Java Notes

toString()

Restituisce una rappresentazione testuale (stringa) che descrive l'oggetto; utile per stampare.

Dentro a *Object*, il metodo *toString()* é definito in modo da restituire una stringa della forma:

classe@hashcode

```
Persona p = new Persona ( " Mario Bianchi " ," Via Firenze 13 " );
String s = p . toString ();
System . out . println ( s );                // stampa Persona@ 7519ca2c
```

Puó essere ridefinito nelle proprie classi (overriding).

```
public String toString()
```

Array

```
int [] <array_name> = new int[dim]
```

Gli array sono in realtà oggetti che hanno un riferimento e un valore.

for-each

```
for ( <type> <variable> : <in> ) { }
```

Gli elementi sono presi in ordine dall'array e deve essere compatibile con gli elementi presenti nell'array. Consente di utilizzare gl elementi in sola lettura.

E' un for semplificato, simile al for in pyton (o bash) con le parentesi in più e ':' al posto di 'in'.

```
for ( String s : nomi ) {
    System.out.println(s);  //stampa tutti gli elementi in nomi
}
```

Letteralmente: per ogni variabile *i* di tipo *String* appartenente nell'array *nomi*, esegui ciò tra parentesi graffe.

Equivale a:

```
for ( int i=0; i<nomi.length; i++) {
    String s = nomi[i];
    System.out.println(s);
}
```

Scanner class

E' un oggetto che risiede nella libreria *java.util.Scanner* che gestisce l'input dei dati.

```
import java.util.Scanner;
...
Scanner input = new Scanner(System.in);
```

```
int x = new int;
x = input.nextInt();
```

Incapsulamento

E' la proprietà per cui i dati che rappresentano lo stato interni di un oggetto possono essere accessibili solo tramite i moetodi dell'oggetto stesso.

Proprietà di incapsulameno: possibilità di controllare l'accesso allo stato degli oggetti tramite appositi metodi nei linguaggi orientati agli oggetti.

L'incapsulamento consente di **nascondere la rappresentazione dello stato interno degli oggetti agli utilizzatori** (non ai programmatori).

Variabili statiche

E' una variabile condivisa da tutte le istanze di classe di una certa classe. Sono variabili che hanno valore comune a tutte le istanze. Assume valore dell'ultima classe utilizzata.

Metodi statici

Un metodo statico può accedere solo a variabili statiche (non può accedere a variabili di istanza, ovvero non statiche).

Gestione della memoria JVM

La JVM è divisa in tre parti:

- Ambiente delle classi: le *classi* del programma.

  Vengono memorizzati il codice dei metodi e le variabili statiche di tutte le classi; sono le parti condivise dai vari oggerri della classe; ~~le variabili statiche sono utilizzabili anche in assenza di oggetti~~.

- Stack: *record di attivazione* dei metodi (chiamate a funzione in C) e tutte le *variabili locali*.

  Vengono memorizzate le variabili locali dei metodi in esecuzione; per le variabili di tipi primitivi viene memorizzato il valore; per le variabili di tipo classe viene memorizzato un riferimento (indirizzo di memoria);

- Heap: *oggetti creati* nel programma, man mano che vengono caricati, e *variabili di istanza*.

  Per ogni oggetto creato vengono memoriazzate le variabili d'istanza (variabili non statiche); ogni oggetto nell'heap contiene anche il nome della classe di appartenenza.

Riferimenti

L'operazione di confonto (==) restituisce true solo se gli oggetti che si confrontano sono lo stesso oggetto, ovvero se hanno lo stesso riferimento.

```
Rettangolo r1 = new Rettangolo (10 ,12);
Rettangolo r2 = r1;
Rettangolo r3 = new Rettangolo (10 ,12);

System.out.println(r1==r2); // stampa true
System.out.println(r1==r3); // stampa false
```

Metodo Equals

Per questo si utilizza il metodo *equals*; esso permette di confrontare gli oggetti e non i riferimenti come nel caso di confronto (==).

Questo metodo è implementato in modo tale da confrontare una per una tutte le variabili interne di una coppia di oggetti.

```
s1.equals(s2)
```

Se si vuole consentire di confrontare oggetti della propria classe é bene ridefinire (overriding) il metodo *equals()*.

Attenzione alla firma!

```
public boolean equals ( Object o ) { .... }
```

Attenzione: bisongna definiere usando il parametro di tipo Object.

Garbage Collector

Una conseguenza grave del fatto che tutte le operazioni su variabili di tipo classe lavorino su riferimenti è che si possono ottenere **oggetti orfani**, ovvero **privi di riferimento**. Una volta che si è perso il riferimento ad un oggetto, esso non è più utilizzabile e diventa quindi garbage.

Java prevede quindi un meccanismo di rimozione degli oggetti privi di riferimento, ovvero il Garbage Collector. Questo meccanismo viene chiamato periodiamente dalla JVM; essa interrompe momentaneamente l'esecuzione del programma e pulisce la memoria delgi oggetti privi di riverimento.

Dichiarazione di una classe

```
public class <nome -classe > {
    <variabili di istanza >
    <variabili statiche >
    <costruttori >
    <metodi di istanza >
    <metodi statici >
```

Variabili e metodi sono chiamati anche membri della classe; l'ordine di dichiarazione all'interno del corpo n

Membri di istanza: codificano lo stato e le funzionalià dei singoli oggetti

Membri statici: codificano lo stato e le funzionalità della classe(condivide da tutti gli oggetti; significativi anche se non esiste nessun oggetto della classe)

l metodo **main è statico** (viene creato prima di invocare qualunque oggetto).

Una variabile dovrebbe essere d'istanza se assume valodi diversi per oggetti diversi.

Una variabile dovrebbe essere statica se assume gli stessi valori per oggetti diversi.

Se un metodo (static) utilizza una variabile d'istanza il compilatore darà errore.

Modificatori di visibilià

Private: utilizzabile sono all'interno della stessa classe.

Senza modificatore: utilizzabile sono nel package che contiene la classe.

Protected: utilizzabile nel package che contiene la calsse e in tutte le classi che ereditano da essa.

Public: utilizzabile ovunque.

I membri pubblici di una classe costituiscono l'interfaccia pubblica della classe (insieme delle risorse e delle funzionalità messe a disposizione alle altre classi.

Le variabili private rappresentano lo stato interno della classe; i metodi privati sono metodi ausiliari a disposizione degli altri metodi della classe.

Metodi

```
<modificatori > <tipo > <nome > (<lista_parametri_formali >) {
....
}
```

Esempio:

```
public static int minimo(int a, int b) {
    if (a<b) return a;
    else
    return b;
}
```

- Modificatori di visibilità o appartenenza a classe o istanze (static)
- Tipo restituito dal metodo
- Nome del metodo
- Parametri formali (input chiamata a metodo)

Passaggio di parametri

Il passaggio dei parametri ai metodi avviene per valore; i metodi lavorano su copie delle variabili passate come parametri.

Le ~~variabili di tipo classe~~ contengono riferimenti agli oggetti; ciò che viene chiamato al momento della chiamata è il riferimento. Il metodo lavora sull'oggetto originale (acceduto tramite una copia del riferimento).

Attenzione: Se un metodo public fa riferimeno ad una variabile privata, allora chiamando il metodo, viene restituito il riferimento all'oggetto è quindi possibile ottenere anche la variabile private.

Overloading

Consente di chiamare più metodi della stessa classe con lo stesso nome, purchè ogni metodo abbia una diversa firma (signature). La firma corrisponde alla sequenza di parametri formali (input chiamata a funzione).

Attenzione: la firma non comprene nè il tipo del metodo, nè i nomi dei parametri formali.

L'overloadin permette di utilizzare lo **stesso nome per metodi diversi che realizzano la stessa funzionalità su dati di tipo diverso.**

Metodo	Firma
int getVal()	getVal()
int minimo(int x, int y)	minimo(int,int)
int minimo(int a, int b)	minimo(int,int)
double minimo(double x, double y)	minimo(double,double)
int minimo(int x, int y, int z)	minimo(int,int,int)

Vargargs

Modo di definire metodi con numero variabile di parametri.

L'ultimo (e solo) parametro formale varargs ha la seguente sintassi:

```
<type>... par
```

All'interno del corpo del metodo il parametro par avrà tipo <type>[]...

la firma di int <metodo>(<type>... par) è <metodo>(<type>[]).

Costruttori e inizializzazione di variabili

Il costruttore è un metodo speciale che viene eseguito al momento di creazione dell'oggetto tramite primitiva new.

Inizializzazione di variabili

Variabili dichiarate localmente nei metodi sono variabili **locali** e **parametri formali**. Variabili dichiarate come membri di una classe sono variabili **statiche** e **variabili d'istanza**.

Le variabili locali vengono allocate nei record di attivazione relativo alla chiamata del metodo (nello stack); devono essere inizializzate esplicitamente (altrimenti errore dal compilatore).

Esempio:

```
int num;
whule (num >=0) {
    num = input.nextInt();
}
```

Errore segnalato dal compilatore:

```
variable num might not have been initialized
```

Le variabili statiche e di istanza vengono allocate nella memoria che descrive l'oggetto (heap); vengono sempre inizializzate (anche se non viene fatto espilicitamente) e vengono assegnati dei valori di default in base al tipo.

type	default value
variabili numeriche	0
boolean	false
variabili di tipo classe	null

Ci sono tre modi per inizializzare una variabile d'istanza o statica:

- assegnamento esplicito all'interno del costruttore
- inizializzazione esplicata nella dichiarazione
- inizializzazione (implicita) con valori di default

```
public class <name> {

    public int x;           // inizializzazione nel costruttore
    public int y = 3;       // inizializzazione nella dichiarazione
    public int z;           // inizializzazione con valore di default

    public <name> (int val) {
```

```
        x = val;
    }
}
-----------------------------
name p = new name(5.0);
System.out.println(p.x); // stampa 5.0;
System.out.println(p.y); // stampa 3.0;
System.out.println(p.z); // stampa 0.0;
```

Costruttori

I costruttori si dichiarano all'interno di una classe essenzialmene come metodi, ma:

- il nome deve coincidere con quello della classe
- il tipo del costruttore non deve essere specificato
- il modificatore static non può essere utilizzato

E' possibile applicare l'overloading (come nei metodi), ma con firma diversa.

Ogni classe ha u costruttore di default che inizializza le variabili d'istanza con il corrispondente valore di default; questo costruttore è disponibile solo se non è definito nessun costruttore. Se viene definito almeno un costruttore allora il costruttore di default non è più utilizzabile. A questo punto, se si vuole un costruttore senza parametri bisogna implementarlo.

This

in ogni corpo di un metodo d'istanza o costruttore è sempre disponibili la variabile this. Essa è un riferimento all'oggetto su cui si invoca il metodo o costruttore ed è anche detto parametro implicito del metodo.

Consenti di usare come nome di parametro formale lo stesso nome di una variabile d'istanza.

```
public class Punto {
    public double x, y;
    public Punto(double x, double y) {
        // assegna i parametri alle variabili d'istanza
        this.x = x;
        this.y = y;
    }
}
```

E' possibile anche restutuire un riferimento alla classe corrente:

```
return this
```

Può essere usato da un costruttore per chiamarne un'altro; **attenzione** è consentito solo come prima istruzione del costruttore.

Senza this()

```
public class Punto {
    public double x, y;

    // prende un solo valore e lo assegna sia a x che a y
    public Punto(double z) {
        this.x = z;
        this.y = z;
    }
```

```
    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Con this()

```
public class Punto {
    public double x, y;

    // prende un solo valore e lo assegna sia a x che a y
    public Punto(double z) {
        this(z,z); // chiama l'altro costruttore
    }

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Packages

Sono un meccanismo per raggruppare le classi. La libreria standard di Java (Java API) è organizzata in packages. Un package raggruppa classi logicamente correlate:

- *java.lang* riunisce classi fondamentali del linguaggio (String, Math, ...)
- *java.util* riunisce classi di frequente utilizzo (Scanner, Random, Timer, ...)
- *java.awt* e java.swing riuniscono classi per costruire interfacce grafiche

I packages sono utili per fare ordine e dividere le classi che fanno parte di una stessa categoria.

I packages necessari devono essere listati all'inizio del file e i vari file java devono essere salvati in diverse directori che corrispondono ai vari packages.

Se non si specifica nessun package, la classe farà parte del package default corrispondente alla directory principale.

I packages possono essere raggruppati formando una struttura gerarchica. Si utilizzano i punti per navigare nella struttura del file system come se fosse '/' in bash.

I Packages sono sottoposti ai modificatori di visibilità.

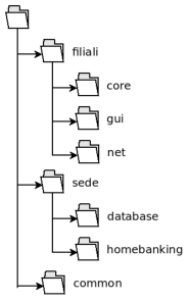

Figure 1: Java packages structure example.

Ereditarietá

La OOP si basa sull'astrarre concetti che hanno caratteristiche comuni tra di loro.
Esistono oggetti che hanno alcune variabili e alcuni metodi in comune ed altri no, perché descrivono concetti correlati ma separati. Si introduce cosí il concetto di ereditarietá, ovvero poter descrivere oggetti come facenti parte di piú classi e come tali, poter utilizzare i metodi.

Per definire una classe come estensione di un'altra si utilizza la primitiva **extends**.

```
public class Student extends Persona {
    ...
}
```

La classe che viene estesa é detta **superclasse**, essa infatti sará raffinata in concetti piú specifici da altre classi. La classe che estende é detta **sottoclasse**, essa é un raffinamento del concetto della superclasse. Nell'esempio, la classe persona é detta superclasse, mentre la classe studente é detta sottoclasse.

Alla creazione di un oggetto di una sottoclasse, viene prima creata un'istanza dell'oggetto superclasse (tramite il rispettivo costruttore), vengono aggiunte le variabili della sottoclasse all'istanza appena creata e viene chiamato il costruttore della/e sottoclasse/i che inizializza.

Super é una direttiva che fa riferimento all'oggetto stesso (come this) ma che consente di utilizzare i costruttori e i metodi della superclasse.

this(...) richiama un costruttore della sottoclasse.

super(...) richiama un costruttore della superclasse.

Super gode delle stesse proprietá di this; se omessa la chiamata super(...), viene chiamata automaticamente dal costruttore senza parametri della superclasse (ovvero nel caso di costruttore di default).

Si utilizza *protected* per estendere la visibiliá a tutte le sottoclassi, anche di altri packages, mentre se si omette il modificatore, la visibilitá sará limita alle sole classi del package di appartenenza della superclasse.

In caso di **overriding**, ovvero quando ci sono piú metodi con lo stesso nome appartenenti a sottoclasse e superclasse e con stessa firma, la sottoclasse ridefinisce (sostituendo) il metodo della superclasse. Se la firma é diversa allora avremo **overloading** come se fecessero parte della stessa classe. Ció permette di ridefinire i metodi che differiscono.

Late binding o **binding dinamico:** é un meccanismo di collegamento "ritardato" della chiamata di un metodo con la classe corrispondente. Solo durante l'esecuzione, la chiamata al metodo viene collegata alla (sotto)classe giusta.

Un'oggetto di una sottoclasse puó essere usato ovunque sia richiesto un oggetto della superclasse; esso puó essere passato ad un metodo, restituito ad un metodo e assegnato a una variabile.

~~al momento della creazione (17 - p.20)~~ Ogni oggetto in Java ha un **tipo apparente** (é quello specificato come tipo della variabile corrispondente) e un **tipo effettivo** (é quello con cui si é costrito l'oggetto). Il compilatore di basa solo sult tipo apparente, in quanto il tipo effettivo puó variare durante l'esecuzione.

A tempo di esecuzione la JVM (interprete) usa il tipo effettivo per controllare il tipo dell'oggetto e per chiamare i metodi della classe corrispondente.

In qualsiasi momento possiamo forzare il compilatore a considerare una variabile come se fosse un oggetto di una sottoclasse, tramite un **type cast**; ovviamente le istanze devono essere in una relazione supertipo-sottotipo ed, in caso di incompatibilitá, il programma verrá arrestato.

```
Persona p = new Studente ( " Guido Guidi " ," Via Roma 12 " );
int m = (( Studente ) p ). getMatricola ();                    // forza il compilatore
Studente s = ( Studente ) p ;                                  // forza il compilatore
```

instanceof: é un predicato che verifica il tipo effettivo di un oggetto, la quale restituisce una espressione booleana *true* se l'oggetto ha tipo effettico corrispondente al tipo effettivo della classe comparata.

```
p instanceof Student
```

Gerarchia di classi

L'ereditarietá e transitiva e singola (ogni classe puó estendere una sola altra classe; ereditarietá cicliche sono vietate).

La classe **Object** é la classe fondamentale dalla quale discendono tutte le altre classi[1] e fornisce a tutte classi alcuni metodi fondamentali [2].

Classi astratte

Una classe che contiene almeno un metodo astratto é una **classe astratta** e un **metodo astratto** é un metodo che prevede solo una intestatione, ma che non é implementato.

```
public abstract class <ClassName> {
    ...
    // intestazione del metodo
    public abstract <type> <MethodName>();          // Metodo astratto
}
```

Una classe astratta non puó creare oggetti, puó solo essere estesa da sottoclassi che ne definisce i metodi astratti, tramite overriding. Puó prevedere costruttori che saranno richiamati dalle sottoclassi tramite super().

Interfacce

Sono classi che esasperano il principio di classi astratte, ovvero sono classi che sono formate solo da metodi astratti che peró non utilizzano il modificatore *abstract* e utilizzano la parola chiave **interface al posto di classi**; una classe che implementa i metodi dell'interfaccia deve usare la parola **implements** invece di *extend*.

Il vantaggio é che una classe puó estendere una sola classe (astratta o meno), mentre **una classe puó implementare tante interfacce**.

```
public class Salame extends Affettato implements ProdottoPesabile , Prodotto Prezzato {
    ...
}
```

Situazioni anomale a run-time

Java é un linguaggio fortemente tipato, cioé prevede l'utilizzo dei tipi che consente di individuare molti errori al momento della compilazione.

- Tentativi di accedere a posizioni di un array che sono fuori dai limiti (indice negativo o maggiore della dimensione)
- Errori aritmetici (esempio: divisione per zero)
- Errori di formato: si chiede all'utente un intero e l'utente inserisce una stringa

Gestione delle eccezioni

Ogni volta che la JVM si trova in situazioni anomale: 1. il programma viene sospeso. 2. viene creato un oggetto classe corrispondente all'anomalia che si é verificata. 3. viene passato il controllo a un gestore di eccezioni (implementato dal programmatore). Se non sono previste gestore delle eccezioni, il programma si interrompe e viene stampato il messaggio di errore.

Costrutto try-catch

Consente di monitorare una porzione di codice (all'interno di un metodo) e specificare cosa fare in caso di anomalia nella porzione di programma monitorata.

[1] per transitivitá
[2] come *toString()* e *equals()*

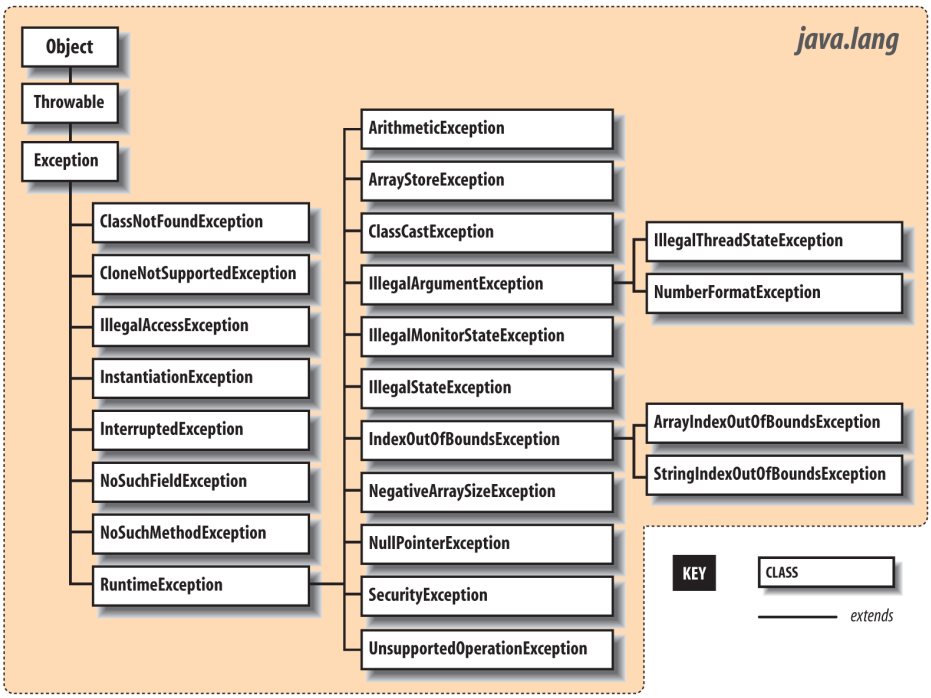

Figure 2: Java Exception class arguments.

```
// ... comandi non monitorati ....
try {
// ... comandi monitorati ....
}
catch ( Exception e1 ) {
// ... comandi da eseguire in caso di eccezione

...

catch ( Exception en ) {
// ... comandi da eseguire in caso di eccezione
}
```

La variabile *e* é un oggetto che puó contenere informazioni utili sull'errore.

Gerarchia delle eccezioni

La classe **Exception** descrive una eccezione generica; situazioni anomale piú specifiche sono descritte da sottoclassi di *Exception*.

Gestire eccezioni

I vari gestori (catch) vengono controllati in sequenza. Viene eseguito solo il primo *catch* che prevede un tipo di eccezione che é supertipo dell'eccezione che si é verificata.

Eccezioni checked e unchecked

- **Checked:** il compilatore richiede che ci sia implementato un gestore
- **Unchecked:** il gestore non é obbligatorio.

Per essere *unchecked* un'eccezione deve essere una sottoclasse di RuntimeException, altrimenti é *checked*.

Esempi tipici di eccezioni checked: - le eccezioni che descrivono errori di input/output (lettura o scrittura su file, comunicazione via rete, ecc. . .) - le eccezioni definite dal programmatore

Lanciare eccezioni

Il meccanismo delle eccezioni puó anche essere usato per segnalare situazioni di errore.

Il comando **throw** consente di lanciare un'eccezione quando si vuole. Si puó usare la classe Exception, una sua sottoclasse giá definita, o una sua sottoclasse definita dal programmatore stesso. *throw* si aspetta di essere seguito da un oggetto, che solitamente é costruito al momento (tramite new). Il costruttore di una eccezione prende come parametro (opzionale) una stringa di descrizione.

```
throw new Exception("operazione non consentita");
throw new AritmeticException();
throw new EccezionePersonalizzata();
```

Il comando *throw* puó essere utilizzato direttamente dentro il *try-catch*, anche se é piú sensato utilizzarlo dentro i metodi; in questo caso permette di interrompere il metodo in caso di situazioni anomale.

Un metodo che contiene dei comandi throw deve elencare le eccezioni che possono essere sollevate. L'elenco deve essere fatto nell'intestazione, usando la parola chiave *throws*

```
public void preleva ( int somma )
throws IOException , I l l e g a l P a r a m e t e r E x c e p t i o n { ... }
```

Attenzione alla s finale! throws si usa nell'intestazione del metodo. throw si usa all'interno (nel punto in cui si verifica l'errore).

```
public class Rettangolo {

    private base ;
    private altezza ;

    // ... altri metodi e costruttori

public void setBase ( int x ) throws EccezioneBaseNegativa {
    if (x <0) throw new EccezioneBaseNegativa()
    else base = x ;
    }
}

public class EccezioneBaseNegativa extends Exception {

    EccezioneBaseNegativa() {
        super ();
    }

    EccezioneBaseNegativa( String msg ) {
        super ( msg );
    }
```

```
}
```

Vettori

Sono strutture dati dinamiche simili agli array ma che possono variare la dimensione a runtime. Esistono con queste caratteristiche gli **ArrayList** e i **Vector**.

Per creare un vettore bisogna creare un oggetto di classe *Vector*, inizialmente vuota.

```
Vector < String > v = new Vector < String >(); // vettore di stringhe
```

Gli elementi possono essere scritti con i metodi **set** e **get** e **add**[3].

Nelle parentesi angolari, il metodo *Vector* si aspetta un tipo classe: gna usare le classi involucro

- Integer per int
- Double per double
- Long per long
- Character per char
- Boolean per boolean

Tutti i metodi come *toString()* e *equals()* che discendono dalla classe *Object* vengono ridefiniti tramite *overriding* per poter lavorare su classe *Vector*.

Altre strutture dati native

- **HashSet:** Descrive insiemi di elementi senza duplicati e senza un ordine predefinito
- **HashMap:** Descrive dizionari, ossia associazioni di chiavi-valori
- **Stack:** Descrive pile di valori
- **Alberi**

Persistenza dei dati

I dati devono sopravvivere alla terminazione del programma ed essere disponibili in una esecuzione successiva.

In Java l'input e l'output sono gestiti come stream. Uno **stream di input** prevede una sorgente di dati (tastiera o file), mentre uno **stream di output** precede una destinazione per i dati (schermo o file).

Si distinguono due classi di librerie per la gestione dei flussi:

1. gestione di stream di caratteri: utili per leggere e scrivere su testo
2. gestione di stream di bytes: utili per leggere e scrivere su file binari (immagini, video, dati, . . .)

1) utilizza le classi **FileReader** e ** FileWriter**, mentre** 2)** utilizza **FileInputStream** e **FileOutputStream**; entrambe fanno parte della *java.io* e devono essere importate nei programmi.

La classe *FileReader* consente di leggere un file un carattere per volta.

```
// chiamata al costruttore; apre il file altrimenti lancia un eccezione.
FileReader reader = new FileReader ( " prova.txt " );
// restituisce -1 se il file é terminato, altrimenti casta a char.
int next = reader.read ();
// chiude lo stream.
reader.close ();
```

[3] hanno una marea di metodi di manipolazione.

Java utilizza le eccezioni di tipo *IOException*; possono essere lanciate da tutti i metodi delle classi di gestione degli stream e sono controllate (devono essere gestite da chi chiama i metodi).

La scrittura su file tramite uno stream di output é analoga alla lettura; si utilizza la classe *FileWriter* e il metodo *write()*.

Input/Output bufferizzato

Per evitare di eccedere in richieste al S.O. di lettura/scrittura, si utilizza un buffer (memoria tampone) che ne velocizza le operazioni.

Vengono utilizzate le classi **BufferedReaders** e **BufferedWriter** per i *stream di caratteri* e **BufferedInputStream** e **BufferedOutputStream** per i *stream di bytes*.

```
BufferedReader reader = new BufferedReader ( new FileReader ( " prova.txt " ));
```

Per manipolare e dare forma ai dati letti/scritti, si utilizzano classi e metodi simili a quelli utilizzati da *Scanner* e *PrintWriter* Come per *Scanner*, si utilizzano i metodi **nextInt()**, **nextLine()**, ecc... per leggere da file; bisogna essere sicuri al tipo di valore che si legge o essere pronti a gestire eccezioni. Come per *PrintWriter* si utilizzano i metodi **println()** e **print()** per scrivere nel file.

Serializzazione degli oggetti

In caso di un programma complesso che usat molte strutture dati, salvare lo stato del programma in un file diventa complicato; in questo caso é utile la **serializzazione degli oggetti**. Esso é una funzionalitá di Java che consente di rappresentare oggetti arbitrariamente complessi come sequenze di byte ben definite; tali sequenze di byte potranno essere salvate su file tramite uno stream.

Per leggere/scrivere un oggetto in un file bisogna usare le classi **ObjectInputStream** e **ObjectOutputStream**.

Nel caso di ObjectInputStream (l'output é analogo) si procede come segue:

1. passare al costruttore di *ObjectInputStream* uno stream di bytes di input (meglio se bufferizzato, *BufferedInputStream*)
2. usare il metodo *readObject()* per leggere dal file facendo attezione; se il file non contiene un oggetto viene lanciata l'eccezione *ClassNotFoundException*
3. chiudere lo stream

Affinché un oggeto possa essere serializzato e salvato su file, la sua classe deve implementare l'interfaccia **Serializable**. Tale interfaccia non prevede metodi e serve solo per far dichiarare al programmatore che autorizza il salvataggio dei suoi oggetti; é un meccanismo di sicurazza (la serializzazione espone l'oggetto).

Alle classi che implementano Serializable viene richiesto (anche se non é obbligatorio) di includere una costante statica *serialVersionUID* (di tipo long) che servirebbe per distinguere tra diverse versioni della stessa classe (es.modifiche successive).

Inner class (Classi interne)

É possibile definire una classe all'interno di un altra.

Classe anonime

Sono dichiarazione e instaziazione di una classe allo stesso tempo.

```
public void metodo () {
    ActionListener listener = new ActionListener (){
        public void actionPerfor med ( ActionEvent evt ) {
            JOptionPane . s how Mes sag eDi alog ( null , " Buongiorno ! " );
        }
    };

    JButton button = new JButton ( " Saluta " );
    button . add Act ion Lis tene r ( listener );
}
```

Classi generiche

Il tipo é generico (T) e sará specificato al momento della chiamata del costruttore

```
public class Coppia <T > {

    public T elemento1 ;
    public T elemento2 ;

    public Coppia ( T elemento1 , T elemento2 ) {
        this . elemento1 = elemento1 ;
        this . elemento2 = elemento2 ;
    }
}
// una coppia di interi
Coppia < Integer > = new Coppia < Integer >(10 ,20);
```