo or

Introdotto a partire da Java 9, può essere utilizzato in tutte quelle situazioni in cui vogliamo eseguire qualcosa se *Optional* è vuoto. A differenza dei metodi precedenti che tornano tutti un valore non *Optional*, *or* consente di tornare un altro *Optional*.

Vediamo un esempio:

```
Optional<String> defaultValue = Optional.of("Mario Rossi");
Optional<String> optional = Optional.of("Massimiliano Tarquini");
Optional<String> optionalEmpty = Optional.empty();

Optional<String> optional2 = optional.or(()->defaultValue);
Optional<String> optional3 = optionalEmpty.or(()->defaultValue);
System.out.println(optional2.get());
System.out.println(optional3.get());
```

L'output della applicazione è il seguente:

```
Massimiliano Tarquini
Mario Rossi
```

In definitiva, se *Optional* contiene un valore il metodo *or* torneà lo stesso Optional, altrimenti un nuovo come prodotto della esecuzione del *Supplier* passato per argomento.

Il metodo *or*, come il metodo *orElseGet* ha una esecuzione di tipo *lazy*: se *Optional* già contiene un valore allora l'espressione lambda passata al metodo *or* non sarà mai eseguita.

Il metodo ifPresentOrElse

Capita spesso di avere un *Optional,* di voler effettuare alcune operazioni sul valore contenuto, e nel caso *Optional* sia vuoto, registrare l'evento magari aggiornando alcune metriche o log. Questo è il caso di questo metodo: se *Optional* contiene un valore allora sarà eseguito il *Consumer* action, altrimenti un tipo *Runnable* come mostrato nel prossimo frammento di codice:

```
Optional<String> value = Optional.empty();
AtomicInteger successCounter = new AtomicInteger(0);
AtomicInteger onEmptyOptionalCounter = new AtomicInteger(0);
```

```
value.ifPresentOrElse(
   v -> successCounter.incrementAndGet(),
   onEmptyOptionalCounter::incrementAndGet);
   System.out.println("successCounter vale: "+successCounter);
   System.out.println("onEmptyOptionalCounter vale: "+onEmptyOptionalCounter);
}
```

La cui esecuzione restituirà:

```
successCounter vale: 0
onEmptyOptionalCounter vale: 1
```

Il metodo condizionale filter

E' l'ultimo dei metodi che possono essere utilizzati per estrarre il valore da *Optional* con la differenza che torna un altro *Optional* e quindi può essere utilizzato come *operazione intermedia* in una pipeline.

Analogamente al metodo *filter* degli *Stream*, questo metodo viene utilizzato per filtrare il valore di *Optional* sulla base di regole. Di fatto, se il predicato preso come attributo è verificato, allora torna *Optional* corrente, altrimenti tornerà un *Optional* vuoto.

```
public static boolean patternMatches(String emailAddress) {
   return Pattern.compile("^(.+)@(\\S+)$")
        .matcher(emailAddress)
        .matches();
}

public static void main(String[] args) {
   Optional<String> email = Optional.of("massimiliano.tarquini@gmail.com");
   Optional<String> risultato = email.filter(FilterMethod::patternMatches);
   System.out.println("il valore è una email valida: "+risultato.isPresent());
   email = Optional.of("massimiliano.tarquinigmail.com");
   risultato = email.filter(FilterMethod::patternMatches);
   System.out.println("il valore è una email valida: "+risultato.isPresent());
}
```

Nell'esempio, il metodo statico *patternMatches* utilizza una espressione regolare per verificare che la mia sia scritta in un formato valido.

```
il valore è una email valida: true
il valore è una email valida: false
```

Nel secondo caso viene tornato un *Optional* vuoto in quanto la mail non contiene il carattere '@'.

Questo metodo in genere viene utilizzato per condizionare l'estrazione del valore di un *Optional* a determinate regole: ad esempio la stringa contenuta deve rappresentare un codice fiscale valido.

Trasformare i valori di Optional

Nella sezione precedente abbiamo visto come recuperare il valore rappresentato da un Optional. Occupiamoci adesso di come trasformarlo.

Il metodo map

Il metodo *map* applica la funzione di mappatura al valore rappresentato, e ritorna un Optional con il valore trasformato. Per dimostrarne il funzionamento creiamo la classe POJO studente come segue:

```
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Data
public class Studente{
  private String nome;
  private String cognome;
  private Integer eta;
}
```

E' una classe che rappresenta uno studente ed è realizzata utilizzando il preprocessore di codice Lombok. Supponiamo di voler verificare che l'età dello studente sia correttamente inizializzata.

Normalmente il nostro codice sarebbe qualcosa del tipo:

```
Optional<Studente> studente = Optional
        .of(Studente.builder()
        .nome("Massimiliano")
        .cognome("Tarquini")
        .eta(20).build());

if(studente.map(valore -> valore.getEta()).isEmpty()){
  System.out.println("Il campo età non è inizializzato");
} else{
  System.out.println("Il campo età è stato correttamente inizializzato");
}
```

Applicando il metodo *map* abbiamo trasformato *Optional<Student>* in un *Optional<Integer>* che dovrebbe rappresentare l'età dello studente. Nel caso in cui l'età dello studente non fosse stata inizializzata avrebbe tornato un *Optional* vuoto.

Da notare che, dal momento in cui eseguiamo il metodo map, le operazioni successive dovranno essere tutte compatibili con il nuovo *Optional<T>* prodotto.

Il metodo flatMap

Simile al metodo precedente è il metodo *flatMap* con l'unica differenza che, il primo trasforma valori di un tipo qualsiasi, il secondo prende solo valori di tipo Optional<U>.

Di fatto, capita spesso di dover gestire situazioni in cui un metodo ritorna il tipo *Optional* invece di un tipo normale come accade nella prossima versione della classe *Studente.*

La definizione è la seguente:

```
@AllArgsConstructor
public class StudenteConOptional {
  private String nome;
  private String cognome;
  private String sezione;
  private Integer eta;
  public Optional<String> getNome() {
    return Optional.of(nome);
  }
  public Optional<String> getCognome() {
    return Optional.of(cognome);
  }
  public Optional<Integer> getEta() {
    return Optional.of(eta);
  }
}
```

```
public static void main(String[] args) {
        Studente studente = new Studente("Massimiliano", "Tarquini", 20);
        Optional<Studente> studenteAsOptional = Optional.of(studente);
        Optional<String> nomeDelloStudente =
                studenteAsOptional.flatMap(Studente::getNome);
        String nome = nomeDelloStudente.orElse("");
        System.out.println(nome);

}
```

La differenza rispetto all'utilizzo di *Map* è proprio nel tipo ritornato. Il metodo *flatMap* ritorna un nuovo tipo Optional come risultato dell'esecuzione dell'espressione lambda presa come argomento. Sarà quindi necessario applicare ulteriori metodi su *Optional* per estrarre il nuovo contenuto.

Pipelining con il valore Optional

Come tutte le monadi, *Optional* dispone di una serie di metodi che tornano un tipo Optional, e di conseguenza possono essere utilizzati per creare una pipeline di operazioni che sono assimilabili alle *operazioni intermedie* degli stream. Se per esempio volessimo stampare il nome dello studente se inizia per M, ricordando la classe studente definita nei paragrafi precedenti sarebbe sufficiente scrivere:

```
Optional<Studente> studenteAsOptional = Optional
    .of(Studente.builder()
        .nome("Massimiliano")  .cognome("Tarquini") .eta(20).build());

studenteAsOptional.map(studente -> studente.getNome())
        .filter(nome -> nome.startsWith("M"))
        .ifPresent(System.out::println);
```

In cui *ifPresent* rappresenta l'operazione terminale.

In alternativa, a partire da Java 9, possiamo trasformare *Optional* in uno stream.

Il metodo stream

Optional sono paragonabili a stream vuoti o con un solo valore. Il metodo stream restituisce uno stream che sarà vuoto se *Optional* è vuoto, con un solo valore altrimenti. Attraverso il metodo *stream,* possiamo concatenare *Optional* con *Stream* alla perenne ricerca di un codice sempre più pulito ed elegante.

una volta ottenuto uno stream, possiamo operare sul valore utilizzando i metodi messi a disposizione dalle *Stream API di Java*.

Nel prossimo esempio concateniamo *Optional* con *Stream* per trasformare in una lista il valore ottenuto nell'esempio precedente:

```
Optional<Studente> studenteAsOptional = Optional
    .of(Studente.builder()
        .nome("Massimiliano")
        .cognome("Tarquini")
        .eta(20).build());

List<String> nomeComeLista = studenteAsOptional.map(studente -> studente.getNome())
    .filter(nome -> nome.startsWith("M")).stream().collect(Collectors.toList());
```

Cosa posso fare e cosa no con Optional?

Non potevano mancare consigli e suggerimenti su cosa conviene fare con *Optional* e cosa invece è meglio evitare. C'è da dire che il dibattito sull'utilità di optional è ancora aperto, e sicuramente, nel prossimo periodo la classe sarà soggetta a revisioni da parte della comunità.

E' comunque vero che se non utilizzato correttamente, il tipo *Optional* non porterà nessun reale vantaggio. Vediamo perché.

Non bisognerebbe mai utilizzare il metodo *get* per estrarre il valore da un *Optional.* Meglio utilizzare i metodi alternativi messi a disposizione dalla classe.

Nel caso di un optional vuoto, il metodo *get* ad ritorna una eccezione *unchecked* di tipo *NoSuchElementException*. Di conseguenza scrivere:

```
Optional<T> valoreComeOptional= . . .;
valoreComeOptional.get().qualcheMetodo()
```

Non comporta vantaggi rispetto a scrivere:

```
T valore = . . .;
valore.qualcheMetodo();
```

Il metodo *isPresent* può essere utilizzato per verificare che il valore sia presente, ma ancora una volta scrivere:

```
Optional<T> valoreComeOptional= . . .;
if(valoreComeOptional.isPresent())
        valoreComeOptional.get().qualcheMetodo()
```

ancora una volta non è molto diverso da scrivere:

```
T valore = . . .;
if(valore != null) valore.qualcheMetodo();
```

Un oggetto *Optional* non dovrebbe mai essere nullo, non dovrebbe mai essere inserito in un insieme oppure utilizzato come chiave in una mappa.

Non passare mai un parametro di tipo Optional ad un metodo.

L'intento dei programmatori quando fu aggiunto *Optional* ero quello di utilizzarlo solo come metodo di ritorno di un metodo per indicare che quel metodo avrebbe potuto tornare anche un valore nullo.

Supponiamo di dover creare un metodo che ricerca su una lista di studenti filtrando per nome ed età:

```
public List<Studente> filtraStudenti(List<Studente> studenti, String nome, Optional<Integer> eta) {

  return studenti.stream().filter(studente -> studente.getNome().equals(nome))
      .filter(studente -> studente.getEta() == eta.orElse(-1)).collect(Collectors.toList());
}
```

Tutto fantastico fino a che il parametro eta non viene passato con valore **null**. Il metodo genererà una eccezione di tipo *NullPointerException* costringendoci a dover gestire il caso **null** cosa che stona se pensiamo che stiamo utilizzando *Optional* proprio per evitare di dover gestire il valori nulli. In questo caso sarebbe più coerente riscrivere il metodo nel modo seguente:

```
public List<Studente> filtraStudenti(List<Studente> studenti, String nome, Integer eta) {

  Integer filtroEta = eta == null ? 0 : eta;

  return studenti.stream().filter(studente -> studente.getNome().equals(nome))
      .filter(studente -> studente.getEta() == filtroEta).collect(Collectors.toList());
}
```

20.Java Threads

Introduzione

Java è un linguaggio multi-threaded, cosa che sta a significare che:

"un programma può essere eseguito logicamente
in molti luoghi nello stesso momento".

Un programma multi-thread contiene due o più parti che possono essere eseguite contemporaneamente, e ogni parte può gestire un'attività diversa allo stesso tempo facendo un uso ottimale delle risorse disponibili specialmente in presenza di CPU con molti core.

Il multithreading estende l'idea del multi-tasking alle applicazioni in cui è possibile suddividere operazioni specifiche all'interno di una singola applicazione in singoli thread (il multi-tasking è quando più processi condividono risorse di elaborazione comuni come una CPU). Ognuno dei thread può essere eseguito in parallelo: il sistema operativo è responsabile di dividere il tempo di elaborazione non solo tra diverse applicazioni (multi-tasking), ma anche tra ogni thread all'interno di un'applicazione (multi-threading) .

Dal punto di vista dell'utente, i thread logici appaiono come una serie di processi che eseguono parallelamente le loro funzioni. Dal punto di vista della applicazione rappresentano processi logici che, da una parte condividono la stessa memoria della applicazione che li ha creati, dall'altra concorrono con il processo principale al meccanismo di assegnazione del processore su cui l'applicazione è in esecuzione.

Obiettivo di questo capitolo è descrivere come programmare applicazioni multi-threaded utilizzando il linguaggio Java. Prima di procedere, è importante capire a fondo il significato di programmazione concorrente, come scrivere applicazioni utilizzando questa tecnica, ed infine la differenza tra thread e processi.

Thread e Processi

"Cara, accendi il rubinetto,

mentre

io cerco di stringere il tubo"

Nonostante possa sembrare bizzarro, quello che per noi può sembrare un comportamento banale, dal punto di vista di un calcolatore rappresenta un problema gravoso.

Eseguire contemporaneamente, due azioni logicamente correlate, comporta uno sforzo non indifferente dovendo gestire problematiche complesse quali:

1 . La condivisione dei dati tra le entità che dovranno concorrere a risolvere il problema;

2 . L'accesso concorrente alle risorse condivise del sistema (stampanti, supporti per i dati, periferiche, porte di comunicazione);

La capacità di un sistema operativo di eseguire più attività contemporaneamente è anche detta multi-tasking. Le tecniche di più comuni per progettare sistemi multi-tasking, fanno uso di *processi multipli* oppure *thread;* per cui parleremo rispettivamente di :

1. process based multi-tasking;

2. thread based multi-tasking ovvero multi-threading.

Il concetto di *processo* in informatica è associato, ma comunque distinto da quello di *thread* (abbreviazione di thread of execution) con cui si intende invece l'unità granulare in cui un processo può essere suddiviso (sotto processo), e che può essere eseguito a divisione di tempo o in parallelo ad altri thread da parte del processore. In altre parole, un *thread* è una parte del processo eseguita in concorrenza anche se dipendente dallo stato generale del processo stesso. Un processo ha sempre almeno un thread (se stesso), ma in alcuni casi può avere più thread che vengono eseguiti in parallelo.

Una differenza sostanziale fra thread e processi consiste nel modo con cui essi condividono le risorse: mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal sistema (come ad esempio socket o pipe), al contrario i thread di un processo condividono le medesime informazioni di stato, la memoria ed altre risorse assegnate al processo.

L'altra differenza sostanziale è insita nel meccanismo di attivazione: un processo è un oggetto monolitico la cui creazione è sempre onerosa per il sistema in quanto devono essere allocate ed assegnate le risorse necessarie alla sua esecuzione (caricamento dalla memoria persistente del programma, allocazione di memoria RAM, riferimenti alle periferiche, e così via, operazioni tipicamente onerose); il thread invece è parte di un processo e quindi una sua nuova attivazione viene effettuata in tempi ridottissimi a costi minimi.

Nella prossima immagine sono schematizzati i concetti appena introdotti:

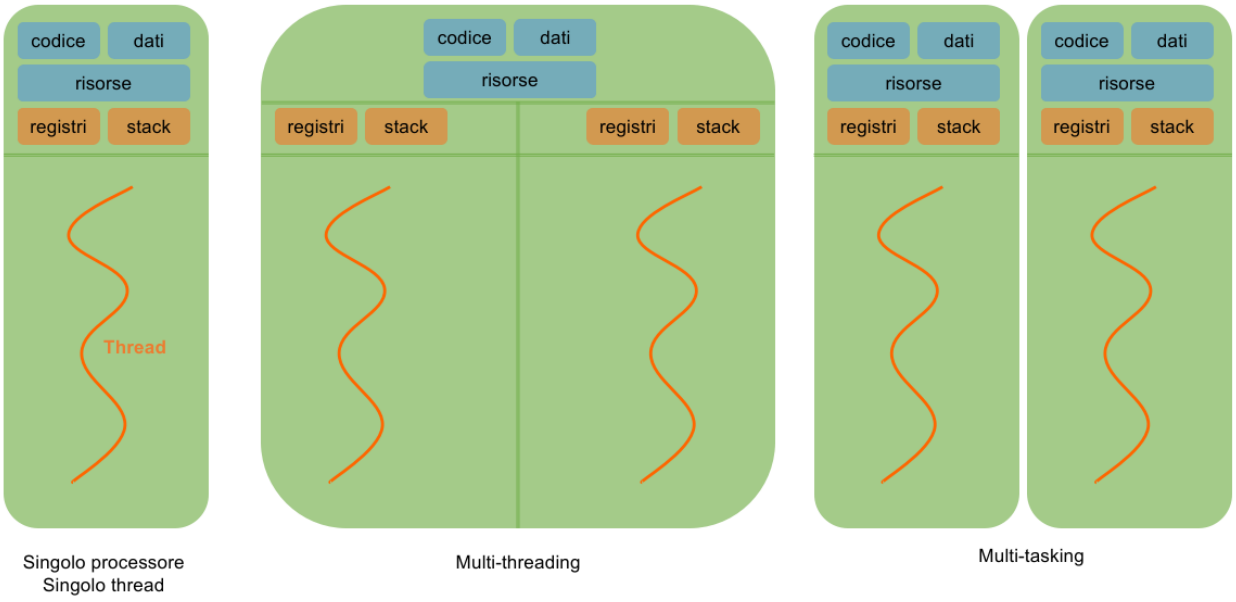


Immagine 54 Processi multi-treading e multi-tasking

Multi-tasking e gestione della concorrenza

Se dal punto di vista dell'utente un thread od un processo sono eseguiti parallelamente ad altri, dal punto di vista del calcolatore non esistono processi realmente paralleli: salvo che non possediate una macchina costosissima in grado di eseguire calcoli in parallelo, i processi si alternano tra loro con una velocità tale da dare l'impressione di essere eseguiti contemporaneamente. *Questo meccanismo è detto di concorrenza: i processi attivi concorrono tra loro all'uso del processore.*

Un algoritmo tipico utilizzato dai sistemi operativi per realizzare il meccanismo di concorrenza tra processi è quello chiamato *Round Robin*. Nella sua variante base, il sistema operativo organizza i processi attivi in una struttura dati detta *coda Round Robin* ed assegna ad ogni processo o thread in coda un intervallo di tempo, conosciuto anche come *quanto*. I quanti vengono assegnati ad ogni elemento in coda in maniera circolare ed in porzioni uguali senza priorità come schematizzato nella prossima figura:

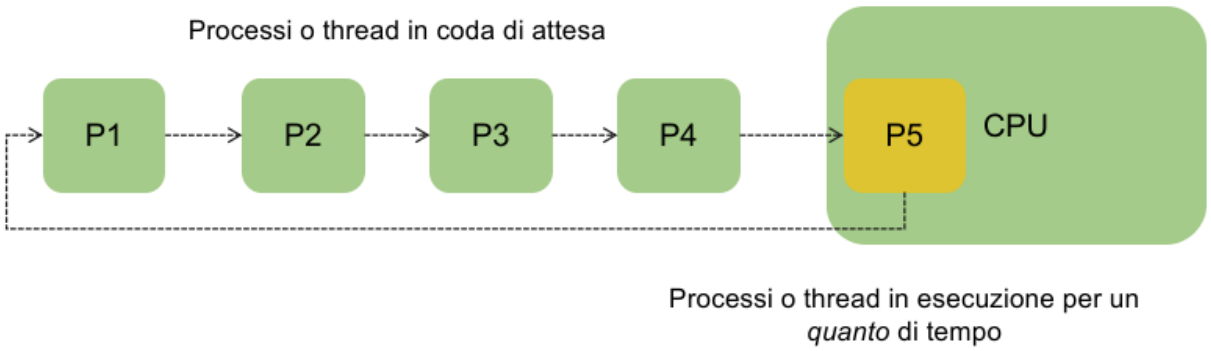


Immagine 55 - Scheduling round robin senza priorità

Ogni volta che il processore ha terminato il quanto dedicato ad un processo deve salvare le informazioni relative allo stato della esecuzione del processo corrente per poi riprenderlo, e ripristinare lo stato del processo successivo. La fase in cui due processi si alternano è detta *context switch* e rappresenta la fase più onerosa del meccanismo di alternanza dei processi.

Il sistema operativo è specializzato nella gestione della concorrenza tra processi e thread, ed è supportato da componenti elettroniche appositamente progettate per aumentare le prestazioni questi algoritmi e ridurre al minimo i tempi di latenza dovuti al *context switch* tra processi.

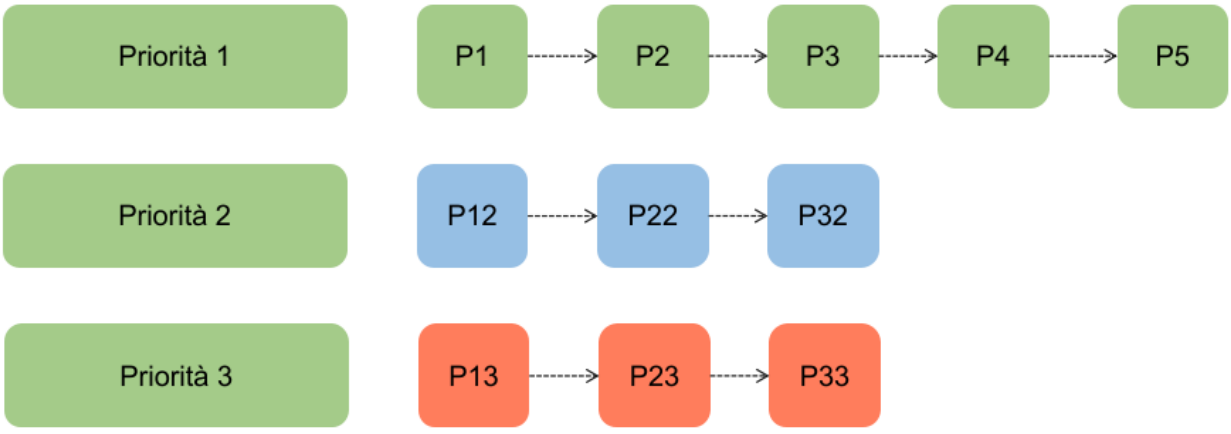


Immagine 56 Round robin con priorità

In realtà, i meccanismi di scheduling dei moderni sistemi operativi tengono conto anche della priorità dei processi: *round robin con code con priorità o preemption.* Secondo il modello basato su priorità, il quanto di tempo non è mai uguale per tutti i processi, ma dipende dalla priorità del processo stesso. Per organizzare la priorità dei processi vengono utilizzate code differenti ognuna con un livello id priorità come mostrato nella figura subito sopra.

Per concludere quindi, nella prossima tabella sono messi a confronto i due diversi approcci.

Process vs Thread based multitasking	
Process Based multitasking	**Thread Based Multitasking**
Due o più programmi sono eseguite in concorrenza (i.e. il browser, editor di testo, programma di mail).	Due o più parti dello stesso programma sono eseguite in concorrenza.
Ogni programma in esecuzione è chiamato *processo*.	Ogni parte indipendente della applicazione in esecuzione è chiamata *thread*.
Processo è la più piccola unità di codice che può essere gestita dallo scheduler del sistema operativo.	*Thread* è la più piccola unità di codice che può essere gestita dallo scheduler del sistema operativo.
Un *processo* ha a che fare con compiti complessi.	Un *thread* è responsabile di compiti semplici.
Richiedono un carico maggiore per la gestione dei quanti di tempo. Context switch è costoso.	Richiedono un carico piccolo. Context switch è poco costoso.
I *processi* sono considerati *heavy weighted*.	I *thread* sono anche detti *light weighted*.
Ogni processo richiede la allocazione di una propria zona di memoria o *address space*.	I *thread* condividono l'*address space* del processo padre.
La comunicazione inter-processo è costosa e limitata alle poche modalità fornite dal sistema operativo come socket o pipes.	La comunicazione inter-thread è semplice e gratuita.
Non è sotto il controllo di Java (ma del sistema operativo).	Sono sotto il controllo di Java e della JVM.

Vantaggi e svantaggi nell'uso di thread

La scelta tra la prima e la seconda soluzione proposte nel paragrafo precedente, non è così scontata come può sembrare. Di fatto, i thread non rappresentano sempre la tattica migliore nell'approccio ad un problema di programmazione in concorrenza. In questo paragrafo, cercheremo di comprendere quali sono i vantaggi e gli svantaggi che l'uso di thread comporta rispetto a quello di processi multipli.

1. I thread sono molto efficienti nell'utilizzo della memoria.

L'uso di processi multipli, necessita di una quantità di memoria maggiore rispetto a quella richiesta dai thread. Di fatto, ogni processo per essere eseguito ha bisogno di una propria area dati ed istruzioni. I thread, differentemente dai processi, utilizzano la stessa area dati ed istruzioni del processo che li ha generati.

2. I thread comunicano mediante memoria condivisa.

Poiché i thread condividono la stessa area di memoria, la comunicazione tra loro può avvenire utilizzando semplicemente puntatori a messaggi. La comunicazione tra processi implica invece la trasmissione di un messaggio da un'applicazione ad un'altra. La comunicazione tra processi di conseguenza, può essere causa della duplicazione di grandi quantità di dati.

3. *La concorrenza tra thread è molto efficiente*.

Processi attivi e thread concorrono tra loro all'utilizzo del processore. Poiché il contesto di un thread è sicuramente minore di quello di un processo, è minore anche la quantità di dati da salvare sullo stack di sistema. L'alternanza tra thread risulta quindi più efficiente di quella tra processi.

4. *I thread sono parte del codice della applicazione che li genera.*

Per aggiungere nuovi thread ad un processo, è necessario modificare e compilare tutto il codice della applicazione con tutti i rischi che comporta questa operazione. Viceversa, i processi sono entità dinamiche indipendenti tra loro. Aggiungere un processo significa semplicemente eseguire una nuova applicazione.

5. I thread possono accedere liberamente ai dati utilizzati da altri thread sotto il controllo del medesimo processo.

Condividere la stessa area di memoria con il processo padre e con gli altri thread figli, significa avere la possibilità di modificare dati vitali per gli altri thread o, nel caso peggiore per il processo stesso. Viceversa, un processo non ha accesso alla memoria riservata da un altro.

6. I thread consentono di sfruttare la meglio i moderni processori.

La maggior parte dei sistemi moderni ha più processori e ogni processore ha più core. Il multithreading consente l'esecuzione di thread diversi da parte di processori diversi, consentendo in tal modo un utilizzo più efficiente delle risorse di sistema.

7. *I thread consentono un'elaborazione più rapida delle attività in background/batch.*

Quando è necessario eseguire più attività contemporaneamente, il multi-threading consente alle diverse attività di procedere in parallelo. Di conseguenza, il tempo di elaborazione complessivo viene ridotto.

8. *I thread riducono i tempi di risposta di una applicazione.*

Gli utenti si aspettano che le applicazioni Java siano veloci. Suddividendo l'elaborazione necessaria per una richiesta in blocchi più piccoli e facendo in modo che thread diversi gestiscano l'elaborazione in parallelo, è possibile ridurre i tempi di risposta. In generale, una applicazione asincrona è più veloce di una applicazione sincrona.

9. Richiedono una attenta sincronizzazione dell'uso delle risorse.

Più thread in una JVM possono eseguire attività simili. Ad esempio, potrebbe essere necessario aggiornare un file di configurazione oppure accedere e modificare liste di elementi condivise. In questi casi è quindi essenziale sincronizzare i thread, quindi solo un thread. Poiché la sincronizzazione è necessaria per garantire la coerenza delle risorse risorsa comuni, progettare una applicazione multi-threaded è generalmente più complicato e richiede tempi di test mediamente più lunghi.

Concludendo, nonostante sviluppare applicazioni multi-threaded sia in generale più complicato che sviluppare applicazioni single-threaded, i thread sono più efficienti dei processi e consentono di sviluppare facilmente applicazioni complesse e molto efficienti. Il costo di quest'efficienza è però alto poiché, è spesso necessario implementare complesse procedure per la gestione dell'accesso concorrente ai dati condivisi con gli altri thread.

Thread in Java

I thread in Java sono quindi dei processi leggeri, parte di un processo principale cui condividono la memoria, e che implementano multi-tasking effettivo all'interno di un singolo processo.

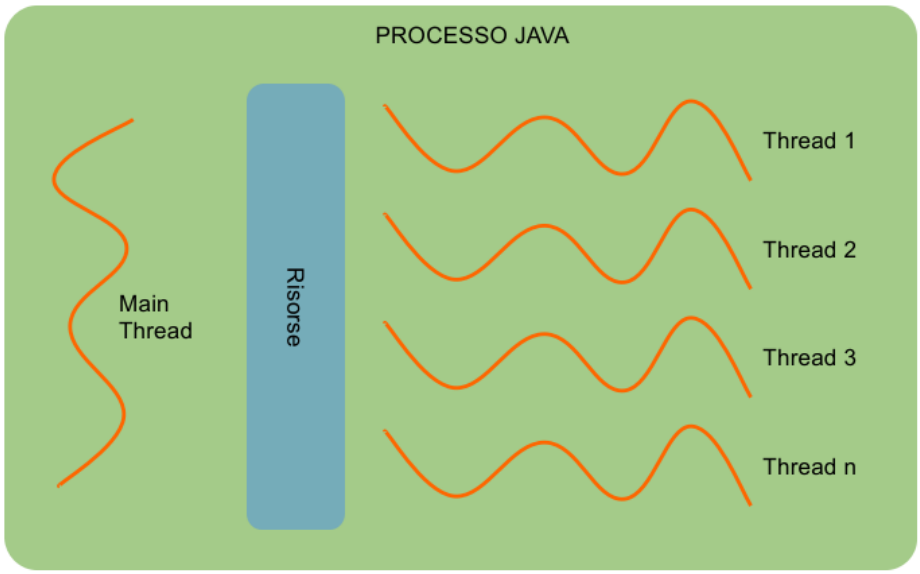


Immagine 57 - Schema di un processo java

Un processo java è generalmente composto da almeno un thread (*main thread*) da cui dipende lo stato del processo in esecuzione. Il *main thread* viene creato dalla JVM per eseguire il processo

che, di fatto, sarà considerato terminato non appena il *main thread* termina la propria esecuzione.

In una applicazione Java multi-threaded, sarà possibile creare thread figli a partire dal thread principale: ognuno dei thread figli condivide la memoria e le risorse già allocate per il processo dalla JVM.

Come per i processi, i thread java posso trovarsi in diversi stati:

1. New

2. Active (Runnable, Running)

3. Waiting/Blocked

4. Timed Waiting

5. *Terminated*

Gli stati di un thread e le loro transizioni sono schematizzati nella prossima immagine:

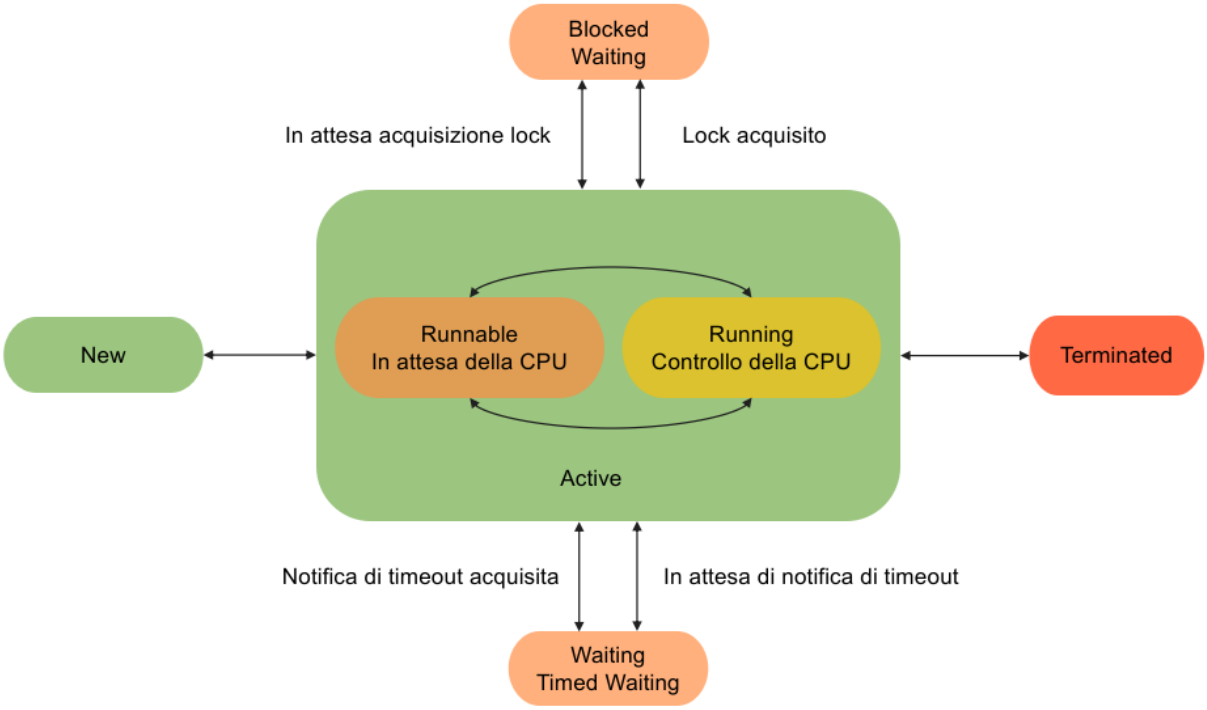


Immagine 58 Stati di unthread e loro transizioni

1. New:

Rappresenta lo stato di un thread non appena viene creato. In questo stato il thread non è ancora stato avviato ed il suo codice deve ancora essere eseguito.

2. Active:

E' lo stato in cui un thread viene trasferito di default non appena viene eseguita la chiamata al metodo che inizia l'esecuzione del codice del thread. Lo stato *Active* è a sua volta composto da due sotto stati: *Runnable* e *Running*. In stato *Runnable* il thread è pronto per essere eseguito ed è in attesa di ottenere il controllo della CPU per la durata del *quanto* di tempo assegnatogli; in stato *Running* il thread ha il controllo della CPU per la durata del quanto di tempo al termine del quale passerà nuovamente in stato *Runnable*.

3. Blocked Waiting:

E' lo stato in cui un thread viene *parcheggiato* perché bloccato in attesa. Ad esempio, se un thread T1 è in attesa di utilizzare una risorsa che non può essere condivisa (come ad esempio una telecamera) e la risorsa è già in uso da un thread T2, T1 verrà posto in uno stato di attesa fino a che T2 non terminerà rilasciando la risorsa rendendola nuovamente disponibile.

4. Waiting:

E' lo stato in cui un thread viene *parcheggiato* per un *tempo indefinito* in attesa che il altro thread completi le sue operazioni.

5. Timed waiting:

E' lo stato in cui un thread viene *parcheggiato* per un *tempo definito* in attesa che il altro thread completi le sue operazioni.

6. Terminated:

E' lo stato di un thread che ha terminato le sue operazioni oppure è stato terminato a causa di un errore fatale.

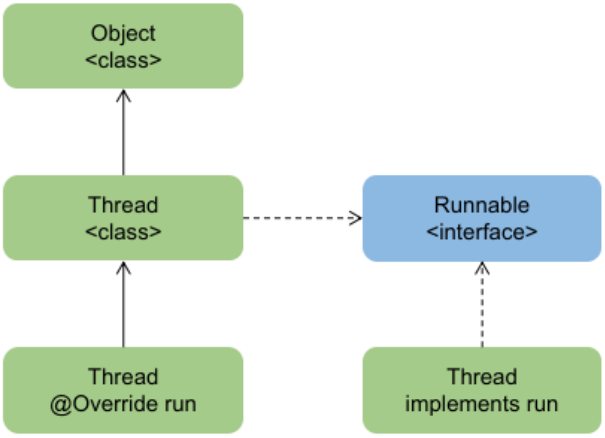


Immagine 59 Creazione di thread: java.lang.Thread e java.lang.Runnable

Java mette a disposizione diversi meccanismi per creare thread. Il primo metodo consiste nell'utilizzare ereditarietà ed è estendere la classe base *java.lang.Thread* mediante la direttiva **extends**.

Tuttavia, poiché l'ereditarietà singola di Java impedisce ad una classe di avere più di una classe padre sebbene possa implementare un numero arbitrario di interfacce, in tutte le situazioni in

cui non sia possibile derivare una classe da *java.lang.Thread*, può essere utilizzata l'interfaccia *java.lang.Runnable* come mostrato nella precedente immagine.

Priorità di un thread

Come abbiamo già anticipato, i vari sistemi operativi affrontano il problema del multi-tasking essenzialmente in due modi:

1. *Preemptive method:*

Il thread con la priorità più alta viene eseguito fino alla sua morte (terminated), alla sua messa in attesa (waiting, timed waiting, blocked waiting) o alla creazione di un thread con priorità maggiore.

2. Time slicing:

Un thread ha comunque un determinato quanto di tempo di esecuzione dopodiché è messo nello stato Runnable.

Normalmente dei due, i primo è quello più comunemente utilizzato anche se con piccole variazioni da sistema a sistema.

Dovendo garantire la portabilità di una applicazione Java su tutte le piattaforme supportate, per la gestione del multi-threading la JVM non si affida al sistema operativo, ma utilizza una politica di gestione dei thread di tipo *fixed-priority pre-emptive scheduling*: quando ci sono thread pronti per essere eseguiti, la JVM sceglie il thread in stato *runnable* con priorità più alta e lo esegue fino a che il thread termina oppure un altro thread con priorità più alta passa in stato *runnable*. Se più thread con uguale priorità passano in stato *runnable*, saranno eseguiti in ordine *FIFO* (First In First Out).

In Java quindi è possibile assegnare ad ogni thread un numero intero da 1 a 10 che indica la sua priorità: quando un thread viene creato eredita la priorità del thread che lo ha creato. E' comunque possibile modificare questa priorità come vedremo nel paragrafo successivo.

Tra tutti i thread in attesa di esecuzione, il run-time di Java sceglierà quello con la priorità più alta; questo thread andrà in esecuzione finché:

1. un thread con una priorità più alta diventa Runnable;

2. il thread non passa in uno stato di terminated oppure decide bonariamente di rilasciare la CPU;

3. *il thread passa in uno stato di attesa (waiting, timed waiting, blocked waiting).*

Per evitare che un thread con priorità molto alta possa prendere il controllo della CPU per un tempo indefinito senza mai rilasciarla, la JVM si riserva comunque il diritto di eseguire thread con priorità più basse se questo può evitare il congelamento (*starvation*) del programma quindi, in definitiva:

i*n una applicazione Java multi-threaded, la priorità è utilizzata per creare thread con differenti livelli di priorità in modo che alcuni thread vengano eseguito utilizzando una corsia preferenziale: più alta è la priorità più alte saranno le chance per il thread di essere eseguito.*

La classe java.lang.Thread

La classe *java.lang.Thread*, non è una classe astratta, implementa l'interfaccia *java.lang.Runnable* e incapsula il codice necessario al funzionamento del thread: avvio, esecuzione, gestione degli stati di attesa, interruzione.

Di tutti metodi definiti nella classe Thread, il metodo *public void run()* deve essere utilizzato per implementare le operazioni eseguite dal thread riscrivendone la definizione nella nuova classe mediante il meccanismo di *overriding*. Io metodo *run()* è anche detto *entry point* per un thread ed è l'unico metodo su cui effettuare *overriding*. Nel caso in cui il metodo non sia riscritto, una volta avviato il thread eseguirà una funzione nulla e terminerà immediatamente.

In generale, il ciclo di vita di un thread è legato al metodo *run()*: non appena il metodo *run()* dovesse tornare, il thread terminerà immediatamente.

Nel prossimo esempio è mostrato un semplice thread che stampa a video i numeri da 1 a 10 e poi termina interrompendo il ciclo di controllo e ritornando. La classe *SimpleThread* è realizzata a partire dalla classe *java.lang.Thread* da cui eredita tutti i metodi necessari alla gestione del ciclo di vita del thread.

```
public class SimpleThread extends Thread{

  @Override
  public void run() {
    int i = 0;
    while(true){
      try {
        sleep(1000);
        System.out.println(++i);
        if(i==10) break;
      } catch (InterruptedException e) {
        e.printStackTrace();
        break;
      }
    }
  }

  public static void main(String[] args){
    SimpleThread primoThread = new SimpleThread();
    primoThread.start();
  }

}
```

Il thead, come tutti gi oggetti java, può essere creato utilizzando l'operatore **new** (in questo momento il thread si trova in stato *New*), ma la chiamata al metodo *start()* lo porta in stato *Active*: in questo stato viene eseguito il metodo *run()* ed il thread entra nella coda di schedulazione per ottenere il controllo del processore.

Il metodo *run()* contiene un ciclo infinito all'interno del quale viene eseguito, come prima istruzione, il metodo *sleep(long millis)* ereditato dalla super classe. Il metodo sleep provoca il passaggio del thread dallo stato A*ctive* allo stato T*imed Waiting* per uno specifico periodo di tempo in millisecondi rappresentato dall'attributo del metodo; al risveglio la variabile *i* viene aggiornata e viene stampato il risultato a video.

Il ciclo *while* viene terminato non appena i prende il valore 10: a questo punto il metodo *run()* termina e con esso il thread passa allo stato di T*erminated*.

Nella prossima immagine è schematizzato il diagramma di flusso dei cambiamenti di stato del thread.

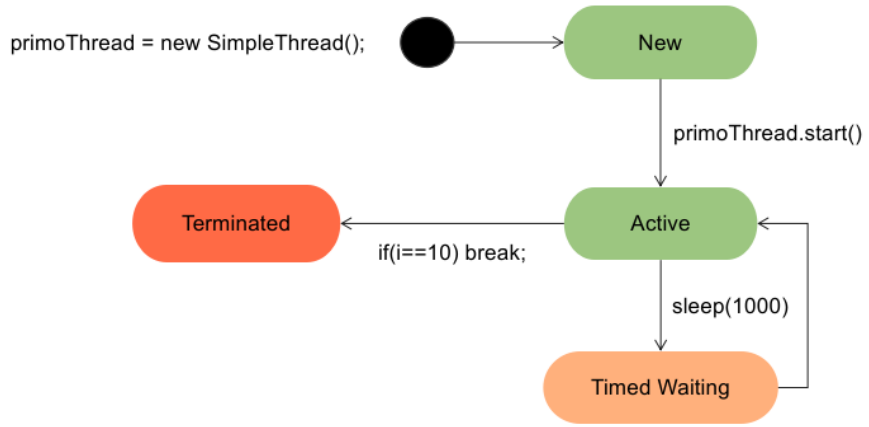


Immagine 60 - Diagramma di cambio di stato del thread

Di fatto quasi tutti i thread contengono cicli infiniti, e questo perché generalmente svolgono compiti ripetitivi per tutta la durata del ciclo di vita del processo padre.

Nella prossima tabella sono elencati i principali metodi ereditati dalla classe *java.lang.Thread*:

Metodi ereditati da *java.lang.Thread*
void run()   E' il metodo che contiene il codice che deve essere eseguito dal thread. E' l'unico metodo della classe Thread che generalmente deve essere modificato per mezzo dell'overriding.
static void sleep(long millis), static void sleep(long millis, int nanos)   Metodi statici. Sospendono il thread mettendolo in uno stato di timed waiting per un tempo determinato espresso in millisecondi nella prima variante, millisecondi+nanosecondi nella seconda. Propaga una eccezione di tipo *InterruptedException* se il thread viene interrotto.
void start()

Esegue il thread: provoca il passaggio del thread da uno stato *New* ad uno stato *Active*. Inizia l'esecuzione del metodo *run()*.

void interrupt()

Segnala che un thread è pronto per essere terminato. La terminazione di un thread sarà trattata a breve.

boolean isInterrupted()

Torna *true* se il thread deve essere terminato. E' diretta conseguenza della chiamata al metodo *interrupt()*.

static Thread currentThread()

Metodo statico: ritorna un riferimento al thread in esecuzione al momento della chiamata al metodo.

static void yeld()

Un suggerimento allo scheduler che il thread corrente è disposto a cedere l'uso corrente del processore. Lo scheduler è libero di ignorare questo suggerimento.

final void join() throws InterruptedException

Quando invocato il thread chiamante va in uno stato di waiting fino a che il thread referenziato non termina. Propaga una eccezione di tipo *InterruptedException* se il thread viene interrotto.

final void join(long millis) throws InterruptedException
final void join(long millis, int nanos) throws InterruptedException

Quando invocato il thread chiamante va in uno stato di *Waiting* fino a che il thread referenziato non termina oppure non scade un timeout. Propaga una eccezione di tipo *InterruptedException* se il thread viene interrotto.

void setPriority(int priority)

Consente di modificare la priorità di un thread. Prende un intero con valore da 1 a 10. La classe Thread dispone delle tre costanti:

Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5 default)
Thread.MAX_PRIORITY (10)

int getPriority()

Ritorna la priorità attuale del thread.

void setDaemon(boolean on)

Imposta un thread come thread demone.

Nel prossimo esempio, andremo ad analizzare il comportamento della JVM in uno scenario in cui più thread con priorità diverse sono in stato *Runnable* pronti per essere eseguiti.

Come anticipato nella tabella precedente, la classe *java.lang.Thread* dispone dei metodi: *setPriority* e *getPriority* che rispettivamente, imposta la priorità di un thread, ritorna la priorità del thread.

I valori ammessi per la priorità di un thread sono compresi tra *java.lanag.Thread.MAX_PRIORITY* e *java.lanag.Thread.MIN_PRIORITY* che valgono rispettivamente 1 e 10. La priorità di base di un thread è *java.lanag.Thread.NORM_PRIORITY* che vale 5.

Nel prossimo esempio, l'applicazione *Priorita* esegue due thread identici aventi però diverse priorità. Il metodo run() dei thread contiene un ciclo infinito che stampa sul terminale una stringa che lo identifichi.

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

@AllArgsConstructor
public class Priorita extends Thread {
  @Getter @Setter
  private String nome;

  public void run() {
    while (true)
      System.out.println(getNome());
  }

  public static void main(String[] argv) {
    Priorita Thread1 = new Priorita("Thread con priorità alta");
    Priorita Thread2 = new Priorita("Thread con priorità bassa");
    Thread1.setPriority(Thread.MAX_PRIORITY);
    Thread2.setPriority(Thread.MIN_PRIORITY);
    Thread1.start();
    Thread2.start();
  }

}
```

Il risultato dell'esecuzione è stato messo in tabella su più colonne da sinistra verso destra. Per comodità è riportato nella prossima pagina:

```
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità alta
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
Thread con priorità bassa
```

Alcune considerazioni importanti:

1. Indipendentemente dalla priorità dei thread, non è possibile determinarne l'ordine ed i tempi di attivazione.

2. Quando un thread con priorità più alta viene messo in stato Runnable, prende il controllo della CPU e viene eseguito fino a che non termina o, nel nostro caso, la JVM decide di lasciare spazio al thread con priorità minore per evitare starvation.

3. Entrambi i thread vengono eseguiti, ma il thread con priorità più alta ottiene un controllo maggiore della CPU rispetto a quello con priorità più bassa.

Capita spesso che thread con priorità molto alta facciano uso di risorse di sistema, necessarie al funzionamento di altri thread. In queste situazioni, i thread con minore priorità potrebbero trovarsi a dover lavorare in mancanza di risorse. In questi casi può essere utilizzato il metodo *yield()* oppure il metodo *sleep(long millis)* affinché, il thread con priorità più alta, diminuisca temporaneamente la propria priorità oppure decida spontaneamente di mettersi in uno stato di attesa consentendo ad altri thread di accedere alle risorse e procedere nel loro lavoro.

Infine, esistono dei thread particolari, con priorità molto bassa detti "thread daemons" o servizi. Questi thread, provvedono ai servizi di base di un'applicazione attivandosi solo quando la macchina è al minimo delle attività (idle). La classe *java.lanag.Thread* è dotata del metodo public final void *setDaemon(boolean on)* che imposta il thread come *thread daemon*.

Cambi di stato di un thread e la classe java.lang.Thread

Completiamo la prima parte di questo capitolo sui thread con una immagine che mette in relazione il ciclo di vita dei thread con i metodi della classe Thread deputati alla gestione o alla transizione di stato.

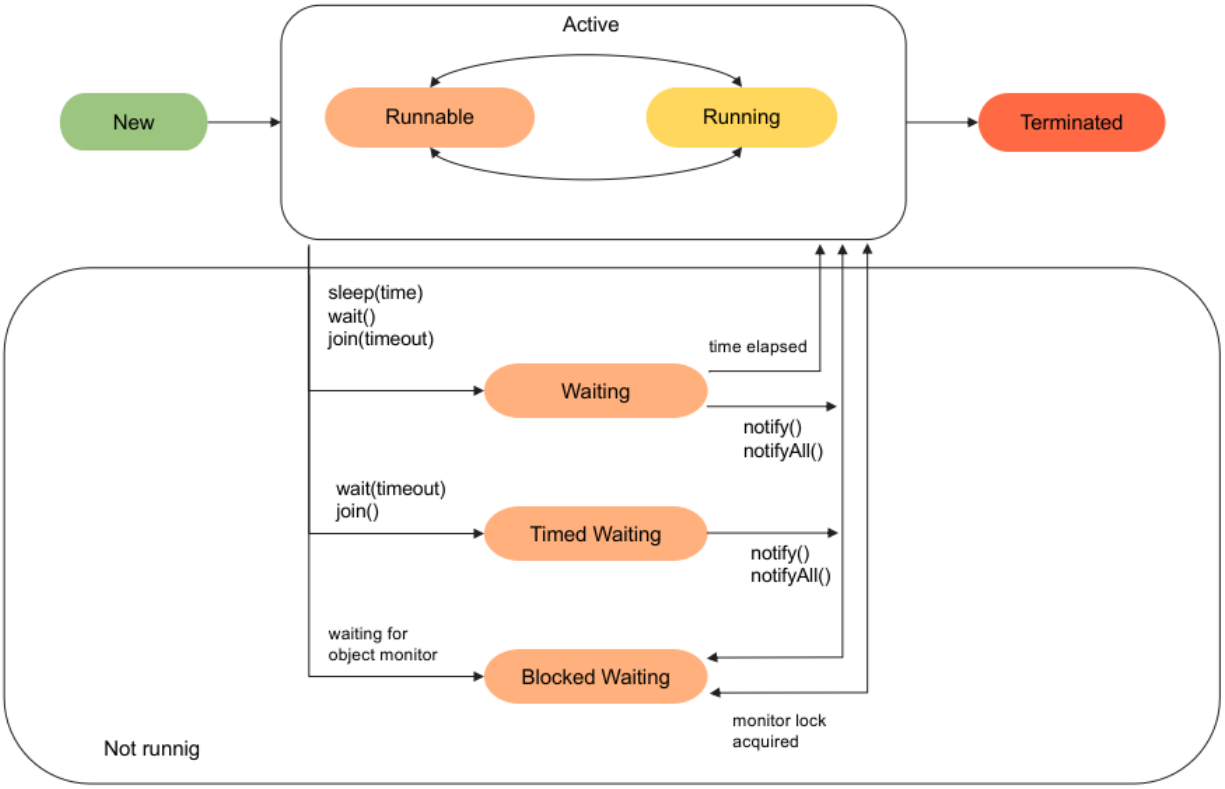


Immagine 61 - transizioni di stato e metodi della classe Thread

I metodi *wait, notify e notifyAll* nopn appartengono alla classe Thread ma vengono ereditati dalla classe Object. Pertanto tutte le classi java li ereditano per definizione.

Stati Waiting e Timed Waiting

Come abbiamo già detto, il modello di gestione della concorrenza in Java prevede che un thread possa prendere il controllo della CPU fino al termine della propria esecuzione prima di liberala a favore del thread successivo in attesa in una coda FIFO. Abbiamo anche detto che i thread hanno una priorità che altera la probabilità che un thread ottenga il controllo della CPU, e che la JVM può decidere autonomamente di rimuovere un thread dallo stato *Running* mettendolo in uno stato *Runnable* qualora ritenga che il thread possa causare una situazione di *starvation* per il processo corrente.

Nel mondo reale però, in una applicazione multi-threaded i thread passano sempre parte del loro tempo in uno stato di attesa e questo è dovuto a due fattori:

1. I thread vanno spesso sincronizzati

Sopratutto quando parliamo di accesso concorrente alle risorse della applicazione.

2. I thread vengono spesso parcheggiati in stato Timed Wait.

I programmatori esperti sanno perfettamente che una applicazione multi-threaded con sotto processi che eseguono un cicli incontrollati senza rilasciare mai la CPU, può provocare problemi di prestazioni non solo alla applicazione, ma anche alla macchina su cui la applicazione sta girando mettendo la CPU sotto stress sia dal punto di vista del carico di operazioni, sia dal punto di vista della dissipazione del calore. Per questo motivo, si è soliti progettare thread in modo che rilasciano periodicamente la CPU.

Per supportare la gestione dei thread, la classe *Thread* mette a disposizione alcuni metodi che possono essere utilizzati per mettere un thread in stato di attesa controllata ovvero in uno stato di *waiting* o T*imed Waiting*.

Il primo di questi lo abbiamo già incontrato; il metodo *sleep(long millis).* Questo metodo, quando invocato, causa il passaggio di un thread in un stato di T*imed Waiting* per tutta la durata del tempo definito come attributo del metodo. Propaga una eccezione di tipo *InterruptedException* qualora il thread venga interrotto prima della scadenza del periodo previsto.

Supponiamo adesso che un thread debba aspettare che un altro thread abbia completato le proprie operazioni. In questo caso possiamo utilizzare il metodo *join()* che mette un thread in uno stato di W*aiting* fino a che il thread su cui è stato chiamato il metodo non viene terminato. Nel prossimo esempio, il *main-thread* della applicazione crea tre thread e si mette in attesa, W*aiting* , della sua terminazione prima di uscire dalla applicazione.

```
public class EsempioThreadConJoin implements Runnable {

  @Override
  public void run() {
    System.out.println("Thread avviato:::" + Thread.currentThread().getName());
    try {
      Thread.sleep(4000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Thread terminato:::" + Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    Thread t1 = new Thread(new EsempioThreadConJoin(), "thread 1");
    Thread t2 = new Thread(new EsempioThreadConJoin(), "thread 2");
    Thread t3 = new Thread(new EsempioThreadConJoin(), "thread 3");

    try {
      // avvia il thread t1
    t1.start();
      t1.join();
      // t2 viene avviato solo al termine di t1
      t2.start();
      // poiché t1 è già terminato non c'è attesa
      t1.join();
      t3.start();
      t1.join();
      t2.join();
      t3.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Tutti i thread sono terminati. Esco dal thread principale");
  }

}
```

Da notare che se un thread è già terminato, la chiamata al metodo *join()* ritorna immediatamente.

Nell'esempio precedente, l'applicazione potrebbe rimanere bloccata in attesa di una thread che non terminerà mai, e questo per molti motivi: *t1* potrebbe contenere un ciclo infinito senza condizioni di uscita, oppure *t1* potrebbe rimanere in attesa a sua volta di un altro thread o di una risorsa condivisa. Per evitare queste condizioni, è possibile utilizzare il metodo *join(long millis)* oppure *join(long millis, int nano)* che consentono di definire un timeout alla scadenza del quale il thread chiamante riprenderà comunque la propria esecuzione:

```
public class EsempioThreadConJoinETimeout implements Runnable {

  @Override
  public void run() {
    System.out.println("Thread avviato:::" + Thread.currentThread().getName());
    try {
      Thread.sleep(180000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Thread terminato:::" + Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    Thread t1 = new Thread(new EsempioThreadConJoinETimeout(), "thread 1");
    Thread t2 = new Thread(new EsempioThreadConJoinETimeout(), "thread 2");
    Thread t3 = new Thread(new EsempioThreadConJoinETimeout(), "thread 3");

    try {
      // avvio il thread t1
      t1.start();
      // attendo che t1 termini o siano passati 4 secondi
      t1.join(4000);
      //avvia i restati thread e rimane in attesa per tutta la durata della loro esecuzione
      t2.start();
      t3.start();
      t1.join();
      t2.join();
      t3.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Tutti i thread sono terminati. Esco dalthread principale");
  }

}
```

Come per *sleep()*, il metodo *join(...)* propaga una accezione di tipo *InterruptedException* qualora il thread su cui è stata fatta la chiamata venga interrotto.

Interruzione di un thread

E' buona norma che una applicazione in esecuzione possa interrompere tutti i thread in esecuzione su richiesta (ad esempio prima di terminare ed uscire). Abbiamo anticipato che il

ciclo di vita di un thread è strettamente legato al metodo *run(): fino a che il metodo run() è in esecuzione, il thread è da considerarsi in stato running.*

Abbiamo anche detto che la classe *Thread* mette a disposizione il metodo *interrupt()* per *segnalare che un thread è pronto per essere terminato*. Per capire meglio questa affermazione consideriamo il prossimo esempio:

```
public class InterruptedThread extends Thread{
  @Override
  public void run() {
    int i = 0;
    while (true) {
      System.out.println(++i);
      if (i == 1000000) break;
    }
  }
  public static void main(String[] args) {
    InterruptedThread simpleThread = new InterruptedThread();
    simpleThread.start();
    simpleThread.interrupt();
  }
}
```

Il metodo *run()* del thread in esempio entra in un ciclo infinito, stampa a video i primi 1000 numeri e poi termina causando la terminazione del thread. Nonostante nel metodo *main* della nostra applicazione avviamo il thread e lo interrompiamo immediatamente dopo. Tuttavia quello che notiamo è che il thread non si ferma realmente, completa il ciclo **while**, stampa i primi 1000000 numeri, e poi finalmente esce.

Il motivo di questo comportamento apparentemente anomalo è legato al fatto che il metodo *interrupt()* nonostante il suo nome non agisce direttamente sullo stato del thread terminandolo, ma modifica una variabile di istanza booleana della classe Thread chiamata *interrupted,* demandando al metodo *run()* la gestione della terminazione effettiva del thread. Tutto questo per dare modo al thread di rilasciare risorse che altrimenti rimarrebbero bloccate in modo anomalo. L'esempio precedente sarà riscritto correttamente nel modo seguente:

```
public class InterruptedThread extends Thread {
  @Override
  public void run() {
    int i = 0;
    while (!isInterrupted()) {
      System.out.println(++i);
      if (i == 1000000)  break;
    }
    //codice per il rilascio delle risorse condivise
  }
  public static void main(String[] args) {
    InterruptedThread simpleThread = new InterruptedThread();
    simpleThread.start();
```

```
    simpleThread.interrupt();
  }
}
```

Adesso il thread si comporterà nel modo atteso interrompendo l'esecuzione non appena viene chiamato i metodo *interrupt()*.

Nasce quindi un ultima domanda: cosa succede se il thread se il thread è in uno stato di attesa *(waiting, timed waiting, blocked waiting)* al momento della chiamata del metodo *interrupt()*? Abbiamo anticipato nel paragrafo precedente che i metodi bloccanti della classe Thread possono propagare un eccezione *checked* di tipo *InterruptedException*. La regola è la seguente:

1. Quando viene invocato il metodo interrupt(), se il thread si trova in uno stato di attesa (waiting, timed waiting, blocked waiting), il relativo metodo bloccante provocherà la propagazione di una eccezione del tipo InterruptedException.

InterruptedException va quindi interpretata sempre coma una richiesta di interruzione del thread. Per questo motivo ogni volta che catturiamo una eccezione id questo tipo dobbiamo gestire le risorse aperte e terminare il thread. In quest'ottica l'applicazione *SimpleThread* dei paragrafi precedenti deve essere riscritta nel modo seguente:

```
public class SimpleThread extends Thread{

  @Override
  public void run() {
    int i = 0;
    while(true){
      try {
        sleep(1000);
        System.out.println(++i);
        if(i==10) break;
      } catch (InterruptedException e) {
        e.printStackTrace();
        break;
      }
    }
  }

  public static void main(String[] args){
    SimpleThread primoThread = new SimpleThread();
    primoThread.start();
  }
}
```

Valgono i seguenti criteri (best practices) per l'interruzione di thread:

Se una chiamata bloccante lancia l'eccezione *InterruptedException*, il thread dovrebbe interpretarla come una richiesta di terminazione, e reagire assecondando la

richiesta.

Se un thread non utilizza periodicamente chiamate bloccanti, dovrebbe invocare periodicamente *isInterrupted()* e terminare se il risultato è true.

Interfaccia java.lang.Runnable

L'ereditarietà singola di Java impedisce ad una classe di avere più di una classe padre, sebbene possa implementare un numero arbitrario di interfacce. In tutte le situazioni in cui non sia possibile derivare una classe da *java.lang.Thread*, può essere utilizzata l'interfaccia *java.lang.Runnable.*

```
public interface Runnable{
        public void run();
}
```

L'interfaccia *Runnable* contiene la definizione di un solo metodo astratto, il metodo *run()* che conterrà il codice del thread da eseguire. Utilizzando *Runnable*, la classe *SimpleThread* può essere riscritta nel modo seguente:

```
public class SimpleThread implements Runnable{
  public void run() {
    int i = 0;
    while(true){
      try {
        Thread.sleep(1000);
        System.out.println(++i);
        if(i==10) break;
      } catch (InterruptedException e) {
        e.printStackTrace();
        break;
      }
    }
  }
}
```

In generale, tutti i thread sono creati utilizzando l'interfaccia *Runnable,* anche la classe *java.lang.Thread* implementa quest'interfaccia, e questo è dovuto alla flessibilità offerta dalle interfacce rispetto alle classi, nel disegno delle gerarchie di ereditarietà. D'altra parte, un'interfaccia rappresenta solo un modello astratto. Ricordiamo, infatti, che un'interfaccia può contenere solo metodi astratti e variabili di tipo **static final**, di conseguenza un interfaccia non può essere utilizzata per creare un oggetto, in altre parole, non potremmo scrivere:

```
Runnable mioThread = new Runnable()
```

In definitive quindi, implementare la classe *Runnable* non è sufficiente a creare un oggetto thread, semplicemente, abbiamo definito una classe che assomiglia ad un thread.

Nel paragrafo precedente, abbiamo detto che, molto del lavoro di un thread è svolto dalla classe *java.lang.Thread* che contiene la definizione dei metodi necessari a gestirne il ciclo di vita completo. Andandola ad analizzare un po più a fondo, notiamo subito che mette disposizione una serie di metodi costruttori che accetta come argomento un tipo *Runnable:*

Metodi costruttori di *Thread* che accettano un tipo *Runnable*
Thread(Runnable target)   Crea un nuovo thread a partire da un tipo Runnable.
Thread(Runnable target, String name)   Crea un nuovo thread a partire da un tipo *Runnable* assegnando un nome al nuovo thread.
Thread(ThreadGroup group, Runnable target)   Crea un nuovo thread a partire da un tipo *Runnable* associandolo ad un gruppo di thread.
Thread(ThreadGroup group, Runnable target, String name)   Crea un nuovo thread a partire da un tipo *Runnable* associandolo ad un gruppo di thread. Assegna un nome al nuovo thread
Thread(ThreadGroup group, Runnable target, String name, long stackSize)   Crea un nuovo thread a partire da un tipo *Runnable* associandolo ad un gruppo di thread e con uno stack di dimensione stacksize. Assegna un nome al nuovo thread.

In definitiva, l'applicazione SimpleThread diventa:

```
public class SimpleThread implements Runnable{
  public void run() {
    int i = 0;
    while(true){
      try {
        Thread.sleep(1000);
        System.out.println(++i);
        if(i==10) break;
      } catch (InterruptedException e) {
        e.printStackTrace();
        break;
      }
    }
  }

  public static void main(String[] args) {
    Thread simpleThread = new Thread(new SimpleThread());
    simpleThread.start();
  }
}
```

Nasce quindi la domanda: come faccio a gestire l'interruzione di un thread se non posso utilizzare il metodo non statico *isInterrupted()* per rispettare i criteri di interruzione di un thread? Ci viene in aiuto il metodo statico *Thread.currentThread()* che, come abbiamo già detto, ritorna un riferimento al thread corrente al momento della chiamata al metodo. L'esempio precedente può essere quindi riscritto nel modo seguente:

```
public class SimpleThread implements Runnable{
  public void run() {
    int i = 0;
    while(!Thread.currentThread.isInterrupted()){
      try {
        Thread.sleep(1000);
        System.out.println(++i);
        if(i==10) break;
      } catch (InterruptedException e) {
        e.printStackTrace();
        break;
      }
    }
     // codice necessario per il rilascio delle risorse
  }

  public static void main(String[] args) {
    Thread simpleThread = new Thread(new SimpleThread());
    simpleThread.start();
  }
}
```

Thread anonimi e thread come espressioni lambda

Come tutte le classi Java, anche i thread possono essere creati come classi anonime utilizzando l'interfaccia *Runnable*. Ricordando quanto già detto sulle classi anonime, possiamo possiamo creare thread anonimi nei due modi seguenti:

```
Thread t = new Thread(new Runnable() {
  public void run() {
    //codice del thread da eseguire
  }
});

t.start();
```

Oppure, in alternativa possiamo usare un approccio più orientato alle classi anonime:

```
Thread thread = new Thread() {
  public void run() {
```

```
    // your code here
  }
}

thread.start();
```

Tuttavia notiamo che l'interfaccia thread risponde perfettamente alla definizione di interfaccia funzionale, e pertanto possiamo creare thread anonimi utilizzando una espressione lambda con tutti i vantaggi che ne conseguono legati sopratutto alle *closure*. Grazie alla sintassi per le espressioni lambda possiamo creare un thread anonimo nel modo seguente:

```
Runnable runnable = () -> {
  // il codic da eseguire va inserito qui
};
Thread t = new Thread(runnable);
t.start();
```

Oppure, in alternativa:

```
Thread t = new Thread(() -> {
  // your code here ...
});
```

Oppure possiamo omettere la creazione di una variabile reference:

```
new Thread(() -> // your code here).start();
```

Thread vs Runnable

La domanda che sorge spontanea è la seguente: quale dei due metodi analizzati è più appropriato a creare un thread in java? Le considerazioni da fare sono tante e molteplici.

1. Come abbiamo già accennato, java non supporta ereditarietà multipla, il che significa che possiamo estendere una sola classe. Una volta che abbiamo esteso la classe Thread, abbiamo perso l'opportunità di estendere o ereditare un altra classe.

2. Nella programmazione ad oggetti, estendere una classe significa generalmente aggiungere nuove funzionalità oppure modificarne il comportamento o la forma. In generale quindi, se non abbiamo bisogno di specializzare o modificare la classe Thread è comunque preferibile utilizzare l'interfaccia Runnable.

3. Come vedremo più in la in questo capitolo, la classe Runnable rappresenta più genericamente un task, e come tale può essere eseguito come un thread, ma anche tramite executors oppure futurable, callable come in tante altre forme. Questa separazione logica del task da eseguire dal thread diventa quindi quasi sempre una buona scelta progettuale.

4. Per la stessa ragione, separare il task dal thread implementando Runnable significa anche favorire il riuso del codice.

5. Ereditare tutti i metodi di Thread rappresenta comunque un overhead inutile dal momento che, in ogni caso, stiamo rappresentando un task e la cosa può essere fatta semplicemente utilizzando runnable.

Di motivazioni per cui implementare *Runnable* è sempre più conveniente che estendere *Thread* se ne possono trovare molte altre anche di tipo più prettamente tecnico tra cui un migliore utilizzo della memoria. A conclusione di questo paragrafo mi limiterò quindi a dare il seguente suggerimento:

A meno che non sia necessario modificare o specializzare una classe, implementare è sempre la scelta consigliata rispetto ad estendere.

Sincronizzazione tra thread

In un ambiente multi-threaded in cui due o più thread concorrono all'utilizzo delle risorse della applicazione, è inevitabile che prima o poi si trovino in conflitto quando devono accedere entrambi ad una risorsa condivisa: i thread si trovano a dover *competere* per ottenere l'assegnazione e questo provoca un conflitto anche detto *race-condition* (letteralmente condizione di gara).

Una *race-condition* è generalmente causa di errori gravi come accade nel prossimo esempio in cui due thread tentano di accedere ad un oggetto che rappresenta un conto bancario per effettuare un prelievo di denaro. Supponendo che il conto bancario abbia un saldo di 600 euro:

1. Il thread A accede al conto chiedendo di ritirare 400 euro;

2. Un metodo dell'oggetto conto controlla il saldo trovando 600 euro;

3. Il thread B accede al conto chiedendo di ritirare 400 euro;

4. Un metodo dell'oggetto conto controlla il saldo trovando 600 euro;

5. Thread A preleva 400 euro ;

6. Thread B preleva 400 euro ;

7. Il conto va in scoperto di 200 euro.

La nostra classe *ContoCorrente* potrebbe assomigliare al prossimo esempio:

```
public class ContoCorrente {
  private double saldo;
  public double getSaldo(){
    return saldo;
  }
  public void deposita(double sommaDaDepositare){
    saldo = saldo+sommaDaDepositare;
  }
  public double preleva(double sommaDaPrelevare){
```

```
        if(saldo >= sommaDaPrelevare){
          saldo = saldo -sommaDaPrelevare;
          return sommaDaPrelevare;
        }
        return 0;
      }
    }
```

Condizioni di questo tipo avvengono, per ovvi motivi, solo nel momento in cui si tenti di modificare la risorsa: un accesso in sola lettura non è mai causa di *race-condition*.

L'unico modo esistente per gestire questi casi è fare in modo che i thread possano accedere in modo *esclusivo* alla risorsa: ogni altro thread che tenti di accedervi dovrà aspettare il che il primo ha terminato le sue operazioni. Diremo quindi che i thread devono essere *sincronizzati*.

Introduciamo quindi il concetto di *thread-safety* così come definito da :

DEFINIZIONE: Una classe è thread-safe se si comporta correttamente quando si accede ad essa da più thread, indipendentemente dalla pianificazione o interleaving dell'esecuzione di tali thread, dall'ambiente runtime, e senza sincronizzazione aggiuntiva o altro coordinamento sulla parte del codice chiamante. [8]

In definitiva, thread-safety indica la caratteristica di una porzione di codice che si comporta in modo corretto nel caso di esecuzioni multiple da parte di più thread.

La classi immutabili in java sono per definizione thread-safe. Un accesso concorrente ad una classe immutabile, per definizione di immutabilità, non può causare *race-condition*.

Consideriamo adesso il prossimo esempio: definiamo per prima cosa la classe *Contatore* come segue:

```
import lombok.Getter;

public class Contatore {

  @Getter
  private int contatore = 0;

  public void update() {
    contatore++;
  }

}
```

La classe *Contatore* è la nostra risorsa condivisa utilizzata da due thread che tenteranno di accedere in concorrenza all'oggetto:

---

[8] Brian Goetz: Java Concurrency in Practice

```
public class EsempioRaceCondition {

  private static Runnable getRunnable(Contatore contatore) {
    return () -> {
      for (int i = 0; i < 1_000_000; i++) {
        contatore.update();
      }
    };
  }

  public static void main(String[] args) {
    Contatore contatore = new Contatore();
    Thread threadUno = new Thread(getRunnable(contatore));
    Thread threadDue = new Thread(getRunnable(contatore));

    threadUno.start();
    threadDue.start();
    try {
      threadUno.join();
      threadDue.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Contatore vale: "+contatore.getContatore());
  }
}
```

Ognuno dei due thread esegue un ciclo di 1.000.000 di iterazioni ed aggiorna il contatore condiviso. Alla fine dell'esecuzione della applicazione quello che ci aspettiamo è che contatore valga 2.000.000; tuttavia, se proviamo ad eseguire l'applicazione più di una volta otterremo:

```
esecuzione 1: 1907775
esecuzione 2: 1887216
esecuzione 3: 1891186
```

Per meglio comprendere il motivo della *race-condition*, è necessario divagare un attimo e parlare delle architetture multiprocessore. Le moderne architetture hanno processori che sono formati a loro volta da molti core ognuno dei quali rappresenta una CPU indipendente dalle altre ed in grado di eseguire le sue funzioni in parallelo.

I dati di una applicazione sono memorizzati nella memoria RAM. Ogni core è in grado di eseguire un numero enorme di operazioni al secondo e poiché la RAM non è in gradi di supportare la

velocità del core, i dati dalla memoria vengono trasferiti in una memoria veloce detta cache (ne esistono generalmente 3 livelli). Una volta nella cache, il core esegue le sue operazioni sul dato che, alla fine, viene nuovamente trasferito nella RAM e reso disponibile per successive computazioni.

Nella prossima immagine è schematizzata una architettura multiprocessore.

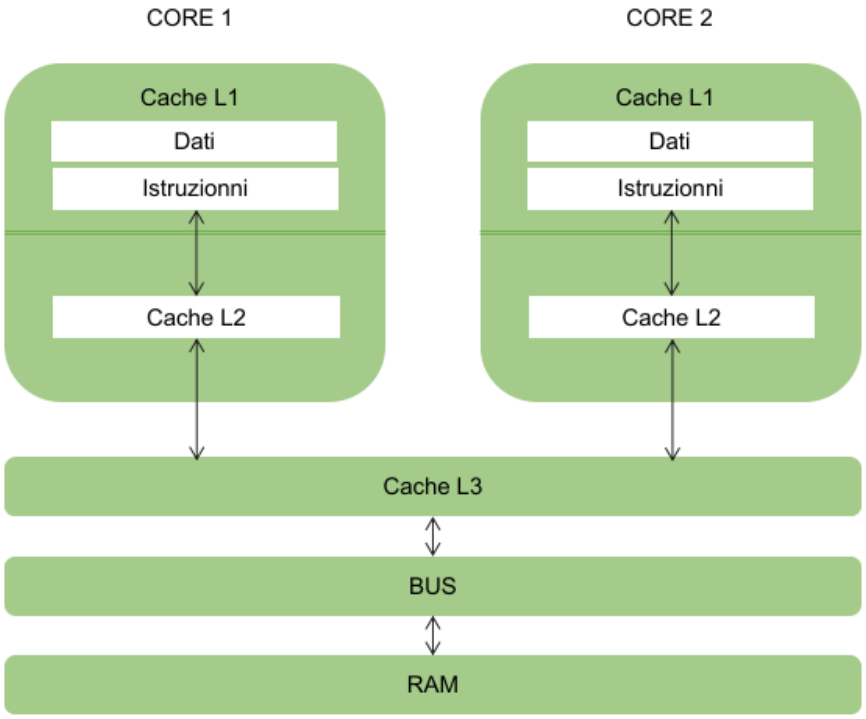


Immagine 62 - architetture multiprocessore

Torniamo quindi al nostro esempio in cui abbiamo due thread e supponiamo che i due thread vengano schedulati per essere eseguiti su due diversi core. Quello che accade è schematizzato nella prossima figura:

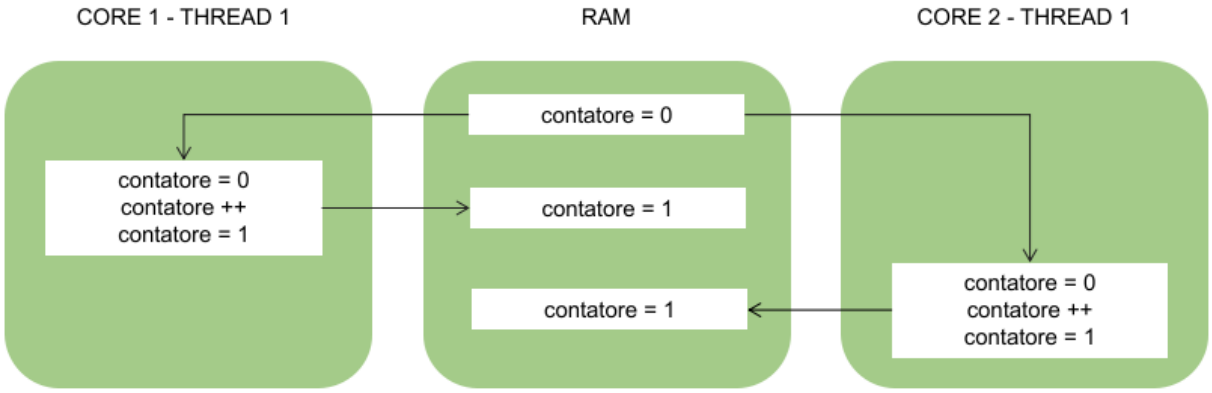


Immagine 63 - accesso interleaved a contatore

1. Entrambi i core, per eseguire le operazioni previste dai thread in esecuzione trasferiscono il valore del contatore nella propria cache;

2. Il primo core esegue le operazioni previste dal thread per il quanto di tempo previsto, e prima di rimettere in coda il thread trasferisce il risultato parziale nuovamente nella RAM;

3. Il secondo core esegue le operazioni previste dal thread per il quanto di tempo previsto, e prima di rimettere in coda il thread trasferisce il risultato parziale nuovamente nella RAM;

4. Al prossimo quanto di tempo il valore del contatore nella RAM viene ritrasferito nella cache dei core ed il processo si ripete fini alla terminazione dei due thread.

Quello che è successo è che, invece di accedere al contatore in modo *sequenziale*, i thread hanno eseguito un accesso di tipo *interleaved* e da qui la causa della *race-condition*.

Quello che invece ci saremmo dovuti aspettare è quanto schematizzato nella prossima immagine:

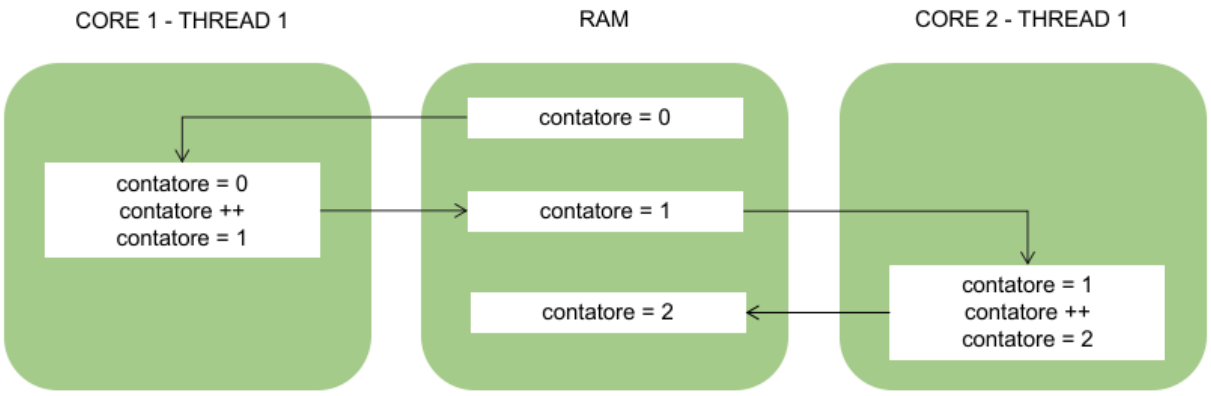


Immagine 64 - accesso sequenziale a contatore

Sezioni critiche

La porzione di codice in cui avviene la *race-condition* è detta *sezione critica*. Una sezione critica è definita formalmente nel modo seguente:

DEFINIZIONE: Una sezione critica, anche detta regione critica, è una porzione di codice che accede a una risorsa condivisa tra più flussi di esecuzione di un sistema concorrente.

Di conseguenza, nell'esempio precedente la sezione critica è rappresentata dal metodo *update()* della classe *Contatore*:

```
import lombok.Getter;

public class Contatore {
  @Getter
  private int contatore = 0;
  public void update() {
    contatore++; //SEZIONE CRITICA
```

```
    }
  }
```

Il problema dell'accesso concorrente alle risorse si risolve quindi alla gestione della sezione critica. Un possibile approccio al problema della sezione critica è quello di rendere la porzione di codice *atomica:*

DEFINIZIONE: Un'operazione atomica, in informatica, consiste in un'operazione di esecuzione indivisibile dal punto di vista logico.

In generale quindi, un'operazione si dice atomica se è indivisibile, ovvero se nessun'altra operazione può cominciare prima che la prima sia finita, e quindi non può esserci *interleaving*. Il risultato di quella operazione è sempre lo stesso se parte dalle stesse condizioni iniziali.

Modifichiamo quindi la classe *Contatore* nel modo seguente:

```
import lombok.Getter;

public class Contatore {

  @Getter
  private int contatore = 0;
  private Object lock = new Object();

  public void update() {
    synchronized(lock){
      contatore++;
    }
  }
}
```

La modifica farà si che solo un singolo thread può essere eseguito all'interno del blocco in un dato momento, evitando quindi una situazione in cui due thread leggono e modificano il valore del contatore in concorrenza. Qualunque thread verrà eseguito leggerà il valore, lo incrementerà e lo riscriverà come una singola operazione atomica.

Per il momento accontentiamoci della prova del nove. Se infatti andiamo ad eseguire la nostra applicazione adesso otteniamo il risultato aspettato:

```
esecuzione 1: 2000000
esecuzione 2: 2000000
esecuzione 3: 2000000
```

E' tutta una questione di ... visibilità

Abbiamo detto che nelle moderne architetture multiprocessore a memoria condivisa, ogni processore ha uno o più livelli di cache che vengono periodicamente riconciliati con la memoria principale come mostrato nella figura *Immagine 62 - architetture multiprocessore*: il valore di una variabile condivisa può essere memorizzato nella cache e la scrittura del suo valore nella

memoria principale (RAM) può essere ritardata. Di conseguenza, un altro thread potrebbe leggere un valore obsoleto della variabile.

Abbiamo quindi un problema di *visibilità della memoria* ovvero, quando la lettura e la scrittura di variabili condivise avviene in due thread differenti, nessuno assicura che il thread lettore possa vedere le modifiche fatte dal thread che accede in scrittura. In generale, non vi è nessuna garanzia che il thread in lettura veda un valore scritto da un altro thread in modo tempestivo, e addirittura, non vi è garanzia alcuna che possa mai vederlo. il risultato è una *race-condition*.

Come affrontiamo il problema della *visibilità*?

Java mette a disposizione diversi strumenti:

1. La parola chiave synchronized;

Assicura che un thread abbia sbloccato una sezione critica prima che un altro thread possa accedervi e richiedere un altro blocco.

*2. La parola chiave **volatile**;*

Assicura che una scrittura su una variabile avvenga prima delle successive letture.

3. L'inizializzazione statica;

Garantisce che l'inizializzazione di una classe avvenga una volta sola al momento del primo accesso alla classe quando viene caricata in memoria dal *classloader* di Java.

Nei prossimi paragrafi analizzeremo in dettaglio i vari metodi di sincronizzazione.

La parola chiave synchronized

Riprendiamo in considerazione la classe *ContoCorrente* definita come segue nei paragrafi precedenti:

```
public class ContoCorrente {
  private double saldo;

  public double getSaldo(){
    return saldo;
  }

  public void deposita(double sommaDaDepositare){
    saldo = saldo+sommaDaDepositare;
  }

  public double preleva(double sommaDaPrelevare){
    if(saldo >= sommaDaPrelevare){
      saldo = saldo -sommaDaPrelevare;
      return sommaDaPrelevare;
    }
    return 0;
  }
```

```
}
```

La sezione critica per questa classe è rappresentata dai metodi che accedono in scritture alla variabile condivisa saldo: *preleva(double sommaDaPrelevare)* e *deposita(double sommaDaDepositare)*. Per i motivi citati in precedenza la classe non è thread-safe, l'accesso a questi metodi è causa di race-condition.

In questi casi è necessario che i due thread siano sincronizzati ovvero, mentre uno esegue l'operazione, l'altro deve rimanere in attesa. Java fornisce il metodo per gestire la sincronizzazione tra thread mediante la parola chiave **synchronized**.

Questo modificatore se aggiunto alla dichiarazione del metodo da sincronizzare assicura che, solo un thread alla volta sarà in grado di eseguire il metodo. La sintassi è la seguente:

```
[modificatori] synchronized tipo_di_ritorno nome(lista_parametri_formali ){
        istruzione
        [istruzione]
}
```

Possiamo quindi riformulare la definizione della classe *ContoCorrente* nella sua forma thread-safe nel modo seguente:

```
public class ContoCorrente {
  private double saldo;

  public double getSaldo(){
    return saldo;
  }
  public synchronized void deposita(double sommaDaDepositare){
    saldo = saldo+sommaDaDepositare;
  }

  public synchronized double preleva(double sommaDaPrelevare){
    if(saldo >= sommaDaPrelevare){
      saldo = saldo -sommaDaPrelevare;
      return sommaDaPrelevare;
    }
    return 0;
  }
}
```

Synchronized: lock su oggetto

Il meccanismo appena descritto descritto non ha come unico effetto quello di impedire che due thread accedano ad uno stesso metodo contemporaneamente, ma impedisce il verificarsi di situazioni anomale tipiche della programmazione concorrente. Per meglio comprendere consideriamo il frammento di codice a seguire:

```
class A{
```

```
        public synchronized int a(){
                return b()
        }
        public synchronized int b(int n){
                return a();
        }
}
```

Supponiamo ora che un thread T1 esegua il metodo b() della classe A e contemporaneamente, un secondo thread T2 effettua una chiamata al metodo a() dello stesso oggetto. Ovviamente, essendo i due metodi sincronizzati, il primo thread avrebbe il controllo sul metodo b() ed il secondo su a(). Lo scenario che si verrebbe a delineare è potenzialmente disastroso:

T1 rimarrebbe in attesa sulla chiamata al metodo a() sotto il controllo di T2 e viceversa, T2 rimarrebbe bloccato sulla chiamata al metodo b() sotto il controllo di T1.

Il risultato finale sarebbe lo stallo di entrambi i thread. Questa situazione, detta *deadlock*, è difficilmente rilevabile e necessita di algoritmi molto complessi e poco efficienti per essere gestita o prevenuta. Java quindi garantisce al programmatore la certezza che casi di questo tipo non avvengono mai. Di fatto, quando un thread entra all'interno di un metodo sincronizzato ottiene il lock sull'oggetto (non solo sul metodo). Ottenuto il lock, il thread potrà richiamare altri metodi sincronizzati dell'oggetto senza entrare in *deadlock*.

Quindi un thread che detiene il blocco può accedere a un altro metodo sincronizzato senza dover acquisire un blocco diverso. Ogni altro thread che proverà ad utilizzare l'oggetto in lock, si metterà in coda in attesa di essere risvegliato al momento del rilascio della risorsa da parte del thread proprietario. Quando il primo thread termina le sue operazioni, il secondo otterrà il lock e di conseguenza l'uso privato dell'oggetto.

Formalmente, lock di questo tipo sono chiamati *lock su oggetto*.

Synchronized: Lock su classe

Differenti dai *lock su oggetto* sono invece i *lock su classe*. Vediamo di cosa si tratta.

Sappiamo che metodi statici accedono solo a dati membro statici e che un metodo o un attributo statico non richiedono un oggetto attivo per essere eseguiti. Di conseguenza, un thread che effettui una chiamata ad un metodo statico sincronizzato non può ottenere il lock sull'istanza di un oggetto inesistente.

D'altra parte, è necessaria una forma di prevenzione dei problemi relativi alla concorrenza anche in questo caso. Java prevede una seconda forma di lock: i*l lock su classe*. Quando un thread accede ad un metodo statico sincronizzato, ottiene un lock su classe: in questo scenario, nessun thread potrà accedere a metodi statici sincronizzati di ogni oggetto di una stessa classe fino a che un thread detiene il lock sulla classe. Una volta che un thread ha ottenuto il blocco a livello di classe, è consentito eseguire qualsiasi metodo sincronizzato statico di quella classe. Una volta completata l'esecuzione del metodo, il thread rilascia automaticamente il blocco.

Questa seconda forma di lock nonostante riguardi tutte le istanze di un oggetto, è comunque meno restrittiva della precedente perché metodi sincronizzati non statici della classe in lock possono essere eseguiti durante l'esecuzione del metodo statico sincronizzato.

Synchronized: blocchi sincronizzati

Nei paragrafi precedenti abbiamo visto come i metodi sincronizzati acquisiscano un lock su un oggetto, o nel caso di lock su classe, su tutti quelli di un determinato tipo. L'ambito in cui un lock imposto con **synchronized** ha effetto (classe o oggetto) è anche detto *scope del lock*;

Se il metodo sincronizzato viene eseguito molte volte e da molti thread, può rappresentare un collo di bottiglia in grado di deteriorare pesantemente le prestazioni di tutta l'applicazione.

In generale, è quindi buona regola limitare le funzioni sincronizzate al minimo indispensabile ovvero, ridurre al minimo lo *scope del lock*. Per questo motivo, la parola chiave **synchronized** può essere utilizzata anche semplicemente per definire, all'interno di un metodo, il blocco si codice da sincronizzare.

Nel prossimo frammento di pseudo codice, utilizziamo **synchronized** per definire un blocco sincronizzato limitando al minimo lo *scope del lock*:

```
Object lock = new Object();
public void criticalMethod() {
        nonCriticalSection_1();
        synchronized(lock) {
                criticalSection();
                }
        nonCriticalSection_2();
}
```

Notiamo immediatamente che ne risulterà uno scope generalmente più piccolo rispetto alla sincronizzazione dell'intero metodo.

Un esempio lo abbiamo già visto quando abbiamo sincronizzato la nostra classe *Contatore*. Riporto nuovamente la definizione per semplicità:

```
public class Contatore {

  @Getter
  private int contatore = 0;
  private Object lock = new Object();

  public void update() {
    synchronized(lock){
      contatore++;
    }
  }
}
```

La parola chiave volatile

La parola chiave **volatile** è di solito associata ad una variabile il cui valore viene salvato e ricaricato ad ogni accesso nella memoria principale senza utilizzare i meccanismi di *caching*. Per meglio comprenderne il funzionamento consideriamo il codice seguente:

```
class TestVolatile extends Thread {
 //volatile
 boolean  running = true;
 public void run() {
   long count=0;
   while (running) {
     count++;
   }
   System.out.println("Thread terminato:" + count);
 }

 public static void main(String[] args) throws InterruptedException {
   TestVolatile t = new TestVolatile();
   t.start();
   Thread.sleep(1000);
   System.out.println("Adesso metto volatile a false");
   t.running = false;
   t.join();
   System.out.println("running adesso vale " + t.running);
 }
}
```

Nell'esempio, l'applicazione *TestVolatile* esegue un thread che dovrebbe uscire non appena la variabile *running* viene messa a false provocando l'uscita dal ciclo **while** del thread. Tuttavia, se eseguiamo la applicazione quello che succede è che viene stampata la stringa '*Adesso metto volatile a false*', e tuttavia il thread continua l'esecuzione senza essere terminato come ci si aspetterebbe.

Cosa è successo?

1. Il primo thread carica in cache il valore della variabile booleana running (impostato a true);

2. Il primo thread continua a leggere il valore in cache della variabile running e non va più a leggere il valore effettivo (quello nella memoria principale) dopo che ...

3. il secondo thread (main thread) lo modifica a false;

4. A causa di questo problema con la visibilità della variabile running, il thread prosegue all'infinito senza terminare.

Quello che accade è schematizzato nella prossima figura:

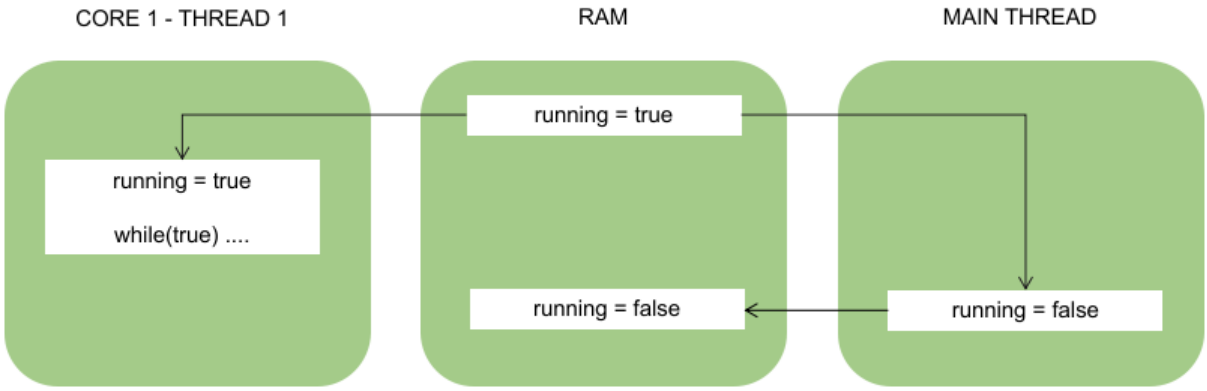


Immagine 65 - visibilità della variabile running

Se però modifichiamo la variabile running utilizzando volatile nel modo seguente:

```
//volatile
volatile boolean  running = true;
```

l'applicazione si comporterà come aspettato e l'output sarà il seguente:

```
Adesso metto volatile a false
Thread terminato: 1593316206
running adesso vale false
```

Dichiarando **volatile** la variabile running costringiamo il thread ad aggiornare la variabile nella memoria principale senza spostarla nella cache del processore. In questo modo il primo Thread termina correttamente.

La sintassi per dichiarare una variabile **volatile** è la seguente:

***volatile** [modificatore] tipo identificatore;*

Valgono per volatile se regole seguenti:

*1. Possiamo dichiarare **volatile** solo variabili membro di una classe (di istanza o statiche). Una variabile locale non può essere dichiarata **volatile** perché lo scope di una variabile locale è limitata al metodo e non è mai condivisa con altri thread.*

*2. Una variabile **volatile** non può essere dichiarata **final**. Di fatto, **volatile** è sempre associato ad una variabile che cambia valore e mai ad una costante. Dichiarare **volatile** una variabile **final** provocherà un errore in fase di compilazione.*

*3. Classi e metodi non possono essere dichiarati **volatile**. Farlo provocherà un errore in fase di compilazione.*

*4. Tutte le operazioni di lettura/scrittura su una variabile **volatile** vengono effettuate nella memoria principale e non nella cache del processore. Le operazioni di lettura/scrittura vengono eseguite in maniera atomica.*

5. Una variabile volatile di tipo reference può puntare ad un oggetto null. Ad esempio

***volatile** Integer i = null;*

è consentito.

Volatile quindi fornisce un altro meccanismo per la sincronizzazione di thread quando accedono ad una variabile statica o ad una variabile di istanza. Tuttavia **volatile** non sostituisce completamente la parola chiave **synchronized**. Vediamo perché, e per farlo ritorniamo a considerare l'applicazione

```
public class EsempioRaceCondition {

  private static Runnable getRunnable(Contatore contatore) {
    return () -> {
      for (int i = 0; i < 1_000_000; i++) {
        contatore.update();
      }
    };
  }

  public static void main(String[] args) {
    Contatore contatore = new Contatore();
    Thread threadUno = new Thread(getRunnable(contatore));
    Thread threadDue = new Thread(getRunnable(contatore));

    threadUno.start();
    threadDue.start();
    try {
      threadUno.join();
      threadDue.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Contatore vale: "+contatore.getContatore());
  }
}
```

Riscriviamo però la classe Contatore come segue utilizzando **volatile** invece che **synchronized**:

```
import lombok.Getter;

public class Contatore {
  @Getter
  volatile private int contatore = 0;
  public void update() {
    contatore++;
  }
}
```

Se andiamo ad eseguire l'applicazione le cose non andranno come aspettato ed il risultato finale non è quello aspettato: il problema della visibilità della variabile *contatore* non è risolta e la applicazione continua a mostrare effetti indesiderati dovuti alla *race-condition* che si verifica poiché due thread accedono alla variabile in concorrenza.

Il motivo di questo è che la parola chiave **volatile** fornisce un meccanismo di sincronizzazione tra thread di tipo *lock-free*; nonostante **volatile** il thread che accede alla variabile non acquisisce nessun lock. Inoltre, come abbiamo detto, **volatile** garantisce che solo le operazioni di *lettura* e *scrittura* vengano effettuate in modo *atomico*; poiché aggiornare una variabile con ++ non è una semplice operazione di scrittura ma una *lettura-incremento-scrittura* **allora** volatile non è sufficiente a garantire che l'operazione sia svolta gestendo la concorrenza in maniera corretta.

Prima di completare questo paragrafo, è necessario fare qualche considerazione relative alle prestazioni. Lettura e scrittura di una variabile **volatile** avvengono nella memoria principale, e sono quindi più costose delle analoghe funzioni se fatte all'interno della cache del processore. Inoltre, l'accesso alle variabili volatile previene il meccanismo di *instruction reordering*, una tecnica utilizzata dalla JVM per migliorare le prestazioni del codice. In definitiva quindi, le variabili **volatile** dovrebbero essere utilizzate solamente se strettamente richiesto per risolvere il problema della visibilità di una variabile.

Solo a titolo di esempio, la Java VM e la CPU possono riordinare le istruzioni di un programma (*instruction reordering*) per motivi di prestazioni, purché il significato semantico delle istruzioni rimanga lo stesso.

Ad esempio le seguenti istruzioni:

```
int a = 1;
int b = 2;
a++;
b++;
```

potrebbero essere riscritte nel modo seguente senza perdere il significato semantico del codice:

```
int a = 1;
a++;
```

```
int b = 2;
b++;
```

Il problema del produttore e consumatore

Quando un thread deve spettare che accada una determinata condizione esterna per proseguire nell'esecuzione, il modificatore **synchronized** non è più sufficiente. Il problema del *Produttore/Consumatore* rappresenta un caso tipico in cui il modificatore **synchronized** non è sufficiente a rendere consistente la concorrenza.

Il problema del produttore-consumatore, già introdotto nella lezione precedente, è caratterizzato da due task, il produttore e il consumatore, che comunicano attraverso un buffer, il quale ha una capacità sconosciuta:

1. *il produttore genera dati e li deposita nel buffer;*

2. *il consumatore utilizza i dati prodotti, rimuovendoli di volta in volta dal buffer;*

3. *i due task producono e consumano continuamente, ma con una frequenza variabile e non nota a priori (quindi la soluzione non si potrà basare su una conoscenza di quando i due task effettueranno le loro operazioni).*

Nella nostra prima implementazione, produttore e consumatore sono rappresentati da due thread che condividono i dati mediante la classe *Magazzino*. Per rendere la simulazione più reale, faremo in modo che il *Produttore* sifermi per un periodo compreso tra 0 e 100 millisecondi prima di generare un nuovo elemento. Il codice delle tre definizioni di classe è il seguente.

```
public class Magazzino {
  private int prodotto;
  public synchronized int getProdotto() {
    return prodotto;
  }
  public synchronized void setProdotto(int prodotto) {
    this.prodotto = prodotto;
  }
}


public class Produttore implements Runnable {
  private Magazzino magazzino;

  public Produttore(Magazzino magazzino) {
    this.magazzino = magazzino;
  }

  public void run() {
    for (int i = 0; i < 5; i++) {
```

```
        magazzino.setProdotto(i);
        System.out.println("Produttore ha inserito: " + i);
        try {
          Thread.sleep((int) (Math.random() * 100));
        } catch (InterruptedException e) {
        }
      }
    }
}
```

Ed infine, la classe *Consumatore*:

```
public class Consumatore implements Runnable {
  private Magazzino magazzino;

  public Consumatore(Magazzino magazzino) {
    this.magazzino = magazzino;
  }

  public void run() {
    int prodotto = 0;
    for (int i = 0; i < 5; i++) {
      prodotto = magazzino.getProdotto();
      System.out.println("Consumatore ha estratto: " + prodotto);
    }
  }
}
```

La classe magazzino, nonostante sia sincronizzata affinché *Produttore* e *Consumatore* non le accedano contemporaneamente, non consente di sincronizzare i due thread affinché:

1 . Il consumatore aspetti che il produttore abbia generato un nuovo numero intero;

2. Il produttore aspetti che il consumatore abbia utilizzato l'ultimo intero generato.

Quanto detto sarà ancora più evidente dopo che avremo eseguito l'applicazione *ProduttoreConsumatore* il cui codice è il seguente:

```
public static void main(String[] argv) {
  Magazzino magazzino = new Magazzino();
  Thread produttore = new Thread(new Produttore(magazzino));
  Thread consumatore = new Thread(new Consumatore(magazzino));
  produttore.start();
  consumatore.start();
}
```

La cui esecuzione produce il seguente risultato:

Consumatore ha estratto: 0
Produttore ha inserito: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Consumatore ha estratto: 0
Produttore ha inserito: 1
Produttore ha inserito: 2
Produttore ha inserito: 3

Condizioni di questo tipo possono essere risolte solo se il Produttore segnala al Consumatore che ha generato un nuovo elemento ed è disponibile per essere utilizzato, viceversa, il Consumatore segnala al Produttore che è in attesa di un nuovo elemento da utilizzare.

La classe *Object* dispone dei metodi *wait*, *notify* e *notifyAll* che consentono di indicare ad un oggetto che deve aspettare in attesa di una condizione o, di segnalare che una condizione si è verificata.

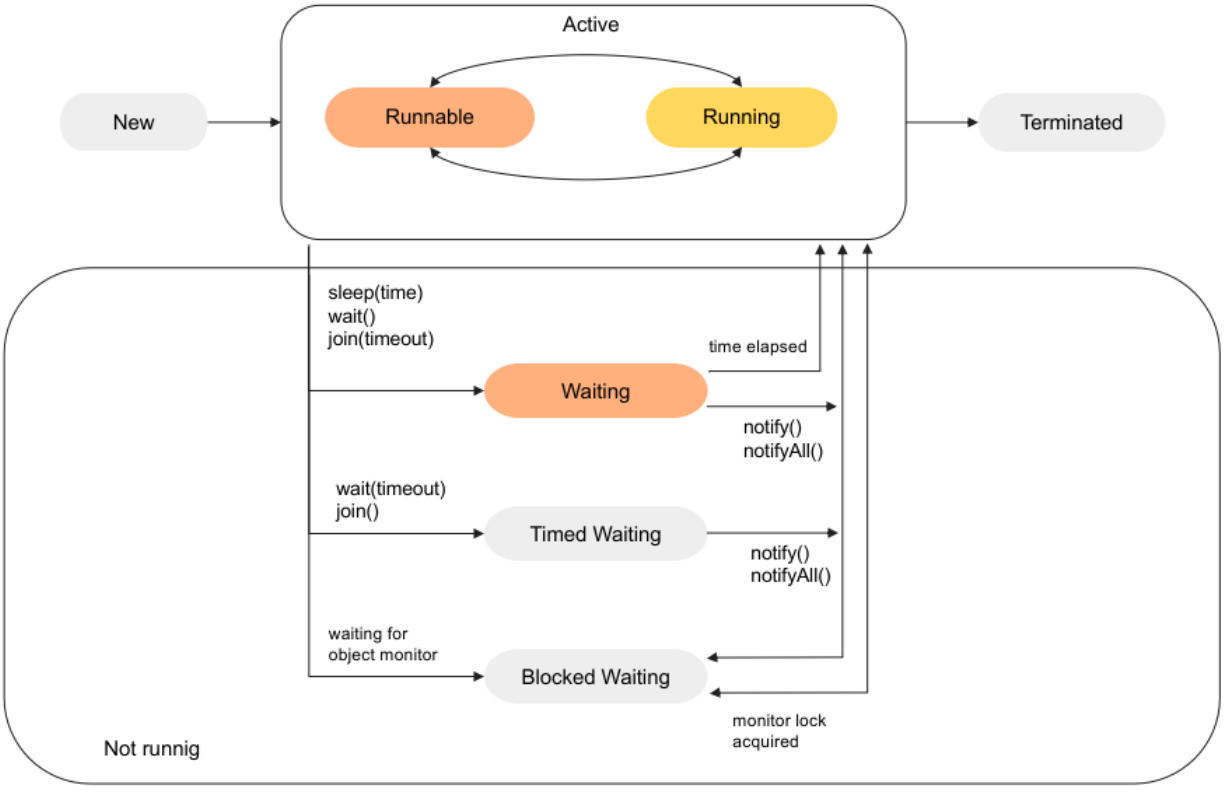


Immagine 66 Produttore e consumatore - wait & notify

Attraverso i metodi *wait* e *notifyAll* produttore e consumatore potranno sincronizzare la propria attività: il consumatore potrà entrare in uno stato *Waiting* chiamando il metodo wait, il produttore potrà notificare al consumatore quando è stato immagazzinato un nuovo elemento da consumare mediante il metodo *notifyAll*.

Produttore e consumatore con wait e notifyAll

Di seguito, una nuova versione della classe Magazzino i cui metodi *getProdotto* e *setProdotto* sono stati modificati utilizzando i metodi *wait()* e *notifyAll()* ereditati dalla classe Object. nella nuova versione della classe, il metodo *getProdotto()* entra in un ciclo attendendo che il produttore abbia generato un nuovo elemento.

Il meccanismo è il seguente:

1. consumatore esegue il metodo sincronizzato getProdotto ed acquisisce il lock sull'oggetto. Non trovando elementi da consumare entra nel blocco if e passa in stato di Waiting rilasciando il lock su oggetto per consentire ad altri thread di accedere.

2. Il thread produttore entra nel metodo sincronizzato setProdotto, inserisce un nuovo elemento in magazzino e chiama notifyAll per notificare a tutti i thread in attesa che ci sono elementi da consumare.

3. Il thread consumatore recupera elemento dal magazzino e si rimette in attesa del prossimo.

```
@Data
public class Magazzino {
  private int prodotto;
  private boolean disponibile;
  public synchronized int getProdotto() {
    if(isDisponibile() == false) {
      try {
        // aspetta che il produttore abbia
        // generato un nuovo elemento
        wait();
      } catch (InterruptedException e) {
      }
    }
    setDisponibile(false);
    // notifica al produttore di aver
    // utilizzato l'ultimo elemento generato
    notifyAll();
    return prodotto;
  }

  public synchronized void setProdotto(int prodotto) {
    this.prodotto = prodotto;
    setDisponibile(true);
    // notifica al consumatore di aver
    // ugenerato un nuovo elemento
    notifyAll();
  }
 }
}
```

L'esecuzione mostra come i thead adesso siano sincronizzati:

```
Consumatore ha estratto: 0
Produttore ha inserito: 0
Produttore ha inserito: 1
Consumatore ha estratto: 1
Produttore ha inserito: 2
Consumatore ha estratto: 2
Produttore ha inserito: 3
Consumatore ha estratto: 3
Produttore ha inserito: 4
Consumatore ha estratto: 4
```

Potrebbe accadere che la classe *Produttore* produca più elementi di quelli che la classe *Consumatore* riesce a consumare. Potremmo quindi pensare che diverse istanza di *Consumatore* possano accedere a *Magazzino*, e per evitare colli di bottiglia potremmo modificare la classe Magazzino affinché gestisca gruppi di elementi, e non un solo elemento.

Proviamo a modificare la classe *Magazzino*:

```
public class Magazzino {
  private List<Integer> prodotti = new ArrayList<>();

  public synchronized int getProdotto() {
    if (prodotti.size() ==0) {
      try {
        // aspetta che il produttore abbia
        // generato un nuovo elemento
        wait();
      } catch (InterruptedException e) {
      }
    }
    // notifica al produttore di aver
    // utilizzato l'ultimo elemento generato
    notifyAll();
    return prodotti.remove(0);
  }

  public synchronized void setProdotto(int prodotto) {
    prodotti.add(prodotto);
    // notifica al consumatore di aver
    // ugenerato un nuovo elemento
    notifyAll();
  }
}
```

Se proviamo ad eseguire tutto funzionerà correttamente fino a che non modifichiamo il metodo *main* nel modo seguente:

```
public static void main(String[] argv) {
   Magazzino magazzino = new Magazzino();
   Thread produttore = new Thread(new Produttore(magazzino));
   Thread consumatore = new Thread(new Consumatore(magazzino));
   consumatore.setName("Consumatore 1");
   Thread consumatore2 = new Thread(new Consumatore(magazzino));
   consumatore2.setName("Consumatore 2");
   produttore.start();
   consumatore.start();
   consumatore2.start();
}
```

Non appena eseguito l'applicazione restituisce l'errore seguente:

```
Consumatore 1 ha estratto: 0
Produttore ha inserito: 0
Produttore ha inserito: 1
Consumatore 2 ha estratto: 1
Exception in thread "Consumatore 1" Exception in thread "Consumatore 2"
java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Unknown Source)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Unknown Source)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Unknown Source)
    at java.base/java.util.Objects.checkIndex(Unknown Source)
    at java.base/java.util.ArrayList.remove(Unknown Source)
....
```

Il meccanismo proposto non è più sufficiente, e non appena più thread tentano di accedere al magazzino il risultato è l'eccezione appena vista. Ovviamente possiamo tentare di risolvere l'errore magari gestendo l'eccezione all'interno del metodo *run* dei thread consumatori, tuttavia il *Java Collection Framework* mette a disposizione una serie di strutture dati bloccanti che consentono di risolvere facilmente il problema.

L'interfaccia BlockingQueue

Il package *java.util.concurrent* contiene la definizione di una serie di classi nate per risolvere il problema del produttore e consumatore. L'interfaccia *BlockingQueue* estende l'interfaccia *Queue* già vista in precedenza.

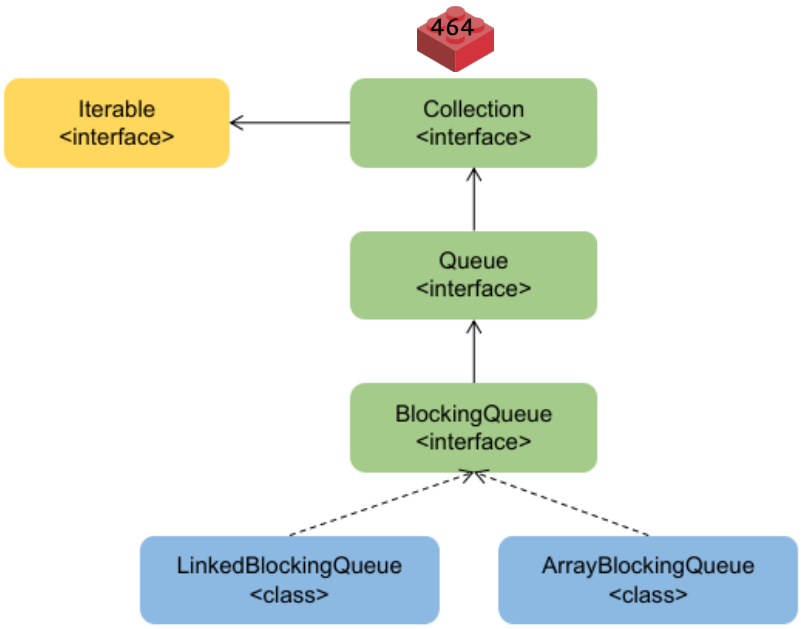


Immagine 67 Interfaccia BlockingQueue

Oltre a supportare tutte le funzioni previste per l'interfaccia *Queue*, *BlockingQueue* supporta la sincronizzazione dei thread attraverso metodi bloccanti, ovvero metodi che bloccano i thread quando necessario a garantire una corretta gestione della concorrenza.

Prima di procedere dobbiamo però fare una distinzione. Esistono due tipo di *BlockingQueue*:

1. *Unbounded Queues.*

Sono le code che possono crescere illimitatamente; se la coda è vuota bloccano i thread consumatori finché non vengono aggiunti nuovi elementi. Sincronizzano i thread gestendone il controllo di flusso.

Quando si progetta una applicazione produttore-consumatore utilizzando *BlockingQueue* a dimensione illimitata, è importante fare in modo che i consumatori possano essere in grado di consumare gli elementi accodati con la stessa rapidità con cui i produttori aggiungono elementi alla coda. In caso contrario, la memoria potrebbe riempirsi e otterremmo un'eccezione *OutOfMemory*. Per limitare l'eventuale problema possiamo utilizzare il secondo tipo di code:

2. Bounded Queues.

Le code con una capacità massima definita al momento della creazione della loro creazione. Con una coda di questo tipo, quando un produttore cerca di aggiungere un elemento alla coda già piena, indipendentemente dal metodo utilizzato per aggiungere l'elemento, sarà bloccato in attesa che un consumatore faccia spazio liberando elementi dalla coda. Sono utilissime in tutte quelle situazioni in cui introdurre una *strozzatura* può far comodo ai fini del tuning applicativo.

Oltre ai metodi non bloccanti ereditati da *Queue*, *BlockingQueue* mette a disposizione due diversi tipi di metodi bloccanti: il primo tipo mette in blocco un thread indefinitamente; il secondo tipo si sblocca alla scadenza di un timeout restituendo *false* o **null** a seconda del caso.

Tutti i metodi sono rappresentati nella prossima tabella: l eprime due colonne riguardano i metodi non bloccanti, le ultime due quelli bloccanti.

	Metodi non bloccanti		Metodi bloccanti	
			Blocco indefinito	**Blocco con timeout**
Aggiungere elementi	*add(E e)*	*offer(E e)*	*put(E e)*	*offer(E e, long timeout, TimeUnit unit)*
Rimuovere elementi	*remove()*	*poll()*	*take()*	*poll(long timeout, TimeUnit unit)*
Esaminare elementi	*element()*	*peek()*	*n/a*	*n/a*

In dettaglio:

Metodi bloccanti di BlockingQueue
void put(E e) throws InterruptedException
Il metodo *put* inserisce un elemento nella coda se lo spazio è disponibile. Tuttavia, se viene raggiunto il limite di capacità della coda, il metodo si blocca finché non viene liberato spazio.
boolean offer(E e,long timeout,TimeUnit unit) throws InterruptedException
Inserisce un elemento se c'è ancora spazio nella coda. In caso contrario, il metodo attende il tempo specificato. Se uno spazio diventa disponibile durante questo periodo, l'elemento viene inserito e il metodo restituisce *true*. Se invece il tempo di attesa scade senza che venga liberato spazio, il metodo restituisce *false*.
E take() throws InterruptedException
Questo metodo prende un elemento dall'inizio della coda, a condizione che la coda non sia vuota. Se la coda è vuota, take() si blocca finché un elemento non diventa disponibile e quindi lo restituisce.
E poll(long timeout, TimeUnit unit) throws InterruptedException
Se la coda non è vuota, prende un elemento dalla cima della coda. Se la coda è vuota, il metodo attende il tempo specificato. Se un elemento diventa disponibile durante il tempo di attesa, viene restituito. Se il tempo di attesa scade senza risultato, il metodo restituisce **null**.

Andiamo finalmente a riscrivere l'applicazione produttore-cosumatore utilizzando *BlockingQueue*. Prima di tutto partiamo dalla definizione di *Magazzino* per poi proseguire con la definizione delle classi *Produttore* e *Consumatore*.

Non dovendoci preoccupare della gestione della concorrenza, utilizzando Lombok la classe Magazzino può essere definita come segue:

```
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Magazzino {
  @Getter
  @Builder.Default
  private BlockingQueue<Integer> magazzino = new LinkedBlockingQueue<>();
}
```

Implementa il pattern *builder* ed ha un solo metodo *getter* che ritorna l'unico membro che rappresenta il magazzino di tipo *BlockingQueue*.

Le classi *Produttore* e *Consumatore* sono le seguenti:

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Produttore implements Runnable {
  private BlockingQueue magazzino;
  @Override
  public void run() {
    for (int i = 0; i < 5; i++) {
      try {
        magazzino.put(i);
        System.out.println("Produttore ha inserito: " + i);
        Thread.sleep((int) (Math.random() * 100));
      } catch (InterruptedException e1) {
        Thread.currentThread().interrupt();
        break;
      }
    }
  }
}
```

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Consumatore implements Runnable{
  private BlockingQueue magazzino;
  @Override
  public void run() {
    int prodotto = 0;
    while (!Thread.currentThread().isInterrupted()) {
      try {
        prodotto = (int) magazzino.take();
```

```
            System.out.println(Thread.currentThread().getName() + " ha estratto: " + prodotto);
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
          }

        }
      }
    }
```

Ed infine la nostra applicazione:

```
  public static void main(String[] args) {
    Magazzino magazzino = Magazzino.builder().build();
    Thread threadProduttore = new Thread(
        Produttore.builder().magazzino(magazzino.getMagazzino()).build());
    Thread threadConsumatore1 = new Thread(
      Consumatore.builder().magazzino(magazzino.getMagazzino()).build(),
      "Primo consumatore");
    Thread threadConsumatore2 = new Thread(
      Consumatore.builder().magazzino(magazzino.getMagazzino()).build(),
      "Secondo consumatore");
      threadConsumatore1.start();
      threadConsumatore2.start();
      threadProduttore.start();

  }
```

L'esecuzione questa volta ritorna l'output atteso

```
Produttore ha inserito: 0
Primo consumatore ha estratto: 0
Produttore ha inserito: 1
Secondo consumatore ha estratto: 1
Produttore ha inserito: 2
Primo consumatore ha estratto: 2
Produttore ha inserito: 3
Secondo consumatore ha estratto: 3
Produttore ha inserito: 4
Primo consumatore ha estratto: 4
```

21.Classloader Java

Introduzione

Il *Java classloader* fa parte di *Java Runtime Environment* ed è responsabile del caricamento dinamico delle classi java richieste dalla *Java Virtual Machine* per eseguire una applicazione.

Grazie al *classloader*, il sistema run-time di Java non ha bisogno di conoscere il *filesystem*, ma demanda al class-loader la responsabilità di reperire e caricare le classi necessarie al funzionamento della applicazione. Le classi Java non vengono caricate in memoria tutte in una volta, ma solo quando richiesto da un'applicazione. Il meccanismo di caricamento su richiesta delle classi è chiamato *dynamic loading and linking*.

Non tutte le classi sono caricate dallo stesso classloader: il *Java Runtime Environment* utilizza diversi class-loader per compiere compiti diversi; Java consente inoltre di creare nuovi class-loader specifici qualora il disegno applicativo lo preveda.

Tipi di class-loader integrati in Java

Nella prossima applicazione utilizziamo il metodo *getClassLoader* per ottenere un riferimento al *classloader* di alcune classi:

```
public class ClassLoadersIntegrati {
  public void stampa() throws ClassNotFoundException {

    System.out.println("Classloader di questa classe:"
      + ClassLoadersIntegrati.class.getClassLoader());

    System.out.println("Classloader di Logging:"
      + Logging.class.getClassLoader());

    System.out.println("Classloader di ArrayList:"
      + ArrayList.class.getClassLoader());
  }

  public static void main(String[] args) {
    ClassLoadersIntegrati classloaders = new ClassLoadersIntegrati();
    classloaders.stampa();

  }
}
```

Se adesso eseguiamo l'applicazione otteniamo il seguente risultato:

```
Classloader di questa classe:sun.misc.Launcher$AppClassLoader@18b4aac2
Classloader di Logging::sun.misc.Launcher$ExtClassLoader@3caeaf62
Classloader di ArrayList::null
```

Come notiamo dall'output della applicazione esistono tre differenti *classloader*: *AppClassLoader*, *ExtClassLoader*, *Bootstrap ClassLoader* che nel compare come **null**.

1. Application classloader

Responsabile del caricamento delle classi che compongono l'applicazione nella JVM. Carica i file trovati nella variabile d'ambiente CLASSPATH oppure specificati dall'opzione *-classpath o -cp* della riga di comando.

E' definito a partire dalla classe *java.lang.ClassLoader* quindi è esso stesso un oggetto di cui possiamo ottenere il riferimento utilizzando il metodo *getClassLoader* della classe *Class*. È figlio del *extension classloaderi*.

2. Extension classloader

E' responsabile di caricare le classi appartenenti alle estensioni java presenti nella clartella *...jre/lib/ext* o da qualsiasi altra cartella specificata nella proprietà di sistema *java.ext.dirs*. Anche questo classloader è definito a partire dalla classe *java.lang.ClassLoader.* E' figlio del *Bootstrap Classloader*.

3. Bootstrap classloader

Se i precedenti due classloader sono classi che necessitano di essere caricate nella memoria della *Java Virtual Machine*, richiedono a loro volta un *classloader* responsabile del loro caricamento. *Bootstrap classloader* è scritto in linguaggio nativo ed è parte della JVM; da qui il motivo per cui compare cone **null** nell'output dell'esempio precedente.

È il principale responsabile del caricamento delle classi interne al JDK in *rt.jar,* e delle altre librerie principali situate nella directory $JAVA_HOME/jre/lib. Funge da padre di tutte le altre istanze di classloader.

I classloader quindi formano una gerarchia padre-figlio (da non confondersi con la gerarchia legata alla ereditarietà) rappresentata nella prossima immagine.

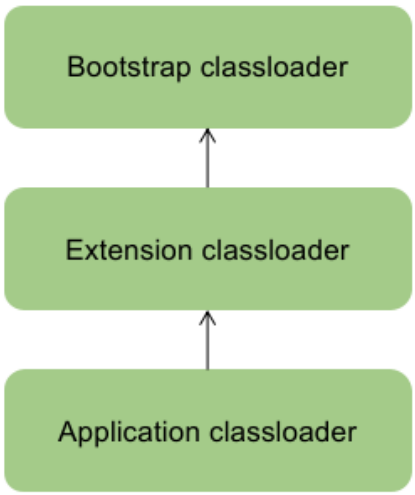


Immagine 68 Gerarchia dei classloader

Come lavorano i classloader: il meccanismo della delega

Lo scopo del *classloader* è quello di caricare una classe quando viene richiesto dalla *Java Virtual Machine* assicurando, allo stesso tempo, che una classe non venga caricata più di una volta da differenti classloader.

Visto che i tre classloader hanno compiti diversi, nasce spontaneo chiedersi perché dovrebbero caricare contemporaneamente la stessa classe. La risposta è rimandata ai paragrafi successivi in cui parleremo della creazione di classloader personalizzati.

L'algoritmo di funzionamento, basato sul modello della *delega*, è schematizzato nella prossima immagine.

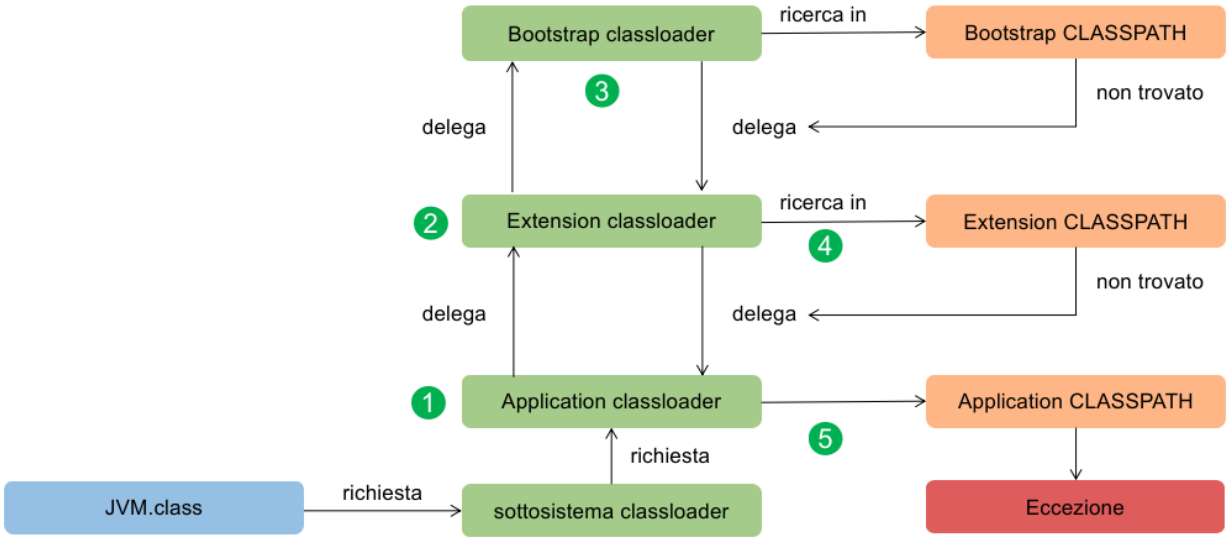


Immagine 69 Classloader - meccanismo della delega

1. Quando la JVM deve utilizzare una classe, invia la richiesta al sottosistema dei classloader che invia la richiesta all'application classloader. Se la classe è già stata caricata viene restituita altrimenti il classloader delega la richiesta all'extension classloader;

2. Extension classloader si comporta allo stesso modo, controlla se ha già caricato la classe richiesta, e nel caso delega a sua volta la richiesta al padre: bootstrap classloader.

3. Se la classe già si trova nella memoria del bootstrap classloader viene tornata, altrimenti, a differenza degli altri casi, il classloader cerca di caricarla. Se non la trova torna la delega al figlio;

4. Extension classloader tenta quindi di caricare la classe, e se non la trova torna la delega all'application classloader.

5. Application classloader tenta di caricare la classe, e se neanche l'ultimo classloader non è in grado di farlo, viene generata una eccezione di tipo java.lang.NoClassDefFoundError oppure *java.lang.ClassNotFoundException.*

Come conseguenza del meccanismo appena descritto vale quanto segue:

DEFINZIONE: Principio di visibilità: il principio di visibilità afferma che una classe caricata da un classloader padre è visibile ai classloader figlio, ma una classe caricata da un classloader figlio non è visibile ai classloader padre.

DEFINZIONE: Proprietà di unicità: il meccanismo della delega garantisce che le classi siano univoche e non vi sia alcuna ripetizione delle classi.

Questo assicura anche che le classi caricate dai classloader padre non vengano caricate dai classloader figli. Se il caricatore di classe genitore non è in grado di trovare la classe, solo allora l'istanza corrente tenterà di farlo da sola.

La classe java.lang.ClassLoader

Java fornisce alcuni implementazioni di classloader. I classloader in Java sono definiti a partire dalla classe java.lang.ClassLoader la cui gerarchia è schematizzata nella prossima immagine:

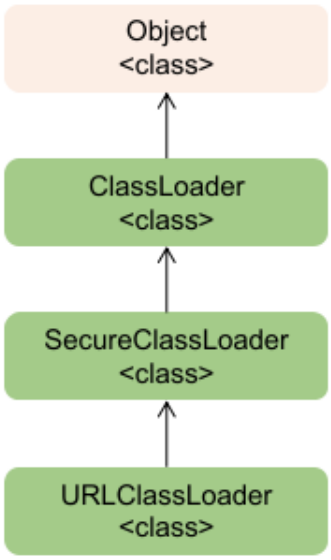


Immagine 70 Gerarchia di ClassLoader

Nella prossima tabella sono elencati i metodi della classe base *ClassLoader* che sottintendono al caricamento delle classi.

Metodi di ClassLoader per caricare classi
public Class<?> loadClass(String name) throws ClassNotFoundException
Carica la classe a partire dal nome. Questo metodo ricerca le classi allo stesso modo del metodo loadClass(String, boolean). Viene richiamato dalla macchina virtuale Java per risolvere i riferimenti alle classi. Richiamare questo metodo equivale a richiamare loadClass(name, false) la cui implementazione predefinita ricerca le classi nell'ordine definito dal meccanismo di delega.
public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException
Questo metodo è responsabile del caricamento di una classe a partire dal nome completo. La Java Virtual

Machine richiama il metodo loadClass() per risolvere i riferimenti alle classi, impostando resolve su true. Tuttavia, non è sempre necessario caricare una classe. Se dobbiamo solo determinare se la classe esiste o meno, allora il parametro resolve andrà impostato su false.

Questo metodo funge da punto di ingresso per il classloader.

---

protected final Class<?> findLoadedClass(String name)

Restituisce la classe il cui nome è specificato come attributo solo se la classe è stata già precedentemente caricata. Altrimenti ritorna ***null****.*

---

protected Class<?> findClass(String name) throws ClassNotFoundException

Ricerca la classe il cui nome è specificato come attributo sul filesystem. Nel caso in cui la classe non è trovata genera una ClassNotFoundException.

Inoltre, loadClass() richiama questo metodo se il classloaer padre non riesce a trovare la classe richiesta. L'implementazione predefinita genera un'eccezione ClassNotFoundException se nessun classloader genitore trova la classe.

---

protected final Class<?> defineClass(String name, byte[] b, int off, int len) throws ClassFormatError

Questo metodo è responsabile della conversione di un array di byte in un'istanza di una classe. Se i dati non contengono una classe valida, genera un ClassFormatError. Inoltre, non possiamo eseguire l'override di questo metodo, poiché è contrassegnato come final.

---

public final ClassLoader getParent()

Questo metodo restituisce il classloader padre per supportare il meccanismo della delega. Alcune implementazioni, come visto nell'esempio precedente, usano null per rappresentare il bootstrap classloader.

---

public URL getResource(String name)

Questo metodo cerca di trovare una risorsa con il nome specificato. Per prima cosa delegherà la ricerca al classloader padre.

Se il genitore è nullo, viene cercato utilizzando il classpath del classloader corrente.

Se fallisce, il metodo richiamerà findResource(String) per trovare la risorsa.

Il nome della risorsa specificato come input può essere relativo o assoluto rispetto al classpath.

Restituisce un oggetto URL per leggere la risorsa, o null se la risorsa non può essere trovata o non ha privilegi adeguati per restituire la risorsa.

Quando JVM richiede una classe, richiama la funzione *loadClass* della classe *ClassLoader* passando il nome completo della classe come argomento.

Per comprendere meglio come funziona il classloader ed il meccanismo di delega, analizziamo il codice del metodo *loadClass*:

```
protected Class<?> loadClass(String name, boolean resolve)
 throws ClassNotFoundException {

  synchronized (getClassLoadingLock(name)) {
    // First, check if the class has already been loaded
    Class<?> c = findLoadedClass(name);
    if (c == null) {
      long t0 = System.nanoTime();
        try {
          if (parent != null) {
            c = parent.loadClass(name, false);
          } else {
            c = findBootstrapClassOrNull(name);
          }
        } catch (ClassNotFoundException e) {
          // ClassNotFoundException thrown if class not found
          // from the non-null parent class loader
        }

        if (c == null) {
          // If still not found, then invoke findClass in order
          // to find the class.
          c = findClass(name);
        }
      }
      if (resolve) {
        resolveClass(c);
      }
      return c;
    }
  }
```

L'implementazione predefinita del metodo ricerca le classi nel seguente ordine:

1. Richiama il metodo findLoadedClass(String) per vedere se la classe è già caricata.

2. Richiama il metodo loadClass(String) classloader padre.

3. Richiamare il metodo findClass(String) per trovare la classe.

La prossima immagine schematizza il meccanismo di delega già visto nell'immagine precedente evidenziando ora lo scambio di messaggi che avviene tra un *classloader* ed il *classloader* padre.

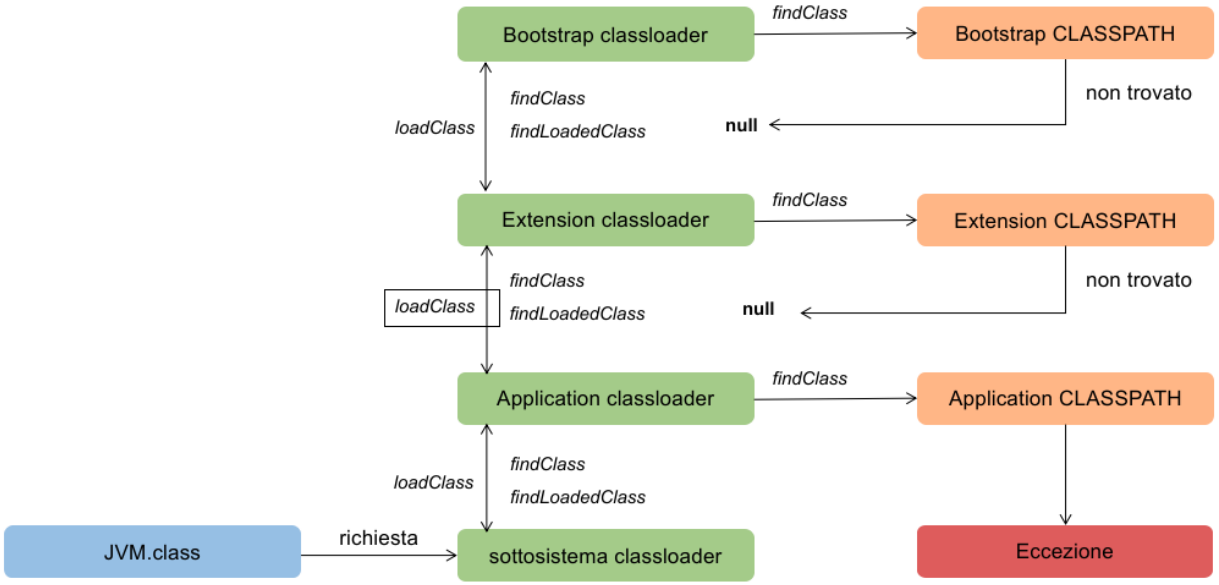


Immagine 71 Meccanismo di delega in dettaglio

Per completare la descrizione del funzionamento del classloader, non ci resta che capire a fondo la differenza tra le due eccezioni: *java.lang.NoClassDefFoundError* ed *java.lang.ClassNotFoundException*. Per poterlo fare dobbiamo prima di tutto introdurre due nuovi concetti.

Static vs dynamic class loading

Java supporta due diverse modalità di caricamento delle classi all'interno della *Java Virtual Machine*.

1. *static loading:*

Parliamo di *static loading* quando una classe viene caricata mediante l'operatore **new** utilizzando quindi il tipo al compile time di un oggetto. In questo caso la classe viene caricata ed istanziata durante la fase di compilazione.

```
class TestClass {
 public static void main(String args[]) {
    TestClass tc = new TestClass();
 }
}
```

2. *dynamic loading:*

Il caricamento dinamico, al contrario, si ottiene attraverso l'identificazione del tipo al run-time. Avviene utilizzando il metodo:

```
Class.forName (String className);
```

oppure utilizzando le java reflection api. Il caricamento dinamico avviene quindi quando non è possibile identificare il nome della classe al compile time.

NoClassDefFoundError vs ClassNotFoundException

E' il momento di rispondere alla domanda, e comprendere la differenza tra *java.lang.NoClassDefFoundError* e *java.lang.ClassNotFoundException?*

Abbiamo detto che entrambe si verificano quando una particolare classe non viene trovata al run-time, tuttavia si verificano in scenari diversi.

1. ClassNotFoundException è un'eccezione che si verifica quando si tenta di caricare una classe in fase di esecuzione utilizzando i metodi Class.forName() o loadClass() e le classi menzionate non vengono trovate nel classpath.

2. NoClassDefFoundError è un errore che si verifica quando una particolare classe è presente in fase di compilazione, ma non in fase di esecuzione.

Vediamo meglio i singoli casi.

ClassNotFoundException

ClassNotFoundException è un'eccezione di run-time di tipo unchecked, generata quando un'applicazione tenta di caricare una classe in fase di esecuzione utilizzando i metodi *Class.forName* o *loadClass* o *findSystemClass*, e la classe con il nome specificato non viene trovata nel classpath.

ClassNotFoundException si verifica sempre in fase di esecuzione perché stiamo caricando la classe utilizzando Classloader, e il compilatore Java non ha modo di sapere se la classe sarà presente nel *classpath* in fase di esecuzione o meno come mostrato nel prossimo esempio:

```
                    public static void main(String[] args) {
        try {
          Class.forName("it.javamattone.ClasseCheNponEsiste");
          ClassLoader.getSystemClassLoader().loadClass("it.javamattone.ClasseCheNponEsiste");
          ClassLoader.getPlatformClassLoader().loadClass("it.javamattone.ClasseCheNponEsiste");
        } catch (ClassNotFoundException e) {
          e.printStackTrace();
        }
      }
```

L'eccezione generata è la seguente:

```
java.lang.ClassNotFoundException: it.javamattone.ClasseCheNponEsiste
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown Source)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Unknown Source)
    at java.base/java.lang.ClassLoader.loadClass(Unknown Source)
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Unknown Source)
    at javamattone.esercizi.classloader.ClassNotFoundError.main(ClassNotFoundError.java:7)
```

NoClassDefFoundError

NoClassDefFoundError è un errore che viene generato quando la *JVM* tenta di caricare la definizione di una classe, ma la definizione di classe che era presente in fase di compilazione non lo è più in fase di esecuzione.

Consideriamo il prossimo esempio:

```
class A
{
 //definizione della classe
}

public class B
{
  public static void main(String[] args)
  {
    A a = new A();
  }
}
```

Quando compiliamo il codice saranno generati due file: *A.class* e *B.class. S*e rimuoviamo il file A.class ed eseguiamo B otteniamo il seguente errore:

```
Exception in thread "main" java.lang.NoClassDefFoundError: A
at MainClass.main(MainClass.java:10)
Caused by: java.lang.ClassNotFoundException: A
at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
```

Classloader personalizzati

Il classloader predefinito della *Java Virtual Machine* è sufficiente a coprire la maggior parte dei casi d'uso in cui le classi risiedono sul filesystem. Tuttavia, esistono situazioni in cui il classloader predefinito non è in grado di far fronte a richieste particolari quali ad esempio:

1. Caricare classi che non risiedono sul filesystem della macchina;

2. Aiutare a modificare il byte-code esistente (agenti di weaving);

3. Creare di classi adattate dinamicamente alle esigenze dell'utente;

4. Implementare un meccanismo di controllo delle versioni delle classi durante il caricamento di bytecode.

In questi casi è necessario creare classloader personalizzati.

Weaving è una tecnica per manipolare il byte-code delle classi Java compilate. La tecnica può essere eseguita in modo dinamico in fase di esecuzione, quando le entità vengono caricate, o in modo statico in fase di compilazione mediante post-elaborazione dei file *.class*.

Creare un classloader personalizzato è tanto facile quanto estendere la classe *ClassLoader*, ed effettuare l'override del metodo *findClass*. Nel prossimo esempio carichiamo la definizione di una classe da un array di byte dal file il cui nome è specificato come attributo di *findClass*..

```
public class CustomClassLoader extends ClassLoader {

  @Override
  public Class findClass(String name) throws ClassNotFoundException {
    byte[] b = loadClassFromFile(name);
    return defineClass(name, b, 0, b.length);
  }

  private byte[] loadClassFromFile(String fileName)  {
    InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
        fileName.replace('.', File.separatorChar) + ".class");
    byte[] buffer;
    ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    int nextValue = 0;
    try {
      while ( (nextValue = inputStream.read()) != -1 ) {
        byteStream.write(nextValue);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    buffer = byteStream.toByteArray();
    return buffer;
  }
}
```

22.Gestione della memoria e Garbage Collector

Introduzione

Chi conosce bene c++ sa perfettamente quali siano i limiti del linguaggio in termini di gestione della memoria la cui responsabilità cade, per intero, sul programmatore. Un programma c++ complesso può causare senza problemi un consumo incontrollato di memoria, e di conseguenza problemi all'ambiente su cui l'applicazione è in esecuzione.

Quello dei *memory leak* è un altro dei problemi a cui vanno spesso incontro le applicazioni c++: di fatto, poiché è responsabilità del programmatore rilasciare la memoria allocata da una applicazione, accade spesso che qualche porzione di questa vada dimenticata o perduta, e mai rilasciata. Se questo processo si ripete nel tempo, l'effetto finale è che l'applicazione consumerà porzioni di memoria sempre più grandi (*leak*) fino al completo esaurimento della memoria disponibile.

Java implementa un diverso meccanismo per la gestione della memoria affidata completamente ad un modulo della JVM chiamato *Garbage Collector - GC*. Il GC di Java ha la responsabilità di rimuovere dalla memoria oggetti non più utilizzati, di limitare il consumo della memoria da parte di una applicazione, ed in generale ottimizzare la risorsa sostituendosi completamente al programmatore. Un programmatore Java non dovrà infatti preoccuparsi di dover gestire la memoria e potrà invece concentrarsi su altri aspetti importanti della applicazione.

Nonostante il GC però, Java non è esattamente un linguaggio *leak-safe*: anche in Java i *memory leak* rappresentano un problema in agguato dietro ogni angolo, tuttavia basterà un po di attenzione da parte del programmatore per evitarli. Esistono comunque molti strumenti, detti di *profiling*, che possono essere utilizzati per identificare e risolvere le cause di *memory leak*.

Esistono diversi tipi di *Garbage Collectors* in Java differenti tra loro per strategia, performance, contesto di utilizzo. In questa sezione comprenderemo un po più a fondo il modello della gestione della memoria in Java ed analizzeremo le diverse implementazioni del *Garbage Collector*.

Il modello della memoria in Java

Per comprendere il funzionamento del GC in Java, e in generale poter definire strategie in fase di disegno, test ed ottimizzazione di una applicazione, è importante comprendere a fondo il modello della memoria della JVM.

Come tutte le altre applicazioni, anche la JVM risiede nella memoria della macchina. Tuttavia, all'interno della JVM, esistono spazi di memoria separati, *Heap, Method Area* e *Native Area*, per archiviare i dati di run-time e il codice compilato. La porzione denominata *Native Area* contiene a sua volta lo *Stack* e la *Cache*.

Il modello generale della memoria in Java è rappresentato in dettaglio nella prossima figura.

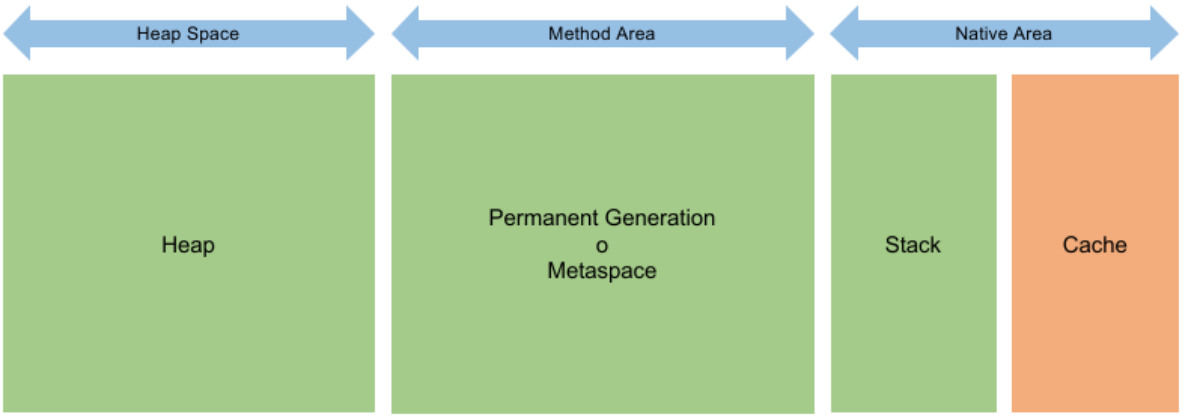


Immagine 72 Modello generale della memoria Java

Memoria Heap

Nella prossima immagine è mostrato lo schema della memoria *heap* di Java. Questa porzione di memoria viene allocata non appena viene avviata la *JVM* ed ha una dimensione iniziale prestabilita (cosa che tratteremo successivamente) che può crescere, durante l'esecuzione della applicazione, fino ad una dimensione massimo oltre il quale non può espandersi a meno di causare la terminazione immediata della applicazione Java.

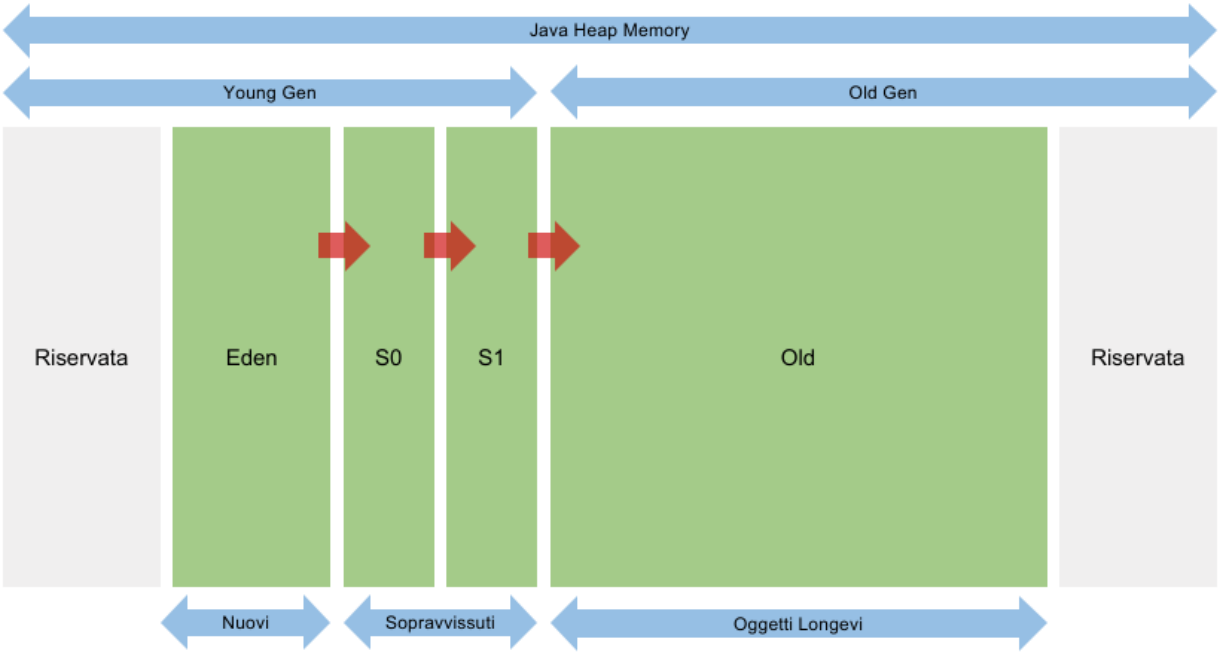


Immagine 73 Schema della memoria heap

Lo *heap* è a sua volta suddiviso in due blocchi logici:

1. Young Generation

E' la porzione dello *heap* dove sono allocati i nuovi oggetti. Ogni nuovo oggetto viene quindi allocato in *Eden*, e come vedremo successivamente, man mano che sopravvivono ai cicli di pulizia da parte del *Garbage Collector* vengono prima spostati in uno degli spazi dedicati ai sopravvissuti, ed infine nella porzione dello *heap* dedicato agli oggetti longevi.

2. Old Generation

E' lo spazio riservato agli oggetti che sono sopravvissuti a molti cicli di pulizia della memoria. La gestione di questo spazio da parte del *Garbage Collector* richiede generalmente più tempo.

Method Area

Questa porzione di memoria include il *Permanent Generation.* Non fa parte della memoria Java Heap..

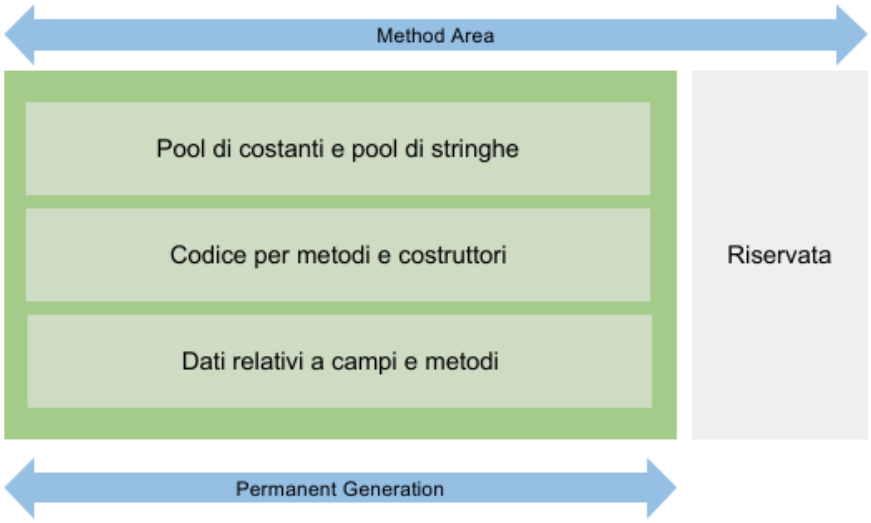


Immagine 74 Memoria della Method Area

Quest'area contiene i metadati dell'applicazione richiesti dalla JVM per descrivere le classi e i metodi utilizzati nell'applicazione. E' popolato da JVM in fase di esecuzione in base alle classi utilizzate dall'applicazione e contiene anche classi e metodi della libreria Java SE.

E' utilizzata per la memorizzazione delle costanti di run-time, a seconda dell'implementazione del gestore della memoria della JVM potrebbe essere utilizzato in vece dello *heap* per la creazione di pool di oggetti immutabili (come le stringhe).

Stack

Lo *stack* viene utilizzato per l'allocazione della memoria statica e l'esecuzione di thread (ricordiamo che una applicazione Java è essa stessa un thread). Mantiene riferimento ai metodi in ordine di chiamata (pertanto l'accesso alla memoria *stack* avviene in modalità LIFO (Last-In-First-Out), per ogni metodo mantiene i valori primitivi ed i riferimenti agli oggetti memorizzati nello *heap*. Ogni volta che chiamiamo un nuovo metodo, viene creato un nuovo blocco in cima allo stack che contiene valori specifici di quel metodo. Quando il metodo termina l'esecuzione, il frame dello stack corrispondente viene svuotato, il flusso torna al metodo chiamante e lo spazio diventa disponibile per il metodo successivo.

Le caratteristiche dello *stack* sono le seguenti:

1. *Cresce e si riduce man mano che vengono chiamati e terminati nuovi metodi, rispettivamente.*
2. *Le variabili all'interno dello stack esistono solo finché il metodo che le ha create è in esecuzione.*
3. *Viene allocato e deallocato automaticamente quando il metodo termina l'esecuzione.*
4. *Se questa memoria è piena, Java genera java.lang.StackOverFlowError.*
5. *L'accesso a questa memoria è rapido rispetto alla memoria heap.*
6. *Questa memoria è thread-safe, poiché ogni thread opera nel proprio stack.*

Lo *stack* non è un area di memoria contigua; in Java ogni thread ha un proprio stack space riservato.

Per comprendere meglio il funzionamento dello *stack* consideriamo il codice a seguire:

```
public class Macchina {
  public String targa;
  public int annoDiImmatricolzione;
  public Macchina(String targa, int annoDiImmatricolzione){
    this.targa = targa;
    this.annoDiImmatricolzione = annoDiImmatricolzione;
  }
}


public class AutoSalone {
  private static Macchina immatricolaMacchina(String targa, int annoDiImmatricolzione){
    return new Macchina(targa, annoDiImmatricolzione);
  }
  public static void main(String[] args) {
    int annoDiImmatricolzione = 2023;
    String targa = "CX678DJ";
    Macchina macchina = null;
    macchina = immatricolaMacchina(targa, annoDiImmatricolzione);
  }
}
```

Una volta eseguita l'applicazione, vengono chiamati in sequenza i tre metodi:

```
main(String[])
immatricolaMacchina(String, int)
Macchina(String, int)
```

1. Quando viene invocato il metodo main viene allocata memoria nello stack per memorizzare il valore primitivo *annoDiImmatricolazione,* una reference ad una Stringa *targa* ed una reference ad un tipo *Macchina*.

2. Il metodo *main* invoca il metodo statico immatricolaMacchina(String, int). Di conseguenza sarà creato spazio sulla cima dello *stack* per memorizzare il valore primitivo *annoDiImmatricolazione,* una reference ad una Stringa *targa* ed una reference ad un tipo *Macchina* che rappresenta l'oggetto che sarà ritornato dal metodo.

3. Una volta invocato il costruttore di *Macchina*, sarà allocata memoria sullo *heap* per memorizzare il valore primitivo *annoDiImmatricolazione,* una reference ad una Stringa *targa* e la reference *this* all'oggetto corrente.

4. Man mano che i metodi chiamati ritornano, saranno rimossi dallo *stack* a partire dall'alto verso il baso (gestione LIFO).

Nella prossima figura è schematizzato il processo appena descritto:

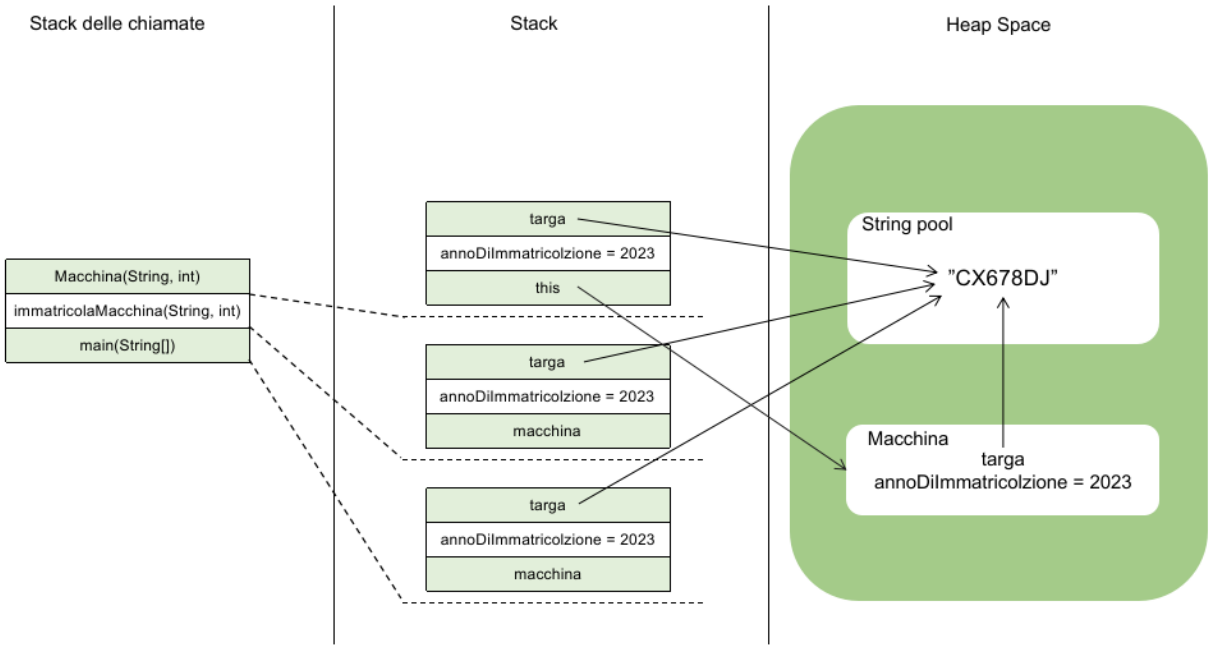


Immagine 75 gestione dello stack

Modelli alternativi per la gestione della memoria

Quella appena discusso è il modello generale della memoria comunemente conosciuto ed utilizzato. Tuttavia, alcune versioni della JVM possono, sopratutto quelle di nuova generazione, possono contenere alcune modifiche come ad esempio l'introduzione di alcuni nuovo spazi di memoria.

1. *Keep Area:*

Un nuovo spazio di memoria nella Young Generation per contenere gli oggetti allocati più di recente. Quest'area impedisce agli oggetti di essere promossi solo perché sono stati assegnati poco prima dell'inizio di un ciclo di pulizia.

2. Metaspace

A partire da Java 8, *Permanent Generation* è stato sostituito da *Metaspace*. A differenza del predecessore, può aumentare automaticamente le sue dimensioni (fino a quanto consentito dal sistema operativo sottostante) anche se Perm Gen ha sempre una dimensione massima fissa.

Configurare la memoria in Java

Quando eseguiamo applicazioni che richiedono molte risorse di memoria, è necessario intervenire sulla configurazione della JVM, e modificare i parametri di utilizzo della memoria.

La configurazione dello *heap space* è schematizzata nella prossima immagine:

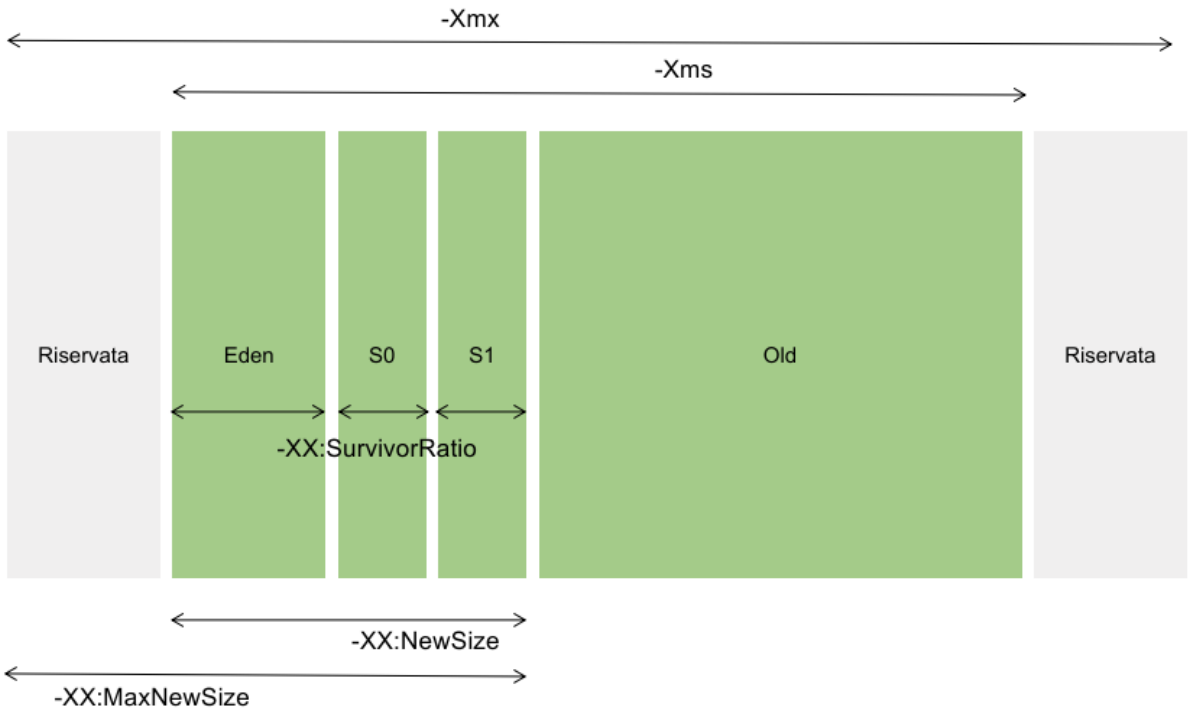


Immagine 76 Configurazione dello heap

1. -Xms Setting:

Dimensione dell'heap iniziale

2. -Xmx Setting:

Dimensione massima dell'heap

3. -XX:NewSize

Dimensione dell'heap di nuova generazione

4. -XX:MaxNewSize

Dimensione massima dell'heap di nuova generazione

5. -XX:SurvivorRatio:

Modifica i rapporti delle dimensioni dell'heap (ad es. se la dimensione *Young Generation* è 10m e il memory switch è XX:SurvivorRatio=2, allora 5m saranno riservati per lo spazio *Eden* e 2,5m ciascuno per entrambi gli spazi *Survivor*, valore predefinito = 8)

6. *-XX:NewRatio:*

Fornisce il rapporto tra le dimensioni di vecchia/nuova generazione (valore predefinito = 2)

I prossimi parametri di configurazione hanno invece effetto sulla memoria *Permanent Generation*:

7. -XX:PermSize:

Dimensione iniziale della *Permanent Generation*;

8. -XX:MaxPermSize:

Dimensione massima della *Permanent Generation.*

L'effetto sulla *Permanent Generation* è mostrato nella prossima immagine.

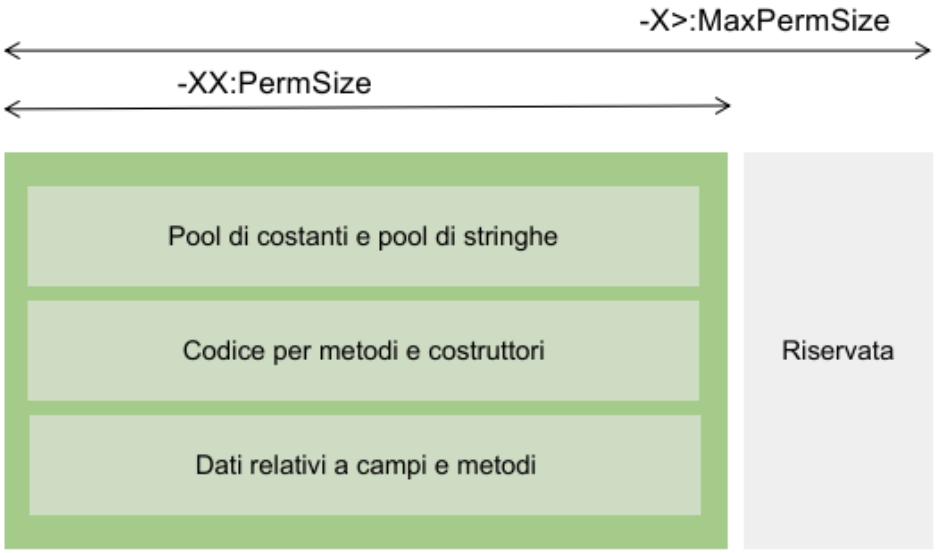


Immagine 77 Configurazione della permanent generation

Errori relativi alla memoria

Quando si verifica un problema di memoria critico, la JVM si arresta in modo anomalo e genera un'indicazione di errore nell'output del programma come descritto nella prossima tabella.

Errori relativi alla gestione della memoria della JVM
java.lang.StackOverFlowError
Indica che lo stack-space è pieno.
java.lang.OutOfMemoryError: Java heap space
Indica che la memoria heap è piena.
java.lang.OutOfMemoryError: GC Overhead limit exceeded
Indica che il Garbage Collector ha raggiunto il suo limite di sovraccarico.
java.lang.OutOfMemoryError: Permgen space
Indica che lo spazio Permanent Generation è pieno.
java.lang.OutOfMemoryError: Metaspace
A partire da Java 8, Indica che il metaspace è pieno.
java.lang.OutOfMemoryError: Unable to create new native thread
indica che il codice nativo JVM non può più creare un nuovo thread nativo dal sistema operativo sottostante perché sono già stati creati così tanti thread che consumano tutta la memoria disponibile per la JVM.
java.lang.OutOfMemoryError: request size bytes for reason
Indica che la memoria di swap è piena perché consumata dalla applicazione.
java.lang.OutOfMemoryError: Requested array size exceeds VM limit–
Indica che la nostra applicazione sta utilizzando un array le cui dimensioni eccedono la dimensione massima consentita dalla macchina su cui l'applicazione è in esecuzione.

E' importante comprendere che le condizioni di errore elencate rappresentano l'impatto che un altro errore, quello effettivo, ha avuto sulla JVM. In questi casi, aumentare la memoria non è detto che rappresenti la soluzione al problema, ma bisognerà intervenire per circoscrivere il motivo ed identificare la giusta strategia di intervento.

Per fare questo, esistono molti strumenti, chiamati profiler, che consentono di monitorare i parametri e la telemetria della JVM ed aiutano ad identificare eventuali leak oppure altre aree di intervento.

Garbage collector: cos'è e come funziona

23.Serializzazione di oggetti

Introduzione

24.Conclusioni

Per concludere ...

Se state leggendo questa sezione conclusiva significa che Java Mattone dopo Mattone è servito a qualcosa. Quantomeno non vi siete annoiati completamente.

Di tutte le cose dette nel libro, quello che vorrei rimanesse è l'amore per il buon codice e la volontà do voler migliorare e crescere senza mai fermarsi.

Scrivere buon codice non è soltanto questione di soddisfazione personale, scrivere buon codice significa migliorare la propria produttività, e sopratutto rispettare il vincolo di professionalità che avete nei confronti della vostra società e dei vostri clienti.

Ma non solo: quando un programma è scritto bene, i problemi diventano più facili da risolvere: un codice ben progettato, e con una solida strategia di problem solving alle spalle, aiuta a risolvere i problemi in modo semplice ed efficacie piuttosto che con un approccio orientato alla *forza bruta*.

La manutenzione diventa più semplice: un codice pulito è più facile da leggere e comprendere quindi si potrà dedicare meno tempo alla comprensione dei sorgenti e concentrarsi invece sulla risoluzione dei problemi.

Le idee sono comunicate chiaramente: sarà più facile collaborare con il team evitando incomprensioni inevitabili il che significa ridurre la probabilità di errori sopratutto per i progetti a lungo termine.

Riassumendo quindi tutto quanto detto nei capitoli precedenti, ecco alcuni buoni consigli per scrivere un buon codice.

Usa sempre nomi descrittivi

> *"Non sono un grande programmatore; Sono solo un buon programmatore con grandi abitudini."*
>
> *Kent Beck*[9]

Quali sono le sezioni? Le classi ed i metodi? Le variabili? Come si comporta logicamente il codice? Queste sono le prime domande che un programmatore si pone quando deve leggere un codice sorgente.

Utilizzare nomi criptici e poco descrittivi per variabili classi e metodi è come mettere un vetro appannato tra noi ed il monitor del computer: tutto diventa sfocato, facciamo fatica a cogliere i dettagli e le logiche di implementazione.

---

[9] *Kent Beck* (1961) è un informatico statunitense, creatore della metodologia di sviluppo del software Extreme Programming

Stiamo sostanzialmente offuscando le logiche applicative ad altri programmatori ed a noi stessi.

Definire variabili con nomi corti semplicemente per risparmiare battute sulla tastiera rende incomprensibile il significato stesso della variabile. Scrivere var1, var2, ... significa costringere il programmatore a leggere tutta la sezione di codice per cercare di dare un senso alla variabile. Diverso sarebbe identificare una variabile con un nome auto-documentante tipo: *variabileCheFaQuestoEQuello*.

Stessa cosa per i nomi di classi e metodi: scrivere *tang()* non è come scrivere *calcolaAngoloTangente()*.

Keep it simple ovvero ... è tutto un leggi e scrivi

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

(Alan Peris 1982 - Epigrams on programming)

O forse dovrei citare Gaetano Barbagallo, il mio primo capo progetto nel 96 quando, di fronte a compiti che apparentemente erano complessi era solito dirmi "*Alla fine tutto è un leggi e scrivi. Se il compito è difficile sei tu che hai sbagliato qualcosa*".

Sono passati quasi trent'anni e ancora queste parole sono per me il faro nella tempesta. Tante volte mi è capitato di mettere mano a codice con funzioni e metodi contenenti centinaia o addirittura migliaia di righe di codice.

Diciamocela tutta ... anche noi ci siamo caduti prima o poi nella nostra vita professionale.

In questi casi non esiste commento in grado di aiutare; possiamo solo passare ore con il debugger attivo ed il quadernetto degli appunti cercando di uscirne limitando i danni.

Un codice pulito è diviso in blocchi atomici, una classe dovrebbe rappresentare un concetto, un metodo dovrebbe fornire un punto di accesso chiaro alla nostra classe, una funzione dovrebbe approcciare con una tecnica divide et impera per semplificare il problema.

Quindi, piuttosto che scrivere blocchi di codice monolitici, dividiamo il problema in sotto problemi semplici da comprendere, usiamo strumenti come gli oggetti e l'ereditarietà per riutilizzare parte del codice che scriviamo.

Elimina il codice inutile

Una delle peggiori pratiche è quella di commentare parti di codice riscrivendolo e modificandolo, mantenendo però il vecchio codice commentato. Passa il tempo, accumulo blocchi di codice commentato, ma nel frattempo il resto del codice è stato modificato a tal punto che il codice commentato non potrebbe più funzionare se ripristinato.

Questa è una pratica che non dovrebbe essere più utilizzata: primo perché rende completamente illeggibile il resto del codice, poi perché è una pratica resa totalmente inutile dall'uso di sistemi di *versioning* come git.

Poi parliamoci chiaro ... quante volte abbiamo rimosso un commento per sbaglio e passato le ore a capire perché la funzione che fino a stamattina funzionava poi ha smesso di funzionare?

Le convenzioni sono importanti

Le convenzioni sono importanti: rappresentano una sorta di contratto tra programmatori, e ci aiutano a strutturare il codice in maniera che altri possano usarlo e modificarlo.

Programmare in java e sostituire la *cammellatura* con stringhe separate da carattere '_' non è mai una buona pratica.

Se per accedere ai dati membro di una classe si usano metodi *getter* e *setter* è inutile chiamarli in altro modo

Ma se proprio le convenzioni non ti piacciono quantomeno meglio rimanere coerenti, e se utilizzi uno stile di programmazione cerca di utilizzarlo sempre ed allo stesso modo.

> "Bello è meglio che brutto.
> L'esplicito è meglio che implicito.
> Semplice è meglio che complesso.
> Il complesso è meglio che complicato.
> Flat è meglio di nidificato.
> Sparse è meglio che denso.
> Conta la leggibilità "
>
> *Tim Peters, The Zen di Python*

Meno non è sempre ... meglio

Cerchiamo di prediligere sempre la leggibilità alla sfida. Compattare dieci righe di codice in una, anche se può sembrare una sfida affascinate, una carezza all'ego del programmatore, ed una convalida alle sua abilità non è mai una buona idea.

Sarà bellissimo da vedersi!

Sarà anche altrettanto comprensibile?

Quindi direi, lasciamo il nostro ego di programmatori fuori dalla programmazione e preoccupiamoci che il codice sia prima di tutto leggibile perché dopo di noi ci sarà un altro programmatore che dovrà leggerlo e domani ancora quel programmatore potreste essere voi.

Mantieni sempre un approccio orientato al problema

Quante volte ci siamo sentiti dire (o abbiamo detto) 'Vabbè prendi quel framework perché lo adatti con poco e risparmiamo tempo e denaro'?

Quante volte abbiamo rimpianto di averlo fatto?

Esistono diversi approcci, paradigmi, architetture e framework. L'unica scelta possibile è quella giusta per affrontare e risolvere il tuo problema. Mai cercare la strada più veloce o più conveniente. Anche ciò 'che la fuori è meglio' non è detto che vada bene per te.

Studia il codice di chi è più esperto di te

Per imparare a scrivere codice pulito la cosa migliore è studiare quello di chi ha più esperienza di te.

"Ogni pazzo può scrivere codice che un computer può capire. I bravi programmatori scrivono un codice che gli umani possano capire ".
- Martin Fowler, Refactoring: migliorare il design del codice esistente

Se non sai come fare, basta guardarti intorno: il web è pieno di ottimi progetti open source condivisi, ma anche nell'azienda dove lavori puoi trovare spunti interessanti.

Chiedi, ruba (i programmatori sono generalmente felici di condividere) l'importante è non smettere mai di cercare di migliorare.

Commenta il codice nel modo corretto

"Codifica sempre come se il tizio che finisce per mantenere il tuo codice sarà uno psicopatico violento che sa dove vivi."
John Woods

Commentare il codice non significa necessariamente scrivere buoni commenti. Quando iniziamo a programmare, quella di commentare il codice è la prima cosa che ci viene detta e ripetuta come un mantra. Il 'namyo renge kio' del programmatore.

Il più delle volte però si finisce sempre con commentare il codice in maniera eccessiva, riempendo il sorgente di commenti che descrivono cose che non serviranno mai a nessuno o che non richiedono di essere descritte.

Ci sono poi linguaggi, come java, che hanno un potere espressivo così alto che un codice ben scritto è auto-documentante.

Ecco alcuni consigli per scrivere commenti:

1. Concentratevi sempre sul perché avete scritto un frammento di codice e non su come lo fate;

2. I commenti dovrebbero sempre focalizzarsi su quello che non è possibile dedurre dal codice;

3. Non insultare l'intelligenza altrui:

```
for(i = 0; i < array.length; i++) { //ciclo per tutta la lunghezza dell'array
```

4. Inutile commentare ogni singola variabile e poi chiamarla var-qualchecosa. Meglio utilizzare un nome che sia auto documentante e concentrarsi su ciò che è veramente importante commentare: torna alla regola 1.

Refactor, refactor, refactor

Prima fallo funzionare, poi ottimizzalo, infine fallo più bello!

Me medesimo - 2022!

Attraverso il refactoring si interviene sulla struttura del codice senza cambiare il comportamento esterno. In questo modo si migliorano alcune caratteristiche non funzionali del software quali la leggibilità, la manutenibilità, la riusabilità o la sua estensibilità. Il refactoring del codice è parte indispensabile del processo di sviluppo. Inutile commentare un codice scritto male ... meglio riscriverlo.

Non smettere mai di desiderare di imparare

Esiste solo un modo giusto per scrivere del buon codice, ma molti sbagliati. Riuscire a padroneggiare una buona tecnica richiede anni di studio ed esperienza e quando sentirete di essere arrivati il mondo sarà così cambiato che bisognerà ripartire da lì.

Il futuro del linguaggio Java

Il 2022 è stato un anno significativo per il linguaggio Java. Il 20 settembre Oracle ha ufficialmente presentato Java19, versione non LTS, dimostrando ancora una volta quanto la comunità degli sviluppatori Java rappresenti la linfa vitale per un linguaggio che continua ad aggiornarsi ed a modificare se stesso per rispondere alle necessità di una comunità che si aggira oggi intorno ai 30 milioni di sviluppatori in tutto il mondo.

Con le due versioni di Java nel 2022, gli sviluppatori hanno visto progressi significativi nelle quattro principali iniziative denominate nei progetti *Valhalla*, *Panama*, *Loom* e *Amber*.

Il progetto Amber

Project Amber esplora funzionalità minori del linguaggio Java orientate alla produttività.

Guidata da *Brian Goetz*, *Amber* è stata lanciata nel 2017. Proprio grazie alla sua natura di progetto orientato a piccole funzionalità con impatto sulla produttività del linguaggio, *Amber* ha già rilasciato diverse di esse tra cui molte citate in questo libro. Solo per citarne alcune:

1. *Inferenza del tipo di variabile locale, o **var**, in Java 10;*

2. *Text blocks in java 15;*

3. *tipi records in java 16;*

4. *Seled class ein Java 17.*

Il progetto Panama

Il progetto Loom

Il progetto Valhalla