

Caratteristiche principali di Java

Sito: [ITS ICT Piemonte - Piattaforma di formazione a distanza](#)
Corso: (D68444-2-2021-20) Programmazione - Java
Libro: Caratteristiche principali di Java

Stampato da: Brian Junior Potosi Ccuno
Data: martedì, 8 novembre 2022, 20:04

Descrizione

Corso programmazione Java



fondo sociale europeo



Appunti del corso



per una crescita intelligente,
sostenibile ed inclusiva

www.regione.piemonte.it/europa2020

INIZIATIVA CO-FINANZIATA CON FSE

Sommario

- 1. Programma del corso**
- 2. Caratteristiche principali di Java**
- 3. Elementi fondamentali**
- 4. Il controllo del flusso**
- 5. I metodi**
- 6. Classi e oggetti**
- 7. OOP progettazione avanzata**

1. Programma del corso

La programmazione orientata agli oggetti in Java

- Ambienti di sviluppo (JDK) e primi approcci al codice
- Le basi della programmazione object oriented: classi e oggetti
- Variabili, attributi, metodi e costruttori
- Identificatori, tipi di dati e array
- Operatori e gestione del flusso di esecuzione
- Costrutti di programmazione semplice: if, operatore ternario, while
- Costrutti di programmazione avanzati: for, do while, for migliorato, switch
- Classi ed oggetti
- Classi innestate, classi anonime

Programmazione avanzata in Java

- Modificatori, package, interfacce, enumerazioni
- Gestione dei thread
- Le librerie alla base del linguaggio
- Java.lang e java.util
- Comunicare con Java
- Input, output e networking

Accesso ai dati con Java

- Java e la gestione dei dati: supporto a SQL e XML
- Caratteristiche di JDBC
- Introduzione a Hibernate DAO

Design Pattern

- Singleton, Adapter, Factory, Builder
- Il modello MVC

Introduzione a J2EE

- Servlet e Filtri
- Introduzione a JSP

Servizi Web

- Realizzazione ed esposizione di API con varie le varie metodologie (put, get, post, ecc)
- Test API attraverso strumenti (PostMan)

Introduzione ai Framework in Java

- Primi passi con Spring
- Inversion Of Control
- Gestione dell'accesso ai dati

2. Caratteristiche principali di Java

Caratteristiche principali di Java

- Java è un linguaggio di alto livello e orientato agli oggetti, creato dalla Sun Microsystem nel 1995.

Le motivazioni, che guidarono lo sviluppo di Java, erano quelle di creare un linguaggio semplice e familiare.

Le caratteristiche del linguaggio di programmazione Java sono:

- La tipologia di linguaggio orientato agli oggetti (ereditarietà, polimorfismo, ...)
- la gestione della memoria effettuata automaticamente dal sistema che si preoccupa dell'allocazione e della successiva deallocazione della memoria
- la portabilità, cioè la capacità di un programma di poter essere eseguito su piattaforme diverse senza dover essere e modificato e ricompilato

Caratteristiche di Java

- Semplice e familiare
- Orientato agli oggetti
- Indipendente dalla piattaforma
- interpretato
- Sicuro
- Robusto
- Distribuito e dinamico
- Multi-thread

Semplice e familiare

- Basato su C
- Sviluppato da zero
- Estremamente semplice: senza puntatori, macro, registri
- Apprendimento rapido
- Semplificazione della programmazione
- Riduzione del numero di errori

Orientato agli oggetti

- Orientato agli oggetti dalla base

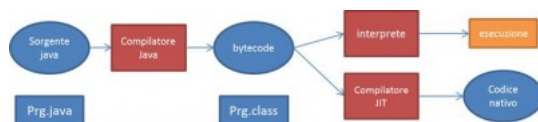
- In Java tutto è un oggetto
- Incorpora le caratteristiche
 - Incapsulamento
 - Polimorfismo
 - Ereditarietà
- Collegamento dinamico
- Non sono disponibili
 - Ereditarietà multipla
 - Overload degli operatori

Indipendente dalla piattaforma

- Più efficiente di altri linguaggi interpretati
- Soluzione: la macchina virtuale: JVM
- Linguaggio macchina bytecode

Interpretato

- Il bytecode deve essere interpretato



- Vantaggi rispetto ad altri linguaggi interpretati
- Codice più compatto
- Efficiente
- Codice confidenziale (non esposto)

sicuro

- Supporta la sicurezza di tipo sandboxing
- Verifica del bytecode
- Altre misure di sicurezza
- Caricatore di classi
- Restrizioni nell'accesso alla rete

Robusto

- L'esecuzione nella JVM impedisce di bloccare il sistema
- L'assegnazione dei tipi è molto restrittiva
- La gestione della memoria è sempre a carico del sistema
- Il controllo del codice avviene sia a tempo di compilazione sia a tempo di esecuzione (runtime)

Distribuito e dinamico

- Disegnato per un'esecuzione remota e distribuita
- Sistema dinamico
- Classe collegata quando è richiesta
- Può essere caricata via rete
- Dinamicamente estensibile
- Disegnato per adattarsi ad ambienti in evoluzione

Multi-thread

- Soluzione semplice ed elegante per la multiprogrammazione
- Un programma può lanciare differenti processi
- Non si tratta di nuovi processi, condividono il codice e le variabili col processo principale
- Simultaneamente si possono svolgere vari compiti

Versioni Java

Version	Release date	End of Free Public Updates	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014 May 2026 for AdoptOpenJDK	January 2019 for Oracle (commercial) At least May 2026 for Amazon Corretto	Indefinitely for Oracle (personal use) At least May 2026 for Amazon Corretto
		December 2030	N/A

Versioni Java (nuovo approccio Oracle)

Version	Release date	End of Free Public Updates	Extended Support Until
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least October 2024 for AdoptOpenJDK At least September 2027 for Amazon Corretto	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

3. Elementi fondamentali

Le variabili e le costanti

- Una variabile è un'area di memoria identificata da un nome
- Il suo scopo è di contenere un valore di un certo tipo
- Serve per memorizzare dati durante l'esecuzione di un programma
- Il nome di una variabile è un **identificatore**
 - può essere costituito da lettere, numeri e underscore
 - non deve coincidere con una parola chiave del linguaggio
 - è meglio scegliere un **identificatore** che sia **significativo** per il programma

esempio

```
public class Triangolo {  
    public static void main ( String [] args ) {  
  
        int base , altezza ;  
        int area ;  
  
        base = 5;  
        altezza = 10;  
        area = base * altezza / 2;  
  
        System.out.println ( area );  
    }  
}
```

Usando le variabili il programma risulta essere **più chiaro**:

- Si capisce meglio quali siano la base e l'altezza del triangolo
- Si capisce meglio che cosa calcola il programma

Dichiarazione

- In Java ogni variabile deve essere **dichiarata prima del suo uso**
- Nella dichiarazione di una variabile se ne specifica il **nome** e il **tipo**
- Nell'esempio, abbiamo dichiarato tre variabili con nomi base, altezza e area, tutte di tipo int (numeri interi)
 - int base , altezza ;
 - int area ;

ATTENZIONE! Ogni variabile deve essere dichiarata **UNA SOLA VOLTA** (la prima volta che compare nel programma)

```
base =5;  
altezza =10;  
area = base * altezza /2;
```

Assegnazione

- Si può memorizzare un valore in una variabile tramite l'operazione di assegnazione
- Il valore da assegnare a una variabile può essere un letterale o il risultato della valutazione di un'espressione
- Esempi:

```
base =5;
altezza =10;
area = base * altezza /2;
```

- I valori di base e altezza vengono letti e usati nell'espressione
- Il risultato dell'espressione viene scritto nella variabile area

Dichiarazione + Assegnazione

Prima di poter essere usata in un'espressione una variabile deve:

- essere stata dichiarata
- essere stata assegnata almeno una volta (inizializzata)
- NB: **si possono combinare dichiarazione e assegnazione.**

Ad esempio:

```
int base = 5;
int altezza = 10;
int area = base * altezza / 2;
```

Costanti

Nella dichiarazione delle variabili che **NON DEVONO** mai cambiare valore si può utilizzare il modificatore **final**

```
final double IVA = 0.22;
```

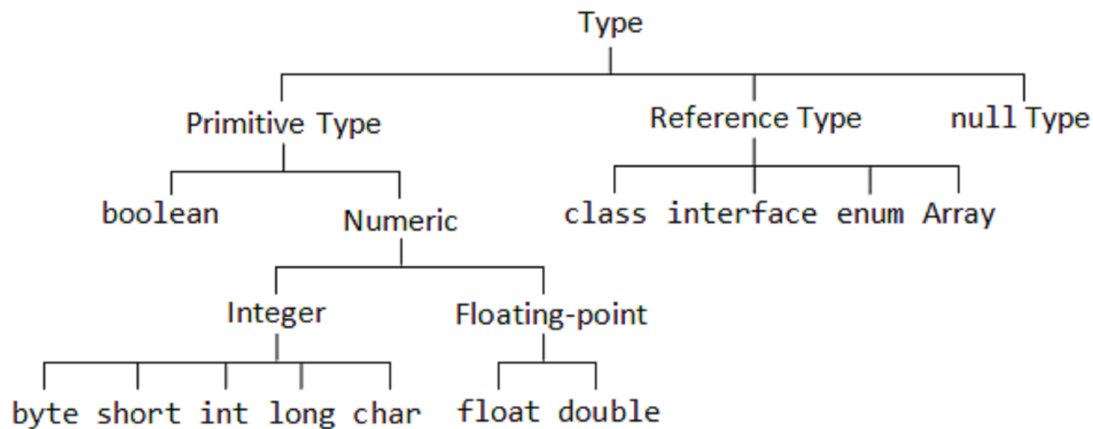
- Il modificatore **final** trasforma la variabile in una costante
- Il compilatore si occuperà di controllare che il valore delle costanti non venga **mai modificato** (ri-assegnato) dopo essere stato inizializzato.
- Aggiungere il modificatore **final** non cambia funzionamento programma, ma serve a prevenire errori di programmazione
- Si chiede al compilatore di controllare che una variabile non venga ri-assegnata per sbaglio
- Sapendo che una variabile non cambierà mai valore, il compilatore può anche eseguire delle **ottimizzazioni** sull'uso di tale variabile.

Input dall'utente

- Per ricevere valori in input dall'utente si può usare la classe Scanner, contenuta nel package **java.util**
- La classe Scanner deve essere richiamata usando la direttiva import prima dell'inizio del corpo della classe

Tipi di dato primitivi

- In un linguaggio ad oggetti puro, vi sono solo classi e istanze di classi:
- i dati dovrebbero essere definiti sotto forma di oggetti



Java definisce alcuni tipi primitivi

- Per efficienza Java definisce tipi primitivi
- La dichiarazione di una istanza alloca spazio in memoria
- Un valore è associato direttamente alla variabile
- (e.g, `i == 0`)
- Ne vengono definiti dimensioni e codifica
- Rappresentazione indipendente dalla piattaforma

Tabelle riassuntive: tipi di dato

Primitive Data Types

type	bits
byte	8 bit
short	16 bit
int	32 bit
long	64 bit
float	32 bit
double	64 bit
char	16 bit
boolean	true/false

I caratteri sono considerati interi

I tipi numerici, i char

- Esempi
- `123` (int)
- `256789L` (L o l = long)
- `0567` (ottale) `0xff34` (hex)
- `123.75` `0.12375e+3` (float o double)
- `'a'` `'%'` `'\n'` (char)

- `'\123'` (\ introduce codice ASCII)

Tipo boolean

- `true`
- `false`

Operatori

Operatori aritmetici

- Di assegnazione: `=` `+=` `-=` `*=` `/=` `&=` `|=` `^=`
- Di assegnazione/incremento: `++` `--` `%=`
- Operatori Aritmetici: `+` `-` `*` `/` `%`

Operatore	Significato
<code>+</code>	addizione
<code>-</code>	sottrazione
<code>*</code>	motiplicazione
<code>/</code>	divisione
<code>%</code>	resto
<code>++var</code>	preincremento
<code>--var</code>	predecremento
<code>var++</code>	postincremento
<code>var--</code>	postdecremento

Operatori di assegnazione

Operatore	Significato
<code>=</code>	addizione
<code>+=</code>	addizione assegnazione
<code>-=</code>	sottrazione assegnazione
<code>*=</code>	motiplicazione assegnazione
<code>/=</code>	divisione assegnazione
<code>%=</code>	resto assegnazione

Operatori relazionali

`==` `!=` `>` `<` `>=` `<=`

Operatore	Significato
<code><</code>	minore di
<code><=</code>	minore di o uguale a
<code>></code>	maggiore di
<code>>=</code>	maggiore di o uguale a
<code>==</code>	uguale a
<code>!=</code>	non uguale / diverso

Operatori per Booleani

- Bitwise (interi): & | ^ << >> ~

Operatore

&&

||

!

^

Significato

short circuit AND

short circuit OR

NOT

exclusive OR

Attenzione:

- Gli operatori logici agiscono **solo su booleani**
 - Un intero NON viene considerato un booleano
 - Gli operatori relazionali forniscono valori booleani

Operatori su reference

Per i riferimenti/reference, sono definiti:

- Gli operatori relazionali == e !=
 - test sul riferimento all'oggetto, **NON sull'oggetto**
- Le assegnazioni
- L'operatore "punto"
- NON è prevista l'aritmetica dei puntatori, vengono gestiti dalla JVM

Operatori matematici

Operazioni matematiche complesse sono permesse dalla **classe Math** (package java.lang)

- `Math.sin (x)` calcola $\sin(x)$
- `Math.sqrt (x)` calcola $x^{(1/2)}$
- `Math.PI` ritorna pi
- `Math.abs (x)` calcola $|x|$
- `Math.exp (x)` calcola e^x
- `Math.pow (x, y)` calcola x^y

Esempio

- `z = Math.sin (x) - Math.PI / Math.sqrt` 🙌

Caratteri speciali

Literal

`\n`

`\t`

`\b`

`\r`

Represents

New line

Horizontal tab

Backspace

Carriage return

Literal	Represents
<code>\f</code>	Form feed
<code>\\</code>	Backslash
<code>\"</code>	Double quote
<code>\ddd</code>	Octal character
<code>\xdd</code>	Hexadecimal character
<code>\udddd</code>	Unicode character

Espressioni aritmetiche

```
public class Triangolo {
    public static void main ( String [] args ) {
        System.out.println (5*10/2);
    }
}
```

Il programma risolve l'espressione $5*10/2$ e stampa il risultato a video

Espressioni aritmetiche e precedenza

singoli "letterali"

- Letterali interi: 3425, 12, -34, 0, -4, 34, -1234,
- Letterali frazionari: 3.4, 5.2, -0.1, 0.0, -12.45, 1235.3423,

operatori aritmetici

- moltiplicazione *
- divisione /
- modulo % (resto della divisione tra interi)
- addizione +
- sottrazione -

Le operazioni sono elencate in **ordine decrescente di priorità** ossia $3+2*5$ fa 13, non 25

Le parentesi tonde cambiano l'ordine di valutazione degli operatori ossia $(3+2)*5$ fa 25

Inoltre, tutti gli operatori sono associativi a sinistra ossia $3+2+5$ corrisponde a $(3+2)+5$ quindi $18/6/3$ fa 1, non 9

operazione di divisione

- L'operazione di divisione / si comporta diversamente a seconda che sia applicato a letterali interi o frazionari
- $25/2 = 12$ (divisione intera)
- $25\%2 = 1$ (resto della divisione intera)
- $25.0/2.0 = 12.5$ (divisione reale)
- $25.0\%2.0 = 1.0$ (resto della divisione intera)
- Una operazione tra un letterale intero e un frazionario viene eseguita come tra due frazionari
- $25/2.0 = 12.5$
- $1.5 + (25/2) = 13.5$ (attenzione all'ordine di esecuzione delle operazioni)
- $2 + (25.0/2.0) = 14.5$

Casting e promotion

- `(nometipo) variabile`

- `(nometipo) espressione`
- Trasforma il valore della variabile (espressione) in quello corrispondente in un tipo diverso
- Il cast si applica anche a `char`, visto come tipo intero positivo
- La promotion è automatica quando necessaria
 - Es. `double d = 3 + 4;`
- Il casting deve essere esplicito: il programmatore si assume la responsabilità di eventuali perdite di informazione
 - Per esempio
 - `int i = (int) 3.0 * (int) 4.5;` i assume il valore 12
 - `int j = (int) (3.0 * 4.5);` j assume il valore 13

casting dei tipi reference (oggetti)

- è permesso solo in caso di **ereditarietà**
- la conversione da sotto-classe a super-classe è **automatica**
- la conversione da super-classe a sotto-classe richiede **cast esplicito**
- la conversione tra riferimenti non in relazione tra loro **non è permessa**

esempio promotion

```
char a = 'a';
// promotion int è più grande e i valori sono compatibili
int b = a;

System.out.println(a); // a
System.out.println(b); // 97
```

esempi type casting

```
byte b = (byte) 261;
System.out.println(b); // 5

System.out.println( Integer.toBinaryString(b) ); // 101
System.out.println( Integer.toBinaryString(261) ); // 10000101

int a = (int) 1936.27;

System.out.println(a); // 1936
```

con il tipo boolean non si può fare il typecasting

```
int a = (int) true; // vietato - ... cannot be converted to ...
boolean falso = (boolean) 0; // vietato - ... cannot be converted to ...
```

Stringhe e Caratteri

Caratteristiche principali

Classi disponibili

- String
 - Modella stringhe (sequenze – array di caratteri)
 - **Non modificabile** (dichiarata final)
- StringBuilder
 - Modificabile
- StringBuffer (non si usa più)
 - Modificabile
- Character
- CharacterSet

Definizione

```
String myString; myString = new String ("stringa esempio");
```

- Oppure

```
String myString = new String ("stringa esempio");
```

- Solo per il tipo String vale

```
String myString = "stringa esempio";
```

- Il carattere " (doppi apici) può essere incluso come "
- Il nome della stringa è il riferimento alla stringa stessa
- Confrontare due stringhe NON significa confrontare i riferimenti

NB: I metodi che gestiscono il tipo String NON modificano la stringa, ma ne creano una nuova

Concatenare stringhe

- Operatore concat

- myString1.concat(myString2)
- String s2 = "Ciao".concat(" a tutti").concat("!");
- String s2 = "Ciao".concat(" a tutti").concat("!");

- Utile per definire stringhe che occupano più di una riga

- Operatore + "questa stringa" + "e formata da tre" + "stringhe"``

- La concatenazione funziona anche con altri tipi, che vengono automaticamente convertiti in stringhe

```
System.out.println ("pi Greco = " + 3.14);
```

```
System.out.println ("x = " + x);
```

Lunghezza stringa

- `int length()`
 - `myString.length()`
 - `"Ciao".length()` restituisce 4
 - `"".length()` restituisce 0
- Se la lunghezza è N, i caratteri sono indicizzati da 0 a N-1

Carattere i-esimo

- `char charAt(int)`
- `myString.charAt(i)`

Confronta stringa con altra stringa

- `boolean equals(String s) * myString.equals("stringa")` ritorna true o false
- `boolean equalsIgnoreCase(String s)`
- `myString.equalsIgnoreCase("StRiNgA")`

Confronta con altra stringa facendone la differenza

- `int compareTo(String str)`
- `myString.compareTo("stringa")` ritorna un valore \geq o \leq 0

Trasforma int in String

- `String valueOf(int)`
- Disponibile per tutti tipi primitivi

Restituisce indice prima occorrenza di c

- `int indexOf(char c)`
- `int indexOf(char c, int fromCtrN)`

Altri metodi

- `String toUpperCase(String str)`
- `String toLowerCase(String str)`
- `String substring(int startIndex, int endIndex)`
- `String substring(int startIndex)`

Esempio

```
String s1, s2;
s1 = new String("Prima stringa");
s2 = new String("Prima stringa");
System.out.println(s1);
/// Prima stringa
System.out.println("Lunghezza di s1 = " +
s1.length());
// 26
if (s1.equals(s2)) ...
// true
if (s1 == s2) ...
// false
String s3 = s3.substring (2, 6);
// s3 == "ima s"
```

[altri esempi sulle stringhe](#)

Array

- Sequenze ordinate di
 - Tipi primitivi (int, float, etc.)
 - Riferimenti ad oggetti (vedere classi!)
- Elementi dello stesso tipo
 - Indirizzati da indici
 - Raggiungibili con l'operatore di indicizzazione: le **parentesi quadre** []
 - Raggruppati sotto lo stesso nome

In Java gli array sono Oggetti

- Sono allocati nell'area di memoria riservata agli oggetti creati dinamicamente (heap)

Dimensione

- Può essere stabilita a run-time (quando l'oggetto viene creato)
- È fissa (non può essere modificata)
- E' nota e ricavabile per ogni array

Array Mono-dimensionali (vettori)

Dichiarazione di un riferimento a un array

- `int[] voti;`
- `int voti[];`

La dichiarazione di un array non assegna alcuno spazio

```
voti == null
```

Creazione di un Array

L'operatore new crea un array:

- Con costante numerica

```
int[] voti;
...
voti = new int[10];
```

- Con costante simbolica

```
final int ARRAY_SIZE = 10;
int[] voti;
...
voti = new int[ARRAY_SIZE];
```

- Con valore definito a run-time

```
int[] voti;
... definizione di x (run-time) ...
voti = new int[x];
```

**Utilizzando un inizializzatore-* (che permette anche di riempire l'array)

- L'operatore new inizializza le variabili
 - 0 - per variabili di tipo numerico (inclusi i char)
 - false - per le variabili di tipo boolean

```
int[] primi = {2,3,5,7,11,13};
...
int [] pari = {0, 2, 4, 6, 8, 10,};
// La virgola finale e' facoltativa
// (elenchi lunghi)
```

- Dichiarazione e creazione possono avvenire contestualmente
- L'attributo length indica la lunghezza dell'array, cioè il numero di elementi
- Gli elementi vanno da 0 a length-1

```
for (int i=0; i<voti.length; i++)
voti[i] = i;
```

In Java viene fatto il bounds checking

- Maggior sicurezza
- Maggior lentezza di accesso

Il riferimento ad array

- Non è un puntatore al primo elemento
- È un puntatore all'oggetto array
- Incrementandolo non si ottiene il secondo elemento

Array di oggetti

Per gli array di oggetti (e.g., Integer) `Integer [] voti = new Integer [5];` ogni elemento e' un riferimento

L'inizializzazione va completata con quella dei singoli elementi

```
voti[0] = new Integer (1);  
voti[1] = new Integer (2);  
...  
voti[4] = new Integer (5);
```

Array Multi-dimensionali (Matrici)

Array contenenti riferimenti ad altri array

Sintatticamente sono estensioni degli array a una dimensione

Sono possibili righe di lunghezza diverse (matrice = array di array)

```
int[][] triangle = new int[3][]
```

Le righe non sono memorizzate in posizioni adiacenti

- Possono essere spostate facilmente

```
// Scambio di due righe  
double[][] saldo = new double[5][6];  
...  
double[] temp = saldo[i];  
saldo[i] = saldo[j];  
saldo[j] = temp;
```

- L'array è una struttura dati efficiente ogni volta che il numero di elementi è noto
- Il ridimensionamento di un array in Java risulta poco efficiente
- Utilizzare altre strutture dati se il numero di elementi contenuto non è noto

Il pacchetto java.util contiene metodi statici di utilità per gli array

- Copia di un valore in tutti gli (o alcuni) elementi di un array
 - `Arrays.fill (<array>, <value>);`
 - `Arrays.fill (<array>, <from>, <to>, <value>);`
- Copia di array
 - `System.arraycopy (<arraySrc>, <offsetSrc>, <arrayDst>, <offsetDst>, <#elements>);`
- Confronta due array
 - `Arrays.equals (<array1>, <array2>);`
- Ordina un array (di oggetti che implementino l'interfaccia Comparable)
 - `Arrays.sort (<array>);`
- Ricerca binaria (o dicotomica)
 - `Arrays.binarySearch (<array>);`

Esempi di Array

Array Monodimensionali

```
int[] list = new int[10];  
  
list.length;  
  
int[] list = {1, 2, 3, 4};
```

Array Multidimensionali

```
int[][] list = new int[10][10];  
list.length;  
list[0].length;  
int[][] list = {{1, 2}, {3, 4}};
```

Array irregolari

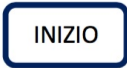



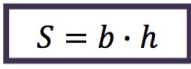
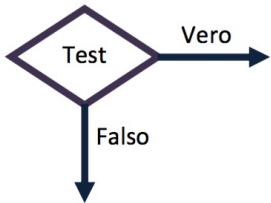


```
int[][] m = {  
    {1, 2, 3, 4},  
    {1, 2, 3},  
    {1, 2},  
    {1}  
};
```

[esempi ed esercizi su array](#)

4. Il controllo del flusso

Workflow - Flusso di esecuzione e gestione

rappresentazione grafica

Oggetto grafico	Denominazione	Significato
	Punto di partenza	Rappresenta un'azione che avvia il processo
	Punto di fine	Rappresenta un'azione che conclude il processo
	Leggi	Rappresenta un'azione di ingresso dati
	Scrivi	Rappresenta un'azione di uscita dei risultati
	Elaborazione	Rappresenta il comando o calcolo da eseguire
	Test	Rappresenta la scelta fra due possibili percorsi
	Linea di flusso	Indica la direzione del percorso del flusso
	Connessione	Rappresenta il punto d'inserimento nel grafico (generalmente contiene una lettera o un numero d'ordine)

Il controllo del flusso

Java mette a disposizione del programmatore diverse strutture sintattiche per consentire il **controllo del flusso**

Selezione, scelta condizionale

if statements

```
if (condition) {
    //statements;
}
```

else if

```
[optional]
else if (condition2) {

    //statements;

}
```

else

```
[optional]
else {

//statements;

}
```

switch Statements

```
switch (espressione) {

    case valore1:

        //statements;

        break;

    ...

    case valoren:

        //statements;

        break;

    default:

        //statements;

}
```

Cicli definiti

Se il numero di iterazioni è prevedibile dal contenuto delle variabili all'inizio del ciclo.

```
for (init; condition; adjustment) {

//statements;

}
```

Esempio: prima di entrare nel ciclo so già che verrà ripetuto 10 volte

```
int n=10;
for (int i=0; i<n; ++i) {
    ...
}
```

Cicli indefiniti

Se il numero di iterazioni non è noto all'inizio del ciclo.

```
while (condition) {

//statements;

}
```

```
do {

//statements;

} while (condition);
```

Esempio: il numero di iterazioni dipende dai valori immessi dall'utente.

```
while(true) {
    x = Integer.parseInt(JOptionPane.showInputDialog("Immetti numero positivo"));
    if (x > 0) break;
}
```

Cicli annidati

Se un ciclo appare nel corpo di un altro ciclo.

Esempio: stampa quadrato di asterischi di lato n

```
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) System.out.print("*");
    System.out.println();
}
```

Cicli con filtro

Vengono passati in rassegna un insieme di valori e per ognuno di essi viene fatto un test per verificare se il valore ha o meno una certa proprietà in base alla quale decideremo se prenderlo in considerazione o meno.

Esempio: stampa tutti i numeri pari fino a 100


```
for (int i=1; i<100; ++i) { // passa in rassegna tutti i numeri fra 1 e 100
    if (i % 2 == 0) // filtra quelli pari
        System.out.println(i);
}
```

Cicli con filtro e interruzione

Se il ciclo viene interrotto dopo aver filtrato un valore con una data proprietà.

Esempio: verifica se un array contiene o meno numeri negativi

```
boolean trovato = false;
for (int i=0; i<v.length; ++i) // passa in rassegna tutti gli indici dell'array v
    if (v[i]<0) { // filtra le celle che contengono valori negativi
        trovato = true;
        break; // interrompe ciclo
    }
// qui trovato vale true se e solo se vi sono numeri negativi in v
```

Cicli con accumulatore

Vengono passati in rassegna un insieme di valori e ne viene tenuta una traccia cumulativa usando una opportuna variabile.

Esempio: somma i primi 100 numeri interi.

```
int somma = 0; // variabile accumulatore di tipo int
for (int i=1; i<100; ++i) { // passa in rassegna tutti i numeri fra 1 e 100
    somma = somma + i; // accumula i valori nella variabile accumulatore
}
```

Esempio: data una stringa s, ottieni la stringa rovesciata

```
String rovesciata = ""; // variabile accumulatore di tipo String
for (int i=0; i<s.length(); ++i) { // passa in rassegna tutti gli indici dei caratteri di s
    rovesciata = s.substring(i, i+1) + rovesciata; // accumula i caratteri in testa all'accumulatore
}
```

Cicli misti

Esempio di ciclo definito con filtro e accumulatore: calcola la somma dei soli valori positivi di un array

```
int somma = 0;
for (int i=0; i<v.length; ++i) // passa in rassegna tutti gli indici dell'array v
    if (v[i]>0) // filtra le celle che contengono valori positivi
        somma = somma + v[i]; // accumula valore nella variabile accumulatore
```

5. I metodi

metodo

- Termine caratteristico dei linguaggi OOP
- Un **insieme di istruzioni con un nome**
- Uno strumento per risolvere gradualmente i problemi scomponendoli in **sottoproblemi**
- Uno strumento per **strutturare** il codice
- Uno strumento per **ri-utilizzare** il lavoro già svolto
- Uno strumento per rendere il **programma più chiaro** e leggibile

quando e perché usare i metodi

1. Quando il programma da realizzare è articolato diventa conveniente identificare **sottoproblemi** che possono essere risolti individualmente
2. scrivere **sottoprogrammi** che risolvono i sottoproblemi richiamare i **sottoprogrammi** dal programma principale (main)
3. Questo approccio prende il nome di **programmazione procedurale** (o astrazione funzionale)
4. In Java i **sottoprogrammi** si realizzano tramite **metodi ausiliari**
5. Sinonimi usati in altri linguaggi di programmazione: **funzioni**, **procedure** e (sub)**routines**

Metodi ausiliari (static)

- **metodi statici**: dichiarati **static**
- richiamabili attraverso nome della classe
- p.es: **Math.sqrt()**

```
public class ProvaMetodi
{
    public static void main(String[] args) {
        stampaUno();
        stampaUno();
        stampaDue();
    }

    public static void stampaUno() {
        System.out.println("Hello World");
    }

    public static void stampaDue() {
        stampaUno();
        stampaUno();
    }
}
```

Metodi non static

- I metodi **non static** rappresentano operazioni effettuabili su singoli oggetti
- La documentazione indica per ogni metodo il tipo ritornato e la lista degli argomenti formali che rappresentano i dati che il metodo deve ricevere in ingresso da chi lo invoca
- Per ogni argomento formale sono specificati:
 - un tipo (primitivo o reference)
 - un nome (identificatore che segue le regole di naming)

Invocazione di metodi non static

- L'invocazione di un metodo non static su un oggetto istanza della classe in cui il metodo è definito si effettua con la sintassi:
- Ogni volta che si invoca un metodo si deve specificare una lista di argomenti attuali
- Gli argomenti attuali e formali sono in corrispondenza posizionale
- Gli argomenti attuali possono essere delle variabili o delle espressioni
- Gli argomenti attuali devono rispettare il tipo attribuito agli argomenti formali
- La documentazione di ogni classe (istanziabile o no) contiene l'elenco dei metodi disponibili
- La classe **Math** non è istanziabile
- La classe **String** è "istanziabile ibrida"
- La classe **StringBuilder** è "istanziabile pura"

Metodi predicativi

Un metodo che restituisce un tipo primitivo **boolean** si definisce **predicativo** e può essere utilizzato direttamente in una condizione. In inglese sono spesso introdotti da **is** oppure **has**: `isMale()`, `hasNext()`.

Esempi sui metodi

6. Classi e oggetti

I principi della OOP

- Definire nuovi tipi di dati.
- Incapsulare i valori e le operazioni.
- Riusare il codice esistente.
- Supportare il polimorfismo.

L'oggetto

- un oggetto è una istanza di una classe.
- un oggetto deve essere conforme alla descrizione di una classe.
- un oggetto è contraddistinto da:
 1. attributi;
 2. metodi;
 3. identità;
- un oggetto non deve mai manipolare direttamente i dati di un altro oggetto
- la classe è una entità statica cioè a tempo di compilazione;
- l'oggetto è una entità dinamica cioè a tempo di esecuzione (run time);

ADT (Abstract Data Types): creare nuovi tipi di oggetto

- i dati (o attributi)
 - contengono le informazioni di un oggetto;
- le operazioni (o metodi)
 - consentono di leggere/scrivere gli attributi di un oggetto;

La descrizione di una classe

- La classe rappresenta la descrizione di un oggetto, contiene le definizioni di proprietà e metodi dell'oggetto che rappresenta.
- La classe consente di implementare gli ADT attraverso il meccanismo di incapsulamento.
- i dati devono rimanere privati insieme all'implementazione
- solo l'interfaccia delle operazioni è resa pubblica all'esterno della classe.

Relazioni fra le classi

- uso: una classe può usare oggetti di un'altra classe;
- aggregazione: una classe può avere oggetti di un'altra classe;
- ereditarietà: una classe può estendere un'altra classe.

Uso

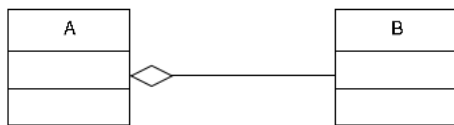
- L'uso o associazione è la relazione più semplice che intercorre fra due classi.

- Per definizione diciamo che una classe A usa una classe B se:
 - un metodo della classe A invia messaggi agli oggetti della classe B , oppure
 - un metodo della classe A crea, restituisce, riceve oggetti della classe B .



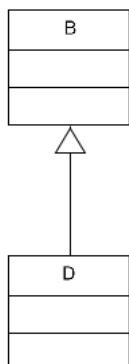
Aggregazione

- una classe A aggrega oggetti di una classe B quando la classe A contiene oggetti della classe B
- è un caso speciale della relazione di uso
- relazione has-a (ha-un)



Ereditarietà

- riuso del codice
- classe derivata o sottoclasse
- classe base o superclasse
- relazione is-a (è-un)



si ottiene il riuso del codice

Consideriamo la classe base B che ha un metodo f(...) e la classe derivata D che eredita da B. La classe D può usare il metodo f(...) in tre modi:

- lo eredita: quindi f(...) può essere usato come se fosse un metodo di D ;
- lo riscrive (override): cioè si dà un nuovo significato al metodo riscrivendo la sua implementazione nella classe derivata, in modo che tale metodo esegua una azione diversa;
- lo estende: cioè richiama il metodo f(...) della classe base ed aggiunge altre operazioni.

Quando usare l'ereditarietà

- Usare l'ereditarietà solo quando il legame fra la classe base e la classe derivata è per sempre, cioè dura per tutta la vita degli oggetti, istanze della classe derivata.
- Se tale legame non è duraturo è meglio usare l'aggregazione al posto della specializzazione.

Polimorfismo

- La parola polimorfismo deriva dal greco e significa letteralmente molte forme.
- Nella OOP tale termine si riferisce ai metodi: per definizione, il polimorfismo è la capacità di un oggetto, la cui classe fa parte di una gerarchia, di chiamare la versione corretta di un metodo.
- Quindi il polimorfismo è necessario quando si ha una gerarchia di classi.

Classi Java

Java è un linguaggio orientato agli oggetti

- In Java quasi tutto è un oggetto
- Come definire classi e oggetti in Java?
- **Classe**: codice che definisce un tipo concreto di oggetto, con proprietà e comportamenti in un unico file
- **Oggetto**: istanza, esemplare della classe, entità che dispone di alcune proprietà e comportamenti propri, come gli oggetti della realtà
- In **Java** quasi tutto è un **oggetto**, ci sono solo due **eccezioni**: i tipi di dato semplici (tipi primitivi) e gli array (un oggetto trattato in modo *particolare*)
- Le classi, in quanto tipi di dato strutturati, prevedono **usi e regole più complessi** rispetto ai tipi semplici

Le classi estendono il concetto di "struttura" di altri linguaggi

Definiscono

- I dati (detti campi o attributi)
- Le azioni (metodi, comportamenti) che agiscono sui dati

Possono essere definite

- Dal programmatore (p.es. Automobile, Topo, Studente, ...)
- Dall'ambiente Java (p.es. String, System, Scanner, ...)

La "gestione" di una classe avviene mediante

- Definizione della classe
- Istanziamento di Oggetti della classe

Struttura di una classe

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
  
}
```

Le classi in Java

- Il primo passo per definire una classe in Java è creare un file che deve chiamarsi esattamente come la classe e con estensione .java
- Java permette di definire solo una classe per ogni file
- Una classe in Java è formata da:
 - **Attributi:** (o campi/proprietà) che immagazzinano alcune informazioni sull'oggetto. Definiscono lo stato dell'oggetto
 - **Costruttore:** metodo che si utilizza per inizializzare un oggetto
 - **Metodi:** sono utilizzati per modificare o consultare lo stato di un oggetto. Sono equivalenti alle funzioni o procedure di altri linguaggi di programmazione

Classi e documentazione

- Come la maggior parte dei linguaggi di programmazione, Java è dotato di una libreria di classi "pronte all'uso" che coprono molte esigenze
- Usare classi già definite da altri è la norma per non sprecare tempo a risolvere problemi già risolti o a reinventare la ruota (DRY)
- La libreria Java standard è accompagnata da documentazione che illustra lo scopo e l'utilizzo di ciascuna classe presente,
- Dalla versione 9 di Java la libreria è stata divisa in moduli
- [Documentazione Java 8](#)
- [Documentazione Java 9](#)
- [Documentazione Java 11](#)
- [Documentazione Java 13](#)

Instanziare una Classe: gli oggetti

Gli oggetti sono caratterizzati da

- Classe di appartenenza - tipo (ne descrive attributi e metodi)
- Stato (valore attuale degli attributi)
- Identificatore univoco (reference - handle - puntatore)

Per creare un oggetto occorre

- Dichiarare una istanza
- La dichiarazione non alloca spazio ma solo un riferimento (puntatore) che per default vale null
- Allocazione e inizializzazione
- Riservano lo spazio necessario creando effettivamente l'oggetto appartenente a quella classe

Notazioni Puntate

Le notazioni puntate possono essere combinate

- `System.out.println("Hello world!");`
- **System** è una classe del package `java.lang`
- **out** è una variabile di classe contenente il riferimento ad un oggetto della classe **PrintStream** che punta allo standard output
- **println()** è un **metodo** della classe `PrintStream` che stampa una linea di testo

Operazioni su reference

Uso degli operatori relazionali `==` e `!=`

- Attenzione: il test di uguaglianza viene fatto sul puntatore (**reference**) e NON sull'oggetto
- Stabiliscono se i **reference** si riferiscono allo stesso oggetto
- È definita l'**assegnazione**
- È definito l'**operatore punto** (notazione puntata)
- **NON** è prevista l'aritmetica dei puntatori

Operazioni su istanze

- Le principali operazioni che si possono effettuare sulle variabili che riferiscono istanze di una classe sono:
 - assegnamento
 - confronto
 - invocazione di metodi
- Il valore di una variabile di tipo strutturato è il riferimento ad un oggetto (istanza di una classe)
- Una stessa variabile può riferire oggetti diversi in momenti diversi a seguito di operazioni di assegnazione sul suo valore
- Se la variabile contiene il valore null non riferisce nessun oggetto in quel momento

Accesso a metodi e attributi non static

- La sintassi è simile al caso precedente, ma ovviamente l'accesso/invocazione è possibile solo tramite un'istanza specifica (ed ogni accesso è diversificato):
- Nel corpo di un metodo non **static** si può accedere a qualunque attributo e metodo della stessa classe
- All'interno del corpo di un metodo si possono riferire in modo abbreviato attributi e metodi definiti nella stessa classe
- Se nel corpo di un metodo non **static** appare il nome di un metodo o attributo non **static** della sua classe è sottinteso che sia riferito all'istanza su cui è stato invocato il metodo

Oggetti e riferimenti

- Le variabili hanno un nome, gli oggetti no
- Per utilizzare un oggetto bisogna passare attraverso una variabile che ne contiene il riferimento
- Uno stesso oggetto può essere riferito da più variabili e quindi essere raggiunto tramite nomi diversi (di variabili)
- Il rapporto variabili - oggetti riferiti è dinamico, il riferimento iniziale non necessariamente rimane legato all'oggetto per tutta la sua esistenza
- Se un oggetto non è (più) riferito da nessuna variabile diventa irraggiungibile (e quindi interviene il garbage collector)

Confronti tra variabili di tipo strutturato

- E' possibile applicare gli operatori di confronto == e != a variabili di tipo strutturato
- Se uno dei due termini del confronto è il valore null si verifica se una certa variabile riferisce un oggetto oppure no, p.e. `saluto3 != null`
- Se entrambi i termini del confronto sono variabili, si verifica se hanno lo stesso valore (cioè riferiscono esattamente lo stesso oggetto)

Confronto tra riferimenti vs. confronto tra oggetti

- Usare == fa il confronto tra i riferimenti non fra i valori contenuti negli oggetti (p.e. le sequenze di caratteri contenute nelle istanze di String)
- Di solito si vogliono confrontare i contenuti non i riferimenti: per questo si usa il metodo **equals**
- Il metodo booleano equals della classe String accetta come argomento il riferimento ad un altro oggetto e ritorna true se le stringhe contenute sono uguali (in modo case sensitive), false altrimenti
- Il metodo booleano equalsIgnoreCase fa lo stesso senza distinguere maiuscole/minuscole

Instanziare una Classe: gli oggetti

Gli oggetti sono caratterizzati da

- Classe di appartenenza - tipo (ne descrive attributi e metodi)
- Stato (valore attuale degli attributi)
- Identificatore univoco (reference - handle - puntatore)

Per creare un oggetto occorre

- Dichiarare una istanza

- La dichiarazione non alloca spazio ma solo un riferimento (puntatore) che per default vale null
- Allocazione e inizializzazione
- Riservano lo spazio necessario creando effettivamente l'oggetto appartenente a quella classe

Notazioni Puntate

Le notazioni puntate possono essere combinate

- `System.out.println("Hello world!");`
- **System** è una classe del package `java.lang`
- **out** è una variabile di classe contenente il riferimento ad un oggetto della classe **PrintStream** che punta allo standard output
- **println()** è un **metodo** della classe `PrintStream` che stampa una linea di testo

Operazioni su reference

Uso degli operatori relazionali `==` e `!=`

- Attenzione: il test di uguaglianza viene fatto sul puntatore (**reference**) e NON sull'oggetto
- Stabiliscono se i **reference** si riferiscono allo stesso oggetto
- È definita l'**assegnazione**
- È definito l'**operatore punto** (notazione puntata)
- **NON** è prevista l'aritmetica dei puntatori

Operazioni su istanze

- Le principali operazioni che si possono effettuare sulle variabili che riferiscono istanze di una classe sono:
 - assegnamento
 - confronto
 - invocazione di metodi
- Il valore di una variabile di tipo strutturato è il riferimento ad un oggetto (istanza di una classe)
- Una stessa variabile può riferire oggetti diversi in momenti diversi a seguito di operazioni di assegnazione sul suo valore
- Se la variabile contiene il valore null non riferisce nessun oggetto in quel momento

Accesso a metodi e attributi non static

- La sintassi è simile al caso precedente, ma ovviamente l'accesso/invocazione è possibile solo tramite un'istanza specifica (ed ogni accesso è diversificato):
- Nel corpo di un metodo non **static** si può accedere a qualunque attributo e metodo della stessa classe
- All'interno del corpo di un metodo si possono riferire in modo abbreviato attributi e metodi definiti nella stessa classe
- Se nel corpo di un metodo non **static** appare il nome di un metodo o attributo non **static** della sua classe è sottinteso che sia riferito all'istanza su cui è stato invocato il metodo

Oggetti e riferimenti

- Le variabili hanno un nome, gli oggetti no
- Per utilizzare un oggetto bisogna passare attraverso una variabile che ne contiene il riferimento
- Uno stesso oggetto può essere riferito da più variabili e quindi essere raggiunto tramite nomi diversi (di variabili)
- Il rapporto variabili - oggetti riferiti è dinamico, il riferimento iniziale non necessariamente rimane legato all'oggetto per tutta la sua esistenza
- Se un oggetto non è (più) riferito da nessuna variabile diventa irraggiungibile (e quindi interviene il garbage collector)

Confronti tra variabili di tipo strutturato

- E' possibile applicare gli operatori di confronto == e != a variabili di tipo strutturato
- Se uno dei due termini del confronto è il valore null si verifica se una certa variabile riferisce un oggetto oppure no, p.e. `saluto3 != null`
- Se entrambi i termini del confronto sono variabili, si verifica se hanno lo stesso valore (cioè riferiscono esattamente lo stesso oggetto)

Confronto tra riferimenti vs. confronto tra oggetti

- Usare == fa il confronto tra i riferimenti non fra i valori contenuti negli oggetti (p.e. le sequenze di caratteri contenute nelle istanze di String)
- Di solito si vogliono confrontare i contenuti non i riferimenti: per questo si usa il metodo **equals**
- Il metodo booleano equals della classe String accetta come argomento il riferimento ad un altro oggetto e ritorna true se le stringhe contenute sono uguali (in modo case sensitive), false altrimenti
- Il metodo booleano equalsIgnoreCase fa lo stesso senza distinguere maiuscole/minuscole

Incapsulamento e visibilità in Java

- **due aspetti** che risultano fondamentali in un software:
 - **Interfaccia:** definita come gli **elementi che sono visibili dall'esterno**, cioè come il sw può essere utilizzato
 - **Implementazione:** la realizzazione pratica interna dei metodi e la loro interazione con le proprietà degli oggetti

Incapsulamento

- L'incapsulamento consiste nell'**occultamento degli attributi** di un oggetto in modo che possano essere **manipolati solo attraverso metodi** appositamente implementati. p.es la proprietà **saldo** di un oggetto **conto corrente**
- Bisogna fare in modo che l'interfaccia sia più indipendente possibile dall'implementazione
- In Java l'incapsulamento è strettamente relazionato con la visibilità

Visibilità

- Per indicare la visibilità di un elemento (attributo o metodo) possiamo farlo precedere da una delle seguenti parole riservate
- **public**: accessibile da qualsiasi classe
- **private**: accessibile solo dalla classe attuale
- **protected**: solo dalla classe attuale, le discendenti e le classi del nostro package
- **package**: se **non indichiamo la visibilità**: sono accessibili **solo dalle classi del nostro package**

Accesso agli attributi della classe

- Gli attributi di una classe sono strettamente relazionati con la sua implementazione.
- Conviene contrassegnarli come **private** e impedirne l'accesso dall'esterno
- In futuro potremo cambiare la rappresentazione interna dell'oggetto senza alterare l'interfaccia
- per consultarli e modificarli aggiungiamo i metodi accessori e mutatori: **getters** e **setters**

Modifica di rappresentazione interna di una classe

- Uno dei maggiori vantaggi di occultare gli attributi è che in **futuro potremo cambiarli** senza la necessità di cambiare l'interfaccia
- Un linguaggio di programmazione **ORIENTATO AGLI OGGETTI** fornisce meccanismi per definire nuovi tipi di dato basati sul concetto di classe
- Una classe definisce un insieme di oggetti (conti bancari, dipendenti, automobili, rettangoli, ecc...).
- Un oggetto è una struttura dotata di proprie **variabili** (che rappresentano il suo stato) propri **metodi** (che realizzano le sue funzionalità)

Metodi **getter** e **setter**

- Danno accesso agli attributi:

```
class NomeClasse{  
  
    private double nome1, nome2;  
  
    public double getNome1(){  
        return nome1;  
    }  
    public void setNome1(double nome1){  
        this.nome1 = nome1;  
    }  
}
```

- Vantaggi:
- possiamo cambiare la rappresentazione interna,
- verificare che i valori siano corretti,
- modificare altri aspetti dell'oggetto

Visibilità ed encapsulation

- Modularità = diminuire le interazioni
- Information Hiding = delegare responsabilità
- private attributo/metodo visibile solo da istanze della stessa classe
- public attributo/metodo visibile ovunque

Getters e setters

Metodi accessori e mutatori per leggere scrivere un attributo privato

```
String getColore() {  
    return colore;  
}  
void setColore(String nuovoColore) {  
    this.colore = nuovoColore;  
}
```

Proprietà visibili

```
class Automobile {  
    public String colore;  
}  
//creazione oggetto  
Automobile a = new Automobile();  
a.colore = "bianco"; // ok
```

Classe incapsulata

```
class Automobile {  
    private String colore;  
    public void vernicia(String colore) {  
        this.colore = colore;  
    }  
}  
  
//creazione oggetto  
Automobile a = new Automobile();  
a.colore = "bianco"; // error  
a.vernicia("verde"); // ok
```

7. OOP progettazione avanzata

Ereditarietà in Java

Riutilizzare il codice

- Uno dei grandi vantaggi della programmazione a oggetti è la facilità nel riutilizzare il codice
- In Java si realizza attraverso **l'ereditarietà**

- Per esempio immaginiamo di programmare un software per veicoli

```
Automobile {attributi: marca, modello}
```

- Dobbiamo modificare il sistema per aggiungere un nuovo attributo, le ruote motrici

a) Modifichiamo direttamente la classe Automobile: Errore!

b) Creiamo una nuova classe che erediti da Automobile:

```
Fuoristrada {Automobile + attributo: 4 ruote motrici}
```

Quando utilizzare l'ereditarietà

- Per modificare classi già esistenti, specialmente quando abbiamo molte classi da gestire
- Per non dover riprogrammare due volte lo stesso codice

Polimorfismo (overloading)

Una classe può avere più metodi con lo stesso nome

I metodi devono essere distinguibili in base a

- Numero dei parametri
- Tipo dei parametri

Il metodo da eseguire viene scelto in base a

- Numero e tipo di parametri

Il metodo da eseguire **NON** viene scelto in base al valore di ritorno

Il modificatore static

- La parola riservata **static** viene usata per indicare il livello di definizione di un **attributo o metodo**
- Se la parola **static** è presente l'attributo o metodo è definito **a livello di classe**
- Se la parola **static non** è presente l'attributo o metodo è definito **a livello di istanza** (esemplare della classe)

Classi Astratte

Una classe astratta è una classe avente **almeno un metodo astratto**

NB: Posso comunque dichiarare una classe **astratta anche se non contiene** alcun metodo astratto!

```
abstract class Forma {  
    ...  
    abstract void stampa();  
    ...  
}
```

Un metodo astratto è un metodo di cui non viene specificata l'implementazione

Una classe astratta

- È una classe non completamente definita
- Non può essere istanziata

Per ottenere una classe concreta (istanziabile) da una astratta occorre definire tutte le implementazioni mancanti ovvero

- Ereditare la classe facendo l'overriding di tutti i metodi astratti

Permette di trattare omogeneamente oggetti con caratteristiche diverse

Man mano che si sale nella gerarchia dell'ereditarietà, le classi diventano sempre più generiche e probabilmente più astratte.

Ad un certo punto la classe superiore diventa a tal punto generica che la si può pensare come una base per le altre classi piuttosto che come una classe di cui creare un oggetto.

METODO ASTRATTO

Voglio obbligare tutte le sottoclassi di una classe A ad avere un metodo `nomeMetodo()` ma allo stesso tempo non voglio implementare tale metodo nella classe A.

SINTASSI:

```
public abstract int nomeMetodo() ;
```

- un metodo astratto deve essere obbligatoriamente pubblico, altrimenti non avrebbe senso
- lascio il metodo indefinito. infatti non apro il relativo blocco con le parentesi graffe ed il codice del metodo.

Le Classi Astratte, oltre ad avere Metodi Astratti, possono avere metodi ed attributi *normali*.

Se una classe è astratta, non posso istanziarla.

NON POSSO CREARE OGGETTI DI UNA CLASSE ASTRATTA

Che senso ha definire dei metodi astratti e di conseguenza rendere una classe astratta?

Definisco un metodo astratto quando voglio forzare tutte le sottoclassi ad avere un determinato metodo.

I metodi astratti funzionano come segnaposto dei metodi implementati poi nella sottoclasse.

le sottoclassi

La sottoclasse deve implementare tutti i metodi che la superclasse aveva astratti.

Se la sottoclasse a sua volta non definisce i metodi che la superclasse aveva dichiarato astratti, anche la sottoclasse deve essere dichiarata astratta.

Static vs. non static

- Ogni attributo o metodo non static esiste concretamente **in ogni istanza creata**
 - Esiste in “molteplici versioni” se vengono create più istanze
 - **Non esiste** concretamente **se il programma non crea almeno un’istanza** (tramite operatore new)
-
- Ogni attributo o metodo static esiste concretamente a livello di classe
 - Esiste in un’unica versione
 - La sua esistenza non dipende da cosa fa il programma

Variabili di classe

- Rappresentano proprietà comuni a tutte le istanze
- Esistono anche in assenza di istanze (oggetti)
- Dichiarazione: static
- Accesso: NomeClasse.attributo

```
class Automobile {  
    static int numeroRuote = 4;  
}  
  
Automobile.numeroRuote;
```

Metodi di classe

Funzioni non associate ad alcuna istanza

- Dichiarazione: static
- Accesso: nome-classe . metodo()

```
class HelloWorld {  
public static void main (String args[]) {  
System.out.println("Hello World!");  
  
//p.es  cos(x): metodo static della classe Math, ritorna un double  
double y = Math.cos(x);  
}  
}
```

Accesso a metodi e attributi static

- Essendo definiti a livello di classe, attributi e metodi static sono acceduti/invocati tramite il nome della classe:
- `Math.sqrt (2);`
- `String.valueOf (Math .PI);`
- In caso di classe istanziabile ibrida si può accedere tramite una qualunque istanza della classe, ma è meglio non farlo.
- Nel corpo di un metodo `static` non si può accedere ad attributi e metodi non `static` della stessa classe

- Il metodo `static` deve poter essere invocato a livello di classe (anche in assenza di istanze) mentre attributi e metodi non `static` esistono solo se c'è almeno un'istanza
- Se nel corpo di un metodo appare il nome di un metodo o attributo `static` della sua classe è sottinteso che sia preceduto dal nome della classe stessa
- Eventuali mescolanze improprie di `static` e non `static` causano errori di compilazione

Interfacce

Un'interfaccia è una specie di classe completamente astratta, cioè del tutto priva della parte di implementazione (quasi... vedi sotto)

- Tutti i metodi sono astratti
- Non vi sono attributi
- È possibile definire solo "attributi" final (in pratica costanti)
- Definendo un attributo in un'interfaccia questo viene automaticamente considerato final

Un'interfaccia

- Ha tutti i vantaggi e le indicazioni d'uso delle classi astratte
- Presenta maggior flessibilità rispetto all'ereditarietà di una classe astratta

Si può pensare a un'interfaccia come a una classe astratta che ha tutti e soli metodi astratti (ci sono però differenze).

Un'interfaccia può essere considerata un modo per cosa dovrebbero fare le classi senza specificare come farlo.

Quindi un'interfaccia non è una classe ma un insieme di requisiti per le classi che si vogliono conformare ad essa.

Sintassi:

```
public interface NomeInterfaccia
{
    int metodo1( ... );
}
```

interface

Si utilizza la parola chiave interface anzichè class

I metodi sono implicitamente pubblici e astratti, non bisogna indicarlo

Se una classe decide di soddisfare i requisiti di un'interfaccia si dice che la classe implementa l'interfaccia.

esempio d'uso

Per indicare che una classe implementa un'interfaccia si utilizza la seguente sintassi:

```
public class NomeClasse implements NomeInterfaccia
{
    //codice relativo alla classe
}
```

Tale classe deve implementare tutti i metodi elencati nell'interfaccia.

proprietà

Le interfacce non sono classi; non si può utilizzare new per crearne oggetti.

I metodi di un'interfaccia sono automaticamente public (quindi non è necessario scriverlo)

Gli attributi di un'interfaccia sono sempre public static final (non è necessario scriverlo).

Un'interfaccia con il nome NomeInterfaccia va salvata nel file NomeInterfaccia.java (come accade per le classi).

ereditarietà multipla

Una sottoclasse può estendere solo 1 superclasse (non permettendo l'ereditarietà multipla)

Con le interfacce invece la situazione è diversa: una classe può implementare quante interfacce vuole.

variabili

Posso dichiarare variabili del tipo dell'interfaccia e, sfruttando il polimorfismo, assegnargli oggetti di classi che implementano tali interfacce.

esempio d'uso

```
public interface NomeInterfaccia {  
    int metodo1( );  
}  
  
public class NomeClasse implements NomeInterfaccia{  
    int metodo1( ) {  
        //codice del metodo  
    }  
}  
  
NomeInterfaccia a = new NomeClasse ( );
```

Tipi di interfacce

- Normali
- Single Abstract method - @FunctionalInterface
- Marker

Con java 1.8 le interfacce sono state modificate: è possibile implementare due tipi di metodi (!!!)

- default
- static

Lambda expressions

Con le interfacce contenenti un singolo metodo astratto, è possibile utilizzare le espressioni lambda

```
Integer raddoppiato = (o) -> o * 2 ;
```