

# Object-Oriented Programming (JAVA)

ORACLE



Libera mente Java™

# Object-oriented programming (Java)

## Introduzione

### 1.1 Caratteristiche generali

Java è un linguaggio di programmazione di alto livello, **Object-oriented** (*orientato agli oggetti*), sviluppato da *Sun Microsystems* nel 1995.

*Object-oriented* è un **paradigma di programmazione** che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. Ci sono diversi vantaggi della programmazione orientata agli oggetti: i software sono facili da modificare e da mantenere, inoltre, consente alti livelli di *riutilizzabilità* del codice.

Java ha una ricchissima libreria per lo sviluppo di interfacce utente e di applicazioni Internet impiegabili con relativa facilità. È un linguaggio **robusto** poiché una delle principali cause di *crash* dei programmi scritti in *C/C++* è l'uso scorretto dell'aritmetica dei puntatori: infatti Java non fornisce tipi puntatori, né tanto meno l'aritmetica dei puntatori. Inoltre, possiede un meccanismo automatico della gestione della memoria (Garbage Collection) che esonera il programmatore dall'obbligo di dover deallocare la memoria quando non ce ne bisogno.

È un linguaggio **efficiente** pur essendo un linguaggio *interpretato*: i programmi Java sono, mediamente, meno di 10 volte più lenti dei corrispondenti programmi *C++*, anche se l'efficienza si riduce per particolari applicazioni Java e in quel caso sarebbe meglio utilizzare altri linguaggi interpretati (*Basic, PERL, etc...*).

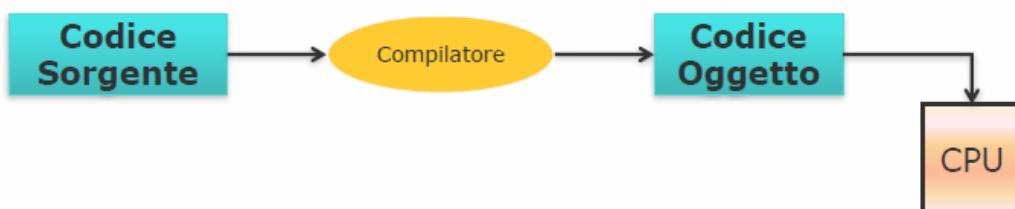
È **sicuro**, poiché l'esecuzione dei programmi è confinata in un "firewall" da cui non è possibile accedere ad altre parti del computer e ciò è estremamente utile per l'esecuzione di programmi scaricati da Internet.

Java è un linguaggio **portabile**: offre la possibilità di compilare il nostro codice su qualsiasi macchina, indipendentemente dal software o dall'hardware. Infatti, i programmi scritti in linguaggi convenzionali (es. *C/C++*) devono essere ricompilati per la nuova piattaforma, mentre in Java il compilatore genera un codice (**Bytecode**) eseguibile per una CPU virtuale, detta macchina virtuale (**JVM**). La macchina virtuale viene poi simulata su una CPU reale.

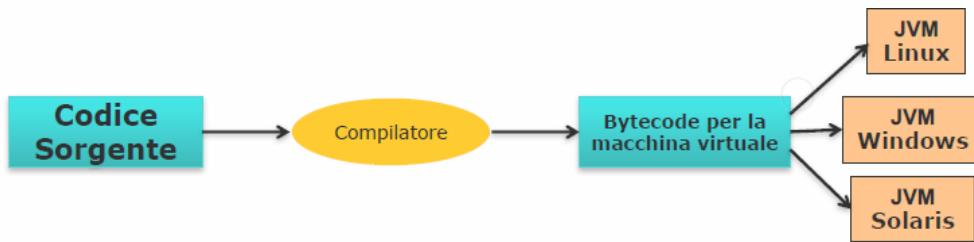
### 1.2 Differenza compilatore e interprete

La **compilazione** e l'**interpretazione** sono due metodi per eseguire un programma.

- ❖ La **compilazione** traduce tutte le istruzioni di un programma in linguaggio nativo, eliminando le ridondanze e poi le sottomette alla CPU, creando un file eseguibile (*oggetto*) ed è immediatamente eseguibile dal processore. Ogni volta che eseguo il programma, eseguo il file eseguibile. Il file sorgente non svolge alcuna funzione durante l'esecuzione del programma. La compilazione viene eseguita da un software, il *compilatore*.



- ❖ L'**interpretazione** traduce ed esegue ogni singola istruzione del programma, traducendola in linguaggio macchina per farla eseguire dal processore. Legge ed esegue il codice sorgente del programma senza creare un file oggetto eseguibile. Tutti questi passaggi si ripetono ogni volta che eseguo il programma. Per questa ragione l'esecuzione di un programma interpretato è più lenta.



## 1.3 Bytecode

Un file di **bytecode** è costituito da una sequenza di caratteri in formato **Unicode** (uno schema di codifica più ricco di *ASCII/ANSI* che sono un sottoinsieme dell'*Unicode*). Il *bytecode* contiene tutte le informazioni che descrivono le classi ed è dato “in pasto” alla *JVM* che ne legge i contenuti. Con opportuni tool è possibile ricostruire il codice sorgente del programma a partire dal *bytecode*.

## 1.4 Java Platform

La **Java platform** è solo software e viene eseguita al di sopra di altre piattaforme hardware: *Java Virtual Machine (JVM)*, *Java Application Programming Interface (Java API)*.

La *Java Platform* è una collezione di software pronti per l'uso, inoltre, è organizzata in librerie di classi e interfacce correlate, detti **packages**. Abbiamo 3 edizioni principali:

- 1) **Standard Edition (Java SE™)**: fornisce ambiente runtime per esecuzione di applicazioni Java; API essenziali per sviluppare applicazioni di vario tipo (applet, applicazioni stand-alone).
- 2) **Enterprise Edition (Java EE™)**: fornisce un *framework* per lo sviluppo di applicazioni *server-side* complesse. Adatta allo sviluppo di applicazioni *Web-based* a livello di impresa, per commercio elettronico.
- 3) **Micro Edition (Java ME™)**: un *Java runtime environment* altamente ottimizzato indirizzato specificamente a “computer piccoli”: smart card, telefonini, PDA.

Il **Java SE** comprende di un **JDK (Java Development Kit)** formato da un insieme base di programmi che consente di far girare applicazioni scritte nel linguaggio *Java*. I programmi più importanti sono il compilatore *Java* (**javac**) che traduce il sorgente *Java* in codice eseguibile dalla macchina virtuale *Java*, e l'esecutore (**java**), che implementa la vera e propria macchina virtuale.

## 1.5 API

**API (Java Application Programming Interface)** è un codice già scritto, strutturato in *packages* relativi a un insieme di argomenti comuni. Un **package** è una collezione di classi correlate e interfacce, che forniscono accesso protetto e *namespace management*. Per usare una classe o un'interfaccia in un package bisogna usare il nome completo della classe (Es: `java.util.ArrayList`) o importare la classe (`import java.util.ArrayList;`) o importare tutto il package (`import java.util.*;`).

## 1.6 JVM (Java Virtual Machine)

La **JVM** è un calcolatore astratto che consiste di un **caricatore di classi** e di un interprete del linguaggio che esegue il *bytecode*. Il caricatore delle classi carica i file *.class*, sia del programma scritto in Java sia della libreria API, affinché l'interprete possa eseguirli. Il verificatore delle classi controlla la correttezza sintattica del codice *bytecode*. La JVM gestisce la memoria in modo automatico poiché possiede un proprio **garbage collector** che consiste nel recupero delle aree di memoria assegnate ad oggetti non più in uso per poi restituirla al sistema.

La **JVM** e le librerie di base costituiscono il *Java Runtime Environment (JRE)*, necessario per esecuzione di programmi Java. Per mantenere l'efficienza di un programma Java sono stati introdotti dei compilatori *Just In Time (JIT)*, che alla prima esecuzione prendono pezzi di *bytecode* e li analizzano.

## 1.7 Programmazione

Un **programma** è un software, ovvero un insieme di programmi che consentono all'hardware di svolgere i compiti richiesti, che può essere eseguito da un elaboratore e che riceve dati in input di un problema per poi produrre una soluzione algoritmica restituendo dati output.

Un programma è un insieme di linee di codice a loro volta costituite da un insieme di istruzioni. Il problema deve essere risolvibile attraverso un algoritmo affinché un programmatore possa codificarlo in istruzioni in un linguaggio di programmazione; in questa fase - detta **programmazione** - viene realizzato il codice sorgente del programma.

Il linguaggio di programmazione può essere di diverso tipo:

- **Basso livello.** È un linguaggio di programmazione più vicino a quello della macchina e ciò è utile poiché rende il programma veloce e ingombra poca memoria, infatti avremo le istruzioni in codice binario eseguibili direttamente. Lo svantaggio è che costringe a pensare soluzioni che riflettono il modo di operare dell'elaboratore, rendendo difficoltoso il riutilizzo del codice e la manutenzione.
- **Alto livello.** È un linguaggio di programmazione orientato verso l'uomo che rende il programma più pesante in termini di memoria, inoltre, non è richiesta una conoscenza dell'hardware. Questo tipo di linguaggio non è adatto allo sviluppo di software di base (SO, Driver, compilatori, ...).

## 1.8 Classi e oggetti

In Java si costruiscono programmi usando **oggetti**. Ogni **oggetto** ha un comportamento e può essere manipolato mediante l'invocazione dei suoi **metodi**. Un *metodo* è costituito da una sequenza di istruzioni che possono accedere ai dati interni dell'oggetto. Un oggetto esiste fino a quanto c'è una variabile di riferimento che si riferisce a lui, poi avviene la distruzione automatica.

Nella OOP, un **oggetto** può contenere elementi concreti, ma anche entità come processi o concetti teorici e astratti. Gli oggetti ma sono identificati dall'avere:

- **stato**, contiene le informazioni conservate nell'oggetto, inoltre condiziona il suo comportamento nel futuro e può variare nel tempo per effetto di un'operazione sull'oggetto;
- “**comportamento**”, è definito dai metodi che possono essere eseguiti dall'oggetto e tali metodi possono modificare anche lo stato dell'oggetto.

L'interazione tra diversi oggetti avviene attraverso lo scambio di **messaggi**. In Java un “**oggetto**” è un'istanza di una certa **classe**, ed esso può possedere alcuni metodi. Quindi un oggetto può ricevere un certo messaggio se possiede un metodo che l'oggetto *sender* è in grado di chiamare.

Per ogni oggetto è necessario sapere qual è il **set dei messaggi** che è in grado di ricevere.

Una **classe** è un'astrazione indicante un insieme di oggetti che condividono le stesse caratteristiche e funzionalità. La *classe* definisce un TIPO, inoltre, in essa vengono definiti tutti i messaggi che ciascun oggetto sarà in grado di ricevere mediante la definizione di **metodi**. Il comportamento di un oggetto è descritto da una *classe*. Ogni classe ha:

- Un **identificatore**.
- Un **insieme di attributi**.
- Un'**interfaccia pubblica**. L'insieme dei metodi (funzioni) che si possono invocare per manipolare l'oggetto.
- Un'**implementazione nascosta**. Codice e variabili sono usati per implementare i metodi dell'interfaccia e non sono accessibili all'esterno della classe.

Una *classe* è a tutti gli effetti un **tipo di dato** (detto **tipo riferimento**), infatti, nella programmazione orientata agli oggetti è sia possibile usare tipi di dato esistenti (`int`, `float`, `char`, ...) detti **tipi primitivi**, sia definirne di nuovi tramite le classi.

Un programma Java è un insieme di oggetti, ognuno istanza di una classe, che si inviano messaggi.

### 1.8.1 Riferimenti a oggetti

In Java, una “*variabile oggetto*” (cioè una variabile il cui tipo sia una classe) memorizza al proprio interno solo la **posizione dell'oggetto** perché risulta più efficiente memorizzare soltanto la posizione dell'oggetto, piuttosto che l'oggetto intero.

Un **riferimento** a un oggetto descrive la posizione dell'oggetto in memoria. E' possibile avere più riferimenti ad uno stesso oggetto.

### 1.9 Variabili

Quando un programma manipola oggetti c'è bisogno di memorizzare gli oggetti stessi e i valori restituiti dai loro metodi, in modo da poterli utilizzare anche in seguito. Per memorizzare valori in un programma Java si usano le **variabili**.

Una variabile è una zona di memoria usata all'interno di un programma: ciascuna **variabile** ha un nome e contiene un valore o può essere utilizzata per riferirsi ad oggetti. Quando si dichiara una variabile di solito se ne specifica anche un valore iniziale, cioè la si **inizializza** attraverso l'operatore **assegnamento (=)**.

Per convenzione i nomi delle variabili cominciano per lettera minuscola, mentre i nomi delle classi cominciano per lettera maiuscola.

- ❖ Le **variabili locali** sono variabili presenti dentro il metodo, la loro presenza nell'esecuzione del programma termina alla fine dell'esecuzione del metodo. La variabile avrà influenza esclusivamente all'interno dello scope dove è inserita. (Lo Scope è la sezione racchiusa tra le due parentesi graffe di apertura e chiusura).
- ❖ Le **variabili di istanza**, anche note come *field* o **campi**, sono dichiarate all'interno di una classe ma all'esterno di ogni metodo, quindi hanno valore anche all'interno di ogni metodo. Le variabili di istanza solitamente possono essere lette e modificate solo dai metodi della stessa classe. Se esistessero due variabili con lo stesso identificativo le *Variabili Locali* prenderebbero la precedenza sulle *Variali di Istanza*. Un'**istanza di una variabile** (non statica) continua ad esistere nella memoria di un programma fino a quando esiste l'oggetto che la contiene. Lo specificatore di accesso **private**

indica che la variabile di istanza può essere letta e modificata solo dai metodi della classe. La variabile di istanza viene inizializzata automaticamente a un valore predefinito: se sono di tipo numerico sono inizializzate a 0 altrimenti se sono variabili oggetto sono inizializzate a NULL.

Dall'esterno è possibile accedere alle variabili di istanza **private** solo attraverso i metodi pubblici della classe e solo raramente le variabili di istanza sono dichiarate **public**. Il controllo di accesso, applicato sia ai metodi che alle variabili di istanza può essere:

- 1) **public**: consente l'accesso al di fuori della classe;
- 2) **private**: limita l'accesso ai membri della classe.

## 1.10 Incapsulamento

L'**information hiding** (o *incapsulamento*) è una strategia adottata anche da altri linguaggi di programmazione con il fine di nascondere i dettagli realizzativi, rendendo invece pubblica un'interfaccia. Usiamo il termine **black box** per indicare un qualsiasi dispositivo, costrutto, i cui meccanismi interni di funzionamento vengono tenuti nascosti all'utente. In Java, il costrutto *class* realizza l'incapsulamento, mentre i **metodi pubblici** di una classe rappresentano l'**interfaccia pubblica** della classe attraverso cui viene manipolata. Gli attributi di una classe sono dichiarati privati attraverso il modificatore **private**.

L'incapsulamento agevola l'individuazione di errori e la modifica dei dettagli realizzativi di una classe: abbiamo una maggior manutenibilità e riuso del codice.

## 1.11 Metodi

Un programma svolge il proprio lavoro invocando **metodi** con i propri oggetti. I metodi rappresentano le operazioni che mettono in comunicazione gli oggetti, permettendo di interagire tra loro.

I metodi costituiscono l'**interfaccia pubblica** della classe determinando ciò che può essere fatto con gli oggetti della classe. Una classe definisce anche una **realizzazione privata**, che descrive i dati interni ai propri oggetti e le istruzioni per l'esecuzione dei propri metodi: tali dettagli non sono noti ai programmatore che usano oggetti. Le variabili vengono passate al metodo sempre per VALORE. L'accesso al metodo avviene sempre tramite l'operatore **dot** (punto). È possibile creare metodi che accettano un numero non definito di parametri attraverso l'uso di "...".

Nel momento in cui avremo una classe che definisce metodi diversi ma con lo stesso nome, si avrà un **overloading**.

## 1.11 Static

Possiamo dichiarare gli attributi come **statici**: questi attributi, detti **variabili di classe**, sono condivisi da tutti gli oggetti di quella classe attraverso l'accesso all'area globale della memoria. Le **variabili di classe** vengono inizializzate non appena parte il nostro programma; le **variabile di istanza**, invece, vengono istanziati solo dopo che è stato invocato il costruttore.

I **metodi statici** (o **metodi di classe**) non operano su una istanza e non sono associati ad un oggetto ma alla classe stessa, basta inserire **static**. Quindi i *metodi statici* manipolano esclusivamente tipi primitivi. Si possono invocare senza alcun riferimento all'oggetto.

Il **main** è statico perché non ci sono oggetti istanziati dato che il **main** è il primo metodo ad essere invocato per avviare l'esecuzione del programma.

## 1.12 Parametri impliciti ed esplicativi

**Parametri esplicativi:** sono i dati in ingresso ad un metodo. Non tutti i metodi hanno parametri esplicativi.

```
System.out.println(greeting);  
greeting.length(); // senza parametri esplicativi
```

**Parametro隐式:** è l'oggetto su cui è invocato il metodo.

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

## 1.13 Immutabilità

Un oggetto si dice **immutable**, quando una volta creato e inizializzato, non può essere più modificato. Un metodo che accede a un oggetto e restituisce alcune informazioni a esso relative, senza modificare l'oggetto stesso, viene chiamato **metodo d'accesso**, mentre un metodo che abbia lo scopo di modificare i dati interni di un oggetto viene chiamato **metodo modificatore**.

Consideriamo come esempio la **classe String**. Un oggetto di tipo *String* è **immutable** in quanto una volta creato, non è possibile modificarlo. Il metodo non modifica l'oggetto ma agisce su una copia dell'oggetto originario, restituendo appunto un nuovo oggetto con il contenuto modificato.

## 1.14 Creare oggetti

In Java per creare un oggetto di una classe è necessario eseguire il **metodo costruttore**. Il costruttore ha sempre lo stesso nome della classe. Esso non deve restituire alcun valore, ma non deve neanche essere dichiarato di tipo **void**, ma di solito viene dichiarato di tipo **public**. Per compiere le operazioni di inizializzazione vengono spesso usati i parametri.

L'espressione **new** fornisce un oggetto e, se lo si vuole utilizzare in seguito, bisogna memorizzarlo in una variabile.

## 1.15 Garbage collection

Quando un oggetto non ha più un riferimento non può più essere referenziato da alcun programma. L'oggetto diventa inaccessibile e quindi inutile e viene chiamato **garbage** (spazzatura).

Java effettua una raccolta automatica e periodica degli oggetti inutili (**garbage collection**) per rendere nuovamente utilizzabile per usi futuri lo spazio di memoria che l'oggetto occupava e per ridurre lo spazio occupato dallo heap. In altri linguaggi è responsabilità del programmatore effettuare il rilascio della memoria occupata da oggetti non referenziati e quindi inaccessibili.

## 1.16 Astrazione

**L'astrazione** è una procedura mentale utile nella descrizione, progettazione, implementazione e utilizzo di sistemi complessi. I dettagli trascurabili vengono *incapsulati* in sottosistemi più semplici che vengono quindi utilizzati come delle *black box*. Non occorre conoscere il loro funzionamento interno ma basta conoscere l'essenza del concetto che rappresentano e l'interazione con il resto del sistema.

L'astrazione è usata per inventare tipi di dati di più alto livello. Nella programmazione OO gli oggetti sono scatole nere e i programmati usano l'Incapsulamento al fine di usare un oggetto conoscendo il suo comportamento, ma non la sua struttura interna.

## 1.17 Definire i costruttori di una classe

In Java un **costruttore** è molto simile a un metodo ma con due differenze:

1. Il nome di un costruttore è sempre uguale al nome della classe.
2. I costruttori non definiscono un tipo per il “valore restituito”, neanche *void*.

I costruttori e i metodi pubblici di una classe costituiscono la sua **interfaccia pubblica**, ovvero le operazioni che qualsiasi programmatore può utilizzare per creare e manipolare oggetti.

Un costruttore assegna un valore iniziale alle variabili di istanza di un oggetto. Ci possono essere anche costruttori con lo stesso nome, in questo caso si parla di **overloading** e il compilatore riesce a distinguere i vari costruttori come per i metodi, guardando la lista dei parametri espliciti.

Per collaudare una classe ci sono due possibilità. La prima consiste nell'utilizzare alcuni IDE che consentono di creare oggetti e di invocarne i metodi e verificando che vengano prodotti i risultati attesi.

In alternativa, si può scrivere una classe chiamata **Test** che contiene il metodo *main*.



## Tipi di dato

### 3.1 Tipi di numeri

Ogni valore, in Java, è un **riferimento a un oggetto** oppure appartiene a un degli **8 tipi primitivi**: 6 sono numerici fra cui 4 rappresentano numeri interi e 2 numeri in virgola mobile.

Il **tipo del dato** consente di esprimere la natura del dato, indica il modo con cui verrà interpretata la sequenza di bit che rappresenta il dato. La stessa sequenza può rappresentare un intero o un carattere. Il tipo di dato determina il dominio dei valori che un dato può assumere e specifica le operazioni possibili sui dati.

Java è un linguaggio fortemente **tipizzato**. Il tipo di ogni variabile o espressione può essere identificato leggendo il programma ed è già noto al momento della compilazione. È obbligatorio dichiarare il tipo di una variabile prima di utilizzarla, inoltre, durante la compilazione sono effettuati tutti i controlli relativi alla compatibilità dei tipi e sono stabilite eventuali conversioni implicite. Dopo la dichiarazione non è possibile assegnare alla variabile valori di tipo diverso, tranne in casi particolari.

### 3.2 Tipi primitivi: interi

Java fornisce **8 tipi primitivi** indipendenti dall'implementazione e dalla piattaforma. Gli **interi** sono **4** e sono:

- Tipo **byte (8 bit)** → Interi con segno tra -128 e 127, valore di default 0
- Tipo **short (16 bit)** → Interi con segno tra -32768 e 32767, valore di default 0
- Tipo **int (32 bit)** → Interi con segno tra -231 e 231-1, valore di default 0
- Tipo **long (64 bit)** → Interi con segno tra -263 e 263-1, valore di default 0

#### 3.2.1 Costanti intere

Una **costante intera** per default è di tipo **int**. Le costanti intere possono essere espresse anche in ottale (prefisso *0*) o in esadecimale (prefisso *0x*). Per costanti intere di tipo **long** bisogna aggiungere il suffisso *L* oppure *I*. In Java le **costanti** sono identificate dalla parola **final**. Dopo aver ricevuto il suo valore iniziale, una variabile con l'attributo final non può più essere modificata altrimenti il compilatore segnalerà un errore. Molti programmatore usano per le costanti nomi con le lettere tutte maiuscole.

Spesso le costanti vengono usate in più metodi, per cui è comodo dichiararle insieme alle variabili di istanza della classe, aggiungendo l'attributo **static** prima di final. **Static** denota una variabile della classe, quindi non ne viene creata una copia per ogni oggetto istanziato ma tutti gli oggetti fanno riferimento alla stessa variabile

### 3.3 Tipi primitivi: numeri con virgola

- Tipo **float (32 bit)** → numeri in virgola mobile con 7 cifre significative (*DOMINIO*: compresi tra 1.4E - 45 e 3.4028235E + 38). Il valore di default è 0.0. Tali costanti vanno terminate con F o f.
- Tipo **double (64 bit)** → numeri in virgola mobile in doppia precisione (15 cifre significative). Sono comprese tra 4.9E - 324 e 1.7976931348623157E + 308. Il valore di default è 0.0. Le costanti con virgola sono di tipo double per default e possono essere terminate con D o d ma non è necessario.

### 3.4 Tipi primitivi: caratteri

Tipo **char** (16 bit) → Si possono usare i caratteri o i relativi codici numerici preceduti da \u, sempre tra apici.  
**Esempio:** char a='A';                    char a='\u0041';                    char a=65;

### 3.5 Tipi primitivi: boolean

Tipo **boolean** (1 bit) → Ammette solo due possibili valori (**true**, **false**). Il valore di default è *false* e non si possono assegnare interi alle variabili booleane. Da ricordare che false non è 0.

### 3.5 Conversione implicita di tipo

La conversione del tipo si applica automaticamente quando assegnamo ad una variabile di tipo più grande un tipo più piccolo, altrimenti il compilatore riceverà errore:

*byte → short → int → long → float → double*  
*char → int → long → float → double*

Per convertire un tipo di dato utilizziamo le regole di **promozione** che si applicano automaticamente alle espressioni che contengono dei valori di due o più tipi di dato, dette anche espressioni di tipo misto. Ogni valore in un'espressione di tipo misto sarà promosso automaticamente a quello più alto dell'espressione (in realtà verrà creata una copia temporanea di ognuno dei valori, lasciando gli originali invariati). Per forzare la promozione e trasformare un dato in un tipo "inferiore" o "superiore", applichiamo il **cast**. Bisogna ricordare che convertire dei dati in tipi inferiori comporta la perdita di informazioni.

## Array e ArrayList

### 4.1 Introduzione

Molto spesso è necessario manipolare grandi quantità di dati fra loro correlati (*collezioni*). In generale, una **collezione di oggetti** è a sua volta un oggetto e Java fornisce per le collezioni di dati l'utilizzo di **array** e **ArrayList** (pacchetto `java.util`).

### 4.2 Array

In Java gli **array** sono degli oggetti. E' una collezione di *reference* a un altro oggetto, di altri array, oppure di tipi primitivi. Un array è una sequenza di elementi dello stesso tipo con una dimensione prefissata. Ogni posizione è individuata da un indice. Per utilizzarli bisogna: dichiararli, crearli e inizializzarli. Un oggetto array nasconde la sua implementazione interna e mette a disposizione una interfaccia per l'utilizzo. L'accesso ai dati è controllato, quindi non si può eccedere nella dimensione di un array: viene generata un'eccezione.

La dichiarazione avviene come in C: `int vect[]` oppure `int[] vect`.

La creazione avviene attraverso l'operatore `new`: `int[] vect = new int[3]`, array di tre interi.

Se l'array è formato da *reference* come elementi, bisogna inizializzare ogni elemento:

`(vect[1] = new SavingsAccount)`.

### 4.3 ArrayList

La classe **ArrayList** (pacchetto `java.util`) gestisce una sequenza di oggetti. Può crescere e ridursi a piacimento e implementa nei suoi metodi le operazioni più comuni sulle collezioni di elementi: inserimento, cancellazione, modifica, accesso dati.

La classe **ArrayList** è generica: contiene elementi di tipo **Object**:

```
ArrayList<Object> nomeVariabile = new ArrayList<>()
```

- Il metodo `size()` restituisce il numero di elementi della collezione.
- Per aggiungere l'elemento alla fine della collezione si usa il metodo `add(obj)` e dopo l'inserimento, la dimensione della collezione aumenta di uno. Invece, per aggiungere un elemento in una certa posizione, facendo slittare in avanti gli altri, si usa il metodo `add(i, obj)`.
- Se si usa **ArrayList** di tipo *Object*, per utilizzare i metodi dell'oggetto inserito occorre fare il **cast**, altrimenti si possono solo usare i metodi di *Object*.
- Per rimuovere un elemento da una collezione si usa il metodo `remove(indice)` che restituisce l'oggetto rimosso e gli elementi che seguono slittano di una posizione all'indietro.
- Per modificare un elemento si usa il metodo `set(indice, obj)` che restituisce l'oggetto rimpiazzato.
- Per accedere ad un elemento bisogna usare il metodo `get(indice)`.
- Per copiare un array si invoca il metodo `clone()`:  
`double[] prices = (double[]) data.clone();`

Se accediamo ad una posizione non valida, quindi fuori dal dominio dell'**ArrayList**, il compilatore genera un'eccezione: **IndexOutOfBoundsException (java.lang)**.

### 4.3.1 Classe wrapper

Poiché l'*ArrayList* memorizza oggetti, vengono utilizzate delle classi dette **wrapper**, per contenere ed eseguire operazioni con i dati primitivi. I dati primitivi, infatti, non posseggono metodi e non sono associati ad una classe. A partire da **Java 5.0**, la conversione tra i tipi primitivi e le corrispondenti *classi wrapper* è automatica.

La **classe wrapper** è un involucro (*wrap*) che ha lo scopo di contenere un valore primitivo rendendolo da un lato un oggetto e dall'altro lo arricchisce con metodi. Tutte le classi *wrapper* sono definite nel package `java.lang` e sono qualificate come **final**. Inoltre, tutte queste classi sono **immutabili**, cioè non è possibile cambiare valore dopo la costruzione.

Tipo primitivo	Classe Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

## Progettazione di classi

### 5.1 Scelta delle classi

Per scrivere una buona applicazione usando un linguaggio ad oggetti come Java è bene fare un'adeguata progettazione iniziale. Il punto su cui concentrarsi sono le **classi** e sulle **entità**, cioè gli oggetti appartenenti alle varie classi individuate. I **metodi**, cioè la parte funzionale, devono essere pensati come associati alle entità. Una classe rappresenta un singolo concetto del dominio dell'applicazione che viene espresso tramite il nome della classe. Una buona **interfaccia pubblica** di una classe deve rispettare due criteri:

1. **Coesione:** una classe deve rappresentare un singolo concetto ed è **coesa** se l'interfaccia contiene solo operazioni tipiche del concetto che la classe realizza.

Una classe con bassa coesione fa tante cose insieme, svolgendo molto lavoro "sparso" e non correlato. Questo tipo di situazione sarebbe da evitare in quanto queste classi risultano:

- complesse da riutilizzare (bassa riusabilità);
- complicate da manutenere (scarsa manutenibilità);
- delicate e critiche in quanto soggette a continui cambiamenti (bassa flessibilità).

Una forma comune di bassa coesione si ha in quelle classi che presentano un grandissimo numero di **metodi (pubblici o privati)**.

2. **Accoppiamento:** riguarda la dipendenza di una classe verso un'altra. Una classe A dipende da una classe B se usa oggetti o metodi di B. E' possibile avere molte classi che dipendono tra di loro (**accoppiamento elevato**) ma ci sarebbero troppi problemi:

- se una classe viene modificata tutte le classi che dipendono da essa potrebbero necessitare di una modifica;
- se si vuole usare una classe in un altro programma bisognerebbe usare anche tutte le classi da cui quella classe dipende.

**UML** è un linguaggio di modellazione e di specifica basato sul paradigma orientato agli oggetti e rappresenta i diagrammi delle dipendenze tra classi e oggetti.

### 5.2 Effetti collaterali

L'**effetto collaterale** è una qualsiasi modifica che può essere osservata al di fuori del metodo, ad esempio un metodo che modifica un parametro esplicito di tipo oggetto. Gli *effetti collaterali* possono introdurre dipendenze e possono causare comportamenti inattesi: una buona regola è di ridurre al minimo gli *effetti collaterali*.

### 5.3 Pre-condizioni

Le **precondizioni** sono requisiti che devono essere soddisfatti affinché un metodo possa essere invocato. Se le *precondizioni* di un metodo non venissero soddisfatte, il metodo potrebbe avere un comportamento sbagliato. Le *precondizioni* di un metodo devono essere pubblicate nella documentazione. Di solito vengono utilizzate per restringere il campo dei parametri di un metodo o per richiedere che un metodo venga chiamato solo quando l'oggetto si trova in uno stato appropriato.

### 5.3.1 Asserzioni

In Java un'asserzione è rappresentata dalla keyword **assert** che permette di verificare se una data espressione è vera o falsa. Se la valutazione dell'espressione risulterà falsa, il programma verrà interrotto con un'eccezione di tipo **AssertionError**; viceversa, se sarà vera, l'esecuzione del programma continuerà normalmente, perché vuol dire che il codice sta funzionando come previsto. Di default, le asserzioni sono disabilitate, per motivi di performance. Per abilitarle si deve usare l'opzione **-enableassertions** (o lo shortcut equivalente **-ea**). Da riga di comando si usa:

`java -enableassertions MyProg`. Una volta testato il programma basta disabilitarle.

### 5.4 Post-condizioni

Le **post-condizioni** devono essere soddisfatte al termine dell'esecuzione del metodo. Vanno riportate nella documentazione come per le *precondizioni*. Le *post-condizioni* vengono usate per analizzare il valore di ritorno di un metodo o per verificare che al termine dell'esecuzione del metodo, l'oggetto con cui il metodo è invocato si trovi in un determinato stato.

### 5.5 Package

Un **package** in Java permette di raggruppare in un'unica entità classi Java correlate. Fisicamente il *package* è una cartella all'interno del sistema operativo. Per eleggere una cartella a *package*, una classe Java deve dichiarare nel suo codice la sua appartenenza a quel *package*, e deve risiedere fisicamente al suo interno. Per convenzione i nomi dei *package* sono scritti in lettere minuscole. Per rendere unici i nomi dei *pacchetti* si possono usare i nomi dei domini Internet alla rovescia e tale nome deve coincidere con il percorso della sottocartella dove è residente il package. Se la dichiarazione è omessa, le classi create fanno parte di un package di **default** (*senza nome*).

Per rendere visibile una classe ad un'altra classe che in un'altra cartella, si usa il comando **import**, che va dichiarato dopo la dichiarazione del *package* e prima della dichiarazione della classe.

#### 5.5.1 Modificatori di accesso

I modificatori di accesso regolano la visibilità e l'accesso un componente Java; essi sono i seguenti:

- **public**: un membro pubblico può essere accessibile a qualsiasi classe di un qualsiasi package. Le variabili d'istanza non vanno mai dichiarate pubbliche;
- **protected**: un oggetto si può accedere dai metodi della classe, di tutte le sottoclassi, e da tutte le classi che si trovano nello stesso package. **Protected** può essere usato per forzare l'uso di alcuni metodi solo da oggetti della stessa classe o di una sottoclasse.
- **private**: restringe la visibilità alla sola classe in cui è definito;
- **nessun modificatore**: se non specificato, un membro è accessibile solo da classi appartenenti al package in cui è definito.

## Interfacce

### 6.1 Che cos'è un'interfaccia

Un'**interfaccia** dichiara una collezione di **metodi astratti**, ovvero metodi senza un corpo che posseggono solo la loro firma (con il tipo del valore restituito), inoltre, sono automaticamente **public** (non serve lo specificatore d'accesso). Un'*interfaccia* non è una classe: non si possono creare oggetti.

Un'*interfaccia* possiede solamente **costanti e metodi d'istanza astratti**. Il vantaggio è che le interfacce riducono l'accoppiamento tra classi ma anche per favorire il riutilizzo del codice.

Una classe può implementare anche più di una *interfaccia*: in questo caso basta elencare tutte le interfacce separate da virgola. Un'interfaccia va definita in un file .java che si chiama con lo stesso nome dell'interfaccia (esattamente come per le classi pubbliche). Per convertire un tipo interfaccia in un tipo classe occorre un **casting**.

### 6.2 Conversione fra tipi

L'operatore **instanceof** permette di verificare se un oggetto appartiene ad un determinato tipo o classe. L'operatore ha come primo operando un oggetto e come secondo operando una classe. Restituisce **true** se il primo operando è un oggetto che appartiene a quel tipo di classe oppure restituisce **false** in caso contrario.

## Polimorfismo

### 6.1 Introduzione

Il **polimorfismo** consente di riferirci con un unico termine ad entità diverse. Il **polimorfismo per i metodi** si ottiene utilizzando l'**overload** e l'**override** dei metodi stessi.

#### 6.1.1 Overload

All'interno di una classe possiamo avere più metodi che sono identificati dallo stesso nome ma che hanno, in ingresso, parametri di tipo e numero diverso. Questo concetto prende il nome di **overload**. Il tipo di ritorno non fa parte della firma di un metodo, quindi non ha importanza per l'implementazione dell'*overload*.

Con l'**overloading** la scelta del metodo appropriato avviene in fase di compilazione, esaminando il tipo dei parametri (*early binding*, effettuato dal compilatore).

Con il **polimorfismo** la scelta del metodo appropriato avviene in fase di esecuzione (*late binding*, effettuato dalla JVM).

#### 6.1.2 Override

Con il termine **override** si intende una vera e propria riscrittura di un metodo di una classe che abbiamo ereditato da una *superclasse*. Dunque, necessariamente, l'*override* implica **ereditarietà**. Quando un metodo in una sottoclassile ha lo stesso nome, lo stesso parametro o firma e lo stesso tipo restituito di un metodo nella relativa superclasse, si dice che il metodo nella sottoclassile esegue l'*override* del metodo nella superclasse.

## 6.2 Classi interne

Java permette di definire una classe (o interfaccia) all'interno di un'altra. Questo tipo particolari di **classi** vengono chiamate **interne** o annidate. Le classi che non si trovano all'interno di altre classi vengono chiamate “**top level**” perché si trovano al loro “livello massimo”.

Con l'introduzione delle classi interne vengono introdotte nuove regole di classe:

- Una **classe interna** può avere qualsiasi marcatori di visibilità, quindi anche `protected` e `private`. Questa novità consente una stretta collaborazione tra *classi interne* e *top level*, in particolare permette di nascondere le classi interne dall'esterno;
- Uno dei grandi vantaggi delle classi interne è che la classe contenuta e la classe che la contiene non hanno nessun problema di visibilità, ovvero vedono l'intero contenuto della classe in entrambe le direzioni, anche se la visibilità è `private`. Il beneficio consistente nello svincolamento dalle regole di visibilità, mentre lo svantaggio consiste nella violazione dell'incapsulamento che potrebbe portare a dei disastri se non opportunamente gestito;
- I metodi della *classe interna* hanno accesso alle variabili e ai metodi a cui possono accedere i metodi della classe in cui sono definite (accesso all'ambiente in cui è definita), inoltre, se definite in un *metodo statico* accedono solo alle variabili statiche non alle variabili di istanza.
- Le classi interne possono accedere a variabili locali solo se sono state dichiarate **final**. Lo stato dell'oggetto può cambiare, ma la variabile non può riferirsi ad un altro oggetto.

## 6.3 Testare una classe

In molti casi si ha la necessità di testare una classe prima che l'intera applicazione sia stata completata. A questo scopo vengono utilizzati i **mock object** (**mock**) al fine di simulare il comportamento di oggetti complessi e non utilizzabili. Ovviamente l'oggetto *mock* deve esporre la stessa interfaccia dell'oggetto che simula, consentendo al client di ignorare se sta interagendo con l'oggetto reale o con quello simulato. L'implementazione di un oggetto *mock* è un'operazione complessa e produce un codice finalizzato ai test. Per tale motivo sono nati **framework** che consentono di creare oggetti *mock* in modo rapido.

## 6.4 Classe Object

E' una classe specifica nel linguaggio Java: è la classe padre di tutte le classi in Java, infatti ogni classe in Java estende implicitamente la **classe Object**. Alcuni metodi importanti di *Object* sono:

- **toString()**: che restituisce una stringa descrittiva dell'oggetto.
- **equals()**: confronta due oggetti e restituisce un booleano. Se volessimo confrontare due oggetti creati da una nostra classe, dovremmo effettuare l'*override* del metodo in modo tale che venga restituito `true` se le variabili d'istanza dei due oggetti coincidono, altrimenti `false`.
- **hashCode()**: viene utilizzato per confrontare l'uguaglianza di due oggetti presenti nella stessa collezione. Se due oggetti sono uguali nel metodo `equals()` allora devono avere anche lo stesso *hash code* (un intero che rappresenta l'univocità di un oggetto).
- **clone()**: restituisce una copia dell'oggetto corrente (comprese le variabili d'istanza e i rispettivi valori). Clonare un oggetto ci permette di lavorare su una copia senza il rischio di modificare l'originale. La copia si limita a copiare i valori delle variabili d'istanza dell'oggetto: viene fatta una copia superficiale o **shallow copy**. Per effettuare una copia profonda, **deep copy**, bisogna riscrivere il metodo `clone()`.

# Ereditarietà

## 7.1 Introduzione

L'**ereditarietà** mette in relazione due classi che hanno caratteristiche in comune: è un meccanismo che estende classi esistenti, aggiungendo altri metodi e campi. Un grande vantaggio è la riusabilità del codice. La classe preesistente (più generica) è detta **SUPERCLASSE** e la nuova classe (più specifica) è detta **SOTTOCLASSE**. Ad esempio, la *classe Object* è la superclasse di tutte le classi: ogni classe è una sottoclasse di *Object*.

In Java le classi sono raggruppate in **gerarchie di ereditarietà**: le classi che rappresentano concetti più generali sono più vicine alla radice mentre le classi più specializzate sono nelle diramazioni.

## 7.2 Differenze ereditarietà e interfacce

Un'**interfaccia** non è una classe, non ha uno stato, né un comportamento ma è un elenco di metodi da implementare.

Una **sottoclasse** è una classe che ha uno stato e un comportamento che sono ereditati dalla superclasse.

## 7.3 Metodi di una sottoclasse

Ci sono tre possibilità per definire una sottoclasse:

1. **Sovrascrivere** i metodi della superclasse, in cui la sottoclasse ridefinisce un metodo con la stessa firma del metodo della superclasse. Vale il metodo della sottoclasse.
2. **Ereditare** metodi dalla superclasse, in cui la sottoclasse non ridefinisce nessun metodo della superclasse.
3. **Definire nuovi metodi**, in cui la sottoclasse definisce un metodo che non esiste nella superclasse.

I metodi della sottoclasse non possono accedere alle variabili private della superclasse.

Per accedere ai metodi della superclasse si usa lo strumento **super**: **super .deposit(amount)**.

Per invocare il costruttore della superclasse dal costruttore di una sottoclasse uso la parola chiave **super** seguita dai parametri del costruttore. Deve essere il primo comando del costruttore della sottoclasse:

**Esempio:** `public class CheckingAccount extends BankAccount{  
 public CheckingAccount(double initialBalance) {  
 super(initialBalance);  
 transactionCount = 0;  
 }  
}`

Se il costruttore della sottoclasse non chiama il costruttore della superclasse, viene invocato il costruttore predefinito della superclasse, ovvero **super()**.

### 7.3.1 Variabili di istanza di sottoclassi

Una sottoclasse può:

1. **Ereditare** le variabili d'istanza della superclasse.
2. **Definire nuove variabili d'istanza** che esisteranno solo negli oggetti della sottoclasse e avranno lo stesso nome di quelle nella superclasse, che **non saranno sovrascritte**. Inoltre, le variabili d'istanza della sottoclasse metteranno in ombra quelle della superclasse, ovvero **una sottoclasse non ha**

accesso alle variabili private della superclasse ed è un errore risolvere il problema creando un'altra variabile di istanza con lo stesso nome poiché le variabili della sottoclasse metteranno in ombra quelle della superclasse.

Si può salvare un riferimento ad una sottoclasse in una variabile di riferimento ad una superclasse memorizzando un qualsiasi oggetto in una variabile di tipo **Object**.

## 7.4 Fattorizzazione

L'ereditarietà viene usata oltre ad estendere le funzionalità di una classe anche per spostare un comportamento comune a due o più classi in una singola superclasse.

Ricordiamo che la OOP è anche **responsibility- driven programming**, ovvero la progettazione basata sulla responsabilità esprime l'idea che ogni classe debba essere responsabile della gestione dei propri dati. Spesso, quando è necessario aggiungere alcune nuove funzionalità a un'applicazione, è necessario chiedersi in quale classe è necessario aggiungere un metodo per implementare questa nuova funzione. L'obiettivo è di dividere la responsabilità tra le classi.

## 7.5 Classi abstract

Le **classi astratte** sono un ibrido tra classe ed interfaccia poiché possiedono alcuni metodi implementati normalmente ed altri astratti. Sono utilizzate per poter dichiarare caratteristiche comuni fra classi di una determinata gerarchia. Le classi che estendono una classe astratta sono **OBBLIGATE** ad implementarne i metodi astratti. I **metodi astratti** non hanno implementazione. Per usare le classi astratte si utilizza il modificatore **abstract** che può essere applicato sia a classi che a metodi.

Un **metodo astratto** non può essere invocato ma potrà essere soggetto a **override**; inoltre potrà essere definito solo all'interno di una classe astratta.

La sintassi è: `public abstract void calcolaArea()`.

Un **metodo astratto** definisce una interfaccia. Una classe dichiarata astratta non può essere istanziata, ovvero non potrà avere oggetti istanziati, ma ne devo dichiarare il costruttore. Il costruttore può essere chiamato solo all'interno di una classe derivata dalla classe astratta.

E' possibile dichiarare astratta una classe priva di metodi astratti, in tal modo evitiamo che possano essere costruiti oggetti di quella classe. Le classi non astratte sono dette **concrete**.

### 7.5.1 Classi astratte vs Interfacce

Un'**interfaccia** indica solo dei metodi da implementare: consente l'uso di "altre classi" per l'elaborazione di dati, può facilmente essere integrata in un progetto sviluppato indipendentemente ed è consentito implementare più interfacce con la stessa classe. Si decide di utilizzare una interfaccia se ci si trova nella situazione in cui alcune classi (non legate fra di loro) condividono i metodi di una interfaccia, se si vuole specificare il comportamento di un certo tipo di dato (ma non implementarne il comportamento).

Una **classe astratta** permette di definire delle variabili di *istanza/ statiche/ final* e fornisce una base per le classi che la estenderanno (una classe può estendere una sola superclasse). Si usa una classe astratta per condividere codice fra più classi, se più classi hanno in comune metodi e campi o se si vogliono dichiarare metodi comuni che non siano necessariamente campi *static* e *final*. Per impedire al programmatore di creare sottoclassi o di sovrascrivere certi metodi, si usa la parola chiave **final**:

- `public final class String` → (questa classe non si può estendere)
- `public final boolean checkPassword(...)` → (questo metodo non si può sovrascrivere)

## 7.6 Ereditarietà multipla

Alcuni linguaggi di programmazione permettono di utilizzare l'**ereditarietà multipla**, in cui una classe può ereditare funzionalità e caratteristiche da più di una superclasse. Questa tecnica si contrappone all'**ereditarietà singola**, in cui una classe può ereditare da una, e solo una, classe base.

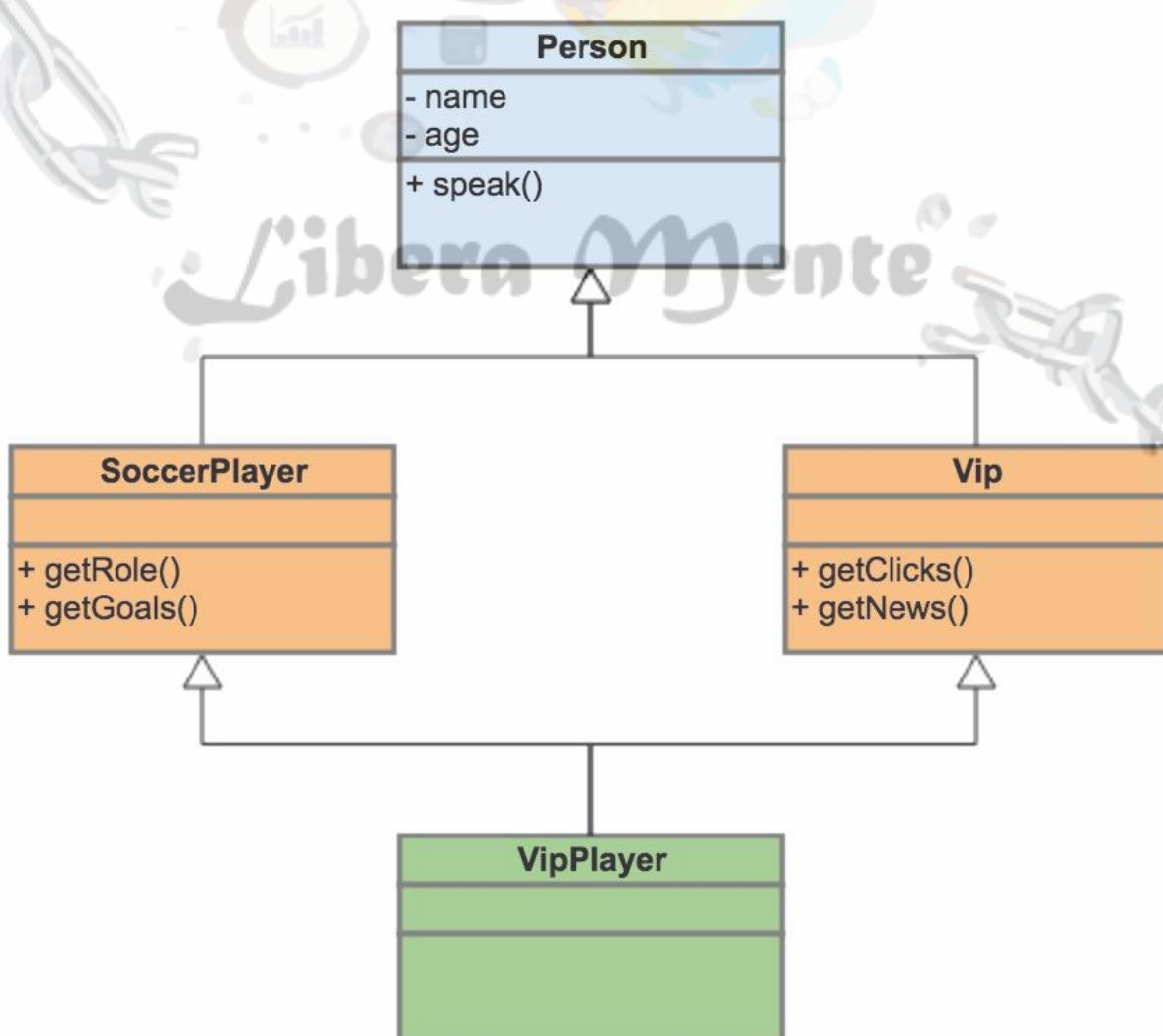
Java, diversamente dal C++ non supporta il concetto di *ereditarietà multipla*. In Java una classe può ereditare da una sola superclasse, ed il concetto di *ereditarietà multipla* è possibile solo grazie all'uso delle **interfacce**.

Un'*interfaccia* permette di definire la “forma” di una classe, indicando nomi di metodi, argomenti, tipi restituiti e proprietà, ma senza permettere alcuna implementazione. Saranno poi le classi che ne faranno uso a dover implementare la logica per ciascun metodo definito nell'interfaccia

L'ereditarietà multipla oltre a molti benefici porta, soprattutto, a molte problematiche. Prima tra queste il **problema del diamante**: esiste una sottoclasse che eredita un metodo presente in due superclassi allo stesso livello con lo stesso nome. Il compilatore non può identificare il metodo esatto che stiamo cercando di usare.

Creiamo un nuovo metodo che usa le implementazioni delle due interfacce. Invece di duplicare codice è possibile usare la parola chiave **super** e richiamare il metodo.

Si usa la sintassi: **NomeSuperClasse . super . nomeMetodoSuperclasse**.



## Debugging

### 8.1 Introduzione

Un **malfunzionamento** di un programma è causato da un difetto nel codice chiamato **errore** o **bug**. Per risolvere questi inconvenienti si testano le varie parti del codice attraverso l'uso di vari strumenti.

### 8.2 Program Trace

Il **Program Trace** tiene traccia dei messaggi che mostrano un cammino di esecuzione e stampano il contenuto dello *stack* di esecuzione. Lo svantaggio dell'uso del *program tracing* risiede nel fatto che bisogna rimuovere i messaggi una volta che il testing è completo e reinserirli se viene individuato un nuovo errore. Un'alternativa è usare la classe **Logger** per zittire le tracce dei messaggi senza rimuoverli.

### 8.3 Logging

Il **logging** è il processo di scrittura dei messaggi di registro durante l'esecuzione di un programma. Il *logging* consente di segnalare e mantenere messaggi di errore e di avviso in modo che i messaggi possano essere successivamente recuperati e analizzati. L'oggetto che esegue l'accesso alle applicazioni è chiamato **Logger**, implementato nel pacchetto: `java.util.logging`.

L'API di Java che contiene il *logging*, consente di configurare i tipi di messaggio scritti attraverso i **levels**: **Level** è una classe utilizzata per definire quali messaggi devono essere scritti nel log e definisce anche la gravità di un messaggio.

Una volta creato il *logger*, questo scrive in memoria. Per farlo scrivere su un file bisogna appoggiarsi ad un altro tipo di classe, detta **Handler**. Ci sono vari tipi di Handler standard:

- ❖ **StreamHandler**: Scrive su un `OutputStream`.
- ❖ **ConsoleHandler**: Scrive su `System.err`
- ❖ **FileHandler**: Scrive su un file o vari file a rotazione.
- ❖ **SocketHandler**: Scrive su una porta TCP remota.

Per scrivere su un file utilizzeremo la classe **FileHandler**. Il logging dei messaggi può essere disattivato quando il testing è completo.

### 8.4 Asserzioni

Le asserzioni sono utilizzate per controllare condizioni di errore e si possono disattivare una volta che il programma supera il collaudo. Le asserzioni sono usate principalmente per controllare le precondizioni.

### 8.5 Debugger

Per programmi molto grandi semplicemente il *logging* non è praticabile. La maggior parte dei programmatore usano un **debugger** per individuare e correggere un errore. Il **debugger** è un programma che esegue il programma sotto collaudo e analizza il suo comportamento a *run-time*.

Un *debugger* permette di arrestare e far ripartire l'esecuzione del programma usando i **breakpoint**, permette di visualizzare il contenuto delle variabili ed eseguire il programma un passo alla volta (**single step**). I debuggers sono sia parte di un IDE o programmi a parte.

## Breakpoint

L'esecuzione del programma è sospesa ogni volta che viene raggiunto un **breakpoint**. Quando l'esecuzione si arresta si può:

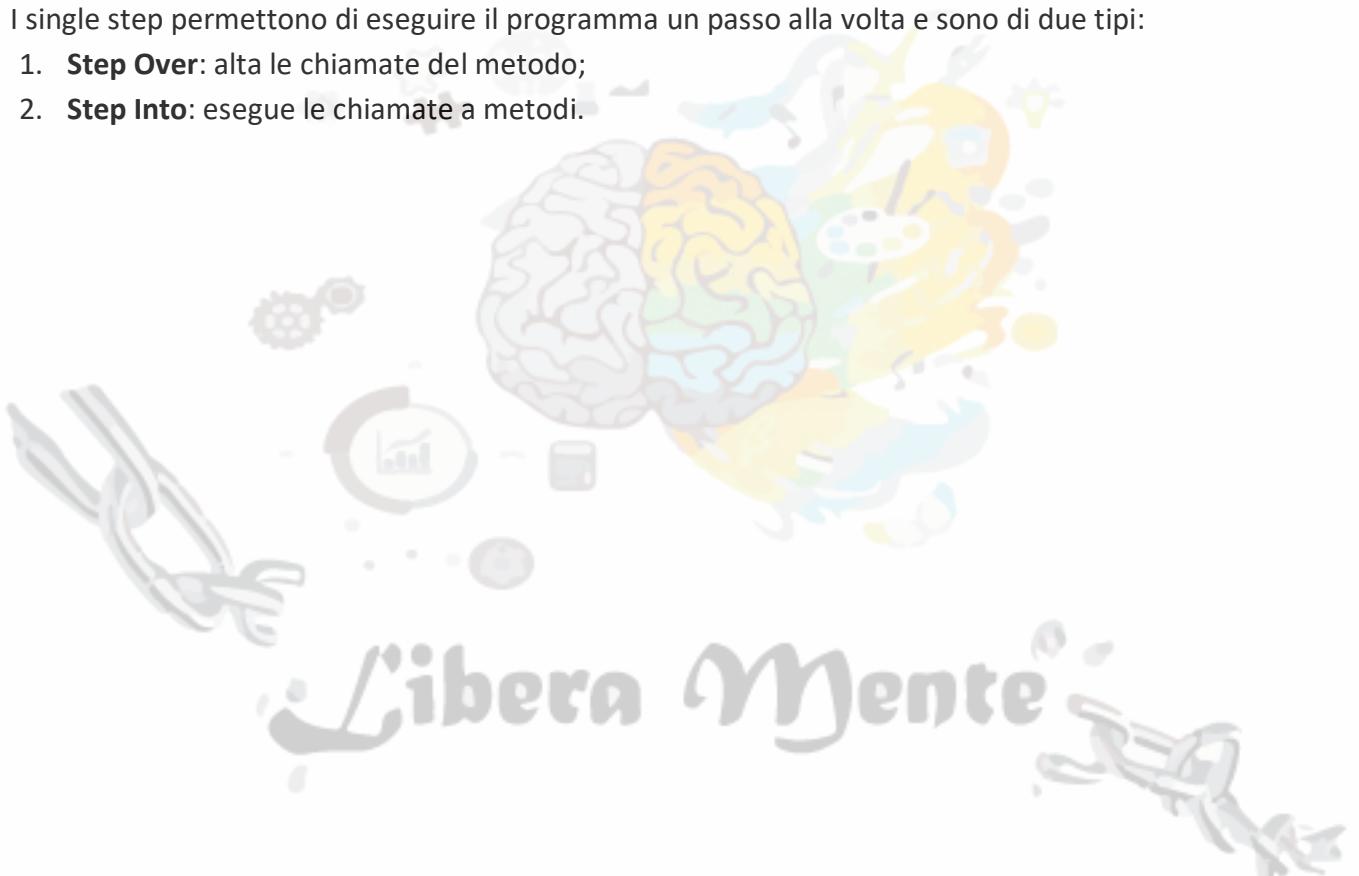
- Ispezionare le variabili;
- Eseguire il programma una linea alla volta;
- Oppure eseguire il programma in maniera normale finché non raggiunge il prossimo *breakpoint*.

Quando il programma termina, anche il *debugger* termina mentre i *breakpoint* restano attivi finché non vengono rimossi.

## Single-step

I single step permettono di eseguire il programma un passo alla volta e sono di due tipi:

1. **Step Over**: alta le chiamate del metodo;
2. **Step Into**: esegue le chiamate a metodi.



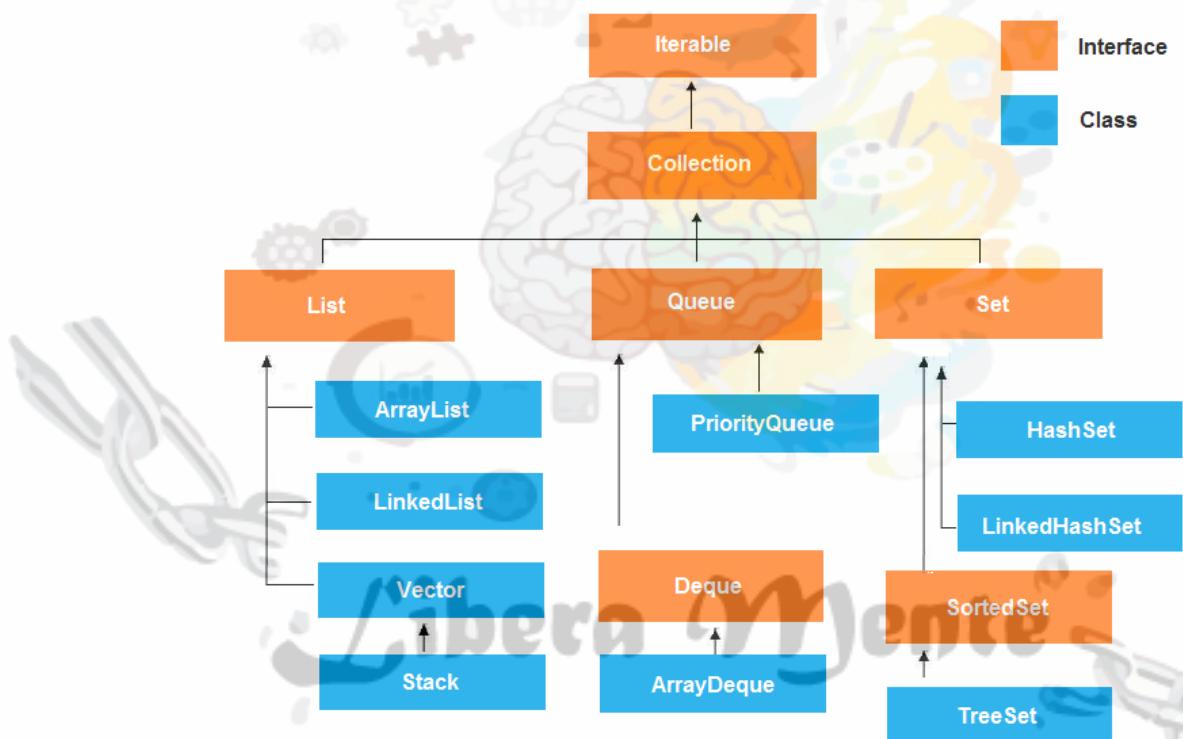
# Java Collections Framework

## 9.1 Introduzione

La libreria standard di Java fornisce una serie di classi che consentono di lavorare con **gruppi di oggetti (collezioni)**. Tali classi della libreria standard prendono il nome di **Java Collections Framework**. Sono tutte classi contenute nel package `java.util` della libreria standard di Java.

Il *Java Collections Framework* prevede due tipologie principali di collezione:

1. **Insiemi**. Un insieme corrisponde a un gruppo di oggetti tutti distinti tra loro (gli elementi possono essere mantenuti secondo qualche ordinamento).
2. **Liste**. Una lista corrisponde a un gruppo di oggetti in cui possiamo avere elementi ripetuti; in questo caso gli oggetti sono generalmente mantenuti in ordine di inserimento.



Il *Java Collections Framework* descrive, quindi, le caratteristiche generali delle principali tipologie di collezioni tramite opportune **interfacce**.

## 9.2 Interfaccia Collection<E>

L'interfaccia **Collection** descrive genericamente le funzionalità di una collezione ed è formata da elementi dello stesso tipo e supporta una serie di operazioni: `add`, `remove`, `contains`, `isEmpty`, `size`, `toArray`, `equals`. Tale interfaccia è estesa da 4 interfacce: **Set**, **List**, **Queue**, **Deque**.

### 1. Queue e Deque

Estendono l'interfaccia *Collection* definendo nuovi metodi per l'inserimento, la rimozione e l'utilizzo dei dati. Ognuno di questi metodi è presente in due formati differenti: se l'operazione fallisce un formato lancia un'eccezione, mentre l'altro restituisce un valore speciale (o l'oggetto o un booleano). I metodi che lanciano un'eccezione sono: `add()`, `remove()`, `element()`.

Le code possono avere un'implementazione di tipo *FIFO* o *LIFO*.

L'interfaccia **Deque** è una collezione lineare che supporta l'inserimento e la rimozione dei suoi elementi ad entrambe le estremità.

## 2. Set

L'interfaccia **Set** è un'estensione di *Collection*, descrive le funzionalità di un insieme e quindi NON ammette elementi duplicati, inoltre, non aggiunge nuovi metodi e pone dei vincoli sull'uso dei metodi:

- **add**: aggiunge un elemento sole se non è presente;
- **equals**: restituisce `true` se i due *Set* confrontati contengono gli stessi elementi indipendentemente dall'ordine.

Inoltre, le classi `HashSet` e `TreeSet` implementano l'interfaccia Set; in particolare:

- ❖ **TreeSet**: viene utilizzata quando è utile mantenere gli elementi ordinati secondo il loro ordine naturale. Inoltre, garantisce un tempo  $\log(n)$  per le operazioni di inserimento, ricerca e cancellazione di un elemento. Un insieme ordinato può essere rappresentato da un albero binario, infatti, per vedere se un elemento è presente:
  1. si parte dalla radice dell'albero;
  2. si scende a destra o a sinistra a seconda che l'elemento cercato sia minore o maggiore di quello corrente;
  3. si ripete il procedimento finché non si trova l'elemento e non si raggiunge una foglia.
- ❖ **HashSet**: non garantisce l'ordine degli elementi ma è più performante rispetto alla *TreeSet*. Utilizza al suo interno una **tabella hash** e le operazioni possono essere eseguite in tempo costante (ossia indipendentemente dal numero di elementi contenuti nell'insieme).  
Una *tabella hash* rappresenta un insieme come un array di liste e usa una funzione hash che, dato un elemento, restituisce un numero. Se la *funzione hash* riesce a "spalmare" bene gli elementi nelle posizioni, le operazioni di ricerca, inserimento e cancellazione di un elemento diventano veloci.  
Gli oggetti componenti un *HashSet* devono ridefinire la funzione **hashCode()**:
  1. se due oggetti sono uguali per `equals()`, `hashCode()` deve restituire per forza due valori uguali;
  2. se due oggetti per `equals` sono differenti, la `hashCode` può dargli valori diversi, oppure uguali (in questo caso si verifica una collisione).
- ❖ **LinkedHashSet**: così come fanno le liste, mantiene l'ordine degli elementi con il loro ordine di inserimento.

## 3. List

L'interfaccia **List** estende *Collection* introducendo l'idea di sequenza di elementi, ciò vuol dire che ogni elemento è caratterizzato da una posizione. Tale interfaccia ammette elementi duplicati, eredita tutti i metodi definiti nell'interfaccia *Collection* e pone dei vincoli su alcuni metodi di *Collection*:

- **add** aggiunge un elemento in ultima posizione (*append*);
- **equals** restituisce `true` se i due *List* confrontati contengono gli stessi elementi nelle stesse posizioni.

**List** possiede come implementazioni: `ArrayList`, `Vector` e `LinkedList`.

- ❖ **ArrayList** implementa sia le interfacce **Deque** che **Queue**. Possiede un parametro di configurazione per la capacità iniziale. Quando viene aggiunto un elemento al di fuori della dimensione, la struttura viene ridimensionata e poi viene aggiunto l'elemento. Questa doppia operazione può portare ad un decadimento delle prestazioni. È possibile ovviare a ciò tramite il metodo `ensureCapacity()` che prende in ingresso la nuova capacità, dopo di che si può fare

`l.add()`. Se viene rimosso un elemento la capacità non diminuisce ma potrebbe tramite il metodo `trimToSize()`.

La ricerca di un elemento richiede un tempo proporzionale al numero di elementi nella lista (scansione sequenziale dell'array, tempo lineare).

L'inserimento di un elemento nel mezzo della lista richiede di spostare in avanti tutti gli elementi che seguono (anche qui tempo lineare).

- ❖ **LinkedList** è una lista concatenata in cui ogni elemento mantiene il reference all'elemento successivo e a quello precedente. Si sceglie di utilizzare una `LinkedList` solo nel caso in cui bisogna aggiungere elementi in testa o se bisogna eliminare e aggiungere elementi frequentemente nel mezzo della lista.

La ricerca di un elemento richiede di scandire la lista da un'estremità in un tempo lineare, come per `ArrayList` e `Vector`.

L'accesso posizionale in una `LinkedList` è lineare e bisogna scandire l'intera lista, mentre costante in un `ArrayList`.

### 9.3 Iteratori

Un **iteratore** è un oggetto di una classe che implementa l'interfaccia `Iterator<E>`. L'interfaccia `Collection<E>` estende l'interfaccia `Iterable<E>` che contiene il metodo `iterator()` il quale restituisce un *iteratore* per la collezione. L'interfaccia `Iterator` prevede già tutti i metodi necessari per usare un iteratore. Non è necessario conoscere alcun dettaglio implementativo.

L'interfaccia `Iterator` prevede i seguenti metodi:

- ❖ **next()** : restituisce il prossimo elemento della collezione e contemporaneamente sposta il "cursore" dell'iteratore all'elemento successivo;
- ❖ **hasNext()** : verifica se c'è un elemento successivo da fornire o se invece si è raggiunta la fine della collezione;
- ❖ **remove()** : elimina l'elemento restituito dall'ultima invocazione di `next()` .

Si noti che l'iteratore non ha alcuna funzione che lo "resetti": una volta iniziata la scansione, non si può far tornare indietro l'iteratore e una volta che la scansione finisce, l'iteratore non è più utilizzabile (bisogna crearne uno nuovo). Gli iteratori sono il meccanismo usato da Java per realizzare i cicli *for-each*. Usare gli iteratori in maniera esplicita consente di fare più cose:

- suddividere (sospendere) la scansione della collezione in due cicli successivi che usano lo stesso iteratore;
- rimuovere elementi dalla collezione mentre si itera su essa (proibito con il *for-each*);
- interrompere il ciclo prima di aver scandito tutta la collezione.

Bisogna fare **attenzione**: quando si usa un iteratore su una collezione è bene non apportare modifiche alla collezione stessa (aggiungere/rimuovere elementi) se non tramite il metodo `remove()` di `Iterator` perché il comportamento dell'iteratore dopo una modifica della collezione non si può prevedere.

## 9.4 Ordinamento

Per ordinare liste di oggetti in Java sono a disposizione due interfacce:

- **java.lang.Comparable**: di solito si utilizza per definire l'ordinamento “naturale” di un oggetto e comprende un metodo con firma. L'ordine naturale è definito dal `compareTo()`.
- **java.util.Comparator**: si utilizza quando non abbiamo una relazione d'ordine o non vogliamo modificare una classe e quindi dobbiamo creare una classe che implementa l'interfaccia **Comparator**, nella quale andiamo a fare l'*overload* del metodo `compare(Object obj1, Object obj2)` che restituirà un intero.

## 9.5 Map

La **Map** (o anche *tabella associativa*) è una struttura dati caratterizzata dal fatto che ogni **entry** inserita nella struttura è caratterizzata da due elementi: una *chiave* e un *valore*. Ogni elemento è, quindi, una coppia *chiave-valore*. Per ogni elemento la chiave è univoca: non ci sono due elementi che hanno lo stesso valore sul campo chiave. Possiamo vedere la *Map* come un *Set* di chiavi, e una *Collection* di valori. *Map* è implementata da un'altra interfaccia, **SortedMap** che è la versione ordinata di *Map*: gli ordinamenti si fanno sui valori della chiave. I metodi più importanti di *Map* sono:

- **V put(K key, V value)** : aggiunge una coppia chiave-valore alla mappa e ritorna il vecchio valore V se era stato già mappato su quel valore della chiave, altrimenti ritorna null.
- **V get(Object key)** : restituisce il valore associato alla chiave passata in ingresso.
- **V remove(Object key)** : rimuove l'intera mappatura chiave-valore e ne ritorna il valore associato a quella chiave.

**HashMap<K, V>** (implementazione di *Map*) non è un'implementazione di *Collection*, ma è comunque una struttura dati molto usata del Java *Collection Framework*: realizza una struttura dati “dizionario” che associa termini chiave (univoci) a valori.

# Programmazione generica

## 10.1 Classi generiche e tipi parametrici

La **programmazione generica** (introdotta dalla versione Java 5.0) consiste nella creazione di costrutti che possano essere utilizzati con molti tipi di dati diversi. In Java, si può raggiungere l'obiettivo della programmazione generica usando l'ereditarietà oppure con variabili di tipo (variabili a cui si può assegnare un tipo non primitivo e che possono essere usate come tipi nelle dichiarazioni). Le *variabili di tipo* rendono più sicuro e di più facile comprensione il codice generico.

Le *variabili di tipo* di una classe generica:

- sono dichiarate tra parentesi angolari dopo il nome della classe;
- sono indicate con una lettera maiuscola;
- sono utilizzate per dichiarare le variabili, i parametri dei metodi e il valore di restituzione nel codice della classe

Esempio: `public class ArrayList<E> { }.` In questo caso, E è una variabile di tipo.

## 10.2 Realizzare tipi generici (classe Pair)

La classe **Pair** memorizza coppie di oggetti, ciascuno dei quali può essere di tipo qualsiasi. Questa classe può essere utile quando si realizza un metodo che calcola e deve restituire due valori.

Ad esempio:

```
Pair<String, Integer> result = new Pair<String, Integer>("Harry Morgan", 1729);
```

La classe generica **Pair** richiede due **tipi parametrici**, uno per il tipo del primo elemento e uno per il tipo del secondo elemento. Solitamente per le variabili di tipo si usano nomi brevi e composti di sole lettere maiuscole:

Variabili di tipo	Significato
E	Tipo di un elemento in una raccolta
K	Tipo di una chiave in una mappa
V	Tipo di un valore in una mappa
T	Tipo generico
S, U	Ulteriori tipi generici

## 10.3 Metodi generici

Un **metodo generico** è un metodo che ha un *tipo parametrico*. Per utilizzare un **metodo generico**, non occorre specificare il tipo effettivo da assegnare alle variabili di tipo. Il tipo del parametro è dedotto dal compilatore dall'uso che ne facciamo. Si possono definire *metodi generici* sia in una classe normale (non generica) che in una classe generica.

## 10.4 Tipi con carattere jolly (wildcard)

Spesso si ha la necessità di formulare vincoli un po' complessi per i tipi parametrici: per questo scopo sono stati inventati i tipi con carattere **jolly** (**wildcard**), che può essere usato in tre diversi modi.

Nome	Sintassi	Significato
Vincolo con limite inferiore	? extends B	Qualsiasi sottotipo di B
Vincolo con limite superiore	? super B	Qualsiasi supertipo di B
Nessun vincolo	?	Qualsiasi tipo

## 10.5 Cancellazione dei tipi (type erasure)

Dato che i tipi generici sono stati introdotti nel linguaggio Java soltanto di recente, la macchina virtuale che esegue programmi Java non lavora con classi o metodi generici: i tipi parametrici vengono “cancellati”, cioè sostituiti da tipi Java ordinari. Ciascun tipo parametrico è sostituito dal relativo vincolo, oppure da Object se non è vincolato. I tipi generici nel codice vengono rimpiazzati con il tipo grezzo corrispondente. Non si possono costruire oggetti o array di un tipo generico.

## 10.6 Decompilazione bytecode

Esistono dei tool per risalire dal *bytecode* al codice sorgente. Si possono verificare gli effetti della type erasure seguendo questi passi:

- scrivere un sorgente con *generics*;
- generare il *bytecode*;
- decompilare.



# Gestione delle eccezioni

## 11.1 Condizioni di errore

Una condizione di errore in un programma può essere causata da:

- **Errori di programmazione:** divisione per zero, cast non permesso, accesso oltre i limiti di un array, etc...;
- **Errori di sistema:** disco rotto, connessione remota chiusa, memoria non disponibile, etc...;
- **Errori di utilizzo:** input non corretti, tentativo di lavorare su file inesistente, etc..;

In JAVA la gestione degli errori può essere fatta usando il meccanismo delle **eccezioni**, che sono oggetti che possono essere creati e lanciati (**throw**) in determinate condizioni, e che possono essere catturati (**catch**) dal codice scritto appositamente per la loro gestione. Le **eccezioni** non devono essere trascurate e devono poter essere gestite da uno gestore competente, non semplicemente dal chiamante del metodo che fallisce.

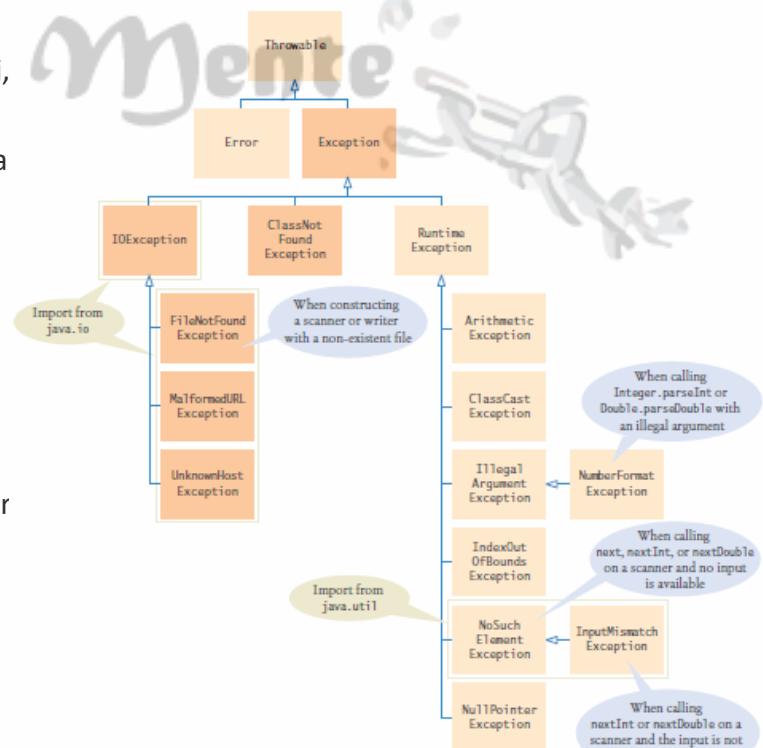
Formalmente, un'eccezione è una violazione di vincoli semantici o di risorsa. Una eccezione è un evento che interrompe la normale esecuzione del programma e se si verifica, il metodo trasferisce il controllo ad un **gestore delle eccezioni** che ha il compito di uscire dal frammento di codice che ha generato l'eccezione e decidere cosa fare.

## 11.2 Superclasse “Throwable”

Java ha una gerarchia di classi per rappresentare le varie tipologie di errore, dislocate in package diversi a seconda del tipo di errore. La superclasse di tutti gli errori è la classe **Throwable** nel package

`java.lang`. La superclasse `Throwable` ha due sottoclassi dirette:

1. **Error:** in genere i programmi non gestiscono questi errori. Questa classe tratta errori fatali, dovuti a condizioni accidentali come l'esaurimento delle risorse di sistema necessa alla JVM (**OutOfMemoryError**), incompatibilità di versioni, violazione di un'asserzione (**AssertionError**), e altri.



2. **Exception:** riguarda tutti gli errori che non rientrano in `Error`. I programmi possono gestir o no questi errori a seconda dei casi. Tutte le classi che rappresentano eccezioni sono sottoclassi della classe `Exception`.

## 11.3 Categoria di Eccezioni

Le **eccezioni non controllate** sono dovute a circostanze che il programmatore **può evitare**, correggendo il programma.

Un esempio è l'eccezione **EOFException** che termina inaspettatamente il flusso dei dati in ingresso.

Può essere provocata da eventi esterni come errore del disco, interruzione del collegamento di rete. Tale problema se ne occupa il gestore dell'eccezione.

Le **eccezioni controllate** sono dovute a circostanze esterne che il programmatore **non può evitare** e il compilatore vuole sapere cosa fare nel caso si verifichi l'eccezione. Un esempio è:

- **NullPointerException**: uso di un riferimento null;
- **IndexOutOfBoundsException**: accesso ad elementi esterni ai limiti di un array.

Non è obbligatorio scrivere un codice per gestire questo tipo di eccezione ma il programmatore può prevenire queste anomalie, correggendo il codice.

## 11.4 Lanciare un'eccezione (throw)

Per lanciare un'eccezione, usiamo la parola chiave **throw** (lancia), seguita da un oggetto di tipo eccezione: **throw exceptionObject;**

La parola chiave **throws** viene usata anche per segnalare le **eccezioni controllate** che il metodo può lanciare.

Un metodo può:

- **gestire l'eccezione**, cioè dire al compilatore cosa fare;
- **non gestire l'eccezione** ma dichiarare di poterla lanciare: in questo caso, se l'eccezione viene lanciata, il programma termina visualizzando un messaggio di errore.

Un metodo può lanciare più *eccezioni controllate*, di tipo diverso, ad esempio:

```
public void calcola(int i)
    throws CloneNotSupportedException,
           ClassNotFoundException
```

## 11.5 Catturare le eccezioni (try – catch)

Per gestire le eccezioni si può usare l'enunciato **try**, seguito da tante clausole **catch** quante sono le eccezioni da gestire. Il procedimento segue questo schema:

- Vengono eseguite le istruzioni all'interno del blocco **try**;
- Se nessuna eccezione viene lanciata, le clausole **catch** sono ignorate;
- Se viene lanciata un'eccezione viene eseguita la corrispondente clausola **catch**;

```
try
{
    istruzione
    istruzione
    ...
}
catch (ClasseEccezione oggettoEccezione)
{
    istruzione
    istruzione
    ...
}
catch (ClasseEccezione oggettoEccezione)
{
    istruzione
    istruzione
    ...
}
...
...
```

### 11.5.1 PrintStackTrace

Per avere un messaggio di errore che stampa lo **stack** delle chiamate ai metodi in cui si è verificata l'eccezione usiamo il metodo **printStackTrace()**.

### 11.5.2 Clausola finally

Sappiamo che il lancio di un'eccezione arresta il metodo corrente però a volte vogliamo eseguire altre istruzioni prima dell'arresto. A tale scopo, usiamo la clausola **finally** per indicare che un'istruzione deve essere eseguita sempre (ad esempio, se stiamo leggendo un file e si verifica un'eccezione, vogliamo comunque chiudere il file). E' preferibile evitare di mischiare catch e finally nello stesso blocco try.

Tale clausola va eseguita quando si esce da un blocco try:

```
try{  
    istruzioni...  
}  
finally{  
    istruzioni...  
}
```

## 11.6 Creare nuove eccezioni

Se nessuna delle eccezioni ci sembra adeguata al nostro caso, possiamo progettarne una nuova. I nuovi tipi di eccezioni devono essere inseriti nella discendenza di **Throwable**, e in genere sono sottoclassi di **RuntimeException**.

Un'eccezione che sia sottoclasse di **RuntimeException** sarà a controllo non obbligatorio.

Un'eccezione di tipo controllato deve essere sottoclasse di **Exception**.

## 11.7 La gestione delle eccezioni (handler)

Per la gestione delle eccezioni, il compilatore Java effettua due controlli:

1. Per ogni eccezione lanciabile dal codice deve esistere un **handler** menzionante la classe (o una superclasse) di tale eccezione.
2. Non devono esistere **handler** per eccezioni non lanciabili dal codice.

Tali controlli non si applicano alle **eccezioni non controllate** (da qui il nome), poiché quasi ogni istruzione può provocare un'eccezione e, secondo i progettisti di Java, il doverle gestire con dei try-catch sarebbe stato di grande fastidio per i programmatore.

Al lancio di un'eccezione, il controllo viene trasferito dal codice anomalo alla clausola **catch** che gestisce l'eccezione. Il trasferimento di controllo causa la brutale terminazione di espressioni o istruzioni: l'esecuzione continua quindi nel blocco del **catch**, ed il codice che causa l'eccezione non potrà più continuare la sua esecuzione.

La determinazione della giusta clausola **catch** è effettuata confrontando la classe dell'oggetto lanciato col tipo dichiarato come parametro del **catch**.

Il **matching** si ha se il tipo dei parametri del **catch** è la classe (o una superclasse) dell'eccezione. In presenza di più *match* viene selezionata la prima clausola.

# I/O e stream

## 12.1 Formato binario e formato di testo

I dati sono memorizzati nei files in due formati:

1. **testo** (successione di caratteri);
2. **binario** (successione di bytes).

In Java *input* e *output* sono definiti tramite i **flussi (stream)**: sequenze ordinate di dati. Mediante i flussi un oggetto di una qualsiasi sorgente di informazione è capace di produrre o ricevere dati.

Abbiamo due tipi di stream:

1. flussi di dati binari (byte stream);
2. flussi di caratteri (character stream).

Ciascun tipo di flusso è gestito da apposite classi:

- per i dati binari, viene usata la classe **InputStream** e la classe **OutputStream**;
- per i caratteri, usiamo la classe **Reader** e la classe **Writer**.

Tutte queste classi sono inserite nel package **java.io** (`import java.io.*;`). Tutti gli stream sono automaticamente aperti quando vengono creati. Ogni stream dovrebbe essere chiuso esplicitamente quando non se ne fa più uso, tramite il metodo **close()**. Il garbage collector, infatti, non può chiudere uno stream se è ancora aperto e quindi non potrà di allocare la memoria. Una buona tecnica per chiudere lo stream è nella clausola `finally` del blocco `try-catch`; è possibile usare anche il `try-with-resources`.

## 12.2 Condizione di errore in `java.io`

La classe **IOException** viene utilizzata da molti metodi del package `java.io` per segnalare condizioni di errore legati al flusso di I/O.

I costruttori che ricevono come parametro il nome di un *file/directory* o un oggetto `File` possono lanciare una **FileNotFoundException** (sottoclasse di `IOException`).

## 12.3 Flussi Standard

I flussi standard sono definiti dalla classe **System** in `java.lang` e sono tutte variabili statiche:

- Standard input (tastiera): `System.in` [Di tipo `InputStream`]
- Standard output (monitor): `System.out` [Di tipo `PrintStream`].
- Standard error (per messaggi di errore): `System.err` [Di tipo `PrintStream`]

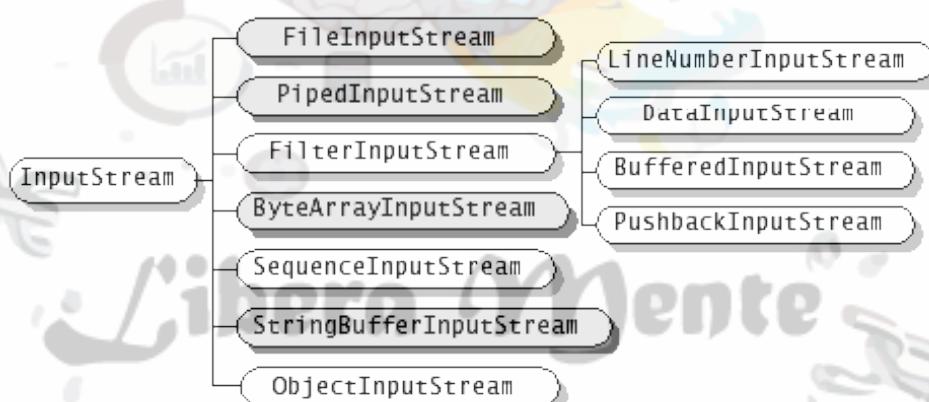
## 12.4 Classe astratta `InputStream`

La classe astratta **InputStream** dichiara metodi per leggere flussi binari da una sorgente specifica. Tra i metodi più importanti abbiamo:

- **read()** [*metodo astratto*]: legge un byte alla volta e restituisce un `int` (da 0 a 255) che rappresenta il byte letto oppure -1 se il file è terminato.
- **close()**: chiude il flusso di input, rilascia le risorse associate al flusso e se vengono effettuate operazioni sul flusso chiuso viene provocata una `IOException`.

Tra le sottoclassi di `InputStream` abbiamo:

1. **FileInputStream**: realizzato per la lettura di flussi di byte da un file in filesystem. Il metodo `read()` è invocato su un oggetto di tipo `InputStream` e legge byte fino a quando l'input è a disposizione: ritorna 1 se ha letto il prossimo byte, -1 se ha letto la fine del file.
2. **FilterInputStream**: Sovrascrive metodi di `InputStream` oppure ne aggiunge alcune funzionalità, eventualmente trasformando i dati lungo il percorso.
  - a. **DataInputStream**: estende `FilterInputStream` e consente di leggere da un oggetto di tipo `InputStream` i tipi primitivi di Java: int, double, String, etc..
  - b. **BufferedInputStream**: estende `FilterInputStream` in modo tale da rendere la lettura bufferrizzata.
3. **BufferInputStream**: prende in ingresso un array di byte lo uso come buffer. Un contatore interno tiene traccia del prossimo byte da fornire al metodo `read()`.
4. **ObjectInputStream**: si possono leggere interi oggetti, liste, da qualsiasi stream, evitando cicli inutili in cui andare a scrivere un singolo dato primitivo alla volta. Per fare queste operazioni bisogna **serializzare** gli oggetti che devo leggere, ciò significa che questi oggetti devono estendere **Serializable**.



### 12.4.1 Classe astratta `OutputStream`

La classe astratta `OutputStream` dichiara metodi per scrivere flussi binari in una destinazione specifica.

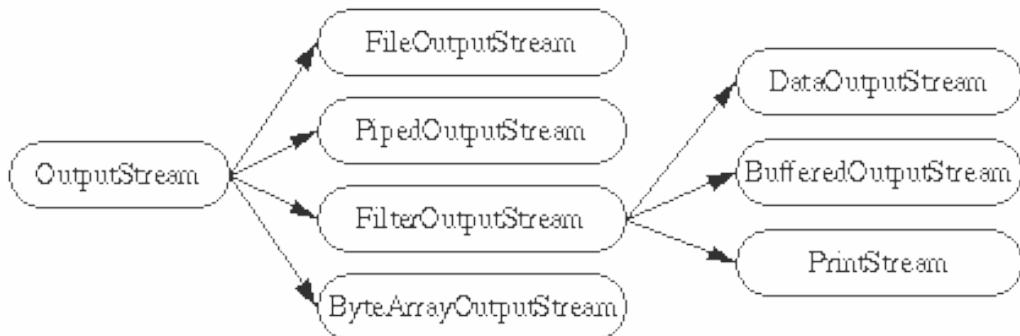
Tra i metodi più importanti abbiamo:

- `write(int b)` [*metodo astratto*]: scrive un byte alla volta e il byte è passato come argomento di tipo `int`.
- `close()`: chiude il flusso di output, rilascia le risorse associate al flusso e se vengono effettuate operazioni sul flusso chiuso viene provocata una `IOException`.

Alcune sottoclassi di `OutputStream` sono:

1. **PrintStream** è una classe concreta nella discendenza ovvero è una sottoclasse di `OutputStream`. Aggiunge a `OutputStream` tutti i metodi per stampare vari tipi di dati (come i metodi `print` e `println`). L'oggetto `System.out` è di tipo `PrintStream` e rappresenta il flusso standard di output (flusso binario).

2. **FileOutputStream** è anch'essa una sottoclasse di **OutputStream** utilizzata per scrivere dati, orientati ai byte e ai caratteri, in un file. Tuttavia, per i dati orientati ai caratteri, è preferibile utilizzare **FileWriter** rispetto a **FileOutputStream**.



## 12.4.2 Classe astratta Reader

La classe astratta **Reader** dichiara i metodi per leggere flussi di caratteri da una sorgente specifica.

Tra i metodi astratti più importanti citiamo:

- **read()** [*metodo astratto*]: legge un carattere alla volta e restituisce un **int** (da 0 a 65535) che rappresenta il carattere letto oppure -1 se il file è terminato.
- **close()**: chiude il flusso di caratteri, rilascia le risorse associate al flusso e se vengono effettuate operazioni sul flusso chiuso viene provocata una **IOException**.

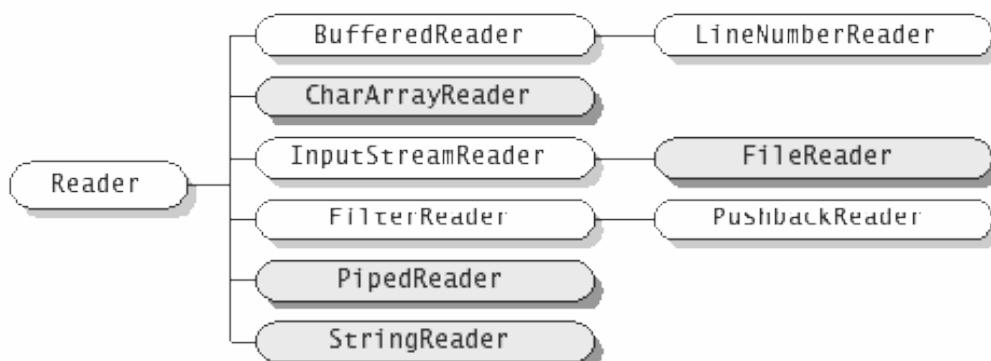
Alcune sottoclassi di Reader sono:

1. **InputStreamReader**: è una sottoclasse concreta di **Reader** usata per convertire un flusso di input binario in un flusso di input di caratteri. Viene utilizzata per leggere da **System.in** per leggere molti caratteri e avere una maggiore efficienza. I flussi si possono convertire in questo modo:

```
InputStream in = new FileInputStream("nomefile.bin");
InputStreamReader reader = new InputStreamReader(in);
```

- a. **FileReader**: è una sottoclasse di **InputStreamReader** il cui costruttore richiede il nome del file e tale classe serve per leggere flussi di carattere da un file, quindi file di testo. Gli oggetti della classe **FileReader** leggono un carattere per volta, invece, per leggere intere linee si può usare un oggetto attraverso l'uso della classe **Scanner**:
 

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
String inputLine = in.nextLine();
```



### 12.4.3 Classe astratta Writer

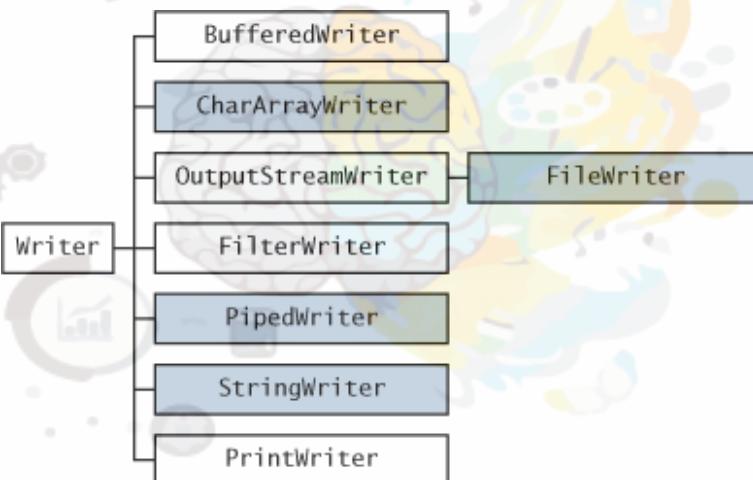
La classe astratta **Writer** dichiara i metodi per scrivere flussi di caratteri da una sorgente specifica.

Alcuni metodi:

- **write(int c)**: scrive un carattere alla volta passato come argomento di tipo int.
- **close()**: chiude il flusso di caratteri, rilascia le risorse associate al flusso e se vengono effettuate operazioni sul flusso chiuso viene provocata una IOException.

Alcune sottoclassi di Writer sono:

1. **Printwriter** è una sottoclasse concreta di Writer e contiene tutti i metodi print e println di PrintStream. Quando si istanzia un oggetto PrintWriter:
  - se il file passato come parametro nel costruttore esiste, allora contenuto del file viene svuotato;
  - se il file non esiste viene creato un file nuovo (vuoto) con il nome passato come parametro nel costruttore.



### 12.4.4 Classe File

La classe **File** può essere utilizzata per manipolare file esistenti. Per leggere/scrivere dati in un oggetto di tipo File dobbiamo costruire un oggetto di tipo FileReader o PrintWriter. Alcuni metodi della classe File sono:

- **public boolean delete()** : cancella il file restituendo true se la cancellazione ha successo;
- **public boolean renameTo(File newname)** : rinomina il file restituendo true se la ridenominazione ha successo;
- **public long length()** : restituisce la lunghezza del file in byte (0 se il file non esiste);
- **public boolean exists()** : testa se il file o la directory denotata da questo File esiste.

## 12.5 Leggere pagine web

Esiste una classe in Java per gestire gli indirizzi delle pagine Web che si chiama **URL** e fornisce un metodo **openStream** che restituisce un oggetto **InputStream**:

```
URL locator = new URL("http://bigjava.com/index.html");
InputStream in = locator.openStream();

// Se si vuole leggere caratteri possiamo:
Scanner in = new
Scanner(locator.openStream());
```

## 12.6 Accesso sequenziale e casuale

Con l'**accesso sequenziale** un file viene elaborato un byte alla volta in sequenza e questa operazione può risultare inefficiente.

Con l'**accesso casuale** possiamo accedere a posizioni arbitrarie nel file e soltanto i file su disco supportano l'accesso casuale: `System.in` e `System.out` non lo supportano. Ogni file su disco ha un puntatore di file che individua la posizione dove leggere o scrivere.

Per l'accesso casuale al file, usiamo un oggetto di tipo `RandomAccessFile`. Possiamo aprire il file in diverse modalità:

1. "r" apre il file in sola lettura; se viene usato un metodo di scrittura viene invocata una `IOException`;
2. "rw" apre il file per lettura e scrittura. Se il file non esiste prova a crearlo.

I metodi dell'accesso casuale sono:

- `read()`: come `read` di `InputStream`, estrae un byte alla volta;
- `write(b)`: scrive il byte `b` a partire dalla posizione indicata dal puntatore;
- `close()`: chiude il file;
- `seek(n)`: sposta il puntatore al byte di indice `n`;
- `int n = f.getFilePointer()`: fornisce la posizione corrente del puntatore nel file;
- `long fileLength = f.length()`: fornisce il numero di byte di un file.

Per aggiornare un file ogni valore deve avere uno spazio fissato sufficientemente grande e ogni record ha la stessa taglia, in questo modo è più facile individuare ogni record. Inoltre, quando tutti i record hanno la stessa taglia è più facile memorizzare i numeri in binario.

## 12.7 Serializzazione

Con "serializzare un oggetto" si intende il processo che rende persistente un oggetto Java, ovvero far sopravvivere l'oggetto oltre la chiusura del programma. Ciò significa salvare lo stato dell'oggetto all'interno di un file. Con la *serializzazione* ogni oggetto riceve un numero di serie nel flusso; se lo stesso oggetto viene salvato due volte la seconda volta salviamo solo il numero di serie. I numeri di serie ripetuti sono interpretati come riferimenti allo stesso oggetto.

È possibile *serializzare* un oggetto a patto che si implementi l'interfaccia `Serializable` (non contiene metodi ma serve a distinguere cos'è *serializzabile* e cosa no). Oggetti non *serializzabili* sono ad esempio i `thread` e tutte le classi di tipo `Stream`. Non vengono serializzate né le variabili statiche né le variabili d'istanza dichiarate `transient`.

Se viene serializzato un `thread` viene sollevata l'eccezione `NotSerializableException`.

Poiché la **deserializzazione** potrebbe essere fatta anche da un'altra classe che si trova su un'altra macchina, bisogna verificare che le classi usate da chi ha serializzato l'oggetto e chi lo sta deserializzando siano compatibili. La costante `SerialVersionUID` associa alla classe un identificativo univoco di tipo `Long` ed è fortemente consigliato definire l'ID e non farlo generare automaticamente.

# Programmazione grafica - GUI

## 13.1 Applicazione grafica

Uno dei problemi maggiori emersi durante la progettazione di Java fu la realizzazione di un toolkit grafico capace di funzionare su piattaforme diverse tra di loro. La prima soluzione fu **AWT**, un package grafico limitato, poiché i programmi grafici AWT assumono un aspetto e un comportamento diverso a seconda della JVM su cui vengono eseguiti. Venne successivamente introdotto il package **Swing** che si distingue molto da AWT in quanto è indipendente dalla piattaforma. *Swing* è un'estensione di AWT, infatti ogni interfaccia grafica *Swing* è basata su un componente AWT.

Un'**applicazione grafica** visualizza informazioni all'interno di una finestra dotata di barra di titolo e cornice (**frame**). La JVM esegue ogni frame su un **thread** separato. Il **thread** è un flusso di esecuzione.

Per visualizzare qualcosa all'interno di un frame occorre definire un oggetto di tipo componente e aggiungerlo al frame ed estenderlo con la classe **JComponent**. Per disegnare le forme abbiamo vari metodi:

1. **paintComponent**: chiamato ogni volta che una componente necessita di essere ridisegnata e le istruzioni di disegno sono inserite in questo metodo;
2. **Graphics**: è una classe astratta e ci consente di manipolare lo stato grafico (ad es. colore);
3. **Graphics2D**: è una classe astratta, estende **Graphics** e ha metodi per tracciare forme grafiche;

Il cast a **Graphics2D** del parametro **Graphics** serve per usare il metodo **draw** e **Graphics** e **Graphics2D** si trovano nel package **java.awt**.

I colori standard **Color.BLUE**, **Color.RED**, **Color.PINK** e altri sono costanti. Per ottenere altri colori si combinano rosso, verde e blu dando, per ognuno, dei valori compresi tra **0.0F** e **1.0F**.

Un'applicazione grafica può ricevere testo in input lanciando un oggetto **JOptionPane**. Il metodo **showInputDialog** visualizza un prompt e attende l'input dall'utente restituendo la stringa digitata dall'utente.

## 13.2 Gestione degli Eventi

Le GUI vengono gestite attraverso la programmazione basata su **eventi**. Infatti, ogni volta che l'utente esegue un'azione come un clic del mouse, la modifica di una finestra o altro, viene generato un **evento**. Esistono diversi oggetti per gestire gli eventi su una GUI:

1. **Ricevitore dell'evento (listener)**: riceve una notifica quando un accade un evento. I suoi metodi descrivono le azioni da eseguire quando si verificano gli eventi. Un programma sceglie gli eventi da trattare installando un ricevitore per ciascuno di essi.  
Per aggiungere un *listener* ad un componente bisogna utilizzare il metodo **addActionListener()** che prende come parametro un **ActionListener**. Nel momento in cui viene istanziato un oggetto **ActionListener** bisogna definire il metodo **actionPerformed()** che prende in ingresso un **ActionEvent**, che è proprio l'evento che il componente manda al *listener*. Il *listener* poi, attraverso **actionPerformed**, eseguirà il codice relativo a quell'evento.
2. **Sorgente dell'evento (source)**: è la componente (dell'interfaccia utente) che ha generato l'evento. Quando capita un evento, la sorgente notifica tutti i ricevitori dell'evento.

Con il paradigma a eventi, l'applicazione è “reattiva”, infatti, il programma principale si limita a inizializzare l'applicazione, istanziando gli osservatori e associandovi gli opportuni **handler**.

Il package **java.awt.event** contiene le classi per i diversi tipi di eventi, le interfacce relative ai ricevitori di eventi (**Listener**) e le classi degli **Adapter** che implementano le interfacce **Listener**.

### 13.3 JPanel e JTextField

**JPanel** è un contenitore estende **JComponent** e serve quando vogliamo aggiungere più componenti ad un frame: aggiungere le singole componenti al frame le sovrapporrebbe. L'ordine in cui vengono aggiunte le componenti rispecchia l'ordine di visualizzazione nel frame.

Per elaborare un testo in input si usano componenti **JTextField**: una sottoclasse di **JComponent** ed è buona norma usare un **JLabel** per descrivere un **JTextField**. Inoltre, bisogna predisporre un pulsante per permettere all'utente di segnalare quando l'input è pronto per l'elaborazione.

### 13.4 Adattatori

Se si vuole scrivere una classe che gestisce un tipo di evento AWT, dobbiamo farle implementare l'interfaccia dell'opportuno **Listener**, quindi dobbiamo scrivere una classe che implementi TUTTI i metodi dell'interfaccia. In alcuni casi abbiamo a disposizione delle classi astratte (**adattatori**) che implementano già tutti i metodi:

- **FocusAdapter**;
- **KeyAdapter**;
- **MouseAdapter**;
- e altri...

Invece di scrivere una classe ed implementare tutti i metodi dell'interfaccia possiamo ereditare i metodi da un adattatore (estendendo la classe) e poi sovrascrivere solo quello che ci interessa. Il problema è che in java non c'è ereditarietà multipla, quindi se una classe vuole gestire diversi tipi di eventi non può ereditare i metodi da diversi adattatori. In quel caso occorre implementare le interfacce.

### 13.5 JFrame

Per usare al meglio i frame e rendere il codice semplice e leggibile si può utilizzare l'ereditarietà (scomponendo i frames complessi in unità comprensibili), progettare sottoclassi di **JFrame**, memorizzare le componenti come variabili di istanza e inizializzare le variabili nei costruttori delle sottoclassi (se l'inizializzazione diventa complessa utilizziamo alcuni metodi di servizio).

La classe **JFrame** in sé contiene solo alcuni metodi per modificare l'aspetto dei frame. Quasi tutti i metodi per lavorare con le dimensioni e con la posizione di un frame provengono dalle varie superclassi di **JFrame** mentre le classi **Component** e **Window** contengono i metodi per modificare le dimensioni e la forma dei frame.

Le componenti di un'interfaccia utente sono organizzate mettendole all'interno di un contenitore (ad esempio **JPanel**). Ogni contenitore ha un **layout manager** (gestore di layout) che si occupa del posizionamento delle sue componenti. Le componenti in un **JPanel** sono inserite da sinistra a destra.

Tre gestori di layout più diffusi in `java.awt` abbiamo:

1. **gestore di layout a bordi (`BorderLayout`)**: è il layout manager di default di un frame ed è diviso in 5 aree center, north, west, south e east. Quando si aggiunge una componente si specifica una posizione (CENTER di default), inoltre, ogni componente è estesa per coprire l'intera area allocata; se si ha l'esigenza di condividere l'area con altri componenti, si possono inserire i componenti in un JPanel;
2. **gestore di layout a scorrimento (`FlowLayout`)**;
3. **gestore di layout a griglia (`GridLayout`)**: posiziona le componenti in una griglia con un numero fissato di righe e colonne. La taglia di ogni componente viene modificata in modo che tutte le componenti hanno la stessa taglia. Ogni componente viene espansa in modo che occupi tutta l'area allocata;
4. **gestore di layout a griglia (`GridBagLayout`)**: possiede le componenti disposte in una tabella, le colonne possono avere taglie differenti e le componenti possono ricoprire colonne multiple. Lo svantaggio di questo layout è che è difficile da usare. Per superare questo ostacolo possiamo usare JPanel annidati.

Per default, JPanel organizza le componenti da sinistra a destra e comincia una nuova riga se necessario.

### 13.5.1 Pulsanti radio

In un insieme di **pulsanti radio** uno solo alla volta può essere selezionato. **Questo sistema è adatto ad un insieme di scelte mutuamente esclusive**. Se un pulsante è selezionato, tutti gli altri nell'insieme sono automaticamente deselezionati. Ogni pulsante è un oggetto di `JRadioButton` nel pacchetto `javax.swing` ed è sottoclasse di `JComponent`. Per verificare se un pulsante è selezionato o meno si usa il metodo `isSelected()` che restituisce `true` o `false`.

### 13.5.2 Bordi

Per default i pannelli non hanno bordi visibili, per aggiungerne uno si usa `EtchedBorder`: un bordo con effetto tridimensionale. Si può aggiungere un bordo a ogni componente:

```
JPanel panel = new JPanel();
panel.setBorder(new EtchedBorder());
Per inserire un bordo con titolo si usa TitledBorder:
panel.setBorder(new TitledBorder(new EtchedBorder(),"Size"));
```

### 13.5.3 Caselle di controllo

Ogni casella ha due stati: selezionata e non selezionata. **Nei gruppi di caselle di controllo la scelta non è mutuamente esclusiva**. Per le caselle di controllo si usa `JCheckBox`, nel pacchetto `javax.swing` ed è una sottoclasse di `JComponent`.

Esempio: `JCheckBox italicCheckBox = new JCheckBox("Italic");`

### 13.5.4 Caselle combinate

Le **caselle combinate** vengono utilizzate per grandi insiemi di scelte mutuamente esclusive. Sono una combinazione di una lista e un campo di testo che visualizza il nome della selezione corrente.

Questa categoria usa meno spazio dei pulsanti radio. La casella combinata può essere editabile se `setEditable(true)`. Le stringhe di testo si aggiungono con il metodo `addItem`. La selezione dell'utente si prende con `getSelectedItem` e restituisce un Object.

In conclusione, diciamo che pulsanti radio, caselle di controllo e caselle combinate di un frame generano un **ActionEvent** ogni volta che l'utente seleziona un elemento.



Caselle combinate

Caselle di controllo

Pulsanti radio

### 13.5.5 Menu

Ogni frame contiene una barra dei menu. La barra contiene dei menu che ha sua volta può avere dei sub-menu e item del menu. Gli item e i sub-menu si aggiungono con il metodo `add`. Un item non ha ulteriori sub-menu e gli item generano eventi del tipo **ActionEvent**.

### 13.5.6 Area di testo

Per le aree di testo si usa **JTextArea** il quale mostra linee di testo multiple. I metodi:

- `setText`: imposta il testo di un campo o un'area di testo;
- `append`: aggiunge testo alla fine di un'area di testo.

# Threads e Programmazione Multithreading

## 14.1 Cosa è un thread

Un **thread** è un **percorso di controllo** all'interno di un processo. I **thread** sono anche detti **processi leggeri** perché hanno un contesto più semplice. Il thread è shared memory.

Un *thread* comprende:

- **TID:** identificare univoco del thread;
- **Contatore di programma;**
- **Insieme di registri;**
- **Stack.**

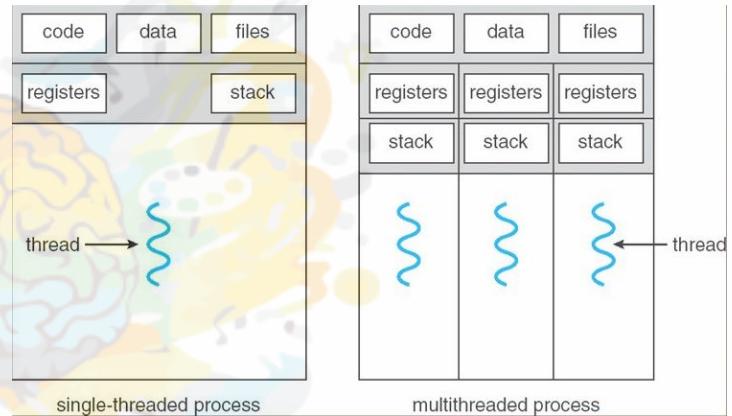
Inoltre, un *thread* condivide con gli altri *thread* che appartengono allo stesso processo:

- la **sezione del codice**;
- la **sezione dei dati**;
- **altre risorse allocate dal processo originale** (file aperti e segnali).

Molti SO moderni permettono che è un processo possa avere più percorsi di controllo, ovvero possedere dei **multi-thread**.

Un processo **multi-thread** è in grado di lavorare a più compiti in modo concorrente.

I vantaggi della programmazione **multi-thread** aumentano nelle architetture multiprocessore, dove i *thread* possono essere eseguiti in parallelo, inoltre l'impiego della programmazione **multithread** in un sistema con più CPU fa aumentare il grado di parallelismo



I *thread* che richiedono una risorsa impegnata vengono messi in uno stato di attesa fin quando la risorsa non viene rilasciata dal *thread* che la possedeva. Questo meccanismo è gestito attraverso l'uso di **semafori**: l'accesso alla risorsa è in **mutua esclusione**, ovvero se un processo è in esecuzione nella sua **sezione critica**, nessun altro processo può eseguire la propria **sezione critica** (la **sezione critica** è un segmento di codice in cui il processo può modificare i suoi valori).

Una soluzione messa a disposizione dal SO è il **mutex**: prima di utilizzare una risorsa, il *thread* acquista il **mutex**, che va a bloccare lo stato della risorsa. Gli altri *thread*, quindi, entrano in uno stato di attesa: ciò provoca un rallentamento di esecuzione, per questo bisogna utilizzare i **mutex** il meno possibile, ed è per questo motivo che si evita il più possibile l'uso di variabili globali.

Situazioni che possono verificarsi sono i cosiddetti **deadlock**: quando uno o più processi attendono indefinitamente un evento che può essere causato da altri processi in attesa. JAVA non ha alcun meccanismo per scoprire un **deadlock**, infatti è compito del programmatore evitare queste situazioni, ad esempio facendo rispettare lo stesso ordine di acquisizione delle risorse da parte di tutti i *thread*.

## 14.2 Thread in Java

In Java i *thread* sono degli oggetti utilizzati per far eseguire un'applicazione parallelamente sulla stessa JVM, allo scopo di ottimizzare i tempi di esecuzione. Quando viene creato un nuovo *thread* per eseguirlo bisogna richiamare il metodo `start()`. I *thread* della JVM sono associati ad istanze della classe `java.lang.Thread`. In Java ogni programma in esecuzione è un *thread*; il metodo `main()` è associato al **main thread** e per poter accedere alle proprietà del **main thread** è necessario ottenerne un riferimento tramite il metodo `currentThread()`.

## 14.2.1 Creare thread in Java

Per creare thread in Java possiamo utilizzare due metodi:

1. creando una classe derivata che estende Thread. Questo metodo è più semplice ma non è possibile ereditare da altre classi. Il thread viene creato dalla JVM non appena si richiama il metodo `start()` e poi il nuovo thread esegue il metodo `run()`.
2. Creando una classe che implementa l'interfaccia Runnable: nella quale viene ridefinito il metodo `run()`; poi l'oggetto creato viene passato al costruttore del Thread e infine può essere invocato il metodo `start()` sul thread creato. Con questo metodo la classe che implementa Runnable rimane libera di ereditare da un'altra classe già definita.

I thread terminano quando finisce l'esecuzione del metodo `run()` oppure sono stati interrotti con il metodo `interrupt()` o con eccezioni ed errori.

La JVM definisce uno scheduler che si occuperà di decidere quale thread deve essere eseguito; a sua volta la JVM è un software gestito dal SO, quindi lo scheduler della JVM rispetterà le priorità del SO.

## 14.3 Sincronizzazione

Con la **sincronizzazione** si accede in modo concorrente ai dati comuni in maniera sicura: bloccando un oggetto. L'istruzione **synchronized** blocca un oggetto durante l'esecuzione di una istruzione. Non è detto che l'oggetto bloccato sia poi effettivamente usato all'interno della istruzione sincronizzata.

Per usare in maniera consistente in un'applicazione **multithreaded** una classe che non è stata progettata come `synchronized`, possiamo seguire due strade:

1. creare una sottoclasse ridefinendo ogni metodo come segue:

```
synchronized metodo (parametri) {  
    super(parametri);  
}
```

2. usare i metodi della classe non `synchronized` in istruzioni `synchronized`.

## 14.4 Comunicare fra i thread: wait e notify

`wait()` e `notify()` sono metodi della classe Object che possono essere chiamati solamente dall'interno di codice sincronizzato:

- `wait()`: rilascia il blocco sull'oggetto e arresta il thread mettendolo in attesa;
- `notify()`: sveglia il thread (ovvero, uno dei thread) in attesa sullo stesso oggetto;
- `notifyAll()`: sveglia tutti i thread in attesa sullo stesso oggetto.

## 14.5 Lock

In alternativa all'uso di `synchronized` si possono usare gli oggetti **lock**. Un oggetto `lock` è usato per controllare thread che manipolano risorse condivise. In Java: `Lock` è un'interfaccia ed esistono diverse classi che la implementano:

- `ReentrantLock`: è la classe più utilizzata.

Il codice che manipola risorse condivise è racchiuso tra invocazioni di `lock` e `unlock`. Inoltre, se tra una chiamata a `lock` e `unlock` viene lanciata un'eccezione, la chiamata ad `unlock` non viene mai fatta. Per risolvere questo problema, occorre mettere l'operazione `unlock` nella clausola `finally`.

# Lambda Expressions

## 15.1 Programmazione funzionale

Rispetto al paradigma di **programmazione imperativo**, in cui si pone l'interesse sulla sequenza di comandi da eseguire, nella **programmazione funzionale** il programma assume la forma di una sequenza di operazioni matematiche mentre l'esecuzione corrisponde alla loro valutazione. Un aspetto fondamentale del *paradigma funzionale* è che richiede allo sviluppatore di dichiarare ciò che si vuole ottenere piuttosto che descrivere come ottenerlo. Tale paradigma, inoltre, non prevede **side-effect** in quanto le variabili vengono create esclusivamente al momento dell'esecuzione della funzione e sono distrutte subito dopo.

La programmazione funzionale è stata introdotta in Java 7 nella 8.

Le **espressioni Lambda** in programmazione si riferiscono a *funzioni anonime*, ovvero che non sono associate ad alcun identificatore. La particolarità di queste funzioni è che possono essere definite una volta, assegnate ad un oggetto ed essere riutilizzate ogni volta che se ne ha bisogno senza possibilità che vi sia interazione (*side effect*) tra le varie esecuzioni. In Java era già possibile definire classi e metodi anonimi ma non vi era la possibilità di poterle manipolare.

Linguaggi che supportano esplicitamente la *programmazione funzionale* seguono i seguenti concetti:

- **First-class functions**: le funzioni sono valori come gli altri;
- **Anonymous functions**: funzioni definite e usate immediatamente;
- Variabili non locali e chiusure;
- **Type inference**: il compilatore deduce per noi il tipo di alcune variabili o funzioni, senza farci perdere tempo a scriverlo.

```
for(int i = 0; i < friends.size(); i++) {  
    System.out.println(friends.get(i));  
}
```

- Stile imperativo:
  - Descrive come raggiungere un determinato scopo

```
friends.forEach(  
    String name ->  
    System.out.println(name));
```

- Stile funzionale:
  - Descrive quale scopo vogliamo raggiungere

Quindi applicare (correttamente) lo stile funzionale permette di:

1. Ridurre il gap tra la specifica e la realizzazione;
2. Nascondere gli aspetti di basso livello (Iteratori, conversioni di tipo, assegnamenti e ri-assegnamenti di variabili);
3. Facilitare il test e manutenzione del codice;
4. Sfruttare il parallelismo (N-core CPUs a disposizione ormai ovunque);

Le espressioni lambda hanno un tipo definito da un'**interfaccia funzionale**, ovvero un'interfaccia con un solo metodo astratto. La sintassi di una *lambda expression* è: [lista di parametri -> istruzione](#).

- lista di parametri è la lista di identificatori separati da virgolette racchiusa tra parentesi tonde (es. due parametri: ( x, y )). Le parentesi possono essere omesse se parametro è singolo e se non c'è alcun parametro si usa lista vuota ( );
- l'istruzione può essere un'istruzione semplice, istruzione composta, o blocco di istruzioni.

Si può usare un'*espressione lambda* in Java solo laddove è prevista un'interfaccia funzionale: un'interfaccia con un solo metodo. Il metodo deve essere astratto (ossia non statico e senza implementazione di default).

Un esempio di interfaccia funzionale è Comparator<T> che ha solo un metodo astratto:

`int compare(T o1, T o2).`

Lo stile funzionale risulta particolarmente efficace in ambiti specifici:

- Manipolazione delle Collection;
- Processing di String(s);
- Multithreading;
- Gestione delle risorse (file, sockets, ...).

## 15.2 Streams Java 8 vs Iteratori

Gli **Iteratori** prevedono una visita della collezione impedendo un'efficiente esecuzione concorrente. Gli **streams** invece consentono la parallelizzazione.

`java.util.stream` contiene classi per elaborare sequenze di elementi e la classe principale è `Stream<T>`.

Gli streams possono essere creati da collezioni o array (tramite il metodo `stream()`), generatori o iteratori. Si usa **filter** per selezionare gli elementi desiderati e **map** per trasformare gli elementi, restituendo un nuovo stream.

## 15.3 Multithreading con gli stream

Le API di Stream forniscono il metodo `parallelStream()` che esegue operazioni sugli elementi degli stream in maniera parallela.

Le operazioni con gli stream si dividono in due categorie:

1. **Operazioni intermedie** (restituiscono Stream<T>);
2. **Operazioni terminali** (restituiscono un risultato di un dato tipo).

Le operazioni su stream non cambiano la sorgente.

Per quanto riguarda, invece, le **regole di scope** (visibilità delle variabili) valgono come per le classi interne e anonime:

- variabili dichiarate nell'ambiente esterno sono visibili nell'istruzione che costituisce il corpo della lambda espressione;
- le variabili locali dell'ambiente esterno utilizzate nella espressione lambda devono essere effettivamente dichiarate **final** oppure il loro valore effettivamente non viene modificato.

A differenza delle classi interne e anonime:

- non introduce un nuovo ambiente di scoping;
- non si può dichiarare una variabile con un nome già definito nel metodo in cui viene scritta.

## 15.4 Classi anonime

Sono classi innestate ma senza nome, quindi godono delle stesse proprietà. Sono utilizzate soprattutto per la gestione degli eventi sulle GUI. Una *classe anonima* non ha costruttore, quindi sfrutterà quello della superclasse o superinterfaccia: ciò significa che una classe anonima estende sicuramente un'altra classe/interfaccia. Una classe anonima viene dichiarata con lo scopo di fare *override* di uno o più metodi della classe che estende.