CONCETTI INTRODUTTIVI

Prima di cominciare a scendere nel dettaglio, definendo quindi le caratteristiche di un linguaggio orientato a oggetti, soffermiamoci su alcuni concetti iniziali. Un **programma** è una sequenza di semplici operazioni, eseguite in successione. Progettare e implementare un programma prende il nome di **programmare**. Un programma è scritto in un qualsiasi linguaggio di programmazione. Questi ultimi si possono classificare in:

- **linguaggi macchina**: istruzioni codificate in sequenze numeri, dipendenti dall'elaboratore che si usa.
- **linguaggi assembly**: istruzioni macchina codificate attraverso codici mnemonici, sempre dipendenti dall'elaboratore che si usa.
- **linguaggi di alto livello**: istruzioni ad un livello concettuale molto elevato, indipendente dall'elaboratore che si usa.

Ovviamente, ci concentriamo sui linguaggi di alto livello. Essi consentono un maggior livello di astrazione, ossia permettono di descrivere l'idea che sta dietro l'operazione da compiere. Essi seguono alcune rigide regole, che ne favoriscono la conversione in linguaggio macchina. Quest'ultima procedura è detta **compilazione** e chi si occupa di essa è detto **compilatore**. I lunguaggi di alto livello possono essere procedurali, come il C, oppure **orientati ad oggetti**, come C++ e Java. I programmi procedurali hanno numerosi vantaggi, grazie alla loro esecuzione sequenziale e la loro aderenza all'architettura di Von Neumann. Tuttavia, però, col tempo hanno riscontrato dei limiti nello sviluppo e nel mantenimento di software più complessi. Così, siamo passati a linguaggi orientati a oggetti. Essi permettono al programmatore di creare vere e proprie entità aderenti alla realtà e non più soltanto a variabili. Alla base della **OOP**, acronimo di *Object Oriented Programming*, c'è:

- **incapsulamento dei dati**: c'è una certa trasparenza sui dettagli che definiscono gli oggetti.

- **ereditarietà**: gli oggetti possono essere definiti seguendo una gerarchia, ereditando caratteristiche comuni dal proprio padre.
- **astrazione**: il meccanismo con cui si specificano le caratteristiche peculiari di un oggetto che lo differenziano da altri.
- **polimorfismo**: c'è la possibilità di eseguire funzioni con lo stesso nome che sono state specializzate per una particolare classe.

La OOP paga tutti questi suoi vantaggi sull'efficienza, infatti non è particolarmente adatta per lo sviluppo di software di base come sistemi operativi o driver. Come già anticipato, i linguaggi OOP manipolano gli **oggetti**. Ogni oggetto può essere **istanza** di una **classe**. La classe determina il comportamento degli oggetti che le appartengono. Per programmare in Java è necessario scrivere le definizioni delle classi che modellano il problema, e poi usarle per creare oggetti. Gli oggetti vengono attivati alla ricezione di un **messaggio**. Gli oggetti possono scambiarsi messaggi. Le classi hanno un'**interfaccia pubblica**, che specifica cosa si può fare con i suoi oggetti, e un'**implementazione privata**, cioè chi usa gli oggetti non sa come sono stati definiti i metodi invocati su di essi. Ogni classe ha un **nome**, che deve iniziare con la lettera maiuscola. Gli oggetti non hanno un nome, ma sono identificati dal **riferimento**. Si possono avere più riferimenti allo stesso oggetto.

INTRODUZIONE A JAVA

Dopo aver avuto una visione di insieme dei linguaggi OO, entriamo, ora, nel merito di Java. Ovviamente, Java è un linguaggio OO che risponde all'esigenza di creare software facili da modificare e da mantenere. La sua libreria è molto ricca e permette di personalizzare interfacce e applicazioni, oltre che, data la mancanza di puntatori, risulta anche molto robusto ed esula dalle molte cause di crash dei linguaggi precedenti. L'efficienza è leggermente ridotta, dato che Java è un linguaggio interattivo e interpretatibile. Inoltre, ogni programma Java viene eseguito all'interno di una **sandbox**: il compilatore genera un codice (**bytecode**), eseguibile per una macchina virtulale, detta **JVM**, *Java Virtual Machine*, in modo che non sia accessibile da altre parti del computer. Una piccola precisazione: il bytecode è una sequenza di caratteri codificati, che contiene le informazioni relative alle classi e che viene dato in pasto alla JVM. Ogni progetto Java viene eseguito all'interno di una **Java Platform**, che contiene la JVM, la **Java API**, Application Programming Interface, generalmente organizzata in librerie di classi e interfacce. Esistono tre edizioni di Java Platform:

- **Java SE**, *Java Standard Edition*, che fornisce un ambiente per applicazioni Java essenziali.
- **Java EE**, *Java Enterprise Edition*, che fornisce un framework per lo sviluppo di applicazioni più complesse, come web-based.
- **Java ME**, *Java Micro Edition*, che fornisce un ambiente ottimizzato per piccoli dispositivi come i cellulari.

Per far girare correttamente un programma Java, è necessario aver installato il **JDK**, il *Java Development Kit*. Per usarlo, potrebbe essere necessario modificare le variabili di sistema del PC all'interno della voce `PATH`. Il parametro **classpath** serve a indicare il percorso in cui cercare i file `.class`, cioè tutti quei file di configurazione per il compilatore.

UTILIZZARE OGGETTI

Un programma Java è un insieme di **oggetti**, ognuno **istanza di una classe**, che si scambiano **messaggi** tra di loro. Quindi, per programmare in Java, è necessario definire delle classi e istanziare degli oggetti. Un oggetto è un'entità che si può manipolare tramite l'invocazione dei suoi **metodi**. Un metodo non è altro che una sequenza di istruzioni che possono accedere ai dati interni dell'oggetto. Ogni metodo è identificato dal **nome** e anche dalla **segnatura**, cioè la descrizione dei suoi argomenti. A causa di questo motivo, si potrebbero avere errori di ***overloading***, cioè una classe che definisce metodi differenti che hanno lo stesso nome. L'insieme dei metodi di una classe definisce la sua **interfaccia pubblica**. Invece, i dati interni definiscono l'**interfaccia privata**. I metodi possono svolgere un'azione oppure possono restituire un valore. Inoltre, un oggetto può avere uno **stato**, generalmente definito dalle sue **variabili d'istanza**. Una **variabile**, invece, è una zona di memoria usata all'interno di un programma, che ha un nome e un valore. Una variabile viene dichiarata e, generalmente, viene dichiarato anche un **valore iniziale**. Per convenzione, i nomi delle variabili cominciano con lettera minuscola, quelli delle classi con lettera maiuscola e, i nomi composti, usano una maiuscola ad ogni inizio di parola (contoCorrente). Quando scriviamo programmi complessi e articolati è buona norma utilizzare i **commenti**, delle spiegazioni destinate al lettore. Per commenti brevi usiamo `//` e per quelli lunghi, li delimitiamo con `/* */` .
Prima di andare avanti possiamo fare l'esempio di una classe predefinita, come la classe `String`, che modella sequenze di caratteri. Per ottenere un riferimento a oggetti `String` è necessario inserire sequenze di caratteri tra doppi apici. Così facendo, utilizzeremo tutti i metodi della classe `String` invocandoli tramite il punto. Una caratteristica particolare di questa classe è che è **immutabile**. Una volta creato un oggetto `String`, per modificarlo, sarà necessario crearne un altro.

```
String saluti = "Ciao"
int n = saluti.lenght()
System.out.println(n) // stampa il numero di caratteri di saluti
```

In generale, in Java, definiamo:

- **parametri espliciti**, i dati in ingresso ad un metodo.
- **parametri impliciti**: oggetto su cui è invocato il metodo.

Una classe può implementare un metodo particolare, chiamato **costruttore**. Il costruttore serve, appunto, a costruire nuovi oggetti. Il costruttore deve per forza chiamarsi come la classe. Per creare un oggetto di un classe deve essere invocato un costruttore, in corrispondenza dell'operatore `new`. Quando viene creato un oggetto, quindi, viene generata una zona di memoria in cui vengono conservati i valori delle variabili che inizializzano l'oggetto e tutti i metodi delle istanze di questa classe. I metodi possono essere:

- **metodi di accesso**, che non cambiano lo stato del parametro implicito.
- **metodi modificatori**, che cambiano lo stato del suo parametro implicito.

Ci siamo quasi, abbiamo quasi descritto tutte le regole basiche per scrivere un buon programma in Java. Durante l'esecuzione, un oggetto può perdere il riferimento e diventa inaccessibile, quindi inutile. Per questo Java utilizza una tecnica di raccoglimento automatico degli oggetti spazzatura, detta ***Garbage Collection***. Spesso, in Java, si tende a creare classi apposite per **testare** i propri programmi. Si costruiscono vari oggetti, si applicano metodi e si visualizzano i risultati. Per usare classi e librerie, oppure per riutilizzare il proprio codice, si possono **importare pacchetti**: `import java.awt.Rectangle`. Di solito si specifica il pacchetto e il nome della classe.

REALIZZARE CLASSI

Fino ad ora abbiamo visto come fare ad usare oggetti istanze di classi predefinite. Le classi, però, è possibile definirle. Un meccanismo basilare per quanto riguarda la costruzione di nuove classi è l'**astrazione**, che serve a inventare tipi di dati di più alto livello, basandosi anche sull'**incapsulamento**, ossia quando un programmatore che sta usando un oggetto, conosce il suo comportamento, ma non deve per forza conoscere la sua struttura interna.
In Java, il comportamento di un oggetto è descritto da una classe. Come già anticipato, un classe ha un'**interfaccia pubblica**, cioè un insieme di metodi che si invocano, e un'**implementazione nascosta**, generalmente variabili usate per implementare i metodi dell'interfaccia. Quindi, una classe è costituita da **metodi**. Un metodo ha un **prototipo**, che definisce il tipo del valore di ritorno, un **nome** e un **corpo**. Un metodo particolare è il **costruttore**, di cui abbiamo già descritto le particolarità. Costruttore e metodi formano l'interfaccia pubblica. Inoltre, insieme al prototipo e al nome, un metodo può avere un **argomento**, che consiste nella dichiarazione di uno o più parametri.
Una variabile dichiarata all'interno di una classe, ma al di fuori di tutti i metodi, è detta **variabile d'istanza**. Ogni metodo può avere accesso ad essa e serve a memorizzare informazioni all'infuori delle invocazioni. Il valore di queste variabili definisce lo **stato** di un oggetto e vengono tipicamente inizializzate dal costruttore. Una variabile d'istanza può essere usata solo dai metodi della classe e non da altri metodi di altre classi, dato che verrà contrassegnata con `private`.
Tutto quello che abbiamo scritto finora serve per la progettazione dell'interfaccia pubblica della classe, a cui seguirà poi un'implementazione esterna. Per fare questo, generalmente, si utilizza una **Classe Tester**, cioè una classe che contiene il metodo `main` e diverse istruzioni che abbiamo già definito.
Oltre alla variabili d'istanza, possiamo definire anche le **variabili locali**, che appartengono al metodo e seguono un ciclo di vita uguale a quello del metodo a cui appartengono. Per controllare gli accessi, usiamo le solite diciture di `public` e `private`. Per denotare l'oggetto di invocazione e passargli il riferimento alla variabile d'istanza, utilizziamo il `this`.

TIPI DI DATI FONDAMENTALI

Cos'è un tipo di dato? Generalmente è una nozione che permette di esprimere la natura stessa del dato. In che modo, quella sequenza di bit, verrà interpretata dal compilatore. Java è un linguaggio **fortemente tipizzato**, infatti è obbligatorio indicare il tipo di una variabile prima di utilizzarla. Java fornisce otto tipi primitivi, che ora vedremo:

- **interi**. Gli interi possono essere di tipo `byte`, 8 bit, `short`, 16 bit, `int`, 32 bit e `long` 64 bit. Come dice stesso la parola permettono di rappresentare numeri interi all'interno di un determinato range.
- **costanti intere**. Una costante intera è per default di tipo `int`. Alle costanti, però, è necessario aggiungere la dicitura `final` prima di dichiarare il tipo. Questo dice al compilatore che il valore della variabile non potrà essere modificato a runtime.
- **numeri con virgola**. I numeri con virgola possono essere di tipo `float`, 32 bit, o di tipo `double`, a 64 bit. Permettono di rappresentare i numeri con la virgola all'interno di un determinato range.
- **caratteri**. Seguono la codifica *Unicode* e sono di tipo `char`, 16 bit. Il carattere viene rappresentato dai singoli apici.
- **booleani**. I booleani possono essere di tipo `boolean`, 1 bit, e ammettono soltanto due risultati possibili, cioè `true` e `false`. Il valore di default è `false`.

Java ha generalmente gli stessi operatori del C:

- **operatori matematici**, come **+**, **-**, *****, **/**, che non sono applicabili però a variabili booleane.
- **operatori relazionali**, come **<**, **>**, **==**, **!=**, che producono risultati di tipo booleano.
- **operatori logici**, come **&**, **||** e **!**.

- **operatori bit a bit**, come **&**, **|**, **^**, che sono applicabili solo per tipi interi e caratteri.

In Java sono consentite le assegnazioni fra tipi diversi se, e solo se, i valori possono essere **convertiti** da un tipo all'altro. Generalmente abbiamo due tipi di conversioni:

- **conversioni implicite**. Si verificano quando per esempio assegniamo un valore numerico ad una variabile numerica di range più ampio (da `int` a `float`), perdendo, però, un po' di precisione. Oppure possiamo citare anche la conversione di numeri in stringhe, quando usiamo operatori matematici nelle espressioni con le stringhe.
- **conversioni esplicite**. Questo tipo di conversione viene detto **cast**. L'operatore di cast è costituito da un nome di tipo tra parentesi tonde, che precede il dato da convertire. Alcuni cast non sono permessi.

Per questo, ora, andremo a definire una tabella di conversioni. ***N***, indica che è impossibile la conversione. ***S*** indica che la conversione viene fatta automaticamente in maniera implicita. ***C***, indica che la conversione viene fatta tramite un casting esplicito.

TABELLA DI CONVERSIONE

Aa da \ a	boolean	byte	short	char	int	long	float	double
boolean		N	N	N	N	N	N	N
byte	N		S	C	S	S	S	S
short	N	C		C	S	S	S	S
char	N	C	C		S	S	S	S
int	N	C	C	C		S	S	S
long	N	C	C	C	C		S	S
float	N	C	C	C	C	C		S
double	N	C	C	C	C	C	C	

Aa da \ a	boolean	byte	short	char	int	long	float	double
Untitled								

In Java è opportuno definire anche un altro tipo. La classe `Math`, per esempio, contiene una serie di **metodi statici** che servono per calcolare funzioni matematiche su tipi primitivi. Un metodo statico non opera su un'istanza, cioè non è associato ad un oggetto, ma alla classe stessa. Questo significa che possono essere invocati senza alcun riferimento. Per definirli si aggiunge `static`. Per invocarli, quindi, basta mettere il nome della classe e il nome del metodo. Ne abbiamo già parlato, ma è opportuno nominare anche le **stringhe** come tipi di dati. Una stringa è una sequenza di caratteri immutabile. Le stringhe possono essere convertite in interi e viceversa.
Con l'oggetto `System.in`, possiamo leggere l'input da console, utilizzando la classe `Scanner` per manipolarlo. In base al tipo, ci sono diversi metodi di lettura:

- `nextInt()`, che legge il prossimo intero.
- `nextDouble()`, che legge il prossimo double.
- `nextLine()`, che legge la prossima riga.
- `next()`, che legge la prossima parola.

COSTRUTTI CONDIZIONALI

In Java, come in altri linguaggi di programmazione, un costrutto condizionale è un'espressione che permette di cambiare l'esecuzione di un programma e di eseguirne una porzione soltanto in base ad una scelta. In Java distinguiamo due principali costrutti.
Il primo che andiamo ad analizzare, è detto **if-else**, cioè se si verifica una determina condizione, svolgi queste operazioni, altrimenti, svolgine altre. Ogni condizione di questo tipo restituisce un booleano. La prima condizione sarà denotata da `if`, mentre useremo il costrutto `else if` se abbiamo un numero arbitrario di altre condizioni, mentre `else` se ne avremo soltanto un'altra. Di solito, quando abbiamo più condizioni annidate, è buona norma separarle da parentesi graffe, per non creare errori di leggibilità durante la compilazione.
Questo appena descritto, è un costrutto universale, cioè può rappresentare efficientemente tutti i tipi di condizione. Nonostante ciò, Java ne mette a disposizione un altro, detto **switch-case**. La semantica di questo costrutto è leggermente più complicata. Lo `switch` prende in consegna un parametro. In base al valore di questo parametro, l'esecuzione salterà direttamente al `case` corrispondente. Strettamente collegato allo switch-case, è l'istruzione **break**. Aggiungendo `break` alla fine di ogni case, esso termina l'esecuzione di quel blocco di programma, permettendone il giusto funzionamento. Non è usato per i costrutti condizionali, ma la dicitura `continue` è molto legata al break. Si tratta di uno statement molto simile che, a differenza del break, non fa terminare il ciclo, ma viene semplicemente interrotta l'esecuzione corrente. Inoltre, allo switch-case si aggiunge alla fine anche un default case, contrassegnando con `default` una condizione che non rientra all'interno dei casi di studio.

COSTRUTTI ITERATIVI

Abbiamo fino ad ora descritto i costrutti condizionali, che eseguono una porzione di codice a seconda di una determinata condizione. I costrutti iterativi, invece, ci permettono di **eseguire ripetutamente** quella porzione di codice. Come per il C, anche in Java, per le iterazioni si utilizzano tre principali costrutti.
Il **ciclo while** esegue un'istruzione all'interno di un blocco di codice, finché rimane vera una certa condizione. La parola chiave è, ovviamente, `while` e può prendere come parametro una variabile o un'espressione. Il while ripete le istruzioni fino a quando quella condizione non diventa vera. Molto simile al while, è il **ciclo do-while**. Esso, insieme alla parola chiave `while`, integra anche la parola chiave `do`. Questo ciclo fa più o meno la stessa cosa, cioè il blocco viene eseguito fino a quando la condizione nel while non rimane vera ma, a differenza del precedente, la condizione viene controllata soltanto alla fine dell'esecuzione. Quindi, anche se la condizione risulta subito falsa, il blocco viene comunque eseguito almeno una volta.
Il ciclo più conosciuto è sicuramente il **ciclo for**. Il for utilizza la parola chiave `for`, appunto, e presenta un'espressione di **inizializzazione**, che viene eseguita una solta volta, un'espressione di **aggiornamento** da eseguire al termine di ogni esecuzione e un'espressione di **terminazione**, che generalmente è sempre un incremento o un decremento. Una pratica comune è quella di utilizzare **variabili di iterazione**, indicate quasi sempre con `i` o con `j`. Col for, però, è più rischioso incorrere in **cicli infiniti**. Una variante molto importante del for è il **for-each**, che serve quando si deve eseguire un determinato blocco di codice per ogni elemento di una collezione, liste o arrays.

ARRAY E VETTORI

Prima di entrare nel merito, ci soffermiamo su cosa effettivamente sono le **collezioni**. Un collezione è una grande quantità di dati, che sono tra loro collegati. In termini più inerenti a Java, una collezione di oggetti è essa stessa un oggetto. Java fornisce molte classi per gestire le collezioni ma, oggi, ci soffermeremo soltanto su due.
La prima sono gli **array**. Un array è una sequenza di lunghezza prefissata che contiene valori dello stesso tipo. Le posizioni sono individuate da un indice. Essendo un oggetto, un array si crea con l'operatore `new` e possiamo accedervi tramite variabili di riferimento. Per dichiarare un array usiamo un tipo di dato, seguito da `[]`. Infine, come abbiamo già anticipato, creiamo un'istanza dell'array tramite `new`, seguito dal tipo e dalla grandezza del vettore in parentesi quadre. Le stesse parentesi quadre, poi, consentono di avere accesso agli elementi dell'array. L'array, però, resta comunque uno strumento molto limitato. Principalmente, a causa della sua natura statica e per l'impossibilità di creare liste usando vere e proprie classi o interfacce, ma soltanto tipi primitivi. Per questo, venne introdotto l'**ArrayList**. `ArrayList` è una classe e gestisce anch'essa sequenze di oggetti. Essa, a differenza dell'array, può crescere o ridursi a proprio piacimento, utilizzando diversi metodi. Il tipo dell'ArrayList viene specificato all'interno di `<>`, mentre il metodo `size()` restituisce il numero di elementi all'interno della lista. Per aggiungere oggetti alla lista si utilizza il metodo `add()`, che può prendere come parametro anche l'indice, se per caso abbiamo intenzione di aggiungere un elemento in un determinato punto, facendo slittare in avanti gli altri. Per accedere agli elementi si utilizza il metodo `get()`, che prende come parametro l'indice. Ha questa particolarità anche il metodo `remove()`, che però serve a rimuovere quel determinato elemento dalla lista. Per rimpiazzare un oggetto, invece, usiamo il metodo `set()`, che usa come parametri l'indice e l'oggetto che vogliamo aggiungere. Come abbiamo detto, l'ArrayList colleziona oggetti e, per i tipi primitivi è necessario usare **classi Wrapper**, che sono degli involucri. Fortunatamente, la conversione tra i tipi primitivi e il loro corrispondente Wrapper è, ora, automatica grazie ad un processo detto **auto-boxing**. Per scorrere tutti gli elementi di un

ArrayList possiamo loopare una variabile iterativa che va da zero fino alla sua lunghezza, inserendola in un ciclo for. Ma, come già anticipato, Java mette a disposizione una versione semplificata, detta **for-each**, cioè un for generalizzato che prendere come parametri una variabile dello stesso tipo della lista e la lista stessa, scandendo poi tutti gli elementi.
In Java sono supportati anche gli **array multidimensionali**, la cui sintassi è molto simile e prevede solo l'aggiunta di altre parentesi quadre quante siano le dimensioni arbitrarie della variabile. Ci teniamo a dire che, quando creiamo una variabile array, viene memorizzato un riferimento all'array. Copiando la variabile otterremo un secondo riferimento allo stesso array. Per fare una vera e propria copia, quindi, servirà il metodo `clone()`, che otterrà due riferimenti separati. Per copiare, invece, gli elementi da un array ad un altro, Java mette a disposizione il metodo `arraycopy()`.

PROGETTAZIONE DI CLASSI

In Java, per costruire un'applicazione efficiente, è bene fare una buona progettazione iniziale, concentrandosi soprattutto sulle **classi**. Una classe è un singolo concetto del dominio dell'applicazione e il suo nome esprime, appunto, questo concetto. Un classe può esprimere un concetto matematico, oppure può essere un'astrazione di un'entità della vita reale. Scegliere bene una classe è fondamentale. Se dal nome non capiamo cosa dovrebbero fare gli oggetti, oppure se il nome non rappresenta un'entità ma una singola funzione, in generale stiamo facendo una cattiva progettazione. Ci sono sostanzialmente due criteri per analizzare la qualità di un'interfaccia pubblica di una classe. Il primo è le **coesione**. Una classe è coesa se la sua interfaccia contiene operazioni tipiche del concetto che la classe realizza. Scrivere classi con bassa coesione, cioè che svolgono un lavoro sparso e non correlato, porta a incorrere in numerosi sintomi di cattiva progettazione. Il codice non potrà essere riusato o mantenuto e, inoltre, avremo una scarsa flessibilità. Il secondo criterio è l'**accoppiamento**. L'accoppiamento è generalmente una dipendenza. Una classe A dipende da una classe B se usa esemplari di B. Avere molte dipendenze causa un accoppiamento elevato che, però, può far incorrere in alcuni problemi durante le modifiche o durante il riuso. Come abbiamo già accennato, un metodo può essere di accesso, che non cambia lo stato del parametro implicito, o modificatore, che lo cambia. Un metodo modificatore può incorrere in numerosi effetti collaterali, perché possono modificare i parametri e creare dipendenze. È opportuno anche che i metodi non stampino messaggi di errore e che si lasci questo compito alle **eccezioni**, che vedremo in seguito. Un'altra cosa importante per una buona progettazione, è l'uso delle **pre-condizioni**. Sono dei requisiti che devono essere soddisfatti affinché un metodo possa essere invocato. Quindi, le pre-condizioni, devono essere allegate alla documentazione del metodo. Un buon modo per controllare le precondizioni, è lanciare le eccezioni. Oppure è possibile utilizzare `assert`. È una condizione necessaria che lo sviluppatore ritiene si debba verificare in un determinato momento. Di solito, un'asserzione è un buon compromesso tra non fare nulla o lanciare eccezioni. Al contrario, le **post-condizioni**, devono essere soddisfatte al

termine dell'esecuzione del metodo. Anch'esse devono essere allegate alla documentazione. Ci sono due tipi di post-condizioni: una avviene quando il metodo funziona correttamente e l'altra quando il metodo si trova in un determinato stato. Molto importanti all'interno di una buona progettazione, sono i **metodi statici**. Come già anticipato, un metodo statico non ha il parametro implicito e non opera su una particolare istanza della classe. Usare bene i metodi statici è fondamentale perché, usarne troppi, significa fare poca programmazione ad oggetti. Al contempo, una **variabile statica**, permette di definire delle proprietà comuni a tutte le istanze della classe. Per questo è buona norma dichiarare le **costanti statiche**. Così facendo, le potremo usare liberamente. La visibilità delle variabili è molto importante. Possiamo farci riferimento tramite il proprio scope, cioè il suo nome, oppure tramite la sua dichiarazione alla fine del blocco. Sovrapporre variabili locali e variabili d'istanza è possibile, e si fa usando il parametro `this` che non permette alle prime di oscurare le seconde. Ultimo, ma non meno importante, è il corretto utilizzo dei **package**. Utilizzando i pacchetti, possiamo già garantirci una sorta di accoppiamento, assicurandoci che siano identificativi e che il nome sia scritto con lettere minuscole. E poi, bisogna evitare le **collisioni**, nel senso evitare che due pacchetti contengano la stessa classe. Per importare i pacchetti utilizziamo `import` che, a differenza delle inclusioni che abbiamo già visto in C, dice solo dove si trova la classe, senza aggiungere nulla al file sorgente.

EREDITARIETÀ

In Java, quando parliamo di ereditarietà, ci riferiamo ad una relazione tra una classe generica, chiamata **superclasse** e altre più specifiche, dette **sottoclassi**. Le sottoclassi ereditano i dati e i comportamenti della propria superclasse. Per fare un esempio, la superclasse potrebbe essere un generico `Veicolo`, mentre le sottoclassi sono `Automobili`, `Autocarri` o `Motocicli`. Un oggetto di una sottoclasse può essere tranquillamente utilizzato al posto di un oggetto della superclasse, dato che ne eredita metodi e attributi. Per definire una relazione di ereditarietà si utilizza la parola chiave `extends`. Un metodo che sovrascrive l'omonimo metodo della superclasse può estendere o sostituirne le funzionalità. Per invocare un metodo della superclasse si usa la parola chiave `super`. Il costruttore di una sottoclasse invoca il costruttore della superclasse senza parametri, utilizzando proprio `super` come primo enunciato. Ovviamente, vengono ereditate anche la variabili d'istanza e sarà possibile aggiungerne di nuove, che però non avranno nulla a che fare con la superclasse. Sarà buona norma, quindi, contrassegnare le variabili d'istanza della superclasse come `protected` e non `private`. Una variabile `protected` sarà comunque privata, ma accessibile a tutte le classi figlie.
Arriviamo, adesso, ad un concetto molto importante all'interno della programmazione a oggetti. Un riferimento ad una sottoclasse può essere utilizzato in ogni punto del programma che preveda la presenza di un riferimento alla superclasse. Un concetto davvero basico per l'ereditarietà e poi, ovviamente, anche per l'intera programmazione ad oggetti, è il **polimorfismo**, che significa effettivamente avere molte forme. Ci riferiamo generalmente alla possibilità, data una determinata espressione, di assumere valori diversi in relazione ai tipi di dato a cui viene applicata. Il polimorfismo si può verificarsi in determinate circostanze:

- in presenza di **overloading** che, come abbiamo visto, permette di definire più versioni dello stesso metodo, variando il numero o il tipo dei parametri.
- in presenza di **overriding**, cioè quando una classe derivata può avere un metodo identico ad un metodo presente nella classe base. Si tratta di

un'operazione che facciamo normalmente con l'ereditarietà, appunto, ereditando metodi astratti dalla superclasse.

- in presenza della **programmazione generica**, che è un metodo che permette la definizione di tipi parametrizzati, che vedremo successivamente nel dettaglio.

In Java, non è consentito che una classe abbia più padri attraverso l'ereditarietà. Il meccanismo, però, viene supportato attraverso l'uso di interfacce. Con l'**ereditarietà multipla**, una classe può estendere una sola classe, ma può implementare un numero illimitato di interfacce.

INTERFACCE

Un'interfaccia, in Java, nasce dall'esigenza di individuare un servizio che sia disponibile per una clientela molto vasta. Per esempio, si potrebbe creare un'interfaccia `serve()` che verrà usata allo stesso modo da più ristoranti.
Un'interfaccia dichiara una collezione di metodi, specificandone il tipo e il valore restituito senza, però, fornire alcuna implementazione. Dichiarando un'interfaccia si dichiara un **contratto**, cioè se una classe vorrà utilizzare questa interfaccia dovrà per forza implementare i suoi metodi. Attenzione, un'interfaccia non è una classe, anche se la dichiarazione è simile. Infatti, un'interfaccia non ha variabili d'istanza e i metodi sono tutti astratti. Per indicare una classe che implementa un'interfaccia, usiamo la parola chiave `implements`. Le interfacce sono molto importanti, perché riducono fortemente l'accoppiamento tra classi. Una serie di classi collegate tra loro, dipenderà soltanto dall'interfaccia e non le une dalle altre.
Per definire un'interfaccia basta dichiara un semplice file `.java`.
A volte, non è necessario definire un'interfaccia all'interno di un file, ma è possibile usare interfacce standard. Per esempio, l'interfaccia `Comparable` contiene il metodo `compareTo`, che serve per confrontare oggetti e, di solito, viene utilizzato per ordinare liste e collezioni. Oppure, l'interfaccia `Iterator`, che indica alcuni metodi per gestire collezioni di oggetti. Di particolare importanza sono le **interfacce di smistamento**, o di ***callback***. Questo meccanismo permette di specificare alcuni blocchi di codice che verranno eseguiti in un secondo momento. Strettamente legato alle interfacce è il concetto di **classe interna**. Si tratta di una classe che viene dichiarata all'interno di un'altra classe e ha uno scopo limitato, infatti non c'è bisogno di renderla visibile in altre zone del programma. Le classi interne utilizzano **oggetti dimostrativi**, detti ***mock***, che forniscono gli stessi servizi di un altro oggetto, ma in modo semplificato. La classe interna e quella completa hanno la medesima interfaccia pubblica.
In Java acquisisce una determinata importanza un **evento temporizzatore**. Un temporizzatore genera eventi a intervalli di tempo fissi. Per utilizzarli si attinge alla classe `Timer` in `javax.swing`.