Lorenzo Cioni

Object Oriented Programming e Java

(work in progress, prima parte)

Abstract

Il presente articolo rappresenta la prima parte di un lavoro mirante alla analisi del paradigma Object Oriented e alla sua realizzazione nel linguaggio Java.

L'articolo presenta una introduzione alle principali caratteristiche del paradigma Object Oriented per poi passare ad un breve esame di alcune "regole del pollice" per la progettazione Object Oriented e chiudersi con un breve excursus del linguaggio Java e di come il modello Object Oriented venga realizzato in tale linguaggio

1 Introduzione: oggetti e classi

Secondo il paradigma dell'Object Oriented Programming[1] un programma è composto da un certo numero di oggetti che cooperano fra di loro. La cooperazione è realizzata mediante la comunicazione fra gli oggetti (detta in ambito OOP scambio di messaggi) il cui scopo è quello di risolvere in modo cooperativo un certo compito.

Ognuno degli oggetti coinvolti è caratterizzato da un compito preciso e il loro uso permette al programmatore di decomporre un problema in più sottoproblemi consentendogli di gestire più facilmente la complessità del problema in questione.

Gli oggetti possono essere visti come moduli funzionali con stato. Un oggetto è, in effetti, caratterizzato da:

- uno stato;
- un certo numero di operazioni e
- una identità.

Lo stato di un oggetto è costituito dalle informazioni che sono il risultato delle operazioni svolte dall'oggetto nel passato e che determinano come un dato oggetto eseguirà le proprie operazioni nel futuro. L'unico modo in cui lo stato di un oggetto può cambiare è attraverso l'invocazione di alcune delle operazioni dell'oggetto, dette pertanto modificatori. Altre operazioni che consentono solamente l'accesso a informazioni proprie dell'oggetto ma non ne modificano lo stato saranno dette accessori[2].

L'insieme delle operazioni ammissibili da un oggetto ne rappresenta un attributo importante ed è l'unico modo per interagire con l'oggetto stesso.

Oltre allo stato e all'insieme delle operazioni ammissibili un oggetto è caratterizzato da una identità, ovvero ogni oggetto è individuabile in modo univoco all'interno di una popolazione di oggetti dello stesso tipo e ciò consente di differenziare il concetto di oggetto da quello di modulo.

Oggetti di uno stesso tipo sono detti essere istanziazioni di una classe. Il concetto di classe pertanto rappresenta uno strumento per descrivere un insieme di oggetti caratterizzati dalle stesse operazioni ammissibili ma dotati ciascuno di un proprio stato e di una propria identità.

Gli oggetti di una stessa classe, infatti, supportano lo stesso insieme di operazioni e condividono l'insieme degli stati possibili.

La descrizione di una classe deve pertanto contenere i seguenti elementi:

- le operazioni consentite sugli oggetti della classe e
- l'insieme degli stati possibili per gli oggetti della classe.

Ognuno degli oggetti è vincolato dalle proprietà della classe di cui è una istanza dal momento che supporta solo le operazioni che la classe elenca come ammissibili e i suoi stati devono appartenere all'insieme degli stati legali per la classe.

Data una classe, di solito, questa è caratterizzata da operazioni che sono raggruppabili in base alla loro tipologia in costruttori, distruttori, accessori, mutatori, operazioni di confronto e iteratori.

Un costruttore è una operazione interna alla classe il cui scopo è quello di inizializzare i campi dati di un oggetto della classe. Ogni classe dovrebbe avere un costruttore e possono darsi casi di classi con più costruttori. Un distruttore è una operazione che al momento di

---

1 Nel seguito si userà l'acronimo OOP e acronimi simili saranno usati per altri aspetti del paradigma Object Oriented fra cui OOD per Object Oriented Design.

2 Il termine accessori è la brutta traduzione del termine inglese accessors mentre il termine modificatori è la traduzione ben più accettabile del termine mutators.

deallocare un oggetto si occupa di rilasciare le risorse esterne a lui allocate. In Java non è indispensabile definire i distruttori ma si possono pensare delle operazioni che si comportano come se fossero distruttori.

Un accessore è una operazione che accede ai campi di un oggetto senza modificarne il contenuto e restituendo tali valori come valori di ritorno al contrario di un modificatore che è una operazione che modifica un oggetto variando il contenuto di uno o più campi dati. Quasi tutte le classi hanno modificatori dal momento che è molto raro che sia possibile assegnare tutti i valori ai campi dati di una classe al momento della sua definizione e che tali campi dati non debbano più essere modificati.

Le operazioni di confronto (di tipo accessore) permettono di confrontare fra di loro oggetti dello stesso tipo per vedere se sono uguali oppure se, in base a un certo ordinamento, uno dei due è più piccolo dell'altro.

Gli iteratori, infine, sono operazioni tipiche di classi che gestiscono collezioni di oggetti accessibili o tramite una chiave o tramite un protocollo iterativo nel qual caso le operazioni (dette iteratori) lavorano sull'item corrente per poi passare al successivo in modo automatico.

Sugli oggetti sono definite anche le operazioni di copia e clonazione e le operazioni di input/output.

Nel caso di oggetti che non gestiscono risorse esterne l'operazione di copia può essere vista come una operazione di assegnamento seguita da più operazioni di inizializzazione, nel caso di oggetti che hanno un comportamento complesso, l'operazione di copia può richiedere la definizione di operazioni ad hoc.

La copia di un oggetto può essere di tipo superficiale o shallow o di tipo profondo o deep. Nel caso di Java la copia di un oggetto è sempre di tipo shallow nel senso che se creo un oggetto e lo assegno ad un altro entrambi finiscono per puntare alla stessa struttura fisica ovvero allo stesso elemento dello spazio del problema. La copia di tipo deep la si può ottenere con una operazione di clonazione.

L'operazione di clonazione fa si che sia creata una vera copia di un oggetto (ovvero un oggetto uguale ad un altro ma indipendente da questo) e deve essere implementata esplicitamente caso per caso, tenendo presente che tale operazione restituisce di default un oggetto della classe Object di cui deve essere fatto il casting esplicito al tipo desiderato (vedi oltre).

Le operazioni di output prevedono la restituzione di un oggetto su un dispositivo di output accedendo all'oggetto mediante il suo handle o, con dizione impropria, puntatore mentre le operazioni di input prevedono l'assegnazione ai campi di un oggetto, acceduto tramite il suo handle, dei valori letti da un dispositivo di input e dovrebbero prevedere uno stato consistente dell'oggetto in caso di errori durante le operazioni di input stesse.

2 Caratteristiche di base: interfaccia, ereditarietà & polimorfismo

Il paradigma OO, pertanto, considera come elementi costituenti di un programma le classi e gli oggetti dove le classi incapsulano i concetti che il programmatore vuole rappresentare in un programma e gli oggetti sono una sorta di concretizzazione di tali concetti. Nei paragrafi che seguono verranno esaminate alcune caratteristiche di base delle classi e, pertanto, degli oggetti.

2.1 Interfaccia

Una classe è pertanto caratterizzata dalle proprie operazioni e dai propri campi dati. L'interfaccia di una classe è costituita dalle operazioni che sono l'unico modo con cui è possibile accedere ad oggetti della classe.

L'incapsulamento rappresenta il procedimento in base al quale i campi dati di una classe sono resi privati alla classe in modo da nascondere rispetto alle altre classi i dettagli implementativi. Il fatto che un campo dati sia di tipo privato rappresenta un inconveniente di tipo minore in quanto è sempre possibile definire una funzione che acceda al campo dati e lo modifichi.

La progettazione di una interfaccia deve essere svolta in modo da garantirne la coesione ovvero in modo da garantire che una classe descriva una sola astrazione e sia caratterizzata da un insieme coerente di operazioni. Oltre a ciò è necessario che le operazioni dell'interfaccia siano primitive (ovvero non siano scomponibili in operazioni più semplici e siano facilmente componibili in operazioni più complesse) e che costituiscano un insieme completo (ovvero contenente tutte le operazioni significative per l'astrazione rappresentata dalla classe). Altri requisiti dell'interfaccia sono la convenienza e la consistenza.

La convenienza si traduce nella presenza di un insieme di operazioni di utilità ottenute componendo le operazioni elementari (che permettono di stabilire la completezza dell'interfaccia) e può essere vista come una sorta di "zucchero sintattico".

La consistenza riguarda gli aspetti formali delle operazioni relativamente allo stile dei loro nomi, dei parametri e dei valori di ritorno ma si rispecchia anche nel comportamento delle operazioni che deve essere coerente in modo da evitare sorprese e incomprensioni ai programmatori che usano la classe.

2.2 Ereditarietà

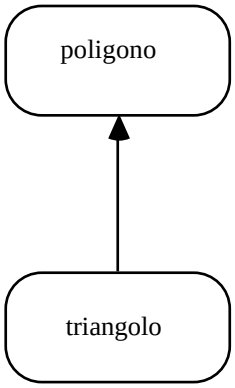
Un aspetto importante dell' OOP è la possibilità di sfruttare le similitudini fra le classi nel senso che può capitare che una classe non sia nient'altro che la versione specializzata di un'altra: le due classi hanno la maggior parte delle operazioni ammissibili identiche o simili sia pure con alcune differenze. Tale relazione di sottoinsieme viene detta ereditarietà.

In modo più formale si dice infatti che una classe, detta sottoclasse o classe derivata, eredita da una classe, detta classe base o superclasse, se gli oggetti della prima sono un sottoinsieme di quelli della classe base.

La relazione di sottoinsieme si traduce nel fatto che gli oggetti della sottoclasse devono supportare tutte le operazioni supportate dagli oggetti della superclasse ma possono svolgere tali operazioni in modo diverso oltre ad avere operazioni aggiuntive e tipiche della sottoclasse. In ogni modo, ogni oggetto della sottoclasse deve essere utilizzabile in tutti i casi in cui è possibile utilizzare un oggetto della superclasse[3].

L'ereditarietà è un concetto linguistico del paradigma OO e rappresenta pertanto una importante relazione fra le classi. Una classe detta derivata eredita da un'altra classe detta base se la classe derivata definisce un sottoinsieme degli oggetti della classe base.

Il concetto di ereditarietà stabilisce il cosiddetto principio di sostituzione in base al quale è possibile utilizzare un oggetto della classe derivata in tutti posti in cui è legale la presenza di un oggetto della classe base.



Ad esempio (vedi figura a lato, il simbolismo è spiegato nel seguito) se definisco la classe dei *poligoni* con le operazioni di tracciamento, scalatura e traslazione posso definire come classe derivata la classe dei *triangoli* in modo da poter affermare che "un triangolo è un poligono" e poter usare le operazioni dei poligoni sui triangoli. In tal modo si è definita una sottoclasse o classe derivata per la quale si possono definire operazioni aggiuntive tipiche della sottoclasse (ad esempio is_rettangolo) oppure implementazioni diverse di operazioni tipiche della classe base (ad esempio area come base * altezza /2).

Definendo una classe derivata è necessario dichiarare solo le differenze dato che l'ereditarietà si traduce in una importazione automatica di operazioni e proprietà. Nell'esempio ho che la classe *triangoli* ha un suo costruttore (che dovrà essere implementato esplicitamente) e può avere una sua versione della funzione per il tracciamento. Un altro punto da evidenziare è che se si aggiunge una operazione in un momento qualunque alla classe base questa viene ereditata in modo automatico dalle classi derivate, indipendentemente dal momento della loro definizione.

Vediamo ora brevemente che relazione ci può essere fra le operazioni della classe base e quelle di una classe derivata.

Una classe derivata D può comportarsi in tre modi diversi rispetto ad una operazione definita nella classe di base B:

- può ereditarla senza modifica;
- può sostituirla con una operazione locale che esegue una funzione diversa oppure
- può estenderla con una funzione locale alla classe che usa quella della classe base o superclasse ed in più svolge altri compiti.

Una classe è caratterizzata da dati privati e da operazioni che consentono l'accesso e la manipolazione di tali dati. È bene al proposito non confondere la presenza dei dati con il diritto di accedere e manipolare tali dati. Se una classe derivata deve, infatti, poter accedere ai dati della classe base deve, a tale scopo, utilizzare le funzioni pubbliche della

3 Va da sè che la dizione "oggetto della classe" stà per "oggetto istanza della classe".

classe base come ogni altra classe, ovvero l'ereditarietà non fa acquisire ad una classe derivata il diritto di manipolare direttamente i dati della sua superclasse vanificandone l'incapsulamento: le operazioni della classe derivata non hanno pertanto nessun privilegio particolare per l'accesso ai dati della classe base.

L'ereditarietà pertanto definisce una relazione di inclusione fra insiemi: gli oggetti della classe derivata sono un sottoinsieme degli oggetti della classe base e perciò sono in numero minore e sono specializzati rispetto a quelli della classe base, nel senso che hanno più operazioni e la descrizione del loro stato interno richiede un maggior numero di strutture dati.

Come vedremo meglio in seguito, l'ereditarietà modellizza una relazione del tipo "è-un" ma non è l'unica relazione possibile fra le classi. Altre relazioni possibili sono l'associazione e l'aggregazione.

Nel definire due classi come una derivata dall'altra[4] è bene tenere presenti i suggerimenti che seguono:

- è consigliabile assegnare le operazioni comuni e i campi dati comuni alla classe base;
- è necessario fare in modo che le operazioni della classe derivata mantengano gli oggetti della classe in stati consistenti come accade per le operazioni della classe base;
- è necessario definire le operazioni della classe in modo che la classe derivata sia chiusa rispetto alle operazioni ereditate perché altrimenti una o più operazioni possono trasformare gli oggetti della classe derivata in oggetti non validi e
- è opportuno definire la classe in modo che le sue classi derivate debbano usare i campi dati per le variazioni di stato e le operazioni per esprimere il proprio comportamento.

Data una classe base ed una classe da questa derivata, è possibile operare una conversione di un oggetto della classe base in un oggetto della classe derivata.

Un oggetto della classe derivata ha almeno tanti elementi (campi dati e operazioni) quanti ne ha uno della classe base[5] per cui risulta possibile assegnare un oggetto di una classe derivata ad uno della classe base con l'unico inconveniente che le informazioni caratteristiche dell'oggetto origine vanno perdute. Oltre a ciò è possibile passare un oggetto di una classe derivata come argomento a una funzione che si aspetta un oggetto della classe base, con gli stessi problemi prima ricordati.

Non è possibile, ovviamente, assegnare un oggetto della classe base ad uno della classe derivata dato che non sarebbe possibile sapere come riempire i campi dati tipici della classe derivata e che significato dare alle operazioni proprie della classe derivata.

2.3 Polimorfismo

La parola polimorfismo significa alla lettera "molteforme". In base a tale etimologia una funzione è detta essere polimorfa se può dare luogo ad azioni diverse in funzione del contesto.

Il polimorfismo si applica alle operazioni degli oggetti e permette di avere operazioni con implementazioni multiple che si differenziano per la cosiddetta signature, ovvero per la lista tipata dei parametri con cui una data operazione viene richiamata a run-time. Una operazione[6] caratterizzata da un comportamento polimorfo è detta essere overloaded, dal momento che è sovraccarica di comportamenti. L'overloading non deve essere confuso con l'overriding in base al quale un metodo in una classe è sovrascritto da un metodo di una classe derivata che ha lo stesso nome ma un comportamento affatto diverso.

3 La progettazione Object Oriented

3.1 Dal problema al programma

Di solito lo stimolo alla stesura di un programma nasce dalla necessità di risolvere un problema descrivibile computazionalmente utilizzando un particolare linguaggio di programmazione e un ambiente sia di sviluppo sia di esecuzione (leggi Sistema Operativo e librerie che formano il cosiddetto supporto run time del programma).

---

4 Se la classe D eredita dalla classe B si dice che D è una classe derivata da B.

5 Si ricorda infatti che il meccanismo di ereditarietà consente di arricchire una classe con operazioni e campi dati senza consentire operazioni di "sottrazione".

6 Nel testo sarà usato anche il termine metodo come sinonimo di operazione.

Le motivazioni che portano alla scelta di un dato ambiente di sviluppo e di un dato ambiente di esecuzione esulano dallo scopo del presente articolo e, a detta dello scrivente, sconfinano spesso nella superstizione se non nel masochismo mentre, dato il titolo del presente articolo, quale sia il linguaggio di programmazione che sarà preso in esame dovrebbe essere, per così, dire patente.

La stesura di un programma è un processo lungo che coinvolge spesso molte persone e che altrettanto spesso si conclude in modo nefasto. Nei casi in cui il processo va a buon fine il prodotto finale è un programma funzionante, "rispettoso" delle specifiche di progetto e conforme alle esigenze dell'eventuale committente[7].

Tradizionalmente e assumendo un punto di vista volutamente semplicistico che linearizza un processo caratterizzato da numerosi feedback, il processo di sviluppo verso il programma che rappresenta il prodotto finale è suddiviso nelle tre fasi di analisi, progettazione e implementazione, che verranno descritte brevemente nei paragrafi che seguono.

La mancanza di linearità che si traduce in feedback da una fase alle precedenti del processo nasce dal fatto i prodotti software evolvono nel tempo sia per l'aggiunta di nuove funzionalità sia per la modifica o l'eliminazione di funzionalità esistenti. Oltre a ciò può capitare che in fase di implementazione risultino evidenti carenze della fase di progetto e anche della fase di analisi per cui si innescano cicli aggiuntivi di sviluppo.

L'OOD facilita la produzione di software facile da modificare e manutenere dal momento che gli oggetti che vengono definiti nel dominio del problema rappresentano entità stabili e fondanti, vedremo meglio nel seguito cosa ciò voglia dire.

3.2 La fase di analisi

Scopo principale della fase di analisi è quello di passare da una comprensione vaga del problema da risolvere ad una sua descrizione precisa, fatta utilizzando anche strumenti formali di descrizione.

Al termine della fase di analisi si ha una descrizione dettagliata che definisce in modo completo e privo di contraddizioni interne i task da svolgere ed inoltre è sia leggibile sia da esperti del dominio del problema sia da programmatori essendo in più rivedibile dagli interessati e confrontabile con la realtà che si propone di modellizzare.

Al termine della fase di analisi si ha un documento che racchiude in sé le caratteristiche di un manuale utente e di un manuale di riferimento ad un livello di dettaglio tale da rimuovere ogni possibile ambiguità. Tale documento contiene la descrizione di ciò che deve essere fatto ma non di come lo si dovrà fare dal momento che, ad esempio, la scelta degli algoritmi da usare viene fatta durante la fase di implementazione.

3.3 La fase di progetto

Nel caso dell'OOD la fase di progetto si traduce nella definizione delle classi mediante la specifica, per ciascuna classe, delle operazioni caratteristiche e delle sue relazioni con le altre classi.

Il risultato finale della fase di progetto è costituito da un insieme di descrizioni delle classi e, per ciascuna di esse, delle sue operazioni e delle relazioni con le altre classi.

3.4 La fase di implementazione

Durante la fase di implementazione si ha la scelta delle strutture dati e del linguaggio di programmazione in cui implementarle.

Oltre a ciò vengono codificate, provate e integrate le classi con le relative operazioni. L'uso dell'OOD incoraggia lo sviluppo incrementale dei programmi mediante l'aggiunta successiva delle classi necessaire.

Durante la fase di implementazione si può far uso di un prototipo che si limita ad illustrare la struttura del prodotto finito senza implementarne tutte le funzionalità. L'adozione del prototipo può avere riflessi sulle fasi da analisi e progetto dal momento che può permettere di arrivare ad una migliore comprensione del problema, comprensione che si riflette in un progetto più lineare ed in una migliore implementazione.

---

7 Mi appello alla benevolenza di tutti coloro che si occupano di Software Engineering, mi rendo conto che quanto precede (e segue) può sembrare un insieme di eresie (e forse lo è) ma non penso sia possibile condensare altrimenti la descrizione del processo di sviluppo, testing e manutenzione del software in alcuni miseri paragrafetti.

L'OOD facilita lo sviluppo di prototipi dal momento che gli oggetti che costituiscono il prototipo sono quelli che faranno parte del programma completo a meno della implementazione delle operazioni non essenziali al funzionamento del prototipo e dato che è spesso fattibile far evolvere il prototipo, se giudicato soddisfacente, nel programma completo di tutte le sue funzionalità.

3.5 Il processo di progettazione OO

Lo scopo principale dell'OOD è quello di decomporre un problema in modo da definire i tipi di dati, ovvero le classi, e per ciascuna di esse le funzionalità e quindi le operazioni eseguibili dai membri della classe.

Il processo di progettazione nel caso dell'OOP è caratterizzato dai seguenti obbiettivi:

- identificazione delle classi;
- identificazione delle funzionalità di ciascuna classe e
- identificazione delle relazioni fra le classi.

Tali obbiettivi non definiscono un processo lineare dato che la definizione di un aspetto di una classe può innescare sia la definizione di altre classi sia la modifica di classi già esistenti.

La definizione di una classe si traduce nella definizione di un tipo di dati astratti. Da questo punto di vista le operazioni di una classe corrispondono a funzioni e procedure che operano su tale tipo di dati.

Al termine del processo di progettazione si ha un elenco di classi con specificate, spesso in modo grafico, le relazioni fra le classi. Di ogni classe viene esposta la finalità e vengono elencate le operazioni.

Si può affermare, a proposito della fase di progettazione, che quanto più questa è svolta in modo accurato tanto più si riducono il tempo necessario per l'implementazione e le successive fasi di test e di debugging.

Un primo passo è quello della definizione delle classi. Una semplice regola pratica è quella di considerare i nomi che compaiono nel problema in corso di specifica come candidati alla definizione delle classi.

Fatto ciò è necessario individuare le operazioni che, come le classi possono corrispondere ai nomi, possono corrispondere ai verbi che compaiono nella descrizione del problema.

Il legame fra classi/nomi e operazioni/verbi deriva dal fatto che le operazioni devono essere associate ad una classe nel senso che per ogni operazione ci deve essere solo una classe responsabile della sua esecuzione.

È possibile attribuire una operazione ad una classe cercando di immaginare come gli oggetti della classe potrebbero usare l'operazione. Se l'uso dell'operazione prevedrebbe l'accesso a strutture dati private di altre classi allora darebbe luogo a inconsistenze e pertanto l'operazione in questione non può essere attribuita alla classe. L'inconsistenza deriva dal fatto che un oggetto accede solo alle operazioni di un altro e, solo in modo indiretto e mediato dalle operazioni, alle sue strutture dati.

Dopo aver individuato le classi ed aver associato loro le operazioni è necessario individuare le relazioni fra le classi.

Le relazioni possibili sono di tre tipi:

- associazione o uso;
- aggregazione o inclusione e
- ereditarietà o specializzazione.

Si dice che una classe A ne usa un'altra B (ovvero che A è associata a B) se gli oggetti della prima manipolano in qualche modo oggetti dell'altra. Se gli oggetti di una classe eseguono i loro compiti senza avere consapevolezza dell'esistenza di un'altra classe con i relativi oggetti allora le due classi sono indipendenti.

Uno degli obbiettivi della progettazione è quello di minimizzare il numero di relazioni ovvero degli accoppiamenti fra le classi: un basso grado di accoppiamento riduce la probabilità di errori in fase di implementazione e inoltre rende più facile la modifica delle diverse classi.

La relazione di aggregazione traduce il fatto che se oggetti di una classe A contengono (non necessariamente come copie ma anche solo come puntatori o handle) oggetti di una classe B allora la classe A contiene la classe B. L'aggregazione è di solito descritta come

una relazione del tipo "ha-un"[8] e può essere caratterizzata da una cardinalità per cui si possono avere relazioni biunivoche (o 1 a 1) e di tipo 1 a molti (o 1 a n).

L'ultimo tipo di relazione è quello di ereditarietà detta anche specializzazione dal momento che gli oggetti della classe derivata sono un raffinamento di quelli della superclasse.

Si dice che una classe derivata D eredita da una classe base o superclasse B se tutti gli oggetti di D sono casi speciali degli oggetti di B. In più le operazioni della classe B (dette di base) sono valide per gli oggetti della classe D che può sia avere operazioni aggiuntive sia arricchire le operazioni di base sia implementarle in modo diverso.

L'ereditarietà definisce una relazione di tipo "è-un" ed è facilmente distinguibile dall'aggregazione. Data la classe dei poligoni e la classe derivata dei triangoli si ha che un triangolo "è-un" poligono e, se si è definita la classe dei segmenti, si ha una relazione di associazione fra tali classi ovvero un triangolo "ha-un" segmento e la relazione ha cardinalità 1:3.

3.6 OOD versus altre tecniche tradizionali di progettazione

Vedremo ora, senza pretesa alcuna di completezza, un breve confronto fra l'OOD e alcune tecniche tradizionali di progettazione quali la decomposizione procedurale e la decomposizione in moduli.

La tecnica di decomposizione procedurale prevede l'identificazione di compiti a cui si associano funzioni e procedure di un dato linguaggio di programmazione. Per l'identificazione si può procedere essenzialmente in due modi.

È possibile scrivere procedure/funzioni che risolvono semplici compiti e poi comporle in procedure/funzioni sempre più sofisticate fino ad ottenere la funzionalità desiderata (tale approccio si dice bottom-up) oppure si può partire da un compito "astratto" e generale e decomporlo in compiti via via più semplici e cooperanti fino ad arrivare a delle operazioni implementabili direttamente nel linguaggio scelto. Tale approccio è detto top-down. Una regola pratica (o del pollice, dall'inglese rule of thumb) per l'identificazione delle procedure/funzioni può essere quella di identificare i verbi che compaiono nella descrizione del problema.

Ciò che differenzia una procedura[9] da una operazione(o metodo) del mondo OO è che ogni operazione è associata alla classe responsabile della sua esecuzione ed è accessibile solo nell'interazione fra oggetti ovvero come risultato di un messaggio di un certo tipo inviato ad un oggetto istanza della classe di cui l'operazione è accessore o mutatore mentre una procedura è utilizzabile in qualunque punto del codice.

Il grosso vantaggio dell'uso delle classi e degli oggetti rispetto alle procedure deriva dal fatto che le classi rappresentano un modo semplice di raggruppare fra loro le operazioni in modo da agevolarne la gestione da parte del programmatore.

Un altro motivo di superiorità delle classi rispetto alle procedure è legato al fatto che le classi realizzano in modo completo l'incapsulamento, nascondendo i dettagli implementativi e le strutture dati interne al resto del codice, mettendo a disposizione le proprie operazioni per l'accesso alle e la modifiche di tali strutture dati.

La tecnica basata sulle procedure in definitiva impone l'uso di un elevato numero di procedure cooperanti e interagenti per la soluzione di compiti non banali senza che queste riescano a proteggere le proprie strutture dati da accessi indesiderati.

La tecnica della decomposizione modulare prevede che un programma sia suddiviso in unità distinte dette moduli. I moduli comunicano fra di loro mediante chiamate di procedura senza che ci sia condivisione di dati.

L'uso dei moduli facilita il compito del programmatore che deve soltanto conoscere l'interfaccia di ciascun modulo così da comporlo in modo corretto con altri moduli senza che questi accedano alle sue strutture dati interne. Il problema principale di questo approccio è legato al fatto che se un modulo implementa una struttura dati e il problema richiede la presenza di una coppia di tali strutture dati non è possibile usare tale modulo istanziandolo due volte ma è necessario definire e implementare un modulo ad hoc. Ad esempio se ho un modulo coda con le operazioni di inserzione, estrazione e di controllo se la coda è piena o è vuota e devo implementare una coppia di code non posso usare due

---

8 L'espressione "ha-un" traduce l'espressione inglese "has-a" come la successiva "è-un" traduce l'espressione "is-a".

9 Lo stesso vale, mutatis mutandis, per le funzioni.

copie dello stesso modulo ma devo definire un modulo che gestisce le due code con le loro operazioni e le loro strutture dati interne. Se invece definisco la classe coda posso istanziare la classe definendo tutti gli oggetti coda di cui ho bisogno, ciascuno con le sue operazioni e le sue strutture dati interne. Una classe permette di definire un numero qualunque di oggetti ognuno dei quali è un modulo.

3.7 Ancora sulle classi

Per sfruttare in pieno la potenza dell'approccio OO è bene inoltre tenere presenti alcuni ulteriori concetti di base. Le classi modellizzano in genere insiemi di elementi ciascuno caratterizzato da più operazioni per cui al momento di individuare una classe ci si deve porre il problema di quali oggetti questa classe descriverà e di quali saranno le operazioni eseguibili su tali oggetti: ad esempio invece di definire una classe *standard output* con le sole operazioni di *open, close* e *write* si ottiene una maggiore generalità definendo la classe *i/o* con operazioni di *read* e *write* oltre che di *open* e *close*.

Oltre a ciò le classi devono essere caratterizzate da un insieme consistente di operazioni significative ovvero da un insieme di operazioni completo ed utilizzabile per svolgere i compiti per i quali si progetta la classe.

Le operazioni dovrebbero essere individuate al momento della definizione della classe e non essere aggiunte incrementalmente man mano che se ne presenta il bisogno. L'aggiunta di operazioni in fase di codifica riflette in genere carenze a livello di progettazione della classe. Un altro problema che dipende da carenze a livello di progettazione è dato dalla impossibilità di svolgere un compito dal momento che non si sa come accedere ai dati privati di una certa classe. Se ciò succede si può o rivedere il progetto delle classi interessate oppure aggiungere una o più operazioni sia di tipo accessore sia di tipo mutatore senza cedere alla tentazione di rendere direttamente pubblici i dati privati in questione e di affolla re l'interfaccia con un numero eccessivo di operazioni.

L'uso di dati privati, imponendo l'introduzione di operazioni per il loro accesso e la loro modifica, potrebbe sembrare un difetto più che un vantaggio. È bene tenere presente tuttavia che la fase di codifica rappresenta solo una delle attività legate alla produzione di software dal momento che sono coinvolte altre attività quali il debugging, il testing e l'aggiornamento con l'aggiunta di nuove caratteristiche e l'eliminazione di caratteristiche obsolete. In tale ottica l'uso di dati privati rappresenta una strategia che facilita la messa a punto e l'aggiornamento dei programmi dato che fornisce un modo per isolare più facilmente gli errori e per far si che eventuali modifiche e aggiunte non si ripercuotano in modo indesiderato e inatteso su altre porzioni del codice.

3.8 Le classi: utilizzatore e progettista

La progettazione e l'implementazione di una classe tira in ballo due figure professionali che sono il progettista della classe e l'utilizzatore della classe che non deve essere confuso con l'utente, ovvero l'utilizzatore del programma al cui interno si trova quella classe particolare.

Il progettista della classe è colui che si occupa del progetto e della implementazione di una classe che sarà usata da altri programmatori detti, pertanto, utenti della classe.

Il progettista punta a definire una classe che sia caratterizzata da una buona interfaccia, in base a quanto esposto nei punti precedenti, e che sia implementata usando sia algoritmi efficienti sia una codifica conveniente e facile da manutenere.

L'utilizzatore vuole, dal canto suo, poter usare la classe solo basandosi sulle sue operazioni che devono rappresentare un insieme abbastanza ampio da essere utile ma non così ampio da non essere maneggevole. L'utilizzatore accede, infatti, all'interfaccia pubblica della classe che contiene le operazioni tipiche della classe ed eseguibili su tutti gli oggetti di quella classe. Il suo obiettivo principale è, in effetti, quello di poter disporre di una toolbox (o scatola degli attrezzi) piena di classi potenti e facili da usare che possano essere utilizzate per un rapido sviluppo delle applicazioni.

3.9 Aspetti formali dell'OOD

Come già visto in uno dei punti precedenti, il primo passo formale nel processo di definizione e implementazione di un programma è la fase di analisi il cui scopo principale è essenzialmente quello di individuare le classi con le relative funzionalità (o operazioni) e le relative relazioni di associazione, aggregazione ed ereditarietà.

Un metodo efficiente per la determinazione degli elementi suddetti è dato dal metodo delle schede CRC. Una scheda CRC è un documento cartaceo che contiene la descrizione di una classe e, a tale scopo, contiene indicazioni sia sui compiti svolti dalla classe, ovvero le sue operazioni, sia sui collaboratori della classe (ovvero le classi con cui questa è in relazione) e infine sui campi dati della classe, ovvero gli elementi della sezione privata della classe che comprendono le strutture dati e le variabili di stato.

Tali schede, una per ognuna delle classi che vengono individuate nella fase di analisi, sono progettate di dimensioni tali da scoraggiare l'assegnazione di un numero eccessivo di operazioni ad una classe e il loro ordinamento può fornire utili suggerimenti sulle eventuali relazioni fra le classi. Come già visto, un modo semplice per individuare le classi con le relative operazioni è quello di elencare i nomi e i verbi che la fase di analisi evidenzia nel dominio del problema che si vuole risolvere.

Fatto ciò è necessario isolare uno dei compiti e, a partire da esso, simularne lo svolgimento, in modo da scoprire quale classe dovrebbe svolgerlo e in che modo, individuando così le classi che sono coinvolte nel processo. Perché il tutto sia significativo è necessario partire da una delle operazioni chiave del problema e non da una delle operazioni accessorie o minori.

Il principale limite delle schede CRC è costituito dal fatto che non sono adatte per tenere traccia in modo soddisfacente delle relazioni fra le classi né per una loro documentazione e pertanto sono in pratica utilizzabili solo nella fase iniziale dell'analisi.

Un aspetto importante di ausilio alla progettazione è rappresentato dalla possibilità di suddividere le classi in categorie, in modo da individuare quali classi siano caratterizzate da schemi comuni.

Una categoria possibile e facile da individuare è quella caratterizzata dai cosiddetti oggetti tangibili, ovvero dagli oggetti direttamente visibili nel dominio del problema. Un'altra categoria facile è quella che individua le interfacce e i dispositivi di sistema le cui classi sono facili da definire essendo sufficiente considerare le risorse del sistema ospite e le relative interazioni.

Un'altra categoria comprende le classi utili per modellizzare la registrazione di attività avvenute nel passato oppure che è necessario svolgere nel futuro. Due categorie importanti raggruppano le classi che caratterizzano sia i reali utenti del programma (e permettono l'accesso a utenti diversi con ruoli diversi quali utente normale e amministratore) sia il sistema nella sua globalità sia i suoi elementi costitutivi e il cui ruolo è quello di gestire le inizializzazioni e lo spegnimento e avviare il flusso di dati in ingresso al sistema. Altre categorie, infine, sono una categoria che raggruppa le classi di base (che incapsulano i tipi di base del linguaggio, vedi oltre, quali stringhe, vettori, interi e così via), una categoria che raggruppa le classi contenitore (usate per memorizzare e recuperare informazioni) e una categoria che raggruppa gli schemi di collaborazione.

Un altro punto chiave è rappresentato dall'individuazione delle relazioni (associazione, aggregazione ed ereditarietà) fra le classi.

La relazione fra classi più facile da riconoscere è l'associazione dato che per la sua individuazione è sufficiente considerare la porzione delle schede CRC che elenca le collaborazioni fra le classi e poiché la collaborazione implica l'associazione.

L'aggregazione esprime la relazione "ha-un" ed è soddisfatta se ciascun oggetto di una classe contiene oppure gestisce in modo esclusivo oggetti di un'altra per cui si dice che una classe è aggregata ad un'altra se la prima contiene oggetti della seconda. Se un oggetto contiene un campo di tipo base oppure di una classe base (ad esempio una stringa) allora tale campo lo si deve vedere come un attributo e non come originante una aggregazione. Ad esempio data la classe *Veicoli* e data la classe *Mezzi di trasmissione* ho che la classe *Automobili* è sottoclasse della prima e la classe *Semiassi* è una sottoclasse della seconda ed in più gli oggetti della classe *Automobili* sono aggregati agli oggetti della classe *Semiassi* dato che "un'automobile ha un semiasse".

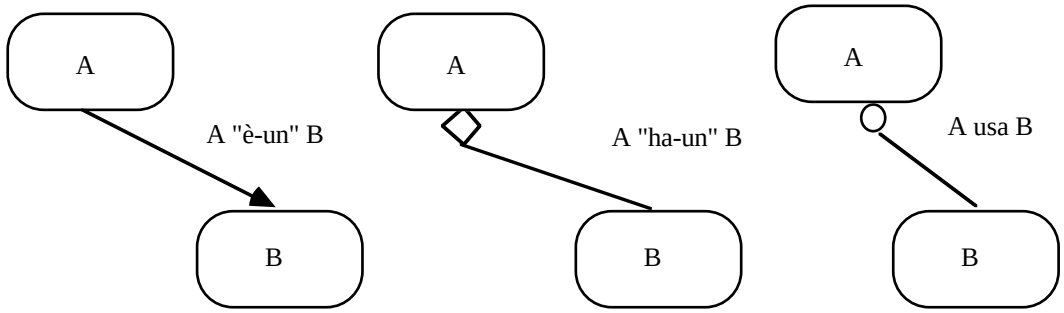
L'ereditarietà è rappresentata da una relazione del tipo "è-un" e può essere individuata se si riesce a determinare che ogni oggetto della prima classe si conforma logicamente alla seconda classe e in più ha un insieme di proprietà specializzate e/o aggiunte.

La relazione di ereditarietà traduce una relazione di inclusione fra insiemi (l'insieme delle automobili è un sottoinsieme dell'insieme dei veicoli) e in più gli oggetti della classe derivata sono una specializzazione di quelli della classe base.

In assenza di una classe comune può tuttavia essere difficile stabilire che una classe A eredita da una classe B. Se nel corso di definizione delle classi si individuano classi che

condividono certe operazioni un modo di procedere è quello di individuare una classe base caratterizzata da tali operazioni comuni e impostare le altre classi come derivate. Ciò facilita il lavoro di implementazione dato che il codice comune viene fornito solo nella classe base dal momento che le classi derivate si limitano a ereditarlo, eventualmente arricchendolo e/o specializzandolo.

I metodi grafici di rappresentazione delle classi (vedi figura) mettono a disposizione del progettista uno strumento potente e flessibile considerando che l'uso di grafici contenenti informazioni sulle relazioni fra le classi consente di organizzare al meglio le relazioni fra le classe coinvolte nel progetto, sebbene sia consigliabile in ogni caso l'uso di documentazione scritta per la descrizione dei dettagli delle classi.



Nei diagrammi le classi sono tracciate come scatole al cui interno sono riportati il nome della classe e, opzionalmente, i nomi dei campi dati e delle operazioni della classe, se non di tutte almeno di quelle più significative. Sui diagrammi sono rappresentate le relazioni fra le classi (vedi la figura precedente) mediante elementi grafici che permettono di individuare se due classi sono in relazione fra di loro o no e di che tipo sia tale relazione (associazione, aggregazione ed ereditarietà).

Nel caso dell'aggregazione è possibile specificare la cardinalità agli estremi della connessione utilizzando la forma 1:1 oppure 1:* o 1:1...* per, rispettivamente, relazioni biunivoche oppure uno a molti, dove molti, nel caso 1:*, può essere anche 0 mentre nell'altro vuol dire 1 o più di 1. Eventuali aggregazioni di tipo molti a uno o molti a molti (n:1 o n:m) devono essere espresse per mezzo di associazioni di tipo più generale quali le interfacce (vedi oltre).

Le relazioni di aggregazione sono implementate mediante campi dati sebbene non tutti i campi dati (ad esempio i campi dati di tipo numerico) debbano essere individuati come associazioni in un diagramma ma piuttosto dovrebbero essere visti come attributi di una classe. Se ho A o— B (ovvero A usa B) ho in genere una relazione di associazione asimmetrica per cui può accadere che A usi B e anche B usi A ma la natura di tale uso sia diversa. Nel caso delle associazioni è possibile avere relazioni di tipo molti a molti. Si ricorda che il significato dell'associazione è il seguente: una classe A usa una classe B se le operazioni degli oggetti di A richiedono di accedere ad oggetti della classe B.

L'associazione stabilisce legami fra le classi che arrivano al livello implementativo, nel senso che cambiamenti nella classe B si possono ripercuotere sulla struttura della classe A potendo richiedere l'aggiunta di operazioni o la modifica di quelle esistenti.

La rappresentazione delle relazioni fra le classi definisce la parte statica del progetto, la parte dinamica essendo costituita dalle interazioni run-time fra gli oggetti delle classi, interazioni che, nel modello OO, si traducono nel cosiddetto scambio di messaggi fra gli oggetti. Tale scambio di messaggi può essere rappresentato mediante il cosiddetto diagramma dei messaggi che mostra il flusso dei messaggi che si originano dalla esecuzione di certe operazioni.

In un diagramma dei messaggi ogni oggetto è rappresentato da un rettangolo che contiene il nome dell'oggetto e la classe cui appartiene. Tali rettangoli sono uniti da archi orientati che puntano all'oggetto responsabile dell'esecuzione di una operazione il cui nome costituisce l'etichetta dell'arco stesso. In tali diagrammi ogni chiamata ad una funzione è rappresentata con una linea che unisce l'oggetto chiamante con l'oggetto che ha tale operazione fra le operazioni ammissibili ed è pertanto detto essere responsabile della sua esecuzione[10].

10 Si ricorda che il paradigma prevede che ogni operazione sia eseguita da al più una classe.

Gli archi oltre ad essere etichettati con i nomi delle varie operazioni sono anche numerati sequenzialmente in modo da stabilire un ordinamento fra le chiamate alle varie funzioni sebbene tale caratteristica sia spesso più un intralcio che un ausilio, considerando anche che lo scopo principale di tali diagrammi è quello di consentire l'individuazione degli oggetti responsabili delle operazioni che un dato oggetto deve eseguire per svolgere un certo compito. Oltre a ciò gli archi possono avere dei marker per caratterizzare ulteriormente le relazioni fra oggetti la cui trattazione esula dallo scopo delle presenti note.

È bene infine notare che se da un lato è molto utile tracciare i diagrammi che esplicitano le relazioni fra le classi di solito i diagrammi dei messaggi, perché siano effettivamente di ausilio alla progettazione, devono essere tracciati limitatamente alle operazioni più interessanti e la cui implementazione non sia ovvia e limitatamente agli aspetti più significativi di tali interazioni.

Le schede CRC, di cui si è parlato in precedenza, contengono una quantità limitata di informazioni sulle classi quali nome della classe, delle sue operazioni e nomi delle classi correlate mentre i diagrammi delle classi aggiungono informazioni sul tipo di relazione che esiste fra due classi. Tuttavia, al momento di passare alla implementazione delle classi sono necessarie ulteriori informazioni che vengono raccolte in elementi detti form, sorta di questionari che contengono domande sulle caratteristiche importanti di ciascuna classe. In genere si ha un form per ogni classe come per ognuna delle operazioni di una classe e ognuno di tali form può essere compilato usando ad esempio una sintassi Java-like (se si è deciso di usare il linguaggio Java a livello di implementazione) per specificare le funzioni con i loro argomenti ed i campi dati. Si fa notare come gli strumenti descritti in pratica si traducano nella produzione di documentazione separata dal codice con conseguenti problemi di coerenza per cui, soprattutto nel caso di progetti di medie dimensioni, è sicuramente più sicuro produrre del codice il più possibile autodocumentato utilizzando anche tool di Computer Aided Software Engineering (CASE) attualmente disponibili come prodotti commerciali ma la cui trattazione esula sia dalle competenze dello scrivente sia dagli ambiti delle presenti note.

Quali che siano gli strumenti utilizzati è bene tenere presente che una fase chiave del processo di sviluppo del software è dato dalla fase di implementazione durante la quale sono necessarie tutte le informazioni che consentano di specificare le strutture dati necessarie, di implementare i legami fra gli oggetti, di codificare le operazioni delle classi per le quali può essere necessario aggiungere campi dati privati ed implementare operazioni ausiliarie private della classe e infine prevedere funzioni per la gestione di risorse esterne quali file, memoria e dispositivi di I/O.

3.10 Le classi: vademecum per il progettista

Riassumendo, nella progettazione delle classi è consigliabile attenersi almeno ai seguenti suggerimenti:

- considerare generalizzazioni ragionevoli;
- minimizzare il grado di accoppiamento fra le classi in modo da minimizzare la portata delle modifiche ad una classe;
- spezzare le classi che svolgono troppi compiti in più classi ed evitare la definizione di classi che definiscono un solo oggetto;
- non introdurre operazioni se non effettivamente utilizzate e/o utilizzabili;
- evitare di attribuire ad una singola classe compiti scorrelati fra di loro;
- usare l'ereditarietà per esprimere funzionalità ripetute;
- mantenere le responsabilità (i compiti ovvero le operazioni) di una classe ad un solo livello di astrazione senza mescolare fra loro livelli di astrazione distinti;
- scegliere in modo oculato i nomi delle classi e delle relative operazioni;
- non modellizzare comportamenti che non hanno relazioni con il problema in esame e
- non modellizzare aggregazioni non necessarie fra le classi.

4 Una breve introduzione a Java

Il linguaggio Java è un linguaggio di programmazione orientato agli oggetti robusto i indipendente dall'architettura. L'indipendenza viene ottenuta mediante la compilazione del codice sorgente in una forma intermedia detta *bytecode* che viene poi interpretata dalla Java Virtual Machine che definisce un ambiente protetto di esecuzione che impedisce al codice di accedere alle risorse reali del sistema ospite. Tale caratteristica garantisce al linguaggio una elevata sicurezza che lo rende adatto ad applicazioni in rete.

L'unità base di un programma scritto nel linguaggio Java è detta compilation unit ed è un file di testo che contiene la definizione di una o più classi e il cui nome è del tipo *nome.java*.

La compilazione di una compilation unit dà origine ad un certo numero di file, uno per ognuna delle classi definite nella compilation unit stessa e il cui nome è *nome_classe.class*.

Perché l'interprete riesca ad eseguire il programma è necessario che ci sia una classe nella quale è definito il metodo *main* dichiarato *public* e *static* e al quale è passato un array (anche vuoto) di stringhe che contiene i parametri passati al programma sulla linea di comando.

Per poter eseguire un programma Java[11] è necessario avere a disposizione sia il compilatore sia l'interprete sia il supporto run-time comprendente l'insieme delle classi predefinite del linguaggio e che possono essere importate nei file sorgenti (vedi oltre).

I passi necessari per l'esecuzione di un semplice programma sono:

- creazione del file sorgente con la definizione delle classi e del metodo main;
- compilazione del file sorgente (con il comando javac nome.java) e
- esecuzione del programma attraverso l'interprete con il comando java nome.

In quanto segue alcuni elementi marginali del linguaggio, marginali almeno dal punto di vista del presente lavoro, quali tipi di dati base e strutture di controllo verranno descritti solo brevemente.

Le strutture di controllo non si discostano molto da quelle tradizionalmente presenti in moltissimi altri linguaggi anche non OO e rappresentano il "lato sporco" di Java ovvero la sua parte non OO. Dei tipi di dati base ci si limiterà a darne una scarna descrizione. Si fa notare come i tipi base quali gli interi siano caratterizzati di default da operazioni non OO dette operatori sebbene sia possibile definire una classe di interi, detta wrapper, i cui oggetti sono accessibili mediante operazioni in modo che sia rispettato in pieno il paradigma OO. In questa ottica se **a** e **b** sono variabili di tipo intero[12], si può scrivere **a + b** oppure **a.sum(b)** per rappresentare la somma fra due interi nei due paradigmi, quello tipico della programmazione imperativa e quello tipico della programmazione OO.

4.1 Tipi di base e strutture di controllo

I tipi base o elementari del linguaggio comprendono i tipi classici della maggior parte dei linguaggi di programmazione quali numeri interi e in virgola mobile, costanti booleane, carattere e stringa.

I tipi interi comprendono i seguenti tipi: byte, short, int e long. Oltre ai tipi interi sono disponibili i tipi in virgola mobile come float e double, il tipo carattere o char e i tipi booleani che non possono essere trattati come interi. Tutti i tipi hanno una dimensione in byte costante e indipendente dall'architettura hardware e ciò aumenta la portabilità del codice. Il tipo stringa è visto come una classe (String) di oggetti costanti mentre per la definizione di quelle che in prima approssimazione si possono dire variabili di tipo stringa il linguaggio mette a disposizione la classe StringBuffer (vedi oltre).

Il linguaggio mette a disposizione l'operazione di casting ovvero la conversione forzata dei tipi. In Java il casting è possibile in modo automatico solo nel caso detto di promotion ovvero quando si promuove un valore di un tipo ad uno di un tipo più "ampio"[13], ad esempio un tipo byte ad un tipo int. Nel caso inverso si deve fare un casting esplicito che può tradursi in una perdita di informazioni. Il casting esplicito ha una sintassi C-like (del tipo **byte b = (byte) a** con **a** di tipo **int**) ed è di solito detta narrowing nel caso, ad esempio, di mappatura di un tipo come int su uno come byte.

Con il termine promozione automatica si indica il meccanismo in base al quale in una espressione che coinvolge tipi diversi quelli meno "ampi" sono, in genere, promossi a quello più "ampio" prima che l'espressione venga valutata. Fanno eccezione le operazioni che coinvolgono le stringhe. In questo caso, ad esempio, la concatenazione indicata con il simbolo +, se il primo elemento è una stringa o un carattere anche gli altri sono visti come stringhe/caratteri e concatenati mentre se il primo è un int anche eventuali caratteri sono convertiti ad int e sommati fra di loro. La promozione automatica spesso si traduce nella necessità di un casting esplicito per eseguire le opportune operazioni di narrowing: ad

---

11 La frase "un progamma Java" deve ovviamente essere letta come "un programma scritto nel linguaggio Java".

12 Va de sé che in queste brevi note non si entrerà in dettaglio relativamente ai nomi legali delle variabili del linguaggio ed amenità del genere.

13 L'ampiezza è misurata dalla dimensione in byte del tipo per cui, ad esempio, il tipo byte occupa 1 byte, il tipo short 2, il tipo int 4 e infine il tipo long 8 e pertanto si può affermare che il tipo int è più ampio del tipo byte ed è meno ampio del tipo long.

esempio nel caso di una espressione aritmetica che coinvolge long e int tutto viene promosso a long prima che questa sia valutata e se il risultato deve essere assegnato ad una variabile di tipo int è necessario un casting esplicito.

Altri tipi base sono gli array monodimensionali e multidimensionali. Un array monodimensionale in Java rappresenta un gruppo di variabili dello stesso tipo cui ci si riferisce come una entità unica i cui elementi sono manipolabili individualmente. Gli array sono dichiarati con una istruzione di dichiarazione che ne consente l'inizializzazione automatica. Un esempio di array monodimensionale inizializzato staticamente al momento della definizione è il seguente:

int giorni_mese[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}

che definisce un vettore di 12 interi assegnando i valori ai singoli elementi in modo che, ad esempio, giorni_mese[2] abbia valore 31, dato che il primo elemento di un array ha indice 0.

Gli array multidimensionali sono a rigore array di array, per maggiori dettagli si rimanda ai testi citati in bibliografia.

Gli array in Java sono sempre allocati nello heap ed hanno una dimensione determinata da una espressione valutata a run time, sebbene una volta che l'array sia stato istanziato la sua dimensione sia una costante e non possa più essere variata.

In Java è possibile definire un array di interi come segue:

int [] numeri **= new int** [n]

che definisce un array di n interi, dove n può essere valutato anche a run time ma una volta che ha un valore questi è la dimensione dell'array che non può essere variata. Definito un array con una certa dimensione è possibile individuare tale dimensione con l'operazione **numeri.length ()** la cui sintassi (esaminata più in dettaglio nel seguito) indica come si stia lavorando in un mondo OO. Un'altra proprietà degli array è che se si assegna un array ad un altro entrambi puntano alla stessa struttura dati per cui una variabile come numeri è a rigore un puntatore ad un array. L'ultima caratteristica degna di nota è rappresentata dal fatto che l'operatore [] con il quale si accedono gli elementi di un array è controllato a run time per cui riferimenti ad elementi di un array con valori illegali dell'indice danno luogo ad eccezioni a run time che devono essere gestite esplicitamente pena la terminazione anomala dell'applicazione.

Le strutture di controllo del flusso, infine, comprendono elementi classici di molti linguaggi imperativi e consentono il controllo del flusso delle istruzioni secondo modalità che ovviamente non sono OO.

Le strutture base comprendono il costrutto **if ... else**, il costrutto per le scelte multiple **switch**, i costrutti per i cicli **for**, **while** e **do-while** (questo analogo al repeat-until) e i comandi **break** (che consente di interrompere un ciclo facendo in modo che l'esecuzione prosegua da una etichetta specificata a fianco di una istruzione), **return** (che causa il ritorno immediato da una subroutine al chiamante) e **continue** (che fa in modo che un ciclo prosegua dalla iterazione successiva a quella in cui viene eseguita tale istruzione per cui le istruzioni successive della iterazione corrente non sono eseguite ma lo potranno essere alla iterazione successiva).

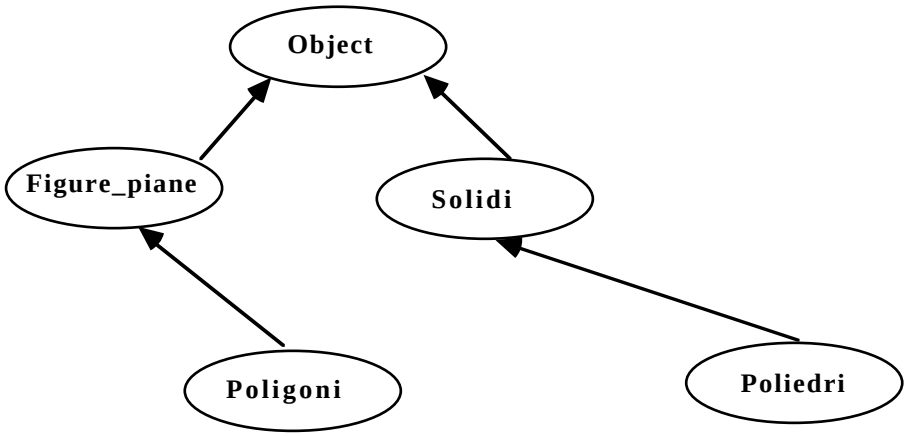
Altre istruzioni che introducono variazioni nel flusso di un programma sono quelle per la gestione delle eccezioni e degli errori quali **try**, **catch**, **throw** e **finally** che verranno esaminate più in dettaglio nel seguito.

4.2 Classi e oggetti

Una classe rappresenta una sorta di template per i propri oggetti, ovvero definisce un tipo di dati di cui gli oggetti costituiscono le istanze. Una classe definisce, pertanto:

- il formato degli elementi di un insieme di elementi correlati detti oggetti (aspetto statico);
- il comportamento degli elementi di tale insieme (aspetto dinamico).

A tale scopo una classe è definita mediante la struttura interna dei suoi oggetti (il formato) e dalla sua interfaccia funzionale (il comportamento) definita per mezzo dei metodi (detti anche operazioni).



Nel linguaggio Java le classi formano un albero di cui la classe *Object* rappresenta la radice. Ogni classe definita discende in un modo o nell'altro da tale classe (vedi figura qui sopra) ed in più ogni classe ha una sola classe da cui deriva dal momento che Java non definisce il concetto di ereditarietà multipla.

Ogni classe viene definita o come estensione della classe *Object* oppure di una classe più generale detta superclasse.

La definizione di una classe avviene, infatti, con una dichiarazione quale la seguente:

```
class    nome_classe [extends  nome_superclasse] {

         <elenco variabili con tipo>

         type  nome_metodo-1  (lista tipata dei parametri) {
               <corpo del metodo>
         }
         .....
         type  nome_metodo-n  (lista tipata dei parametri) {
               <corpo del metodo>
         }
}
```

dove gli elementi in grassetto individuano i cosiddetti elementi sintattici obbligatori[14], la parte fra [] se presente individua la superclasse della classe che si sta definendo (se tale parte è mancante la classe discende direttamente dalla classe *Object*) e l'elenco delle variabili con tipo ha la sintassi seguente:

```
type1    var1, var2;
type2    var3, ....., var-n;
```

e una lista tipata di parametri è composta da un elenco di coppie **tipo variabile** separate dal carattere ",". Un altro modo di definire una classe è il seguente:

```
class    nome_classe [extends  nome_superclasse] {
         public type    nome_metodo-1  (lista tipata dei parametri) {
               <corpo del metodo>
         }
         .....
         public type    nome_metodo-n  (lista tipata dei parametri) {
               <corpo del metodo>
```

14 Tale convenzione sarà usata più o meno coerentemente anche nel seguito, intendendo per elementi sintattici obbligatori tutti gli elementi caratteristici di un costrutto del linguaggio.

```
        }
        <elenco variabili definite private e con tipo type>
}
```

nel quale i metodi sono esplicitamente definiti come **public**, ovvero accessibili dall'esterno della classe, mentre le variabili sono definite come **private**, ovvero non visibili dall'esterno della classe (incapsulamento). Le operazioni possono essere definita anche come **private** se sono locali alla classe e non accessibili dal suo esterno mentre non esiste, se non come concetto astratto, una differenza fra metodi accessori e metodi modificatori.

Tutti gli oggetti sono allocati nello heap e la loro istanziazione avviene con una istruzione la cui sintassi è la seguente:

tipo nome_oggetto = **new** **tipo** (lista_parametri)

nella quale il **tipo** individua la classe dell'oggetto, la parola chiave **nw** fa in modo che l'oggetto sia creato e la lista dei parametri rappresenta l'insieme dei valori utilizzati da un metodo particolare, detto costruttore (vedi oltre) che viene eseguito al momento della istanziazione dell'oggetto stesso. La variabile *nome_oggetto* agisce come un puntatore in modo che la copia di un oggetto fatta come nell'esempio che segue:

Point p = **new Point** (10, 20)
Point q = p

si traduca non nella copia effettiva ma solo nella definizione di un riferimento ulteriore allo stesso oggetto mentre la dulpicazione richiede la creazione di due oggetti e l'assegnamento dei valori contenuti nei campi del primo alle variabili dell'altro. Ogni volta che un oggetto non ha più riferimenti ad esso si ha una deallocazione della memoria ad esso allocata mediante una procedura di *garbage collection.* Tale caratteristica nel caso di oggetti che gestiscono risorse esterne non esime dalla definizione di metodi ad hoc detti distruttori. Un'altra caratteristica degli oggetti definiti in Java è che le variabili locali agli oggetti devono essere esplicitamente inizializzate prima del loro uso dato che la loro mancata inizializzazione dà luogo ad errori di compilazione. Tutte le altre variabili, compresi i campi dati delle classi, sono inizializzate a *null* in modo che tutti i valori di tipo puntatore o hanno valore *null* oppure hanno il contenuto restituito da una chiamata a **new.**

4.3 Ancora su classi e oggetti

Una classe in Java contiene istanze di variabili e di metodi e definisce un template le cui istanze sono dette oggetti. La creazione di una nuova classe aggiunge un nuovo tipo con le operazioni eseguibili sulle variabili di quel tipo (gli oggetti) mentre il nome della classe rappresenta uno dei tipi delle variabili che è possibile dichiarare.

Se ho una classe *Coordinate* posso dichiarare una variabile di tale tipo (ovvero un oggetto della classe *Coordinate*) come segue:

Coordinate p**;**

ma in questo caso la variabile p non punta a nessun oggetto reale della classe *Coordinate*. La variabile **p** è in effetti un puntatore che però non è possibile manipolare come fosse un intero e che in questo caso è inizializzata ad un valore "neutro" e cioè a *null*.

Ogni classe è caratterizzata come è noto da variabili locali alla classe, dette istanze di variabili, per cui posso avere:

```
class Coordinate {

   int    x, y;      // coordinate cartesiane
   float  r, ϑ;      // coordinate polari
}
```

La dichiarazione di un oggetto di una classe avviene utilizzando l'operatore **new** con il quale si ottiene la creazione di una istanza della classe e l'assegnazione del suo riferimento ad una variabile di tale tipo.

Ad esempio:

Coordinate p = **new Coordinate ();**

oppure in generale

type nome_variabile = **new type ([lista_parametri_per_il_costruttore]);**

Nell'esempio la variabile **p** è un riferimento ad una istanza della classe *Coordinate* in modo che una eventuale istruzione del tipo

Coordinate p = q;

crea un ulteriore riferimento allo stesso oggetto senza che nessuna memoria sia allocata per la variabile **q** in modo che se si ha

p = null;

l'oggetto originario rimane riferito dalla variabile **p** e solo quando anche tale variabile assume il valore *null* allora l'oggetto sarà non più referenziato e perciò verrà distrutto dal *garbage collector* .

Ogni oggetto creato con un comando **new** possiede le proprie istanze di variabili per cui se ho:

Coordinate p = **new Coordinate ();**
Coordinate q = **new Coordinate ();**

sia **p** sia **q** possiedono le variabili **x** e **y** (ad esempio) in modo che sia possibile riferirle ed assegnarle indipendentemente le une dalle altre

p.x = 2;
q.x = 4;

L'accesso alle variabili degli oggetti avviene mediante l'operatore dot (.) che consente l'accesso alle variabili di dichiarate tipo *public* (oppure senza specificazione, vedi oltre). La sintassi generale dell'operatore dot è la seguente:

nome_oggetto.nome_variabile

Oltre alle variabili pubbliche un oggetto è caratterizzato da un certo numero di metodi pubblici ed inoltre può avere metodi e variabili private (ovvero dichiarate di tipo **private**, vedi oltre). Un metodo è una operazione eseguibile su oggetti di una data classe che può accedere direttamente alle variabili sia pubbliche sia private degli oggetti della classe senza che sia necessario usare l'operatore dot dato che le variabili appartengono allo stesso scope del metodo.

La sintassi della dichiarazione di un metodo è la seguente:

type nome_del_metodo **(**[lista_tipata_dei_parametri_formali]**) {**
    <corpo del metodo>
}

nella quale **type** è un tipo qualunque oppure ha il valore **void** se il metodo non restituisce nulla e la lista_tipata_dei_parametri_formali rappresenta l'insieme dei parametri formali necessari per invocare il metodo, ciascuno con il proprio tipo. Tale lista è opzionale da cui la coppia di []. Nell'esempio della classe *Coordinate* è possibile avere:

class Coordinate **{**

```
int    x, y;     // coordinate cartesiane
float  r, ϑ;     // coordinate polari
```

```
    void init (int a, int b) {
        x = a;
        y = b;
    }
    void init (float a, float b) {
        r = a;
        ϑ = b;
    }

    <altri metodi>

}
```

La classe ha un metodo **init** polimorfo[15], ovvero un metodo del quale si hanno due versioni. Quale delle due venga scelta dipende a run-time in base alla lista dei parametri effettivi (detta signature) usata al momento della chiamata del metodo per cui se si ha

p.init (2, 3);

si usa la prima forma di **init** mentre se si ha

p.init (3.6f, 62.5f);

si richiama il metodo nella seconda forma.

I metodi, come visto, accedono alle variabili locali senza usare l'operatore dot. La sintassi per l'accesso ai metodi di una classe, come visto nell'esempio, è la seguente:

nome_oggetto.nome_metodo (lista_parametri_attuali);

in cui la lista dei parametri attuali è un elenco di variabili che in numero e tipo corrispondono ad una delle signature del metodo. Nell'esempio il valore 2 sostituisce il parametro formale **a** e viene assegnato alla variabile **x** e, in modo analogo, 3 sostituisce il parametro formale **b** e viene assegnato alla variabile **y**. Nell'altro caso le sostituzioni coinvolgono gli stessi parametri formali mentre gli assegnamenti coinvolgono le variabili **r** e ϑ.

Dall'interno di un metodo è possibile riferire l'oggetto corrente mediante la parola chiave **this**. Ad esempio se si richiama il metodo **p.init** all'interno di tale metodo la parola chiave **this** punta allo stesso oggetto cui punta **p** (vedi oltre).

Ogni oggetto, anche della stessa classe o di una diversa possiede la propria istanza di **this**.

4.4 Esempi di classi: vettori e stringhe

Vedremo ora due classi tipiche, fornite dal linguaggio come predefinite e/o inserite in librerie che fanno parte del supporto run time: vettori e stringhe. Nel caso delle stringhe vedremo come, essendo queste descritte dalla classe *String* come essenzialmente delle costanti, sia stato necessario definire una classe *StringBuffer* per consentire la descrizione di stringhe variabili.

Il problema principale con gli array come definiti in Java è che non è possibile variarne la dimensione una volta che questa è stata allocata. Per eliminare tale vincolo la libreria **java.util** è caratterizzata da una classe *Vector* che implementa un array di dimensione crescente utilizzando un artificio in base al quale se un primo array si esaurisce ne viene creato uno di dimensioni doppie che si sostituisce al precedente e ne possiede il contenuto nelle analoghe posizioni. La classe *Vector* , i cui elementi saranno detti vettori nel seguito, deve andare bene per tutti i possibili tipi e ciò è reso possibile dal fatto che tale classe memorizza valori della classe *Object*, che è la classe base del linguaggio, quella da cui tutte le altre discendono anche array e stringhe. Volendo definire un vettore di numeri si

15 Il polimorfismo dà luogo all'overloading dei metodi ed è implementato mediante il binding dinamico.

può usare una classe wrapper come *Integer* che memorizza interi ed è, come le altre discendente, della classe *Object*. Quando un valore viene assegnato ad un elemento di un vettore con l'operazione **nome_vettore.addElement** (che aumenta la dimensione del vettore di 1 e aggiunge il nuovo elemento al vettore) su questi viene eseguito un casting al tipo Object, operazione che non crea alcun problema. Altre operazioni della classe, riportate solo per completezza, sono la **nome_vettore.size ()** che restituisce il numero di elementi correntemente memorizzati nel vettore e la coppia **nome_vettore.elementAt(**index**)** e **nome_vettore.setElementAt(**obj**,** index**)**: la prima restituisce l'oggetto contenuto nella locazione specificata dalla variabile **index** e la seconda memorizza l'oggetto passato come parametro nella locazione specificata dalla variabile **index**. Dato il casting implicito effettuato al momento dell'inserimento, il recupero di un oggetto da un vettore rende necessaria una operazione di casting esplicito al tipo effettivo dell'oggetto che si recupera.

Per quanto riguarda le stringhe si ha che Java possiede una classe di libreria per le stringhe detta *String* i cui elementi sono immutabili nel senso che una volta che sono stati definiti non è più possibile modificarli con assegnamenti diretti di elementi di una stringa mentre è possibile assegnare un nuovo valore ad una variabile già assegnata ovvero se ho:

String messaggio = "Errore di ascesso in memoria"

l'operazione

messaggio[11] = 'c'

dà luogo ad un errore mentre è possibile riassegnare la variabile come segue

messaggio = "Errore di accesso in memoria"

Un modo di editare una stringa è quello di estrarne una parte da concatenare con altre stringhe e da assegnare anche alla stessa variabile. Una tale operazione di estrazione è possibile con l'operazione **substring** la cui sintassi è

messaggio.substring(primo_carattere, ultimo_carattere)

Nel caso si debbano eseguire numerose operazioni di editing è più semplice ed efficiente utilizzare una classe appositamente definita nel linguaggio e detta *StringBuffer*.

Tale classe consente la definizione di stringhe di caratteri editabili e di cui si può variare la lunghezza nel senso che è possibile inserire sia caratteri sia porzioni di stringa in una posizione qualunque di un oggetto della classe *StringBuffer*.

Una volta che l'elaborazione di tale oggetto è terminata è possibile assegnarlo ad un oggetto della classe *String* mediante una semplice operazione di casting esplicito, in modo da migliorare l'efficienza di utilizzo di tale oggetto.

Allo stesso modo di altri oggetti in Java, le stringhe come oggetti della classe *String* sono puntatori in questo caso ad oggetti immutabili.

Sugli elementi della classe *String* il linguaggio definisce l'operatore di concatenazione indicato con il simbolo + e che causa al conversione degli operatori in stringhe se il primo operando è di tipo *String*, mediante l'uso implicito dell'operazione di conversione *toString*, tipica della classe.

Altre operazioni definite su oggetti della classe *String* sono l'operazione **equals** per il test di uguaglianza fra stringhe e l'operazione **compareTo** per il confronto lessicografico fra le stringhe.

La sintassi della **equals** è la seguente

string1.equals(string2)

e restituisce *true* se le due stringhe hanno contenuto uguale o *false* altrimenti mentre la **compareTo** ha una sintassi del tipo

string1.compareTo(string2)

che restituisce un valore negativo se string1 precede string2 nell'ordinamento lessicografico, un valore nullo se le due stringhe sono uguali o un valore positivo se string1 segue string2 nell'ordinamento lessicografico.

La classe *String*, in breve, consente di definire oggetti che puntano ad array non modificabili di caratteri. La classe *StringBuffer*, d'altro canto, consente di creare stringhe modificabili dinamicamente.

Entrambe le classi sono dichiarate come **final** dal momento che è possibile definire una classe che ne rappresenti l'estensione e ciò per motivi di efficienza.

La classe *StringBuffer* permette di rappresentare sequenze riscrivibili di caratteri la cui lunghezza può aumentare.

Vediamo con l'ausilio di un esempio alcune operazioni tipiche della classe *StringBuffer*.

Il metodo di base che in questo caso viene richiamato implicitamente al momento di definire una variabile del tipo *StringBuffer* è il costruttore della classe che alloca di default spazio per 16 char[16] sebbene si possa passare come parametro un valore string nel qual caso il costruttore alloca uno spazio pari a 16 char più il numero di caratteri allocati per il parametro. Se ad esempio ho:

```
StringBuffer dynamic = new StringBuffer ("Stringa di prova");
```

con

```
dynamic.length ();
```

si ottiene la lunghezza in caratteri della stringa (in questo caso 16) mentre con

```
dynamic.capacity ();
```

si ottiene la capacità totale in caratteri allocata per la stringa, capacità che in questo caso è pari a 32.

La classe *StringBuffer* è caratterizzata da altri metodi fra i quali si segnalano i seguenti:

ensureCapacity	che fissa la dimensione del buffer;
setLength	che imposta la lunghezza del buffer a valori inferiori a quelli allocati;
charAt	che estrae un char da una string;
setCharAt	che modifica un char all'interno di una string;
getChars	che copia una porzione di una string in un array dinamico di char;
append	che concatena fra loro due string e
insert	che permette di inserire dei caratteri in una posizione qualunque di una string.

Una volta che un oggetto della classe *StringBuffer* è stato manipolato secondo le necessità lo si può trasformare in una stringa costante e immodificabile mediante un casting esplicito al tipo *String*.

Ad esempio si può avere:

```
StringBuffer sb = new StringBuffer ("Salve");
StringBuffer sb1 = new StringBuffer;

sb1 = sb.charAt (0);        // sb1 prende S
sb.setCharAt (4, 'i');      // sb diventa Salvi
sb.setlength(2);            // tronca sb a Sa
```

16 Con il termine char si indicano variabili o costanti di tipo carattere e con il termine string variabili o costanti di tipo String o StringBuffer in funzione del contesto.

4.5 Classi e costruttori

Un costruttore è una operazione che ha lo stesso nome della classe cui appartiene e per la quale non si dichiara il tipo di ritorno neanche come **void**. Il costruttore di una classe viene eseguito al momento della istanziazione di un oggetto della classe ovvero quando viene eseguito il comando **new** ma prima che l'handle dell'oggetto sia assegnato alla variabile che ne conterrà il puntatore. Tutti gli assegnamenti ai campi dati della classe sono eseguiti nel corpo del costruttore.

Una classe può avere più di un costruttore e questi si differenziano per la lista dei parametri con la quale sono richiamati.

Se un costruttore viene richiamato associandolo alla parola chiave **super** vuol dire che riferisce il costruttore della superclasse per cui al costruttore deve essere passata la lista dei parametri legale per il costruttore della superclasse. Tale chiamata deve essere il primo comando che compare nel corpo del costruttore. Nel caso che la classe derivata non usi la parola chiave **super** nel costruttore la classe base viene costruita con il costruttore di default. Se tale costruttore non è stato definito si ha un errore a run time. Se un costruttore invece contiene la parola chiave **this** allora vuol dire che richiama un'altra versione del costruttore della stessa classe. La parola chiave **this** deve essere il primo comando nel corpo del costruttore da cui si deduce che **super** e **this** sono mutuamente esclusive.

L'esecuzione di un costruttore si traduce in generale nei passi seguenti:

- vengono inizializzati i campi dati con i valori opportuni ovvero), false e null;
- se il costruttore contiene la parola chiave **super** o la parola chiave **this** viene eseguita la chiamata all'altro costruttore;
- i campi dati per i quali il costruttore ha i valori sono inizializzati e
- viene eseguito il corpo del costruttore che, essendo a tutti gli effetti un metodo, è caratterizzato da un insieme di istruzioni che ne costituisce il corpo.

Un costruttore è pertanto un metodo che inizializza automaticamente un oggetto al momento in cui questi viene creato ed è caratterizzato da un nome uguale a quello della classe. La chiamata del costruttore avviene subito dopo che l'oggetto è stato creato e prima che l'operatore **new** assegni il puntatore all'oggetto ad una variabile. Un costruttore non ha un tipo di ritorno, neanche come **void** dato che restituisce in modo implicito un oggetto dello stesso tipo della classe di cui è un costruttore. Un costruttore deve, comunque, definire un oggetto che si trovi in uno stato significativo e sia utilizzabile dal codice nel quale viene definito. Nell'esempio fatto posso avere:

```
class Coordinate {

   int    x, y;      // coordinate cartesiane
   float  r, ϑ;      // coordinate polari

   Coordinate (int x, int y) {
      this.x = x;
      this.y = y;
   }
   Coordinate (float r, float ϑ) {
      this.r = r;
      this.ϑ = ϑ;
   }

   <altri metodi>

}
```

per cui se ho

```
Coordinate p = new Coordinate (2, 3);
```

si ha che la **new** richiama il costruttore con una signature di interi, crea l'oggetto e assegna il relativo riferimento alla variabile **p**. L'uso di **this** fa si che i parametri formali possano

avere lo stesso nome delle variabili locali ovvero **this.**x fa riferimento alla variabile **x** dichiarata come int e non al parametro formale **x**.

La presenza di più costruttori (e in generale di più versioni di uno stesso metodo) si basa sul meccanismo di overloading e rappresenta il modo con cui Java implementa il polimorfismo in base al quale un metodo può assumere più forme fra le quali ne viene scelta una a run time in base alla signature. È ovvio che le varie signature devono essere distinte. L'utilizzo tipico è quello che prevede l'uso di due (o più) costruttori uno dei quali con una signature vuota mentre gli altri si aspettano un certo numero di parametri in modo che il primo possa essere usato per inizializzare un oggetto con dei valori di default e il secondo (o gli altri) con dei valori specifici.

Un'altra caratteristica del polimorfismo è quella dell'uso della parola chiave **this** all'interno di un metodo (e perciò anche di un costruttore) per riferire un'altra versione dello stesso metodo. Ad esempio:

```
class Coordinate {

   int    x, y;      // coordinate cartesiane
   float  r, ϑ;      // coordinate polari

   Coordinate (int x, int y) {
      this.x = a;
      this.y = b;
   }
   Coordinate () {
      this (0, 0);
   }
   Coordinate (float r, float ϑ) {
      this.r = a;
      this.ϑ = b;
   }

   float  distanza (int x, int y) {
          int dx = Math.abs  (this.x - x);
          int dy = Math.abs  (this.y - y);
          return distanza = Math.sqrt (dx * dx + dy * dy);
   }

   float  distanza (Coordinate p) {
          return distanza (p.x, p.y);
   }
   <altri metodi>
}
```

si hanno tre costruttori, il secondo dei quali richiama il costruttore con signature di interi passandogli i valori di default, ovvero le coordinate cartesiane del punto creato da una **new** con un costruttore privo di parametri espliciti:

```
Coordinate p0 = new Coordinate ();
```

La classe possiede anche un metodo per calcolare la distanza Euclidea di due punti fra di loro avendone le posizioni in coordinate cartesiane. Ad esempio è possibile una situazione quale la seguente:

```
Coordinate p0 = new Coordinate (2, 5);
Coordinate p1 = new Coordinate (3, 1);

float r = p0.distanza (p1);
float r = p0.distanza (9, 2);
```

dove nella prima forma si richiama il metodo **distanza** che accetta come parametro un elemento della classe *Coordinate* e che richiama lo stesso metodo ma con una signature di interi.

Ogni volta che si richiama il metodo **distanza** con signature di interi si ha che la parola chiave **this** permette di riferire le variabili p0.x e p0.y.

Più in dettaglio la **p0.distanza (9, 2)** fa si che **this.x** assume il valore **2** (valore assegnato a **p0.x** dal suo costruttore) mentre la variabile di tipo int x assume il valore 9. Considerazioni analoghe valgono per le variabili **this.y** e **y.**

Altri due metodi che si può pensare di assegnare alla classe sono i seguenti:

- un metodo per convertire da coordinate cartesiane in coordinate polari e
- un metodo per convertire da coordinate polari in coordinate cartesiane

ma il loro numero può crescere a piacere in funzione dell'uso al quale la classe è destinata.

4.6 Ereditarietà

Una classe discende sempre da un'altra e tale relazione è espressa implicitamente (e la classe discende dalla classe *Object*) oppure esplicitamente mediante la parola chiave **extends**.

Ad esempio se si ha:

```
class Triangoli extends Poligoni
{
   <altre operazioni e campi dati>
}
```

si definisce una classe derivata **Triangoli** che estende la classe base **Poligoni** e perciò definisce oggetti più specializzati rispetto a quelli della classe base.

Fra una classe derivata e la classe base esiste una relazione secondo la quale è possibile assegnare valori della classe derivata a variabili della classe base e, in questo caso, è possibile determinare a quale tipo di oggetto punti una variabile ricorrendo all'operatore **instanceof** per cui se ho

```
class A extends B
{
   <operazioni e campi dati per A>
{
```

posso avere

```
A a1 = new A (lista_parametri);
B b1 = a1;
```

ho che l'oggetto b1 punta a un oggetto della classe derivata, fatto che può essere individuato come segue

```
if (b1 instanceof A)
{
   A a2 = (A) b1
}
```

ovvero si può fare un casting esplicito ad un oggetto della classe base.

L'ereditarietà permette pertanto di definire una classe come derivata da un'altra in modo che la prima erediti tutti i metodi e le variabili dell'altra e possa usare tali metodi e variabili per implementare nuovi metodi. Ad esempio sulla base della classe *Coordinate* è possibile definire la classe seguente:

```
class   3D   extends   Coordinate {
        int   z;
        float ϕ;
        <costruttore>
        <altri metodi e variabili>
}
```

nella quale non è necessario dichiarare le variabili **x**, **y**, **r** e θ che sono ereditate dalla classe base mentre la classe 3D ha il suo costruttore che può richiamare quello della classe base utilizzando la dichiarazione **super**. Una classe derivata può infatti richiamare direttamente un metodo della propria classe base utilizzando la parola chiave **super** all'interno di uno dei suoi metodi, utilizzando come signature quella attesa dal metodo della classe base. Ad esempio la classe 3D può avere come costruttore il seguente metodo:

```
3D (int x, int y, int z) {
    super (x, y);
    this.z = z;
}
```

il quale richiama il costruttore della classe base passandogli i parametri da questo attesi e poi inizializza la variabile z che è tipica della classe derivata.

Una classe derivata può inoltre ridefinire uno dei metodi della classe base mediante una operazione detta di overriding. Più in dettaglio, se la classe base B ha un metodo **m1**, una sua classe derivata D può sia usare **m1** per implementare un metodo locale sia definire un metodo locale **m1** che, con un meccanismo di overriding, nasconde il metodo della classe base.

L'overriding può riguardare anche i costruttori e le variabili ma in questo caso rappresenta una violazione della filosofia di base dell'ereditarietà. È possibile, per evitare overriding accidentali, dichiarare le variabili come non sovrascrivibili (ovvero come variabili sulle quali non si può eseguire una operazione di overriding) usando il costrutto **final** anteposto al tipo, come nell'esempio che segue:

```
final int   SI   1;
final int   NO   0;
```

Analogamente un metodo dichiarato **final** in una classe non può essere sovrascritto all'interno di una sottoclasse e può essere collegato in modo statico all'interno del bytecode prodotto dal compilatore invece che in modo dinamico, dato che il compilatore sa che tale metodo non potrà essere sovrascritto, in modo da ottenere una efficienza maggiore.

Come già detto in precedenza, un oggetto non più riferito all'interno di un programma Java viene deallocato in modo automatico con un meccanismo di garbage collection (letteralmente "raccolta della spazzatura"). Se l'oggetto gestisce risorse di sistema come file o font è possibile, prima che questi sia distrutto, voler eseguire del codice che si occupi del rilascio di tali risorse. Tale codice viene incapsulato in un metodo dichiarato come **finalize**. Il garbage collector prima di distruggere un oggetto non più riferito e che possiede un metodo dichiarato come **finalize** esegue tale metodo in modo che sia possibile il rilascio delle risorse di sistema usate dall'oggetto e poi lo distrugge in modo da rilasciare la memoria da questi occupata.

4.7 Packages & interfacce

Un *package* è un contenitore di classi il cui uso permette di individuare una classe con una sorta di fully qualified name che permette di distinguere classi con nome uguale come appartenenti a package distinti.

I package sono memorizzati su base gerarchica che si riflette in una struttura a directory e sottodirectory sul sistema ospite e devono essere importati esplicitamente con un comando di **import** in modo che le classi in essi definite siano accessibili al resto del codice.

Il meccanismo dei package permette di strutturare lo spazio dei nomi del linguaggio in modo che classi con nome identico ma distinte non entrino in conflitto fra di loro in quanto appartengono a package distinti.

Una altro uso dei package è quello di consentire il controllo della visibilità delle classi. Per l'uso dei package si hanno vari comandi che verranno brevemente esaminati in quanto segue.

Il primo comando legale che può comparire in un file sorgente Java è il comando **package** che dice al compilatore quale package contiene le classi utilizzate nel file. I package sono memorizzati in una struttura di directory che corrisponde alla struttura gerarchica del package. La sintassi del comando **package** è la seguente:

package pkg1[.pkg2[.pkg3[. [.pkg-n]]]]]]

con il quale si definisce una gerarchia di package. Ad esempio le classi di utilità sono contenute nel package **java.utils** che, in ambiente Unix, è memorizzato nella directory **java/utils** mentre la posizione della radice della gerarchia (in pratica la directory **java**) è individuata dal contenuto della variabile di ambiente **Classpath**.

Altri esempi sono il package **java.awt.image** contenuto in **java/awt/image** e **java.applet** contenuto in **java/applet**.

Le classi in un file Java possono essere accedute o con il loro nome completo detto fully qualified name (in questo modo la classe **Vector** del package **java.utils** è acceduta come **java.utils.Vector**) oppure con il solo nome della classe avendo cura di importare il package che la contiene con il comando **import** la cui sintassi è la seguente:

import gerarchia_dei_package.*

oppure

import gerarchia_dei_package.nome_classe

Nel primo caso vengono importate tutte le classi contenute nel package individuato dalla gerarchia mentre nell'altro caso si ha l'importazione della sola classe specificata. Ad esempio se ho:

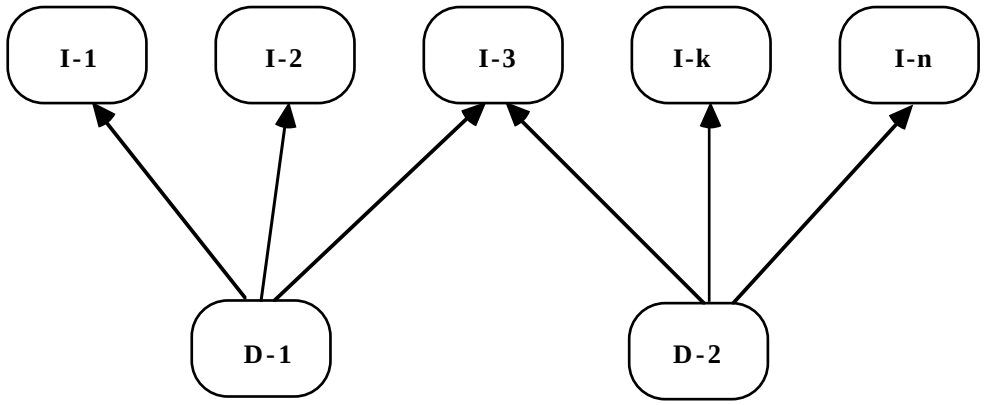
import java.utils.*

il compilatore importa tutte le classi del package mentre se ho

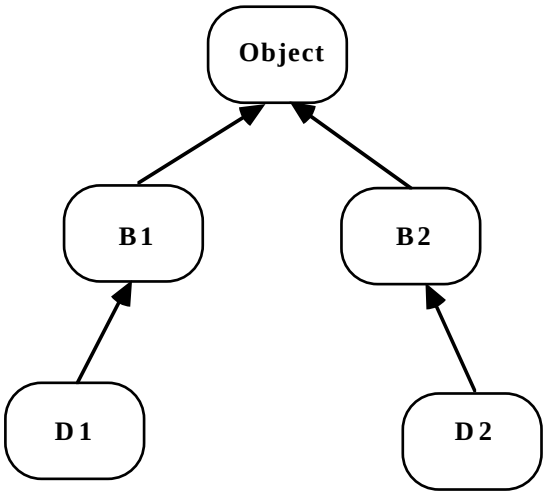
import java.utils.Date

il compilatore importa solo la classe specificata. Le classi predefinita del linguaggio sono contenute nel package **java** e le funzioni basilari sono contenute in **java.lang** (cui corrisponde la directory **java/lang**) che viene importato automaticamente dal compilatore.

Una interfaccia invece è un modo per definire una classe completamente astratta. L'uso principale delle interfacce in Java è quello di implementare l'ereditarietà multipla (vedi figura che segue).



In figura con I-1 e simili si indicano le interfacce mentre le due classi derivate D-1 e D-2 possono implementare più interfacce in modo tale da implementare l'ereditarietà multipla. Nell'esempio è come se D-1 ereditasse sia da I-1 sia da I-2 sia da I-3 e un discorso analogo vale per D-2.



Un'interfaccia è analoga ad una classe senza avere però né variabili né codice nel corpo dei metodi dichiarati al suo interno. Una classe può implementare un numero qualunque di interfacce dato che per realizzare una interfaccia basta che contenga i metodi che caratterizzano l'interfaccia stessa.

Tale caratteristica permette a classi che non hanno legami diretti in termini di gerarchia delle classi di implementare la stessa interfaccia (vedi figura a lato): le classi D1 e D2 non hanno nessuna relazione fra di loro ma ciò non toglie che possano implementare la stessa interfaccia.

Una interfaccia è definita con il comando seguente:

```
interface   nome_interfaccia    {

   public type    nome_metodo-1 (lista_tipata_dei_parametri);
   ....
   public type    nome_metodo-n (lista_tipata_dei_parametri);

   <variabili>
}
```

Le variabili definite nell'interfaccia sono implicitamente definite come **final** ossia sono costanti e non possono essere modificate in sede di implementazione dell'interfaccia sebbene possano essere inizializzate al momento della dichiarazione utilizzando valori costanti. Tutti i metodi di una interfaccia devono essere dichiarati come **public**.

Le interfacce definite con il comando **interface** sono implementate all'interno delle classi con il comando

implements nome_interfaccia

mentre i metodi dichiarati nell'interfaccia sono implementati all'interno della classe che ne contiene il codice ad essi relativo.

Ad esempio è possibile avere:

```
interface   i1 {

   public void    m1 ();
}
```

e poi definire

```
class c1 implements i1 {
   void m1 (lista_tipata) {
      <corpo di m1>
   }

   <altri metodi e variabili di c1>
}
```

Una classe può implementare più interfacce i cui nomi sono elencati come parametri del comando **implements** separati da una virgola. Se una classe implementa due o più interfacce caratterizzate da uno stesso metodo allora tale metodo sarà accessibile a tutti gli oggetti che accedono alla classe tramite una qualunque delle due interfacce.

Una classe può pertanto estendere una classe base e implementare una o più interfacce con una sintassi quale la seguente:

class D **extends** B **implements** i1, i2 **{**
  <corpo di D>
}

È possibile usare il nome di una interfaccia come il tipo degli oggetti di una classe che implementa l'interfaccia mentre a run-time alla variabile viene associato il metodo corretto che implementa l'interfaccia (il costruttore).

Un altro uso delle interfacce è quello di usarle per definire un insieme di costanti (sotto forma di coppie **nome valore**) che debbano essere importate in più classi ma senza che queste possano modificarne il valore.

In questo caso l'interfaccia non dichiara nessun metodo per cui le classi che la implementano non devono in realtà implementare nulla ma si limitano a importare un insieme di variabili costanti come se fossero delle variabili locali dichiarate come **final** e inizializzate con dei valori che risultano essere condivisi fra più classi.

4.8 I/O e persistenza

Ingresso/Uscita

Il linguaggio Java possiede il package **java.io** che contiene tutte le classi che consentono di eseguire operazioni di I/O all'interno di programmi Java.

Per l'esecuzione di tali operazioni Java definisce il concetto di stream visto come una sorgente o una destinazione (o pozzo) per i dati visti come byte. Gli stream di default sono sequenze di caratteri e gli stream più comuni sono i file sebbene altre sorgenti o destinazioni possibili per i byte possano essere:

- i caratteri in una stringa;
- i caratteri in un blocco di memoria o
- i byte in un socket di rete.

L'uso del concetto di stream permette di astrarre dalle caratteristiche di tali sorgenti/destinazioni in modo che all'interno di un programma le si possa vedere come file. Il linguaggio usa un meccanismo suo proprio per la gestione separata di:

- formattazione,
- bufferizzazione,
- lettura di byte e
- scrittura di byte.

Il linguaggio ha una classe astratta detta **OutputStream** caratterizzata da un insieme ristretto di operazioni, quali scrittura di un byte o di un array di byte e chiusura di uno stream, e una classe analoga detta **InputStream** caratterizzata da analoghe operazioni di lettura.

Dalla classe astratta **OutputStream** sono ottenute per derivazione le classi concrete **FileOutputStream** e **ByteArrayOutputStream** le cui operazioni di scrittura sono definite in modo da inviare dei byte o ad un file o ad un blocco di memoria. In modo analogo dalla classe **InputStream** derivano le classi **FileInputStream** e **ByteArrayInputStream** le cui operazioni di lettura sono definite in modo da prelevare dei byte o da un file o da un blocco di memoria.

La classe **OutputStream** ha in più come classe derivata la classe **PrintStream** che può essere vista come un adattatore che si occupa di formattare numeri, stringhe ed altri oggetti che sono tradotti in sequenze di byte che sono inviate allo stream che è stato passato come parametro al costruttore. Ad esempio è possibile avere:

FileOutputStream fout = **new FileOutputStream (**"esempio.txt"**);**
PrintStream pout = **new PrintStream (**fout**);**

in cui si può avere anche una bufferizzazione insieme alla formattazione utilizzando il meccanismo dei filtri illustrato dall'esempio che segue:

PrintStream pout = **new PrintStream (new BufferedOutputStream (new FileOutputStream (**"esempio.txt"**)));**

in cui **PrintStream** esegue la formattazione, **BufferedOutputStream** esegue la bufferizzazione e **FileOutputStream** è la classe che effettivamente emette in uscita i byte dei dati.

Per quanto riguarda l'I/O di testo e di tipo binario Java supporta output di testo mentre non supporta l'input di testo e supporta in modo flessibile l'input e l'output di tipo binario.

L'output di testo formattato è possibile utilizzando la classe **PrintStream** e i metodi **print** o **println**.

Nel caso si stampi un oggetto, ovvero nel caso si applichi la **print** o la **println** ad un oggetto, gli viene prima applicato il metodo **toString** in modo da trasformarlo in una stringa che viene successivamente emessa in output.

La classe di base *Object* definisce il metodo **toString** in modo che produca una stringa che contiene il nome dell'oggetto e l'indirizzo per cui è bene che le classi derivate sovrascrivano tale operazione in modo da definire l'operazione **toString** che dato un oggetto ne listi tutti i campi dati.

La **PrintStream** non ha un'analoga classe per l'input formattato in modo che se si deve leggere un numero è necessario
- leggere le sue cifre,
- metterle come caratteri in una stringa e
- convertire la stringa in un numero.

In Java è possibile eseguire operazioni di input e di output di tipo binario con le classi **DataInputStream** e **DataOutputStream** che sono caratterizzate da metodi quali **readType** e **writeType** in cui **Type** può assumere una delle forme Int, Short, Long, Float, Double, Boolean, Char o Chars in base ad una sintassi quale la seguente:

object_name.method_name(valori);

ovvero, ad esempio:

outfile.writeChars("pippo");

Tali operazioni, dato che le dimensioni e i formati dei tipi di base di Java sono fissi, sono indipendenti dal processore su cui viene fatto girare un programma Java e lo stesso vale per l'ordinamento dei byte che è sempre di tipo big-endian. Ciò consente a due processi Java di comunicare senza problemi attraverso stream di input e di output, indipendentemente dal processore su cui sono in esecuzione.

Un pò di cautela deve essere usata con le stringhe dato che per scrivere una stringa se ne scrive per prima cosa la lunghezza e poi si scrivono i caratteri mentre in lettura si legge prima la lunghezza della stringa e poi i singoli caratteri in un oggetto di tipo **StringBuffer** di cui, a fine lettura, si fa un casting esplicito al tipo **String**. Ad esempio il scrittura posso avere:

```
String s = "Stringa di prova";
DataOutputStream out = new DataOutputStream (new FileOutputStream ("file.dat"));
out.writeInt (s.length());
out.writeChars(s);
```

mente in lettura ho

```
DataInputStream in = new DataInputStream (new FileInputStream ("file.dat"));
int len = in.readInt();
StringBuffer b = new StringBuffer (len);
for (int i = 0; i < len; i++) b.append(in.readChar());
String s = b.toString();
```

e ciò perché il linguaggio non fornisce i metodi **readString** e **writeString**.

È possibile usare il metodo **readLine** (che restituisce una linea di testo completa come un elemento di tipo **String**) della classe **DataInputStream** in modo da leggere linee di testo una alla volta.

L'accesso in lettura e scrittura a file per leggere e scrivere caratteri è fornito in Java dalle classi **FileInputStream** e **FileOutputStream** che richiedono il nome del file da aprire in lettura o scrittura come parametro del costruttore mentre in assenza di distruttori il file deve essere chiuso esplicitamente quando non deve essere più acceduto.

L'accesso in lettura o scrittura va fatto mediante la definizione di un adattatore come viene illustrato dall'esempio che segue:

FileInputStream filein **= new FileInputStream (**"file.dat"**);**
DataInputStream in **= new DataInputStream (**filein**);**

a questo punto posso accedere a **in** con uno dei metodi suddetti, ad esempio

in.readInt();

e al momento in cui il file non mi serve più eseguo l'istruzione

filein.close ();

L'accesso in lettura/scrittura ad un file lo si ottiene attraverso la classe **RandomAccessFile** che non deriva né da **InputStream** né da **OutputStream** dato che Java non prevede ereditarietà multipla ma che implementa le interfacce **DataInput** e **DataOutput** che dichiarano le operazioni **read/write** delle classi **DataInputStream** e **DataOutputStream**.

Un file di tipo **RandomAccessFile** può essere aperto in lettura

RandomAccessFile in **= new RandomAccessFile (**"nome"**,** "r"**);**

oppure in lettura e scrittura

RandomAccessFile inout **= new RandomAccessFile (**"nome"**,** "rw"**);**

e supporta l'accesso casuale mediante l'operazione

inout.seek(offset);

in cui **offset** è una misura della distanza dall'inizio del file.

Spesso è utile poter elaborare caratteri che arrivano sia da un file sia da altre sorgenti quali ad esempio i caratteri contenuti in una stringa. Java permette di dichiarare degli stream speciali che si interfacciano con le stringhe che però non leggono e scrivono sequenze di caratteri ma piuttosto sequenze di byte per cui si parla in modo più appropriato di stream di tipo array di byte. Per scrivere caratteri si fa uso pertanto di **ByteArrayOutputStream** mentre i dati possono essere acceduti in due modi:

- come sequenze di byte in un array Byte [] con l'operazione **toByteArray** oppure
- come un elemento di tipo **String** mediante l'operazione **toString**.

Java non ha supporto alcuno per un output formattato di numeri per cui il programmatore deve definirsi le classi che gli servono in funzione dei propri scopi. Ad esempio se si vogliono delle formattazioni particolari dei numeri si deve definire una nuova implementazione della **PrintStream** che esegua le formattazioni desiderate.

Un'altra possibilità è quella di definire un adattatore ovvero una nuova classe derivata da una classe esistente di cui usa il costruttore e contenente un insieme di metodi per l'esecuzione delle operazioni di output formattato necessarie. In modo analogo si può definire un adattatore per la lettura di input formattati.

Nel caso di errori relativi agli stream Java interrompe la computazione e solleva una eccezione di tipo **IOException** che di solito non viene gestita dalla funzione che esegue le operazioni di I/O ma viene propagata al gestore opportuno detto **IOException** handler che viene associato alle singole operazioni di I/O con la clausola **throws IOException** e che si occupa sia di segnalare l'eccezione sia di gestirla in modo esplicito (vedi il paragrafo sulle eccezioni).

Persistenza
Un oggetto è detto essere persistente se viene memorizzato in un file o in un'altra struttura stabile. La persistenza si riferisce alla memorizzazione degli oggetti in quanto tali e non dei loro campi dati singolarmente e delle loro funzioni sequenzialmente su file su disco.

Un programma può:
- creare degli oggetti istanze di più classi;
- salvare tali oggetti come oggetti persistenti e
- recuperarli in un secondo tempo per ulteriori elaborazioni oppure passarli ad altri programmi e/o altri host.

Java consente di creare oggetti persistenti utilizzando il metodo **writeObject** della classe **ObjectOutputStream** mentre permette l'accesso ad oggetti con il metodo **readObject** della classe **ObjectInputStream**. Il valore di ritorno della **readObject** è un oggetto della classe **Object** per cui se ne deve operare il casting esplicito al tipo desiderato.

Java conosce automaticamente come leggere e cole scrivere tutti gli oggetti definibili nel linguaggio: tutti i campi sono salvati e ripristinati nei momenti opportuni.

In scrittura gli oggetti sono salvati in un flusso sequenziale e per ogni oggetto nuovo vengono memorizzati il numero di serie, il tipo e il contenuto mentre se l'oggetto viene riferito di nuovo se ne memorizza solo il numero di serie.

È tuttavia possibile evitare che il contenuto di certi campi dati sia salvato dato che tali campi contengono informazioni legate ad una particolare istanza del programma ma non significativi per nessun altra istanza successiva e lo si può fare dichiarando tali campi come **transient** in modo che il loro contenuto non sia né salvato né recuperato.

La persistenza è quindi un metodo per fare in modo che gli oggetti continuino ad esistere e conservino le loro informazioni mentre il programma che li ha creati e abitualmente li gestisce non è in esecuzione. L'implementazione della persistenza consente di affrontare il problema del salvataggio e ripristino degli oggetti in un modo rispettoso della filosofia OO. Nel caso di persistenza tout court un oggetto è dichiarato persistente e la sua gestione è demandata al supporto run time del linguaggio altrimenti se la memorizzazione e il recupero sono possibili solo tramite chiamate esplicite a metodi del linguaggio si parla di persistenza leggera. In Java 1.1 è disponibile solo la persistenza leggera.

4.9 Categorie Static *e* Abstract

Java possiede altre due categorie con cui dichiarare i metodi e le classi e che consentono la dichiarazione di metodi simili a funzioni globali oppure di classi e metodi come scatole vuote la cui implementazione è demandata alle sottoclassi. Tali categorie sono individuate rispettivamente dalle parole chiave **static** e **abstract**.

Un metodo dichiarato **static** è un metodo utilizzabile al di fuori di tutti gli oggetti. Un esempio di metodo dichiarato **static** è il metodo **main** che rappresenta l'entry point usato dall'interprete per eseguire un programma Java. Un metodo dichiarato **static** può richiamare solo metodi **static** e non può fare riferimenti a **this** o **super** in quanto per tale metodo tali parole chiave sono prive di significato.

Tali metodi possono contenere solo variabili di tipo **static** che sono a tutti gli effetti variabili globali e perciò accessibili da qualunque parte del codice. Infine un metodo dichiarato come static si comporta come una funzione globale che può essere richiamata da qualunque punto del codice riferendola come:

nome_classe.nome_metodo (lista_parametri)

La parola chiave **abstract** può essere usata per dichiarare una classe in modo da definire solo la sua struttura senza dover implementare tutti i metodi della classe. I metodi non implementati devono essere dichiarati come **abstract** a loro volta e, per essere utilizzabili, devono essere implementati in una sottoclasse.

Ad esempio, se ho:

```
abstract class A {
        abstract void m1();
        <altri metodi>
}
```

perché il metodo **m1** sia utilizzabile è necessario definire una sottoclasse derivata B che implementi il metodo m1 ovvero è necessario avere:

```
class B extends A {

    void m1 () {
       <corpo di m1>
    }
    <altri metodi>
}
```

in modo che la classe A rimanga non specificata con alcuni dei suoi metodi e che l'implementazione venga fatta solo nelle sottoclassi che devono utilizzare i metodi suddetti.

4.10 La visibilità

Il linguaggio Java consente di definire vari livelli di visibilità di variabili e metodi mediante l'uso di parole chiave al momento della loro definizione.

La visibilità è condizionata dal fatto che un package è un contenitore di classi ed altri package e che una classe è un contenitore per campi dati (o variabili) e metodi. La classe è infine la più piccola unità di astrazione del linguaggio per cui metodi e variabili sono visibili all'interno di una classe in cui sono definiti indipendentemente dal loro grado di visibilità.

Le parole chiave che è possibile utilizzare sono: **public**, **private** e **protected** che può comparire da sola o combinata con **private** nella forma **private protected**. In più si può avere l'assenza di specificazioni che equivale alla situazione di default di visibilità all'interno dello stesso package (vedi qui di seguito).

Dato che la classe è la più piccola unità di astrazione del linguaggio le parole chiave **public**, **private** e **protected** non alterano la visibilità all'interno di una classe.

In generale un elemento (che può essere sia un metodo sia una variabile) dichiarato come **private** è visibile solo all'interno della classe in cui è definito e non è accessibile al di fuori di questa.

Se un elemento, viceversa, è dichiarato come **public** è visibile da tutte le classi di tutti i package anche se queste non sono legate in termini di ereditarietà.

La parola chiave **protected** conferisce all'elemento una visibilità limitata all'interno dello stesso package ma solo a classi che sono legate da un rapporto di ereditarietà. La categoria **protected** agisce come la classe **private** se non che una classe derivata ha accesso agli elementi dichiarati **protected** nella classe base.

La mancanza di una specificazione del tipo **public**, **private** o **protected** è la situazione di default cui pertanto è associata una visibilità di tipo package sia alle sottoclassi sia alle altre classi contenute nello stesso package.

L'uso della combinazione **private protected**, infine, garantisce l'accesso all'elemento da parte di tutte le sottoclassi della classe corrente anche se sono contenute in package distinti.

Si noti che la definizione di un metodo public non definisce una funzione globale dato che il metodo si applica solo agli oggetti di una ben precisa classe sebbene sia accessibile da qualunque punto del codice. È ovvio che un elemento è sempre visibile all'interno della classe in cui viene definito.

Alla luce di quanto detto si possono rivedere gli esempi fatti individuando le visibilità relative degli elementi dichiarati, siano essi metodi, variabili oppure classi.

5 Il modello a oggetti in Java

Da un punto di vista tecnico l'OO si può pertanto ricondurre ai punti seguenti:

- definizione di tipo astratti;
- definizione di relazioni di ereditarietà fra i tipi e
- polimorfismo delle funzioni (o metodi)

ma anche altri elementi contribuiscono a rendere l'OOP un paradigma valido e a fare di java un linguaggio potente, semplice e robusto. Tali elementi sono:

- la modalità di creazione e di distruzione degli oggetti
e
- l'uso di collezioni ed iteratori.

Per quanto riguarda il primo punto Java prevede che gli oggetti siano creati nello heap (o memoria dinamica) a run time e vengano distrutti automaticamente al momento in cui non sono più referenziati. Il fatto di creare gli oggetti nello heap consente al programmatore di allocare a run time un numero a priori illimitato di oggetti.

Con il termine **collezione** si definisce una classe i cui oggetti contengono puntatori ad altri oggetti. Esempi di **collezioni** (in pratica elementi che crescono dinamicamente a run time) sono i seguenti:
- vettori (accesso casuale agli elementi singoli);
- liste concatenate (inserimento casuale);
- gli insiemi e
- elementi "complessi" quali code, pile, alberi e tabelle hash.

Ogni collezione ha metodi utilizzabili per inserire ed estrarre gli elementi. I metodi per l'inserimento si chiamano di solito **push** o **add** mentre quelli per l'estrazione possono sia far ricorso a schemi di indirizzamento per recuperare i singoli elementi (è il caso dei vettori e dell liste) sia, in casi complessi oppure nei casi in cui si vogliono recuperare o confrontare più elementi, far uso dei cosiddetti **iteratori**.

Un iteratore è un oggetto che sceglie, sulla base di certi criteri, alcuni elementi di una collezione e li passa all'oggetto client dell'iteratore stesso. L'**iteratore** presenta una collezione di oggetti come se fosse una sequenza astraendo, pertanto, sia dalla struttura interna sia dalla implementazione. L'uso di più collezioni discende sia dal fatto che ciascuna ha una propria interfaccia e un proprio comportamento sia da fatto che ciascuna esegue i propri metodi con efficienze diverse. Ad esempio l'accesso ad un elemento in un vettore costa O(1) mentre in una lista concatenata costa, in genere, O(n).

5.1 la gerarchia single-rooted

In Java tutte le classi discendono da una classe base primitiva detta *Object* in modo che la gerarchia delle classi abbia una sola radice e, in assenza di ereditarietà multipla, sia rappresentabile come un albero.

Grazie a questo schema il progettista può definire classi che o discendono direttamente dalla classe base o da una classe da essa derivata avendo sempre la certezza che queste appartengono sempre alla stessa gerarchia.

L'utilizzatore, dal canto suo, sa che tutti gli oggetti hanno una interfaccia base in comune per cui alla fin fine sono tutti dello stesso tipo. In più è possibile garantire che tutti gli oggetti abbiano un certo numero di funzionalità di base quali il fatto di essere allocati nello heap e la caratteristica di accettare i parametri passati in un certo modo. Infine l'unicità della gerarchia fa si che sia più semplice implementare il meccanismo di garbage collector e che sia sempre possibile conoscere il tipo a cui appartiene un oggetto.

5.2 Oggetti e puntatori

In Java gli argomenti di un metodo non possono essere passati mai per riferimento per cui un metodo non può mai cambiare il valore a cui si riferisce uno dei suoi argomenti a meno che questi non sia un oggetto nel qual caso il metodo può modificarne lo stato. Come visto ogni operazione ha in Java due argomenti impliciti che sono **this** e **super**. In particolare **this** è un riferimento ad un oggetto proprio come ogni altra variabile associata ad un oggetto: una funzione (o metodo) può restituire il riferimento ad un oggetto restituendo this e può anche restituire il riferimento ad un nuovo oggetto allocato nello heap. Tali riferimenti sono automaticamente distrutti dal garbage collector non appena non sono più necessari.

In molti casi può essere necessario copiare oggetti e riferimenti. Per fare ciò non è possibile ricorrere alla variabili che in Java sono utilizzate non per memorizzare oggetti ma puntatori ad oggetti ovvero sono utilizzate per accedere agli oggetti. Se ciò facilita la costruzione di array di variabili dato che queste hanno, per un dato tipo, la stessa dimensione ciò crea problemi al momento della copia dato che la copia di una variabile crea solo un altro riferimento allo stesso oggetto piuttosto che creare un nuovo oggetto. Se si vuole forzare la duplicazione di u oggetto si deve usare una operazione di **clone**.

Il modello ad oggetti in Java che vedi gli oggetti come riferiti da variabili è molto adatto ad esprimere il polimorfismo attraverso un binding dinamico di oggetti alle variabili.

Le variabili oggetto[8] si comportano come puntatori nel senso che il linguaggio non ha una sintassi speciale per il riferimento o l'accesso ai membri di una classe e in più le variabili non devono essere inizializzate al momento della loro definizione e durante la loro vita possono puntare a più oggetti distinti o anche a nessun oggetto. La creazione di un oggetto avviene mediante una istruzione del tipo

class varname **= new class ();**

mentre la dichiarazione

class varname

si limita a dichiarare o un puntatore non inizializzato (se **varname** è una variabile locale) oppure un puntatore *null* se **varname** è il campo dati di una classe. In più dato che Java non fa distinzione fra puntatori a oggetti e riferimenti per cui è bene sempre documentare lo scopo per cui viene introdotta ogni variabile.

5.3 Le eccezioni

Una eccezione è rappresentata da un oggetto che viene creato al momento che un errore run time si verifica in seguito al verificarsi di qualche situazione anomala. Si dice che l'eccezione viene "sollevata" nel punto in cui si verifica l'errore e può essere "catturata" da un opportuno gestore delle eccezioni (detto "exception handler") che è stato scritto dal progettista per trattare quel particolare tipo di errore. Una eccezione non può essere ignorata e in assenza di un gestore in grado di catturarla si ha la terminazione anomala del programma al cui interno è stata sollevata l'eccezione.

Una eccezione è quindi una condizione anormale che si genera in una sequenza di codice durante l'esecuzione per cui si ha necessità di:

sollevarla in modo che venga richiamato un gestore dell'eccezione e
- gestirla tramite il codice del gestore.

Una eccezione è in Java un oggetto che descrive una condizione di errore che si è verificata in qualche punto del codice nel senso che al verificarsi di un errore viene creato un oggetto di classe **Exception** all'interno del metodo che ha dato luogo all'errore ed è su tale oggetto che si basa il meccanismo che verrà descritto nei paragrafi che seguono.

Concetti di base

Java è caratterizzato da un meccanismo sofisticato per trasferire il controllo a un gestore delle eccezioni dopo che ne è stata sollevata una in seguito ad un errore. Gli errori che si possono verificare nel corso della esecuzione di un programma rientrano in varie categorie fra cui:

- errori di input da parte dell'utente;
- errori a livello di dispositivi fisici quali porte seriali e stampanti;
- errori legati a limiti fisici quali spazio libero sui dischi rigidi e nella RAM e
- errori a livello delle funzioni che cooperano con altre per lo svolgimento di un certo compito.

in presenza di errori che anche solo potenzialmente impediscono la prosecuzione della computazione, il programma dovrebbe almeno:
- avvisare l'utente;
- salvate i dati relativi all'attività in corso e
- terminare l'esecuzione.

Tali compiti non possono in genere essere svolti dal codice che rileva la condizione di errore perché i compiti che tale codice dovrebbe essere in grado di svolgere esulano dalle sue capacità per cui è necessario, per avere una gestione effettiva degli errori, trasferire il controllo a del codice ad hoc detto gestore delle eccezioni o exception handler.

In presenza di errori sono state adottate nel tempo e nei diversi linguaggi svariate strategie, che ci limitiamo ad elencare, quali:

- restituire un codice di errore che deve essere testato in modo che l'esecuzione prosegua da una routine individuata da tale codice;
- ignorare l'errore ossia non eseguire nessuna azione speciale in presenza di un errore;
- settare una variabile errore di tipo globale, di solito di nome errno, in modo che in presenza di più errori in cascata solo l'ultimo sia registrato dalla variabile;
- emettere solo un messaggio di errore e poi abortire il programma in seguito ad un qualche acknowledgment dell'utente;
- abortire il programma in modo più o meno silenzioso;
  eseguire una operazione di jump ad un gestore degli errori;
- generare una signal ovvero un interrupt a livello software oppure
- sollevare una eccezione.

Java ha adottato l'ultima strategia che rappresenta un meccanismo sicuro e conveniente per sollevare eccezioni in presenza di errori e trasferire il controllo con le informazioni sull'errore ad una porzione di codice che ha il compito di gestire la situazione e il cui nome è gestore delle eccezioni o exception handler.

Come sollevare una eccezione

In Java l'oggetto eccezione che viene lanciato deve appartenere ad una classe derivata dalla classe **Throwable** sebbene non sia necessario dichiarare alcuna variabile per eseguire l'operazione.

Quando una funzione solleva una eccezione viene inviata la ricerca di un gestore dell'eccezione e la funzione non termina in modo normale. Se si ha un oggetto di tipo **Stack** e si esegue una operazione di **pop** sulla pila vuota si ha una eccezione di underflow che causa l'esecuzione della istruzione presente nel corpo della **pop**

throw StackError ("Stack underflow"**);**

per cui l'esecuzione della funzione **pop** viene abortita e viene mandato in esecuzione un gestore opportuno.

Come gestire le eccezioni

In Java i gestori delle eccezioni sono specificati con blocchi etichettati con la parola chiave **try** ad esempio:

```
try
{
   <codice>
}
catch (StackError e);
{
   <codice dell'handler>
}
```

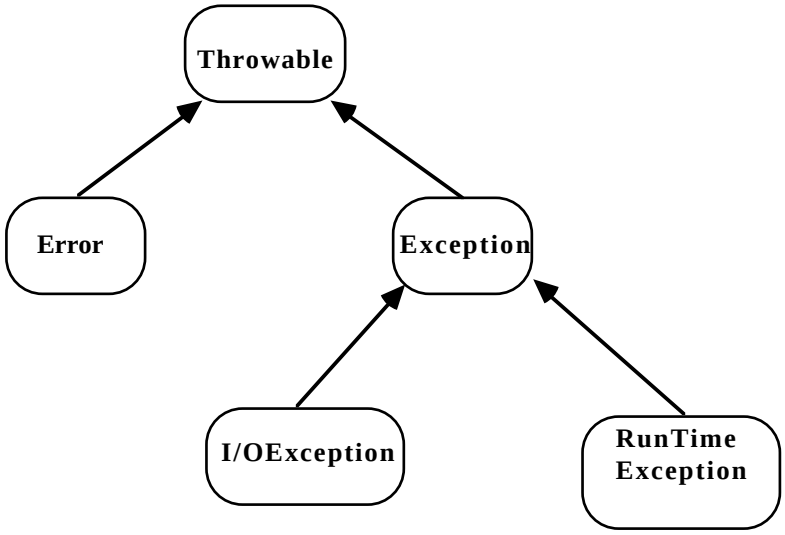
per cui posso avere

```
try
{
   read(inputfile);    // la read esegue un parsing che usa uno stack
}
catch (StackError s);
{
   message("Errore in accesso al file");
   inputfile.close();
}
```

Un blocco **try** può gestire più eccezioni ciascuna con il proprio gestore individuato da una opportuna istruzione di **catch**.

Gerarchie di eccezioni

Dato che i gestori delle eccezioni specificano il tipo delle eccezioni che gestiscono è possibile strutturare le classi delle eccezioni con una gerarchia di ereditarietà quale quella illustrata nella figura che segue nella quale un arco orientato stabilisce una relazione di ereditarietà fra due classi.



Le eccezioni appartengono a classi che derivano direttamente o indirettamente dalla classe **Throwable** che ha due discendenti dirette.

La sottoclasse **Error** rappresenta la root di una gerarchia che descrive condizioni fatali di errore che derivano da errori interni alla Java Virtual Machine (ovvero all'ambiente al cui interno sono eseguiti di regola i programmi Java) o da meccanismi di caricamento delle classi. tali errori non sono gestibili in alcun modo e pertanto non presentano interesse per il programmatore.

La sottoclasse **Exception** rappresenta a sua volta la root di una gerarchia di eccezioni che sono sotto il controllo del programmatore. La sottoclasse **RunTimeException** descrive errori logici che derivano da errori del programmatore e permette la definizione delle sottoclassi rappresentative delle eccezioni tipiche di un programma. Nel caso degli esempi fatti si ha:

class StackError extends RunTimeExceptions

il cui costruttore invoca quello della superclasse con una chiamata a **super**.

Le eccezioni della sottoclasse **IOExceptions** sono dette esplicite perché le eccezioni sollevate da tale classe (e altre simili che non sono esaminate in queste note per semplicità) devono essere pubblicizzate mentre le eccezioni del tipo **RunTimeException** sono dette implicite e non devono essere rese pubbliche ma, dato che derivano da errori di logica del programma, devono essere risolte in modo da riportare lo stato della computazione in uno stato consistente.

Può capitare a volte di voler catturare una eccezione per eseguire alcune azioni di ripristino locali prima di risollevare la stessa eccezione in modo che venga gestita da un gestore di tipo generale. In Java ciò è possibile se si cattura la classe base **Throwable** e dopo aver eseguito le operazioni del caso si esegue di nuovo una **s** sullo stesso oggetto per invocare il gestore.

Ad esempio si esegue una istruzione del tipo

catch (Throwable t);

e nel corpo del codice associato alla catch si esegue una **throw** senza nessun argomento in modo da sollevare di nuovo la stessa eccezione che viene pertanto istradata ad un gestore in uno scope più esterno.

Il fatto che una eccezione non venga catturata da nessun gestore di nessun livello si traduce in genere in una terminazione anomala della computazione.

L'uso delle eccezioni introduce un certo grado di incertezza in operazioni altrimenti banali quali la chiamata di funzione nel senso che una funzione può sollevare una eccezione e non terminare mai in modo normale. Se la funzione è programmata da altri può darsi che non sia noto il tipo dell'eccezione che viene sollevata per cui nessun gestore si preoccuperà di gestirla con conseguente terminazione anomala del programma per cui sarebbe bene sapere sempre se una funzione può sollevare una qualche eccezione e di che tipo essa sia.

In Java una funzione deve dichiarare tutte le eccezioni di tipo esplicito che solleva, dove per eccezione esplicita si intende una eccezione derivata dalla classe **Exception** con eccezione di quelle derivate dalla classe **RunTimeException** che dipendono da errori a run time e non logici. Le più comuni eccezioni di tipo esplicito sono le eccezioni di I/O.

Le funzioni che leggono o scrivono oggetti in caso di errori nell'accesso agli stream sollevano una eccezione e lasciano che sia un opportuno gestore a catturare l'errore e prendere gli opportuni provvedimenti.

In Java le funzioni devono rendere pubblica la possibilità di una eccezione in modo che qualcuno possa avere gli strumenti per scrivere un gestore per tale classe di eccezioni.

Java prevede l'uso dello specificatore **throw** che deve essere usato per tutte le eccezioni di tipo esplicito per dichiarare la possibilità che una funzione sollevi delle eccezioni esplicite. Ad esempio posso avere:

public void read (DataInputStream in**) throws IOException**

e nel corpo della **read** posso avere una o più istruzioni di accesso in lettura a stream che possono sollevare eccezioni di tipo **IOException** per cui la **throw** dichiara la classe delle potenziali eccezioni relativa alle funzioni primitive contenute al suo interno (dove con funzione primitiva si definisce un metodo proprio di una delle classi di I/O contenute in **iava.io**).

Se ho ad esempio

int i **=** in**.readInt();**
string s **=** in**.read.chars();**

ho che la **throw** esterna sulla **read** "copre" le eccezioni potenzialmente sollevate dalle **readInt()** e **read.chars()** sullo stream **in** di tipo **DataInputStream**.

Se una funzione invece non contiene la clausola **throw** allora è garantito che non solleverà eccezioni di tipo esplicito. Non è viceversa necessario dichiarare che una funzione può sollevare una eccezione implicita ovvero una eccezione della classe **Error** o **RunTimeException**. Se però si riesce a prevedere che una funzione può sollevare una eccezione di tipo **RunTimeException** è possibile prevedere dei controlli nel corpo della funzione che permettano di evitare che ciò accada.

In generale si dovrebbe cercare di scrivere del codice che non solleva eccezioni a run time piuttosto che cercare di prevedere quali eccezioni a run time una certa porzione di codice possa sollevare.

Come criteri di progetto si possono adottare i seguenti:

- le eccezioni dovrebbero essere eccezionali ovvero dovrebbero essere riservate per situazioni inattese nel normale flusso della computazione e che non possano essere risolte nello scope corrente;
- il numero delle eccezioni che il codice può sollevare deve essere minimizzato e i tipi delle eccezioni che si possono verificare devono essere ben documentati;
- è bene catturare le eccezioni che si è in grado di gestire ovvero se non si sa come gestire una condizione di errore in una certa posizione del codice è bene non catturarla dato che molto probabilmente in uno degli scope più esterni c'è un gestore in grado di gestire le eccezioni di tale tipo;
- è bene in relazione al punto precedente, invece, catturare e risollevare una eccezione se è necessario eseguire localmente una porzione per scopi particolari di codice prima che venga richiamato il gestore della eccezione;
- è bene non catturare una eccezione che si sa già essere gestita da un gestore in uno scope più esterno e

- nel caso si debba scegliere fra sollevare una eccezione e proseguire la computazione è bene, se possibile, scegliere la seconda alternativa.

In molti casi è infatti preferibile che un programma prosegua la sua esecuzione con qualche inconsistenza piuttosto che una eccezione non catturata ne causi la terminazione.

5.4 Il Multithreading

Il multithreading (più thread) permette la gestione di più task contemporaneamente all'interno di uno stesso programma. Un thread è infatti una porzione di programma che può essere eseguita in modo indipendente da altre porzioni di programma in modo che l'intero programma possa reagire più velocemente a richieste di suoi client (fra cui l'utente). L'uso tipico è nell'implementazione di interfacce utente e il massimo del beneficio lo si ha se il programma può essere eseguito su una architettura multiprocessore, nel qual caso i thread possono essere distribuiti sui processori disponibili in un certo istante. L'implementazione di un ambiente multithreading richiede la disponibilità di politiche di gestione di risorse condivise che in Java sono implementate con operazioni di lock e di rilascio di tali risorse mentre i thread stessi sono implementati con oggetti.

5.5 Tipi a run time

In OOP è spesso necessario operare un test per la concordanza dei tipi ovvero spesso di deve scoprire se un puntatore del tipo della classe base punta ad un oggetto di una classe derivata di un certo tipo. Java mette a disposizione l'operazione **getClass** della classe *Object* che restituisce un oggetto di tipo **Class** che descrive la classe a cui l'oggetto appartiene. La classe **Class** è caratterizzata da un insieme di operazioni dette collettivamente una riflessione che forniscono informazioni sulla classe cui l'oggetto appartiene. Per avere il nome esatto della classe cui l'oggetto appartiene si deve usare l'operazione **getName** sull'oggetto. Tale operazione restituisce una stringa che contiene il nome della classe.

Nel caso di

```
Class c = s.getClass();
System.out.println(c.getName());
```

la prima istruzione assegna a **c** le informazioni sulla classe cui appartiene **s** mentre la seconda stampa il nome della classe cui appartiene **s.**

In alcuni casi può essere necessario eseguire una indagine a run time sul tipo di un oggetto, indagine che viene detta test di "è-un-esemplare-di". Nel caso della classe *Poligoni* e della classe derivata *Rettangoli* e di oggetti delle due classi si può dover controllare se un dato oggetto della classe *FormeGeometriche* supera il test "è-un-esemplare-di" *Rettangoli* ovvero è un oggetto di tale classe oppure appartiene ad una classe derivata. Il procedimento di conversione di un puntatore della classe base in un puntatore di una classe derivata, ammesso che il cast sia eseguibile, è spesso detto downcasting.

Se **s** è un oggetto di tipo *FormeGeometriche* e **r** è un oggetto del tipo *Rettangoli* in Java si usa il casting nella notazione classica:

```
Rettangoli    r  = (Rettangoli) s
```

mentre per verificare se s punti realmente ad un oggetto della classe *Rettangoli* oppure ad un oggetto di una classe derivata da questa si può usare l'operatore **instanceof** che come argomenti ha un riferimento ad un oggetto e il nome di un tipo e ritorna true se l'oggetto è una istanza di quel tipo o di un tipo derivato per cui si può avere:

```
if  (s instanceof Rettangoli)
    {
    Rettangoli    r  =  (Rettangoli s)
    <operazioni della classe Rettangoli applicabili a r>
    }
```

5.6 Copia profonda e copia superficiale

La copia dei puntatori è detta copia superficiale dato che il suo effetto è la duplicazione di un riferimento in modo da avere due riferimenti che puntano allo stesso oggetto. Nel caso in cui l'operazione di copia crei un nuovo oggetto identico all'originale si parla di copia profonda. La copia profonda la si può ottenere con una operazione di clonazione e si traduce nella creazione di un nuovo oggetto identico all'originale ma indipendente da questo.

In Java la copia dei riferimenti è sempre superficiale per cui è possibile eseguire copie profonde solo con una operazione di **clone** la quale restituisce un oggetto di tipo *Object* per cui se ne deve fare il casting esplicito al tipo desiderato. L'operazione di **clone** è una operazione fornita dalla classe *Object* che di default esegue la clonazione di un oggetto duplicandone tutti i campi. tale operazione non è di solito accessibile in quanto è dichiarata protetta e prima di eseguire verifica che l'oggetto che si vuole clonare appartenga ad una classe che implementa l'interfaccia **Cloneable** o no: in questo caso viene sollevata una eccezione a run time che deve essere catturata e gestita. Per questi motivi è spesso conveniente, soprattutto per classi semplici, inserire fra i metodi un metodo di clone che ci permetta di duplicare oggetti della classe senza ricorrere all'operazione fornita dal linguaggio.

Un'altra caratteristica di Java è legata al funzionamento dell'operatore di confronto (==) che, se applicato ad oggetti, lavora sui riferimenti e non opera un confronto sul contenuto degli oggetti per cui il confronto fra due cloni fatto con tale operatore dà sempre come risultato un valore *false*. Si parla, in analogia alla copia, di confronto superficiale. Volendo fare un confronto profondo si deve ricorrere a metodi ad hoc oppure predefiniti per le singole classi. Nel caso della classe *String*, ad esempio, il confronto fra le stringhe viene fatto come segue:

string1.equals(string2)

Il progettista deve prevedere per le classi da lui definite il metodo **equals** con una sintassi simile alla precedente (ovvero **obj1.equals(obj2)**) e la cui semantica prevede il confronto fra le strutture interne e i contenuti dei campi dati dei due oggetti. Tale metodo deve essere definito esplicitamente in modo da sovrascrivere il metodo omologo che altrimenti la classe erediterebbe dalla classe *Object* e che si limita a confrontare le localizzazioni degli oggetti e non i loro contenuti.

5.7 Costruttori virtuali, riflessione & binding

Supponiamo di avere una stinga che descrive un tipo (ad esempio la stringa "Coordinate") e si voglia costruire un oggetto di quel tipo (ossia si voglia eseguire una **new Coordinate**) in modo da poter operare su oggetti di tale tipo. Tale problema è detto spesso costruttore virtuale dato che richiede la costruzione a run time di tipi variabili sebbene si traduca nella effettiva costruzione di un oggetto di un certo tipo. Java supporta il concetto di costruzione virtuale con il meccanismo seguente.

La classe **Class** ha una funzione dichiarata static **Class.forName (s)** che cerca l'oggetto corrispondente alla stringa **s** per cui **Class.forName (**"Coordinate"**)** restituisce un oggetto della classe **Class** che descrive la classe **Coordinate**.

A questo punto è possibile invocare il metodo **newIstance** per ottenere un nuovo oggetto della classe. Ad esempio se ho:

String s = "Coordinate"**;**
Class.forName (s**).newInstance();**

ottengo un nuovo oggetto della classe **Coordinate** costruito con il costruttore di default (per cui ho una eccezione a run time se la classe Coordinate non ha un costruttore di default).

La classe **Class** inoltre permette ad un programma di informarsi sulle proprietà di nuove classi in modo che gli utenti di tale programma possano definire a run time nuove classi e far si che il programma prenda in considerazione tali nuove classi.

In Java 1.1 si hanno tre classi **Field**, **Method** e **Constructor** che descrivono i campi dati, i metodi e i costruttori delle classi in più la classe **Class** ha i metodi **getFields()**,

getMethods() e **getConstructors()** che restituiscono array dei campi di tipo **public**, dei metodi e dei costruttori supportati dalla classe.

In Java tutte le funzioni hanno un binding dinamico e solo le funzioni dichiarate come **final** sono caratterizzate da un binding statico.

Il binding è la modalità del legame fra una istruzione di chiamata di funzione e il codice che corrisponde al corpo della funzione. Se una funzione è dichiarata **final** ovvero ha un legame di tipo statico con il codice che la implementa allora ciò si traduce in una impossibilità di overriding del metodo che a sua volta si traduce in una maggiore efficienza.

La maggiore efficienza deriva dal fatto che essendo impossibile sovrascrivere la funzione il corpo di questa può essere sostituito dal compilatore ad ogni chiamata alla funzione in modo che non si abbiano chiamate di funzione, con tutto ciò che questo comporta, ma solo esecuzione di codice cosiddetto inlined.

Il binding dinamico è il metodo con cui i linguaggi OO, Java compreso, implementano il polimorfismo (vedi). Se si ha una funzione **draw();** questa può avere un codice diverso in funzione del tipo di oggetto che deve essere tracciato e l'associazione è fatta a run time per cui si parla di binding dinamico che ci permette di fare

shape.draw();

in cui **shape** a run time può assumere un qualunque valore di oggetto che abbia fra i suoi metodi il metodo **draw ()getFields()** per cui può essere che **shape** rappresenti un segmento oppure un cerchio o un rettangolo oppure anche una sfera.

6 Conclusioni

Il presente lavoro ha avuto come obbiettivo principale quello di presentare il paradigma OO ed analizzare come tale paradigma sia realizzato dal linguaggio di programmazione Java, del quale si è data una sommaria descrizione evidenziandone i tratti caratteristici. Molto resta ancora da esaminare sia del paradigma OO sia del linguaggio Java. In relazione al linguaggio, alcuni suoi elementi chiave che verranno esaminati in una seconda puntata, seguito del presente lavoro, comprendono sia le applet sia i beans: con le applet il discorso si sposta nel mondo delle reti di computer mentre i beans spalancano le porte del mondo della riusabilità del software.

Bibliografia

Cioni L., "Co-operative principles in application design", proceedings del 4th International Workshop on Software Engineering and Artificial Intelligence for High Energy Physics, Pisa 3-8 Aprile 1995, pagg: 89-94

Eckel B., *Thinking in Java*, Prentice Hall, 1998

Hortsmann C. S., *Practical Object-Oriented Development in C++ and Java*, Wiley Computer Publishing, 1997

Naughton P., *Il manuale Java*, Mc Graw Hill, 1996